Satisfaction Meets Practice and Confidence

Tom Bienmüller
SVP Products
BTC Embedded Systems AG
Oldenburg, Germany
bienmueller@btc-es.de

Tino Teige
Chief Research Engineer Innovation & Technology
BTC Embedded Systems AG
Oldenburg, Germany
teige@btc-es.de

***Abstract*—The industrial application of formal methods and in particular of software verification tools, e.g. based on satisfiability checking and symbolic computation as being in the main focus of the SC^2 network, necessitates two main requirements. The methods and tools, first, need to actually aim at the problem class that occur in practice and, second, have to guarantee a high level of confidence. In this position paper, we raise two challenges in the domain of software verification: 1) enhancing the field of application of software verification tools in practice by means of efficient support of IEEE-754-based *floating-point* models and 2) enhancing confidence of software verification tools by means of generating *certificates* for their computed analysis results.**

Keywords—formal specification; formal verification; model checking; floating-point; proof certificates

I. Introduction

As an industrial associate partner of the SC^2 network, we certainly play a minor role in the academic research activities performed in that network. Even though having a matching academic background for this network, the benefit we see in our participation lays merely in the contribution we can give: show which topics are very close to the emerging needs real end users have who successfully apply formal test and verification tools already today. This could enable researchers to align their focus of work. Furthermore, as we as a tool vendor are close to real world applications, we can also provide data on these models coming from practice, allowing academic partners to validate the efficiency of their approaches in a real world setting. The benefit for us is obvious: getting in touch with leading researchers in a very important domain and get insight into ideas and technologies on how to answer to the emerging needs from end users.

II. BTC Embedded Systems AG

BTC Embedded Systems AG (BTC-ES), a majority shareholding of BTC Business Technology Consulting AG, provides products and services for formal verification, automated validation, and automated testing of embedded systems software. BTC-ES products significantly reduce required efforts for testing of embedded systems software, and considerably increase the quality of the developed systems. Additionally, BTC-ES offers on-site and off-site services around the testing products and testing in general in order to support customers in using the BTC-ES products successfully and efficiently. BTC-ES cooperates with two larger enterprises, dSPACE GmbH and IBM Rational Software. These two partners and BTC-ES strive to providing products and services for modeling, auto code generation, test automation, and formal verification to customers. dSPACE GmbH and BTC-ES support the tool chain Matlab®, Simulink®, Stateflow® and dSPACE's TargetLink®. IBM Rational and BTC-ES provide solutions for the tool chains concerning the modeling tools Statemate® and Rhapsody® UML. Together with dSPACE GmbH, BTC-ES has a strong focus on the automotive domain with key customers in Germany and Japan. Due to the cooperation with IBM Rational the BTC-ES products are also widespread across several other domains, e.g. aerospace, defense, rail, telecommunication and consumer. The core competences of BTC Embedded Systems AG are:

- model-driven development process
- formal specification and verification of requirements
- automatic test case generation for models and requirements
- automatic test case generation based on code
- automatic test case, trace, and vector execution
- model and code coverage

III. Use Cases

As tool provider for automated testing and formal verification methods in safety critical development BTC-ES is challenged to keep pace with the increasing demands from end users. There are two major aspects relevant to the SC^2 network:

- enhanced C-Code language support derived from the increasing use of IEEE-754-based floating-point processors, and
- higher demands on the confidence in formal verification tools in safety-critical development projects.

A. IEEE-754-Based Floating-Point Support

Even though not thoroughly established yet, production application of formal verification techniques more and more requests to efficiently support *floating-point arithmetic* for basic operations such as addition and multiplication, but also for complex operations such as square-root calculation, exponential and trigonometric functions. Current and well-

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: E. Ábrahám, J. Davenport, P. Fontaine, (eds.): Proceedings of the 1st Workshop on Satisfiability Checking and Symbolic Computation (SC^2), Timisoara, Romania, 02-07-2016, published at http://ceur-ws.org

established model checking techniques based on *propositional satisfiability* (SAT) checking or *binary decision diagrams* can deal with those floating-point operations, but the computation performance is very much negatively impacted by the necessity to reduce these operations to some Boolean decision problem aka bit-blasting. Even though it is a good approach to reuse the generic and well-established techniques for this type of problem, the obvious drawback is that it can be a hard problem for a Boolean solver to deal even with a small amount of combined floating-point variables.

To illustrate the problem of bit-blasting for more complex arithmetic floating-point functions like the exponential function *exp()* as provided by the standard C library `math.h`, consider the following rather small formula to be solved:

```
1.4 == exp(x)
```

with *x* being a 64-bit floating-point variable ranging in the interval [-104.0, 89.0].

To solve above formula, one can apply cbmc[1], a bounded model checker for C and C++ programs [1][2]. Before passing the formula to cbmc, one has to cope with a suitable definition of the function *exp()* which is not provided by cbmc. We remark that the C standard does not precisely define library functions like *exp()* but just gives a rough description. The implementation of *exp()* in our experiment consists of about 700 LOC including comments. Using cbmc with a SAT solver as its backend, the C code is translated into a SAT formula containing about quarter of a million variables and more than a million of clauses, provoking a rather high runtime of about 5 minutes.

In recent years there is a trend to extend SAT by dedicated theories leading to the problem of *satisfiability modulo theories* (SMT). The SMT community currently provides a large set of theories accompanied with powerful SMT solvers. Concerning above problem, the interval-based SMT solver iSAT3[2] [3][4] is a promising choice as it supports non-linear real arithmetic involving complex mathematical functions like *exp()*. Due to this built-in support, the internal formula representation has no space penalty, yielding a very fast solving time of above formula of fractions of a second. The obvious drawback of the iSAT3 approach (currently) is the lack of floating-point interpretations of these complex mathematical functions, though some first steps towards floating-point interpretation of basic operations were undertaken most recently [5].

The SMT approach perfectly fits into the challenge we are facing. Though there already are first attempts to standardize the theory of floating-point arithmetic within the SMT community [6][7], powerful tool support–in particular not relying on bit-blasting–is still a challenging and ongoing task, e.g. [8]. We strongly believe that strengthening the effort on the development of efficient floating-point SMT solvers being able to handle complex mathematical functions will be very beneficial for our customers, in particular with regard to the increasing trend of using floating-point processors and production code in industrial applications.

B. *Confidence in Formal Verification Tools in Safety-Critical Development Projects*

In reactive embedded software development for automotive applications, formal verification techniques such as model checking reached a high technology readiness level[3]. Off-the-shelf products implementing those techniques are available and applied not only in pilot projects and research, but also in production. This evolution comes along with increasing demands in the technology confidence. Furthermore, applying exhaustive formal verification techniques are viewed under return on investment constraints, obviously balancing efforts for formal verification with traditional testing approaches. Besides that, as model checking claims to be a “complete test”, it is assumed that all bugs in a system under verification are detected using that technology. More precisely, no witness trace showing a violation of a formal requirement specification remains unrevealed. Even though the techniques indeed are invented to give the ultimate answer, this expectation leaves one important aspect out of consideration: *what about semantic bugs in the model checker itself influencing that ultimate answer*? Albeit it is well understood that a software product of a size of a model checker contains bugs itself, occurrences of such bugs in production projects obviously have disproportionately dramatic impact on both the quality of the system and the cost to establish them, but also in the product’s confidence. Note that we are not talking about tool crashes here, but about bugs which are based on semantically wrong computations on a system under verification: the model checking result is wrong.

Software products of this importance and impact require dedicated measures to assure quality. Traditional software quality assurance techniques such as systematic testing are applied before shipping the product to end users. This kind of *offline* quality assurance works well but it can of course not exhaustively simulate the application at the end-user’s site. This means that still some bugs in the tool may remain undetected even though tremendous effort is spent for this kind of offline quality assurance. The arising question therefore is: what about the remaining bugs in model checkers which are not detected offline to the end-user’s application? How can we detect the occurrence of semantic bugs in a model checker in some kind of *online* quality assurance in parallel to the production operation?

The question of online quality assurance during real production application of model checkers is essentially the same as the question that the system under verification has been translated correctly to a model checkers input plus correct implementation of the model checking algorithms. In terms of a model checker, the above question can also be reduced to answer the following two queries:

1) Is a detected counter-example a genuine counter-example of the system under verification?

[1] http://www.cprover.org/cbmc
[2] https://projects.avacs.org/projects/isat3
[3] https://en.wikipedia.org/wiki/Technology_readiness_level

2) How can be made sure that claiming absence of a counter-example is a trustful answer that a counter-example indeed does not exist?

Intuitively, 1) is quite easy to answer. If a counter-example is generated, then this counter-example can be viewed as simulation trace. A simulation trace contains both stimulus and computed values of the representation in the model checker. Replaying that trace on the original system under investigation by checking if some stimulus indeed leads to the computed reactions as stated in the trace allows to safely judge whether the counter-example is a genuine one also showing malfunction of the original design[4].

Giving an answer to 2) is much harder. No counter-example is generated which would allow for straight-forward validation of the model checker's outcome. Alternatively, one could argue to simply let a second, different model checker answer the same inquiry in parallel to the first one. Upon successful application of the second model checker, diversity arguments could be applied: whenever two different model checkers return the same result then the probability of a wrong result is very low. That is true and theoretically the problem is solved with this argumentation. However and in particular in real world applications, there could be some constraints hindering the approach of really being applicable:

- As we are in a proprietary setting: is there a second model checker available at all?
- Even if there is a second model checker available, is that model checker able to produce the same result in terms of complexity?
- Would an end user accept a potential run time and space increase?

An alternative approach might be based on formal proofs of absence of bugs, preferably on the original software layer. When reducing the original formal requirement specification to (state or line) reachability in a program then we aim at a *certificate of unreachability* (of the corresponding state or line). The definition of such certificates desirably should not rely on any concrete analysis technique and should be revisable by a small stand-alone proof checker. This ultimate goal seems to be a very challenging and hardly investigated scientific topic but would have a very big impact on the confidence level of the system under verification and on formal verification methods in general due to the possibility of validating verification results *online* on customer site.

Let us roughly consider a typical verification tool chain as depicted in Fig. **1**: the original C program is instrumented according to the formal requirement specification, e.g., by introducing a new Boolean observer variable which is set to true if and only if the given requirement becomes violated. Thereafter, the C program is typically translated into some internal language in order to perform several transformation steps into a syntactically simpler form. Some usual rules are inlining of function calls, flattening of structural data types, reducing to the cone of influence, unwinding internal loops, and rewriting to single assignment code.

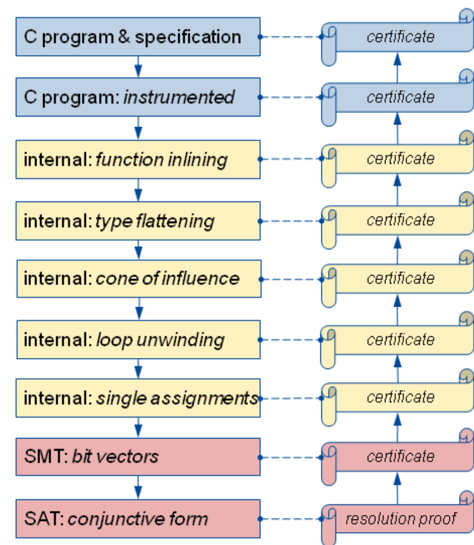

Fig. 1. Typical verification tool chain starting with a C program and a specification and leading to a corresponding SAT formula (left) and idea of lifting back a potential proof certificate from the SAT to the C layer (right).

When employing an SMT and/or SAT solver as the backend engines of the analysis tool chain, the code is finally translated into an SMT and/or SAT formula. It usually holds that the SMT/SAT formula is unsatisfiable if and only if the state where the observer variable is true is unreachable (for some bounded search depth).

Whenever it turns out that the SMT/SAT formula is unsatisfiable, we know that the given formal requirement is fulfilled–at least in theory. In practice however the verification tool chain mentioned above is a very complex collection of software being prone to error. Efficient SMT and SAT solvers in particular are highly optimized, even on bit level. In order to enhance trustworthiness of an unreachability result of the overall verification tool chain, it is desirable to generate a certificate which facilitates a separate validation of the result, namely independent from the actual model checking software.

One natural approach is to generate a certificate on the SMT/SAT solver level and then to lift this certificate back through all the intermediate translation and transformation layers up to the original code level as indicated in Fig. **1**. Though there is an extensive work on certificates of unsatisfiability for SAT formulas (based on resolution proofs of unsatisfiability, e.g. [9]), it seems that certificates of unreachability for imperative programming languages like C is hardly investigated. We believe that certification of verification results will become a major topic in future, in particular when establishing formal methods and their trustworthiness in industry.

[4] Here, we abstract from bugs in simulators enabling to replay a counter-example on the original system under investigation.

IV. Conclusion

As industrial associate partner of the SC^2 network we described some major challenges which we believe to be very important to be addressed in order to bring formal verification alive in broad production application in automotive embedded software development.

First, with raising attractiveness of IEEE-754 floating-point processors in the automotive domain, adjusted model checking techniques will be needed. Here SMT solving avoiding bit-blasting holds a lot of promise, not only for basic floating-point operations but also for complex ones like exponential function, square-root, etc. We strongly believe that SMT is the right choice to approach this new problem class as we obtained promising results here already also in cooperations with other academic partners of the SC^2 network.

Second, application of model checkers in broad software production dramatically increases the demand of confidence in the applied tool suites. Obviously, errors in one of a chain of tools used for developing and testing, in particular, safety critical software may have tremendous impact in efforts and costs on end-user's side for quality assurance. In worst case, the bug is not detected at all leading to issues in the produced system, the car, later on with effects hopefully not causing harm to human beings.

For model checkers, the worst case scenario is semantic bugs: the model checker's analyzed semantics of a system under verification differs from the semantics of the original design, leading to wrong verification results. Returning a wrong witness trace for a formal requirement violation is the easy scenario here–this can simply be tackled by subsequent simulation against the original semantics. But what is about a model checker returning no counter witness, claiming that there is no bug in the system under verification with respect to an inquired formal requirement? We believe that providing evidence for results a model checker produces is very beneficial when talking about bringing formal verification into production. This is why we propose to think about some kind of proof certificates, which enable to probe whether a model checker's result "is true with almost absolute certainty".

Acknowledgment

This work was supported by the H2020-FETOPEN-2016-2017-CSA project SC^2 (712689).

References

[1] Daniel Kroening, Michael Tautschnig: CBMC - C Bounded Model Checker - (Competition Contribution). TACAS 2014: 389-391

[2] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, Tom Bienmüller: Successful Use of Incremental BMC in the Automotive Industry. FMICS 2015: 62-77

[3] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, Tobias Schubert: Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. JSAT 1(3-4): 209-236 (2007)

[4] Karsten Scheibler, Bernd Becker: Implication Graph Compression inside the SMT Solver iSAT3. MBMV 2014: 25-36

[5] Karsten Scheibler, Felix Neubauer, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, Detlef Fehrer, Bernd Becker: Accurate ICP-Based Floating-Point Reasoning. FMCAD 2016 (to be published).

[6] Philipp Rümmer, Thomas Wahl: An SMT-LIB Theory of Binary Floating-Point Arithmetic. SMT 2010.

[7] Martin Brain, Cesare Tinelli, Philipp Rümmer, Thomas Wahl: An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. ARITH 2015: 160-167

[8] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, Daniel Kroening: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design 45(2): 213-245 (2014)

[9] Lintao Zhang, Sharad Malik: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. DATE 2003: 10880-10885