Software Certification for Temporal Properties with Affordable Tool Qualification

Songtao Xia and Ben Di Vito

Mail Stop 130
NASA Langley Research Center
Hampton, VA 23281
{s.xia, b.l.divito}@larc.nasa.gov

Abstract. It has been recognized that a framework based on proof-carrying code (also called semantic-based software certification in its community) could be used as a candidate software certification process for the avionics industry. To meet this goal, tools in the "trust base" of a proof-carrying code system must be qualified by regulatory authorities. A family of semantic-based software certification approaches is described, each different in expressive power, level of automation and trust base. Of particular interest is the so-called abstraction-carrying code, which can certify temporal properties. When a pure abstraction-carrying code method is used in the context of industrial software certification, the fact that the trust base includes a model checker would incur a high qualification cost. This position paper proposes a hybrid of abstraction-based and proof-based certification methods so that the model checker used by a client can be significantly simplified, thereby leading to lower cost in tool qualification.

1 Introduction

Safety critical programs, such as those controlling an airplane, a nuclear power plant, or a medical system, are subject to the highest level of verification and validation (V & V) effort, which is often outlined by administrative authorities. For example, to deploy an autopilot program onboard an aircraft, the vendor must supply evidence to a Federal Aviation Administration (FAA) representative that shows compliance to FAA's guidelines [11]. The certification process used in the aviation industry currently relies heavily on peer review and testing.

Many properties of a software product, such as correctness, or general as well as domain-specific safety, may be proven via deduction, synthesis, or other techniques [10, 1, 9, 5]. If the vendor proves the property and presents the proof to the FAA representative, the representative may check the proof and conclude that the system is indeed safe or correct relative to a specification. Such a scheme is known as proof-carrying code. In a general setting, the vendor may not have to provide a proof, but some intermediate, semantic-based objects (collectively called a certificate) that help to establish the proof. The generalized category of approaches is known as semantic-based software certification. The soundness of such a framework, however, is based on the assumption that programs constituting the "trust base" are correctly implemented.

Semantic based software certification was originally designed for the safe distribution of software in an untrusted environment. The approach can be adapted to software certification in avionics and other industries. A significant gap between research and industrial practice is the lack of qualified tools.[1].

It is necessary to distinguish two sets of tools. Besides the trust base, there are often other tools involved in a semantic-based software certification/reverification process. Naturally, the tools in the trust base should be more strictly scrutinized because their failure can allow errors to propagate to final products. It is expected that tools in the trust base will incur higher cost during qualification because of their higher criticality.

This suggests the need for architectural principles for designing tools to achieve desired trust goals. Choosing an optimal partitioning of components into trusted and untrusted sets becomes an important decision. Considering the high cost of qualification, the functionality needs to be decomposed in a way such that the combined cost of qualifying the tools is minimal.

Thus, the problem of selecting tools to qualify is a choice among approaches that have the required expressive power and trust attributes, and also allow a decomposition of the functionality that incurs acceptable qualification cost. Of particular interest in this paper is the case of abstraction-carrying code[14], which certifies temporal properties. Its trust base contains a model checker, which is an additional component beyond those of most other certification methods.

2 Abstraction-Carrying Code

In a sense, the concept of semantic-based program certification can be understood as decomposed program verification. Consider a vacuous program certification technique, where the certificate contains nothing. In this case, the regulatory authority (represented by and referred to hereafter as a DER, Designated Engineering Representative) has to verify the program on her own. If hints, for example, a loop invariant, are provided by the vendor, the DER is relieved of discovering this fact. But she needs to reverify that the loop invariant is indeed a loop invariant. On the trust base side, a data-flow analyzer that she may trust is now replaced by a simpler data-flow fact verifier. Because simpler programs are less expensive to qualify, and because we have assumed that a tool in the trust base requires a stricter, more expensive qualification process, the setting in which the vendor provides such a hint is beneficial in terms of tool qualification cost. As a principle, we should exploit this trade-off between the amount of information (size of certificate) provided by the vendor and the complexity of the trust base.

Traditionally, semantic-based program certification is proof-based. In theory, this scheme works for any properties that can be formalized in the underlying

[1] Other issues include, and are not limited to, recognition, training, and expressive power/tool support.

logic. And in practice, proofs can be generated for many safety properties even if approaches other than theorem proving are used. However, sometimes for general temporal properties, generating a proof may not be feasible. A different paradigm based on abstraction-carrying code is proposed. Table 2 lists several different, real or imaginary certification settings with their expressive power and associated trust base.

Certification Method	Properties	Trust Base
Null Certificate	provable properties	every tool needed for proving
Touchstone	type and memory safety	VCGen and proof checker
AutoBayes	domain specific safety and memory safety	VCGen and proof checker
Any proof	provable properties	VCGen and proof checker
Abstraction-carrying code	temporal properties	VCGen, proof checker and model checker

The first row in the table refers to the no-certificate situation. The second row roughly corresponds to the setting of the original PCC work by Necula and Lee [10], where type safety and memory safety is of concern, where the trust base contains a verification condition generator (VCGen) and a proof checker. Row 3 represents the application of PCC techniques in the verification of domain-specific properties. For example, work by Denney et. al., automatically generates programs with proof-carrying code style proofs for domain specific safety properties[5]. Row 4 corresponds to the setting where the client uses a proof assistant (and probably a lot of human effort) to prove properties of concern.

Our focus is on Row 5, which corresponds to the abstraction-carrying code research by Xia and Hook. The idea is to apply predicate abstraction [6] and model checking to a program to verify an LTL property. Predicate abstraction may be automated by adopting counter-example driven predicate discovery [2, 4]. In this process, a predicate abstraction of the program is generated and passed to a DER. A DER will first verify that the abstract model is faithful to the program and then verify that the property does hold on the abstract model. The faithfulness check can be implemented much the same way as in PCC, that is, via a VCGen and a proof checker.

The proof-carrying code literature has elaborated how the VCGen and proof checker may be constructed in a simple manner. For example, a typed assembly language approach [8] can adopted for VCGen construction and a higher order logic framework [10] is used in proof checking. Our research is focused on how to restructure a model checker to achieve similar results.

3 Qualifiable Model Checkers

One of the initial design goals of abstraction-carrying code is to reduce the size of the certificate because of the need to transport it and check it at run time. In certifying for administrative approval, however, size is not an important factor. There are approaches that generate proofs for certain sub-categories of properties after predicate abstraction/model checking[9, 7]. But such an approach may not be feasible for general LTL formulas. Therefore, the ability of abstraction-carrying code to certify temporal properties is still useful. In ACC, the trust base includes a model checker, which is absent from PCC. We are going to explore the trade-off between certificate size and complexity of the trust base to build a more cost-effectively qualifiable model checker.

The model checker to be used by a DER in abstraction-carrying code is different from a general purpose model checker: it checks an abstract model known as a Boolean program (BP)[3]. A typical BP statement tests if a propositional formula holds given an environment, represented as another propositional formula, and changes the state accordingly. Compared to a model checker for a target program, for example, the Java Pathfinder [13], the model checker for a Boolean program does not need to handle the semantics of the object language. In contrast, more than half of the code in Java Pathfinder implements the semantics of a virtual machine.

Still, this model checker is a fairly complicated program. For example, Moped [12], contains 10 K lines of C code, not counting the supporting BDD library. To reduce the size of this model checker while still achieving the requirements of re-verification, we resort to the tools that already exist in the trust base: the VCGen and the proof checker. Specifically, we use a hybrid approach to reuse some of the model checking work that would be performed by the vendor during the original analysis. This tool can be more complicated because it is not in the trust base. We enhance the vendor's model checker to record every transition made by the model checker. That is, for a transition (a BP statement t) that moves the system state (represented as a propositional formula) from s_1 to s_2, we note down the triple (s_1, t, s_2). Then the reduced model checker does not have to compute s_2, but just verify that $t(s_1)$ is s_2. Further, because the application of t to a state can be reduced to the test of satisfiability in propositional logic, we may simply keep a record of the piece of evidence that a proposition can be satisfied. This way, we can replace the SAT solver, or the BDD package used in the model checker with a Boolean evaluator. The DER will run this reduced model checker, which, when a satisfiability problem in the Boolean domain is needed, will just verify the proof presented by the vendor. In this way, the semantic engine needed to analyze the Boolean program is simplified.

We are at the very early stage of this investigation. We are aware that the complexity of the trust base is only one of the many factors involved in tool qualification. Still, we expect to build a prototype system that can be plugged into our previous implementation of an ACC system called ACCEPT/C and evaluate the effective of this reduced model checker. This will allow exploration of the primary trade-off: delivering more detailed evidence at certification time in exchange for the benefits of reduced-complexity verification tools.

References

1. A. Appel. Foundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science (LICS)*, June 2001.
2. T. Ball. Formalizing counter-example driven predicate refinement with weakest preconditions. Technical Report MSR-TR-2004-134, Microsoft Research, 2004.
3. T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science 2057*, pages 103–122. Springer-Verlag, May 2001.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, pages 154–169, 2000.
5. E. Denney and B. Fischer. Certifiable program generation. In *Proceedings of Generative Programming and Component Engineering*, 2005.
6. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of Conference on Computer Aided Verification (CAV) 97, Lecture Notes in Computer Science 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
7. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *Proceedings of Conference on Computer-Aided Verification (CAV)*, pages 526–538, 2002.
8. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
9. K. S. Namjoshi. Certifying Model Checkers. In *Proceedings of 13th Conference on Computer Aided Verification (CAV)*, 2001.
10. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
11. RTCA SC-167 and EUROCAE WG-12. Software considerations in airborne systems and equipment certification, December 1992.
12. S. Schwoon. Moped software. Available at http://wwwbrauer.informatik.tu-muenchen.de/˜ schwoon/moped/.
13. W. Visser, S. Park, and J. Penix. Applying Predicate Abstraction to Model Check Object-oriented Programs. In *Proceedings of the 33rd ACM SIGSOFT Workshop on Formal Methods in Software Practice.*
14. S. Xia. *Abstraction-based Certification of Temporal Properties of Software Modules.* PhD thesis, OGI School of Science and Engineering, Oregon Health and Science University, 2004.