SIGCSE: G: Scaling Up Automated Verification: A Case Study and A Formalization IDE for Building High Integrity Software

Daniel Welch
School of Computing, Clemson University
dtwelch@clemson.edu

ABSTRACT

This research aims to show through a detailed case study that scaling up modular verification to component-based software is not only possible, but when combined with appropriate tool support, can be made more comprehensible and practicable to researchers and students in the software engineering (SE) curriculum. The study involves an interplay of multiple components that have novel designs and object-oriented interfaces to encapsulate non-trivial data structures and algorithms. Each component is annotated with formal specifications that are all designed to be modular, reusable, and amenable to automated verification and analysis. The components are constructed using a formalization integrated development environment (F-IDE) that we've built for this purpose. Experimental evaluation of the F-IDE itself will be performed in the context of the software engineering curriculum at Clemson University over the course of upcoming semesters, while evaluation of the various components involved in the study will be performed on the basis of the provability of mechanically generated proof obligations.

1 PROBLEM AND MOTIVATION

The ability to formally specify and automatically verify functional correctness of software with respect to its formal specification is an ideal long pursued in the area of formal methods specifically, and the computing research community in general. Starting roughly in the 1960's [9] and leading through Tony Hoare's seminal issuing of the 'verifying compiler' grand challenge [10], advances in automated theorem proving, software engineering, and programming languages have collectively helped to bring the ideal of formally verified software closer to reality. Unlike testing, which can only reveal defects in software—not prove their absence, full formal verification has the power to guarantee that code behaves according to its given specification under all inputs and valuations.

There are several barriers to achieving the verified software vision put forth by Hoare. The first is the simple fact that the challenge demands languages that (1) permit users to write formal behavioral specifications for their code in the form of pre/post conditions and loop invariants and (2) automatically prove that component implementations and client code satisfy such specifications.

Perhaps the biggest barrier to overcome however lies within the mindset of software development community itself, which still considers the problem of automatically verifying software to be simply too difficult: citing either the daunting complexity of automated provers, or the sometimes deep mathematical insight needed in general to prove theorems. Nevertheless, the view that programs are necessarily difficult to verify seems counter-intuitive when one considers that most programmers—including those without extensive mathematical backgrounds—are able to intuitively reason about their code, and convince themselves and others that their programs work as intended.

One central thesis guiding this work then is that well-engineered software components (e.g., those that adhere to established software engineering principles such as abstraction, modularization, and reusability) will not only lead to proof obligations that mirror the simplicity of the programmer's intuitive understanding of their code [12], but will also enable verification to scale up to larger, component-based systems. While there has indeed been substantial progress in developing suitable abstractions for, and verifying relatively isolated linear data structures such as stacks, queues, sets, and lists, the question of how well existing techniques for specification and verification will scale when faced with larger, more inherently complex layered data structures remains largely unexplored territory. And though specification and verification of the components involved in a system of any size will no doubt be a time consuming, difficult, and expensive activity—the fact remains that the system overall can (and should) be engineered from well-designed, *reusable* components that effectively amortize the high cost of their verification across subsequent usages.

The question of scalability naturally then gives rise to the final barrier: which is a general lack of tool support for writing high integrity software of this kind [3]. Traditional programming languages have for many years enjoyed the backing of powerful Integrated Development Environments (IDEs) that provide everything from a centralized workbench for projects of all sizes, to powerful code navigation and completion features that enhance user productivity and ease cognitive burden. To this end, a new class of IDEs broadly termed 'Formalization IDEs' (or, F-IDEs) aim to similarly assist users in crafting formal specifications and ease interaction with some underlying proof system.

To explore the question of scalability of component-based software verification and the role tools play in the process, this work offers two contributions. The first is an F-IDE built to support formal specification and push button verification of imperative programs written in RESOLVE [21]—an integrated programming and specification/modeling language. The second contribution is a case study that encompasses the formalization of a more complex, layered, component-based software system designed to demonstrate the scalability of our approach to component specification and verification.

The rest of the paper is organized as follows. Section 2 discusses several existing specification and verification languages with an emphasis on those that have undertaken development of F-IDEs of their own, while section 3 details the case study at a high level and discusses some of the features of our F-IDE for RESOLVE. Section 4 discusses ongoing work and plans for experimental evaluation of our F-IDE in the SE curriculum at Clemson University.

2 BACKGROUND AND RELATED WORK

While there are many programming and verification languages (several of which are assessed in [13]), only a handful (such as, for example, SPARK 2014 [8] and Spec# [2]) provide *fully integrated* environments that combine a full range of features such as responsive editing support, executable code-generation, and annotated verifier feedback/debugging. Though full integration significantly raises development and maintainability overhead, the accessibility benefits that result can be significant, especially in an educational setting. Even interactive proof assistants such as Coq (which until now have relied almost exclusively on expert systems such as Emacs with multiple supporting tools) have undertaken a significant effort in integrating proof assistant style workflows into Eclipse [6].

In this section we restrict our discussion to only a handful of efforts that are most closely related to our approach: specifically those that (1) tackle full functional verification of imperative, sequential programs and (2) enjoy prominent fully integrated frontend tool support in the form of an F-IDE.

KeY. The first effort we discuss is KeY [1]—a long running research project that provides a collection of tools geared towards deductive verification of object-oriented programs written in Java. Specifically, KeY uses the Java Modeling Language (JML) [16] to express formal behavioral contracts through specially designated class and method level comments. For verification, JML annotated programs are first translated into a set of proof obligations (POs) that are expressed in a language called Java Card Dynamic Logic (or simply JavaDL—an extension of Hoare logic), and are then sent to KeY's integrated backend prover. While this prover supports automation to a certain extent, in cases where this fails, the system is also capable of serving as an interactive proof assistant that allows users to systematically apply 'taclets' (i.e., *tactics*) to the current proof state—manually guiding the system towards the goal.

In terms of F-IDE support, KeY broadly supports two systems. The first is a standalone Java application that serves as the standard frontend for the tool, simply called KeY GUI. This environment functions as a general purpose KeY editor that allows users to perform functional verification of JML-annotated programs, explore resultant proof obligations, and interact with the prover by both applying existing 'taclets', and creating new ones. The second frontend is an extension that integrates KeY into Eclipse [7]. Upon writing JML-annotated code, or changing existing code, the plugin automatically invokes KeY's prover in the background (e.g., upon saving the document), and marks methods within the editor accordingly depending on whether or not the corresponding proof succeeded. The F-IDE tracks these annotations persistently across runs, visually indicating dependencies not yet fully verified.

Dafny. The second effort is Dafny [17], an object oriented, imperative language from Microsoft Research which (like RESOLVE) is designed from the ground up to support formal specification and automated verification. As a result, the language includes builtin syntactic slots for formal annotations such as pre/post conditions, loop invariants, and others. This not only eliminates the need to retrofit them in through special comments (as in the case of KeY) but also results in generally simpler specifications and proof obligations, since Dafny generally avoids features present in mainstream languages that are known to complicate formal reasoning such as uncontrolled referencing and aliasing [14].

Verification in Dafny works by first accepting a formally specified program and translating it into an intermediate verification language called Boogie 2 [18], proof obligations are then generated from this intermediate representation and sent to Microsoft's popular automated theorem prover Z3 [5] for verification.

F-IDE support for Dafny is provided through an extension to Microsoft Visual Studio [19]. Similar to KeY, Dafny's F-IDE provides users with design-time verifier feedback by continuously running the verifier in the background—triggered by each keystroke. To keep the system responsive, the IDE makes extensive use of caching (so proofs don't have to be recomputed unnecessarily) and multi-threading to allow multiple POs to be proven concurrently by different Z3 solver instances. The environment also includes the Boogie Verification Debugger (BVD) [15], which allows users to interactively explore states leading up to a failed assertion, such as a violated precondition. To mirror the way traditional program debuggers work (by inspecting concrete values of variables), such states are augmented with values generated by Z3 counterexamples.

Eiffel. Eiffel, the language that pioneered Design by Contract (DbC) is supported by an IDE called EiffelStudio that provides a complete environment for specifying and writing high integrity software. Due to the language's emphasis on safety and formal specification, Eiffel code has been used as a language frontend for multiple third party verification efforts such as [18]. An even more recent, impressive feat is the specification and verification of Eiffel's primary container library [20].

3 APPROACH AND UNIQUENESS

In this section we discuss the case study at large and detail some features of the F-IDE we've built to support specification and verification in RESOLVE.

3.1 Case Study: Minimum Spanning Forests

The case study we present illustrates the role mathematical modeling and specification design plays in building, scaling, composing, and ultimately verifying larger component-based[1] software systems and their implementations. The culmination of the study lies in the design and implementation of a component that finds minimum spanning forests (MSFs) in potentially disconnected graphs.

Figure 1 provides a high level illustration the relationships between the components involved in the study. In this figure, the cloud-like shapes represent component interfaces and the square boxes denote realizations. The solid arrows connecting boxes to clouds denote a *realizes* relationship while the dashed arrows connecting realizations to interfaces denote a *relies on* relationship.

We use the RESOLVE language to formally specify, implement, and (ideally) verify in isolation the various realizations that make up this system. One of the primary advantages of using this particular language is its unique emphasis on modular specification that enforces a strict separation between the mathematics used in formal specifications, and the executable, programmatic code used in realizations.

[1]We use the term 'component' and 'component based' to mean an interface coupled with one or more (interchangeable) realizations.

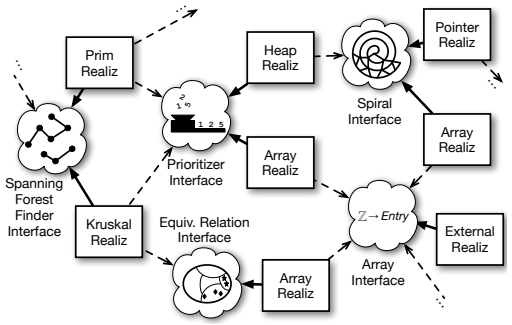

Figure 1: A design-time diagram of the case study.

As a result, each component in Figure 1 is formally specified in terms of strictly mathematical models which make the resulting interfaces not only highly *abstract* (since such models are inherently free of implementation- and computation specific biases) but also *modular* (since client and realization level code only needs to reason in terms abstract, interface specifications).

For example, a realization of the Spanning Forest Finder that uses Kruskal's algorithm relies on (1) a prioritizing component to sort edges by weight, and (2) an Equivalence Relation Tester to determine whether adding an edge to an under construction MSF will result in a cycle. Indeed—as shown in Figure 1—since the Kruskal realization relies only on the specifications of the operations and types exported by these interfaces, reasoning is kept abstract and modular. One could even conceivably swap out one implementation of the Prioritizer for another, and it wouldn't affect the verification of Kruskal's realization—as reasoning is performed strictly on the basis of formal interface specifications.

And while specification and verification of these components is an inherently time consuming, difficult, and expensive activity, reuse has the power to amortize the high cost over a component's subsequent lifespan. And though the study at hand falls notably short of a typical 'real world' application in terms of size and subject matter, the complex and layered nature of the data structures and algorithms involved nevertheless represents a significant modeling and verification scalability challenge for component-based software. The following summarizes several of these challenges.

- Specification and verification in the presence of user-defined, extensible, higher-order mathematical theories (some examples include theories for multisets, graphs, etc).
- Verification that cuts across multiple theories. Currently, pure SMT based verification schemes and decision procedures are tailored to a set of highly specific theories, or combinations thereof. Since this case-study involves multiple user defined theories, it will be instructive to study the proof obligations arising from component interconnections—the proofs of which will inevitably require results from multiple domains.
- Verification in the presence of *abstraction relations*—as opposed to abstraction functions. The need for abstraction relations, (discussed at length in [22]), is critical when expressing formal specifications of optimization problems (such as finding MSFs) for which there may exist multiple 'correct' solutions.

Due to space constraints, in this paper we only briefly discuss aspects one particular component's specification. Readers interested in reading more about the other components involved and their specifications are encouraged to refer to [23].

3.2 A Formalization IDE for RESOLVE

The F-IDE we've constructed, which integrates support for RESOLVE into commercial and open source JetBrains IDEs [11], leverages a unique combination of features from the approaches summarized in section 2. To illustrate some of these features, while also providing a general overview of RESOLVE's approach to formal specification, we detail elements of the prioritizing component discussed in the previous section. Figure 2 shows a screenshot of the environment open with the interface of this component.

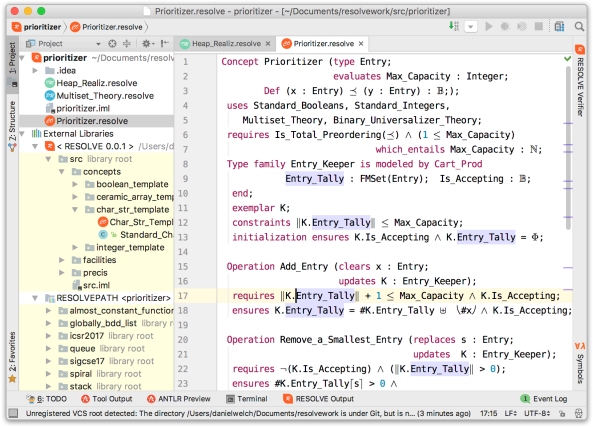

Figure 2: RESOLVE JetBrains IDE integration.

The F-IDE adheres to a traditional layout: the project explorer in the top half of the left-hand-side pane provides access to files in the current project, while the bottom half (under "external libraries") provides access to RESOLVE's standard library of reusable, verified components, as well as third-party projects on RESOLVPATH—a distinguished directory under which user projects are placed.

The active editor in Figure 2 shows a concept interface for a prioritizing component. The interface is parameterized by a generic type `Entry`, a parameter `Max_Capacity` that serves as an upper bound on the number of entries the prioritizer can hold, and a binary predicate $\preccurlyeq$: `Entry` × `Entry` ⟶ $\mathbb{B}$ that provides a general means of ordering the entries that make up the abstract state of the interface. The `requires` clause that follows establishes a module level precondition which stipulates that `Max_Capacity` is positive, and that $\preccurlyeq$ is a total preordering (i.e., total and transitive).

The `uses` clause brings in several additional modules that provide access to various type declarations and mathematical notations. As Figure 3 shows, the IDE tracks these and provides contextual completions for modules across projects as users are typing.

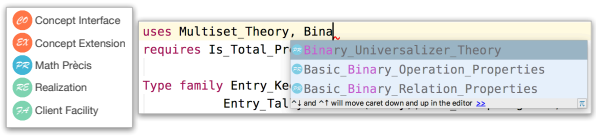

Figure 3: Module uses completions.

Since RESOLVE supports a variety of module types, icons are used to clearly delineate the type of module being imported: for example, math theory (prècis) modules use , while concept interfaces use .

3.2.1 Abstract Specification. The prioritizer can be conceptualized as a machine that operates in two phases: an insertion phase in which new entries are added, and an extraction phase in which elements are removed one-by-one in order according to the application-dependent $\preccurlyeq$ relation.

The `Type family` declaration makes this conceptualization explicit by introducing a type `Entry_Keeper` that we mathematically model as the cartesian product `Cart_Prod` of a finite multiset[2] `FMSet` (for storing entries), and a boolean flag `Is_Accepting` (for tracking the current phase). The `exemplar` binding provides a reference to the model type within the scope of its declaration, and is subsequently used in the `constraints` to assert that the number of elements the machine can hold is within `Max_Capacity`. The `initialization` clause that concludes the type declaration `ensures` that the machine starts out empty (i.e., equal to the empty multiset, Φ) and accepting new entries.

Operation specifications include the usual annotations such as pre/post conditions formalized via `requires` and `ensures` clauses—respectively—in addition to *parameter modes* that make explicit the effect an operation will have on arguments passed. Modes include `clears` (the argument passed will be reset to its initial value), `evaluates` (an expression is expected), and `updates` (the argument will be updated as specified in the `ensures` clause).

In English, the specification for `Add_Entry` therefore requires that the machine is accepting new entries and has enough room for a new entry `x`, while ensuring that the outgoing multiset is equal to the (#-denoted) incoming multiset unioned with the desired, additional entry: `#x`.

The bottom half of Figure 4 shows a snippet of `Multiset_Theory` which houses the notations and operators used to formalize the prioritizer's specification, as well as a selection of theorems and corollaries required for automated proof. The top half on the figure illustrates a realization of the `Add_Entry` operation in the context of a heap-based prioritizer realization. The behavior of the loop in this particular implementation is captured through several formal annotations including a `maintaining` clause (i.e., a *loop invariant*) that must hold at the beginning and end of each iteration and a `decreasing` clause (i.e., a *progress metric*) for proving termination. This figure also illustrates how the F-IDE presents proof obligations (or, verification conditions) to users. Lines that generate one or more proof obligations are marked by a icon in the gutter and can be selected and viewed in the "RESOLVE Verifier" panel.

3.2.2 F-IDE Assistance. As expected of modern environments, the F-IDE provides contextual completion suggestions for partially typed keywords and references (illustrated in Figure 4). Such completions are performed in an intelligent way: That is, the IDE only suggests keywords in regions where they are grammatically valid,

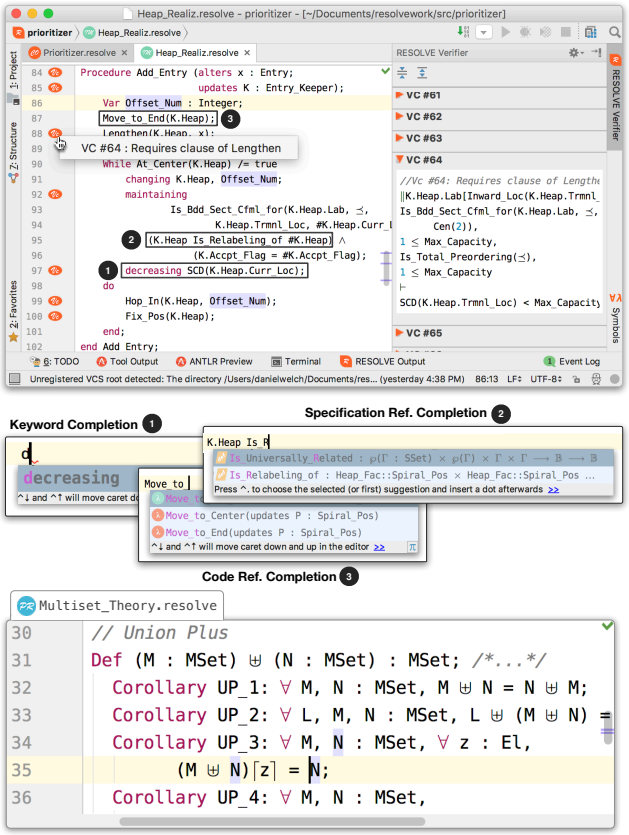

Figure 4: An implementation of `Add_Entry` (top) and a snippet of `Multiset_Theory` (bottom).

and avoids offering completions for abstract mathematical definitions in a programmatic contexts (e.g., statement lists) where references to strictly program variables and operations are expected.

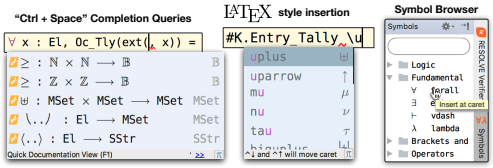

Figure 5: Support for mathematical symbols

To underscore the separation between formal mathematical specification and executable code, RESOLVE makes extensive use of non-ASCII unicode symbols throughout specifications. The F-IDE allows users to easily insert these special characters into the active editor via LaTeX-style insertion commands (Figure 5—middle, right) or by 'pinging' the scope for available operators (Figure 5—left).

3.2.3 Usage in Software Engineering Education. The F-IDE is fully integrated with the RESOLVE compiler which allows learners to verify their code in 'push-button' manner (Figure 6, left), then, from this verified code, generate property preserving, executable Java—and run it with the click of a button (Figure 6, right, bottom). More than simplifying interaction with the compiler, the environment also provides assistance with carrying out Design by Contract

[2] A multiset is an unordered collection of elements that may contain duplicates, such as ⟅1, 1, 3, 5, 3, 1⟆. Multiset cardinality is $\| \ldots \|$, union is $\uplus$, and $\lceil \ldots \rceil$ 'tallies' the number of times an element appears in a given multiset.

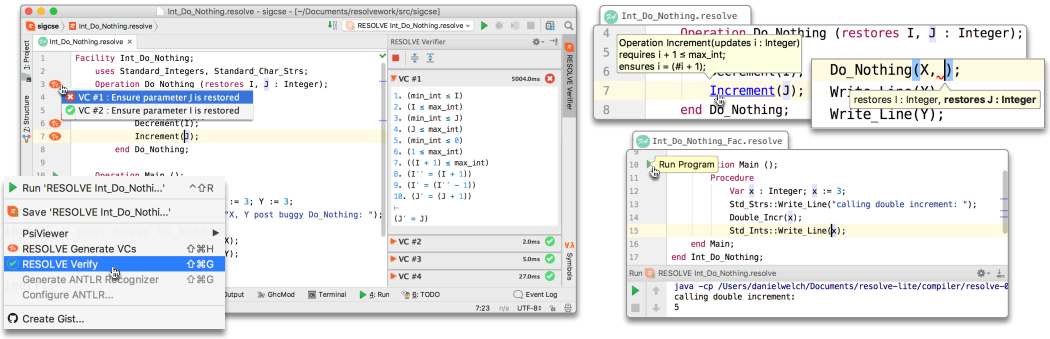

Figure 6: RESOLVE's in-house automated prover (left), DbC assistance (right, top), and code gen (right, bottom).

(DbC). This is done through tooltips that not only show pre- and post conditions at the site of calls and module instantiations, but also displays (while typing) the formal parameter mode associated with each argument as it's being passed.

4 FUTURE WORK AND CONTRIBUTIONS

The components outlined in section 3.1 have been specified and implemented in a prototype version of the RESOLVE compiler using the evironment detailed. There are two avenues of ongoing and future work. First is a detailed evaluation of the proof obligations arising from case study component realizations. This evaluation, performed in a manner similar to [12], will attempt to establish the 'mathematical obviousness' of proof obligations arising from more complex, layered components within this study. The second avenue is an evaluation of the F-IDE itself in the SE curriculum at Clemson, where students will use it to create, specify, and verify some simpler components. We plan to analyze and compare student experiences using the F-IDE presented here with an existing, more well established web-based IDE that has been used for several years in software engineering education [4].

This research aims to contribute (1) an F-IDE that assists in scaling up verification and (2) a formal case study using multiple components that employ novel theories and abstractions that raise a number of outstanding challenges, including: verification that cuts across multiple mathematical theories, contracts that exhibit relational behavior (inherent in optimization problems), and verification in the presence of naturally arising abstraction relations.

ACKNOWLEDGMENTS

This research is funded in part by NSF grant CCF-1161916.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice.* Lecture Notes in Computer Science, Vol. 10001. Springer, Cham, Switzerland.

[2] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (June 2011), 81–91.

[3] Dines Bjørner and Klaus Havelund. 2014. 40 Years of Formal Methods. In *FM 2014 (LNCS)*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.), Vol. 8442. Springer-Verlag, Cham, Switzerland, 42–61.

[4] C. T. Cook, H. Harton, H. Smith, and M. Sitaraman. 2012. Specification engineering and modular verification using a web-integrated verifying compiler. In *ICSE 2012*. 1379–1382. DOI: https://doi.org/10.1109/ICSE.2012.6227243

[5] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008 (LNCS)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, Berlin, Heidelberg, 337–340.

[6] Alexander Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. 2016. Coqoon. In *TACAS 2016 (LNCS)*, Marsha Chechik and Jean-François Raskin (Eds.), Vol. 9636. Springer, Berlin, Heidelberg, 316–331.

[7] Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. 2014. An Interactive Verification Tool Meets an IDE. In *IFM 2014 (LNCS)*, Elvira Albert and Emil Sekerinski (Eds.), Vol. 8739. Springer, Cham, Switzerland, 55–70.

[8] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. 2015. SPARK 2014 and GNATprove. *Tools Technol. Transf.* 17, 6 (Nov. 2015), 695–707.

[9] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[10] Tony Hoare. 2003. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM* 50, 1 (Jan. 2003), 63–69.

[11] JetBrains. 2017. IDEs. Software, https://www.jetbrains.com/. (2017).

[12] Jason Kirschenbaum, Bruce M. Adcock, Derek Bronish, Hampton Smith, Heather K. Harton, Murali Sitaraman, and Bruce W. Weide. 2009. Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping?. In *ICSR 2009 (LNCS)*, Stephen H. Edwards and Gregory Kulczycki (Eds.), Vol. 5791. Springer, Heidelberg, Berlin, Heidelberg, 31–40.

[13] Vladimir Klebanov and others. 2011. The 1st Verified Software Competition: Experience Report. In *FM 2011 (LNCS)*, Michael J. Butler and Wolfram Schulte (Eds.), Vol. 6664. Springer, Berlin, Heidelberg, 154–168. DOI: https://doi.org/10.1007/978-3-642-21437-0_14

[14] Gregory Kulczycki, Hampton Smith, Heather Harton, Murali Sitaraman, William F. Ogden, and Joseph E. Hollingsworth. 2012. The Location Linking Concept: A Basis for Verification of Code Using Pointers. In *VSTTE 2012 (LNCS)*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer, Berlin, Heidelberg, 34–49.

[15] Claire Le Goues, K. Rustan M. Leino, and Micha l Moskal. 2011. The Boogie Verification Debugger (Tool Paper). In *SEFM 2011 (LNCS)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–414.

[16] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2013. *JML Reference Manual.* Draft Revision 2344.

[17] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR 2010 (LNCS)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, Heidelberg, Berlin, Heidelberg, 348–370.

[18] K. Rustan M. Leino and Philipp Rümmer. 2010. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *TACAS 2010 (LNCS)*, Javier Esparza and Rupak Majumdar (Eds.), Vol. 6015. Springer Berlin Heidelberg, Berlin, Heidelberg, 312–327.

[19] K. Rustan M. Leino and Philipp Rümmer. 2014. The Dafny Integrated Development Environment. In *F-IDE 2014 (EPTCS)*, Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry (Eds.), Vol. 149. EPTCS, 3–15.

[20] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. 2015. *A Fully Verified Container Library.* Springer, Cham, 414–434.

[21] Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather Harton, Wayne Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce W. Weide. 2011. Building a Push-button RESOLVE Verifier: Progress and Challenges. *Formal Aspects of Computing*. 23, 5 (sep 2011), 607–626.

[22] Murali Sitaraman, Bruce W. Weide, and William F. Ogden. 1997. On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations. *IEEE Trans. Softw. Eng.* 23, 3 (March 1997), 157–170. DOI: https://doi.org/10.1109/32.585503

[23] Daniel Welch and Murali Sitaraman. 2017. Engineering and Employing Reusable Software Components for Modular Verification. In *ICSR 2017 (In publication) (LNCS)*, Goetz Botterweck and Claudia Werner (Eds.), Vol. 10221. Springer International Publishing, Cham, Heidelberg, 139–154.