Domain Analysis of Formal Model Checking Tools

Fadi Wedyan, Reema Freihat
Department of Software Engineering
Hashemite University
Zarka 13133, Jordan
Email: fadi.wedyan@hu.edu.jo
Email: reema.freihat@gmail.com

Suzan Wedyan
Department of Computer Science
Amman Arab University
Amman, Jordan
Email: susanwedyan@gamil.com

Hani Bani-Salameh, Hala Yousef
Department of Software Engineering
Hashemite University
Zarka 13133, Jordan
Email: hani@hu.edu.jo
Email: Hala_alboriny89@yahoo.com

***Abstract*—Finding faults in early stages of software development decreases the cost of testing and increases system reliability. For safety-critical software, it is vital to perform throughly verification procedures. Model-Checking can find many faults in an early stages, and therefore, it represent an important verification approach. In this paper, we use domain engineering to capture important features of model-checking tools. We used FODA method (Feature Oriented Domain Analysis) to describe the common features of important model-checking tools. We expect our results to help software engineers to identify tools that better fit their needs. Our results can also help in improving these tools.**

***Index Terms*—Formal method; Model checker; domain engineering; FODA.**

I. Introduction

Increasing software reliability and fault-tolerance requires performing many validation and verification activities. Finding faults in an early stages not only increases reliability, but also minimize the cost of finding faults [1]. By using formal models and model-based testing approaches, we can detect faults at the design stage. Many critical systems such as transportation systems, airplanes systems, and medical systems utilize model checking tools to avoid life-threatening software failures [2].

Formal methods can be defined as mathematical modeling techniques than can be applied for the development of computer software [3]. Formal methods support the specification, design, and verification phases of software development. Several tools are available for performing formal verification activities. Clarke et al. [3] classified tools used in formal verification into three types, these are: (1) equivalence checkers, (2) model checkers, and (3) theorem provers. Equivalence checker compares two models for equivalence by applying different heuristics. A model checker checks if the system verifies a model against specified properties. Theorem prover uses mathematical techniques to proof a model and design [4].

In this paper, we analyze model checker tools using domain engineering. We applied a well-known methodology in domain analysis called FODA (Feature Oriented Domain Analysis). Using FODA, a feature model of analyzed tools is built by extracting and analyzing important features [5]. We evaluated the following model checkers: BLAST [6], UPPAAL [7], SPIN [8], CBMC [9], and MRMC [10]. These are widely used tools that deal different types of development environments. We extracted the main features of each tool and specified each tool limitations.

The rest of the paper is organized as follows. Section II summarizes related work. Section III describes the model checker tools evaluated in this paper. Section IV describes how we applied domain engineering, FODA, and domain analysis to evaluate the tools. Finally, conclusions and future work are given in Section V.

II. Related Work

Many researchers had evaluated model checkers when applied in specified domains (i.e., for a specific type of software). In this work, we evaluated model checkers depending on their important features, which helps in understanding the evaluated tools across domains.

Kim et al. [11] evaluated model checkers as unit testing tools. They performed an experiment applying BLAST and CBMC to test the components of a storage platform software for flash memory. Post et al. [12] presented an approach combining abstract interpretation and source code bounded model checking, where the model checker is used to reduce the number of false error reports. The approach was applied to source code from the automotive industry written in C.

Schlich and Kowalewski [13] studied the applicability of existing model checking approaches for C code to embedded systems. They performed a case study in which CBMC as one representative model checker is applied to a specific piece of micro-controller code. Mhlberg and Lttgen [14] evaluated the utility of software model checker BLAST for revealing errors in Linux kernel code. Similarly, Post and Kuchlin [15] evaluated Linux device drivers, combining several tools and techniques into one integrated tool-chain. They extended Microsofts Static Driver Verifier (SDV) project, and combine its implementation with the public domain bounded model checker CBMC as a new verification back-end.

Yang et al. [16] show how to use model checking to find errors in file systems. They built a system, called FiSC, for model checking file systems and applied it to three widely-used, heavily-tested file systems. Cadar et al. [17] presented a tool called EXE tool that automatically generates inputs that crash real code. The tool was applied to network filters. Kolb et al. [18] applied BLAST on an industrial strength C implementation of a protocol stack. Ku et al. [19] presented

978-1-5386-1949-0/17/$31.00

a publicly-available benchmark suite for evaluation of model checkers. They evaluated the benchmark using the SatAbs model checker. Choi [20] presented a solution to model-checking automotive operating systems for the purpose of safety analysis. The approach was evaluated on the Trampoline kernel code using the model checker SPIN. Similarly, Choi et al. [21] presented a collaborative approach using model checking and testing for the efficient safety checking of an automotive operating system. The approach was implemented as a prototype tool and used to the verification of an open source automotive operating system based on the OSEK/VDX international standard.

III. Model Checker Tools: Definition

A model checker is a tool designed to verify system correctness [4]. A model checker can be viewed as an algorithmic technique to verify a system description against a specification [4]. In this section, we describe the 5 model checker tools we evaluated in this paper.

A. BLAST [6]

The Berkeley Lazy Abstraction Software Verification Tool (BLAST) is a software model checking tool for C programs which checks whether the software under test satisfies the behavioral requirements of its associated properties [22]. Blast explores and might refine abstractions of the program state space based on lazy predicate abstraction and interpolation-based predicate discovery. [23]. The term "lazy absrtaction" describes the methodology of Blast in integrating the three steps of abstraction, verification, and refinement [24].

Blast builds the abstraction model on the fly using predicate abstraction. The model is then checked for reachability, if there is no abstract path to the specified error label, Blast reports that the system is safe [22].

B. UPPAAL [7]

UPPAAL is a tool for the verification of real-time systems. The tool is developed jointly by Uppsala University and Aalborg University. UPPAAL was applied in case studies ranging from communication protocols to multimedia applications. The tool is designed to verify systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization [25].

UPPAAL is based on the theory of timed automata. The modeling language of the tool offers features as bounded integer variables and urgency. The query language of Uppaal, used to specify properties to be checked, is a subset of computation tree logic (CTL) [25]. The query language describes communicating abstract state machines with handshake synchronization and communication via shared variables. It also provides a C-like syntax for describing guards and updates on transitions [7].

Uppaal uses a client-server architecture, where the client side contains the GUI while the model checking engine is contained on the server. The GUI is implemented in Java and the engine, or server, is compiled for different platforms (Linux, Windows, Solaris). Using this design allows the tool to run on different machines [25], [26].

C. SPIN [8]

SPIN is a generic verification system that supports the design and verification of asynchronous process systems [27], [28]. SPIN supports the verification of UML statecharts [29]. SPIN focuses on proving the correctness of process interactions, which are specified in SPIN by the means of: rendezvous primitives, asynchronous message passing through buffered channels, and through access to shared variables [27], [30], [31].

SPIN provides an intuitive, program-like notation for specifying design choices. No implementation details are necessary. It provides a concise notation for expressing general correctness requirements. It also provides a methodology for establishing the logical consistency of the design and matching correctness requirements [27].

D. CBMC [9]

The C Bounded Model Checker (CBMC) implements bit-precise bounded model checking for C programs. CBMC verifiers the absence of violated assertions under a given loop unwinding bound [32]. CBMC finds the violation of assertions in C programs. It also proves safety of assertions under a given bound.

Given a C program, annotated with assertions and with loops unrolled to a given depth, CBMC transaltes the program into a formula, if the formula is satisfiable, then an execution leading to a violated assertion exists [32]. CBMC can deal with programs that use dynamic memory allocation (e.g., dynamic arrays, lists), CBMC provides a GUI that allows users to interactively step through counterexample traces [33]. The GUI allows users to step through traces as they would in a debugger. Potential faulty parts of the code are highlighted for the user [33].

E. MRMC [10]

The Markov Reward Model Checker (MRMC) is a tool for verifying properties over probabilistic models. MRMC has the features of supporting computing time and reward-bounded reachability probabilities, (property-driven) bisimulation minimization, and on-the-y steady-state detection [34].

The recent version of the tool includes time-bounded reachability analysis for uniform CTMDPs and CSL model checking by discrete-event simulation [10].

IV. Experiment

Pressman [35] described the intent of domain engineering as "... to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain". Domain engineering consists of three activities, these are: (1) domain analysis, in which features about the applications studied are collected, (2) domain design, in which a model is built for the software under development, and (3) domain implementation,

978-1-5386-1949-0/17/$31.00

TABLE I
FEATURE SUPPORTED BY MODEL CHECKER.

Features	BLAST	UPPAAL	SPIN	CBMC	MRMC
Providing techniques for increasing efficiency					
Speed	✓	✓	✓	X	✓
Memory consumption	✓	✓	✓	X	✓
Verification properties type					
Safety-critical requirements	✓	X	✓	✓	✓
Network properties	X	✓	✓	X	X
Boundary checking	X	✓	X	✓	X
Multi-threading issues	X	X	✓	X	X
Logical design errors	X	X	✓	X	X
Reachability probabilities	X	X	X	X	✓
Steady-state detection	X	X	X	X	✓
Platform/Operating system					
Windows	✓	✓	✓	✓	✓
Mac	X	✓	✓	✓	✓
Unix	X	X	✓	X	X
Linux	✓	✓	✓	✓	✓
Type of Input					
C	✓	X	✓	✓	X
C++	X	X	X	✓	X
Java	X	X	X	✓	X
UML Models	X	✓	X	X	X
Promela	X	X	✓	X	X
Markov chains	X	X	X	X	✓

which includes building the software by re-using existing components [35].

In this paper, we are concerned with domain analysis. We collected common features for the model checker tools. We identified information used in developing model checker tool. The information is then organized with the purpose of making it reusable when creating a new tool.

The analysis is performed using FODA. Next, we describe FODA and how it was implement in our study.

A. Using FODA to analyze model checkers

FODA is a domain engineering approach emphasizing feature analysis. A feature is defined as user-visible characteristic of a software system [36]. FODA aims at identifying and analyzing the systems based on their supported features. According to FODA, the analysis should be based on the features provided by the software in a certain domain, rather than the implementation and design details [5], [37].

FODA involves the following activities: (1) context analysis: definition of the product domain (model checkers), (2) domain model development, and (3) domain design and implementation [5], [38].

B. Domain Analysis

Domain analysis is a process that identifies features and capabilities that are commonly found in software of a specific domain [35]. Domain analysis is systematic, and provides a common understanding of the domain. Context analysis is the first activity of FODA, in this work, the context is ”model checker tools”. We studied the features in model checkers and applied each of them on tools. We identified 19 features of five model checker tools (BLAST, UPAAL, SPIN, CBMC, and MRMC) which are widely used and well-known model checker tools.

We classified the extracted features into four groups, these are:

1) Providing techniques for increasing efficiency. We use memory consumption and response time as features that describe efficiency.
2) Verification properties type. Here, we group the type of property that a model checker can verify. This group can be used to check the functionalities that a tool provides.
3) Platform/Windows. Here we specify the platforms that each of the tool supports.
4) Type of inputs. The tools vary on the form of the inputs it uses for verification. While some tools use UML and other models as inputs, some tools require the existence of the source code of the verified software.

The summary of our analysis is given in Table I. As shown in the table, we identified 19 features. These finding can utilized by testers as a checklist of the features of a tool that they can verify.

V. CONCLUSIONS

In this paper, we used domain engineering for model checker tools to identify and analyze their common features. We used five well-known tools for evaluation. Our analysis show that most of the tools provide mechanisms for increasing their performance (i.e., efficiency). The tools vary in the type of checking performed. Among the evaluated tools, MRMC checks more properties than other tools. Most of the tools run on Windows, Linux, and MAC, while only SPIN can run on UNIX. Finally, the tools accept different inputs. BLAST, and CBMC accepts only the source code of the program.

978-1-5386-1949-0/17/$31.00 ©2017 IEEE

While UPPAAL and MRMC can verify models (UML models, Markov chains, respectively).

In the future, we intend to include more tools. We also intend to quantify the evaluation of the features in order to provide a better understanding of the tools strengths and limitations.

ACKNOWLEDGMENT

This work was supported in part by grant #2015/3 from Hashemite University to Fadi Wedyan.

REFERENCES

[1] T. Fatima, K. Saghar, and A. Ihsan, "Evaluation of model checkers spin and uppaal for testing wireless sensor network routing protocols," in *Applied Sciences and Technology (IBCAST), 2015 12th International Bhurban Conference on*. IEEE, 2015, pp. 263–267.

[2] P. Herber and B. Hünnemeyer, "Formal verification of systemc designs using the blast software model checker." in *ACESMB@ MoDELS*, 2014, pp. 44–53.

[3] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[4] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[5] H. Bani-Salameh, C. Jeffery, and M. Hammad, "Developers social networks–tools analysis based on the 3cs model," *International Journal of Networking and Virtual Organisations 17*, vol. 13, no. 2, pp. 159–175, 2013.

[6] University of California at San Diego, http://cseweb.ucsd.edu/~rjhala/blast.html, April, 2017.

[7] UPPAAL, http://www.uppaal.org/, April, 2017.

[8] SPIN, http://spinroot.com/spin/whatispin.html, April, 2017.

[9] CBMC, http://www.cprover.org/cbmc/, April, 2017.

[10] MRMC, http://www.mrmc-tool.org/trac/, April, 2017.

[11] M. Kim, Y. Kim, and H. Kim, "A comparative study of software model checkers as unit testing tools: an industrial case study," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 146–160, 2011.

[12] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 188–197.

[13] B. Schlich and S. Kowalewski, "Model checking c source code for embedded systems," *International journal on software tools for technology transfer*, vol. 11, no. 3, pp. 187–202, 2009.

[14] J. T. Mühlberg and G. Lüttgen, "Blasting linux code," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2006, pp. 211–226.

[15] H. Post and W. Küchlin, "Integrated static analysis for linux device driver verification," in *International Conference on Integrated Formal Methods*. Springer, 2007, pp. 518–537.

[16] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 393–423, 2006.

[17] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[18] E. Kolb, O. Šerỳ, and R. Weiss, "Applicability of the blast model checker: An industrial case study," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2009, pp. 218–229.

[19] K. Ku, T. E. Hart, M. Chechik, and D. Lie, "A buffer overflow benchmark for software model checkers," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 389–392.

[20] Y. Choi, "Model checking trampoline os: a case study on safety analysis for automotive software," *Software Testing, Verification and Reliability*, vol. 24, no. 1, pp. 38–60, 2014.

[21] Y. Choi, M. Park, T. Byun, and D. Kim, "Efficient safety checking for automotive operating systems using property-based slicing and constraint-based environment generation," *Science of Computer Programming*, vol. 103, pp. 51–70, 2015.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *International SPIN Workshop on Model Checking of Software*. Springer, 2003, pp. 235–239.

[23] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 58–70, 2002.

[25] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236.

[26] A. Ravn, J. Srba, and S. Vighio, "A formal analysis of the web services atomic transaction protocol with uppaal," *Leveraging Applications of Formal Methods, Verification, and Validation*, pp. 579–593, 2010.

[27] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[28] G. J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage preserving reduction strategies for reachability analysis," in *Proc. 12th IFIP WG*, vol. 6, 2016, pp. 349–363.

[29] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker," *Formal aspects of computing*, vol. 11, no. 6, pp. 637–664, 1999.

[30] R. P. Kurshan, *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton university press, 2014.

[31] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*. Springer, 1993, pp. 25–60.

[32] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[33] A. Groce, D. Kroening, and F. Lerda, "Understanding counterexamples with explain," in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 453–456.

[34] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The ins and outs of the probabilistic model checker mrmc," *Performance evaluation*, vol. 68, no. 2, pp. 90–104, 2011.

[35] R. S. Pressman, *Software engineering: a practitioner's approach*, 7th ed. Palgrave Macmillan, 2010.

[36] A. Van Deursen, P. Klint, J. Visser *et al.*, "Domain-specific languages: An annotated bibliography." *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[37] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," DTIC Document, Tech. Rep., 1990.

[38] L. S. De Oliveira and M. A. Gerosa, "Collaborative features in content sharing web 2.0 social networks: A domain engineering based on the 3c collaboration model," in *International Conference on Collaboration and Technology*. Springer, 2011, pp. 142–157.