Synthesis of Assurance Cases for Software Certification

Hamid Bagheri
University of Nebraska-Lincoln
Department of Computer Science and Engineering
bagheri@unl.edu

Eunsuk Kang
Carnegie Mellon University
School of Computer Science
Institute for Software Research
eskang@cmu.edu

Niloofar Mansoor
University of Nebraska-Lincoln
Department of Computer Science and Engineering
nmansoor@cse.unl.edu

Abstract

As software is rapidly being embedded into major parts of our society, ranging from medical devices and self-driving vehicles to critical infrastructures, potential risks of software failures are also growing at an alarming pace. Existing certification processes, however, suffer from a lack of rigor and automation, and often incur a significant amount of manual effort on both system developers and certifiers. To address this issue, we propose a substantially automated, cost-effective certification method, backed with a novel *analysis synthesis* technique to automatically generate application-specific analysis tools that are custom-tailored to producing the necessary evidence. The outcome of this research promises to not only assist software developers in producing safer and more reliable software, but also benefit industrial certification agencies by significantly reducing the manual effort of certifiers. Early validation flows from experience applying this approach in constructing an assurance case for a surgical robot system in collaboration with the Center for the Advanced Surgical Technology.

ACM Reference Format:
Hamid Bagheri, Eunsuk Kang, and Niloofar Mansoor. 2020. Synthesis of Assurance Cases for Software Certification. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381728

1 Introduction

As software is increasingly being used to control major critical infrastructures in our society, there is an urgent need to develop effective methods for *certifying* that a system provides an acceptable level of safety and reliability. In traditional safety-critical domains, such as automotive, medical devices, and avionics, a set of certification standards and regulations are in place to ensure that a system meets a certain level of safety and reliability before its release. Many of these standards are *process-oriented*, where the acceptability of a system is determined by the use of particular development or testing processes, rather than the properties of the system itself. For instance, a standard from the U.S. Food and Drug Administration (FDA) recommends the use of code coverage analysis and testing as evidence that an adequate amount of validation has been applied to the system [18].

There is little concrete evidence, however, that these standards have been or will continue to be effective for systems where software plays a dominant role. While the use of certain validation techniques is *likely* to improve the quality of software, its actual impact on quality is rather *implicit*, and does not directly demonstrate whether the system satisfies its critical requirements [11]. For instance, despite mature certification procedures regulated by the FDA, thousands of medical devices are recalled each year in the US, many of them due to software quality issues [17].

A different approach, called *product-based* or *case-based* certification, has been gaining traction among researchers and certifiers. [12]. In this approach, the certification of a system involves the construction of an *assurance case*—a documentation of an *explicit* argument that the system satisfies its desired requirements. An assurance case contains one or more *claims* about critical requirements of a system and a set of *evidence* demonstrating that the system indeed satisfies those claims. For instance, a case for a surgical robot system may claim that a robotic arm never moves out of its intended range of motion (possibly causing harm to a patient or a physician in the operating room). A claim itself is decomposed into a set of *sub-claims*, each of which states an assumption about the environment (e.g., "The patient remains secured on a table") or a property of code (e.g., "The robot control module correctly processes a request for an out-of-bound motion"). Various forms of evidence, ranging from the results of testing, static analyses, and formal verification to expert judgment, are provided to justify these sub-claims.

While state-of-the-art approaches have achieved noteworthy success in many domains, they suffer from two key shortcomings: (1) the informal nature of an argument and its susceptibility to logical flaws and (2) the lack of readily available analyses for generating sufficient evidence to support the argument. Indeed, assurance cases are, by and large, specified informally and lack a precise semantics. It is thus difficult for a certifier to rigorously validate an argument for potential flaws (e.g., a missing assumption or a logical inconsistency among subclaims). In addition, the link between a claim and its evidence is given a similarly informal treatment, and it is often up to the subjective judgment of the certifier to determine whether the evidence is sufficient to justify the claim. This lack of rigor also makes it challenging for the developer to determine how much evidence needs to be produced in support of a claim. To make the matter worse, existing certification methods are intended for one-time certification of a whole system, and not designed to support frequent, incremental evolution of software [13]. Re-certifying the system involves rebuilding the entire assurance case and generating the necessary evidence again; hence re-certifying the system after each software update would be economically prohibitive.

To address this state of affairs, in this paper, we set out to explore novel connections between system-level and code-level reasoning to provide a substantially automated, cost-effective mechanism for software certification: Specification and verification techniques are used to decompose a system-level property into component properties, which are, in turn, used to enable scalable, application-specific program analyses to generate evidence that the system as a whole

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381728

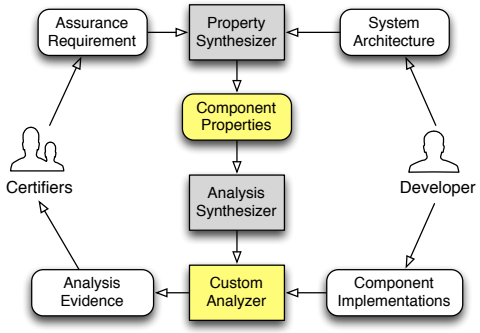

Figure 1: Approach overview. Boxes represent process modules, and ovals represent documents or artifacts. Grey indicates a module created in this work; yellow indicates an output from these modules.

satisfies its property. In particular, to eliminate the need to build ad-hoc analyses for evidence generation, we will leverage a novel *analysis synthesis* technique to automatically generate analysis tools that are custom-tailored to producing the necessary evidence. In comparison to existing methods, the proposed approach will offer (1) a **rigorous guarantee** that the resulting assurance case is free of logical flaws and (2) **reduced costs** for both developers and certifiers by replacing error-prone reasoning tasks with automated analysis.

The rest of this paper introduces the working scheme of our approach, reports and discusses our experience on applying the approach in constructing an assurance case for a surgical robot system in collaboration with the Center for the Advanced Surgical Technologies (CAST) [1], surveys related work, and concludes.

2 Approach

The proposed approach, shown in Figure 1, consists of two major components: (1) Property Synthesizer, which, given an *assurance requirement* and a *system architecture*, infers properties for individual components such that their composition forms a logically valid argument for a claim that the system satisfies a critical requirement; and (2) Analysis Synthesizer, which produces customized, application-specific program analyses to check that each system component meets its allotted properties.

2.1 Property Synthesizer

The Property Synthesizer accepts an *assurance requirement* that a certifier wishes to audit, and a *system architecture* that describes its components and types of data that are communicated among them. Then, the Synthesizer generates a set of *properties* that the individual components must satisfy in order to establish the given requirement. The composition of these properties forms an argument for a claim that the system satisfies the requirement. To be more precise, the Synthesizer performs the following task:

> Given a system-level requirement R and a set of n components $C_1, ..., C_n$, generate properties $P_1, ..., P_n$ such that if $C_i \models P_i$ for $i \leq n$, then $C_1 \oplus ... \oplus C_n \models R$.

where $C_i \oplus C_j$ is a composition of components C_i and C_j for $i, j \leq n$.

In particular, it generates *weakest* properties that are sufficient to establish R; in other words, for any $i \leq n$, if P_i is replaced with a weaker property P_i' such that $C_i \models P_i'$ but $C_i \not\models P_i$, then $C_1 \oplus ... \oplus C_n$ may no longer satisfy R. This, in turn, enables components that are not responsible for establishing R to be removed from the analysis phase, reducing the developer's burden of producing evidence.

```
1  // RobotApp: Main control
2  component RobotApp {
3    // Given a target coordinate, invoke a solver to convert into
4    // angle and return feedback to the user
5    event touchInput(coord:Point3D):Feedback { call Kinematics.solve;}
6    // Configure a new feedback value to be returned to the user
7    event genFeedback(feedback : Force) {...}
8  }
9  // Kinematics: Converts a coordinates to an angle
10 component Kinematics {
11   // Convert a 3D coordinate into an angle needed
12   // to move an arm to the target
13   event solve(coord:Point3D):Angle { call RobotApp.genFeedback;}
14 }
15 // RobotControl: Sends robot actuator commands
16 component RobotControl {
17   // Generate an actuator command to move an arm to the setpoint angle
18   event moveArm (setpoint:Angle) { call RobotComm.sendCmd;};
19 }
20 // Safety claims
21 claim MaxAngleSafety { // If the touch input exceeds the bound,
   // the angle of the robot arm is set to its maximum range.
22   RobotApp.touchInput(coord) = _ and coord > MAX_BOUND and
23   RobotComm.sendCmd(MOVE, angle) then angle = MAX_ANGLE }
24 claim OutOfBoundFeedbackProduced { // If the touch input
25   // exceeds the bound, non-zero feedback is provided back to the user.
26   RobotApp.touchInput(coord) = feedback and
27   coord > MAX_BOUND then feedback > 0 }
28 // Real-world types
29 type Point3D { x, y, z : int }
30 type Force { newton : double }
31 type Angle { radian : double }
32 type Command { val : { MOVE, PING, JOG } }
   // Enumerated type
33 const Point3D MAX_BOUND, Angle MAX_ANGLE;  // User-defined constants
```

Figure 2: Snippet of the Architecture Specification for the Surgical Robot System. The keywords and and then represent logical conjunction and implication.

To make the idea concrete, consider Figure 2 that shows a sample architecture specification for the surgical robot system, describing three components: RobotApp, Kinematics, and RobotControl. Each component defines a set of *input* events and may generate an *output* event by invoking events in other components. For instance, event solve in the Kinematics component takes a 3D coordinate as an input and returns an angle of the movement of a robotic arm necessary to reach the target coordinate (line 13).

Each specification also defines one or more *claims* that must be established on the specified system. For instance, Figure 2 represents two separate claims that state desired safety requirements of the robot (lines 20-27). The second claim states that to prevent the robot arm from moving out of a safe range, the system must produce feedback to the surgeon as a warning.

Each type declaration (e.g., Angle) is associated with its code-level representation (double). To make explicit the correspondence between specification- and physical entities, we adopt the notion of *real-world types* [19]. A custom analyzer generated by our Analysis Synthesizer then leverages the mapping between each type and its representation as an *abstraction function* to compute an abstraction of program variables and reason about properties expressed over the real-world types.

To synthesize properties of components, an input architecture specification (along with real-world types) will be translated into a constraint-based representation that is amenable to automated analysis (in particular, a first-order relational logic in Alloy [10]). A technique called *contract decomposition* [9] is then applied to automatically derive component properties that are sufficient to ensure system-level claims. To extract *minimal* properties, the synthesizer leverages the capability of a modern verification engine to produce a *minimal unsatisfiable core (MUC)*, which corresponds to a minimal subset of constraints that are needed to establish a claim.

```
1  RobotApp.property P1 { // Given a touch input, RobotApp moves the
2    // robotic arm according to the output from the Kinematics solver.
3    (recv touchInput(c)) and (Kinematics.solve(c) returns = setpoint)
4    then (call RobotControl.moveArm(angle) and angle = setpoint) }
5  // If the input coordinate is out of bound, the Kinematics solver
6  // returns max.angle range.
7  Kinematics.property P2 { (recv solve(c) and c > MAX_BOUND)
8    then solve.ret = MAX_ANGLE }
9  // RobotControl issues a correct MOVE command given an input angle setpoint.
10 RobotControl.property P3 { recv moveArm(setpoint)
11   then (call RobotComm.sendCmd(MOVE, angle) and angle = setpoint) }
12 // If RobotApp, Kinematics, and RobotControl satisfy their properties,
13 // then maximum angle safety is established.
14 argument A1 { (P1 and P2) and P3 then MaxAngleSafety }
15
16 // Given a touch input, RobotApp provides any feedback from
17 // the Kinematics solver back to the user.
18 RobotApp.property P4 { recv touchInput(c) and recv genFeedback(f)
19   then call Kinematics.solve(c) and touchInput.ret = f }
20 // If the input coordinate is out of bound, the solver
21 // generates a non-zero feedback.
22 Kinematics.property P5 { (recv solve(c) and c > MAX_BOUND)
23   then (call RobotApp.genFeedback(feedback) and feedback > 0)}
24 argument A2 { P4 and P5 then OutOfBoundFeedbackProduced }
```

Figure 3: Component properties generated by the Property Synthesizer from the surgical robot specification. The expression (recv e) states that a component has received an input event e.

When applied to the specification in Figure 2, the synthesizer produces the set of properties shown in Figure 3. For example, consider the claim OutOfBoundFeedbackProduced from Figure 2 (lines 24-27). This property does not rely on the behavior of RobotControl, since it only concerns the generation of feedback by Kinematics back to the user through RobotApp (P4 and P5 in Figure 3). However, the decomposition procedure initially produces an output that assigns some additional property P6 to RobotControl such that (P4 ∧ P5 ∧ P6) implies the claim. Given these properties, our core extraction algorithm then identifies P6 as an over-constraint and returns a weaker set of properties (P4 ∧ P5) as the final output.

2.2 Analysis Synthesizer

The Analysis Synthesizer takes properties generated in the prior step and synthesizes *application-specific* program analyses that are specifically intended to check that each component satisfies its assigned property. Once generated, the analyses are performed on the respective components and their results, if positive, are presented to the certifier as the evidence along with the assurance case.

Static analysis techniques have made tremendous progress in the past decade. However, applying these techniques to real-world safety-critical systems remains challenging, in part because despite significant advances in detecting general defects common across a variety of systems (e.g., buffer overflows), existing techniques fall short of detecting the violations of *system-specific* rules (e.g., "the computed angle of the robot arm must never exceed a safe threshold"). Using general-purpose analyzers to check domain-specific properties of a system often requires substantial expertise and manual effort, in specifying these properties, devising sound abstractions to enable scalable analysis, and manually discharging assumptions (e.g., expected behavior of a library) that a general-purpose analysis may not be designed to handle [8].

To address these challenges, we propose the *automated synthesis of domain-specific checkers* so that each system of interest can have analysis engines tailored to its specific characteristics and the properties to be examined, thereby significantly relieving developers from the cumbersome and error-prone task of developing customized checkers. We leverage well-understood classes of static analysis problems, such as type qualifiers [6], to facilitate automated derivation of customized static analyzers that are capable of effectively checking system-specific properties. More specifically, we build on prior research that have established the generic characterizations and realizations of type qualifiers to infer the custom instantiations of these frameworks [3–5]. Unlike all prior techniques that demand a type system developer to manually specify new type qualifiers and the semantics that they entail, and then to create the checker that enforces the semantics, our approach automatically synthesizes custom checkers for the properties that the Property Synthesizer has inferred for the individual components in the system.

To enable the automated synthesis of custom checkers, we utilize a library of analysis procedures and a library of checker templates. The scope of these libraries determine the range of rule violations that can be scrutinized by the specialized checkers. The generated checkers are then incorporated as extensions to a pluggable type system [5], which in turn applies them to the code base and generates alerts when violations are detected. The benefit of the library-based approach is the opportunity for *amortizing* the overall cost of the system-specific analysis for both certifiers and developers: Once constructed by domain experts, these fragments of analysis procedures can be reused in automated synthesis customized checkers for multiple scenarios, without program analysis or verification expertise.

Note that while our goal is to substantially automate the process of constructing an assurance case, there are aspects of construction that may not be amenable to automation. For instance, there are certain types of evidence that cannot be generated using an automated analysis, and instead require expert judgment—for instance, an assumption about a human actor (e.g., "A patient remains secured and still on the operating table") or a property of a non-software component (e.g., "The mechanical arm does not malfunction in an erratic manner"). Therefore, domain experts (e.g, a mechanical engineer) will continue to play an essential role in any certification process, including our proposed methodology. A systematic method of integrating different types of evidence is an important research problem on its own, but is beyond the scope of this work.

3 Evaluation

In this section, we show that our ideas can be reduced to practice. We have developed a prototype to synthesize trustworthy, yet customized and domain-specific, static-analysis environments. Our prototype is realized on top of the Checker framework [5], a state-of-the-art static program analysis tool with default checkers that can detect common issues in Java programs, such as possible buffer overflow vulnerabilities, null-pointer dereferences, and memory leaks. There are two main reasons that motivate our choice of the Checker framework as a platform for realizing the initial prototype. First, it has an active community of developers and a discussion group, and is widely used in several research projects. Second, it is open-source and publicly available. Note that our synthesis approach is general and can work with other pluggable type systems as well.

We applied our prototype to the code base of a control software for a family of surgical robots [14], developed at the Center for the Advanced Surgical Technologies Laboratory (CAST) [1]. As a concrete example, Figure 4 shows a simplified version of an assurance case for the surgical robot system. The claim to be established, using this case, states that the system never permits a robotic arm to move out of its safe range (cf. Fig. 2, lines 20-23). This claim is contingent on three sub-claims about the properties of three software components in the system (RobotApp, Kinematics, and RobotControl). The inferred rules are transformed into customized checkers

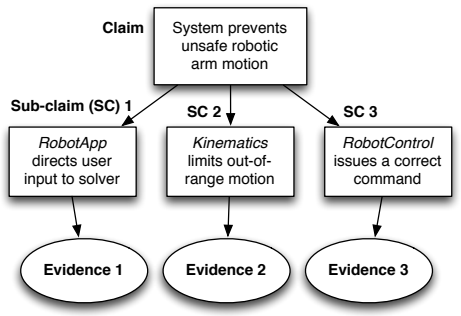

Figure 4: An example assurance case for the surgical robot system.

for the checker framework. The analysis engine uses these synthesized checkers to find rule violations in the code base, which are then examined by developers.

Among others, Property P2 (cf. Fig. 3) states that given an out of bounds coordinate, the *Kinematics* solver returns a max angle value. Verifying this property requires intra-procedural data flow analysis. This property is realized by setting the arm angle values to max values in specific conditions relating to the type of joint. The checker first needs to verify whether it is possible to reach the part of the program that sets the arm angle values to the maximum. It then needs to ensure that the value of the arm angle variable corresponding to the angle type is updated because in case the program goes into the specific branch but does not update the value, it may cause a serious problem. The results indicate that the customized analyzers can be synthesized for the properties inferred for each system component.

4 Related Work

Assurance cases. A lack of rigor and automation has been recognized as a significant barrier to a wider adoption of assurance cases for software certification. and there are a number of on-going and prior work on formal tool support for assurance case development. Among these, Pernsteiner et al. developed a formal assurance case to establish safety requirements of a radiotherapy system and generated a set of evidence to support the case using a combination of verification and analysis tools [16]. Similarly, Near et al. developed a formal assurance case for a proton therapy system and build custom-tailored code analysis tools to check the code-level properties in the assurance case [15]. In both cases, the construction of an assurance case and the selection of appropriate tools for evidence generation were performed manually; our goal is to automate these tasks.

Specification inference. A body of prior work exist on inferring properties or assumptions on parts of a system in order to establish an end-to-end specification. Most of these efforts, however, aim at synthesizing *operational* specifications in the style of state machines (e.g., [7]) or temporal logic ([2]). In comparison, our goal is to generate relational specifications of input-output component behaviors.

Extensible type system. There is an active area of research on developing techniques that enable multiple extensible type systems, including user-defined type systems, to be applied in a single language. Among others, the Checker Framework [5] promotes adding pluggable type systems to the Java programming language, and provides a framework on top of which developers can build custom analyzers. Building on these approaches, we propose a novel analysis synthesis framework that facilitates an automated realization of an application-specific custom analysis by providing a description of the corresponding properties to be analyzed, which in turn promises to considerably decrease the analysis cost while relieving developers from having to develop customized checkers.

5 Conclusion

This paper contributes a cost-effective, rigorous approach for software certification by automating many of the error-prone tasks that are manually performed by system developers and certifiers today. It offers a synergy between system-level and code-level reasoning, where specification and verification techniques are used to decompose a system-level property into component properties, which are used to facilitate synthesis of application-specific program analyses, custom-tailored to producing the necessary evidence. Early validation through experience of applying the approach to constructing an assurance case for a surgical robot system supports the feasibility of the ideas for software certification, while eliminating the need to build ad-hoc analyses for evidence generation. As next steps, we plan to (1) improve the generality of the Analysis Synthesizer to cover a wider range of properties, (2) explore approaches to make the Property Synthesizer and generated custom analyzers more scalable, and (3) demonstrate our techniques on other real-world case studies.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by awards CCF-1755890, CCF-1618132, CCF-1918140 and CNS-1801546 from the National Science Foundation.

References

[1] The Center for Advanced Surgical Technology. https://www.unmc.edu/cast/.
[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *ICSE*, pages 265–275, 2009.
[3] T. Carlson and E. Van Wyk. Type Qualifiers As Composable Language Extensions. In *Proceedings of GPCE*, pages 91–103, 2017.
[4] C. David, P. Kesseli, D. Kroening, and M. Lewis. Program Synthesis for Program Analysis. *ACM Trans. Program. Lang. Syst.*, 40(2):5:1–5:45, May 2018.
[5] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and Using Pluggable Type-checkers. In *Proceedings of ICSE*, pages 681–690, 2011.
[6] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, 1999.
[7] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12, 2002.
[8] C. S. Gordon. Synthesizing program-specific static analyses. https://arxiv.org/abs/1810.06600, 2018.
[9] I. Incer, A. Sangiovanni-Vincentelli, C.-W. Lin, , and E. Kang. Quotient for Assume-Guarantee Contracts. In *Proceedings of MEMOCODE*, 2018.
[10] D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.
[11] D. Jackson. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.
[12] T. Kelly and R. Weaver. The goal structuring notation–a safety argument notation. In *Dependable Systems and Networks (DSN) Workshop on Assurance Cases*, 2004.
[13] É. Leverett, R. Clayton, and R. Anderson. Standardisation and certification of the internet of things. In *Proceedings of WEIS*, 2017.
[14] N. Mansoor, J. A. Saddler, B. Silva, H. Bagheri, M. B. Cohen, and S. Farritor. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of ESEC/FSE*, 2018.
[15] J. P. Near, A. Milicevic, E. Kang, and D. Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *ICSE*, pages 31–40. ACM, 2011.
[16] S. Pernsteiner, C. Loncaric, E. Torlak, Z. Tatlock, X. Wang, M. D. Ernst, and J. Jacky. Investigating safety of a radiotherapy machine using system models with pluggable checkers. In *Proceedings of CAV*, pages 23–41, 2016.
[17] U.S. Food and Drug Administration (FDA). List of Device Recalls. https://www.fda.gov/medicaldevices/safety/listofrecalls. Accessed: 2018-11-14.
[18] U.S. Food and Drug Administration (FDA). General principles of software validation; final guidance for industry and fda staff. https://www.fda.gov/downloads/medicaldevices/.../ucm085371.pdf, 2017.
[19] J. Xiang, J. C. Knight, and K. J. Sullivan. Real-world types and their application. In *Proceedings of SAFECOMP*, pages 471–484, 2015.