Scaling Up Automated Verification: A Case Study and Formal-IDE for the Construction of High Integrity Software

Daniel Welch
School of Computing, Clemson University
100 McAdams Hall
Clemson, SC 29631
dtwelch@clemson.edu

ABSTRACT

This work aims to show through a detailed case study that scaling up automated verification to larger non-trivial data structures is not only possible, but when combined with appropriate tool support, can be made more comprehensible and practicable to users in a variety of settings, including the undergraduate curriculum. The study involves an interplay of multiple components annotated with formal interface contracts and the components are all designed to be modular, reusable, and amenable to automated verification and analysis. The components are built using a formal integrated development environment (F-IDE). The plan is to evaluate the F-IDE in an upper-level undergraduate software engineering course in the Spring semester at Clemson University.

Keywords

verification; contracts; software components; formal methods; IDE

1. INTRODUCTION AND MOTIVATION

Efforts to verify the correctness of software artifacts automatically with respect their formal contracts has come a long way in a relatively short period of time. Much of the work conducted thus far however has remained focused on the verification of relatively simple, isolated algorithms and data structures such as stacks, queues, and lists. While experience developing contracts amenable to automated verification of these structures has done much to motivate the creation and subsequent evolution of verification languages across a variety of dimensions, the question of how well existing techniques will scale when faced with larger, more inherently complex component-based systems and data structures remains largely unexplored territory.

Difficulty, high development cost, and a general lack of tool support are often cited reasons—among others—as to why high integrity software has never fully taken root in the industry or the curriculum at large [1]. The overriding belief

then is that current techniques for formally specifying such software are simply incapable of scaling.

To explore the question of scalability, and the role tools play in enabling learning of formal methods concepts in the classroom, we use RESOLVE [3], an integrated language that combines imperative programming and contracts for functional verification and has a verifying compiler [5]. The case study involves design, specification, and implementation of a host of component assemblies from the ground up and it employs a formal integrated development environment (F-IDE) designed for this purpose. The components involved, built as separate concepts within the F-IDE, are brought together in realizing a Kruskal based component for finding minimum spanning forests (MSFs) in disconnected, edge weighted graphs.

2. BACKGROUND AND RELATED WORK

Beyond case study specific details, the literature suggests two classifications of existing tools which one could use to conduct such a study. The first can be broadly categorized as “pure”, mathematically-oriented tools, such as Coq IDE (Coqoon) [2] or Isabelle/Eclipse [6]. These tools are graphical frontends for languages (namely, Coq and Isabelle) that aim to assist users in constructing machine-checked proofs of arbitrary mathematical assertions including theorems, lemmas, and corollaries. Such tools provide users with easy access to large libraries of “tactics”, and the ability to visualize proof progress through derivation trees that show the various steps and transformations made by such tactics as they are applied.

Complementing these are more the traditional, “practical” tools that aim to assist programmers looking to formally annotate their code with behavioral contracts, and ultimately verify that such code meets the contracts provided. The Dafny IDE is a recent example of such a tool [4]. Built for usage with Microsoft’s Visual Studio, it combines both the syntactic and semantic tooltips and code completion features users of modern IDEs have come to expect, with a responsive proving backend that provides continuous background analysis as users write their code.

3. APPROACH AND UNIQUENESS

The F-IDE we use to conduct our case study, built on top of the IntelliJ IDEA platform¹, leverages a unique combination of features from the two approaches summarized. From the “practical” side of F-IDE developments, the tool includes

¹<https://www.jetbrains.com/>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCSE '17 March 08–11, 2017, Seattle, WA, USA

© 2017 ACM. ISBN 978-1-4503-4698-6/17/03.

DOI: <http://dx.doi.org/10.1145/3017680.3022456>

Figure 1: Illustration of F-IDE features.

not only responsive refactor, navigation, and code completion features for writing both contracts and code, but also enjoys full integration with the RESOLVE compiler and experimental integration for its verifier (Figure 1). Thus, much in the same way one might run and re-run a compiler to diagnose and fix syntactic and semantic errors, integration with an automated verifier enables learners to iteratively design and ‘debug’ errors in their formal contracts.

On the other hand, inspired more by the “pure” theorem proving oriented tools, the F-IDE used also includes the ability to visually explore derivation of proof obligations from a given block of code with the click of a button. This enables learners in higher level (graduate) courses to explore how RESOLVE derives proof obligations from executable code from a set of sound and relatively complete ‘proof rules’.

4. RESULTS AND CONTRIBUTIONS

This work contributes the following: (a) An F-IDE that aims to both assist in scaling up verification, while also helping to make construction of high integrity software a more comprehensible and practicable activity to learners in graduate and undergraduate settings. (b) A formal case study centered around finding a minimum spanning forest of a graph using multiple reusable components that raise a number of outstanding challenges, including: verification that cuts across multiple mathematical theories, authoring contracts for components that exhibit relational behavior (inherent in optimization problems such as finding minimum spanning forests), and verification in the presence of higher

order predicates arising from the specification of a generic component that encapsulates sorting.

5. ACKNOWLEDGMENTS

A special thanks to all members of the RSRG, especially Murali Sitaraman, Joan Krone, Bill Ogden, and Bruce Weide. This research has been funded in part by NSF grants CCF-0811748, CCF-1161916, DUE-1022941, and DUE-1611714.

6. REFERENCES

- [1] D. Bjorner and K. Havelund. 40 years of formal methods. In *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442*, pages 42–61, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
- [2] A. Faithfull, J. Bengtson, E. Tassi, and C. Tankink. Coqoon. In *TACAS 2016*, pages 316–331, Berlin, Heidelberg, 2016.
- [3] G. Kulczycki et al. *A Language for Building Verified Software Components*, pages 308–314. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [4] K. R. M. Leino and V. Wüstholtz. The Dafny Integrated Development Environment. In *F-IDE 2014*, pages 3–15, 2014.
- [5] M. Sitaraman et al. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.
- [6] A. Velykis. Isabelle / eclipse. Software, available on <http://andriusvelykis.github.io/isabelle-eclipse/>.