Traceability and Model Checking to Support Safety Requirement Verification

Shuanglong Kan
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics, Nanjing, China
kanshuanglong@nuaa.edu.cn

ABSTRACT

Ensuring safety-critical software safety requires strict verification of the conformance between safety requirements and programs. Formal verification techniques, such as model checking and theorem proving, can be used to partially realize this objective. DO-178C, a standard for airborne systems, allows formal verification techniques to replace certain forms of testing. My research is concerned with applying model checking to verify the conformance between safety requirements and programs. First, a formal language for specifying software safety requirements which are relevant to event sequences is introduced. Second, the traceability information models between formalized safety requirements and programs are built. Third, the checking of a program against a safety requirement is decomposed into smaller model checking problems by utilizing traceability information model between them.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*

General Terms

Verification,Languages,Reliability

Keywords

Safety-critical software, safety requirements, model checking, traceability, event automata

1. RESEARCH PROBLEM

Safety-critical software is typically subject to strict safety verification process. Since an occurrence of an error in such software may lead to unacceptable results, such as death, injury, loss of property, or environmental harm. The development of safety-critical software must be in compliance with applicable safety standards [22] , e.g., IEC 61508 [8] for various kinds of programmable devices, DO-178B [12] for airborne systems, and ISO 26262 [9] for the automotive industry.

Formal methods are powerful techniques to verify programs, and they are gaining wide industrial acceptance. DO-178C [13], the new standard for airborne systems, includes a supplement on formal methods compared with DO-178B. Model checking, which is largely automatic when compared with other formal techniques (e.g. theorem proving), can be applied to verify programs against safety requirements.

However, there are some hard problems in applying model checking to safety requirement verification.

(1) Software safety requirements are usually derived from system-level safety analysis, e.g., Failure Mode and Effect Analysis (FMEA) [11] and Fault Tree Analysis (FTA) [7]. These requirements are expressed in natural language or semi-formal language statements and coarse-grained. So a method for decomposing safety requirements should be provided so that they can be traced into programs. For example, a safety requirement from [22] is *"avoidance of falling metal blanks"*. It is hard to identify which elements of a program contribute to this requirement, since it is too coarse-grained. The requirement can be decomposed as "(a) *the feed belt conveys a blank to a table* (b) *if the table is in load position*". The decomposed requirement consists of an action (a) and a state (b) which are sufficient to be traced in programs.

In addition, in order to verify safety requirements with model checking, the requirements must be converted into formal specifications. So a formal language which can express decomposed software safety requirements should be provided, and the translation from safety requirements into formal specifications should also be discussed.

(2) There is a gap between safety requirements and programs. Traceability between safety requirements and programs can bridge this gap, and traceability is a prerequisite of model checking safety requirements. Traceability can be seen as a relation between the elements in safety requirements and the elements in programs. There are two problems should be considered: (1) the representation of traceability information (or traceability information model) and (2) the method for building accurate and complete traceability between safety requirements and programs.

(3) Safety requirements are usually not expressed in conventional model checking specifications such as assertions and Linear-time Temporal Logic (LTL). So model checking techniques cannot be used directly, and safety requirement verification must be decomposed into problems which can be

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2666606

solved by existing model checking techniques. The decomposition method and the selection of model checking techniques should be considered.

I present a formal language called Event Automata (EA) for formalizing safety requirements related to event sequences, since most safety-critical systems are discrete-event systems. Traceability information models and conformance verification are based on the definition of EA.

2. RELATED WORK

There are three research fields related to my work: (1) formalization of software safety requirements, (2) traceability links from requirements to programs, (3) model checking techniques.

Formalizing a safety requirement is a prerequisite for model checking. Hansen et al. [15] proposed a method to derive software safety requirements from the results of fault tree analysis and formalize the software safety requirements into real-time interval logic formulae.

Traceability is one of the core principles mandated by all safety standards [8, 12, 9]. There is some recent literature concentrating on safety requirement traceability management [22, 2]. Nejati et al. [22] and Briand et al. [2] proposed a methodology for establishing traceability between safety requirements and SysML designs. The traceability information between a safety requirement and a design can be used to extract a design slice related to the safety requirement. The design slice is easy to understand and inspect.

Moreover, Ariss et al. [10] proposed a translation from a safety requirement expressed in fault tree into a statechart. The traceability between the statechart and a functional model is established during the translation. Then the statechart is integrated into the functional model by traceability information. The integration results in a integrated functional and safety specification model which is easy to validate. Different traceability information models are for different objectives (e.g., [22, 2] is for design slices and [10] for integration).

There is a lot of work on software model checking techniques. Spin [17] and Verisoft [14] are two model checkers for concurrent software using explicit model checking [4]. Symbolic Model Checking (SMC) [20] based on Satisfiability Modula Theory (SMT) is efficient and widely accepted in software model checking. Bounded Model Checking (BMC) [5] is an efficient SMC for finding bugs in programs, but it cannot provide proofs for the correctness of programs. In order to prove the correctness of a program, most SMC techniques construct an inductive invariant for the program. Predicate abstraction [6] uses predicates to construct an inductive invariant. Lazy abstraction [16] constructs an inductive invariant by unwinding a program into an Abstract Reachability Tree (ART). The abstract state spaces of the nodes in the tree can be generated by interpolants [21] or predicates [6]. A recently proposed model checking technique called IC3 [1] is proved to be efficient in the verification of hardware. Cimatti et al. [3] investigated IC3 in the setting of software verification and experimental results demonstrate its great potential.

My research is concerned with defining a traceability information model for model checking and constructing a proof for the correctness of a program. The model checking techniques used in my work include lazy abstraction and interpolants.

3. A SKETCH OF PROPOSED APPROACH

Most safety-critical systems are discrete-event systems, and system safety analysis (e.g., FMEA and FTA) is closely relevant to events. So my work concentrates on model checking programs with respect to properties which are related to event sequences generated by the programs. The framework of my research is illustrated in Fig 1. Rectangles represent artifacts indexed from 1 to 8. Arrows represent actions indexed from a to e. The details of five actions are as follows.

formalize: A software safety requirement arising from system safety analysis must be translated into a formal specification, and the specification is event sequence oriented.

build: The traceability between the formal specification and a program must be built. The elements of the formal specification that should be traced in the program must be decided. The traceability information model is sufficient for conformance verification.

comprise: The formal specification, the program, and the traceability information model (elements in thick dashed rectangle) are the three necessary parts of conformance verification.

reduce: The program can be reduced with respect to the requirement. Nejati et al. [22] proposed to use safety requirements to extract design slices. There are various state space reduction techniques that can be used to reduce the program.

decompose: The conformance verification between the safety requirement and the program must be decomposed into smaller model checking problems which can be solved by existing model checking techniques.

4. EXPECTED CONTRIBUTIONS

My work expects to provide a traceability information model to bridge the gap between software safety requirements and programs so that model checking techniques can be applied to check the conformance between them. The expected contributions are as follows.

- Giving a formal language which is event-driven and state-of-art. The formal language can also specify infinite state properties and safety requirements are conveniently formalized into it.
- Defining traceability information model whose objective is for model checking safety requirements.
- Methodologies for building the traceability between formal safety requirements and programs.
- Methodologies for decomposing safety requirement verification into problems which can be solved by existing model checking techniques.

5. RESEARCH METHODS AND CURRENT STATUS OF WORK

Safety requirements, specified by FTA or FMEA, have their own distinguish features. Firstly, a safety requirement represents a set of error paths which should be eliminated

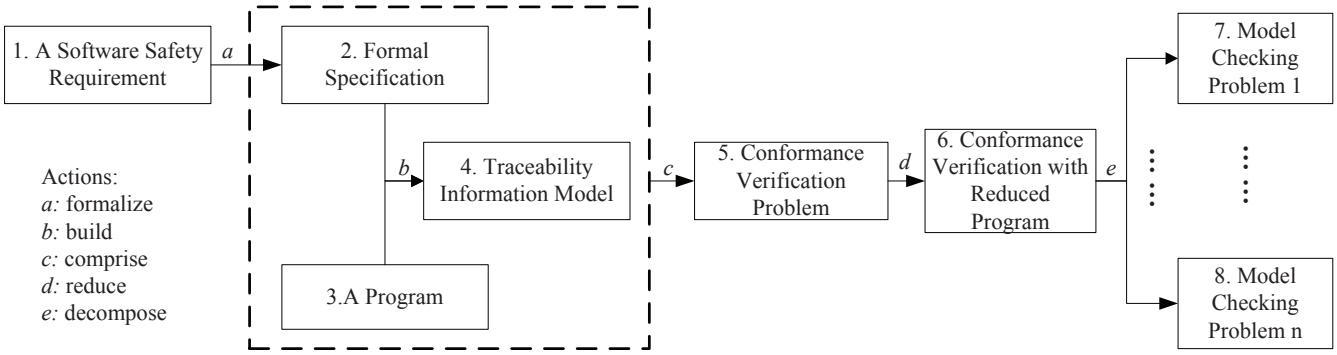

Figure 1: Framework of Model Checking Safety Requirements

in systems. Secondly, the basic elements of safety requirements are events and states. These elements constitute a set of error paths. The formal language should take into account these features so that safety requirements can be easily formalized. A formal language, which is event-driven and state-of-art, fits the above features.

5.1 Event Automata

I introduce a formal language called Event Automata (EA) to specify software safety requirements. An EA represents exactly a set of error paths which result in a failure in systems. A program is correct with respect to an EA if and only if all event paths generated by the program are not accepted by the EA. The definition of EA retains the notions of states, initial states, and accepting states of Finite State Automata (FSA). In addition, it contains variables and two new important notions.

The first important notion in EA is *events* and events are parameterized. An event consists of a name and a parameter list. Each parameter in a parameter list can be a variable or a concrete value. For example, $receive(1)$ is an event. *receive* is its name and concrete value 1 is its parameter. An event with all its parameters being concrete values is called ground event. If at least one parameter of an event e is a variable then the event can be seen as a set of ground events, where each ground event in the set replaces each variable parameter in e with a concrete value. Though all events generated by an program are ground events, events with variable parameters can simplify the specifications of safety requirements.

The second important notion is transitions. A transition in an EA is a five tuple $(q, event, guard, actions, q')$, where q is the source state, q' is the target state, *event* is the trigger event and it drives the EA to move from q to q', *guard* is the condition, and *actions* is a sequence of assignments. Its semantics is as follows: suppose the EA is in state q, when it encounters *event*, and *guard* is evaluated to be true, then the transition is enabled. When the transition is enabled, *actions* is executed and the EA enters state q'.

The formalization of safety requirements can be partially automatic. Ariss et al. [10] proposed an automatic method to translate safety requirements into statecharts. statecharts and EAs are similar in syntax and semantics.

5.2 Traceability

Defining traceability information model between an EA and a program should first decide which elements of the EA should be precisely traced in the program. My current work indicates that if all events defined in the EA and the initial state of the EA are precisely traced in the program then other elements of the EA, such as transitions and states, can be automatically traced, and with the traceability information, it is sufficient to apply existing model checking techniques to verify the conformance between the EA and the program. The traceability information model between an EA and a program is defined as the relationship between the program elements, such as program locations and statements, and the elements in the EA. A program location can be seen as the beginning of an execution or the ending of an execution. A program statement can be seen as the execution of a statement, and this semantics is corresponding to an event. There is a set of elements in a traceability information model which must be built manually. With this set of elements, other elements of the model are built automatically.

Since traceability information models are built manually, the accuracy of the human participation process must be considered. Because verification results are closely related to the accuracy and completeness of traceability information.

5.3 Model Checking

Conformance checking between an EA and a program is to verify whether there exists an event path generated by the program accepted by the EA. Conformance checking in my work is based on lazy abstraction paradigm [16]. Based on this paradigm, conformance checking is decomposed into three levels: tree level, path level, and edge level.

In the tree level, the algorithm focuses on constructing an unwinding of the combination of an EA and a program without considering whether the unwinding is *well-labeled* or not. Depth-first Search (DFS) is applied to construct the unwinding. Once an possible error path is found (i.e., the path is possible an accepting event path), the tree level algorithm applies path level algorithm to refute the path. If the path level algorithm returns that the path is feasible then a counterexample is found else the path is refined to reduce the state space of the path.

In the path level, the algorithm concentrates on deciding whether a path is feasible. If a path is feasible then it is an error path. In other words, the program does not satisfy the safety requirement. If a path is not feasible then the information obtained during refuting the path can be used to refine and strengthen the state space of the path. Since a

path is a sequence of edges in an unwinding, the treatment of an edge in the path level applies the edge level algorithm.

In the edge level, the algorithm concentrates on the treatment of an edge in a path. An edge in a path is the combination of a transition t in the EA and a program fragment cut by t. The edge is associated with a precondition and a postcondition. The edge level includes two operations: *Strenthenging* and *Preimage*. *Strenthenging* checks whether a clause in the precondition also holds in the postcondition. *Preimage* computes the preimage of the postcondition with respect to the edge and the precondition. These two operations can be obtained by adapting existing model checking algorithms.

The current status of my work has proceeded to the edge level. The future work is to give a description for the two edge level operations and implement a prototype to evaluate the efficiency and effectiveness of my method.

6. EVALUATION STRATEGIES

The preliminary strategies for evaluating my technique consist of two aspects: (a) the cost of using EAs to formalize software safety requirements and the cost of building the traceability between EAs and programs. (b) the effectiveness of model checking the conformance between EAs and programs.

The evaluation of (a) will be based on the practices in industrial case studies. For each case study, the effort of translating software requirements into EA specifications and building the traceability are manageable, and how many man-hours of a case study is recorded. The effort comparisons of different case studies will also be provided.

The evaluation of (b) will be based on experiments. For the two edge level operations, the experiments evaluate different techniques (e.g. IC3 and interpolants) supporting the operations. In addition, for different programs, the experiments will compare run times and memory consumption of the conformance checking for these programs.

7. LIST OF PUBLICATIONS

I have been a PhD student for one and a half years. I have an accepted paper [19] and a submitted paper [18].

8. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 61272083. I also want to thank my advisor Professor Huang from Nanjing University of Aeronautics and Astronautics for the guidance and support he provides to me.

9. REFERENCES

[1] A. R. Bradley. SAT-Based Model Checking without Unrolling. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[2] L. C. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, and T. Yue. Traceability and SysML Design Slices to Support Safety Inspections: A Controlled Experiment. *ACM Trans. Softw. Eng. Methodol.*, 23(1):9, 2014.

[3] A. Cimatti and A. Griggio. Software Model Checking via IC3. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.

[5] E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[6] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[7] U. N. R. Comm. *Fault Tree Handbook*. NUREG-049, 1981.

[8] I. E. Commission. Functional Safety of Electrical/Electronic/Programmable Electronic Safetyrelated Systems (IEC 61508). 2005.

[9] I. Draft Standard. Road VehiclesÍcFunctional Safety. 2009.

[10] O. el Ariss, D. Xu, and W. E. Wong. Integrating Safety Analysis with Functional Modeling. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 41(4):610–624, 2011.

[11] C. Ericson. *Hazard Analysis Techniques for System Safety*. John Wiley & Sons, 2005.

[12] R. T. C. for Aeronautics (RTCA) Inc. DO-178B-Software Considerations in Airborne Systems and Equipment Certification. 1992.

[13] R. T. C. for Aeronautics (RTCA) Inc. DO-178C-Software Considerations in Airborne Systems and Equipment Certification. 2011.

[14] P. Godefroid. Software Model Checking: The Verisoft Approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[15] K. M. Hansen, A. P. Ravn, and V. Stavridou. From Safety Analysis to Software Requirements. *IEEE Trans. Software Eng.*, 24(7):573–584, 1998.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In J. Launchbury and J. C. Mitchell, editors, *POPL*, pages 58–70. ACM, 2002.

[17] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[18] S. Kan, Z. Huang, Z. Chen, and W. Li. Partial Order Reduction for Checking LTL Formulae with the Nexttime Operator. *Journal of Logic and Computation*.

[19] S. Kan, Z. Huang, Z. Chen, and B. Xu. Bounded Model Checking of C Programs Using Event Automaton Specifications. *Journal of Software*.

[20] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[21] K. L. McMillan. Lazy Abstraction with Interpolants. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[22] S. Nejati, M. Sabetzadeh, D. Falessi, L. C. Briand, and T. Coq. A SysML-based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies. *Information & Software Technology*, 54(6):569–590, 2012.