Manju Nanda
Yogananda Jeppu *Editors*

Formal Methods for Safety and Security

Case Studies for Aerospace Applications

CSIR-NAL

Springer

Formal Methods for Safety and Security

Manju Nanda · Yogananda Jeppu
Editors

Formal Methods for Safety and Security

Case Studies for Aerospace Applications

Springer

Editors

Manju Nanda
Aerospace Electronics and Systems Division
CSIR-National Aerospace Laboratories
Bangalore, Karnataka
India

Yogananda Jeppu
Electronic CoE
Honeywell Technology Solutions
Bangalore, Karnataka
India

ISBN 978-981-10-4120-4 ISBN 978-981-10-4121-1 (eBook)
https://doi.org/10.1007/978-981-10-4121-1

Library of Congress Control Number: 2017934065

Preface

The first workshop on "Application of Formal Methods for Safety and Security Critical Systems (AFMSS-2016)" was organized by CSIR-National Aerospace Laboratories, Bangalore, India, on May 25–27, 2016. This workshop presented a platform for the discussion and representation of research and developments in the field of formal methods, and its applications for ensuring safety and security of safety-critical systems. A decent number of research papers from prospective authors were submitted for this workshop. The editors have selected 12 papers after the double-blind peer-review process by experienced subject expert reviewers chosen from various industrial and research domains, in India and abroad. The proceeding of AFMSS-2016 is a mix of papers from some latest findings and research and ongoing project activities of the authors.

The workshop aimed at bringing together formal methods practitioners, engineers, technologists, and academia. The aim is to create a forum for further discussions, an integrated information dissemination outlet in all aspects of formal methods and its applications. The call for paper addressed the following broad topics.

- Design, specification, code generation, and testing based on formal methods.
- Methods, techniques, and tools to support automated analysis, certification, debugging, learning, optimization and transformation of complex, distributed real-time systems and embedded systems.
- Verification and validation methods that address shortcomings of existing methods with respect to their industrial applicability (e.g., scalability and usability issues).
- Tools for the development of formal design descriptions.
- Case studies and experience reports on industrial applications of formal methods, focusing on lessons learned or identification of new research directions.
- Impact of the adoption of formal methods on the development process and associated costs and time.

- Standardization of formal methods application in industrial forums.
- Formal method-based tools analysis certification aspects toward safety-critical applications.

All the papers are focused on the thematic presentation areas of the workshop and they have provided ample opportunity for presentation in the sessions. Research in the application of formal methods has its own history and importance. This volume presents an exciting collection of contributions resulting from a successful call for papers. The selected papers including both review and research papers highlight the current focus of applications of formal methods on safety and security of safety-critical systems. There is no doubt that the tips, lessons learnt, and further advancements in formal methods and its applications in the safety and security of safety-critical systems presented in this volume will be of use to researchers and professionals alike.

It is been a great honor for us to edit the proceedings. We have considerably enjoyed working in cooperation with the international advisory, and program and technical committees to call for papers, review papers, and finalize papers for presentation and inclusion in these proceedings. The AFMSS conference and proceedings are a credit to a large group of people and everyone should be proud of the outcome. First and foremost are the authors of the papers, columns, and editorials whose works have made the workshop a great success. We had a great time putting together this proceeding. We extend our deep sense of gratitude to all for their warm encouragement, inspiration, and continuous support for making it possible.

We hope the readers will appreciate the good contributions made and justify our efforts.

Bangalore, India Manju Nanda
May 2016 Yogananda Jeppu

Acknowledgements

The first edition of AFMSS has drawn an impressive number of research articles authored by academicians, researchers, industry, and practitioners from across the world. We thank all of them for sharing their knowledge and research findings on an interactive platform like AFMSS and thus contributing toward producing such a comprehensive workshop proceedings of AFMSS.

The level of enthusiasm displayed by the organizing committee members right from the day is commendable. The extraordinary spirit and dedication shown by the organizing committee in every phase throughout the workshop deserve sincere and heartfelt thanks. It has indeed been an honor for us to edit the proceedings of the workshop. We have been fortunate enough to work in cooperation with a brilliant program and technical committee consisting of eminent academicians, industry personnel to call for papers, review papers, and finalize papers to be included in the proceedings.

We would like to express our heartfelt gratitude and obligations to the benign reviewers for sparing their valuable time and putting in effort to review papers in the stipulated time and providing their valuable suggestions and appreciations in improvising the presentation, quality, and content of this proceeding. The eminence of these papers is an accolade not only to the authors but also to the reviewers who have guided toward perfection.

Last but not the least, the editorial team at Springer deserves a special mention and our sincere thanks to them not only for making ours and everyone’s dream come true in the shape of this proceedings, but also for its hassle-free and in-time publication of this volume.

Contents

About the Editors

Dr. Manju Nanda has over 20 years of experience in design, development, and qualification of safety-critical embedded systems. Her core competencies lie in the field of safety-critical software engineering and embedded systems for safety-critical applications. She is involved in the design and development of safety-critical embedded systems in various domains such as medical, automotive, and aerospace. In the medical domain, she has worked on the design and development of drug infusion pump and controller, baby incubator, pulse oximeter, and semi-automatic clinical analyzer. In the automotive domain, she worked on developing the proof of concept of DC motor speed control unit, and crack detection and warning unit (CDWS). In the aerospace domain, she has worked on the design, development, certification, and qualification of smart fatigue meter, enhanced smart fatigue meter, SARAS stall warning and aircraft interface computer (SWS/AIC), SARAS automatic flight control system (AFCS), and SARAS engine indication and crew alerting system (EICAS). She has published over 400 technical documents related to the projects. She has published papers at international and national conferences, and in peer-reviewed journals.

Dr. Yogananda Jeppu holds a B.E. in electronics and communication, from Mangalore University, and a postgraduate degree in missile guidance and controls from Pune University. He has a Ph.D. in certification of safety-critical control systems using model-based techniques. He has been working in the field of control system design and implementation, simulation of aerospace systems, verification and validation for aircrafts and missiles for the past 28 years. He has several publications on formal methods, randomized testing, orthogonal array testing, and missile guidance and control. He is a recipient of many awards, most notable of which are the Commendation Certificate for "Significant Contributions made to the Integrated Guided Missiles Programme," and the "National Aerospace Laboratories Technology Shield for Outstanding Achievement in LCA Control Law Design, Certification and Successful Flight Tests." He started his career in 1987, working on missiles and the Indian Light Combat Aircraft programme with the Defense R&D Organization. He is currently working at Honeywell Technology Solutions as a staff engineer.

Chapter 1
Formal Methods—A Need for Practical Applications

Manju Nanda, J. Jayanthi and Yogananda Jeppu

Abstract This an introduction to the formal methods perception in the community today. There are many incidents of software failure, and formal methods are advocated as a sure fire solution to the problems. This is however not seen to be so by the majority. There are barriers to the formal methods that need to be overcome by the practicing engineer before it is accepted as a process in all the safety critical software development projects. The first Workshop on "Application of Formal Methods for Safety & Security Critical Systems (AFMSS-2016)" brings together the engineers, researchers, and academia who have used these techniques in the field and gathers their insight into the twelve selected papers. This paper connects the dots.

Keywords Formal methods · Safety critical · Software failures · Tool · Process

1.1 Introduction

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
>
> —Edsger Dijkstra

Formal methods are not new. They have been in existence for a very long time. In fact it is more than half a century now since the initial publications on mathematically correct programs. Many books and papers were written in the 1960s on this subject [1–5]. The two conferences in 1968 and 1970 sponsored by NATO

M. Nanda (✉) · J. Jayanthi
National Aerospace Laboratories, Bangalore, India
e-mail: manjun@nal.res.in

J. Jayanthi
e-mail: jayanthi@nal.res.in

Y. Jeppu
Honeywell Technology Solutions, Bangalore, India
e-mail: Yogananda.Jeppu@Honeywell.com

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_1

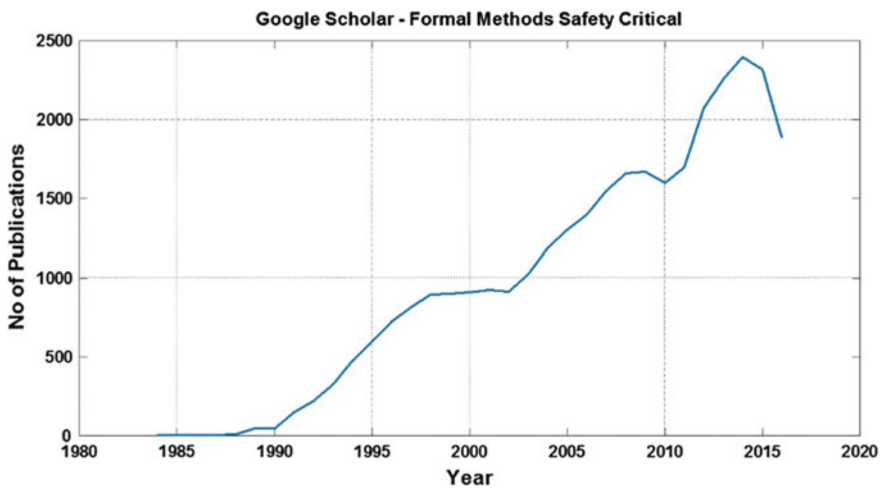


Fig. 1.1 Literature count on formal methods in safety critical

were responsible for getting together the software community and to acknowledge that programming was error prone [6]. The word “software engineering” was coined in these conferences. Even though the mathematics of programming was found to be important, it failed to come into the engineering domain as such.

A search of Google scholar for the terms “formal methods” and “safety critical” brings out the plot as shown in Fig. 1.1. This shows that the formal methods have picked up significantly in the last decade, doubling the number of all the previous years. The initial papers in this field are in 1984. A report from 1984 makes an interesting read. The following (quoted verbatim) by the author Currie reflects the thoughts in 1984, so much valid even in today’s software development scenario [7].

> These scrutinizes generally have not been highly formal - the tools and languages used to produce the programs do not lend themselves to formal methods. Note that introducing a formalism does not in itself improve the situation; the important thing is, in some sense, the “obviousness” of the proof of the program. For example, I would feel very dubious about a program which could only be proved with the assistance of a sophisticated theorem-prover: who proves the theorem-prover? On the other hand, it is clear that appropriate formalism in the specifications, languages, tools and computers used, can make verification much easier by preserving this “obviousness” of proof through its various transformations from specifications to code in ROMs.

A good example of formal modeling of requirements is by Patrick Doyle [8]. In the 1970s, Dr. Patrick Doyle was asked to develop a sales commission tracking system. He constructed a model of the system using set theory. He wanted to get this vetted by the board who had commissioned the project. Patrick gave the board an option of a long document that captured the requirements or a short course in set theory. They opted for the course and he could explain the model of the system to them using set theory and formally specify the requirements. This set of requirements were modified and signed off. The system worked fault-free from the very first time. This good example has however not been followed through. Practical formal methods did not gain momentum. We find errors in systems today.

1.2 Error and Failures in Software Systems

In 2005, IEEE spectrum published an article "Why software fails" [9]. Today after a decade there is an interactive Web interface to look at some of the failures in IT [10]. This commemorates the failure in the IT industry in the last decade. The conclusion is given as:

> While it's impossible to say whether IT failures are more frequent now than in the past, it does seem that the aggregate consequences are worse.

Software systems have failed time and again. The 2016 list of errors makes a good read, some of them even hilarious [11]. The Nest thermostats failed due to a software glitch leaving the users in the cold. A software upgrade fixed this problem. HSBC had a major software outage that lasted two whole days. In another incident, prisoners were being left early due to a software glitch in the prison system. In the aerospace industry and the medical field, which are safety critical, there have been software failures. The F35 bug report released early this year lists the software errors in the avionics system of the fighter aircraft [12]. The medical recall by the FDA of a faulty ventilator system with a potential to cause death is an example. "If a patient does not receive the amount of air set on the machine, they may need to be removed from the ventilator and placed on a different system. A patient not receiving enough oxygen, can result in possible injury or death" [13].

The reason for most of the software errors is not software alone, but it is the way the software interacts with the environment. The faulty or unthought of requirements cause most of the errors in the software. Figure 1.2 shows the errors in the

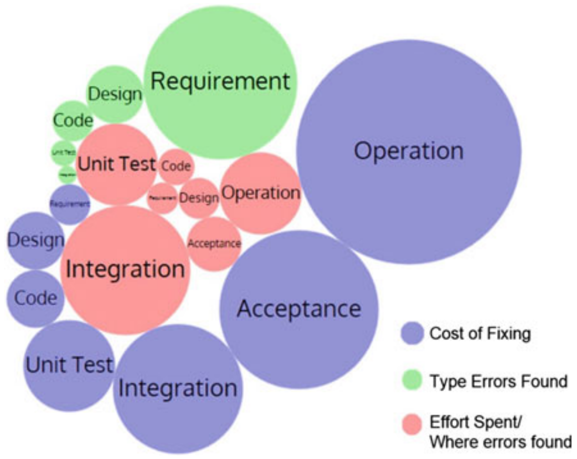


Fig. 1.2 Errors—where, what, and how much

software, where they are found, and the cost of fixing these errors [14]. The top three bubbles define that the maximum errors are found in requirements, design, and lastly code. The cost of fixing, a well-known fact, is the maximum as one fixes them later in the project lifecycle. Unfortunately, errors are found during the system-level integration tests, operations, and lastly the unit level tests. The fact that requirement errors are found during the integration tests or operations is the cause for faults having a catastrophic effect on the users. There is a need to correct these much earlier.

1.3 A Paradigm Shift in Systems Engineering

The classical system development process is the V model as shown in Fig. 1.3. The system development starts with a concept of operation, "ConOps," as it is popularly known, where a prototype is developed and initial algorithms may be tried out. This phase leads to the requirements gathering and finalization phase. The system-level requirements, the safety analysis, and the allocation of requirements to hardware and software are done at this phase. The requirements flow down to the high-level software and hardware specifications. These capture the intent of the software and the hardware. The next phase is the design where the software and hardware are designed separately. The output of this phase is a detailed design of the software. Today it is also a set of executable models. The next phase is coding or development. The use of autocode is very prevalent in the software development process in this phase.

The development proceeds upward on the V model from here. This is the test phase with unit tests on the small functions on hardware kits or the final board. This leads to the integration testing where all the codes are compiled together and downloaded on the actual hardware for an end to end test. This is followed by the system-level tests. In the case of aircraft avionics system, this phase is done on very costly rigs that mimic the actual aircraft on ground, normally called Ironbird. This is followed by acceptance tests and certification and deployment.

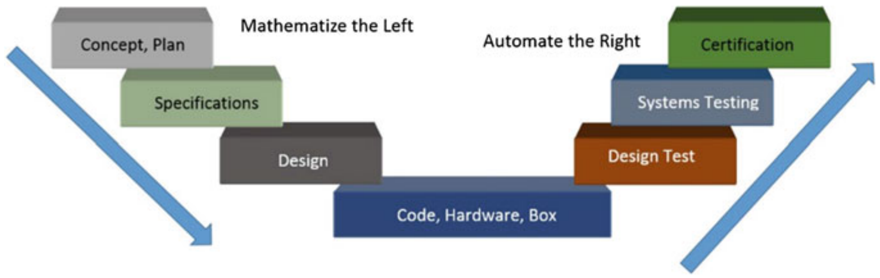


Fig. 1.3 System development V model

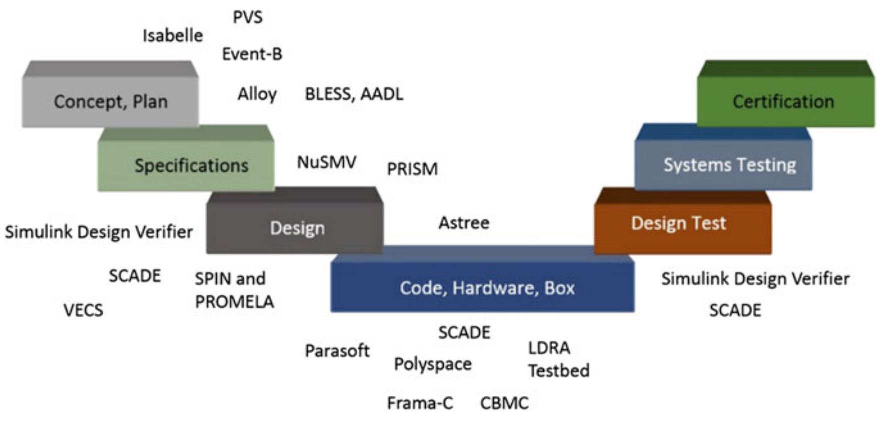


Fig. 1.4 System V model—location of formal method tool usage

In this traditional V model, we find errors in requirement during the final test phase. The amount of effort for testing is huge. Development activity in an aircraft project had 4 to 5 engineers doing the coding but at the peak a team of 40 engineers was doing the testing. This scenario is very common in the aircraft industry. It is very paradoxical that a large effort is spent on the right side to find defects of the left side. Systems engineering needs a paradigm shift to address this issue.

The mantra to be followed is “Mathematize the Left and Automate the Right.” It makes sense to put in more effort on the left side to find errors and correct them before they trickle down to the right side. Formal methods is an excellent tool to carry out this activity. There are many tools available today to enable the engineer to design systems better. Figure 1.4 shows some of the tools that can be used in the various phases of the system development process. At the requirements capture phase, tools such as PVS [15] can prove the correctness of the requirement by automated theorem proving. The tools and languages such as Event B [16] can help the designer in building up the system from an abstract to gradual refinements and increased complexity.

At the design level, models of the system can be checked for correctness against the specifications and assertion using automated model checkers. NuSMV [17], Simulink Design Verifier [18], and SCADE [19] are some of the open-source and commercial software available today. SCADE and Simulink Design Verifier can provide a certifiable C code automatically that can be compiled with the additional driver for direct implementation on the onboard computer. At the C code level, Abstract Interpreters such as Astree [20], Polyspace [21], LDRA Testbed [22], and Parasoft [23] are some of the commercial tools that are available today. The list is not exhaustive. These tools have been evaluated in some of the projects and are mentioned here. There are other tools included in Fig. 1.4 [24–31].

The right side of the V model can now be completely automated with the requirements that have been used as assertions. The test case can be automatically generated to provide requirements coverage and model coverage. Automated test case reviews can be done for completeness. These artifacts can be generated overnight or on weekends. The artifacts are available for reviews and certification. This paradigm shift would be a practical approach to our problem of defective software. This is an ideal situation. The industry does not see this in this way. There is a major question asked very frequently by the management. What is the return on investment?

1.4 Return on Investment

The formal methods dilemma today (quoted verbatim) [32]

> If formal methods is a best practice of software engineering, then an engineer who does not employ it is either negligent or incompetent. But formal methods is beyond the capability of typical software engineers (otherwise, why do we need formal methods experts and researchers?) or is too time-intensive to employ, so it cannot be considered to be a best practice today.

The question of return on investment was posed on the LinkedIn, and this elicited some very useful insights and viewpoints into the use of formal methods and how it is perceived by the community and the management. These views are presented verbatim (with minor language corrections) here

1. Formal methods as such have no ROI on themselves. They are supporting tools. But when used wisely in the context of the whole engineering process, the ROI can be enormous, albeit often indirectly. It starts with having correct Requirements/Specifications. One can see that what follows as a kind of compilation/translation process. Hence, formal methods can help in the abstract thinking to get the Requirement/Specifications right (and complete), whereas their use in the subsequent steps is more one of verification that it was done correctly.
   A major ROI that is often overlooked is that it can make systems a lot more efficient: firstly because they will be less complex and will take less resources, and secondly because the cost of correcting errors later on can be very expensive and is a function of the complexity. We witnessed this dramatically a few years ago when redeveloping our RTOS kernel from scratch using formal methods (using TLA+). The code size shrank with a factor 10. Maintainability was a lot easier too.
2. Before selling formal methods to managers and "pencil pushers," you need to sell them to practitioners. Not enough of them are comfortable with the theory and logic underlying formal methods. Then, although English is prone to all the mistakes talked about in too many papers on "How to engineer requirements the

wrong way," it is learned by many and flexible enough to capture any thoughts and requirements in any domain, which formal methods are often not.

3. Frequently the fact that formal methods do not solve a problem which the industrial engineering teams believe that they have, along with lack of tool support, which can be a show-stopper. Lack of commercial awareness and poor cultural fit between the researchers and practitioners (is another cause).
4. From a science-philosophical point of view, you would need to carry out a lot of systems engineering projects TWICE—always in one version with formal methods and then simultaneously also in another version without formal methods—such as to have a proper control group (for example, in pharmaceutical experiments with a placebo control group). Alas, such project duplications are never and nowhere done, because the duplication costs would be too high—nobody would like to pay the price of such a science-philosophically solid comparative empirical study. Alas, any "case study" without control group is at best "anecdotal"—in the worst case completely worthless from a scientific point of view.
5. "What are the issues in transferring researchers-driven technology to industry?" Well, when the question was applied to formal methods, the answer was that the methods themselves are the biggest issue. Most of the formal methods we know today are not designed in the first place with industry-end users in mind. To the best of my knowledge, I have seen no publication that evaluates a formal method from the perspective of industry participants who could share insights into why they like/dislike a method.
6. We have been very successful with using formal models in OS development. However, every time we start a new project using formal models, we have to face the question on how detailed the model should be. If it is too detailed on an industrial serious software project, formal models will very soon reach its practical limits. On the other side, with increasing abstraction levels formal models lose their additive effect rapidly. We noticed that there is no one-fits-all solution for that. We did not also find yet good decision criteria to help us on finding the right abstraction level for a formal model.
7. I'd say it depends on what you are designing. If you look at it from a process perspective, the whole software development process is like a funnel, from vague requirements over a (hopefully) not so vague design to a concrete implementation.

   While this is usually fine for systems where there is no "right" or "wrong" solution (the look and feel of a Web shop, for example), other systems are much more restrictive. You do not want to be vague about alarm and shutdown criteria in the control software of a nuclear power plant or in aircraft control.

   So I think using formal methods will be worthwhile in only some scenarios, while in others semi-formal notation (such as UML) will suffice.
8. The initial hurdle, when a formal method of some sort is suggested, is usually ignorance. The people who are in a position to decide what methods and processes to use will be familiar with the current fashionable techniques, but regard

formal methods as somehow very difficult, outdated, and impractical—and usually that is all that they know (i.e., they have no experience, but they know it so surely that it is enough to decide against using such methods—"everyone knows this").

If you get beyond that, they may accept a trial of some sort, perhaps running in parallel with more traditional informal methods. As soon as they see the notation and realize that it is not familiar—or heaven forbid, has some letters upside down or reversed—they will assume that your specification is "difficult" (as "everyone knew it would be"). If you have a notation that allows you to use the text "forall" and "exists" instead, you may make a little headway, but then you may bang into another difficulty where the designers who are used to writing informal specifications suddenly realize that you are suggesting that they write in a notation which will not allow them to make hand-waving generalizations any more. This suddenly sounds like a lot of work, and they may go off the idea, saying that it is too expensive (as "everyone knew it would be").

If you get beyond that, you need to convince the implementers to faithfully implement what the formal specification describes. Your chosen method may include the refinement of the specification into code, in which case engineer ignorance will usually scupper you again—it will be rejected as too difficult and expensive. If your method is more lightweight, the implementers will look at the carefully crafted specification, wonder why you bothered, and then implement something that *they* understand which approximates what you said—probably based on the natural language supporting description. In other words, they will try to do what they usually do with informal specifications, and your hard work will be deemed a waste of time.

Luckily, if you can get past the implementation stage, the "payback" of the formal specification in terms that the integration and system test teams need is usually well received. You need to provide them with some sort of animated oracle based on the specification, or a set of generated test descriptions and results which they can use directly. But compared to their normal job of trying to figure out what to test from vague, incomplete and often inconsistent specifications, this is a dream come true.

Along the way, you will struggle to convince project management that the increased up-front cost of specification ("design") will be more than paid back during the implementation, test, and maintenance phases. Project managers are usually measured by delivering jam today, and kudos for bringing in the jam during test and maintenance will usually go to another manager anyway. So this means the managerial buy-in has to be at a level where the overall program costs matter over a considerable timescale. Unfortunately, at that level awareness or interest in the details of the development process is rare, and worse you will find it hard to produce evidence to back up the cost saving claim (though there is some, it is not "industry accepted").

So the short answer is that there are many difficulties introducing formal methods. But I would be very interested to hear if anyone has found effective ways to solve them!

These in general summarize the views on the usage of formal methods in the industry today. This is also supported by the many surveys on usage and problems of using formal methods in the industry.

1.5 A Need for Case Studies

There are barriers to the application of formal methods in projects. This is brought out in the survey in the aerospace industry [33]. The first barrier is education. This is also brought out in the LinkedIn discussion on "need to sell it to practitioners." General education on formal methods is needed especially for the engineer. This is a major gap that needs to be filled. The universities are working on formal method tools and improving them. This is still a mathematical and algorithmic exercise. The practicing engineers need to be shown the practical aspects of the use. It should start as tool use exercise and build up the mathematical background around this practical application to remove the "magic" of the tool. At a company level certification authorities need to be educated on how to evaluate the formal method artifacts. A tool qualification is not the only solution. The formal tool needs to be seen as an additional engineering tool to validate requirement. It should be acceptable as the "bode" command or the "fft" command of the control and signal processing tools today. This ease of usage will only come with the use of these tools on many projects and benchmark problems.

The second most barrier is the tool themselves. The tools today do not cater to all types of problems or the workflows. There are far too many tools in the market. This is somewhat like the early days of the Computer Aided Control System Design (CACSD) tools [34]. Every laboratory and university had its flavor of the control system design tool. This is today standardized into a few commercial tools. The commercial tools available today do a lot of practical work. They are excellent in finding issues in mode transitions and logic. The safety critical system is however not limited to these alone. The dynamics of a control system are so different with time-varying entities. It is always possible to have very high values of variable inside the compiled code even though the inputs are limited. This appears strange to computer engineers but such odd behaviors have caused failures in flight [35].

The third barrier is the industrial barrier. Often enough we see projects are already on their way with the Plan for Software Aspects of Certification (PSAC) already frozen. The argument provided by the project teams is "yeah, this is good, we wish you had approached us two years back." In the case of new projects, the standard reply is "we are having legacy code—which has NO errors." At such times, reminding the team of the Arianne 5 mishap does not win votes for the use of formal methods in projects. We have seen interesting work being done on proof of concept of proving the efficacy of the tool on many projects. The results are good but it is either too late or not budgeted for in the project plan.

These three main barriers to the implementation of formal methods require a strong push from the formal methods community to overcome it. The inertia is too large for an individual or a project group to overcome. This workshop hopes to build this ecosystem for formal methods practitioners. Some of the papers presented here indicate a positive trend toward acceptance of the methods in safety critical applications.

1.6 Paper Summary

There were four invited papers which are not available as part of this compendium. The twelve papers present different views and ongoing work on the use of formal methods in research, industry, and academia. There is a variety in the papers from a unified-framework architecture which integrates various formal method tools used in the development of safety-critical systems and their underlying software to a university paper on UTM firewalls. A paper extends the paradigm of constrained objects with a temporal component which caters to the time-varying aspect of vehicular systems. Another paper presents the use of AADL for safety validation of an embedded real-time system. The usage of tools such as NuSMV, Simulink Design Verifier, and SCADE is covered as industry experience papers. The future of the use of formal methods is brought on in a conclusion as "increasing complexity of SoC designs and the software algorithms and the distributed form they take, combined with the short release cycles of hardware and software, it is no surprise that formal methods will be a key technology in ensuring reliability of future software and hardware designs."

1.7 Final Words

An ounce of practice is worth more than tons of preaching

—Mahatma Gandhi

This is a quote that will help the formal methods community to move forward. There is a need for a platform where even small proof of concepts are shared with the community at large. The tool vendors, the partitioning engineer, and the academia need to get together to solve the engineering problems now after 20 or more years being in the academic domain. The more we use the tools and apply it to the problem at hand, the more will we be dexterous in the usage of the methods. The finals words are not ours.

Some people don't like change, but you need to embrace change if the alternative is disaster

—Elon Musk, founder, CEO and CTO of SpaceX

References

1. Schwartz J (1967) Mathematical aspects of computer science. In: Proceedings of symposia in applied mathematics. American Mathematical Society, Rhode Island
2. Burstall RM, Landin PJ (1968) Programs and their proofs: an algebraic approach. Technical report, DTIC Document
3. Hoare C (1969) The axiomatic basis of computer programming. Commun ACM 12 (10):567–583
4. Knuth D (1968) The art of computer programming, vol 1: fundamental algorithms. Addison-Wesley, Reading, MA
5. McCarthy J (1962) Towards a mathematical science of computation. In: Popplewell C (ed) IFIP world congress proceedings, pp 21–28
6. Naur P, Randell B (1969) Software engineering: report on a conference sponsored by the Nato science committee, Garmisch, Germany, Scientific Affairs Division NATO
7. Currie IF (1984) Orwellian programming in safety-critical systems, ADA168085, Royal Signals and Radar Establishment Malvern (England)
8. Cohen B, A brief history of formal methods. https://www.researchgate.net/publication/233960390
9. Charette RN (2005) Why software fails [software failure]. IEEE Spectr 42(9):42–49. doi:10.1109/MSPEC.2005.1502528
10. Lessons From a Decade of IT Failures. http://spectrum.ieee.org/static/lessons-from-a-decade-of-it-failures. Accessed 20 Sept 2016
11. Lee C (2016) Top software failures 2015/2016: Amazon, RBS. Starbucks—the worst software glitches this year. http://www.computerworlduk.com/galleries/infrastructure/top-10-software-failures-of-2014-3599618/#1. Accessed 20 Sept 2016
12. Director, Operational Test & Evaluation (2016) F-35 Joint Strike Fighter (JSF), [FY2015 Annual Report, 2016]. http://www.dote.osd.mil/pub/reports/FY2015/pdf/dod/2015f35jsf.pdf. Accessed 20 Sept 2016
13. Covidien, Puritan Bennett 980 Ventilators. http://www.fda.gov/MedicalDevices/Safety/ListofRecalls/ucm460955.htm. Accessed 20 Sept 2016
14. Fischer N, Salzwedel H (2012) Validating avionics conceptual architectures with executable specifications systemics. In: Cybernetics and informatics volume 10-Number 4. ISSN 1690-4524. http://www.iiisci.org/journal/CV$/sci/pdfs/HNB229PW.pdf. Accessed 20 Sept 2016
15. PVS, https://github.com/nasa/pvslib
16. Event B, http://www.event-b.org/install.html
17. NuSMV, http://nusmv.fbk.eu
18. Simulink Design Verifier, http://in.mathworks.com/products/sldesignverifier/
19. SCADE, http://www.esterel-technologies.com/products/scade-system
20. Astree, http://www.astree.ens.fr/
21. Polyspace, http://in.mathworks.com/products/polyspace/
22. LDRA Test Bed, http://www.ldra.com/en/
23. Parasoft, https://www.parasoft.com
24. Alloy Analyzer, http://alloy.mit.edu/alloy/
25. SPIN and PROMELA, http://spinroot.com/spin/whatispin.html
26. VECS https://cse.cs.ovgu.de/vecs/index.php
27. PRISM, http://www.prismmodelchecker.org/
28. BLESS, AADL and OSATE, http://osate.github.io/
29. Frama-C, http://frama-c.com/
30. CBMC, http://www.cprover.org/cbmc/
31. Isabelle, https://isabelle.in.tum.de/
32. Abramson D, Pike L (2011) When formal systems kill: computer ethics and formal methods. Newsletter on philosophy and computers, vol 11, No. 1. American Philosophy Association

33. Davis JA et al, Study on the barriers to the industrial adoption of formal methods. Formal methods for industrial critical systems, vol 8187. Lecture Notes in Computer Science, pp 63–77
34. Frederick DK, Herget CJ, Kool R, Rimvall M (1987) ELCS-the extended list of control software. Eindhoven University of Technology, Department of Mathematics and Computer Science
35. Jeppu N, Jeppu Y, Murthy N (2015) Arguing formally about flight control laws. Paper presented at international conference on industrial instrumentation and control (ICIC), pp 378–383. 28–30 May 2015. doi:10.1109/IIC.2015.7150771

Chapter 2
Formal Methods and Tools for Safety of Critical Systems

K.S. Kushal, Manju Nanda and J. Jayanthi

Abstract Advances in the quality of Safety-Critical Software Systems are very much essential in addressing the correctness, safety and security attributes of the system. The development processes of such critical systems are imperative at corresponding stages in accomplishing its key attributes. The use of formal methods and tools coupled with formal verification techniques presumes explicit definition of system and its properties which meets the specifications. A meticulous mathematical notation used to represent the critical systems at early stages of their development process is the substratum of Formal Methods. Model checking, a formal verification technique, encompasses specification and modelling languages that improve the overall software architecture. This paper describes various tools at different phases of development process of Safety-Critical Systems, aiding formal methods and verification techniques in software practices. Also we present a unified-framework architecture which integrates various such tools used in the development of Safety-Critical Systems and their underlying software.

Keywords Formal methods · Formal verification techniques · Model-checkers
Specification languages

K.S. Kushal (✉) · M. Nanda · J. Jayanthi
Aerospace Electronics & Systems Division, CSIR-National Aerospace Laboratories, Bengaluru, India
e-mail: ksk261188@gmail.com

M. Nanda
e-mail: manjun@nal.res.in

J. Jayanthi
e-mail: jayanthi@nal.res.in

2.1 Introduction

The use and application of mathematical techniques, Formal Methods, in exploring the systems relative behaviour is benefitted with predictive and accurate computation of properties of the system. Safety-Critical Systems are such systems which exhibit discrete behaviour and state diversifications, with abrupt transitions enabling the state changes successively. Formal Methods defy this distinct behaviour of these systems and predicts the system properties with the help of mathematical models. Such models also ensure in improvising the quality attributes in developing the system. This in turn increases the confidence in achieving highly integrated software. The formal methods mandate the use of formal specifications and models using formal languages during the development life cycle. The formal specifications govern the development stages such as design, architecture, implementation, coding, verification, validation and maintenance. These provide an accurate means of predicting and analysing the behaviour of the system. Application of these in the early stages of design and development life cycle of the software system provides corrective and preventive measures in their specification for the determination of their satisfiability.

With the use of formal methods and its techniques a compelling capability of improving the quality of the software being developed is provided, by the mathematical models/theorems [1]. During recent times, with the advances in the complexity of the hardware and the software counterparts of the Safety-Critical Systems, a rigid and a robust technique is very much required during its development phase in their entire life cycle. Various qualitative metrics are to be considered in analysing the impact of the application of formal methods from that of the conventional methods. Quantifying the software of Safety-Critical Systems in applications such aerospace, security applications (such as Net-Centric Warfare and Cyber-Physical Systems), pose a great challenge in assessing their qualitative metrics for functionality, safety and security. Safety, reliability and security are the major concerns in engineering an integrated module for Safety-Critical application. The consideration of these criticalities early in the development phase provides substantial foundation and enhances the efficacy of the application of formal methods and their supported tools. In addition to these methodologies, formal verification techniques yield affirmative solutions in deciding the critical decisions and theories involved in the design and development of Safety-Critical System. These further induce a sense of confidence in the practical developmental procedures of the software counterparts for Safety-Critical Systems.

In this paper, Sect. 2.1 gives an introduction about the need for Formal Methods. Section 2.2 provides an insight into the pre-work about the formal methods and tools for Safety-Critical Systems, while Sect. 2.3 gives an approach adopted in presenting a unified-framework architecture for the application of formal methods in Safety-Critical Systems. Section 2.4 deals with the conclusion and future scope of the work presented in this paper.

2.2 Literature Survey

2.2.1 Formal Methods-Based Database—Intelligent Knowledge Database (IKD)

Over a decade, several processes aiding the enthusiasts, researchers and developers in retrieving the information related to formal methods and formal verification techniques have proven to be a major impediment. These kinds of information retrieval systems and engines needs to be intelligent and flexible in making the search options and the process of recovering the necessary information much simpler and user friendly. Such systems handles humongous amount of data pertaining to formal methods. Dr. (Ms.) Nanda et al. proposed an intelligent, integrated, unified and efficient intelligent knowledge database system [2]. This system served the purpose of retrieving the information from a single source. This is very much essential in enabling the analysis of safety critical applications wherein the formal methods and the formal verification techniques are applicable. The IKD tool acts as an extensible knowledge-based tool which provides with the crucial information pertaining to formal methods, their applicability and the tool support comprehensively. This tool is very robust with the segregation of the information related to these methodologies and its applications at abstract phases of the development life cycle. This tool basically makes use of the keyword search engine [3] as proposed by Monika Henzinger in retrieving the Web Information. Thus, this tool is alike encyclopaedia for various formal methods and formal verification techniques, as well as the supported tools/tool chains along with their application differentiated under a broad classification set.

In the due course of design and development of such a knowledge database [4], it was surveyed that there tends to be a skeptical view about the usefulness of these methodologies and techniques. Also the need for a unified platform wherein such methodologies and techniques along with their respective tool support can provide the necessary artifacts supporting the need to consider such methodologies. This in turn enhances the safety-critical system's software counterpart expectations as desired.

2.2.2 Development of Tool Related and Tool Applicability Metrics

The advancements in the software counterparts of Safety-Critical Systems are flourishing. This has resulted in an increase in the functionalities being embed in the software. This is as a result of a substantial increase in the number of electronic components within avionics systems. As an impact of these there is an increase in the automation required to address the criticality that are involved in their development life cycle. All these factors yield a lighter-greener aircraft with state-of-art technologies, control and functionalities. The association of formal methods and formal verification techniques with the adherence to the standards has found acceptance for evolution, over a wide spectrum of methodologies, tool and techniques which can formally meet the specifications as desired.

A one point solution to this problem is to use normal conventional techniques and methodologies. These techniques which are aided by their respective tool/s for specific phases in their developmental life cycle are utilised. The analysis of the results at each stage in their development life cycle—i.e. requirement, architecture, design, code and testing are performed to ensure the software safety and reliability of certain Safety-Critical Systems. This proves to be more tedious and costly in producing the desired quality of safety-critical software [5]. From the rigorous analysis, it was evident that the conventional techniques were less effective and efficient because of the below factors:

- Highly person dependent—varies from individual to individual (interpretation of the specifications)
- More effort was required in terms of time and money

These factors are derived by the type of tools, their capabilities, operating environments, feature-set and their engines based on the applications for which they were used. Also the metrics generated by these tools or tool-set is also evaluated under various scenarios in order to conclude that conventional methods or techniques are not best suited for such kind of applications. Using formal methods and techniques, associated tools were found to provide with much better quality output. This plays a very crucial role in instigating a high level of confidence in assuring the correctness and completeness of the software. It was also found that the compliance to the specifications was formally proved. The metrics such as completeness, property violations, verification of the critical specifications and its properties, and validating the output with that of the specifications [1] with the generation of test cases were evaluated.

2.2.3 Development of Process Related Metrics

The processes in the development life cycle for safety-critical software are the phases relevant to their development process, as shown in Fig. 2.1. These are broadly classified intothe following:

i. **Requirements Capture and Management**: The specification of the requirements for the design and development of the software desired for the Safety-Critical System. These specify the required functionality that is desired for the Safety-Critical System to be embedded with.
ii. **Architecture Design**: The capture of the requirements and interpretation of the entire software as blocks, with the behavioural metrics such as data flow representation and control flow representation.

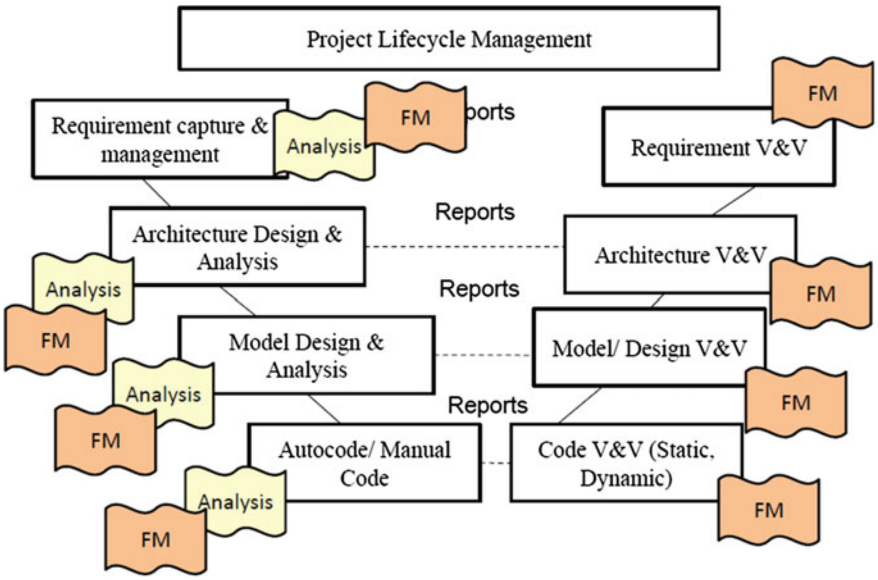


Fig. 2.1 Software development life-cycle—V Model

iii. **Model Design**: The implementation of the software behaviour with the specification of required attributes and necessary arguments. This also includes the mechanisms and the objectives that conform to the actual specifications.
iv. **Auto-code/Manual Code**: The annotation of the attributes, arguments with the specifications, written usually using a high-level language (C, C++, JAVA, Ada) in representing the desired system. This is either manually generated or automated from the design or sometimes from the architecture design.
v. **Testing (Verification and Validation)**: The verification of the annotation with the inclusion of pre and post conditions in validating the system performance. This can also be either automated or manually written with suitable constraints as assertions, flagging the time consumption in the process. This is dependent on the type of compilers used and the language in which the code is generated (if automated) or written (if manual).

2.3 Approach

With the application of conventional Software Development Life-Cycle (SDLC), there is zero automation and it requires manual intervention at each individual and relevant phases. Also there is no application of formal methods and techniques in conventional SDLC. This proves less effective with the necessity for lot of manual effort, time and cost. The evolution of this particular methodology resulted with partial automation of the SDLC as Model-Based SDLC. This process stands ahead with more effectiveness as the manual effort required is comparatively less (only required at suitable phases, with application of formal methods and techniques also at certain phases), and also cost and time effective. The complete automation of the entire SDLC with the integration of various formal methods and techniques, tool/s that support formal methods and techniques are included. This provides a seamless transition from one phase to another. This also ensures the correctness and completeness of each phase. This is basically dependent on the tool or sets of tools used while automating the entire process.

The evaluation of metrics generated from the inherent phases of SDLC and the tool/s support at each phase of development, suggests a need for migration from no automation process (conventional SDLC) to complete automation. This helps in imbibing a higher confidence level at relevant phases of development of software for a particular Safety-Critical System. The exclusive implementation of formal methods and formal verification techniques in either of the processes will adhere to one of the standards in their field of application such as Aerospace (RTCA DO-178B/C [6, 7], ARP 4761 [8]), Industrial (IEC 61508 [9]), Military (MIL-STD-254 [10]), with the assurance of a seamless integration of various tools and techniques that are based on formal methods.

2.3.1 RTCA DO-178B/178C Software Development Life-Cycle

The Aerospace standard RTCA DO-178B/178C SDLC includes the following set of details at each phase, which are necessary for software to be qualified. This is without the supplements and there is no mandate as for the application of formal methods and techniques. There are two phases [6]:

1. System Life-Cycle Phase
    i. Planning Process
        a. Plan for Software Aspects of Certification
        b. Software Development Plan
        c. Software Configuration Management Plan
        d. Software Quality Assurance Plan
        e. Software Verification Plan
    ii. Development Process
        a. Software Requirements Data
        b. Software Design Description
    iii. Verification Process

   a. Hardware Software Integration Test Plan
   b. Low Level Integration Test Plan
   c. HIS Verification Cases and Procedures
   d. Verification Results

2. Certification Phase

   i. Software Accomplishment Summary
   ii. Software Configuration Index
   iii. Software Life-Cycle Environment Configuration Index
   iv. LLI Test Report
   v. HIS Test Report
   vi. Traceability Matrix—Traceability of System Requirements to Software Requirements and Software Requirements to Test Cases.

The above-mentioned phases are in coherence with the RTCA DO-178B workflow. With the inclusion of formal methods and the process of automation, the standards have been evolved with additional supplements being provided with DO-178C as follows:

1. Tool Qualification (RTCA DO-330)
2. Model-Based (RTCA DO-331)
3. Object-Oriented (RTCA DO-332)
4. Formal Methods (RTCA DO-333)

The standards mentioned above mandates the use of formal methods and techniques accordingly depending upon the type of the software being developed.

The work in this paper concentrates on automating the SDLC and provides a seamless integration between various tools either developed in-house or commercially off-the-shelf tools. This also aims at integrating various formal techniques such as formulae, theorem provers, model checkers. The approach is defined based on various tools or set of tools that are supporting formal methods and techniques, at different phases of the development process of the software for Safety-Critical Systems. This forms a basis for the development of a unified-framework architecture that provides a single platform to integrate various formal methods-based tools and techniques and also handle the process of SDLC with much ease. This is done with the development of expertise over the appropriate tools at the appropriate phases by means of case studies. Also the metrics generated by these tools or techniques at each phase, for the safety application, are analysed by means of case studies. Each case study to substantiate the SDLC process is being followed with RTCA DO-178B Level A Criticality.

At each phase of the SDLC in the unified-framework architecture, the software is subject to extensive formal methods. The application of the formal methods and techniques in the unified-framework architecture monitors the process at each phase substantially supported by the analysis also based on formal methods and techniques at their respective phases. The dependency of each with their subsequent phases is addressed by this framework. This also provides a backward traceability

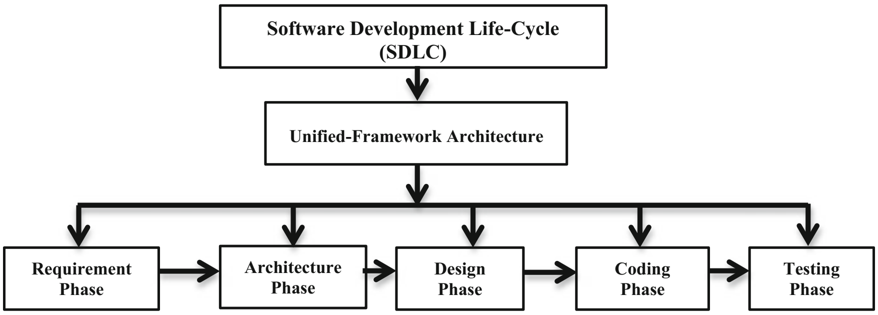


Fig. 2.2 Unified-framework architecture

and compatibility with the phases in the development life cycle, as shown in Fig. 2.2. The safety analysis of the software is addressed in the analysis of the software at its relevant phase. The unified framework also supports the conventional and partially automated workflows apart from the complete integrated automation in order to compare and evaluate the metrics. The framework also provides the standard guidelines and also mandates the application of criticality levels at suitable phases and applications.

For example; the safety analysis at the architecture phase is analysed with the help of FTA (Fault-Tree Analysis) [11], and FMEA (Failure Mode and Effect Analysis) [12], with the help of formal tools such as OpenFTA [13] and OSATE [14] respectively. The metrics generated by these tools are evaluated to establish the robustness and also include high-level confidence in the development of the software.

2.4 Conclusion and Future Scope

The analysis of the metrics generated from various tools or set of tools, their evaluation led to the need for the development of a unified formal guidelines, metrics for safety and security applications. This is needed in order to ensure the necessity for such applications and at CSIR-NAL we are developing a unified-framework architecture to seamlessly integrate tools and techniques, proving their applicability. As a part of the future work, design and implementation of the unified-framework is being carried out. The metrics are generated with extensive case study from the unified-framework for benchmarking the formal methods-based tools and techniques. This activity is also being carried out at CSIR-NAL. The improvisation of these benchmarks will also be handled in future by taking case studies from various other safety applications such as medical, military. It is also planned to integrate the safety semi-formal method-based techniques as a part of this unified-framework.

Acknowledgements The authors would like to thank the Director, CSIR-NAL, Bengaluru for supporting this work.

References

1. Heitmeyer C (2005) Developing safety-critical systems: the role of formal methods and tools. In: Proceedings of the 10th Australian workshop on safety related programmable systems, Sydney, Australia, 25–26 Aug 2005, pp 13–29
2. Nanda M, Jayanthi J, Madhan V (2012) Intelligent knowledge database (IKD) tool for formal methods. Int J Soft Eng Appl (IJESA) 3(6):117–127. doi:10.5121/ijesa2012.3609
3. Henzinger M (2000) Tutorial: web information retrieval. IEEE Proceedings of 16th international conference on data engineering. 29 Feb–03 Mar 2000, San Diego, CA, pp 693. ISBN: 0-7695-0506-6, ISSN: 1063-6382
4. Dondossola G (1998) Formal methods in the development of safety critical knowledge-based components. Proceedings of the european workshop on validation and verification of knowledge-based systems, CEUR workshop proceedings. 06–08 June 1998, Povo, Trento, Italy, pp 01–12. ISSN: 1613-0073
5. Place PRH, Kang KC (1993) Safety-critical software: status report and annotated bibliography. Technical Report, CMU/SEI-92-TR-5, ESC-TR-93-182, June 1993
6. King T (2012) Reusing certified, safety-critical avionics software. In: 2012 IEEE/AIAA 31st digital avionics systems conference (DASC). 14–18 Oct 2012, Williamsburg, VA, pp 6A1-1–6A1-6. ISBN: 978-1-4673-1699-6
7. RTCA DO-178B/C, The Aviation Golden Standard, http://www.rtca.org
8. ARP 4761—Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment http://standards.sae.org/arp4761/
9. IEC 61508 – functional Safety Standard for Electronic Safety related Systems http://www.iec.ch/functionalsafety/
10. MIL-STD-254 https://www.document-center.com/standards/show/MIL-STD-254
11. Joshi A, Vestal S, Binns P (2007) Automatic generation of fault trees from AADL models. In: Workshop on architecting dependable systems (DSN '07), Critical Systems Research Group
12. Grunske L, Han J (2008) A comparative study into architecture-based evaluation methodologies using AADL's error annex and failure propagation models. In: IEEE computer society, 2008 11th IEEE high assurance systems engineering symposium, pp 283–292. ISSN: 1530/2059/08
13. OpenFTA—Advanced Tool for Fault Tree Analysis http://www.openfta.com/
14. OSATE—Open Source AADL Test Environment http://osate.github.io/

Chapter 3
Taming the Enemy: Framework for Comparative Analysis of Safe String Libraries

Manupriya Srivastava, T. Rajani, S.N. Anitha Kumari, Chitra Viswanathan and Subrata Rakshit

Abstract Strings are of special concern in secure programming because they account for most of the data exchanged between an end user and a software system. Weaknesses in string representation and manipulation have led to a broad range of software vulnerabilities in C language programs. Over the years, several safe string libraries have been written in an attempt to prevent these vulnerabilities. The purpose of this work is to develop a framework for comparative analysis of safe string libraries. This encompasses (a) devising metrics for comparison of safe string libraries (b) creation and execution of testsuites for each library under consideration with the purpose of calculating the comparative metrics. This framework can be used as a sound basis for recommending the usage of a composition of specific safe string libraries.

Keywords Safe string libraries · String manipulation error · Bound protection metric · Functional coverage metric

M. Srivastava (✉) · T. Rajani · S.N. Anitha Kumari · C. Viswanathan · S. Rakshit
Centre for Artificial Intelligence and Robotics, DRDO, Bangalore 560093, India
e-mail: mpriya@cair.drdo.in

T. Rajani
e-mail: rajani@cair.drdo.in

S.N. Anitha Kumari
e-mail: anitha@cair.drdo.in

C. Viswanathan
e-mail: chitrav@cair.drdo.in

S. Rakshit
e-mail: srakshit@cair.drdo.in

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_3

3.1 Introduction

Strings—such as command-line arguments, environment variables, and console inputs—are of special concern in secure programming because they account for most of the data exchanged between an end user and a software system.

Weaknesses in string representation and manipulation [1] have led to a broad range of software vulnerabilities and exploits. Many of these vulnerabilities in existing C code result from the use of potentially dangerous functions such as strcpy(), strcat(). Unfortunately, because these functions are standard, they continue to be supported, and developers continue to use them—often resulting in disastrous consequences.

3.1.1 How Are Strings Represented in C?

A string in C is a contiguous sequence of characters terminated by and including the first null character as shown in Fig. 3.1. The length of a string is the number of bytes preceding the null character. The size of a string is the number of bytes allocated to the array associated with the string. For a properly null-terminated string of type char, length <= (size −1).

3.1.2 Common String Issues in C

Unbounded string copies

An unbounded string copy happens when data is copied from an unbounded source to a fixed size destination string, which may result in overwriting of the adjacent memory. Use of dangerous functions such as gets() and strcpy() leads to unbounded string copies.

Off-by-One Errors

An off-by-one error occurs when the space allocated to the string is one less than the size needed to hold the terminating null character.

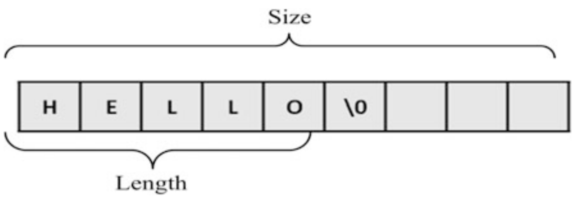


Fig. 3.1 Representation of a string in C

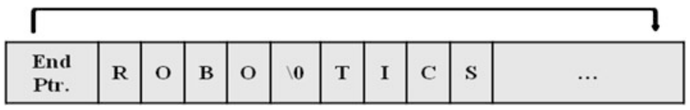


Fig. 3.2 Pointer delimited string representation

Null-Termination Errors

If a string is not properly null-terminated, string operations, such as strlen() and strcpy() will give incorrect results.

String truncation

String truncation issues are caused, to a certain extent, by efforts to prevent buffer overflows. It is often recommended by security experts, to use functions that limit the number of bytes copied. Example, use strncpy() in place of strcpy() and snprintf() in place of sprintf(). While these safer functions may be effective in mitigating buffer overflows, they often result in truncation of strings.

3.1.3 Why Are Strings in C the Way They Are?

It has been known for a long time [2], that for safe string handling, one can use an array and put a pointer at the beginning of the array which points to the end of the array as shown in the Fig. 3.2.

This would solve many of the issues mentioned in Sect. 3.1.2. This would also prevent string truncation due to inclusion of null character in the middle of the string. Furthermore, by enforcing bounds checking, it can be made much harder to overwrite adjacent chunks of memory.

This design was, however, never adopted, because, on early 1970 UNIX machines, memory was a scarce commodity. A null character (ASCII 0) is one byte, while a pointer needed at least 2 bytes. This made each null-terminated string a byte shorter than a pointer delimited string, saving lots of memory. The null-terminated string continues to be used even today, and we continue to suffer from the associated problems.

3.2 Safe String Libraries

String related issues have been recognized as one of the main sources of vulnerabilities in C language programs. Over the years, several safe string libraries have been written in an attempt to prevent these vulnerabilities.

Safe string libraries can be categorized as static or dynamic, depending on how they allocate space.
Static String Libraries: These libraries allocate fixed-sized strings. This means that once the character array associated with the string is filled, it is impossible to add excess data. Examples of static safe libraries are Strsafe from Microsoft [3], ISO/IEC TR 24731 Specification Part 1 for Bounds Checking Library Functions [4]. Since, in the case of static safe string libraries, excess data is discarded, data loss is a possiblility. Hence, the resulting string needs to be fully validated.
Dynamic String Libraries: In the dynamic approach, strings are resized as required. The dynamic approach scales better and does not discard excess data. The major disadvantage is that if inputs are not limited, they can exhaust memory and be used in denial-of-service (DoS) attacks. Examples of dynamic string libraries include Managed String Library [5] from Robert Seacord of SEI, CERT, CString Library [6] from Microsoft, SafeStr [7] from Matt Messier and John Viega and ISO/IEC TR 24731 Specification Part 2 for dynamic allocation library functions.

3.3 Related Work

A high-level comparison of a few safe string libraries was carried out by the creators of BString library [8]. But the study is not complete and precise in as much as they have not used any metrics.

Our study comprises a detailed and metric-based comparative analysis of safe string libraries. Such a study is not available in the current literature.

3.4 Purpose of Work

As described in a previous section, several safe string libraries—static and dynamic—have been created. However, how does one determine which library to use in a given security critical situation. In current literature, there is no metric-based approach for selecting or recommending the usage of safe string libraries. The purpose of this work is to develop a framework for comparative analysis of safe string libraries. This framework can then be used as a sound basis for recommending the usage of a composition of specific safe string libraries. This work encompasses the following activities:

1. Selection of static/dynamic string libraries for comparison.
2. Selection of parameters of interest on which to carry out comparison of safe string libraries.
3. Creation of a test-suite for each library under consideration.
4. Devising metrics for measuring the parameters.

5. Creating performance test cases to measure the performance of selected string operations.
6. Comparing the performance of each library using percentile method.
7. Executing the test cases and documenting the results.
8. Metrics calculation for each library.
9. Performing the comparative analysis of static and dynamic safe string libraries.

3.5 Selection of Libraries

From a list of about 15 safe string libraries, after detailed literature survey and experimentation, several libraries have not been selected for analysis for one of the following reasons:

1. Unable to download.
2. Unable to compile.
3. No longer maintained for the latest OSs.
4. Extremely poor usability.

Finally three static libraries and five dynamic libraries have been selected to undergo the analysis process. They are listed here as follows:

Static Safe String Libraries

1. Secure CRT Library for Windows.
2. MS StrSafe for Windows.
3. LibSafeC for Linux.

Dynamic Safe String Libraries

1. Managed String Library for Windows.
2. SafeStr for Linux and Windows.
3. MFC CString for Windows.
4. Better String Library for Windows and Linux.
5. Simple Dynamic String (SDS) for Linux.

These libraries have been compiled and tested under the following:

- Windows 7 Professional OS with Visual C++ 2010 Professional edition.
- Redhat Linux OS with gcc Ver 4.7.

3.6 Selection of Parameters of Interest

The parameters selected for comparison of safe string libraries are as follows:

1. Functional coverage.
2. Bounds protection.

3. Performance.
4. Portability.

3.7 Creation of Test Suites

A reference list of 22 basic functionalities related to strings, as listed below, has been created. Under each functionality, there is a reference list of possible good and bad test cases.

1. Allocation.
2. Assignment.
3. String copy.
4. String concatenation.
5. Character extraction.
6. String comparison.
7. Inserting a character in a string.
8. Replacing a character in from string with another character.
9. Deleting a character from a string.
10. String duplication.
11. String length.
12. Joining multiple strings into one.
13. String splitting.
14. Searching.
15. Reading/scanning from stdin.
16. Reading/scanning from file.
17. Writing to a file.
18. Writing to console.
19. Conversion of string to int/long (signed/unsigned).
20. Conversion of int/long (signed/unsigned) to string.
21. Conversion of string to float/double (signed/unsigned).
22. Conversion of float/double (signed/unsigned) to string.

A test-suite has been created for each library under consideration, with test cases for whichever functionalities and their variants are available in each library.

3.8 Devising Metrics for Safe String Libraries

Metrics have been devised for the following parameters:

1. Functional coverage.
2. Bounds protection.
3. Performance.

Table 3.1 Calculation of G_i and W_i

Reference test cases for functionality F_i	Availability of test cases for the library under consideration
Good test case1	1
Good test case2	1
Good test case3	0
Bad test case1	1

Table 3.2 Rule table

Impact of running bad test case	Score
Does not allow any harm	1
Allows but returns error message	0.5
Allows and does not return error message	0

3.8.1 Metric for Functional Coverage

For measuring the functional coverage of a library, only the good test cases are taken into account. The weightage W_i, given to a library for a particular functionality F_i, is calculated as shown below:

$$G_i = 1/(\text{Total No of Good Test Cases in } F_i)$$
$$W_i = G_i * \sum(\text{Availability of Good Test Case Functionality}(0/1))$$

The usage of this metric is explained through the example given in the Table 3.1. If, for a particular functionality F_i, there are 3 reference good test cases, then G_i works out to be 1/3. Since for the selected library, only 2 good test cases are available, W_i for that library becomes 2/3.

The functional coverage (FC) of the library is calculated by averaging the weights from W_1 to W_{22}.

$$\text{FC} = (1/22) * \sum W_i$$

3.8.2 Metric for Bounds Protection

For calculation of bound protection metric, only the bad test cases are taken into consideration. Scoring of the bad test cases has been done based on the rules given in Table 3.2.

If a bad test case does not allow any harm to be done, it gets a score of 1. If a bad test case allows harm to be done, but returns an error message, it gets a score of 0.5. If a bad test case allows harm to be done, but does not even return an error message, it gets a score of 0.

Table 3.3 Scoring table

Activity	Does not allow	Allows but returns some error message	Allows and does not return error message
Copying a fixed string to a destination string without allocation of memory to the destination string	1	0.5	0
Copying a fixed string to a null string	1	0.5	0
Copying input from a file to a destination string of smaller size	1	0.5	0
Copying a fixed string to a destination string of smaller size	1	0.5	0
Concatenating a fixed string to a destination string without allocation of memory to the destination string	1	0.5	0
Concatenating a fixed string to a null string	1	0.5	0
Concatenating input from a file to a destination string of smaller size	1	0.5	0
Concatenating a fixed string to a destination string of smaller size	1	0.5	0
Insertion of a character below lower bounds	1	0.5	0
Insertion of a character above upper bounds	1	0.5	0
Reading a string from stdin of length greater than memory allocated to variable	1	0.5	0
Reading a string from stdin of length equal to memory allocated to variable	1	0.5	0
Converting a NULL string pointer to int/long (signed/unsigned)	1	0.5	0
Converting a large int/long value and storing to a string whose size is less than the resultant string	1	0.5	0
Converting a int/long value to NULL pointer	1	0.5	0

Based on these rules, a scoring table has been created for all the bad test cases in the reference list of functionalities. A partial scoring table is shown in Table 3.3.

The bounds protection metric (BPM) is calculated as follows:

$$P = \sum (\text{Availability of Functionality} * \text{Score obtained})$$
$$\text{BPM} = 1/(\text{No of Available Functions}) * P$$

For example, Table 3.4 shows the scores obtained for the concatenation function 'bconcat' of BString library. The concatenation operation has 5 bad test cases. The test cases have been executed and the results have been captured in a file. The results have been analyzed and the scores have been recorded for each of the test cases.

Table 3.4 Bound protection metric (BPM)—Bstrlib

Activity (negative test cases)	Result	Avail. of Func. A	Score obtained S	A * S
Concatenate a fixed string to a destination string without allocation of memory to the destination string	Automatically extends the memory to hold the string	1	1	1
Concatenate a fixed string to a null string	Throws error and the string is not altered	1	1	1
Concatenate a null string to a fixed string	Throws error and the string is not altered	1	1	1
Concatenate input from a file to a destination string of smaller size	Automatically extends the memory to hold the string	1	1	1
Concatenate a fixed string to a destination string of smaller size	Automatically extends the memory to hold the string	1	1	1

3.8.3 Performance Percentile

Test cases have been created for comparing the performance of the following operations of each selected library:

1. copy
2. concatenation
3. search.

The following steps have been carried out to calculate the performance percentile:

1. The average speed of the above selected functionalities for each library has been captured in number of instructions per second over 3 consecutive runs.
2. Setting the library with the fastest performance in a particular functionality as scoring 100%, the remaining libraries have been assigned a percentile for that functionality accordingly. The results are shown in Table 3.5. The results in Table 3.5 are shown graphically in Fig. 3.3.
3. The performance percentile (PP) of each library has been calculated by averaging out the values obtained for the 3 functionalities for each library.

Table 3.5 Performance percentile of selected functionalities

Library/Operations	Std. C Library	Secure CRT	Bstring	Safestr	Managed string	LibSafeC	SDS	StrSafe	CString	CBString	Std:: String
Copy	*100*	30	70	~0	40	20	60	50	10	90	80
Concatenation	40	10	*100*	30	~0	60	70	80	20	90	50
Search	90	*100*	70	30	20	80	~0	~0	40	60	50

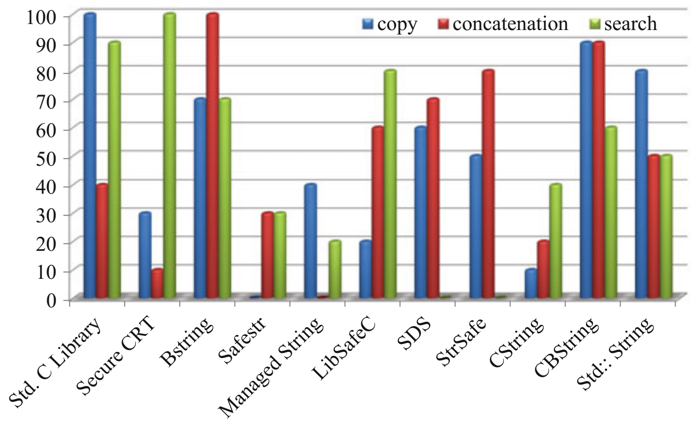


Fig. 3.3 Performance percentile of selected functionalities

3.9 Results

In this section, we present the actual results on the functional coverage, bounds protection, performance and portability of a list of static as well as dynamic libraries and compare these values with those of the standard C library.

3.9.1 Static Safe String Libraries

(1) Secure CRT Library for Windows

Secure CRT library [9] has been created by Microsoft based on the ISO/IEC TR 24731 specification, which defines less error-prone versions of C standard functions. Example—strcpy_s() replaces strcpy().

Advantages

1. It provides parameter validation
   (a) Checks for null values passed to the functions.
   (b) Checks enumerated values for validity.
   (c) Checks that integral values are in valid ranges.
2. The buffer size needs to be passed to any function that writes to a buffer.
3. It ensures that strings are properly null-terminated.
4. It provides enhanced error reporting.
5. It provides format string syntax checking.

Actual Results

- Bound Protection Metric = 88.89.
- Functional Coverage = 79.55.
- Performance Percentile = 70.

Disadvantages

1. This library is still based solely on char * buffers.
2. Buffer length synchronization needs to be carried out manually.
3. It does not enhance the functionality of the std. C library.

(2) <u>StrSafe Library from Microsoft SDK</u>

The functions in StrSafe library carry out additional processing for proper buffer handling.

Advantages

1. The size of the destination buffer is passed as a parameter to the functions. This helps to prevent buffer overwrites.
2. Null-termination of strings is guaranteed.

Actual Results

- Bound Protection Metric = 38.46.
- Functional Coverage = 18.18.
- Performance Percentile = 30.

Disadvantages

1. As functional coverage is very poor, this library alone cannot be used in a program. For functionalities such as allocation of memory, character extraction, searching, comparison, and replacing, normal std. C library functions are to be used.
2. BPM and Performance Percentile of this library are also poor.

(3) <u>LibSafeC for Linux</u>

This library [10] implements a subset of the functions defined in the ISO/IEC TR 24731 specification.

Advantages

1. Null-termination of strings is guaranteed.
2. Fewer changes are needed in existing programs.
3. Compile-time checking is supported.
4. Enhanced error reporting is provided.
5. Re-entrant code is supported.

Actual Results

- Bound Protection Metric = 88.89.
- Functional Coverage = 61.36.
- Performance Percentile = 60.

3.9.2 Dynamic Safe String Libraries

(1) Managed String Library (Windows and Linux)

Managed string library (MSL) for C has been created by Robert Seacord at CERT, CMU. MSL introduces a new type which is called a 'managed string'.

Advantages

1. It allows the user to define a set of 'safe' characters. This can be used for data sanitization purposes.
2. The user can define the default maximum size of a string. This helps in avoiding denial-of-service attacks, in case an attacker tries to over allocate memory.
3. It provides enhanced error reporting.
4. It succeeds/fails loudly by raising a runtime-constraint violation whenever memory cannot be allocated.

Actual Results

- Bound Protection Metric = 89.74.
- Functional Coverage = 61.36.
- Performance Percentile = 20.

Disadvantages

1. The fprintf_m() and sprintf_m() functions currently lack functionality to print out arguments using the double or float specifiers.
2. This library is poorly defined in multi-threading environments.

(2) SafeStr for Windows and Linux

SafeStr Library has been written by Matt Messier and John Viega.

Advantages

1. Strings are always null-terminated.
2. It provides extensive functionality. Example- split, join.
3. It provides enhanced error handling.

Actual Results

- Bound Protection Metric = 90.74.
- Functional Coverage = 81.82.
- Performance Percentile = 20.

Disadvantages

1. The functions, which interact with stdio/file, are not working as desired. Example- safestr_readline(), safestr_getpassword(), safestr_fprintf().
2. The function safestr_ncopy() or functions which use the flag SAFESTR_COPY_LIMIT are not working as desired.
3. It depends on XXL library [11] for error handling and reporting.

(3) MFC CString Class of Visual C++

CString Class is provided by Microsoft to extend the functionality provided by the C++ runtime library.

Advantages

1. CString class provides constructors and operators for constructing, assigning, and comparing CStrings and std. C++ string data types.
2. CString objects behave like built-in primitive types and simple classes.
3. It supports unicode characters.
4. A CString object throws an exception, when it runs out of memory.

Actual Results

- Bound Protection Metric = 98.11.
- Functional Coverage = 81.82.
- Performance Percentile = 23.

Disadvantage

1. CString library is not portable and is supported only on Windows platform.

(4) Better String Library for Windows and Linux (Bstring and CBString)

Bstring library [8] has been developed by Paul Hsieh. A Btring is basically a header which wraps a pointer to a char buffer.

Advantages

1. It uses segment arithmetic rather than just having pointer arithmetic.
2. It provides enhanced error reporting.
3. Even when a function returns an error, the underlying string is not modified in any way.
4. It provides a mechanism which helps to enforce project safety rules.
5. A C++ wrapper has been created to enable bstring functionality for C++.

Actual Results

- Bound Protection Metric = 98.18.
- Functional Coverage = 81.82.
- Performance Percentile = 90.

Disadvantages

1. A bstring needs to be explicitly destroyed, else it may lead to memory leak.
2. There is no unicode/wide character support in bstrlib.

(5) Simple Dynamic Strings (SDS)

Simple dynamic strings [12] was created by Salvatore Sanfilippo to augment the limited std. C library string handling functionalities by adding heap allocated strings.

Advantages

1. SDS strings can be directly passed to std. C library functions, Example—printf ("%s\n", sds_string);
2. Single allocation has better cache locality.

Actual Results

- Bound Protection Metric = 62.07.
- Functional Coverage = 61.36.
- Performance Percentile = 40.

Disadvantage

1. If an SDS string is shared in different places in the program, all the references need to be modified when you modify the string.

3.10 Conclusion

Table 3.6 shows the comparison of string libraries based on

- Bound Protection Metric.
- Functional Coverage.
- Performance Percentile.
- Portability.
- Dependencies.
- Unicode support.
- Development status.

The results for BPM, FC, and PP are graphically captured in Fig. 3.4.

Recommendations for usage

1. As evident from Table 3.6 and Fig. 3.4, BString is the better string library, as its name describes. BString and its C++ wrapper CBString are scoring high on bound protection, functional coverage and performance. This library is available as open source, it is portable, and meets almost every requirement a programmer needs. So we recommend BString as a better choice for a string library for any kind of usage.

Table 3.6 Comparison of string libraries

Library/Properties	Static/Dynamic	BPM	Performance percentile	Functional coverage	Portable	Open source	C/C++ Lib	Interoperability w/C lib.	Dependencies	Unicode support	Development status
Std. C library	S	0.00	100	79.55	Yes	Yes	C	Yes	None	Yes	Stable
Secure CRT	S	88.89	70	79.55	Yes	Yes	C	Yes	ANSI C	Yes	Stable
LibSafeC	S	88.89	60	61.36	No	Yes	C	Yes	None	Yes	Stable
MS StrSafe	S	38.46	30	18.18	No	No	C	Yes	MS SDK	Yes	Stable
Bstring	D	98.18	90	81.82	Yes	Yes	C	Yes	None	No	Stable
Safestr	D	90.74	20	81.82	Yes	Yes	C	Yes	XXL	No	Not maint.
Managed String	D	89.74	20	61.36	Yes	Yes	C	Yes	ANSI C	Yes	Not maint.
SDS	D	62.07	40	61.36	Yes	Yes	C	Yes	None	No	Stable
MFC CString	D	98.11	23	81.82	No	No	C++	Yes	MFC	Yes	Stable
CBString	D	98.18	80	81.82	Yes	Yes	C++	Yes	None	No	Stable
Std::String	D	48.78	50	68.18	Yes	Yes	C++	Yes	C++/STL	Yes	Stable

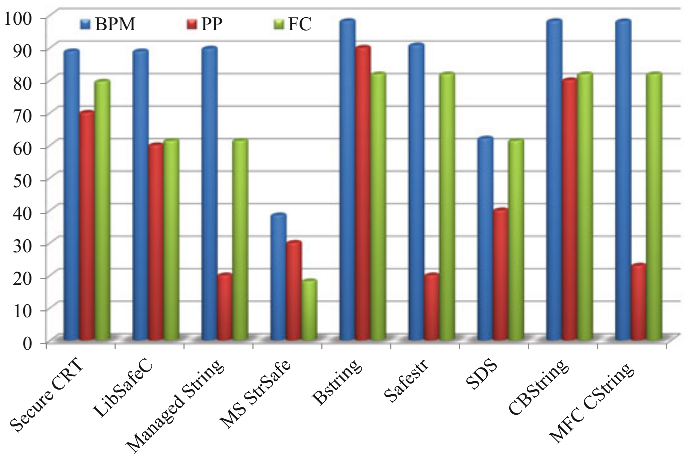


Fig. 3.4 Comparison of safe string libraries

2. The libraries MFC CString, SafeStr, Managed String, LibSafeC, and Secure CRT are also scoring high in BPM and functionality. If you are considering safety as the primary criteria and are not too concerned about performance, these libraries also are good options. In fact, CString is used extensively in Windows programming, but it is not portable.

References

1. Seacord R (2005) Managed string library for C. http://www.drdobbs.com/managed-string-library-for-c/184402023. 01 Oct 2005
2. Stross C. Where we went wrong. www.antipope.org
3. Microsoft Help Library, MSDN/help/1-5036/ms.help?method=page&id=-1&topicversion=100&topiclocale=
EN-US&SQM=1&product=VS&productVersion=100&locale=EN-US
4. ISO/IEC TR 24731—1:2007, Information technology–Programming languages, their environments and system software interfaces-Extensions to the C library—Part 1: Bounds-checking interfaces
5. Burch H, Long F, Seacord R (2006) Specifications for managed strings
6. MSDN Library for Visual Studio 2012—ENU, Microsoft Corporation 2012
7. Messier, Veiga. Safe C String Library v1.0.3, 30 Jan 2005. http://www.zork.org/safestr/
8. Hsieh P (2015) The better string library. http://bstring.sourceforge.net
9. http://msdn.microsoft.com
10. Thomas N (2001) Lock it down: use Libsafe to secure Linux from buffer overflows. http://www.techrepublic.com
11. Messier, Veiga. XXL v1.0.1, 30 Jan 2005
12. Salvatore S (2006) Simple dynamic strings-readme

Chapter 4
Dynamic Constrained Objects for Vehicular Network Modeling

Jinesh M. Kannimoola, Bharat Jayaraman and Krishnashree Achuthan

Abstract We present a paradigm called dynamic constrained objects for a declarative approach to modeling complex systems. In the basic paradigm of constrained objects, the structure of a complex system is specified through objects (as in object-oriented languages), while the behavior of a complex system is specified declaratively through constraints (as in constraint languages). The emergent behavior of such a complex system is deduced through a process of constraint satisfaction. Our focus in this paper is on systems whose states change with time. Such time-varying behaviors are fundamental in many domains, especially in mission and safety-critical applications. We present an extension of constrained objects with special metric temporal operators over time-series data, and we discuss their properties. We refer to the resulting paradigm as dynamic constrained objects and we illustrate their use for vehicular network modeling. Here, the network of roads and the roadside infrastructure are specified through objects, and the movement of vehicles and associated safety and liveness conditions are specified through time-series variables and metric temporal operators. The paper presents a language called DCOB, for dynamic constrained objects, and examples of its use for vehicular network modeling.

Keywords Constraint object · Dynamic systems · Temporal logic
Constraint satisfaction

J.M. Kannimoola (✉) · K. Achuthan
Amrita Center for Cybersecurity System and Networks, Amrita School of Engineering, Amrita Vishwa Vidyapeetham, Amrita University, Amritapuri, India
e-mail: jinesh@am.amrita.edu

K. Achuthan
e-mail: krishna@amrita.edu

B. Jayaraman
University at Buffalo, Buffalo, USA
e-mail: bharat@buffalo.edu

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_4

4.1 Introduction

The advent of the Internet of Things and autonomous vehicles has transformed conventional Vehicular Ad hoc Networks (VANETs) into an Internet of Vehicles (IoV). IoV integrates humans, vehicles, and roadside infrastructure in order to provide a safe and smooth flow of traffic [1]. Formal modeling and reasoning are critically needed in IoV due to its high complexity and requirement for safety, and this has been a topic of considerable recent interest [2–6]. This paper introduces a natural modeling approach for a vehicular network in which the hierarchic structure of the system is specified in an object-oriented way and the behavior of the system is specified in a constraint-based way. Our approach was inspired by previous work on constrained objects [7] where a physical system is abstracted as an assembly of objects whose attributes that are subject to behavioral laws expressed as declarative constraints. The resultant behavior of complex systems achieved by a process of constraint satisfaction over the attributes of objects.

This paper extends the paradigm of constrained objects with a temporal component which caters to the time-varying aspect of vehicular systems. Time-series variables and metric temporal operators are added to the paradigm of constrained objects and their use is illustrated through several examples. Time-series variables record the state changes as time progresses, while the metric temporal operators specify the overall dynamic behavior of the system in form of temporal constraints that should hold over the system lifetime. We present a language called DCOB (for dynamic constrained objects) and show how it can be used to model various entities in an IoV, including vehicles, roads. The temporal and non-temporal constraints on objects ensure a safe and smooth flow of traffic.

The remainder of this paper is structured as follows: Sect. 4.2 briefly describes the related work in this field; Sect. 4.3 gives the syntax of the basic constrained object language and Sect. 4.4 provides the detailed description of ‘dynamic’ extension of COB. Finally, Sect. 4.5 gives the application of Dynamic COB in the vehicular system modeling.

4.2 Related Work

Researchers have proposed different logical reasoning approaches to capture the unique properties of vehicular systems with a focus on the collision-free movement of vehicles on the road. The primary objective of California PATH project [2, 6] was the development of a fully automated highway system (AHS). In AHS, the highway is split into different platoons of closely spaced vehicles. It defines three essential maneuvers, namely join, split, and change, which ascertain the safe platooning of vehicles. The network is split into different controller layers to provide modularity in operation. Safety is defined for collision avoidance and the controllers verified by the proof rules. Lygeros et al. [8] presented a safe controller design using a hybrid automaton—a very appropriate model to handle systems with continuous and discrete behavior. The onboard and roadside controllers keep the vehicle platoons in a safe distance by offering maximum throughput in the highway system. Damm et al. [9] simplified the complexity of the collision avoidance system using a criticality function, which is a quadratic function on position and velocity. The layered approach in the system is modeled using the parallel composition of a hybrid automaton. Vehicle under a certain critical threshold level can perform collision-free maneuvers.

Multilane Spatial Logic (MLSL) [10] is another effort to address safety in multilane maneuvers. Here, each vehicle maintains a local view of roads and nearby vehicles. A vehicle always reserves some area in a lane and claims area in another lane for a lane change. The reservation, claim, position, velocity, and acceleration are formally combined with the notion of traffic snapshot. Linker et al. [3] presents a formal diagrammatic language called traffic diagram along with an extended 95 MLSL. Quantified differential dynamic logic (QDL) with quantified hybrid programs (QHP) [5] is an appropriate way to model a distributed hybrid system. QHP is a dynamic logic program to support quantified differential equation system for distributed hybrid dynamics. The safety of control system is proved in a highly modular way. Controllers are the combination of dynamics and controls, which support different maneuvers. The controller's safety is verified using a proof system, which ensures that the vehicle is always at a safe distance from its leading vehicle. Hafner [11] focuses on a collision avoidance at the intersection based on the capture set calculation. A set of bad states is defined over capture set based on car dynamics. If a vehicle hits the boundary of capture set, then the control actions are initiated by the vehicle. Each vehicle estimates its trajectory and other vehicle trajectories using Kalman filter. Most of this approach followed an imperative modeling method to solve the core issues in the vehicular system. The declarative modeling approach adopted in our work is the fundamental difference compared to existing methods. We are combining the concepts of objects and classes together with declarative constraints to provide a more intuitive way of modeling vehicular systems.

Constraint satisfaction problems (CSPs) have been extensively studied in scheduling and planning in the artificial intelligence domain [12]. Hernandez et al. [13] show the practical use of constraint satisfaction and optimization and Kilby et al. [14] formulate vehicle routing as a constraint satisfaction problem. The constraints are employed to minimize the shipping cost by restricting the time of each route and the capacity of the vehicle based on the demand of each client. The language Kaleidoscope [15] and ThingLab [16] are early attempts at the integration of constraint satisfaction with object orientation. Kaleidoscope has support for constraint hierarchies, multi-methods, and constraint constructors. It accepts user-defined constraints, and the compiler simplifies them to primitive constraints that can be solved by a primitive solver. ThingLab [16] provides an interactive graphical simulation environment for physics and geometry. ThingLab is an object-oriented language with constraints for specifying the relations between parts of an object. A constraint is defined by a rule and a set of methods for satisfying the constraint.

The COB language provides a systematic way to represent a complex engineering system using simple, quantified, preference, and conditional constraints. The thesis [17] highlights the expressive power of COB for modeling metabolic pathways and systems biology. The more detailed description of COB paradigm is given in the next section.

4.3 COB: A Constrained Object Language

The constrained object (COB) [7] program defines a sequence of classes, each of which contains a set of attributes, constraints, predicates, and constructors. Each constrained object is an instance of some class. The grammar below defines the current structure of a COB program.

```
          program  ::= class_definition +
 class_definition  ::= [ abstract ] class class_id [ extends class_id ] { body }
             body  ::= [ attributes attributes ]
                       [ constraints constraints ]
                       [ predicates pred_clauses ]
                       [ constructors constructor_clause ]
       attributes  ::=  decl ; [ decl ; ]+
             decl  ::=  type id_list
             type  ::=  primitive_type_id | class_id | type[ ]
primitive_type_id  ::=  real | int | bool | char | string
          id_list  ::=  attribute_id [ , attribute_id ]+
```

An attribute is a typed identifier, which supports both primitive and user-defined types defined by a class. Constraints specify the relation between attributes. The grammar below defines the basic structure of COB constraints.

```
           constraints  ::=  constraint ; [ constraint ;]+
            constraint  ::=  creational_constraint | quantified_constraint
                             | simple_constraint
 creational_constraint  ::=  attribute = new class_id(terms )
 quantified_constraint  ::=  forall var in enum : constraint
                             | exists var in enum : constraint
     simple_constraint  ::=  conditional_constraint | constraint_atom
conditional_constraint  ::=  literals → constraint_atom
       constraint_atom  ::=  term relop term | constraint_predicate_id(terms )
                 relop  ::=  = | != | > | < | >= | <=
                  term  ::=  constant | var | attribute | (term ) | func_id(terms )
                             | sum var in enum : term
                             | prod var in enum : term
                             | min var in enum : term
                             | max var in enum : term
```

The COB language supports simple, conditional, quantified, and aggregate constraints. Simple constraints can be a constraint atom or a conditional constraint. A constraint atom relates attributes using relational operators while conditional constraints are predicated on a conjunction of literals. Creational constraints defined the object creation constraint and quantified constraints defined by the enumeration of attributes.

We give a simple example of temperature diffusion in a heat plate to illustrate the use of some of the constructs. This problem can be modeled mathematically by using Laplace's equations in two dimensions, but the COB program below (for a 10×10 heat plate) captures these equations by setting up a number of constraints: The temperature of any of the interior points is the average of the temperature of it's neighboring four points. The constructor initializes the perimeter points of the 10×10 plate. In the example below, we have 64 linear equations with 64 unknown variables. Solving these constraints gives the temperature of the internal points.

```
class heatplate {
  attributes
    real [][] Plate;
  constraints
    forall I in 2..9:
      forall J in 2..9:
        4 * Plate[I,J] =
          (Plate[I-1,J] + Plate[I+1,J] + Plate[I,J-1] + Plate[I,J+1]);
  constructors heatplate(A,B,C,D) {
    forall M in 1..10: Plate[1,M] = A;
    forall N in 1..10: Plate[10,N] = B;
    forall K in 2..9: Plate[K,1] = C;
    forall L in 2..9: Plate[L,10] = D;
  }
}
```

4.4 Dynamic COB with Metric Temporal Operators

The paradigm of Dynamic COB extends the basic constrained object paradigm with constructs for specifying time-varying properties. DCOB permits the use of a `series` variable which is declared as follows:

`series` variable [initialization]

A series variable takes on an unbounded sequence of values over time. In the paradigm of dynamic constrained objects, the concept of time as a metric unit is adopted, and a built-in variable `Time` represents the current time and this variable is automatically incremented by one unit to record the passage of time. The user can adopt any other discrete granularity for time by multiplying with a suitable scaling factor, e.g., `MyTime = 0.01 * Time`.

The temporal constraints are defined in terms of past and future values of the series variable. For every series variable v, the expression'v and v' refers to the immediate previous and next values of v, respectively. These operators can be juxtaposed to refer to successive values of v in the past or future. The past values of v at any point in time are referred to by 'v, ''v, '''v, …, and the future values of v at any point in time are referred to by v', v'', v''', ….

For example, the declaration below introduces a series variable `Position` for the positions of a moving entity, along with a constraint that its position increases by one at each time step:

```
series int Position;
Position = 'Position + 1;
```

The value of a series variable v at some specific point in time can be accessed by v<i>. This notation is often used to give the initial values for series variables. For example, the fibonacci series can be defined as follows:

```
series int fib;
fib<0> = 1;
fib<1> = 1;
fib = 'fib + ''fib;
```

The informal semantics of a DCOB program is that `Time` is initialized to 0 and the constraints in all objects are enforced at this time. For every series variable v, its value at v<0> is taken as the value of v. Then `Time` is set to 1 and the constraints of all objects are evaluated assuming that for every series variable v, its value at v<1> is taken as the value of v. This computation continues indefinitely but a user has the option to terminate the computation when `Time` reaches a certain limit value or if it satisfies a certain condition.

We introduce two metric temporal operators, `F` and `G`, as follows:

1. `F` p states that constraint p must hold at some time point in the future;
2. `F`<t> p states that constraint p must hold exactly after t units of time;
3. `F`<t1, t2> states the constraint p must hold sometime starting after t1 units of time but before t2 units of time.
4. `G` p states that constraint p must hold at all time points in the future;
5. `G`< t> p states that constraint p must hold at all time points after t units of time; and
6. `G`<t1, t2> states the constraint p must hold at all time points starting after t1 units of time but before t2 units of time.

The following example provides a simple illustration of a traffic light. The constraints state that the traffic light starts with a red light, stays at red for 120 time units, then changes to yellow for 20 time units, then changes to green for 180 time units, then changes to yellow for 40 time units, and changes back to red. This behavior is repeated indefinitely.

```
enum Color = {red, yellow, green}
class trafficlight {
   attributes
       series Color c;
   constraints
      c = red & not ('c = red) -->
                 G<0,120> c = red &  F<120> c = yellow;
      c = green & not ('c = green) -->
                 G<0,180> c = green &  F<180> c = yellow;
      c = yellow &  'c = red -->
                 G<0,60>c = yellow &  F<60> c = green;
      c = yellow & 'c = green -->
                 G<0,60> c = yellow &  F<60> c = red;
 constructor
     trafficlight()  { c<0> = red; }
 }
```

The metric temporal operators with the series variable enables a concise and clear specification of temporal constraints in a real-time setting. We explain the first constraint in more detail:

```
c = red & not ('c = red) --> G<0,120> c=red & F<120> c=yellow;
```

When Time = 0 the value of c = red since this is the initial value of c as given in the constructor. At this time, the value of 'c is undefined (hence not equal to red), and the antecedent of the above constraint is satisfied. In the consequent of the constraint, the condition G<0,120> c = red states that c<0> = c<1> = c<2> = … = c<119> = red and the condition F<120> c = yellow states that c<120> = yellow. In a similar manner, the remaining constraints determine values for c at various time points. At Time = 320, the value of c again becomes red and the first constraint again applies. This pattern is repeated at intervals of 320 time units.

4.5 Vehicular Network Modeling

In this section, we further illustrate the application of DCOB for Vehicular Networking specification. The temporal constraint specifications in the DCOB framework allow one to formulate both the quantitative and the qualitative regulation of time for the safety and liveness of the transportation system. Roads and vehicles are basic objects in the vehicular system. Objects model the structure of a vehicular system while constraints specify the rules and desired behavior in a declarative manner. A road object represents a piece of a road in the system with a unique id, start, and end position and set of lane objects. Each lane has a specific capacity and the number of vehicles at each point of time represents the traffic on the road.

```
class lane{
    attributes
        int N,Capacity;
        series vehicle V[];
        series int Traffic;
    constraints
        %capacity constraints
        count(V,Traffic);
        T <= C;
    constructors lane(N1,C){
          N=N1; Capacity=C;
    }
}


class road {
    attributes
        int ID,Distance;
        string Start,End;
        lane L[];
    constructors road(ID1,D,S,E,L1){
        ID=ID1;
        Distance=D;  Start=S;
        End=E; L=L1;
    }
}
```

The capacity constraints in the `lane` class impose the traffic rules, which guarantees that the road traffic must be less than or equal to the capacity of the road. We have used a `count` predicate to obtain the number of vehicles in a lane at any point in time. Here, `Traffic` is defined as a series variable which maintains the traffic at every point in time.

The vehicle is subject to a set of movement as well as environmental constraints. The continuous changing properties, such as velocity, position, and acceleration are represented discretely in our model by the series variable. A vehicle occupies different roads over time and a vehicle's dynamics will change with the front vehicle. Therefore, we define road and front vehicle as the series variables in the class `vehicle`.

```
class vehicle {
    attributes
        int ID,Vmax,Amax,Dmin,Dmax;
        string Start,End;
        series real Vel,Acc,Pos;
        series road Road;
        series vehicle Front;
    constraints
        %movement constraints
        Acc <= Amax;Acc >= Dmin;
        Vel > 0;    Vel <= Vmax;
        'Pos-Pos=Vel;
        'Vel-Vel=Acc;
        %safety constraints
        Pos+(Vel*Vel)/(2*Dmin) <
              Front.Pos+(Front.Vel*Front.Vel)/(2*Front.Dmax);
        Acc=Dmax --> F<0,2>Vel=0 ;
        %flow constraints
        Front.'Vel='Vel & Front.'Vel<Front.Vel -->
                                     (F<1,3>Vel= Front.Vel) ;
        (Front.Pos - Pos) > 2*Safe --> Acc' >= Acc;
        %destination constraints
        F<>(Road.End = End);

    constructors vehicle(ID1,Vmax1,Amax1,Dmin1,Dmax1,F,R1,S1,E1){
        ID = ID1; Vmax = Vmax1;
        Amax = Amax1; Dmin = Dmin1;
        Dmax = Dmax1; Front = F;
        Road<1> = R1; Start = S; E = End;
    }
}
```

The safety constraints ensure the safe distance from the follow vehicle. The term ‘safe distance’ means that the following car can avoid collision with minimum breaking when the front vehicle stops with sudden deceleration. The break constraints are expressed in the conditional constraint with the metric temporal operator. It ensures that vehicle will stop within the next two seconds when the full break is applied. The flow constraints express the liveness of traffic stream without an immediate speed change. If a lead vehicle and follow vehicle move in the same lane with the same speed and the lead vehicle increases the speed, then the following vehicle must increase its speed same as the front car. The temporal constraints guarantee that gradual change within three seconds without sudden speed change. The destination constraints specify that the vehicle must arrive at its destination.

4.6 Conclusion and Future Work

Formal modeling is an important step toward the development of provably correct systems. In this paper, we presented a novel modeling language using the concept of dynamic constrained objects. The dynamic operators help to define the real-time properties of the system and make use of metric temporal operators analogous to the 'eventually' and 'always' operators of propositional temporal logics.

A prototype implementation of the DCOB language has been developed using SWI-Prolog, which supports constraint satisfaction, and the examples presented in this paper have been tested using this prototype implementation. Our aim is to develop a fully fledged specification language for vehicular systems that capture the unique dynamic and safety critical properties of these systems. In this paper, we illustrated use of these temporal operators in the specification of safety and liveness properties of the system.

Real vehicular systems contain hundreds of vehicular nodes and roads, and encompass complex notions such as path finding, dynamic route selection, maneuvers. We plan to integrate the concepts of preference and constraint hierarchies in DCOB in order to deal with complex safety and liveness properties of advanced vehicular systems. In modern vehicular system, roadside and other infrastructural units play an important role in the safe and effective movement of vehicles. The future model must include such infrastructure objects as well.

References

1. Fangchun Y, Shangguang W, Jinglin L, Zhihan L, Qibo S (2014) An overview of internet of vehicles. China Commun 11(10):1–15
2. Alvarez L (1996) Automated highway systems: safe platooning and traffic flow control. Ph.D. thesis, Department of Mechanical Engineering, University of California at Berkeley
3. Linker S (2015) Proofs for traffic safety: combining diagrams and logic. Ph.D. thesis, Oldenburg, Universitat Oldenburg, Diss.
4. Loos SM, Platzer A, Nistor L (2011) Adaptive cruise control: hybrid, distributed, and now formally verified. In: FM 2011: formal methods, Springer, Berlin, pp 42–56
5. Platzer A (2010) Quantified differential dynamic logic for distributed hybrid systems. In: Computer science logic. Springer, Berlin, pp 469–483
6. Varaiya P (1994) Models, simulation, and performance of fully automated highways. California Partners for Advanced Transit and Highways (PATH)
7. Jayaraman B, Tambay P (2000) Constrained objects for modeling complex systems. Springer, Berlin
8. Lygeros J, Godbole DN, Sastry S (1998) Verified hybrid controllers for automated vehicles. IEEE Trans Autom Control 43(4):522–539
9. Damm W, Hungar H, Olderog ER (2006) Verification of cooperating traffic agents. Int J Control 79(05):395–421
10. Hilscher M, Linker S, Olderog ER, Ravn AP (2011) An abstract model for proving safety of multi-lane traffic manoeuvres. In: Formal methods and software engineering. Springer, Berlin, pp 404–419

11. Hafner MR, Cunningham D, Caminiti L, Del Vecchio D (2013) Cooperative collision avoidance at intersections: algorithms and experiments. IEEE Trans Intell Trans Syst 14 (3):1162–1175
12. Bartak R, Salido MA, Rossi F (2010) New trends in constraint satisfaction, planning, and scheduling: a survey. Knowl Eng Rev 25(03):249–279
13. Herńandez-Arauzo A, Puente J, Varela R, Sedano J (2015) Electric vehicle charging under power and balance constraints as dynamic scheduling. Comput Ind Eng 85:306–315
14. Kilby P, Prosser P, Shaw P (1999) Guided local search for the vehicle routing problem with time windows. In: Meta-heuristics, Springer, pp 473–486
15. Lopez G, Freeman-Benson B, Borning A (1994) Kaleidoscope: a constraint imperative programming language. In: Constraint programming. Springer, Berlin, pp 313–329
16. Van Hentenryck P, Simonis H, Dincbas M (1992) Constraint satisfaction using constraint logic programming. Artif Intell 58(1):113–159
17. Pushpendran M (2006) A constrained object approach to systems biology. ProQuest

Chapter 5
Adoption of Formal Methods in Software Safety Analysis

Ankita Srivastava and S.K. Goswami

Abstract Safety analysis of hardware component of computer-based system (CBS) is very much standardized with failure mode and effects analysis (FMEA) and fault tree analysis (FTA). However, safety analysis of software components is very abstract. As quantitative software reliability analysis is still a subject of academics, safety analysis is generally approached over qualitative study. Model-based system design has eased the study which support host-based simulation to establish the correctness of the linkage derived out of the manual effort. If the system size increases, it becomes difficult to capture all scenarios as well as correctness of the analysis. As completeness is very difficult to establish, this particular paper tried to establish correctness of the analysis using formal method. The main safety analysis is done through manual identification of software components and developing fault tree. Subsequently safety property is designed based on system requirement derived from system's intended function on fault condition. These safety properties are then subjected to formal proof engine to identify existence of any false path. If no such path is found, the analysis is proven to be correct.

Keywords Formal methods · Safety analysis · Fault tree analysis
Fault modeling · Model based design

5.1 Introduction

In order to develop any system, the design process must be based upon a sound model that captures important features of any system. If we design any system traditionally, there are chances that it is inconsistent, incomplete and unambiguous. Much depends on human expertise in traditional methods of system development.

A. Srivastava (✉) · S.K. Goswami
R&D(ES), Nuclear Power Corporation of India Ltd., Mumbai, India
e-mail: ankita@npcil.co.in

S.K. Goswami
e-mail: skgoswami@npcil.co.in

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_5

Use of Formal Methods requires mathematical notation to specify a system. It should have deterministic semantics which allows building of a clear mathematical model consisting of a set of data flow equations to build system design. While doing safety analysis of our system, we should know which faults can occur and in how many probable ways those particular components will fail. Hence, it is required to understand how to describe a formal model describing both the system behaviour and the fault behaviour.

5.2 Work

We followed a method of incorporating safety analysis techniques in a formal model used for design of our system. For this job, the formal language is chosen as SCADE which is based on Lustre for declarative behaviour and Esterel for imperative behaviour. The proof engine is chosen as Design Verifier which accepts SCADE language as input and subsequently applies model checking techniques of formal verification. System contained various modules which in turn had sub modules and the way these major modules are interconnected is also shown in Fig. 5.1. We generated safety properties from specification and verified if the properties are satisfied against the model. Formal Methods enables us to prove that a design is safe w.r.t. its requirements. This allows finding bugs very early in design cycle. Also, we can tell that a system is free from bugs using model checking method provided by DV.

We did fault tree analysis to identify exactly which events will lead to an identified undesirable event and then we will evaluate behaviour of the system in the

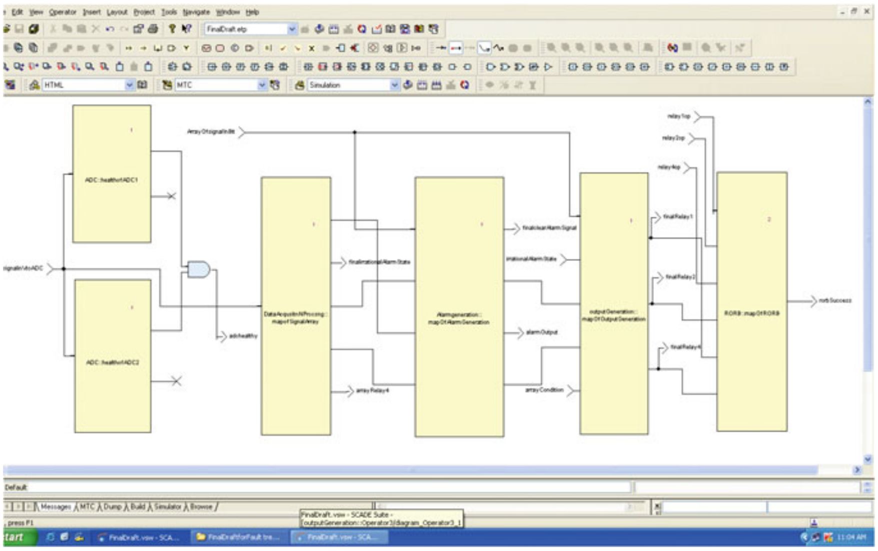

Edit View Operator Insert Layout Project Tools Navigate Window Help
FinalDraft.etp
HTML
MTC
Simulation
ADC::healthofADC1
ADC::healthofADC2
ArrayOfsignalInBit
signalInVitsADC
adchealthy
DataAcquisitnNProcsng::mapofSignalArray
finalirrationalAlarmState
arrayRelay4
Alarmgeneration::mapOfAlarmGeneration
finalclearAlarmSignal
irrationalAlarmState
alarmOutput
arrayCondition
outputGeneration::mapOfOutputGeneration
relay1op
relay2op
relay4op
finalRelay1
finalRelay2
finalRelay4
RORB::mapOfRORB
rorbSuccess
Default
Messages MTC Dump Build Simulator Browse
press F1
FinalDraft.vsw - SCADE Suite - [outputGeneration::Operator3/diagram_Operator3_1]
start
FinalDraft.vsw - SCA...
FinalDraftforFault tre...
FinalDraft.vsw - SCA...
11:04 AM

Fig. 5.1 Top view of model of our system

presence of those events. If those events cannot be eliminated, then we will make our system to go into a failsafe scenario in presence of those events. FTA will also eliminate many events from the fault tree. In this way, we will get to know behaviour of our system in the presence of a particular event or combination of events.

When system model is combined with the fault model, formal method can be used for automated support for behavioural safety analysis. We will get to know the system where a particular event will lead a system and its output. Based on the outcome, we will modify our system. Behaviour of the system and safety properties was specified. Formal verification technique was applied to prove that model satisfies all safety properties. The Fault modelling was done as it is required to apply formal verification tools to perform safety analysis. Validation of correctness of fault tree that was constructed was also done.

Further details are stated in steps as followed in the works.

Step 1. **Formalizing derived safety requirements**:
An observer operator was designed consisting of the complete system and the desired property. It was then verified that whether property is always valid throughout the system or is falsifiable somewhere (counterexamples will be generated). This whole exercise guarantees exhaustiveness.

Step 2. **Fault tree analysis**:
While doing safety analysis of our system we should know which faults can occur and in how many probable ways those particular components will fail. A fault tree model enables us to understand where the system can fail, how it can fail etc. We will list down all the possible single events or combinations of events which will lead to the top event. An example fault tree is shown in Fig. 5.2.

Step 3. **Fault modelling and formal proof**:
Various faults that we have listed out in fault tree will now be added in our system model. We will add failures in all the possible ways in which they can occur, i.e. all the basic events of the fault tree.
The procedure now is to consider faults which are likely to occur in the system and then improve the model to handle these faults and compute the result on the output(s) with each fault present. If system is behaving correctly even in the presence of the chosen set of faults, then the system is considered to be fault free.
We did analysis of failure of individual modules too.
Each fault was injected (i.e. modelled in our model), and we validated our safety properties once again (this time in the presence of such faults) and found out some faults were not affecting any of the safety property and some were. In this way, we evaluated our fault tree for correctness and validated it. On basis of this analysis, our system was rectified and made more fault tolerant. This shows that the system will be able to handle all these faults, i.e. either rectify them or take the system in a failsafe direction.
One of the examples of safety property associated with a fault:
If there is ADC fault, depending on the parameter either setback relay or trip relay will de-energize (Fig. 5.3).

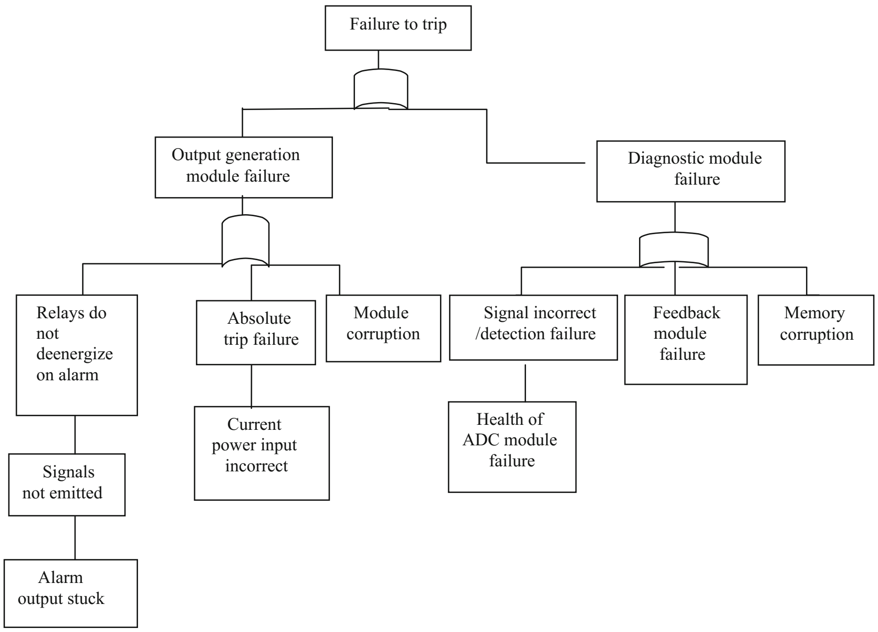
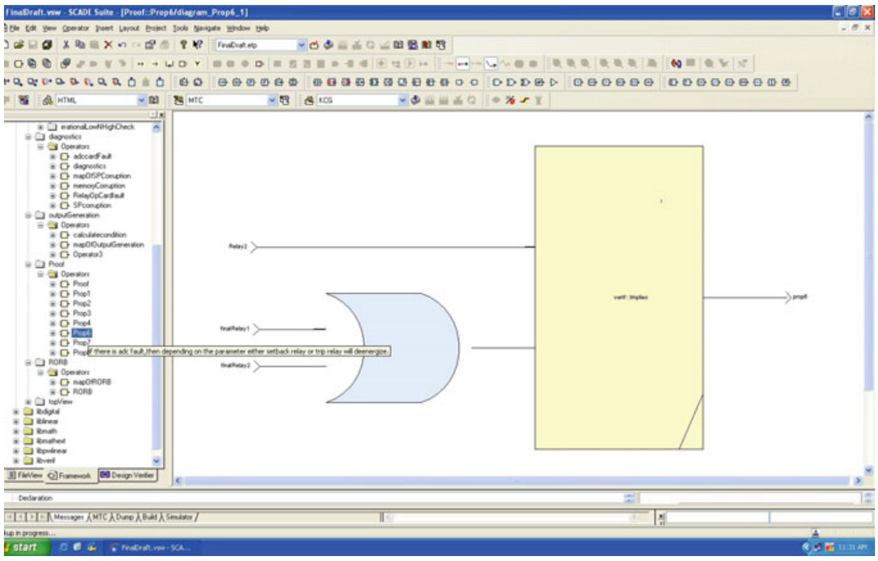

Failure to trip
Output generation module failure
Diagnostic module failure
Relays do not deenergize on alarm
Absolute trip failure
Module corruption
Signal incorrect /detection failure
Feedback module failure
Memory corruption
Current power input incorrect
Health of ADC module failure
Signals not emitted
Alarm output stuck

Fig. 5.2 Fault tree



If there is adc fault, then depending on the parameter either setback relay or trip relay will deenergize.

Fig. 5.3 Fault modelling and checking

5.3 Conclusion

In the absence of quantitative method of studying software reliability, performing safety analysis is very challenging. It becomes very difficult to demonstrate failure of particular module lead to which direction of system failure. However, the technique adopted here gave ample scope of identifying the system behaviour in case any particular component is either not available or fails in unsafe direction. Even then, when the system model increases in size it becomes extremely difficult by manual inspection on the correctness as well as adequacy of the analysis. Formal analysis-based safety property derived out of FTA greatly reduces the rigour of reviewing the safety analysis. Because of model driven design using formal language ensures very limited extra effort to assess the adequacy of the FTA. However, selection of formal language supported by model driven design and supporting proof engine greatly influences the outcome. In our case, SCADE and DV were used. However, DV like many other Model Checker is constrained by size of the design as well as use of real numbers.

Chapter 6
Model-Based Safety Validation for Embedded Real-Time Systems

Gracy Philip and Meenakshi D'Souza

Abstract Architecture Analysis and Design Language (AADL) is an emerging industry standard notation for modeling architecture of embedded real-time systems. AADL has capability to model normal and faulty behavior of the system. It can model error behavior, for all of its components, their interactions and propagations. This capability makes it suitable for modeling safety critical systems, where fault detection, isolation and re-configuration are of paramount importance. This paper proposes using AADL to do safety validation of an embedded real-time system. AADL is used as a basis to systematically derive requirements and parameters specific to safety validation at both platform level and application level. Behavior and error models corresponding to system level AADL models are used to derive the parameters. Such parameters can then be monitored at the system integration testing level automatically using a safety monitor. We present our ideas using a detailed case study involving safety validation of a automatic flight control system (AFCS) from its AADL model that includes behavior and error modeling.

Keywords Safety assurance · Architecture analysis and design language (AADL) · Error behavior · Flight control system (FCS)

6.1 Introduction

Safety assurance of embedded system is an involved task in which expertise in many disciplines has to match seamlessly. Functional hazard analysis and system safety analysis are safety analysis activities suggested by ARP-4761 [8] for safety assurance process for avionics software. These analysis activities are to be

G. Philip (✉)
CEMILAC, DRDO, Bangalore 5600037, India
e-mail: gracy.philip@cemilac.drdo.in

M. D'Souza
IIIT-Bangalore, 26-C, Electronics City, Bangalore 560100, India
e-mail: meenakshi@iiitb.ac.in

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_6

continuously evolved throughout the system development life cycle. As per [5], safety is not a software property alone, even when all critical functionalities are implemented in software. System need not be safe even when it is following all its requirements. It means there could be lot of inconsistent and incomplete requirements, leading to hazardous situations, which were not envisaged. So it is imperative that, we should have a systematic method of generating system models, which could be used for specifying safety requirements and to bring out safety evaluation criteria.

AADL [3] is a popular architecture description language used for modeling embedded software. AADL has features include both software and platform components, ideal for modeling architectures of embedded software and systems. EMV2 [2] and BLESS [4] are error and behavior annex, respectively, which enhance the capability of core AADL. AADL can be used for modeling embedded systems systematically and further, for generation of safety evaluation criteria.

There are two aspects of safety: one is as introduced by the safety critical functionality, failure to achieve, it will impact safety. We call this application-aware safety parameters. Second aspect is due to the vulnerabilities introduced by digital implementation in embedded systems environment, which were not relevant in earlier electromechanical environment. We call this platform-aware safety parameters. The second aspect also includes the intricacies of human–machine interface and complications in decision making, which are error prone. So, safety requirements are to be derived considering above two aspects.

In [6], we introduced a software monitor at the hardware–software integration test level, which monitors application-aware safety parameters and platform-aware safety parameters in real time, in parallel with requirements-based testing. Advantage of this environment is that the system under testing undergoes all normal and failure mode testing in this environment. These tests are executed in near, real-time environment representative of operational environment. The safety monitor could bring out major safety critical failures as per [6]. Selection of safety parameters is cumbersome activity and was manually done in [6].

This paper extends the safety validation method, where in safety parameters are generated systematically using system model generated using AADL. The embedded system is modeled using AADL, its error annex and behavior annex. Safety parameters are systematically generated from the model. We illustrate our approach using a case study involving flight control system.

The paper is organized as follows: Sect. 6.2 introduces AADL and describes a systematic approach to system modeling using AADL and its annexes. Section 6.3 describes our case study of a flight control system, its modeling using AADL and typical safety aspects. We recap the notion of safety monitor and the two aspects of safety that can be monitored in Sect. 6.4. Section 6.5 describes how to generate safety monitor parameters using AADL models and its annex models. Comparison with related work is presented in Sects. 6.6 and 6.7 concludes the paper.

6.2 Modeling a Safety Critical System in AADL

Avionics Architecture Description Language (AADL) [3] allows description of hardware and software components, networks, I/O devices, communications buses, etc. It can also model real-time requirements including scheduling, processing, and memory management. Interactions are modeled using data and event ports, conditional and unconditional flow, and binding between components. AADL has thread, thread groups, subprogram and process components, which can be used to model the software application. AADL can also model different modes of a system and their transitions under different conditions. It is possible to model processors, memories, minimal operating system, virtual processors, virtual bus, etc.

AADL models can be hierarchical, with system of systems, system component and subcomponents. One can inherit and instantiate particular system usage for analysis purpose. It is possible to perform various types of analysis, such as schedulability analysis, system latency analysis, and change impact analysis, using its plug-in annexes. Error model annex is one such analysis plug-in, which is especially useful for system safety analysis.

6.2.1 Error Modeling Using Annex EMV2

The error model annex [2] can be used to annotate the AADL model of an embedded system to support safety analysis. For embedded real-time systems, there are various types of errors which, if not handled, will lead to hazardous failures and abortion of mission. Therefore, it is essential to identify all these systematically in the system modeling phase. EMV2 provides support for architecture fault modeling within the AADL model environment. Error modeling can be done at component level, interactions between components and at composite level.

Component level error behavior: A component can be any basic element in the system such as input/output devices, user interfaces, sensors, actuators, switches, data bus, processors, memory devices, processes, threads, subprograms, databases. Any component participating in the hierarchical system can be an error source or an error sink.

There are five types of errors supported by EMV2 error library for components. They are service error, timing-related error, value-related error, replication error, and concurrency error. Service error can be item omission, sequence omission, service omission, transient service omission, late service start, early service termination, etc. Timing-related errors can be timing error, early delivery, late delivery, high rate, low rate, delayed service, early service, etc. Depending on the kind of components, we can specify the right kind category and kind of failures. Similarly, value-related errors can be out of range, below range, above range, etc. Out of order, stuck value, race condition, and starvation also can be modeled using above type of errors.

Interaction between components: Error behavior: Focus here is on error propagation between system components and with the environment. When the component becomes an error source, how it gets propagated to other systems in the environment, and how an incoming error is handled is also addressed here.

Composite error behavior: Composite error behavior depicts how the main system handles or absorbs errors of subsystems. Errors which cannot be handled are sent out to the environment in terms of a failed system or a degraded system behavior.

Error behavior state machine modeling using EMV2: After identifying the types of errors relevant to each component, its error behavior is to be modeled. The presence of errors can modify the behavior of the system. Some error may lead to failure of the system, while some other failures can be handled. Error behaviors supported by EMV2 are fail stop, degraded fail stop, fail and recover, degraded recovery, etc.

6.2.2 Behavior Modeling Using Behavior Annex BLESS

BLESS [4] is a framework for behavioral interface specification and verification in AADL. Its key specifications include design specification that captures both functional and timing properties that aligns with the AADL environment. BLESS sub-clause has sections for variables, states, and transitions. States must be one of initial, complete, final or execute. Assertions are used in BLESS to indicate behavior specifications. Port assertions express what is true, when an event is sent or received on a port. State assertions express what is true while the machine is in that state. More details on BLESS can be found in the case study section.

6.2.3 Basic System Modeling Using AADL

For model-based safety analysis, first step is generating a system model from its functional requirements. This section introduces a novel method for generating AADL model manually from system functionality. AADL model has to be complete in terms of its functionality, error behavior, and real-time requirements. The following are the steps identified for the same.

Identification of input elements interacting with real-time environment: At the first level, interaction of the system with real environment in terms of its input is to be captured. That is, all inputs (sensors, human interface, data from cooperating systems, etc.) have to be captured. A single device input can have one or more input data.

Identification of processing requirements for each of the components: Processing requirements on each of the input is to be captured next. One input can have one to many relations to a processing system, but there has to be at least one processing element for each of the input data.
Identification of output components: Once all functional processing elements are identified outputs are to be mapped logically. Each functionality can be a subcomponent in the model.
Identifying major subsystems: Subsequently, each functional subcomponent within the system can be expanded to more detailed level, maintaining the consistency of data flow and interfaces. Each functional subcomponent can further get decomposed to processing subcomponents. Data and event flows between subcomponents can be identified at this level. I/O interfaces, sensor interfaces, and software processes are to be further expanded till all inputs are mapped to output through a flow.
Identification of system modes: Modes of operation of the system associated with functionality due to mode of computing platform (initial, normal, degraded and failed) are identified at this level.
Identification of processor(s), thread(s), and other hardware components: Next step is allocation of resources that is binding processors to process and then, to threads, attaching frequency requirement of input and output, protocols, etc. This is the phase in which computing platform details are loaded to the model components and model becomes a true representation of actual system.
Modeling error behavior and integrating to basic model: Once AADL functional model as per the above steps is created, next step is to model error behavior and integrate into the system. In this data error, event malfunctions, failure modes of each process, transition conditions, and propagation conditions are identified and modeled using annex EMV2 [2].
In this step, all input failures and output failures are also to be considered. All input failures will not result in output failures, if there is an error sink in between. All input failures are sources of failures, and all output failures have a source of failure or sources of failures. Those failures which are having error sink have to be mitigated or have to be proven that there are no effects of sinking the failure to the system failure. All other failures are to be prevented. If it cannot be prevented redundancy has to be brought into the system. Error detection and switch over conditions are to be modeled.
Modeling system behavior using BLESS: Functional behavior of the system in terms of initialization, real-time events, etc. are modeled in this step. Real-time behavior of the system is captured as thread implementation using BLESS. Such a thread implementation typically captures the variables that the thread handles, change of values of such variables and assertions that capture some of their properties that need to be met.

6.3 Automatic Flight Control System: Case Study

We now present a case study of safety validation of a digital fly-by-wire automatic flight control system (AFCS). We discuss the AFCS, its AADL models and safety validation of AFCS using parameters generated from AADL models.

6.3.1 Overview of AFCS

The AFCS (refer to [1] for an example) is a dual channel digital automatic flight control system, which helps to reduce the workload of the pilot when engaged. One channel is the control channel and other channel is the monitoring channel, which will be in standby mode. In case of failure of controlling channel, the monitoring channel will take over the control. At any time, only one channel will drive the output.

AFCS also does landing gear control activities, depending on the flight mode of the aircraft for automatic landing. It takes inputs from inertial navigation system, air data system, pilot stick, and autopilot panel. It also gives out data for pilot displays, crash data recorder and gets U home system. All the six actuators will continue to receive commands even in the case of failure of any one of the channels. There is dual redundancy available for hardware but, no redundancy in case of software, toward minimizing complexity in design and validation. So it is essential that the software is almost 100% fault free, toward safe and reliable flight control.

Figure 6.1 depicts the high-level AADL model of AFCS. Figure 6.1 shows the model of the top-level system in which two channels are interacting with the redundancy management and control algorithms. We also include an error model corresponding to this high-level AADL model. Each component subsystem in the high-level AADL model is refined into a detailed system. Figure 6.2 depicts one such subsystem corresponding to the controller. The various ports with which the controller interacts including the devices for input, the output valves to which the actuator commands are sent, the CCDL bus, etc., are depicted in the figure.

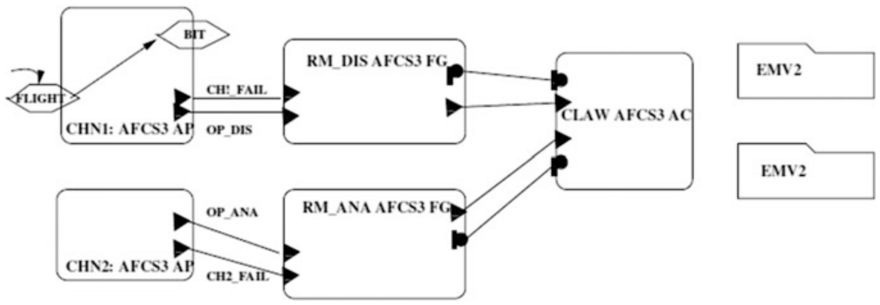


Fig. 6.1 High-level AFCS model in AADL

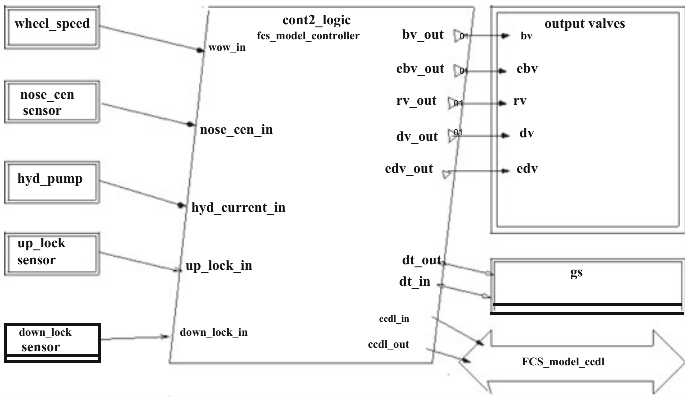


Fig. 6.2 Detailed AFCS model of one controller of AFCS in AADL

6.3.2 *Error Modeling of AFCS*

The error model for AFCS depicts various possible errors that we would like to capture, as occurring in the original AFCS system. Such errors, when explicitly captured, help designers to analyze potential faults in the system and devise appropriate fault handling mechanisms. The text extract below depicts a small fragment of the AADL error annex EMV2 model for AFCS. There are two kinds of errors depicted here. In the first part of the figure, error propagation along the discrete and analog channels is modeled, and the second part models composite error behavior in the flight guidance (FG) subsystem and autopilot (AP) subsystem.

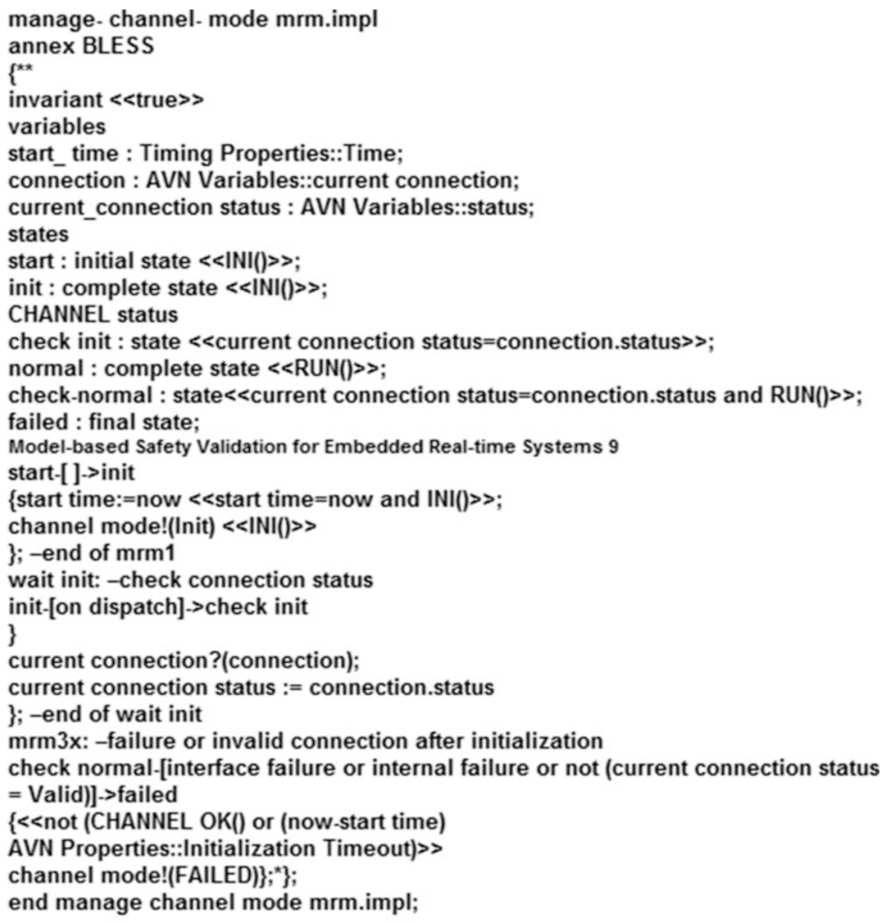
```
annex               emv2f**
use types           ErrorLibrary;
use behavior        ErrorLibrary::Failstop;
error propagations
                    From_rm_dis: in propagation fbadValueg;
                    From_rm_ana: in propagation fbadValueg;
                    OutPort: out propagation fbadValueg;
end propagations;
component error
behavior propagations
                    Failstop-[]! outportfbadValueg;
                    Operational-[From rm disfbadValueg and From rm
                    anafbadValueg] !outportfbadValueg;
end component;**g;
```

The extract below is an error model corresponding to a composite behavior depicting failure of one or more components. The actual ways in which the components fail are not captured in the model below, but, it states whether the system will continue to be operational, will degrade or will fail and stop.

```
composite error behavior
                                states
                                [AP1.Operational and AP2.Operational and
                                FG1.Operational and FG2.Operational

                                and AC.operational] ! Operational;

                                [AC.Operational and 1 ormore (FG1.failstop,
                                AP1.failstop) and FG2.Operational and AP2.Operational
                                or 1 ormore (FG2.failstop,AP2.failstop)
                                and FG1.operational and AP1.operational] ! degraded;

                                [AC.failstop or 1 ormore (AP1.failstop, FG1.failstop)
                                and 1 ormore (AP1.Failstop,FG1.Failstop)
                                and 1 ormore (AP1.Failstop,FG2.Failstop)] ! failstop;
end composite;
```

6.3.3 Behavior Modeling of AFCS

The text extract below depicts a small fragment of behavior modeling using BLESS annex of AADL. We have modeled implementation details of a thread corresponding to a processing channel common for all triple redundant architectures. Such an implementation indicates the variables manipulated by the thread, states indicating the states of the channel and transitions involving change of states.



```
manage- channel- mode mrm.impl
annex BLESS
{**
invariant <<true>>
variables
start_ time : Timing Properties::Time;
connection : AVN Variables::current connection;
current_connection status : AVN Variables::status;
states
start : initial state <<INI()>>;
init : complete state <<INI()>>;
CHANNEL status
check init : state <<current connection status=connection.status>>;
normal : complete state <<RUN()>>;
check-normal : state<<current connection status=connection.status and RUN()>>;
failed : final state;
Model-based Safety Validation for Embedded Real-time Systems 9
start-[ ]->init
{start time:=now <<start time=now and INI()>>;
channel mode!(Init) <<INI()>>
}; –end of mrm1
wait init: –check connection status
init-[on dispatch]->check init
}
current connection?(connection);
current connection status := connection.status
}; –end of wait init
mrm3x: –failure or invalid connection after initialization
check normal-[interface failure or internal failure or not (current connection status
= Valid)]->failed
{<<not (CHANNEL OK() or (now-start time)
AVN Properties::Initialization Timeout)>>
channel mode!(FAILED)};*};
end manage channel mode mrm.impl;
```

Behaviour Model of Channel Mode Change

6.4 Safety Validation of an Embedded System

As per model-based safety engineering, there are two important aspects of safety assurance, first is modeling of system to make the requirements complete and consistent and second is validation of the system specifically with respect to safety. Once the safety requirements are complete, the next step is to validate the system. As per DO-178B [7], there are many levels of testing, which are to be conducted systematically. Each of the levels has their own criteria for generating test cases and

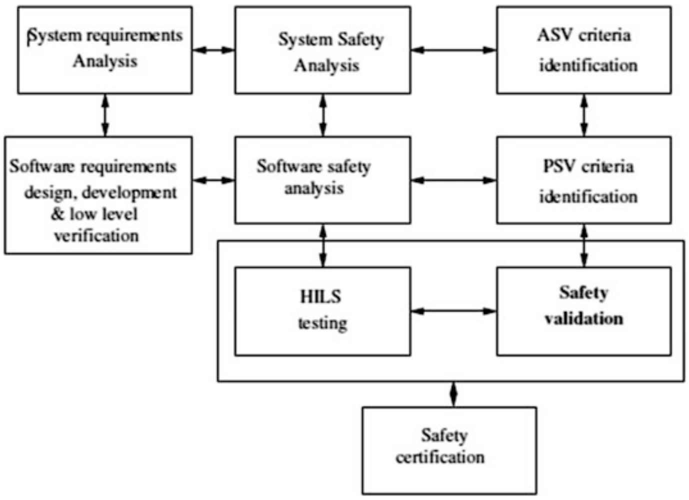

System requirements Analysis
System Safety Analysis
ASV criteria identification
Software requirements design, development & low level verification
Software safety analysis
PSV criteria identification
HILS testing
Safety validation
Safety certification

Fig. 6.3 Safety validation as a part of system testing

expected results. An important point to note is that overall safety of a system can be compromised in spite of each of the levels of testing being completely fulfilled.

We proposed the notion of a safety monitor in [6] to address this issue and capture additional aspects of safety assurance during the integration testing phase. Safety monitor design begins at the requirements phase, continues into design and coding phases and is implemented as a part of the testing phase. Figure 6.3 depicts the process life cycle that a safety monitor follows along with the existing safety assurance life cycle.

Based on the requirements pertaining to safety and on the platform constraints influencing safety, various parameters to be monitored are listed and these parameters are monitored as a part of the testing phase. Application-aware safety monitor (ASM) takes care of safety captured as a part of the overall system functionality. Parameters to be monitored here include all application level data that capture the behavior of the system as running in the system simulated environment. Platform-aware safety monitor (PSM) takes care of safety interpreted as a non-functional requirement. This includes parameters that are monitored with respect to computing platform environment. Behaviors violating safety, if found, are analyzed, traced back to appropriate life cycle phases and the required modifications are made to repeat the process of safety assurance.

Existing methods for system safety analysis, including that of safety monitor above, are manual, error prone, and disintegrated. There is chance of critical failure modes arising out of interactions remaining dormant, till system integration testing. This sets the context of our proposal.

6.5 Derivation of Safety Parameters from AADL Models

AADL, through its capabilities, brings in model-based system engineering, where in system hazards and failure modes can be identified earlier, in the requirements phase itself. As mentioned earlier, it can model system of systems and their direct and indirect interactions and interdependencies. Several parameters for safety can be extracted automatically from detailed AADL models.

Platform-aware safety parameters are error behaviors of the computing platform components, which are introduced for digital implementation. These components and their modes can be easily picked up from the model. Introduced system components could be processor, memory, databases, data bus, and data buffer. Input and output ports of these components, their status modes and errors will constitute the platform-aware parameters to be monitored in real time.

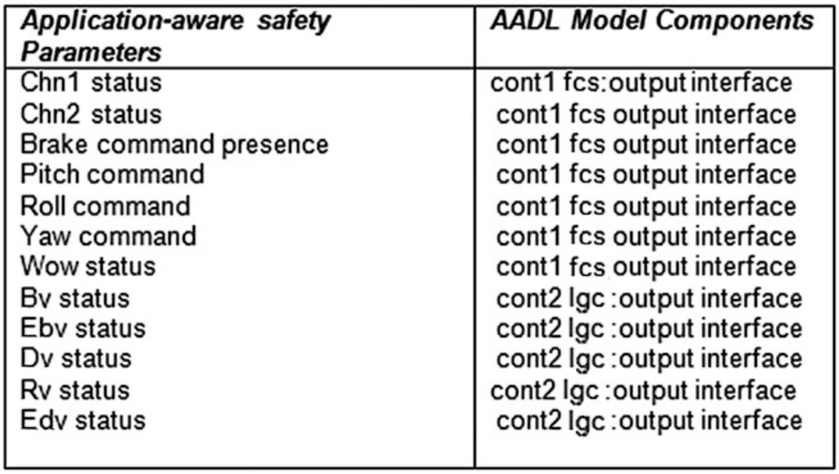
Application-aware safety parameters are error behaviors of the real-time system components, which are not part of computing environment. These are system components picked up from the environment which form input and output of the model and all intermediate error behaviors in the transition path from input to output. Modes and their transitions of certain components will also contribute to such parameters.

6.6 Safety Validation of Flight Control System

The following is a list of platform-aware safety parameters that were automatically derived from the AADL model. The components to which each of the parameters belongs to are also depicted in the table below.

Platform-aware safety parameters	AADL model components
Brake pressure_validity	this_cont2_lgc, thread pressure_range_check
Brake_pressure_buildup_flag	this_cont2_lgc, thread pressure_build_up
Mil_bus_link_status	this_milbus, thread validate mil_data
Brake_command_presence	this_cont2_lgc, thread_break_command
Brake_command_delay	This_cont2_lgc, thread time_elapsed
AP_status	AFCS3:AP, threadEngage_Autopilot
AP_interface_status	AFCS3:AP,read_user_input
AP_switch_over_status	AFC3:AP, thread channel_change_over
AP_hw_fail	AFCS3:AP,thread check_bit_status
AP_sw_fail	AFCS3:AP, thread check_wdm_status
AP1_exception_vector	AFCS3:AP, device main_processor
AP2_exception_vector	AFCS3:AP, device main_processor
Chn1_mode	AFCS3:AP channel_status_change_over
Chn2_mode	AFCS3:AP channel_status_change_over

Similarly, the table below lists the application-aware safety parameters and their corresponding components.



Application-aware safety Parameters	*AADL Model Components*
Chn1 status	cont1 fcs:output interface
Chn2 status	cont1 fcs output interface
Brake command presence	cont1 fcs output interface
Pitch command	cont1 fcs output interface
Roll command	cont1 fcs output interface
Yaw command	cont1 fcs output interface
Wow status	cont1 fcs output interface
Bv status	cont2 lgc :output interface
Ebv status	cont2 lgc :output interface
Dv status	cont2 lgc :output interface
Rv status	cont2 lgc :output interface
Edv status	cont2 lgc :output interface

6.7 Conclusion

We have introduced a method for generation of safety criteria, for safety monitoring, used for safety assurance of embedded airborne safety-critical software. Systematic method of generating consistent and complete system model using AADL was also explained. Error annex EMV2 and behavior annex BLESS are used for modeling of fault tolerant, real-time behavior of the system. Platform-aware and application-aware parameters pertaining to safety can be automatically generated from such AADL models. As a part of our ongoing work, we are focussing on developing a tool base for automatic generation of such parameters and going further, test cases for monitoring these parameters to be integrated with the system level testing setup.

References

1. Official website of tejas. http://www.tejas.gov.in/technology/fly-by-wire.html
2. Delange J, Feiler P (2014) Architecture fault modeling with the AADL error-model annex. In: Proceedings of 40th EUROMICRO conference on software engineering and advanced applications. IEEE, pp 361–368
3. Feiler P, Gluch D (2012) Model-based engineering with AADL: an introduction to the SAE architecture analysis and design language. Addison Wesley
4. Larson BR, Chalin P, Hatcliff J (2013) Bless: formal specification and verification of behaviors for embedded systems with software. In: Proceedings of NASA formal methods: 5th international symposium. Springer, Berlin, pp 276–290

5. Levenson N (2012) Engineering a safer world: systems thinking applied to safety. MIT Press
6. Philip G, D'Souza M (2015) Safety validation of an embedded at hardware-software integration test environment. In: Proceedings of 9th ICACCT. Springer, Berlin
7. RTCA (1992) Software considerations in airborne systems and equipment certification DO-178B
8. SAE ARP 4761 (1996) Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment

Chapter 7
Arguing Formally About Flight Control Laws Using SLDV and NuSMV

Natasha Jeppu and Yogananda Jeppu

Abstract Software systems have failed in the recent past. This is most often attributed to wrong requirements often caught very late in the program or escapes from the rigorous process leading to failures. There is a necessity to ensure that the requirements are correct up front before the design and verification process start. Formal methods have become popular these days and a lot of impetus is there in the industry to apply these techniques to safety critical projects especially in flight controls. This paper looks at two tools NuSMV, an open source model checker, and Simulink Design Verifier, a commercial model checker. It is seen that these can be practically applied to projects and design. These are very successful in finding defects in design and requirements as demonstrated on a set of mutants.

Keywords Formal methods · NuSMV · Simulink design verifier
Safety critical · Aerospace

7.1 Introduction

Software has failed even as recently as 2015 in aerospace systems. The new bug list of F35 published in 2016 indicates that we have not mastered the art of engineering good software [1]. A preliminary study was reported on the use of Simulink Design Verifier (SLDV) for aerospace application [2]. This paper described mode transition logic and demonstrated how easy it was for engineering students to work with the SLDV software to describe behavior formally. Formal methods are mathematical techniques for specifying, developing, and verifying software. This statement is from the DO 333 aerospace standard supplement to DO 178C, which was

N. Jeppu (✉)
National Institute of Technology Karnataka, Surathkal, Mangalore, India
e-mail: Natasha.Jeppu@gmail.com

Y. Jeppu
Honeywell Technology Solutions, Bangalore, India
e-mail: Yogananda.Jeppu@Honeywell.com

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_7

released in late 2011 [3, 4]. This supplement brings out case studies on the use of formal methods in aerospace software development process. A good study of use of formal methods using DO 333 is also reported in the NASA report [5]. Formal methods have picked up impetus in the last few years with Airbus using it for reducing the test activity in their projects [6, 7]. Other aircraft companies are looking at using formal methods in the certification process. This has however not been very well accepted by the managers nor by the engineers [8].

The barriers to the utilization of formal methods in aerospace projects are brought out in a survey [9]. The major barrier indicated is surprisingly education. The engineer in the industry needs to be educated in the use of formal methods. There is a need for highly trained experts in formal methods as mentors for the engineers. Certification engineers need to be trained and educated in the formal methods process. This also brings out the necessity for practical application papers on the use of formal methods. There is a need for bringing out the ease or complexity of use of formal methods. This paper is an attempt to capture the experiences of using formal methods in form of the use of Simulink Design Verifier—a commercial formal methods tool from MathWorks [10]. An open source tool NuSMV is also used to compare the performance on mode transition logic problem [11]. Several models are available on the MathWorks Web site so that the community can use it to experience the use of the tool [12].

7.2 Simulink Design Verifier

This is a toolbox from MathWorks which work on the Simulink design. It is primarily a formal model checker. Model checking is defined as quote "Model checking is a verification technique that explores all possible system states in a brute-force manner" [13]. The design model is checked against the verification subsystem which is a set of assertions. There is a facility to put assumptions or constraints on the inputs and define an implication as a combination of logic blocks. SLDV supports timed automata. It comes with a set of blocks that can help the user set up time dependency in terms of number of frames. The newer version of SLDV has a feature to take a legacy C code, make it into an S-Function, and use it with the SLDV to find errors in the C code. An S-Function is a Simulink function compiled from a C code and executing in the Simulink environment.

A simple example of an autodestruct system demonstrates the use of SLDV. In an aerospace application, if the pitch angle ||Theta|| is >=25 degrees the autodestruct flag is set to True. This is modeled in Simulink as shown in Fig. 7.1. The system had an error—the absolute was missing in the code [1]. The formal assertions for the system are defined in Fig. 7.2. If the angle (In) >=25 OR angle <=−25 implies that (=>) the autodestruct (in1) input is True. The block (P) indicates to SLDV that this needs to be proved using the inbuilt model checker. SLDV proves that the model without the absolute is wrong and provides counter example with input "in" set to −25.0 to prove it wrong. This is a simple example but all the complicated

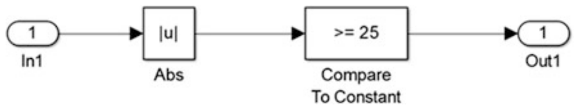


Fig. 7.1 Model of the autodestruct system

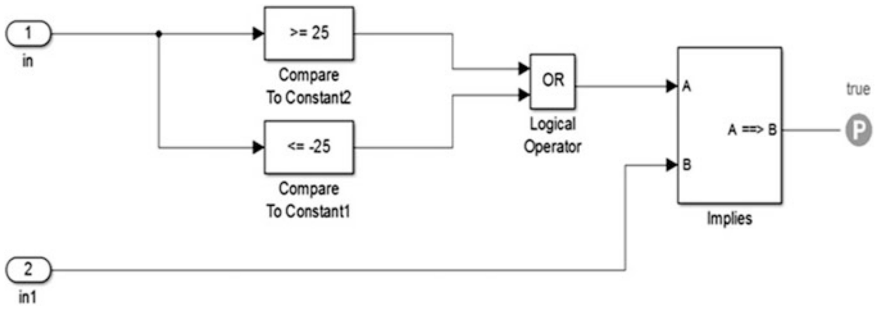


Fig. 7.2 Assertions added to the autodestruct system

examples can be simplified in some form or the other and represented with the logic, comparators, timed blocks in combination with the (P) block to prove a specific assertion.

7.3 NuSMV

NuSMV is a software tool for the formal verification of finite state systems developed jointly by Fondazione Bruno Kessler FBK-IRST and Carnegie Mellon University [11]. It is a reimplementation and a reengineering of the Symbolic Model Verifier (SMV) model checker developed by McMillan at Carnegie Mellon University during his Ph.D. [14]. NuSMV checks finite state machines or systems against specifications in the temporal logic. The language of NuSMV allows the description of finite state systems. It supports both synchronous and completely asynchronous behavior. The purpose of the language is to describe the transition relation of a finite Kripke structure.

The above problem is implemented in the NuSMV language as shown below. LTLSPEC G identifies the assertion as in the SLDV case.

```
MODULE main
VAR
Theta : -30 .. 30;
auto_dest : boolean;
```

```
  ASSIGN

  auto_dest := case
  (Theta) >= 25 : TRUE;
  TRUE : FALSE;
  esac;

  LTLSPEC G (((Theta >= 25) | (Theta <= -25)) ->
  (auto_dest = TRUE));

  -- specification G ((Theta >= 25 | Theta <= -25) ->
auto_dest = TRUE) is false
  -- as demonstrated by the following execution se–
quence
  Trace Description: LTL Counterexample
  Trace Type: Counterexample
   -> State: 1.1 <-
     Theta = -15
     auto_dest = FALSE
   -> State: 1.2 <-
     Theta = -28
   -- Loop starts here
```

NuSMV and SLDV, both behave the same way by generating test cases as counter examples that falsify the assertion if the absolute is missing in the design. This is the power of formal methods that can easily validate control systems, and safety properties in aerospace applications. The rest of the paper brings out examples of the use of NuSMV and SLDV in proving the correctness of the system.

7.4 Autopilot Mode Transition

An autopilot is a system that ensures the aircraft flies a specific course reducing the pilot's workload on long flights. The innermost component of an autopilot is a mode transition logic that changes the autopilot behavior based on pilot inputs or internal software flags. A typical mode transition design and test method were described earlier as an example of using assertions in validation of the design [15]. A simplified model of the mode transition is available on the MathWorks file exchange that models only the vertical modes of an autopilot [16].

Typical vertical modes of an autopilot are the Pitch Attitude Hold (PAH) mode, the Altitude Hold (ALT HOLD) mode, the Speed Hold (SPD HOLD) mode, the Vertical Speed Hold (VS HOLD) mode, and the Altitude Select mode (ALT SEL) as shown in Fig. 7.3. The PAH mode maintains the aircraft's pitch angle fixed

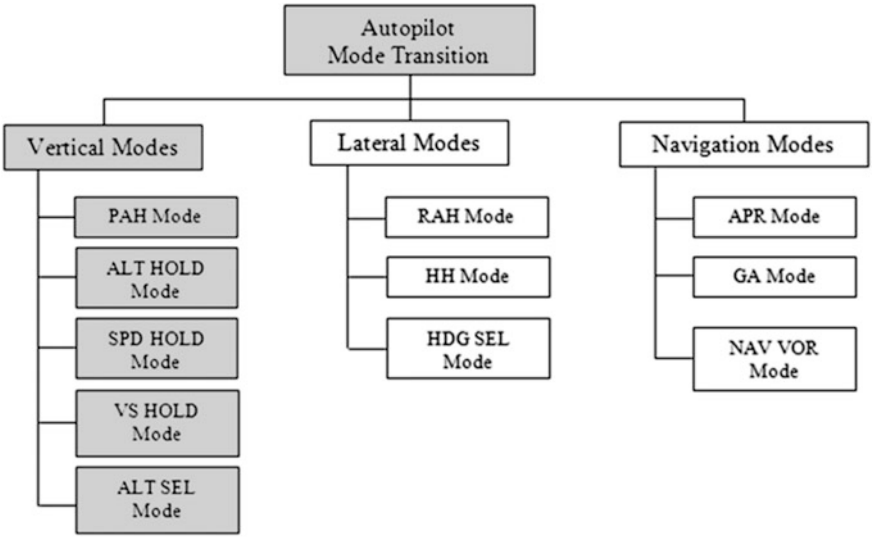
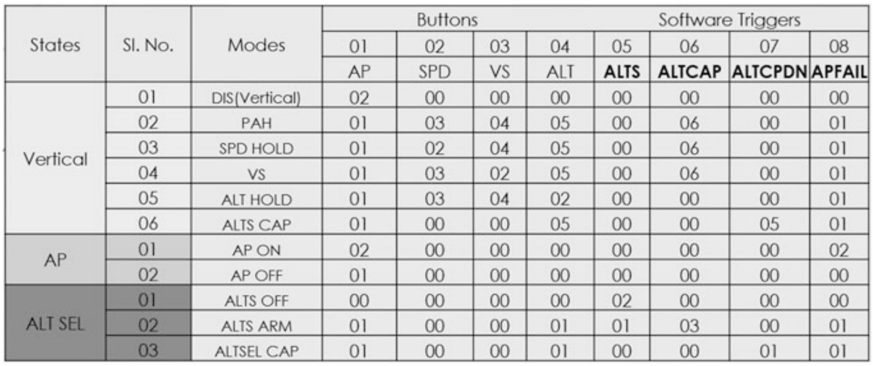


Fig. 7.3 The autopilot modes

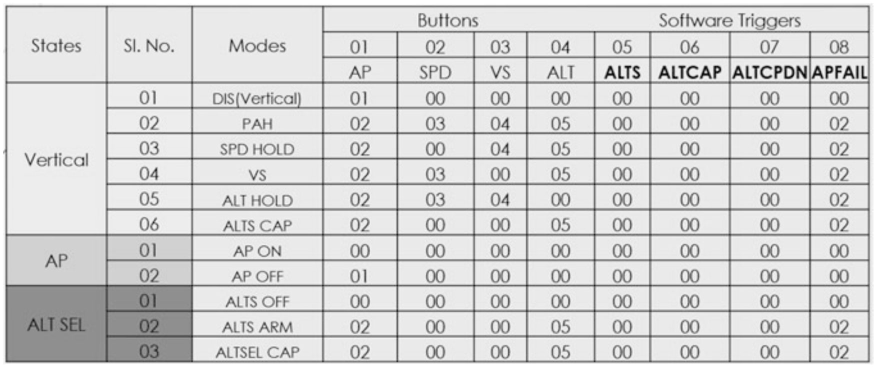
constant at the value when the mode is selected. The ALT HOLD mode holds the aircraft's height or altitude at a value when the mode is selected. In this mode, the pitch angle is free to change. The SPD HOLD mode ensures that the aircraft maintains its speed at a value when it is selected but the pitch angle and altitude can change. Similarly, VS mode holds a constant climb rate. The ALT SEL mode is a special mode where the aircraft climbs toward a selected altitude and on reaching it transfers to ALT HOLD mode automatically. All these modes and the transitions are based on pilot selection or internal triggers. These are described in the design as a set of two tables—the mode transition table and the condition table.

Figure 7.4 shows the transition table as given in the example [16]. There are three distinct independent major modes—Vertical, AP, and Alt Sel. Vertical has 6



States	Sl. No.	Modes	Buttons				Software Triggers			
			01	02	03	04	05	06	07	08
			AP	SPD	VS	ALT	**ALTS**	**ALTCAP**	**ALTCPDN**	**APFAIL**
Vertical	01	DIS(Vertical)	02	00	00	00	00	00	00	00
	02	PAH	01	03	04	05	00	06	00	01
	03	SPD HOLD	01	02	04	05	00	06	00	01
	04	VS	01	03	02	05	00	06	00	01
	05	ALT HOLD	01	03	04	02	00	00	00	01
	06	ALTS CAP	01	00	00	05	00	00	05	01
AP	01	AP ON	02	00	00	00	00	00	00	02
	02	AP OFF	01	00	00	00	00	00	00	00
ALT SEL	01	ALTS OFF	00	00	00	00	02	00	00	00
	02	ALTS ARM	01	00	00	01	01	03	00	01
	03	ALTSEL CAP	01	00	00	01	00	00	01	01

Fig. 7.4 Transition table



States	Sl. No.	Modes	Buttons				Software Triggers			
			01	02	03	04	05	06	07	08
			AP	SPD	VS	ALT	**ALTS**	**ALTCAP**	**ALTCPDN**	**APFAIL**
Vertical	01	DIS(Vertical)	01	00	00	00	00	00	00	00
	02	PAH	02	03	04	05	00	00	00	02
	03	SPD HOLD	02	00	04	05	00	00	00	02
	04	VS	02	03	00	05	00	00	00	02
	05	ALT HOLD	02	03	04	00	00	00	00	02
	06	ALTS CAP	02	00	00	05	00	00	00	02
AP	01	AP ON	00	00	00	00	00	00	00	00
	02	AP OFF	01	00	00	00	00	00	00	00
ALT SEL	01	ALTS OFF	00	00	00	00	00	00	00	00
	02	ALTS ARM	02	00	00	05	00	00	00	02
	03	ALTSEL CAP	02	00	00	05	00	00	00	02

Fig. 7.5 Condition table

submodes which starts with disconnect (DIS) and the 5 other modes defined earlier. The Vertical major mode can exist in only one submode in a frame. At start (when switched on), the autopilot is in Vertical → DIS, AP → AP OFF, and ALTSEL → ALTSOFF. The (→) indicates the major → submode pair. The autopilot transits from these modes based on the table. The number in the 4th column indicates the modes that it can transit to. There are 4 buttons that the pilot can press—AP, SPD, VS, and ALT. There are 4 software triggers that the system can generate—ALTS (altitude select), ALTCAP (altitude capture), ALTCAPDN (Altitude Capture Done), and APFAIL (Autopilot has failed).

The table is read as follows. These are the system behavior requirements.

The autopilot can transit from Vertical → DIS (01) (Sl No 1) to 02 (read in column under AP, same row) if AP button is triggered.

The autopilot can transit from AP → AP OFF to AP → AP ON (01) if AP button is triggered.

The autopilot can transit from Vertical → SPD HOLD (03) to Vertical → VS (04) if the button VS is triggered.

Normally, the transition happens if certain conditions are true. This is captured in the condition table (Fig. 7.5). Thus, the transition table is read in conjunction with the condition table. The requirements are actually modified in conjunction with the condition table as

The autopilot transits from Vertical → DIS (01) (Sl No 1) to 02 (read in column under AP) if AP button is triggered AND condition number 01 is true.

The autopilot transits from AP → AP OFF to AP → AP ON (01) if AP button is triggered AND condition number 01 is true.

The autopilot transits from Vertical → SPD HOLD (03) to Vertical → VS (04) if the button VS is triggered AND condition number 01 is true.

The condition number 01 could be a set of parameters that set it to True indicating that the autopilot can come on. A typical set of condition would be C01 = (100 kts < Speed < 300 kts) AND (−5 deg < Pitch Angle Theta < 15 deg) AND (−20 deg < Roll Angle < 20 deg). Condition 04 could be a bound on the current vertical speed.

7.5 Automated Validation

A set of Matlab scripts are available on the MathWorks file exchange Web site. These help in validating a mode transition design given in the above form [16, 17]. This method has been tried with different mode transitions like a Radar application or the famous Therac-25 modes and found to work well. One of the scripts generates an English text output that defines the modes. A typical output generated automatically given the design looks like below. This can be read and verified for correctness.

(1) If in State DIS (Vertical) AND Trigger AP occurs THEN transition to PAH if condition C1 Is TRUE
(2) If in State PAH AND Trigger AP occurs THEN transition to DIS (Vertical) if condition C2 Is TRUE

Two additional scripts generate a Matlab code and a NuSMV code from the same two tables. These can be used for the validation of the mode transition using assertions in SLDV or in NuSMV, respectively. The assertions are defined based on the behavioral truth about the system. Four assertions are identified.

1. If Vertical mode = ALTCAP then ALTSEL mode = ALTSEL CAP
2. If Vertical mode = ALT HOLD then ALTSEL mode = ALTS OFF
3. If Vertical mode = DIS then AP = AP OFF
4. If Vertical mode = DIS then ALTSEL = ALTS OFF

The SLDV assertion of (2) and (3) above is shown in Fig. 7.6. The first input "in" is the Vertical mode. If this is equal to 5, i.e., ALT HOLD, it "implies that" the third input "in2", i.e., ALT SEL mode is equal to 1 (ALTS OFF). Similarly, if the second input "in1" AP is equal to 2, i.e., AP OFF "implies that" Vertical mode equals 1 (DIS). The ease of putting in assertions is very clear from the figure.

A similar assertion in NuSMV would look like this

$$\text{LTLSPEC G } ((\text{Vm} = \text{ALT\_HOLD}) \rightarrow (\text{ASm} = \text{ALTSOFF}))$$
$$\text{LTLSPEC G } ((\text{Vm} = \text{DIS}) \rightarrow (\text{APm} = \text{APOFF}))$$

Executing the SLDV and the NuSMV indicates that the assertions are true and therefore we are confident that the design is good. This does not show the capability of NuSMV or SLDV to find design errors. A set of 9 errors or mutation were

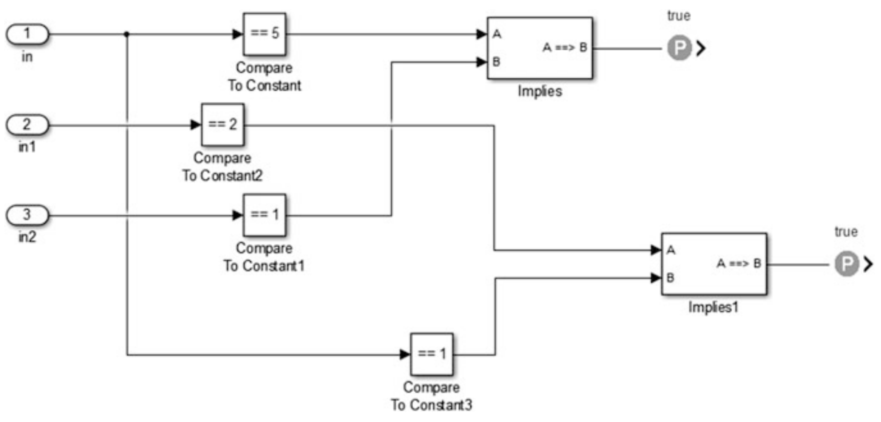


Fig. 7.6 SLDV assertions for (*2*) and (*3*)

introduced into the Simulink model and the NuSMV to identify if the tools bring out the error. Eight mutants are caught by the formal methods. One mutant is a "dud" as the later blocks correct this error, and therefore the model behaves correctly even with the mutation present. These mutants are available in file exchange [16].

7.6 Formal Method Versus Random Tests

There is always an argument when one discusses formal methods—how does it compare against testing. This aspect is addressed here. NuSMV model cannot be tested with a test vector whereas the Simulink models can be tested. Simulink has both a simulation platform and a formal methods platform. The Autopilot mode transition was modeled in Simulink and the assertions added as above. Very specific errors were injected into the model to create 9 mutant files. The mutant files are the actual model copy with a very specific error introduced deliberately into it. Random test case vectors are injected into the test harness and the test stops as soon as the assertion is falsified. The same exercise is repeated with the SLDV and the time takes for falsification of the assertion is captured. The test and SLDV proof are repeated 5 times, and a mean and standard deviation are computed from the trials and tabulated in Table 7.1.

The overall mean times for SLDV and random tests are similar at about 20 s. The mean times for individual mutants are, however, very different. The overall minimum time for SLDV is 12.2 s, and for random this is 1.31 s. The overall maximum time of SLDV is 23.6 s, and for random 91.05 s. There is a large disparity in the time to falsify the assertion in the random tests. This is also seen in the mean standard deviation across all tests being small for SLDV at 1.3 s, and the

Table 7.1 Summary of mutants and test timing

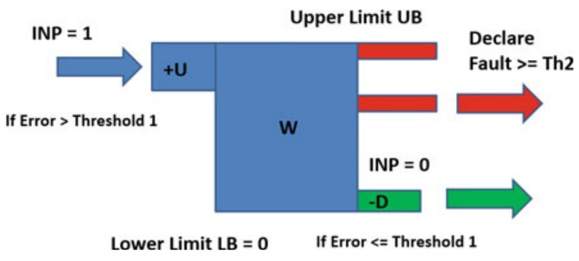
S. No.	Mutant	SLDV mean (s)	SLDV std (s)	Random mean (s)	Random std (s)
1	DIS replaced by code 2 instead of 1	16.8	0.45	1.31	0.79
2	VS_HOLD replaced by code 1 instead of 4	21.4	0.55	64.61	15.17
3	APOFF made APON in AP mode	21.4	0.55	4.56	2.74
4	ALTS trigger connected directly without checking for ALTHOLD	23.6	2.30	7.24	6.52
5	Condition C2 for ALTCAP trigger removed	12.2	0.45	5.17	5.46
6	ATLCAP connected directly without checking for ATLSARM state	18.4	2.41	25.74	12.11
7	Condition C2 changed to APOFF true instead of APON true	21.6	5.18	1.35	1.08
8	ALTCPDN connected directly without checking for ALTCAP state	–	–	–	–
9	Switching transition conditions for ALTSARM and ALTSCAP in ALTSEL mode	19.2	0.45	91.05	94.76

random tests have 17.3 s. The large standard deviation for the last mutant at 94.7 s indicates that random tests can take a very long time before they find an error. This is also seen in an earlier study on test cases for control system models [18]. SLDV finds the deliberately injected errors in the models very easily. Can it find errors in safety critical flight code and models?

7.7 Up Down Counter

Several examples of SLDV finding bugs found during testing in flight control models are available [1]. These errors were found during the extensive software test activity. SLDV finds all these errors very easily. A very recent error found in a flight code for an Up/Down counter is described here as an example. Up/Down counters are blocks that count up with a fixed amount if there is an error in the signal, and count down at a slower rate if the error does not exist. Thus, these blocks can penalize a more frequent error than an intermittent fault that could be attributed to noise. The schematic of an Up/Down counter is shown in Fig. 7.7.

Fig. 7.7 Schematic of up/down counter



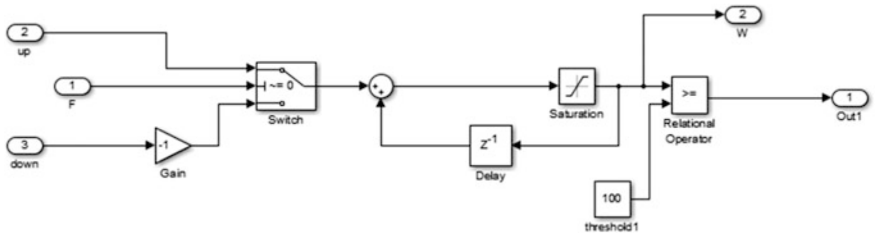


Fig. 7.8 Model of the up/down counter

The input to the counter is INP. This is set to true or 1 when the error (what we are monitoring) is greater than a threshold "Threshold 1". INP equal to adds the up count +U to the counter W. W is initially 0. INP equals 0 adds "−D" the down count to the count W. The lower limit for W is LB which is normally 0. The upper limit for W is UB. The value of W is clipped at UB. If W is greater than or equal to a trigger threshold "Th2", the output of the Up/Down counter is set to True.

In an aircraft flight control law, this was modeled in Simulink and coded in C language for implementation on board. The variable W was defined as an unsigned integer. During testing, a very specific combination led to W becoming negative which caused a wraparound, and it triggered a fault even though the error was less than threshold. This was tested using SLDV to see if it could catch the error. In the Simulink model, the data type for the adder block internal variable was defined as UINT16. The model is shown in Fig. 7.8.

There are two behavior properties that define the truth about the system or are the assertion

1. If the output is True previously and the input is True in the current frame implies the output is True in the current frame
2. If the output is False previously and the Input is False in the current frame implies the output is False in the current frame

These are modeled in SLDV as shown in Fig. 7.9. If Not first frame AND previous output (out1) is NOT True AND Input (F2) is NOT True AND U > D Implies Output is Not True. Similar assertions are made for the other properties. SLDV brings out a counter example to prove the assertion wrong replicating the condition observed during the test activity.

SLDV has new feature in the newer version of Matlab where a C code can be compiled into the Simulink environment. This compiled code is made compatible for SLDV analysis using options provided in the software. The actual code can be tested against the assertion. This is done for the Up/Down counter C code, and this error is easily caught by the formal methods analysis. The ease with which an assertion using the blocks can find errors in the C code is a very useful workflow. Legacy code from earlier projects, code libraries can now be validated in the Simulink environment.

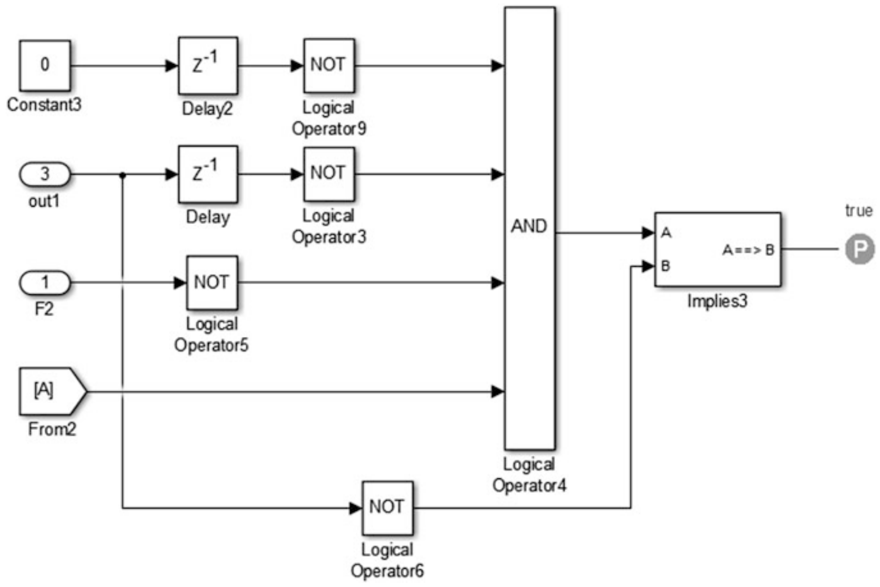


Fig. 7.9 Assertions of the up/down counter

7.8 Conclusion

SLDV and NuSMV, two model checking tools can be easily deployed in project and find errors in design very easily. The only constraint is that the requirements should be clear. It is very important that the system engineers clearly define the intent of their design as requirements rather than the design itself. It is very important that the requirements specify the "what" rather than "how". The use of formal methods drives home this discipline in defining requirements. The assertions are easily implemented by novice engineers with help from systems experts specifying the correct intent. Automation is brought in very easily to write NuSMV code from specification. It would be worthwhile, a exercise to specify requirements and automatically convert them into NuSMV or SLDV assertions. This will reduce the burden of the engineers trying to learn a new language or modeling paradigm.

The title uses the term "argue". The use of formal methods in a project brings out the argument between the engineers that is very important. We need to argue technically on the correctness of the design. The message and argument "is this what you want?" going all the way to the system designers by using these methods will lead to a better system design and safer world.

References

1. Jeppu Y (2016) Testing of safety critical control systems. http://in.mathworks.com/matlabcentral/fileexchange/39047-testing-of-safety-critical-control-systems. Accessed 15 July 2016
2. Jeppu N, Jeppu Y, Murthy N (2015) Arguing formally about flight control laws. Paper presented at international conference on industrial instrumentation and control (ICIC), pp 378–383, 28–30 May 2015. doi:10.1109/IIC.2015.7150771
3. RTCA, Inc (2011) Formal methods supplement to DO-178C and DO-278A. RTCA DO-333, USA
4. RTCA, Inc (2011) Software considerations in airborne systems and equipment certification. RTCA DO-178C, USA
5. Darren C, Steven PM (2014) Formal methods case studies for DO-333. NASA/CR–2014-218244
6. Laurent O (2010) Using formal methods and testability concepts in the avionics systems validation and verification (V&V) process. In: Third international conference on software testing, verification and validation, Paris, pp 1–10. doi:10.1109/ICST.2010.38
7. Moy Y, Ledinot E, Delseny H, Wiels V, Monate B (2013) Testing or formal verification: DO-178C alternatives and industrial experience. In: IEEE Software, vol 30, no 3, pp 50–57, May–June 2013. doi:10.1109/MS.2013.43
8. Stidolph DC, Whitehead J (2003) Managerial issues for the consideration and use of formal methods. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods: international symposium of formal methods Europe, pp 170–186. Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-45236-2_11
9. Davis JA et al (2013) Study on the barriers to the industrial adoption of formal methods. In: Pecheur C, Dierkes M (eds) Formal methods for industrial critical systems. In: 18th international workshop, FMICS 2013. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-41010-9_5
10. MathWorks. Simulink design verifier. http://in.mathworks.com/products/sldesignverifier/. Accessed 15 July 2016
11. NuSMV. http://nusmv.fbk.eu. Accessed 15 July 2016
12. Jeppu N (2014) Exploring design verifier. http://in.mathworks.com/matlabcentral/fileexchange/48858-exploring-design-verifier. Accessed 15 July 2016
13. Baier C, Katoen J-P (2008) Principles of model checking. MIT Press
14. McMillan KL (1992) Symbol model checking an approach to the state explosion problem. Dissertation, Carnegie Mellon University. http://www.kenmcmil.com/pubs/thesis.pdf
15. Rao M et al (2015) A methodology to design a validated mode transition logic. In: Vijay V et al (eds) Systems thinking approach for social problems. Lecture notes in electrical engineering, vol 327. doi:10.1007/978-81-322-2141-8_24
16. Jeppu N (2014) Exploring design verifier—2. http://in.mathworks.com/matlabcentral/fileexchange/51567-exploring-simulink-design-verifier-2. Accessed 15 July 2016
17. Jeppu N (2014) Exploring design verifier—3. http://in.mathworks.com/matlabcentral/fileexchange/54945-exploring-simulink-design-verifier-03. Accessed 15 July 2016
18. Jeppu Y et al (2014) Generating test cases with 100-percent requirements coverage using design of experiments. J Aerosp Inf Syst (Special Section on Software Challenges in Aerospace) 11:632–648. doi:10.2514/1.I010159

Chapter 8
Formal Methods: Techniques, Applications, Thrust Areas and Future Prospects

Krishnamani Kalyan

Abstract Formal methods are one of the oldest techniques of proving correctness of programs. Along with its contemporaries like Boolean Satisfiability and Theorem Proving it is also one of the areas which is still being actively researched. This in itself is a manifestation of the potential of the underlying techniques and the wide spectrum of applications it can positively impact. In this survey, we shall explore the most popular techniques, with an emphasis on industrial application of such techniques, and the various applications for which they are employed. We will then see some of the current and rapidly developing areas where formal methods could be invaluable, and we shall speculate some the future prospects of research and advancement in this field.

Keywords Formal methods · Safety critical · Model checking
Abstraction · Autonomous

8.1 Introduction

Formal methods refer to the set of techniques based on mathematics for the specification and verification of software and hardware systems. As it is grounded on well established mathematical techniques, it leads to unambiguous description of the designs and verified robust designs. Over the past several years, researchers have developed several techniques and tools for both specification and verification of the design implementations against such a specification. Some of the well-known formal specification languages include Esterel [1], Lustre [2], B-Method [3], ACSL [4], TLA+ [5], Why3 [6], which are also supported by formal verification tools. By making the specification of the functionality of a system (hardware or software) unambiguous, the verification process becomes more robust as one exactly knows what design functionality is being verified.

K. Kalyan (✉)
Nvidia Corporation, Santa Clara, CA, USA
e-mail: kalyans@nvidia.com

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_8

Lately, both hardware and software vendors have been increasingly adopting the use of formal methods to ensure reliability of their products. Although such methods are not adopted across all the products or all the features of a product, they are frequently employed to verify the most critical features of a product. With the increased interest in mobile and cloud applications, Internet of Things (IoT), artificial intelligence (AI)-powered robotics and self-driving vehicles, the need for formal methods is more apparent than ever before.

In this article, we will explore the widely used formal verification techniques, mention the popular use cases of formal methods in industry. We then talk about the important technical advancements that are making formal methods popular and more effective in an industrial setting. We conclude with some and some use cases, and speculate the prospects in years to come.

8.2 Formal Methods

Formal methods have two main components to it—Formal Specification and Formal Verification. Although we have heard more about the latter, it is just because of the metrics we attach to product reliability results—such as the number of bugs that were found by a particular verification tool, or the time needed by one verification tool compared to another. The former is equally important in coming up with an unambiguous description of the functionality of the designs we intend to develop. In fact, it is the first step in our development process. In the following sections, we will mention a few (formal) specification languages and also mention the verification tools that support these specification mechanisms.

8.2.1 Formal Specification

Formal specification is the first step in any system design process. One has to specify what the intended goals of the system are before getting down to build a hardware or a software system. Traditionally, specification has remained in voluminous documents, written in a natural language, like English, for example. Since design specification documents are the medium of communication between the architects (who make the blueprint of the design) and the designers (who code the design), it is imperative that we do not introduce ambiguity. By describing the intended functionality of the design in a natural language, we are doing exactly that—introducing more ambiguity and leaving essential details to the designer's interpretation.

Over the years, many system designers have realized the importance of unambiguous specification languages. First order logic is one of the simplest and most powerful specification languages with sound and complete deduction systems. There are many formal specification languages in the literature that are based on this simple logic. Further Temporal Logic, Temporal Logic of Actions, Separation Logic, etc., aided formal methods to a great extent.

There are several formal specification tools that are being widely researched and used in industry. The B-Method [3] is based on Abstract Machine which is just a set of states and operations that transform states from one to another. The specification language also allows us to specify the initial values of state variables as well as invariants. The B methodology encourages to specify the system at a very high level of abstraction and progressively refine it by adding more details to the abstract machine. Alloy language [7] is based on the notion of relations (Set Theory), that allows us to describe all desired configurations of a system as a set of constraints. ACSL [4] is ANSI C Specification Language that allows us to annotate C programs with a rich variety of properties such as assertions, loop invariants, variants, and quantified invariants. It is very similar to Java Modeling Language (JML). TLA+ [5] specification language is again a variant of state transition system and one can attach temporal properties to the specification. These specification languages are not just academic research tools, but are also applied in industry. The B-Method is used in several subway system specifications [8] by Alstom and Siemens, while TLA+ has been recently used by Amazon in some of its Web services [9], and ACSL is used in specification and verification of safety-critical avionics software.

8.2.2 Formal Verification

Formal verification corresponds to the set of tasks involved in establishing a mathematical proof that the designed system behaves according to its (formal or informal) specification. Even when the specification resides in a document written in a natural language, it is possible, though cumbersome, to understand and extract some crucial properties that could be verified of the system. There are various techniques developed over the past several years that cater to different kinds of hardware and software systems, as well as techniques that address certain specific kinds of problems. Most of the formal specification languages used in industry are also supported by well-developed formal verification tools. For example, B-Method is supported by Atelier-B, Alloy by Alloy Analyzer, TLA+ by TLA+ model checker, and ACSL by Frama-C. Formal specification is as important as the verification process itself, but we will focus more on formal verification for the rest of the article, beginning with the most popular formal verification techniques in the following section.

8.2.3 Formal Verification Techniques

There are several formal verification techniques addressing different kinds of verification problems and providing different degrees of automation. For industrial size problems, application of formal verification ranges from highly automated push-button tools such as Equivalence Checkers and Static Analyzers, tools such as model checkers that need some user intervention to abstract the models, and also those that require a lot of user interaction like Theorem Provers. They find their use in different applications of verification in hardware and software industries. Let us look into the main techniques used by popular formal verification tools that are employed in industry.

Model checking: It is one of the most popular formal verification techniques employed by industrial scale verification tools. This technique relies heavily on Satisfiability (SAT) solvers [10, 11] and Binary Decision Diagrams (BDDs) [12]. More recently, there are also techniques that combine BDDs and Satisfiability Modulo Theory (SMT) solvers [13]. SMT solvers intelligently combine solvers for different theories [14] to solve a propositional formula over variables involving several theories such as integers, Booleans, reals, and bit-vectors. Software programs are typical examples of such combinations. In contrast, SAT solvers solve a propositional formula over Boolean variables. Hardware designs are typical examples of such formulas. Model checkers proceed by constructing a model of the hardware or software program under verification. The model is essentially a state transition system or more generally a graph (or a BDD which is a canonical representation of a Boolean formula). Along with the model, we also provide the properties that we want to verify in our program. The property is specified as a logical formula. The solvers employ BDD-based techniques to traverse the constructed graph, checking if the logical formula holds in all reachable states. Model checkers employing SAT solvers boil down the graph representation into a Boolean formula and check for satisfiability of the formula. Most often, in order to overcome the combinatorial blow up of the state space resulting from the graph construction, most model checkers employ a technique called Bounded Model Checking (BMC) [15] that works with Boolean expressions of the corresponding finite state machine. BMC starts with an initial counterexample length, say k, and generates a propositional formula that is satisfiable if and only if such a counterexample exists. The procedure can be iteratively repeated for larger values of k. This technique finds minimal length counterexamples really fast and is most commonly used as a bug-hunting technique, rather than a proof technique. Some of the well-known model checkers include NuSMV [16], SPIN [17], and CBMC [18].

Equivalence checking: Software and hardware designs frequently change from one generation or one version to the next with the addition of several new features. One of the desired goals in industry is to make sure that the old functionality of the design is not altered (altered functionally, there might always be performance improvements) by the introduction of new features. Most often, this is achieved by checking the equivalence between the old and new designs with respect to certain behaviors (coded as properties) of interest. For example, the design itself might change but the properties of the protocol may still have to remain unaltered. Another use case is when we have an executable specification (say a C program of the design that can be executed and simulated) and when we need to verify this against a hardware implementation, equivalence checking is used. Formal equivalence checking techniques typically proceed by constructing a canonical representation (like a BDD) of the specification and the implementation and check for equivalence between the structures. With the increasing popularity of High-Level Synthesis (HLS) tools, equivalence checking is gaining more importance.

Theorem proving: This is one of the oldest formal verification techniques but still of immense interest to computer science researchers. Theorem proving is used in software and hardware designs where absolutely high confidence in the design is required whatever the scale of the design is. Theorem proving proceeds with the user providing axioms and producing new inference steps using rules of inference in an underlying theory. Essentially, the user guides the prover in searching for the proof by providing intermediate lemmas and hence it requires a proficient user in order to be able to use the theorem prover efficiently. In industry, floating point arithmetic, avionics software, etc., are some of the places where theorem proving is used to establish the correctness of the designs. Some of the well-known theorem provers include Coq, Isabelle, ACL2, HOL, to name a few. Deductive verification techniques are more commonly referred to as Automated Theorem Proving.

8.3 Applications

Formal verification is widely used in software as well as hardware industries. The underlying techniques in the tools fall under one of the categories that we discussed, but the tools and techniques themselves are specialized to solve a specific problem. In the following sections, we cite examples of such specific verification use cases and map them to the techniques discussed earlier.

8.3.1 Software Formal Verification

One of the most common techniques used in software formal verification is Static (code) Analysis. There are several analyses that fall under the umbrella of static code analysis. Essentially the code is analyzed for all possible executions, without explicitly executing them. In an industrial setting, abstraction-based static analysis (abstract interpretation [19]) is used. This usually involves generating an abstract program in a different domain (say Boolean domain) and reason about the correctness of the properties of the abstract program. Commonly used abstraction techniques (like predicate abstraction) are conservative and it has been shown [20] that if the property holds in the abstract program, it entails the correctness of the concrete program. One of the earliest success stories of static analysis in industrial size software came from Astrée Static Analyzer [21] used in the formal verification of safety-critical avionics software at Airbus Industrie. It was soon followed by Microsoft Research in their SLAM [22] project for using Static Analysis to verify

Windows device drivers. Coverity [23] is one of the companies (acquired by Synopsys) whose static analysis tools are widely used by software companies for verification purposes.

Deductive program verification is another technique that is widely employed in the verification of safety-critical software. This technique works by translating the program and its specification into an intermediate representation. The specification corresponds to the properties (assumptions, assertions, loop invariants, variants, etc.) that are usually annotated within the program itself. The technique generates a set of verification conditions which when proven establishes the correctness of the behavior of the original program. The verification conditions themselves are generated using predicate abstraction or other lattice-based techniques. Frama-C [24], KeY [25], ESC Java [26] etc., use this approach. Frama-C which is inspired from Krakatoa [27], combines deductive verification with several program analysis techniques.

Bounded model checking is another technique that is used for software verification purposes. CBMC [18] is one of the tools that does bounded model checking of C programs. There are techniques that combine the precision of model checking with the efficiency of static analyzers. CPA checker [28] is one such tool that combines the techniques and allows the user to configure the system during verification.

8.3.2 Hardware Formal Verification

In hardware, formal verification is most widely used in equivalence checking and verification of floating point arithmetic. Typically, a reference model is written in C or System Verilog or some other high-level programming language and the behavior is tested using simulation. Once the behavior is satisfactory, an RTL (Register Transfer Level) design is coded in one of the hardware description languages (HDL) that uses the high-level description as a reference. Since high-level descriptions (such as a C program) would consume inputs and produce outputs instantaneously, simulation is easy and takes no time. On the other hand, HDL implementations are more close to hardware and take several clock cycles to compute an output from an input. It is therefore imperative to establish the equivalence between the C model (or any high-level description) against the HDL implementation. The tools [29] that enable this typically use Binary Decision Diagrams (BDDs) with several optimizations and bit-vector reasoning engines to accomplish the task. Another common use case is to verify power optimizations in hardware designs, clock gating, for example. Copies of the design, with power optimizations turned on and off are subject to equivalence checking to ensure that the optimizations do not introduce any bugs. Equivalence checking is also used to establish the equivalence between the RTL logic and the generated netlist.

Verification of floating point arithmetic is another area where there is a lot of interest in formal verification tools. Designs of such units are typically too complicated for simulation to cover interesting corner case scenarios and formal verification tools are invaluable in establishing the correctness of such designs. Theorem proving is one of the techniques that is widely used in the verification of floating point arithmetic [30].

Assertion-based verification is another area that is gaining popularity due to the expressive power of System Verilog Assertion language (SVA) and with major hardware formal verification tool vendors supporting a large subset of SVA. Ultimately, it boils down to the understanding of the design and requires a skilled verification engineer to write effective assertions to identify the bugs. The verification results in this case are only as good as the quality of the assertions. Here is where formal specification languages can be of immense help as most of the important properties can be harvested from the specification which is already a formal description of the functionality of the design.

There are many other light-weight formal techniques that are used in both hardware and software industries—such as checking for null pointers, and memory leaks, but due to space constraints we cover only the major techniques. In the following section, we shall identify some upcoming areas where formal verification can make a huge impact—the "thrust" areas for formal methods.

8.4 Thrust Areas

The advancement in technologies like parallel computing has made formal verification techniques more scalable in trying to address the needs of today's scale and complexity. In this section, we will see some of the recent areas where formal methods are already making an impact and is expected to make a positive impact in the coming years.

Data center software: With most of our software moving to the cloud which is just a distributed network of machines, verification of such distributed software is becoming highly important. More critical pieces of software like medical data, or data coming from self-driving cars or air traffic, are expected to reside in the cloud and the reliability of such distributed software is becoming a primary concern. Toward this end, Amazon [9] is using TLA+ to verify the fault-tolerance properties of its DynamoDB which is a replicated "no SQL" data store, distributed over several data centers. A formal specification of the fault-tolerance algorithm was detailed in TLA+ specification language and a distributed version of TLA+ model checker was used to verify the properties.

Mobile applications: A new area of interest would be formal verification of mobile applications. Infer [31] developed by Monoidics (acquired by Facebook) is one of the examples in this space. Infer is a static analysis tool based on Separation Logic [32] and is used to verify mobile apps for null pointer exceptions and memory leaks before being shipped to the app store.

Internet of Things: Another emerging area is Internet of Things (IoT) which is just a network of sensors and microprocessors to process sensor data (previously called “sensor networks”). With IoT expected to be a ubiquitous network especially in healthcare applications, formal verification is seen as a key technology [33] to ensure their reliability. Since IoT networks will have the challenges of distributed computing, SoCs and power efficiency optimizations, it would be an interesting area for researchers of formal methods.

Robotics and intelligent autonomous systems: The recent interest in new applications like intelligent autonomous robots (self-driving cars, drones and rescue robots) has made formal methods an indispensible part of the verification chain. Since most of these devices will have embedded software, the challenges are similar to software verification. Some of the theoretical underpinnings and tool prototypes that employ formal techniques already exist [34, 35] and with the advances in recent parallel computation methods, these techniques could scale to autonomous systems. This could also help in the dual problem of Program Synthesis where a program could be synthesized that does not have any of the behaviors corresponding to faulty scenarios.

Computational biology and drug research: The immense computational power provided by systems today and the advances in big data analytics have renewed research interest in areas such as computational biology. Formal specification analysis techniques and tools help model the behavior of biological systems and analyze them using algorithms [36] rather than in a wet lab, thereby speeding up our understanding and accelerating processes involved in drug research, for example. Some of the examples that have already been researched include study of bacterial growth and study of reactions of enzymes in our physiological systems to chemicals in the drugs. Some recent research in this area [37] is an encouragement that such techniques will be a useful complement to experiments conducted in a biology laboratory.

In the following section, we try and speculate the areas where formal methods are improving and the challenging areas of for future research.

8.5 Future Prospects

The development of formal tools over the last decade can be attributed to the development of several key technologies besides novel verification algorithms themselves. Some of the key recent technologies that have improved formal methods are discussed in this section.

Improved scalability: One of the key ways scalability of formal verification tools is being addressed is by exploiting the use of multi-core processors and distributed computer networks. New data structures like Bdd Array and algorithms for traversing component BDDs in a BDD array in parallel [38] or even a complete parallel BDD library [39] have helped in improving the scalability of the tools that employ such techniques. Further, distributed model checkers like TLA+ [9], model checkers for parameterized systems like Cubicle [40] that make use of libraries like Functory [41] that help in distributing tasks over multiple cores or over multiple machines in a network, handling network fault-tolerance in a seamless manner, demonstrate the efficiency and scalability improvements for application of formal methods in industry.

Machine learning and assertion harvesting: Efficient machine learning implementations have helped analyze large data sets to infer useful information. Since most of the software and hardware would be built from smaller building blocks combined in different configurations (such as number of instances of a memory module or number of objects of a certain class), machine learning techniques could be used to harvest properties to be checked from the historical data of bugs at the interfaces. This would save some manual effort of writing assertions and help identify bugs at an early stage. The design itself would become more modular and easy to maintain.

New applications and newer challenges: Some of the applications like autonomous vehicles might demand fault diagnosis and repair and recovery from the fault in real time, which would in turn require a light-weight formal verification algorithm running on the device or on the cloud and communicating with the device to identify a faulty (or in the case of autonomous vehicles) a dangerous behavior and enable performing course correction. This problem is the same challenge as the verification of autopilot software. Applications like Computational Biology involves big data analytics combined with formal analysis techniques. These new applications pose new challenges and opens up an array of interesting problems for researchers of formal methods.

With several new applications and improved scalability, formal methods could very soon become a predominant verification technique in the coming decade. This requires a lot of effort from the engineering community in being able to change the status-quo of verification. The advantages of formal methods clearly out-weigh the comfort of staying with the status-quo and there is a strong belief among the formal methods community that it will indeed be an invaluable complement, if not a replacement, for existing techniques very soon.

8.6 Conclusions

The increasing complexity of SoC designs and the software algorithms and the distributed form they take, combined with the short release cycles of hardware and software, it is no surprise that formal methods will be a key technology in ensuring reliability of future software and hardware designs. Today's formal tools can handle designs of a very large scale and offer the users several ways to abstract the designs, combine a variety of techniques and more importantly offer a useful graphical user interface so that basic formal techniques, for bug-hunting in the early stages of the design process, can be readily adopted by most engineers, even when not extensively trained in advanced formal methods.

Yet, as outlined in the previous section, there are many interesting problems to be solved by researchers of formal methods. The thrust areas are also some of the areas where technology is changing at a rapid pace and new technical advancements (like machine learning) are some of the areas that formal methods could take a cue or two from, and in turn help make some of these algorithms more robust. Formal methods are well poised to become a dominant verification technique in the years to come.

References

1. Berry G, Gonthier G (1992) The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Programm 19(2):87–152
2. Caspi P, Pilaud D, Halbwachs N, Plaice JA (1987) LUSTRE: a declarative language for real-time programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages, POPL ’87, pp 178–188
3. Abrial J-R, Lee MKO, Neilson DS, Scharbach PN, Ib Holm Sørensen (1991) The B-method. In VDM ’91 formal software development methods, pp 398–405. Springer, Berlin
4. Baudin P, Filliâtre JC, Hubert T, Marche C, Monate B, Moy Y, Prevosto V (2012) ACSL: ANSI/ISO C specification language
5. Lamport L (2002) Specifying systems: The TLA+ language and tools for hardware and software engineers. Addison-Wesley Inc, Longman Publishing Co., Boston, MA
6. Filliatre J-C (2013) One logic to use them all. In: Proceedings of the 24th international conference on automated deduction, CADE ’13, pp 1–20
7. Jackson D (2002) Alloy: a new technology for software modelling. In: Tools and algorithms for the construction and analysis of systems, pp 20–20. Springer
8. Behm P, Benoit P, Faivre A, Meynadier J-M (1999) Météor: a successful application of B in a large project. In: Proceedings of the wold congress on formal methods in the development of computing systems—Volume I–Volume I, FM ’99, pp 369–387
9. Newcombe C, Rath T, Zhang F, Munteanu B, Brooker M, Deardeuff M (2015) How amazon web services uses formal methods. Commun ACM 58(4):66–73
10. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7)
11. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3)
12. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 100(8):677–691
13. Cavada R, Cimatti A, Franzén A, Kalyanasundaram K, Roveri M, Shyamasundar RK (2007) Computing predicate abstractions by integrating BDDs and SMT solvers. In: Formal methods in computer-aided design, 7th international conference, FMCAD 2007, Austin, Texas, pp 69–76, 11–14 Nov 2007
14. Manna Z, Zarba CG (2002) Combining decision procedures. In: Formal methods at the crossroads. From Panacea to Foundational Support, 10th anniversary colloquium of UNU/IIST, the international institute for software technology of The United Nations University, Lisbon, Portugal, 18–20 March 2002, Revised Papers, pp 381–422
15. Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Tools and algorithms for construction and analysis of systems TACAS ’99, pp 193–207
16. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NuSMV: a new symbolic model verifier. In: Computer aided verification, 11th international conference, CAV ’99, Trento, Italy, pp 495–499, 6–10 July 1999
17. Holzmann GJ, Peled D (1996) The state of SPIN. In: Computer Aided Verification, pp 383–389. Springer
18. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Tools and algorithms for the construction and analysis of systems, pp 168–176. Springer

19. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp 238–252
20. Graf S, Hassen Saidi S (1997) Construction of abstract state graphs with PVS. In: CAV, pp 72–83
21. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2002) Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen T, Schmidt DA, Sudborough IH (eds) The essence of computation: complexity, analysis, transformation. Essays dedicated to Neil D. Jones, vol 2566. LNCS, pp 85–108. Springer-Verlag
22. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. SIGPLAN Not 37(1):1–3
23. Bessey Al, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, McPeak S, Engler D (2010) A few billion lines of code later: using static analysis to find bugs in the real World. Commun ACM 53(2):66–75
24. Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B, Frama-C: a software analysis perspective, vol 27, pp 573–609. Springer
25. Weiß B (2009) Predicate abstraction in a program logic calculus. In Leuschel M, Wehrheim H (eds) Proceedings, 7th international conference on integrated formal methods (iFM 2009), vol 5423 of LNCS, pp 136–150. Springer
26. Flanagan C, Qadeer S (2002) Predicate abstraction for software verification. In: POPL '02: proceedings of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages, pp 191–202
27. Filliâtre J-C, Claude Marché (2007) The Why/Krakatoa/caduceus platform for deductive program verification. In: Computer aided verification, pp 173–177. Springer
28. Beyer D, Henzinger TA, Théoduloz G (2007) Configurable software verification: concretizing the convergence of model checking and program analysis. In: Proceedings of the 19th international conference on computer aided verification, CAV '07. Springer-Verlag, Berlin, pp 504–518
29. Koelbl A, Jacoby R, Jain H, Pixley C (2009) Solver technology for system-level to rtl equivalence checking. In: Design, automation & test in Europe conference and exhibition. DATE '09., pp 196–201. IEEE
30. Harrison J (1995) Floating point verification in HOL. In: Higher order logic theorem proving and its applications, pp 186–199. Springer
31. Calcagno C, Distefano D, Dubreil J, Gabi D, Hooimeijer P, Luca M, OHearn P, Papakonstantinou I, Purbrick J, Rodriguez D (2015) Moving fast with software verification. In: NASA formal methods. Springer, pp 3–11
32. O'Hearn PW (2012) A primer on separation logic (and automatic program verification and analysis)
33. How to Cut Verification Costs for IoT. http://semiengineering.com/how-to-cut-verification-costs-for-iot/. 2014
34. Bertoli P, Cimatti A, Roveri M, Traverso P (2006) Strong planning under partial observability. Artif Intell 170(4):337–384
35. Cimatti A, Roveri M (1999) Conformant planning via model checking. In: Biundo S, Fox M (eds) Recent advances in AI planning, 5th european conference on planning, ECP '99, Durham, Springer, pp 21–34, 8–10 Sept 1999
36. Bartocci E, Lío P (2016) Computational modeling, formal analysis, and tools for systems biology. PLoS Comput Biol 12(1)
37. Wang Q, Miskov-Zivanov N, Telmer C, Clarke EM (2015) Formal analysis provides parameters for guiding hyperoxidation in Bacteria using phototoxic proteins. In: Proceedings of the 25th edition on Great Lakes symposium on VLSI, GLSVLSI '15, pp 315–320
38. Cimatti A, Franzen A, Griggio A, Kalyanasundaram K, Roveri M (2010) Tighter integration of BDDs and SMT for predicate abstraction. In: Proceedings of the conference on design, automation and test in Europe, DATE '10, pp 1707–1712

39. Van Dijk T, Laarman A, Van De Pol J (2013) Multi-core BDD operations for symbolic reachability, electron. Notes Theor Comput Sci 296:127–143
40. Conchon S, Goel A, Krstić S, Mebsout A, Zaïdi F (2012) Cubicle: a parallel SMT-based model checker for parameterized systems. In: Computer aided verification, pp 718–724. Springer
41. Filliatre J-C, Kalyanasundaram K (2011) Functory: a distributed computing library for objective Caml. In: Trends in functional programming, pp 65–81. Springer

Chapter 9
Design Fault Identification in MBD for Safety Critical Systems

Benkmann Ruben, Gourish Kumbar and S. Mouneshwar

Abstract This paper provides a novel method of identifying a set of design faults in Simulink\TargetLink models and production code generation well before generating the test cases. These models can be used for development of safety critical systems. We have developed stand-alone application which transforms TargetLink model to EA compatible UML (Unified Modeling Language) model represented as packages and composite structure diagrams. The entire process makes use of dSPACE and MathWorks tools.

Keywords Design Fault · UML · Model Based Development · Simulink V model

9.1 Introduction

Industries such as aerospace and automotive applications are developing more products using MBD (model-based development) because of its demands on high-quality products, critical safety requirements, and reduced time to market. Virtual model of many systems can be conveniently created using MDB, and it can be simulated to arrive at right design space before production.

B. Ruben
Robert Bosch Automotive Steering GmbH, Schwäbisch Gmünd, Germany

G. Kumbar (✉)
RBEI\ETC1, Bangalore, India
e-mail: Gourish.kumbar@in.bosch.com

S. Mouneshwar
RBEI\ETB3, Bangalore, India
e-mail: MouneshwarShivasharana.Wadikar@in.bosch.com

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_9

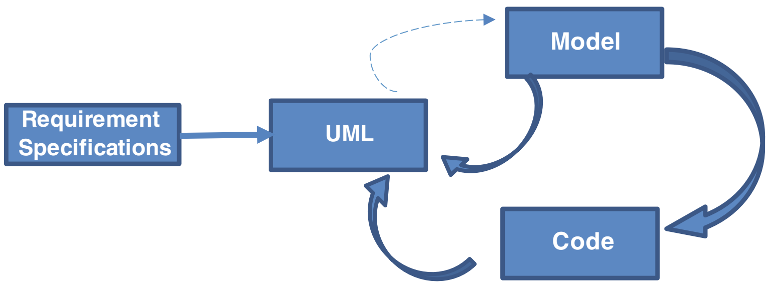


Fig. 9.1 Overview of fault identification process

MBD allows us in many ways to have better development life cycle through the following:

1. Better product quality,
2. Achieve development of functions that represents complex system design
3. Reduce the development time.

The Unified Modeling Language (UML, [1]) offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context.

UML can be used as a formal design technique for the development of critical systems [2].

In our approach, we try to combine MBD and UML representations with some kind of transformation between the two.

From the requirement, we create the architecture in UML using EA. The UML created is used as reference architecture to create Simulink model. We perform transformation of Simulink model to UML internally to verify the consistency with reference UML (architecture). We find the faults of the created Simulink model. This is depicted in Fig. 9.1.

"TargetLink-To-EA" is the tool that performs all the transformations, and it is a windows-based application to visualize the model into UML diagram. All the UML representations can be opened in the EA or can be stored as image (for documentation purpose).

9.2 Workflow

Figure 9.2 shows the typical V-model for applications that use MBD. We concentrate basically on the modeling (develop models) and the auto-code generation. All the requirements are transformed into UML. Design team develops the behavior of each model according to the requirement but retaining the structure of the UML (golden reference) architecture. All processes are accomplished without making use of any test cases. Considering 1 and 2 paths in the V-model, these are only for

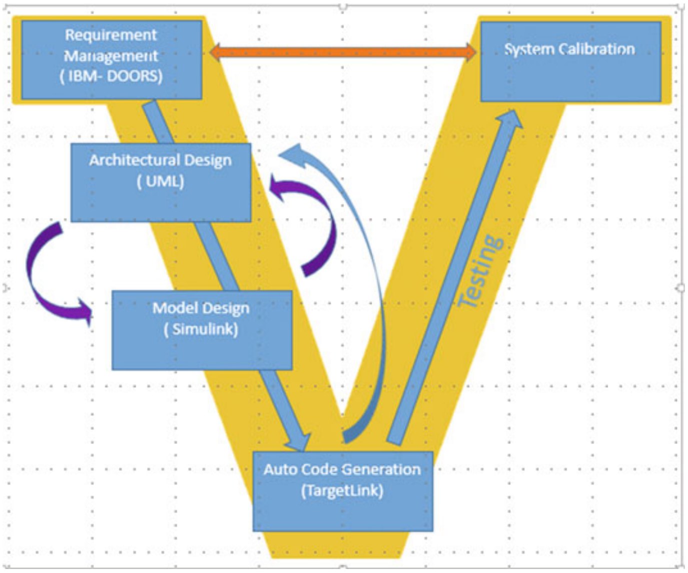

Requirement Management (IBM- DOORS)
System Calibration
Architectural Design (UML)
Model Design (Simulink)
Testing
Auto Code Generation (TargetLink)

Fig. 9.2 V-model

validation stages to uncover certain faults and not the violation of the V-model principles itself.

Design faults are identified at two stages, and only specific set of faults can be uncovered. Test case generation is still necessary to uncover many design and requirement mismatches.

We classify all the faults based on the severity and position of the faults in the model. For example, they are classified as L1 and L2 faults, implying L1 is the fault in first-level hierarchy of the model and the severity is minor. We try to restrict the design based on the thumb rule that the hierarchy does not go beyond three levels.

Under certain scenarios, architectural design in UML needs some changes. These changes are done only after rigorous analysis.

Stage I: As depicted in the data flow diagram in Fig. 9.3, requirement specification is maintained in SCM tool such as DOORS to better capture, trace, analyze, and manage changes to requirements and helps you to demonstrate compliance to regulations and standards.

Stage II: Next level of implementation is done using UML modeling where required functionality is described at abstract level.

Stage III: From the UML modeling, the model-level functionality is simulated, and validation and verification are done against the expected\desired results.

Stage IV: Once simulation results are validated and verified, then next level is the c code generation in Generic ANSI C\AUTOSAR standard.

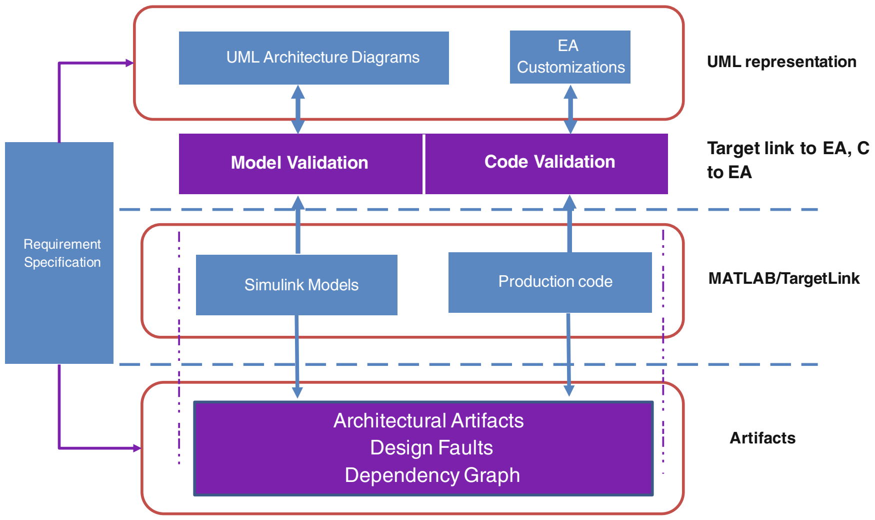


Fig. 9.3 Data flow

9.3 Validation Against Model and C Code Generation

The simulated model is transformed into XML which is compatible to EA using TargetLink-To-EA. Table 9.1 shows the structural mapping from model to UML. During the transformation from model to UML diagram, the root element of the subsystem is represented as follows:

- Blackbox view—depicting inputs and outputs and
- Internal dependency diagram—depicted the data flow from input to output and its subsystems.

Table 9.1 Structural mapping

TargetLink representation	EA representation
Inport\outport	Port
Subsystem at top level	Composite structure diagram
Line	Dependency
Subsystem function	Customized stereotypes
Annotation block	Notes
Outport argument properties	Invariants

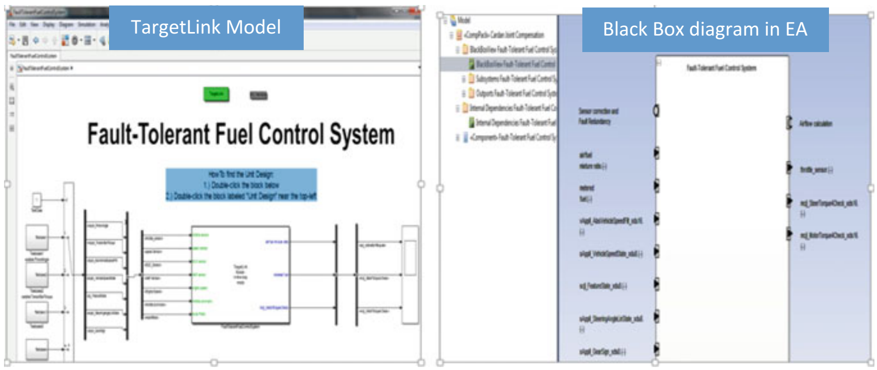


Fig. 9.4 Blackbox representation of fault-tolerant fuel control system

- Subsystems which are called from main subsystem are represented as ports with different stereotypes with unique representation as class in the composite structure diagram:
  - Calling subsystem on the left-hand side of the diagram and
  - Called subsystem on the right-hand side of the diagram.

An example of transforming TargetLink model to UML diagram by our "**TargetLink-To-EA**" application is shown in Fig. 9.4. We have applied our tool to a number of TargetLink models such as a fault-tolerant fuel control system, "Voice_Coil_Flexible_Positioner" provided by the dSPACE [3].

Design faults that can be uncovered using the approach are classified as hierarchical mismatch, false paths, false interfaces (data path defects), control logic defects, and many more.

9.4 Results

This concept is applied in implementation of fault-tolerant fuel control system. Results show consistency in architectural design from requirement till code generation and do not impact the development time. We were able to find faults such as unused interfaces and flow paths.

The methodology also allows us to categorize the faults and improve the error handling. Figure 9.4 shows the blackbox representation, and Fig. 9.5 shows the internal dependency diagram of the fault-tolerant fuel control system transformed using TargetLink-To-EA.

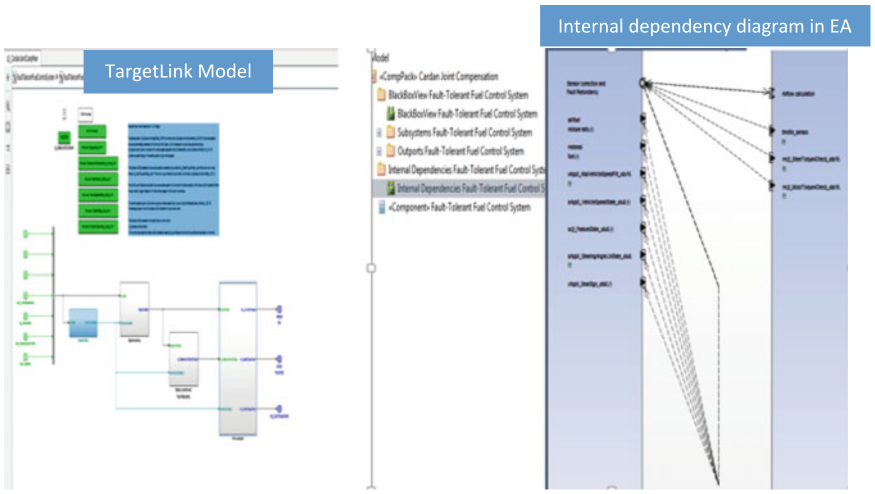

TargetLink Model
Internal dependency diagram in EA
Model
«CompPack» Cardan Joint Compensation
BlackBoxView Fault-Tolerant Fuel Control System
BlackBoxView Fault-Tolerant Fuel Control System
Subsystems Fault-Tolerant Fuel Control System
Outputs Fault-Tolerant Fuel Control System
Internal Dependencies Fault-Tolerant Fuel Control System
Internal Dependencies Fault-Tolerant Fuel Control System
«Component» Fault-Tolerant Fuel Control System

Fig. 9.5 Internal dependency diagram of fault-tolerant fuel control system Subsystem

9.5 Conclusion

Our approach finds the design flaws at early stages such asmodel development and code generation. It identifies various defects at design stage that are generally visible only after testing through rigorous test cases.

It provides the consistency in design architecture from UML design to code generation and testing. There is a need of connected tool chain of specific tools for UML and modeling\code generation to accomplish this concept.

References

1. Object Management Group. OMG Unified Modeling Language Specification v1.5: Revisions and recommendations, Mar 2003. Version 1.5. OMG Document formal/03-03-01
2. Jürjens J, Grünbauer J (2003) Critical systems development with UML: overview with automotive case-study. In: Software and systems engineering, TU Munich, Germany
3. https://www.dspace.com

Chapter 10
Formal Methods Workflow for Model-Based Development

Gaurav Dubey and Manoj G. Dixit

Abstract To manage the rapid growth and complexity of software, model-based development workflow is being increasingly adapted in industry. This paper provides a brief overview of some of the important verification and validation steps in this workflow and illustrates them using Simulink Design Verifier tool of MathWorks.

Keywords Model-based development · Verification and validation (V&V) Functional verification · Structural coverage analysis · Simulink Design Verifier (SlDV)

10.1 Introduction

Modern automotive and aerospace-embedded control applications consist of large number of interacting software components often in a distributed environment. The software components are mapped onto a network of computational nodes, and they interact by exchanging messages. The applications span multiple domains: from event based to continuous control. The safety critical applications are given extra attention during verification and validation phase to ensure that desired intent of functionality and timing is met. This makes the development cycles usually long and iterative. To manage the complexity, emphasis is given to follow a systematic methodology based on component-based development (CBD).

CBD methodology often advocates decomposition of system into a set of reusable components by identifying their interfaces. The integration problem is then reduced to interface compatibility. Autosar [1] is an example of standardization effort in automotive domain for CBD. It provides a specification language to document components, their interactions, and mapping to underlying distributed architecture. Standards such as ISO 26262 [2] and DO-178B [3] have given a

G. Dubey (✉) · M.G. Dixit
MathWorks India Pvt. Ltd., Bangalore, India
e-mail: Gaurav.Dubey@mathworks.in

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_10

special emphasis on verification and validation requirements through classification of applications based on safety critical level. There is also a significant drive toward identifying issues much earlier in the development phase. A big thrust in this direction is toward model-based development (MBD). MBD advocates iterative development through the construction of executable specifications of different levels of granularity. MBD also has an additional advantage in V&V step, and it helps in detecting issues early enough which would have otherwise taken a long time. Simulation is a widely used method to check whether the implemented design is able to meet the requirements intent or not. However, with ever-increasing feature complexity and feature interactions, there is push toward using more rigorous methods based on formal verification [4] to give better guarantees of feature functionality.

Simulink-/Stateflow-based [5] toolchain from MathWorks provides an ideal platform for MBD of embedded controllers. Simulink/Stateflow language provides a modeling framework for control design using a set of predefined library of graphical blocks. The control design can be done at various levels of abstraction: from closed-loop continuous control models to discrete control algorithm ready to be deployed in microcontrollers. User can verify correctness of their designs using software in loop or processor in loop simulation mode. Simulink Design Verifier (SLDV) [6] tool from MathWorks supports multiple advanced V&V features for MBD. The tool contains a formal verification engine to efficiently analyze design models in Simulink/Stateflow. This paper presents an early stage V&V workflow suitable for MBD and demonstrates it using SLDV.

The rest of the paper is organized as follows: Sect. 10.2 gives an overview of MBD workflow for V&V, Sect. 10.3 gives an overview of SLDV analysis engine and describes how various steps in the above workflow can be achieved using the tool.

10.2 V&V Workflow for MBD

In the well-accepted and widely used V-cycle [7] approach of feature development, the requirement elicitation, functional decomposition, and unit-level component development are important steps. The functional decomposition step involves identifying different functional units and their requirements to achieve system-level intent, whereas component development involves implementing functional units as separate components. In MBD workflow, all these steps are iterative and Simulink/Stateflow is a widely used tool in coming up with various design choices at each stage before selecting a suitable one. V&V activities are carried out at each substep to catch defects as early as possible. These defects vary from functional issues to modeling error. Figure 10.1 shows some of the important V&V steps and the cost associated with them. This document focuses on the latter four steps and describes them below in detail:

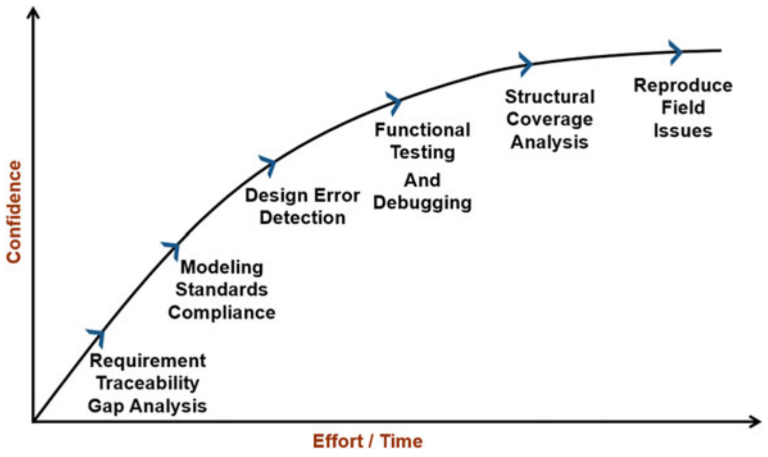


Fig. 10.1 Important steps in V&V and cost metric

1. **Detecting early design errors**: There are two main classes here: logic errors and run-time errors. The former class is about identifying whether there are any logical inconsistencies in the design that makes some part of the design dead; i.e., it can never execute under any input combination. This often may involve complex dependencies between various control conditions in the design model. The errors in the latter class can be encountered during some simulation run of the design. For example, certain evolution of controller renders a denominator in a division operation to have value 0. Thus, model involves division-by-zero error. We argue that it is often useful to remove these errors before moving ahead with rigorous steps of functional testing.
2. **Functional verification**: This step checks whether the design model is meeting the desired intent specified in the requirements. Writing functional test cases based on requirements documents is often the standard way of achieving this step. To ensure better quality and improved guarantees from critical applications, there is a big push toward using rigorous methods such as formal verification to verify whether design meets given intent under *all* situations. Formally capturing design intent is a big challenge in this problem.
3. **Structural coverage analysis**: This step checks, among other things, whether all branching decisions in the design model are exercised or not. More rigorous checks involve whether each condition in the decision separately affects the decision or not. This check is called modified condition decision coverage (MCDC) [8] check. Often this gives high confidence about the design. For MBD, this step is recommended by standards such as ISO26262.
4. **Defect localization**: As the design complexity increases, localizing defects to specific part of the design is difficult. Often there are indirect dependencies that

are difficult to figure out through manual inspection. Another related problem is of model comprehension. As model complexity increases, it is becoming very essential to build tools and methods that slice the design model to show the part of the design model relevant for a particular scenario.

10.3 V&V Workflow Using SLDV

Figure 10.2 is a high-level overview of architecture of SLDV analysis engine. The tool takes a design model in Simulink/Stateflow as input and verification conditions as input and performs analysis using various formal method-based tools. The verification conditions can be user given, e.g., proof objective or they can automatically detected by tool, e.g., check whether control input of switch block is toggled in both ways. SLDV supports primarily four types of analysis workflows: automatic test generation, design error detection, property proving, and defect localization. Intuitively, test generation analysis is used for structural coverage of design models, design error detection involves identifying dead logic and run-time errors, property proving involves formal verification of functional requirements, and defect localization involves slicing the design model to compute a subpart of the design that affects specific output. We will cover each of these workflows in detail in the subsequent sections.

As shown in Fig. 10.1, the tool accepts control designs consisting of various blocksets from Simulink library, Stateflow, MATLAB, and even C/C++.

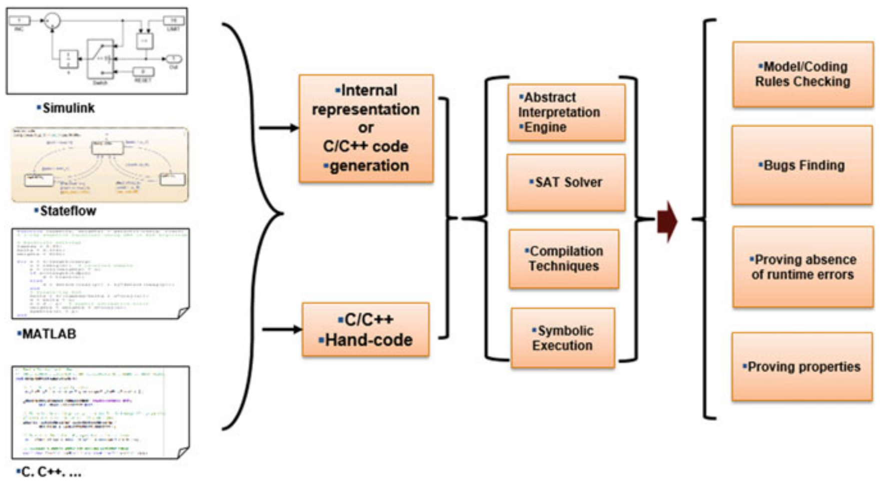

•Simulink
•Stateflow
•MATLAB
•C. C++. ...
•Internal representation or C/C++ code •generation
•C/C++ •Hand-code
•Abstract Interpretation •Engine
•SAT Solver
•Compilation Techniques
•Symbolic Execution
•Model/Coding Rules Checking
•Bugs Finding
•Proving absence of runtime errors
•Proving properties

Fig. 10.2 SLDV analysis engine architecture

The first step involves constructing an equivalent internal representation of the input design model which is suitable for analysis. This step also involves insertion of verification conditions to be analyzed on the design model. Often this is based on the modeling elements used in the given design. For example, if the design contains an arithmetic computation involving division, the tool automatically inserts verification condition to check for division-by-zero. The internal representation with verification conditions is given as input to formal engines for analysis. The tool uses variety of formal engines for analysis. Each engine processes the input representation based on the method they implement. Below, we give overview of some of these techniques.

Abstract interpretation engine is used to check run-time errors. Abstract interpretation [9] is a well-known methodology where input programs which contain complex arithmetic/computations and verification conditions are transformed into programs and verification conditions in another abstract domain for analysis. The abstract domains are often simpler and scalable to analyze, and the transformation is sound. For example, if there is no design error for division-by-zero in abstract domain, then so is true in concrete domain.

SAT solvers implement methods to find a solution to satisfy predicate defined in appropriate theory. For example, SAT solving for a Boolean formula involves finding a valuation that makes the formula satisfiable. The more general theories, e.g., SMT [10], for rational/real arithmetic define proof system to find a solution for a predicate in that domain. Symbolic execution [11] is another well-known technique to study multiple behaviors of program simultaneously. Symbolic execution engine computes a symbolic formula representing a set of behavior in the given input model. This symbolic formula combined with the desired verification condition can be given as an input to a SAT solver to find a system behavior that satisfies given condition.

Model checkers take given representation and error property to be checked as input and formally prove absence of errors: If the property is valid, then there is NO behavior of the input design that can negate the property. Another strong feature of model checkers is to demonstrate property violation, if any using a counterexample trace. SLDV uses model checking engine to formally analyse safety properties for the given design.

Program slicing [12] is a well-accepted method to reduce the program complexity by removing the part of the program that does not affect certain criteria, e.g., output variable. SLDV dependency analysis engine achieves this at model level. Specifically, given a signal/block under consideration, it finds out all part of the design that affects the value of the signal during some time step.

Below, we describe how SLDV analysis engine can be used in achieving some of the steps in V&V workflow given in Sect. 10.2.

10.3.1 V&V Workflow Details

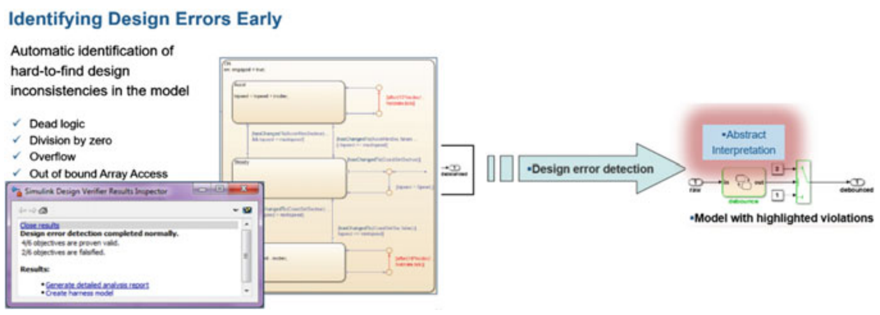
10.3.1.1 Detecting Early Design Errors

SLDV can be used for statically detect run-time errors and dead logic and can derive design ranges. Design errors detected include dead logic, integer overflow, division-by-zero, and violations of design properties and assertions. The tool additionally highlights blocks in a model containing design errors and blocks proven to be without them. For each block with an error, it also computes signal-range boundaries and generates a test vector that reproduces the error in simulation. Figure 10.3 shows snapshot of the results produced by tool for design error check.

10.3.1.2 Functional Verification

Functional verification involves authoring functional test cases and proving properties on design model. Simulink Test is a tool from MathWorks that provides blocksets for authoring functional test cases. It includes a test sequence block that lets you construct complex test sequences based on functional scenarios and include assessments for them. The assessment criteria include absolute/relative tolerances, limits, logical checks, and temporal conditions. The tool additionally helps in managing and executing tests in various environments. Figure 10.4 shows snapshot of a simulation testing workflow using Simulink Test tool.

SLDV includes a model checker using which functional properties authored using Simulink blocksets can be formally verified on the design model. For this tool includes special temporal blocks such as detector, extender to model properties. If the property is falsified, the tool generates a counterexample to demonstrate the violation scenario.


Identifying Design Errors Early
Automatic identification of hard-to-find design inconsistencies in the model
✓ Dead logic
✓ Division by zero
✓ Overflow
✓ Out of bound Array Access
Simulink Design Verifier Results Inspector
•Design error detection
•Abstract Interpretation
•Model with highlighted violations

Close results
Design error detection completed normally.
4/6 objectives are proven valid.
2/6 objectives are falsified.

Results:

- Generate detailed analysis report
- Create harness model

Fig. 10.3 Design error detection using SLDV

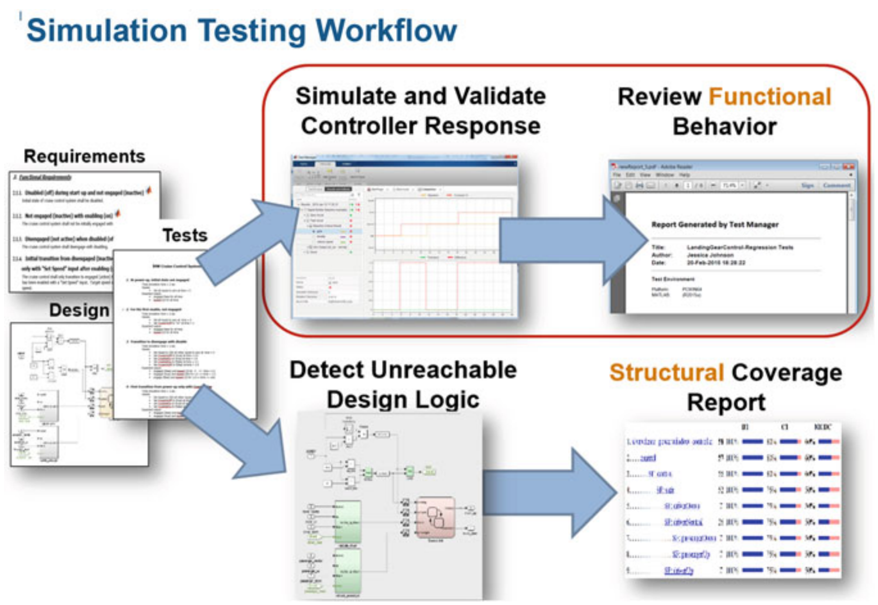

Simulation Testing Workflow
Requirements
Tests
Design
Simulate and Validate Controller Response
Review Functional Behavior
Report Generated by Test Manager
Detect Unreachable Design Logic
Structural Coverage Report

Fig. 10.4 Simulation testing using Simulink Test

10.3.1.3 Structural Coverage Analysis

Simulink verification and validation tool from MathWorks measures model coverage to indicate untested elements in the given design. For example, it checks for status of various logical conditions and switch positions during simulation. The tool highlights the elements that are not covered by given set of simulation runs. Based on this, users may modify the requirements, test cases, or design to meet your coverage goals. Figure 10.5 shows snapshot of workflow using coverage measurement. Automatic test generation engine of SLDV can be used to generate test cases to compute missing coverage.

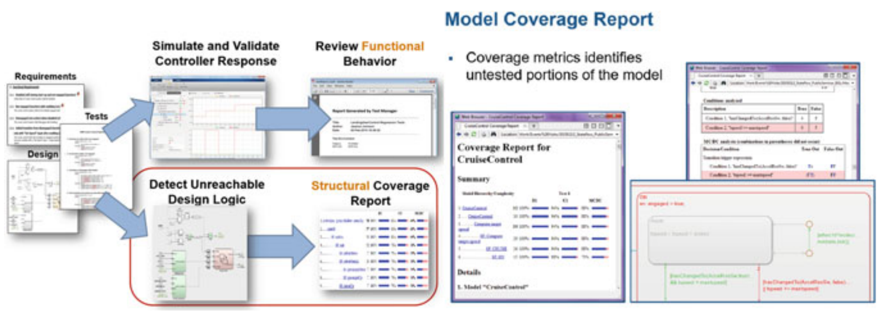

Model Coverage Report
Coverage metrics identifies untested portions of the model
Requirements
Tests
Design
Simulate and Validate Controller Response
Review Functional Behavior
Detect Unreachable Design Logic
Structural Coverage Report
Coverage Report for CruiseControl
Summary
Details
1. Model "CruiseControl"

Fig. 10.5 Coverage measurement using Simulink verification and validation tool

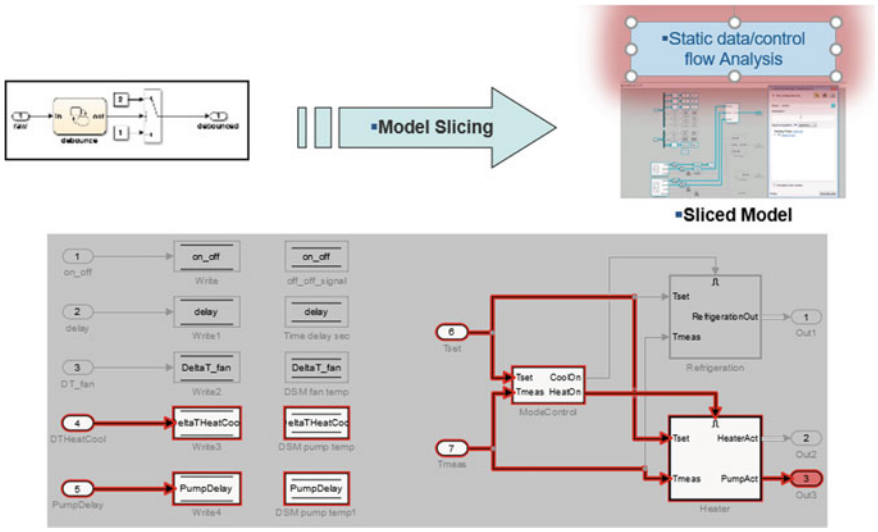


Fig. 10.6 Slicer tool workflow

10.3.1.4 Defect Localization

Large models often contain many levels of hierarchy, complicated signals, and complex mode logic. Model slicer tool within SLDV implements a dependency analysis engine to determine the interdependencies of blocks, signals, and model components throughout a model. The slicer can be used for better understand functional dependencies in large or complex models, where determining dependencies can be a lengthy process (Fig. 10.6).

10.4 Conclusion

In this paper, we presented a formal method-based workflow for MBD. The workflow was described using SLDV tool of MathWorks. The tool implements advanced analysis engine based on formal methods techniques. The paper focused on four key steps in V&V analysis workflow and gave overview of how these steps can be carried out using SLDV tool.

References

1. AUTOSAR Consortium. (http://www.autosar.org)
2. ISO26262. (http://www.iso.org/iso/catalogue_detail?csnumber=43464)
3. DO178B. (https://en.wikipedia.org/wiki/DO-178B)
4. Clarke EM et al (1999) Model checking. MIT Press
5. Simulink/Stateflow. (http://www.mathworks.com)
6. Simulink Design Verifier. (http://www.mathworks.com/products/sldesignverifier)
7. V-Model (Software Development). (https://en.wikipedia.org/wiki/V-Model_(software_development))
8. Hayhurst KJ et al (2001) A practical tutorial on modified condition/decision coverage. Tutorial, NASA
9. Cousot P et al (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. POPL, pp 238–252
10. SMT. (https://en.wikipedia.org/wiki/Satisfiability_modulo_theories)
11. Godefroid P et al (2005) DART: directed automatic random testing. In: PLDI 2005, pp 213–223
12. Weiser M (1981) Program slicing. In: ICSE 1981, pp 439–449

Chapter 11
Race That Never Ends!

B.M. Parinitha, Manupriya Srivastava and Chitra Viswanathan

Abstract In modern computer systems, "concurrent programming" is a never-ending necessity to boost processing speed, thereby achieving the best possible performance. The major drawback of concurrent programming is that it introduces more errors (that are difficult to be identified) than sequential programming, and for this purpose, it is very essential to have tools that can discover such errors. In order to decide the appropriate tool, one must learn about the strengths and weaknesses of the available dynamic analysis tools and then choose the most appropriate. A study has been carried out to assess which among the current dynamic analysis tools cater to detecting these race conditions. However, the study reveals poor detection capability by the tools; hence, prevention by coding standards is imperative. As the saying goes, there are "No silver bullets," to race condition issues. They have been and continue to be around us. The only law that must be followed is: "Proper ordering maintained between operating system and program order."

Keywords Dynamic analysis · Race detection · Juliet Testsuite Valgrind · Parasoft Insure++ · Secure coding guidelines

B.M. Parinitha (✉) · M. Srivastava · C. Viswanathan
Centre for Artificial Intelligence and Robotics (CAIR), DRDO,
C.V Raman Nagar, Bangalore 560093, India
e-mail: parinitha@cair.drdo.in

M. Srivastava
e-mail: mpriya@cair.drdo.in

C. Viswanathan
e-mail: chitrav@cair.drdo.in

M. Nanda and Y. Jeppu (eds.), *Formal Methods for Safety and Security*,
https://doi.org/10.1007/978-981-10-4121-1_11

11.1 Introduction

Usually, C and C++ programs read and write to files as part of their normal operations. Irregularities in how these programs interact with the file system operations, which are defined by the underlying operating system, lead to race conditions. Concurrent programming is difficult and the quality of programmers is growing increasingly suspect, specifically in languages like C and C++ that are low-level and offer little defense against general programming errors. As both software and hardware systems grow gradually more complex, programmers need additional help, and for this reason, it is very essential to have tools that can identify such errors. These tools can also be used to improve program quality, particularly correctness and speed. Many such tools use some type of program analysis to determine remarkable information regarding programs. To achieve insight about optimizing, debugging, extending, and refactoring large systems, programmers frequently rely on dynamic program analysis tools, which monitors a program under execution and report properties of that execution.

11.1.1 Terminology

Weakness: Software weaknesses [1] are flaws, faults, bugs, and other errors in software implementation/design, and if unaddressed, it could effect the systems and networks being vulnerable to attack, e.g., buffer overflows.

Vulnerability: A software vulnerability [1] is defined by National Institute of Science and Technology (NIST) as the property of system security requirements, design, implementation, or operation that might be accidentally triggered or intentionally exploited leading to security failure. A vulnerability is an error in software that could be used by a hacker to gain access to a system or network.

Exposure: An exposure is a system configuration issue or a error in software that allows access to information or capabilities that can be used by a hacker as a first step to gain access into a system or network.

Overview of CWE: Common Weakness Enumeration (CWE) [1] is a formal list or dictionary of general software weaknesses that can occur in software's architecture, design, or implementation that can lead to exploitable security vulnerabilities. CWE was generated to serve as a common language for describing software security weaknesses; serve as a standard measuring stick for software security tools targeting these weaknesses; and offer a common baseline standard for identifying, mitigating, and preventing weaknesses.

11.1.2 *Literature Survey*

In 2005, "Seven Pernicious Kingdoms" [2] taxonomy was presented by Tsipenyuk et al. They classify vulnerabilities into seven categories, which they call kingdoms. Among the seven kingdoms is "Time and State." Programmers would like to assume that their code is being executed in an orderly, uninterrupted, and linear fashion. Multitasking OSs running on multicore, multi-CPU machines do not play by these rules—they juggle between multiple users and threads of control. This results in TOCTOU vulnerabilities. Under this category, the following vulnerabilities are included:

1. Race condition within a thread,
2. Signal handler race condition, and
3. Time-Of-Check-Time-Of-Use (TOCTOU).

It is hard to understand and debug multithreaded programs. There are several static and dynamic techniques and tools that have been developed to automatically find data races in multithreaded programs. Due to the complexity of model checking techniques, their analyses are not scalable, but they are capable of handling synchronization issues. Dynamic data race analysis techniques are either lockset based or happens-before based or a combination of both. In the technical report by Harrington and Freund [3], they improve the performance of dynamic race detectors by skipping the access checks made on thread-local objects that are ensured to be race free using dynamic escape analysis with RoadRunner, FastTrack dynamic race detector, and WALA static analysis framework. A dynamic data race detector—ThreadSanitizer—is presented in the paper [4]; by creating dynamic annotations, performance evaluation of tool is carried out using chromium (open-source browser project) and is compared with Helgrind and Memcheck 3.5.0 of Valgrind tool suite performance. A policy to declare and enforce thread safety in C/C++ programs using annotations is described in the paper [5], implemented as a compiler warning currently. In paper [6], Kahlon et al. have present a new shared variable detection technique in concurrent programs (mainly focused on Linux device drivers) for static data race detection. Savage et al. [7] have described a tool—Eraser with binary rewriting techniques—to examine every shared-memory reference and confirm that consistent locking behavior is observed. The combination of happens-before and lockset approaches is covered in the paper [8], a hybrid dynamic data race detector for Java programs. Data races are detected by lightweight detector called LiteRace [9] that samples and analyzes only selected portions of a program's execution by using an effective sampling algorithm and reducing the dynamic data race detection runtime overhead. In paper [10], Pozniansky and Schuster have presented MultiRace, a novel testing tool, which combines two very powerful techniques for on-the-fly detection of apparent data races. FastTrack [11], race detection algorithm, achieves better performance in comparison with other existing algorithms by less information tracking.

There is no literature on assessment of dynamic analysis tools in C and C++ languages, whereas a report by Spathoulas [12] contains assessment of tools on concurrent programming in Java.

11.1.3 Purpose of This Study

In order to analyze software defects, numerous commercial, open-source, and research tools are developed and deployed in the field of dynamic program analysis. It is unfortunate that there is less public information about the experimental assessment of these tools in terms of accuracy and seriousness of the warnings they report. Furthermore, commercial tools are not only expensive, but also come with license agreements that forbid the publication of any experimental or evaluative data. The central idea behind the study is to evaluate and compare available dynamic analysis tools with respect to their capabilities in detecting software weaknesses by using appropriate metrics. Moreover, a theoretical and experimental analysis has been performed on two dynamic analysis tools that are used for detecting a variety of C/C++ bugs. The tools are evaluated based on their ability to detect specific CWEs related to race conditions. For the evaluation, a test suite (Juliet test suite) is used, containing C/C++ programs with concurrency bugs, and the reports are analyzed. In current literature, no assessment of dynamic analysis tools with respect to detection of race condition issues is available. A tool may be strong in finding some vulnerabilities, while it may be weak in others. This study is aimed at determining the detection capabilities of dynamic analysis tools with respect to dynamic race detection. This knowledge will enable us, in the long run, to establish a Dynamic Analysis Framework (DAF) which will combine multiple dynamic analysis tools with complementary strengths to give better results.

11.2 Dynamic Race Detection

11.2.1 What Is Concurrency?

Concurrency is the idea of managing access to shared resources correctly and efficiently. Figure 11.1 depicts multiple tasks accessing a shared resource. In real life, several things such as bank transactions, traffic movements, and office tasks are happening concurrently. Concurrency happens when two or more separate execution flows are able to run simultaneously. Examples of independent execution flows include the following:

1. Threads: A process may contain multiple threads of execution. Threads are lightweight processes that exist within a process and are usually managed by the operating system.

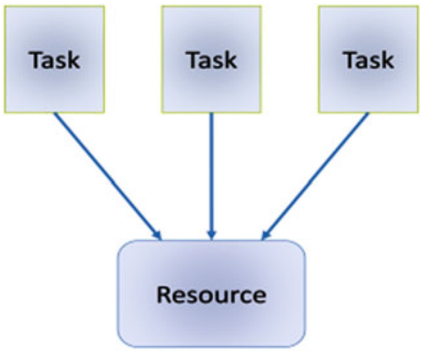


Fig. 11.1 Concurrency

2. Processes: A process is an instance of a computer program that is being executed in their own address space. It contains the program's execution environment.
3. Tasks: Tasks are lightweight and are usually provided by the language runtime.

Race condition (RC) can be caused due to the following flow variants:

1. Trusted flow: This involves interaction of threads within a multithreaded process.
2. Untrusted flow: This involves either "vulnerable software" or exists "outside of trusted software."

Example: File-related vulnerabilities, Symlink vulnerabilities, and Temporary files.

11.2.2 Properties for Race Conditions

Race conditions result due to runtime environments, including operating systems that must control access to shared resources, especially through process scheduling.

Three properties are essential for a race condition to exist:

1. Concurrency property: There should be at least two control flows executing concurrently.
2. Shared object property: A common race object must be accessed by both of the concurrent flows.
3. Change state property: At least one of the control flows must modify the state of the race object.

Race window: This is a code segment that accesses the race object in a manner that opens a window of opportunity for race condition. Race conditions from independent processes cannot be resolved by synchronization primitives. These kinds of concurrent control flows can be synchronized using a file as a lock.

Lock: This is a synchronization object that is either accessible or owned by a thread. There are two types of lock operations: lock() and unlock(). A lock can only be unlocked by its current owner. The lock() operation is blocking if the lock is owned by another thread. A lock file is used as a proxy for the lock. If the file exists, the lock is captured or else the lock is released. For windows, mutexes are used for synchronization. The operations defined on mutexes are create mutex() and release mutex().

11.2.3 Time-Of-Check-Time-Of-Use (TOCTOU)

TOCTOU race condition happens when, between the time a given resource is checked, and the time that resource is used, a change occurs in the resource to invalidate the results of the check. TOCTOU race conditions can happen during file I/O operations. In a preemptively multitasked environment, anything could happen in between the execution of two assembly code operations. Conditions may change in between, and the check becomes invalid.

Consequences:

- Access control: Access to unauthorized resources.
- Integrity: Change in undesirable ways.
- Non-repudiation: Possibility of deleting files by a malicious user without fool-proof attribution.

In a file-related race condition, the file's directory is the race object. The time between which the file is checked and time it is opened, the vulnerability window is set for an attacker to replace a file that is being opened. It may sound impossible, but an attacker carries out a trick to slow down the program. If the vulnerable program is running with elevated privileges, a file not normally accessible may be opened and written to. For example, in the sample code, shown in Fig. 11.2, when an attacker replaces some_file with a link during the race window, this code can be exploited by writing into any file of the attacker's choice.

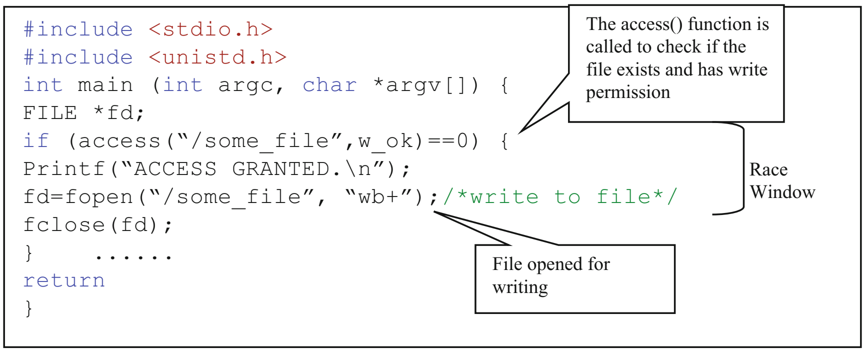
11.2.4 What Is Dynamic Detection?

Dynamic detection [7, 13] involves checking of a program during its execution and searching for problems.

- The program may be "instrumented" with extra instructions.
- The additions do not change program functionality and are used only to monitor conditions of interest.

For example following shell commands can be used during read-write operation:

```
rm /some_file
ln /myfile /some_file
```



```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
FILE *fd;
if (access("/some_file",w_ok)==0) {
Printf("ACCESS GRANTED.\n");
fd=fopen("/some_file", "wb+");/*write to file*/
fclose(fd);
}    ......
return
}
```

The access() function is called to check if the file exists and has write permission
Race Window
File opened for writing

Fig. 11.2 Sample program

Many tools have been developed for locating race condition either statically or dynamically. Most of the tools have serious deficiencies and cannot accurately identify all the race conditions.

11.2.5 Dynamic Race Detection Techniques

1. *Lamport's happens-before*: Happens-before is one of the earliest dynamic detection techniques. It defines a partial order for events in a set of concurrent threads. In a single thread, happens-before reflects the temporal order of event occurrence between threads: A happens before B if A is a lock access in one thread, and the next access to that lock (event B) is in a different thread and if the accesses obey the semantics of the lock (cannot have two successive locks, or two successive unlocks, or a lock in one thread and an unlock in a different thread). Figure 11.3 shows the usage of Lamport's Happens-before.

Data races between threads are *possible* if access to shared variables is not ordered by *happens-before*.

2. *Lockset analysis*: In this analysis, a consistent locking discipline is maintained. A locking discipline is a programming rule which ensures the absence of data races. For example, a mutual exclusion lock is protecting every variable shared

Fig. 11.3 Lamport's happens-before

Let event *a* be in thread 1 and event *b* be in thread 2.
If
$a = unlock(\mu)$
$b = lock(\mu)$
$a \rightarrow b$ (*a* happens-before *b*)

Fig. 11.4 Lockset analysis

```
In this analysis two data structures are used:
$LocksHeld(t)$ = set of locks currently held by thread $t$.
[Initially set to Empty]
$LockSet(x)$ = set of locks that could potentially be protecting $x$.
 [Initially set to the universal set]
    When thread $t$ acquires lock $l$
    $LocksHeld(t) = LocksHeld(t) \cup \{l\}$
    When thread $t$ releases lock $l$
    $LocksHeld(t) = LocksHeld(t) - \{l\}$
    When thread $t$ accesses location $x$
    $LockSet(x) = LockSet(x) \cap LocksHeld(t)$
Report "data race" when $LockSet(x)$ becomes empty.
```

between threads. Every data structure is confined by a single lock. All accesses to the data structure are made while capturing the lock. The first and simplest edition of the Lockset algorithm enforces a simple locking discipline by protecting every shared variable by some lock. A lock is held by any thread, whenever it accesses a variable.

There are two kinds of locksets. *Candidate locksets C(v)* is shared per variable. It contains all locks that may be protecting the variable. *Locks_held(t)* contains the locks currently held by a thread. Eraser is a dynamic data race detector for multi-threaded programs which uses this technique by intercepting operations held on runtime locks. Figure 11.4 shows the usage of Lockset analysis.

3. *Hybrids of happen-before and lockset*: It is a combination of lockset-based detection with a limited form of happens-before detection. RaceTrack is a dynamic analysis tool which uses this technique by providing proficient detection of data race conditions through adaptive tracking.

11.2.6 Dynamic Race Detection—Disadvantages

1. It can check only actual executed paths.
2. It incurs runtime overhead.

11.3 Assessment Methodology

In order to evaluate and compare available dynamic analysis tools with respect to their capabilities in detecting race conditions, the following steps were carried out.

11.3.1 Overview of Benchmark Frameworks

A benchmark framework is a repository of code with known bugs. Dynamic analysis tools can be tested, analyzed, and evaluated (to measure the detection, false alarm, and confusion rates) in an effective and affordable way with the help of benchmarks.

Juliet Test Suite: This test suite [14, 16] is provided by NSA Centre for Assured Software and published on NIST Web site. It is a collection of C or C++ and Java programs with well-known flaws. This test suite helps to understand the capabilities of software assurance tools.

- Based on Mitre's CWE classification system, Juliet version 1.2 for C/C++ contains 61,387 test cases for 118 CWEs.
- These test cases are small pieces of buildable code. Each test case contains exactly one flaw.
- In addition to the target flaw, there are one or more non-flawed constructs also.
- OMITGOOD and OMITBAD Macros can be used to detect the amount of true positives and false positives raised by an dynamic analysis tool.

In the context of the Juliet Test Suite:

- True positive (TP): A flaw of target type reported in flawed code.
- False positive (FP): A flaw of target type reported in non-flawed code.
- False negative (FN): If the tool does not report a flaw of the target type in flawed code, then it could be considered as a false negative.

Selection of Benchmark: In this study, Juliet Test Suite was selected in order to assess the tools for the following reasons:

- It follows the CWE structure. It properly classifies vulnerabilities, so it is very easy to know which types of vulnerabilities are found.
- It consists of test cases with a variety of lines of codes.
- It comprises of synthetic test cases for both C and C++ languages.

Table 11.1 CWE mapping

Name	CWE ID	Test cases in Juliet	Available Toolsuite in Valgrind	Rule availability in Parasoft Insure ++
Time-Of-Check-Time-Of-Use	367	Yes	No	No
Signal handler race condition	364	Yes	No	No
Race condition within a thread	366	Yes	Yes (Helgrind and DRD)	No
Race condition	362	No	Yes (Helgrind and DRD)	No

11.3.2 Mapping CWEs to Rules (Bug Patterns) in Dynamic Analysis Tools

Mapping of CWEs (C/C++) to rules/bug patterns has been carried out for the following tools—Parasoft Insure++ and Valgrind. The closest rules/tool suite in each tool has been mapped to corresponding CWEs as shown in Table 11.1.

11.3.3 Analysis of Tools

The test cases for the selected CWEs from the Juliet Testsuite have been run through the tools—Parasoft Insure++ and Valgrind. Three metrics [15], i.e., precision, recall, and *F*-score, have been calculated for the selected CWEs. The details of each metric have been explained in Sect. 11.3.4.

11.3.4 Metrics and Metrics Calculation

Precision

- Fraction of results from tool that were “correct,”
- Same as “true positive rate,” and
- Complement of “false positive rate.”

Precision is the ratio of weaknesses reported by a tool to the set of actual weaknesses in the code analyzed.

$$\text{PRECISION} = \#\text{TP}/(\#\text{TP} + \#\text{FP})$$

Recall

- Recall signifies the fraction of real flaws that were reported by a tool.
- Recall is also known as "sensitivity" or "soundness."

$$\text{RECALL} = \#\text{TP}/(\#\text{TP} + \#\text{FN})$$

F-score

An *F*-score is calculated using the following formula:

$$\begin{aligned} F\text{-SCORE} &= 2 * (\text{PRECISION} * \text{RECALL})/(\text{PRECISION} + \text{RECALL}). \\ \text{Precision} &= \text{No. of TP}/(\text{No. of TP} + \text{No. of FP}). \\ \text{Recall} &= \text{No. of TP}/(\text{No. of TP} + \text{No. of FN}) \\ \text{Recall} &= \text{No. of TP}/(\text{No. of test cases}) \end{aligned}$$

Methods of executing Juliet Test suite in the dynamic analysis tools are as follows:

1. Ran each test case with OMITGOOD and OMITBAD configuration and obtained the reports.
2. Considered OMITGOOD configuration reports to find number of true positives.
3. Verified whether reported flaw is a true positive or incidental flaw as follows. If the flaw reported exactly where the Juliet test case has a comment like this "/* POTENTIAL FLAW: <flaw of target type> */", then that flaw is treated as a true positive; otherwise, it is a incidental flaw.
4. Considered OMITBAD configuration to find out number of false positives.

11.3.5 Description of Dynamic Analysis Tools Used

Valgrind [17] is a "program-execution monitoring framework." Helgrind is a tool in the Valgrind suite for detecting synchronization errors in multithreaded programs. It supports C, C++, and Fortran programs that use the POSIX (Portable Operating System Interface) threading primitives.

It can detect 3 types of errors:

1. POSIX pthreads API misuses.
2. Potential deadlocks.
3. Data races.

Helgrind detects deadlocks arising from lock ordering problems. It observes the order in which threads obtain locks. Helgrind detects Heap races as well as Stack races.

POSIX threads, also recognized as Pthreads, is the most extensively accessible threading library on Unix systems. The POSIX threads programming model is based on the following abstractions:

- A shared address space : All the threads running inside the same process share the same address space. All data, whether shared or not, is known by its address.
- Regular load and store operations, which permit to read values from or to write values to the memory shared by all threads running in the same process.
- Atomic store and load-modify-store operations.
- Threads. Each thread represents a concurrent activity.
- The types of synchronization objects have been defined in the POSIX threads standard are: mutexes, condition variables, semaphores, reader-writer synchronization objects, barriers and spinlocks.

Fig. 11.5 POSIX threads

DRD is a Valgrind tool used for detecting errors in multithreaded C and C++ programs. The tool works on all programs which use POSIX threading primitives or that use threading concepts built on top of the POSIX threading primitives. Figure 11.5 gives details on POSIX threads.

Parasoft Insure++ [18] is an automated runtime application testing tool that detects errors—memory-related errors, variable initialization and definition conflict errors, library errors, pointer errors, I/O errors, and logic errors. The tool uses source code instrumentation technology.

11.3.6 Experimental Setup

The experimental setup consists of two tools, Valgrind which is an open-source tool, present default in Linux OS, and Parasoft Insure++ which is a commercial tool with the following platforms supported:

Microsoft Windows (32-bit and 64-bit):

- Visual C++.

Linux 32 and 64 bit:

- GNU gcc/g++.
- Intel ICC.

Steps for executing Juliet Test suite in Valgrind—Helgrind and DRD are as follows:

1. Editing the makefile for executing the program set in Valgrind—Helgrind and DRD.

2. Executing the edited makefile to generate an executable file for the complete set of program in the given CWE by enabling appropriate Valgrind–Helgrind and DRD tool options.
3. Collection and analysis of results in XML files.

Steps for executing Juliet Test suite in Parasoft Insure++ are as follows:

1. Set the path for executing programs in Parasoft Insure++ tool.
2. Editing the makefile for executing the program set in Parasoft Insure++.
3. Executing the edited makefile to generate an executable file for the complete program set.
4. Collection and analysis of results.

11.4 Analysis of Tool Results

Within each section, the results of Valgrind as well as Parasoft Insure++ with respect to CWEs are shown with the help of graphs. There are varying levels of performance as described in Table 11.2.

Time and State:

1. CWE-364: signal handler race condition

Description: Race conditions occur frequently in signal handlers, as signal handlers support asynchronous actions. Race conditions have a variety of root causes and symptoms. A malicious user may exploit this signal handler race condition to corrupt the software state, probably leading to a denial of service or even code execution.

Time of Introduction

- Architecture and design.
- Implementation.

Results for CWE-364 are shown above in Table 11.3. No results in Valgrind. No rules in Parasoft Insure++.

Table 11.2 Performance levels

S. No	*F*-score	Level of performance
1	<0.4	Poor
2	≥ 0.4 and <0.6	Satisfactory
3	≥ 0.6 and <0.8	Good
4	≥ 0.8	Excellent

Table 11.3 CWE-364: signal handler race condition

Tool	Precision	Recall	*F*-score	#TP	#FP
Valgrind	0	0	0	0	0
Parasoft Insure++	0	0	0	0	0

Table 11.4 CWE-366: race condition within thread

Tool	Precision	Recall	*F*-score	#TP	#FP
Valgrind	1	0.972222	0.985915	35	0
Parasoft Insure++	0	0	0	0	0

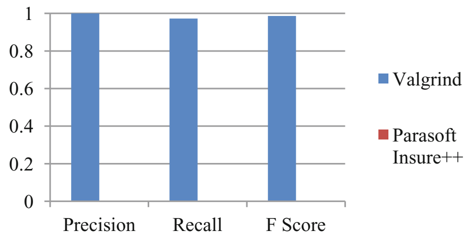


Fig. 11.6 CWE-366 race condition within thread

2. CWE-366: race condition within a thread

Description: If two threads of execution utilize a resource at the same time, there exists the possibility that resources may be used while invalid, in turn leading to undefined state of execution.

Time of Introduction

- Architecture and design.
- Implementation.

Results for CWE-366 are shown in Table 11.4 and captured graphically in Fig. 11.6.

Valgrind performance is excellent and no rules in Parasoft Insure++.

3. CWE-367: Time-Of-Check-Time-Of-Use (TOCTOU)

Description: The software verifies the status of a resource before using that resource, but the resource's state can change between the check and the use in a way that invalidates the results of previous check. As a result, the software is made to perform invalid actions when the resource is in an unexpected state. It can happen with shared resources such as files, memory, or even variables in multi-threaded programs.

Time of Introduction

- Implementation.

Results for CWE-367 are shown in Table 11.5.

Table 11.5 CWE-367: Time-Of-Check-Time-Of-Use (TOCTOU)

Tool	Precision	Recall	*F*-score	#TP	#FP
Valgrind	0	0	0	0	0
Parasoft Insure++	0	0	0	0	0

Table 11.6 Results of static analysis tools

Name	CWE ID	Test cases in Juliet	Rule availability in Parasoft	Rule availability in Klocwork	Rule availability in LDRA	Results from Parasoft C++ test, Klocwork and LDRA
TOCTOU	367	Yes	Yes	Yes	Yes	0
Race condition	362	No	Yes	Yes	Yes	0

Table 11.7 Results of dynamic analysis tools

Name	CWE ID	Test cases in Juliet	Valgrind results (errors)	Parasoft Insure++ results (errors)
Signal handler race condition	364	18	0	0
Race condition within a thread	366	36	35	0
TOCTOU	367	36	0	0

No results in Valgrind, and there are no rules in Parasoft Insure++.

Results of Static Analysis tools: Results of a previous study of assessing static analysis tools have been taken for comparing results of static and dynamic analysis tools with respect to their capability to detect race conditions and are shown in Table 11.6. In spite of rule availability for these CWEs in static code analysis tools, there were no results.

Table 11.7 depicts the assessment results of dynamic analysis tools.

Valgrind gives better results than Parasoft Insure++.

11.5 Comparisons of Tool Assessment Results—Dynamic Analysis and Static Analysis

Table 11.8 displays the comparison of tool assessment results of dynamic and static analysis tools with respect to the selected CWEs.

From this table, it is evident that dynamic analysis gives slightly better performance with respect to race conditions.

Table 11.8 Comparison table

S. No	CWE ID	Dynamic analysis tools		Static analysis tools		
		Open-source Valgrind tool	Parasoft Insure++	Parasoft C++ test	Klocwork insight	LDRA
1	CWE-364: signal handler race condition	No result	No rule	Poor performance	No rule	No result
2	CWE-366; race condition within a thread	Excellent performance	No rule	No result	No rule	No result
3	CWE-367: Time-Of-Check-Time-Of-Use (TOCTOU)	No result	No rule	No result	No rule	No result

11.6 Secure Design and Coding Guidelines

Program analysis tools assessed so far are not showing very promising results as is evident from the tables above. It becomes imperative to go by the adage "Prevention is better than cure" and tackle the race issues earlier in the Secure Software Development Life Cycle (SSDLC) using secure design and coding guidelines [19, 20]. The pitfalls when working with shared data, whether in the form of files, database, network connections, or shared memory, are leading to security compromise, and these are easily made mistakes.

In the section below, a few of the prevention measures that could be considered at the design and coding level are described:

1. Secure file operations.
    - Ensure file was opened successfully.
    - Set proper permissions on created files.
    - Use file descriptors not filenames.
    - Use fchow, fstat, fchmod, mkstemp functions.
    - Never reuse file names.
    - Use random file names for temporary files.
    - Use routines that operate on file descriptors—O_CREAT and O_EXCL flags to open system call.
    - Unlink the temporary files as soon as possible.
    - Use mkdir, rmdir, link, etc. carefully.
    - Use temporary file event logger.
2. Avoiding race conditions.
    - Make sure race windows does not overlap.
    - Make race windows as mutually exclusive.

- Make use of language facilities—synchronization primitives (SP) such as mutex variable, semaphores, pipes, condition variable, critical section objects, and lock variables.
- System security patches should be installed regularly.
- Avoid using ptrace.

3. Securing signal handlers.

- Use safe signal handlers.
- Signal handlers should not make any system calls and should terminate as quickly as possible.

4. Principle of least privilege.

- Use of access control—protect the resources from unauthorized access.
- Properly handle privilege elevation—minimize the time a program runs with privileges.

11.7 Conclusion

Performance of Valgrind is better than other available commercial tools for detecting race conditions. Vulnerability detection capabilities of Valgrind are inadequate in certain areas such as signal handlers. Analysis of the results in Valgrind is difficult as the results are verbose. The results from the study of dynamic analysis tools show that current vulnerability detection capabilities are inadequate in the area of dynamic race detection. In order to establish a Dynamic Analysis Framework (DAF) that performs substantially better, dynamic analysis tools showing good results in this area must be explored.

References

1. Common Weaknesses Enumeration (CWE) Version 2.5
2. Tsipenyuk K, Chess B, McGraw G (2005) Seven pernicious kingdoms: taxonomy of software security errors. http://cwe.mitre.org/documents/sources/SevenPerniciousKingdoms.pdf
3. Harrington E, Freund SN (2010) Using escape analysis in dynamic data race detection. Williams College Technical Report CSTR201401
4. Serebryany K, Iskhodzhanov T (2009) ThreadSanitizer—data race detection in practice. In: WBIA '09, Dec 12, New York City, NY. Copyright 2009 ACM 978-1-60558-793-6/12/09
5. Hutchins DL, Ballman A, Sutherland D (2014) C/C++ thread safety analysis
6. Kahlon V, Yang Y, Sankaranarayanan S, Gupta A (2007) Fast and accurate static data-race detection for concurrent programs. NEC Labs, Princeton, USA & University of Utah, Salt Lake City, USA

7. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (1997) Eraser: a dynamic data race detector for multithreaded programs. In: ACM transactions on computer systems, vol 15, no. 4, pp 391–411, Nov 1997
8. O'Callahan R, Choi J-D (2003) Hybrid dynamic data race detection. In: PPoPP '03, 11–13 June 2003, San Diego, California, USA. Copyright 2003 ACM 1-58113-588-2/03/0006
9. Marino D, Musuvathi M, Narayanasamy S (2009) LiteRace: effective sampling for lightweight data-race detection. In: PLDI '09, 15–20 June 2009, Dublin, Ireland. Copyright c_2009 ACM 978-1-60558-392-1/09/06
10. Pozniansky E, Schuster A (2003) Efficient on-the-fly data race detection in multithreaded C++ programs. In: PPoPP '03: proceedings of the ninth ACM SIGPLAN symposium on principles and practice of parallel programming, pp 179–190
11. Flanagan C, Freund N (2009) FastTrack: efficient and precise dynamic race detection. In: PLDI '09, 15–20 June 2009, Dublin, Ireland. Copyright @ 2009 ACM 978-1-60558-392-1/09/06
12. Spathoulas A (2014) Assessing tools for finding bugs in concurrent java. School of Informatics, University of Edinburgh
13. Yuan Y, Rodeheffer T, Chen W (2005) RaceTrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings SOSP '05. Copyright 2005 ACM
14. Juliet Test Suite Version 1.2 Documentation
15. NSA Centre of Assured Software Study (2011) On analyzing static analysis tools
16. Boland T, Black PE (2012) Juliet 1.1 C/C++ and java test suite. NIST
17. Valgrind Documentation
18. Parasoft Insure++ rule set
19. https://developer.apple.com/library/mac/documentation/security/conceptual/securecodingguide.re.visionhistory.html
20. www.cert.org/books/secure-coding/
21. http://www.cl.cam.ac.uk

Index