

F-IDEs with Features and VCs Designed to Assist Human Reasoning When Verification Fails

Yu-Shan Sun
Clemson University
Clemson, SC, USA
yushans@clemson.edu

Daniel Welch
Pennsylvania State University
University Park, PA, USA
dtw5246@psu.edu

Murali Sitaraman
Clemson University
Clemson, SC, USA
msitara@clemson.edu

This paper summarizes our efforts to aid human reasoning when verification fails through the use of two distinct Formalization Integrated Development Environments (F-IDEs) that we have developed. Both environments are modular and facilitate reasoning about the full behavior of object-based code. The first environment, referred to as the web-IDE, has been used for several years to teach aspects of formal specification and verification, including why and where verification conditions (VCs) arise and how to use them when verification fails. The second F-IDE, RESOLVE Studio, remains experimental, but is a more fully-fledged environment backed by a sequent-based VC generator that produces VCs with fewer extraneous givens. While the environments and VC generation techniques are necessarily language specific, the principles of alternative VC generation methods, F-IDE features, and observations about their impact on novices and experienced users are more generally applicable.

1 Introduction

As the importance of tools and environments with features for supporting software verification is becoming better understood, a variety of systems have been developed to fill this need. Indeed, the usability of *auto-active* [20] specification and verification languages such as Why3 [7], Dafny [19], RESOLVE [28], and AutoProof [31]—in which users indirectly interact with automated provers through formal contracts such as loop invariants—hinge almost entirely on the feedback provided to users through their respective front-end environments.

Auto-active style feedback typically takes the form of a collection of necessary and sufficient verification conditions (VCs) for proving correctness of code w.r.t. some formal specification. Different tools report (failed) VC details differently. For example, Dafny tends to report assertion failures through Z3-generated counter examples [24], whereas AutoProof provides a higher-level English explanation for each VC. While each tool has its distinct characteristics, they share the common need of providing effective, non-ambiguous support to users (preferably of all experience levels) when verification fails.

The F-IDE contributions of this paper fall under the categorizations of *usefulness* and *ease of use*. We consider how VC usefulness is achieved and how it can be effective. We detail how ease of understanding for a VC is achieved, focusing, for example, on how reasoning complexity is simplified for beginners by minimizing givens.

When necessary and sufficient VCs are generated and proved correct, their details are understandably of little consequence to software engineers. So why focus on VC generation at all? We consider two reasons. The first is practical: when verification fails, VCs represent a first, crucial foothold for novices and experts alike when attempting to identify which fixes are needed at the source code or specification level. The second reason is pedagogical: even when verification succeeds—students should be able to examine the details of a VC, understand which line of code (or construct) generated it, and why it was provable (e.g., making connections to concepts learned in discrete math courses). Consequently, easing

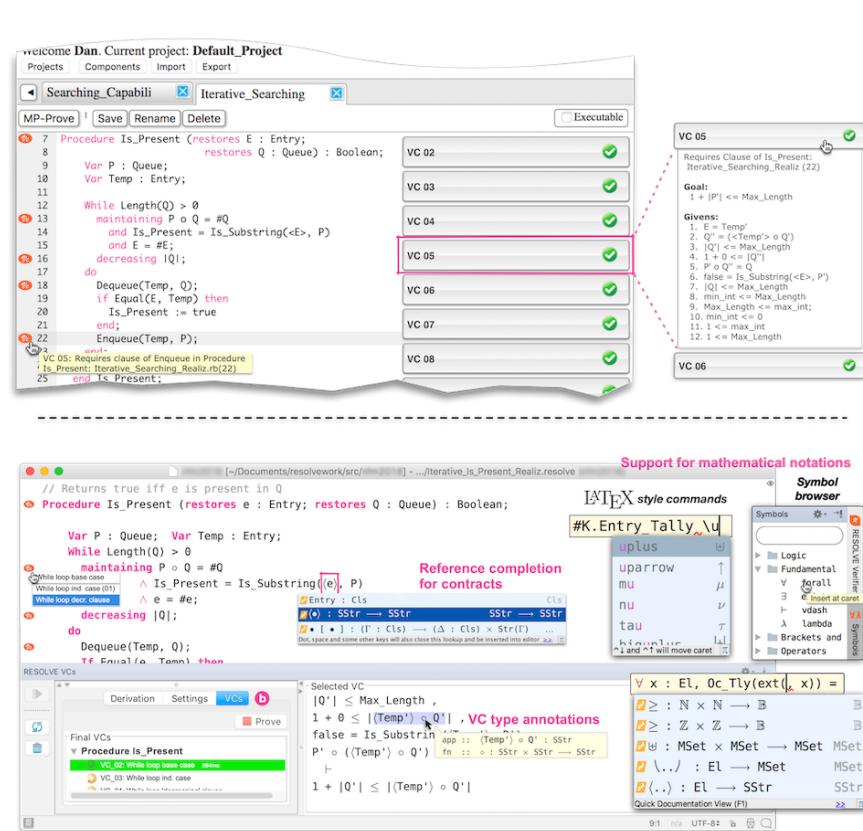


Figure 1: RESOLVE’s web-based F-IDE (top), and RESOLVE Studio (bottom).

the human reasoning process (through simpler, smaller VCs) and supporting student inquiry into what happens when a program is verified are chief motivations for the technical VC generation enhancements and tool features discussed in this paper.

We highlight aspects of two F-IDEs that we have built as the primary means of viewing VCs, proving, and debugging code that has failed to verify. The language targeted by both environments is RESOLVE [28]—an integrated specification and programming language that is imperative and object-based. The language—with variants adapted to popular languages such as C++ and Java [14]—has been used to teach modeling and reasoning principles (such as Design by Contract) from beginning undergraduate CS education to graduate level courses. Over 25,000 students across multiple institutions have benefited from these efforts over two decades [5, 14, 15].

The older of the two environments is the web-IDE (Fig. 1, top), which has been in use for over a decade as the primary development environment for RESOLVE.¹ The second, more recent F-IDE is a desktop based environment named RESOLVE Studio (Fig. 1, bottom). Built on top of the JetBrains IDE platform,² RESOLVE Studio combines the usual modern IDE amenities with a version of the RESOLVE compiler that includes the revised VC generation scheme detailed in this paper. Additionally, in an effort to guide the design and application of proof rules, RESOLVE Studio allows users to interactively derive VCs from Hoare style triples. While this environment remains under development, it has seen some usage (albeit limited) in a graduate programming languages course to construct specifications using

¹<https://resolve.cs.clemson.edu/teaching>

²<https://www.jetbrains.com/>

small, student-defined mathematical theories. The contributions of this paper are twofold:

- A general demonstration of how specification and auto-active verification of software can be supported in the context of two F-IDEs.
- A revised set of proof rules for generating sequent-based VCs in a *parsimonious* manner that omits irrelevant givens with the intent of improving human reasoning and VC comprehension.

The paper is organized as follows. Section 2 provides background on the RESOLVE language. Section 3 illustrates the need for simpler VCs by examining a failed VC in an older version of our web-IDE. Section 4 provides a high level overview of RESOLVE’s VC generation process, while section 5 details the proof rules that enable generation of parsimonious VCs. Section 6 shows VC feedback and other features in RESOLVE Studio. Sections 7 - 9 contain respectively: an experimental evaluation of the revised proof rules, related work, and conclusions with directions for future work.

2 RESOLVE Background

RESOLVE [28] is an imperative, object-based programming and specification framework designed to support modular verification of sequential code. Every RESOLVE component has a formal interface specification called a **Concept**. Below is a snippet of one such concept for bounded queues.

```

Concept Queue_Template (type Entry; evaluates Max_Length : Integer);
  requires Max_Length > 0;
  uses String_Theory;

  Type family Queue is modeled by Str(Entry);
  exemplar Q;
  constraints |Q| ≤ Max_Length;
  initialization ensures Q = Λ; // note: Q = Λ implies |Q| = 0

```

The concept shown is parameterized by a generic type `Entry` and an integer `Max_Length` that places an upper bound on the number of entries that can be stored. This bound must be positive (as per the module level **requires** clause) and may be an arbitrary expression of type `Integer` (as per the **evaluates** parameter mode). The imported module, `String_Theory`, gives the concept access to a mathematical theory of strings—including operators for: string length $|\bullet|$, concatenation \circ , the empty string Λ , and a singleton string formation function $\langle\bullet\rangle$.

2.1 Abstract Specification

Specifications in RESOLVE are *model-based*: that is, each programmatic type is conceptualized through the **Type family** construct, which, in this case, declares that the (program) type `Queue` is mathematically modeled as a string of generic entries, i.e.: `Str(Entry)`. The **exemplar** queue, `Q`, that follows gives specifiers a formal name to an example queue within the declaration. It is used immediately thereafter in the **constraints** to assert that (1) not *all* strings are models of valid queues, but only those of length `Max_Length` or less, and (2) that queues are empty upon **initialization**.

Operations are declared after the model. Below is one such operation for `Enqueue`.

```

Operation Enqueue (alters e : Entry; updates Q : Queue);
  requires |Q| < Max_Length;
  ensures Q = #Q ◦ ⟨#e⟩;

```

Formal contracts for Enqueue are communicated through **requires** and **ensures** clauses (i.e.: pre- and post-conditions). The contract also encompasses *specification parameter modes*. A mode of **updates** means that while the value of the incoming queue (#Q) may be meaningful, its outgoing value (Q) is to be updated in the manner specified in the **ensures** clause. The **ensures** clause can be read as follows: “the outgoing value of Q is equal to the #-denoted incoming queue concatenated with the singleton string containing the incoming value of entry e.” The **alters** mode means that the incoming #e may contain a meaningful value, but its outgoing value e is left unspecified. This gives implementers of the Enqueue operation the most flexibility as to which value is stored in the parameter e after a call to Enqueue completes.

To facilitate modular reasoning, the language enforces a strict separation between the abstract state expressed in interface specifications and executable, implementation level code. Thus, implementations must provide the correspondence information (abstraction functions or relations) necessary to connect the concrete state (in the implementation) to the abstract state (in the interface). Readers interested in a more complete description of RESOLVE should consult [11, 13, 28, 29] or one of the case studies [22, 30, 34] carried out using the language.

3 The Need for Simpler VCs: Examining a Failed VC

While even modest enhancements made to our presentation of VCs (e.g., annotating the source of VCs next to the line(s) on which they arise) have helped, the complexity of debugging VCs for beginners is considerably increased when they include a large, obfuscating number of irrelevant givens.

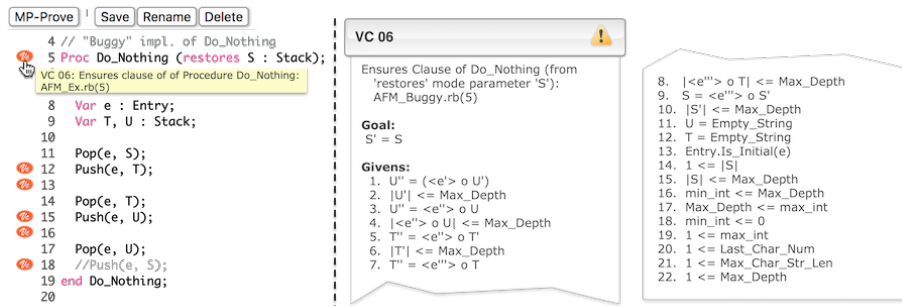


Figure 2: A VC that fails to prove that stack S is restored; note: the version of RESOLVE pictured does not accept non-ASCII notations—so Empty_String is used instead of Λ , < . > instead of $\langle \cdot \rangle$, etc.

For illustration purposes, Fig. 2 shows a buggy implementation of a Do_Nothing operation on stacks³ which **requires** $|S| \geq 1$, but fails to verify since the last call to Push is commented out. In this case, the failed VC was attempting to prove that S had been restored to its original state—as per the **restores** parameter mode on S which automatically adds $S = \#S$ as a conjunct to Do_Nothing’s **ensures** clause. After pressing the **MP-Prove** button, VCs are generated for pre-conditions of invoked operations, in addition to the post-condition of the operation being proved. Givens added to each VC arise from module level **requires** clauses and external **constraints** from imported modules.

The process is completed when all VCs are verified or the system times out. Since many initial student attempts contain errors and fail to verify, to provide quicker feedback, the timeout has been set

³Stacks and queues are modeled the same—though their operations (and specs.) differ

to be minimal on the web-IDE at the risk of not being able to discharge some otherwise provable VCs. Further, once at least three VCs fail to prove, the other VCs are not attempted. Hovering over the orange VC badges on specific lines gives information about the source of the VC(s) that arise from that line.

4 VC Generation Process

Generation of VCs that are both necessary and sufficient in order to prove that an implementation is correct w.r.t. its specification is a syntax-directed process. A more detailed description of RESOLVE’s VC-derivation scheme and its proof rules can be found in [13, 29].

Prior to the application of any statement level proof rules is a pre-processing step in which user code is logically grouped into *assertive-code* blocks wherein all mathematical assertions are made explicit. The traditional Hoare triple of the form $\{P\} c \{Q\}$ looks like the following in our presentation, as assertive code may include any number of preceding statements or assertions such as *Assume P*:

$$\mathcal{C} \setminus c; \text{ Confirm } Q;$$

Here, \mathcal{C} is a context containing a collection of typed symbols, c is a sequence of zero or more program statements, and Q is an assertion that must be confirmed to hold at the end.

Throughout the remainder of the paper, we employ a san-serif style font when typesetting the names of proof rules, as well as any meta-operators we define along the way.

4.1 Abstract Syntax

RESOLVE’s specification language is built on a many-sorted first order logic organized into several constituent parts. First, we assume an initial set of sort symbols $S = \{\text{SSet}, \dots\} \cup \{\mathbb{B}\}$. Here, *SSet* is the proper class of sets and the notation $T : \text{SSet}$ introduces a new user defined sort T into context.

- A set \mathcal{T} of nullary constants $c_1, \dots, c_n : T \in \mathcal{T}$ where c_1, c_n could also have any other sorts in S .
- A set \mathcal{X} of variables $x, y, z : T \in \mathcal{X}$ representing arbitrary elements which range over a domain T .
- A set \mathcal{F} of typed function symbols $f, g, \dots, h : T_1 \times \dots \times T_n \rightarrow T$ which range over \mathcal{F} . Here $T_1 \times \dots \times T_n$ represents the domain, while the (non-subscripted) T represents the co-domain.
- Lastly, a set \mathcal{P} of predicate symbols $p, q, \dots, r : T_1 \times \dots \times T_n \rightarrow \mathbb{B}$; which also includes a reserved binary predicate for equality ($=$) as well as nullary predicate symbols for *true* and *false*. We denote the arity of a given function or predicate o as $\text{ar}(o)$.

These sets constitute the language’s vocabulary $\mathcal{V} = \mathcal{T} \cup \mathcal{X} \cup \mathcal{F} \cup \mathcal{P}$; where \mathcal{V} can be enriched via the definition of new constants, functions, and predicates in RESOLVE’s object theories. As a result, these sets are not necessarily disjoint in practice. So determining whether a nullary “symbol” denotes a variable (e.g, bound under a quantifier) or the name of a module level definitional constant is dependent on the current context, \mathcal{C} . With these categories fixed, we now define formulas (which denote truth values) and terms (which serve as the fundamental building blocks of formulas).

Definition 4.1. *The set of formulas and terms of our specification language over vocabulary \mathcal{V} is given by the following abstract syntax.*

$$\mathbf{Form}_{\mathcal{V}} \ni \phi, \psi ::= P(t_1, \dots, t_{\text{ar}(P)}) \mid \text{true} \mid \text{false} \mid \neg \phi \mid \phi \circ \psi \mid \mathcal{Q}\bar{x}_n, \phi \mid t$$

$$\mathbf{Term}_{\mathcal{V}} \ni t, y, \tau ::= t_0(t_1, \dots, t_{\text{ar}(t_0)}) \mid t \rightarrow y \mid t_0 \times \dots \times t_n \mid \lambda \bar{x}_n, t \mid \dots \mid (\psi) \mid \#? s$$

where $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ and $\mathcal{Q} \in \{\forall, \exists, !\exists\}$.

Formulas consist of the usual binary connectives and quantifiers while terms permit (respectively) function application,⁴ function (arrow type) constructors, Cartesian products (\times), lambda abstraction, parenthesized formulae, and terminal symbols s . We use the notation \bar{x}_n as shorthand for a list of variable binders: $x_1, \dots, x_n : \tau$ of term/type τ .⁵

Next, we establish our syntax for assertive code fragments.

Definition 4.2. *A fragment of assertive code consists of zero or more program statements interleaved with assertive statements of the form:*

$$\begin{aligned} \mathbf{Stmt}_\gamma \ni s &::= \text{Assume } \phi; \mid \text{Confirm } \phi; \mid \text{Stipulate } \phi; \mid v := t; \mid \dots \mid id(y_1, \dots, y_n); \\ \mathbf{AsrtCode} \ni a &::= s^* \text{Confirm } \bigwedge seq^+; \quad \mathbf{Seqnt} \ni seq ::= \Gamma \vdash \Delta \end{aligned}$$

where Γ and Δ are sets of well-formed-formulas (wffs).

The statement production rule admits verification language specific statements (including `Assume` and `Confirm` clauses—which we discuss in Sect. 5) as well as strictly programmatic ones such as function assignment ($v := t$) and procedure calls ($id(y_1, \dots, y_n)$). We omit any remaining programmatic statements for brevity.

Lastly, since our parsimonious rules rely on the ability to compute sets of free variables from specifications, we employ a contexted “specification free variable” function $\text{SFV} : \mathcal{C} \times \mathbf{Term}_\gamma \rightarrow \wp(\mathbf{Term}_x)$. The following example demonstrates the function’s behavior.

Example 4.1 (SFV). After processing theory modules for mathematical Booleans, Integers, and Strings, suppose \mathcal{C} contains the following constants and predicates:

$$\mathcal{C} = \{ \Lambda : \mathbf{SStr}, \wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}, |\bullet| : \mathbf{SStr} \rightarrow \mathbb{N}, 0 : \mathbb{Z}, \dots \}.$$

The meaning of the types assigned to the constants should be reasonably familiar with the exception of \mathbf{SStr} —which is a user-defined sort representing the proper class of all heterogeneously typed (math) strings. Next, we enrich \mathcal{C} with the following operation signature, where the programmatic type `Static_Array` is mathematically modeled by $\mathbb{Z} \rightarrow \text{Entry}$.

$$\begin{aligned} \mathcal{C}' = \mathcal{C} \cup \{ & \mathbf{Oper} \text{ Op } (e : \text{Entry}; Q : \text{Queue}; A : \text{Static_Array}); \\ & \mathbf{ensures} \ Q \neq \Lambda \wedge e = \#A(0) \wedge \\ & A = \lambda j : \mathbb{Z}, \text{ if } |Q| = j \text{ then } \#e \text{ else } \#A(j); \} \setminus \end{aligned}$$

Denoting the `ensures` clause as ψ , then $\mathcal{C}' \setminus \text{SFV}(\psi)$ yields: $\{Q, \#A, A, \#e, e\}$. Note that constants like $\Lambda, |\bullet|$, etc. and bound variables j were excluded, while specification (program) variables (e.g. $e, \#e$) were included.

4.2 Sequent Calculus Review

Formally, the final `Confirm` assertion (which always terminates a fragment of assertive code) is represented as a conjunction of Gentzen-style sequents $\mathbf{G}_1, \dots, \mathbf{G}_n$, each of which constitute the final VCs produced by our goal-directed program proof process.

Each Gentzen sequent has the form $\mathbf{G} \equiv \varphi_1, \dots, \varphi_m \vdash \psi_1, \dots, \psi_n$ where m and n are non-negative and $\varphi_1, \dots, \varphi_m, \psi_1, \dots, \psi_n$ denote sets of wffs within a given sequent’s *antecedent* (Γ) and *succedent* (Δ), respectively. The wffs in these sets are added from different specification contexts based on our proof rules. Each sequent \mathbf{G} semantically adheres to the usual interpretation: $\bigwedge_{i=1}^m \varphi_i \vdash \bigvee_{j=1}^n \psi_j$.

⁴Outfix and infix style applications are also accepted, though we omit these for brevity

⁵Note that this effectively makes the specification language higher order, as one can write, e.g., $\forall f : T \rightarrow T, f(x)$; we forego discussion of this and its ramifications on proof automation—as this paper is primarily concerned with tools and VC-generation

$$\begin{array}{ccc}
\text{NotLeft} \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta} & \text{AndLeft} \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} & \text{OrLeft} \frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta} \\
\text{NotRight} \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta} & \text{AndRight} \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} & \text{OrRight} \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}
\end{array}$$

Figure 3: Standard sequent reduction rules for \neg , \wedge , and \vee .

To simplify the VCs, we apply the standard sequent reduction rules shown in Fig. 3 to the antecedents and succedents of the sequents that make up the final `Confirm` until they contain only atomic formulas.

We also fix convenience functions $\text{SFV}_{\text{set}}(S)$ which extracts the sfv set from the wffs in a given set S , and $\text{SFV}_{\vdash}(\text{seq})$ which extracts the sfv set from each side of a given sequent, seq .

4.3 Goal-Directed Proof Rule Application

Once assertive code has been constructed, the approach we use to generate VCs is *goal-directed* and is illustrated at a high level in Fig. 4.

Starting with the penultimate statement (prior to the final `Confirm` conjunction), each statement is eliminated one at a time via the application of its corresponding proof rule. After each statement rule is applied, sequent reduction rules and others (such as rules for handling equality or theory-specific rewrite rules) are then applied to further simplify the formulae in each sequent.

After all statements have been eliminated via application of their corresponding rules, the conjuncted sequents in the final `Confirm` are broken apart and sent off to RESOLVE's in-house congruence closure prover for verification [11, 15]. The particulars of the prover are not directly relevant to this paper. Prover backends with which we have experimented include specialized decision procedures [1] as well as SMT solvers [30] based on Z3 [24].

5 Proof Rules for Forming Parsimonious VCs

In this section, proof rules for generating *parsimonious* VCs are formalized and illustrated on a fragment of example assertive code. As with the sequent reduction rules, in the proof rules that follow, it is necessary and sufficient to prove what is above the line in order to prove what follows below the line.

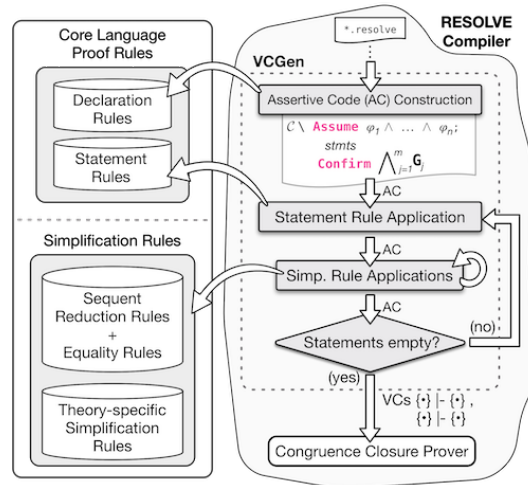


Figure 4: RESOLVE's VC generation scheme.

Confirm Rule

Confirm clauses add a sequent with a single goal to the final sequent conjunction. This, and the “assume” rule that follows in the next section, represent an intermediate type of assertion which generally arise from the application of other (larger) proof rules. For example, applying RESOLVE’s call rule generates (within the assertive code) a Confirm clause asserting that the called operation’s argument-specialized precondition ϕ holds. The confirm rule is given below:

$$\text{Confirm} \frac{\mathcal{C} \setminus c; \text{Confirm} \wedge \Psi \cup \{\vdash \phi\};}{\mathcal{C} \setminus c; \text{Confirm} \phi; \text{Confirm} \wedge \Psi;}$$

Here and in subsequent rules we employ the turnstile \vdash as a shorthand sequent constructor. For example, $\vdash \phi$ denotes a fresh sequent with no antecedents and a single wff ϕ in its succedent. In cases where we wish to add ϕ to the antecedent of some non-empty sequent s , we will write $\Gamma_s, \phi \vdash \Delta_s$ where Γ_s and Δ_s refer to sets of existing wffs in the antecedent and succedent of s , respectively.

Example 5.1. Recalling the math string notation outlined in section 2, suppose the system generated the following assertive code fragment:

```
Assume |S| ≤ 2 ∧ |T| ≤ 2 ∧ S = Λ ∧ T = ⟨1⟩ ∘ ⟨2⟩;
Confirm (S = Λ ∨ T = Λ) ∧ (|S| + |T| = 2);
Confirm ∧ Ψ;
```

Application of the Confirm rule yields:

```
Assume |S| ≤ 2 ∧ |T| ≤ 2 ∧ S = Λ ∧ T = ⟨1⟩ ∘ ⟨2⟩;
Confirm ∧ Ψ ∪ { {} ⊢ {(S = Λ ∨ T = Λ) ∧ (|S| + |T| = 2)} };
```

The VC Generator then simplifies the final confirm with an application of AndRight then OrRight to the first (and only) sequent’s succedent in the set Ψ , yielding:

```
Assume |S| ≤ 2 ∧ |T| ≤ 2 ∧ S = Λ ∧ T = ⟨1⟩ ∘ ⟨2⟩;
Confirm ∧ Ψ ∪ { {} ⊢ {S = Λ, T = Λ} } ∪ { {} ⊢ {|S| + |T| = 2} };
```

In future assertive code listings, we omit the set Ψ for brevity.

Parsimonious Assume Rule

Assume clauses add antecedents to sequents in the final Confirm. The key to the new parsimonious rule, however, is to only add givens that are “relevant” to a given sequent. Conservative variants of our VC generator simply added each conjunct of an encountered assume clause to the list of givens for each VC—resulting in VCs with many redundant givens (e.g., Sect. 3). Below is our parsimonious variant of the original rule.

$$\text{ParsimoniousAssume} \frac{\mathcal{C} \setminus c; \text{Confirm} \wedge \sigma(\Psi, \phi)}{\mathcal{C} \setminus c; \text{Assume} \phi; \text{Confirm} \wedge \Psi;}$$

The core addition to our revised rule (aside from the usage of sequents) is the addition of a parsimonious “selection” function σ that is applied to the set of existing sequents Ψ in the final Confirm and the formula ϕ that is being assumed—i.e., $\sigma(\Psi, \phi)$. We define this selection function as follows:

$$\begin{aligned} \sigma : ((S : \wp(\mathbf{Sqnt}_{\mathcal{V}})) \times (\psi : \mathbf{Form}_{\mathcal{V}})) &\rightarrow \wp(\mathbf{Sqnt}_{\mathcal{V}}) \triangleq \\ &\{s : S \mid \forall \phi : \langle \psi \rangle, \\ &\text{if FVC}(\phi, \langle \psi \rangle) \cap \text{SFV}_{\vdash}(s) \neq \emptyset \text{ then } s = \Gamma_s, \phi \vdash \Delta_s \text{ else } s = \Gamma_s \vdash \Delta_s\}. \end{aligned}$$

Specifically, σ takes a set of existing sequents S along with a formula ψ and produces a set of (potentially modified) sequents. Here, the $\langle\langle\bullet\rangle\rangle$ operator is used to split ψ into a list of conjuncts. The comprehension in the body of σ tests whether or not the transitive closure of all free variables of ϕ appearing across of the collection of any conjuncted clauses in ψ (computed via a free-var closure function $\text{FVC}(\phi, \langle\langle\psi\rangle\rangle)$) intersects with the free variables obtained from the existing sequent s . If the intersection is non-empty, ϕ is added to the antecedent of s , otherwise s remains unchanged.

While the benefits of the parsimonious assume rule will not be immediately evident from example 5.1, its impact on the number of givens in VCs generated by larger programs and algorithms will be examined further in Sect. 7.

Example 5.2. To illustrate why the FVC function is needed, we consider the following assertive code (where each specificationnal free variable is subscripted by a v):

Assume $p(c_v) \wedge c_v = b_v \wedge s_v = x_v$
Confirm $\wedge \{ \} \vdash \{ p(b_v) \}$

Suppose we break apart the Assume clause (i.e., using $\langle\langle\bullet\rangle\rangle$) and consider each conjunct isolation, one at a time, from left to right. The sfv set from the first clause, $p(c_v)$ (i.e.: $\{c_v\}$), does not overlap with the sfv set from the sequent in the final confirm ($\{b_v\}$), thus it would not be added as an antecedent. However, the second clause *would* be added to the antecedent (since b_v overlaps). But the first clause—which has been since discarded—is needed along with the second in order to prove the goal (the third is rightfully excluded as it has no overlaps). The FVC closure function solves this problem by combining the sfv set of a given source clause (in this case $p(b_v)$), with any other clauses in the provided list with overlapping specificationnal free variables. For example:

$$\text{FVC}(p(b_v), \langle\langle p(c_v), c_v = b_v, \dots \rangle\rangle) = \{c_v, b_v\}$$

which enables us to add required clauses while still filtering unrelated ones.

Postprocessing Rule: Folding Top Level Equalities

We also introduce a separate assertive code postprocessing/simplification rule that performs substitutions for equalities appearing as top level formulas in a given sequent's antecedent. Here, such equalities must be of the general form $v = t$, where $v \in \text{SFV}_{\text{set}}(\Gamma - \{v = t\})$ ⁶.

$$\text{ApplyEqLeft} \frac{\mathcal{C} \setminus \text{Confirm} \wedge \varphi_i[v \rightsquigarrow t] \vdash \delta_j[v \rightsquigarrow t]}{\mathcal{C} \setminus \text{Confirm} \wedge \varphi_1, \dots, \varphi_n, v = t \vdash \delta_1, \dots, \delta_m} \text{ where } \varphi_i \in \Gamma, \delta_j \in \Delta \text{ and } i \leq |\Gamma|, j \leq |\Delta|$$

Returning to example 5.1, since ApplyEqLeft requires $v = t$ to appear as a top-level formula in the antecedent (both of which are currently empty) and that all assertive statements prior to the final confirm clause are eliminated, the rule cannot yet be applied. Thus, we must first apply the ParsimoniousAssume rule which yields:

Confirm \wedge
 $\{ |S| \leq 2, |T| \leq 2, S = \Lambda, T = \langle 1 \rangle \circ \langle 2 \rangle \} \vdash \{ S = \Lambda, T = \Lambda \}$
 $\{ |S| \leq 2, |T| \leq 2, S = \Lambda, T = \langle 1 \rangle \circ \langle 2 \rangle \} \vdash \{ |S| + |T| = 2 \};$

⁶The notation $\varphi[v \rightsquigarrow t]$ in the rule denotes the substitution of a variable v for a term t in clause φ

Following this, the VC generator’s simplifier then applies any applicable sequent based reduction rules,⁷ which, in this case, consists of four back-to-back applications of ApplyEqLeft (two for each sequent), resulting in:

$$\begin{array}{l} \text{Confirm } \wedge \\ \{|\Lambda| \leq 2, |\langle 1 \rangle \circ \langle 2 \rangle| \leq 2\} \vdash \{\Lambda = \Lambda, (\langle 1 \rangle \circ \langle 2 \rangle) = \Lambda\} \\ \{|\Lambda| \leq 2, |\langle 1 \rangle \circ \langle 2 \rangle| \leq 2\} \vdash \{|\Lambda| + |\langle 1 \rangle \circ \langle 2 \rangle| = 2\}; \end{array}$$

The first sequent is provable via the trivially true equality: $\Lambda = \Lambda$ (recall that since each wff in the succedent is disjuncted, this trivial equality makes the entire succedent true). The second follows from application of various corollaries available in our theory of strings including, e.g.:

$$\text{Corollary Len1: } \forall \alpha, \beta : \text{SStr}, |\alpha \circ \beta| = |\alpha| + |\beta|;$$

as well as results involving natural numbers, +, and the base case of our string theory’s inductively defined length $|\bullet|$ function (which establishes $|\Lambda| = 0$).

Note that while one could conceivably apply additional rules (such as reflexivity) to “close/prove” sequents on the fly during VC generation, we instead choose dispatch each VC only at the proving stage for consistency and reporting purposes—e.g.: displaying results after a “prove” button is pressed. Otherwise, a proved, missing VC may be surprising or misleading to a learner.

Improving Verifier Feedback for Contradictory Code

The process of removing irrelevant givens requires care, as it can lead to incompleteness that negatively impacts verifier feedback. Consider, e.g, the contradictory fragment of code starting with (A) in Fig. 5.

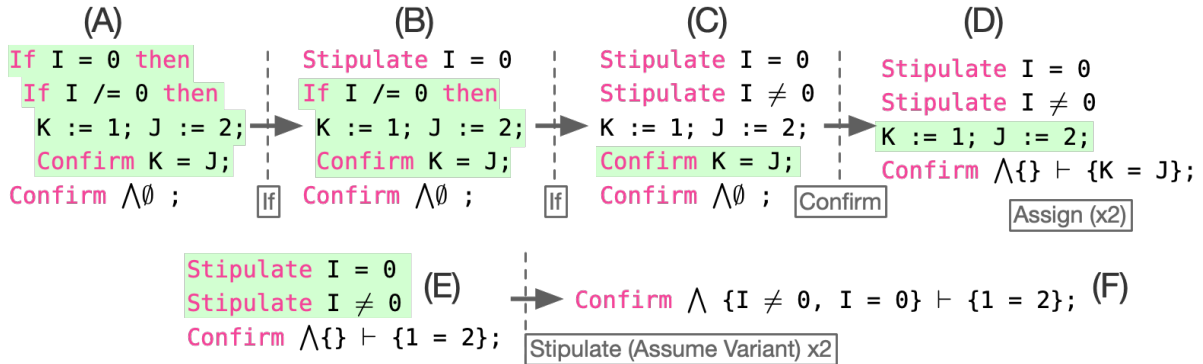


Figure 5: Deriving a vacuously true sequent from contradictory nested if-statements using a variant of the Assume rule called Stipulate

While the VC generator sets up assertive code for all nominal paths through the body, Fig. 5 focuses on the derivation where the innermost if-statement’s body is presumed to execute. Note that each if-statement’s condition is translated by the system into a **Stipulate** statement—meaning any control predicates resulting from the application of proof rules to **If**, **While**, etc. statements always get included as antecedents in the final confirm’s sequents (regardless of variable overlap). To see why **Stipulate** is needed, consider step (E) in Fig. 5 and replace each **Stipulate** keyword with **Assume** and proceed with the derivation. Since there are no overlaps between the assumed clauses and the final confirm, each

⁷Beyond the traditional statement rules, we broadly categorize any rule involving the \vdash meta operator as a *sequent* rule

would be discarded, leaving the unprovable sequent $\vdash \{1 = 2\}$ to timeout during a proof attempt. This is not advantageous from a feedback perspective, as student code often contains similar contradictions and unreachable logic (though perhaps less obvious ones). Under this approach, vacuously true VCs can be flagged and reported to the user—identifying suspect code.

6 Sequent-Based VC Generation in RESOLVE Studio

In this section, using a sorting example, we provide an overview of the VC reporting, simplification, and derivation tracing features included in our second environment, RESOLVE Studio.

6.1 Example: Fully Generic Sorting

For a more complex example, we consider the specification and implementation of a generic sorting algorithm for queues. However, rather than adding a sort operation to the `Queue_Template` concept (which would require updating each existing implementation) we instead use an *enhancement* module—which allow users to write layered code for secondary operations using the primary operations of the base concept (in this case, `Queue_Template`). The queue `Sorting_Capability` enhancement is below.

```

Enhancement Sorting_Capability (Def  $\trianglelefteq$ : Entry  $\times$  Entry  $\longrightarrow$   $\mathbb{B}$ ) for Queue_Template;
  uses Basic_Ordering_Theory, String_Theory with Occ_Tly_Permute_Ext;
  requires Is_Total_Preordering( $\trianglelefteq$ );

  Operation Sort (updates Q : Queue);
    ensures Q Is_Permutation #Q  $\wedge$  Is_Cfml_w(Q,  $\trianglelefteq$ );
end Sorting_Capability;

```

The enhancement is parameterized by an (abstract) binary predicate \trianglelefteq that determines ordering of the queue’s entries. The module level pre-condition subsequently **requires** that the relation passed is a total preordering—i.e.: it must be both total and transitive:

$$(\trianglelefteq\text{-total}) \forall x, y : \text{Entry}, x \trianglelefteq y \vee y \trianglelefteq x \quad (\trianglelefteq\text{-trans}) \forall x, y, z : \text{Entry}, x \trianglelefteq y \Rightarrow y \trianglelefteq z \Rightarrow x \trianglelefteq z.$$

The `Sort` operation takes a single queue, `Q`, and **ensures** two properties: (1) that the resulting queue is a permutation of (exactly) the incoming queue’s entries, and (2) that the ordering of all entries in the outgoing queue is conformal with (`Is_Cfml_w`) the \trianglelefteq ordering predicate. Both the permutation and conformality predicates were imported from `String_Theory`, augmented with a string occurrence tallying and permutation theory extension containing the required predicates.

For more information on the predicates that make up the sorting specification—and the way in which we define them in RESOLVE theories (including aspects of the math type system)—please consult [33].

6.2 An Annotated Implementation

Fig. 6 shows a selection sorting implementation of `Sorting_Capability` loaded (with VCs already generated) in RESOLVE Studio.

Note that enhancement implementations are oblivious to any one particular implementation of the base concept, and must be written in terms of either local operations and/or by calling the primary operations of the base concept. The implementation iterates over the input queue, extracts the minimum

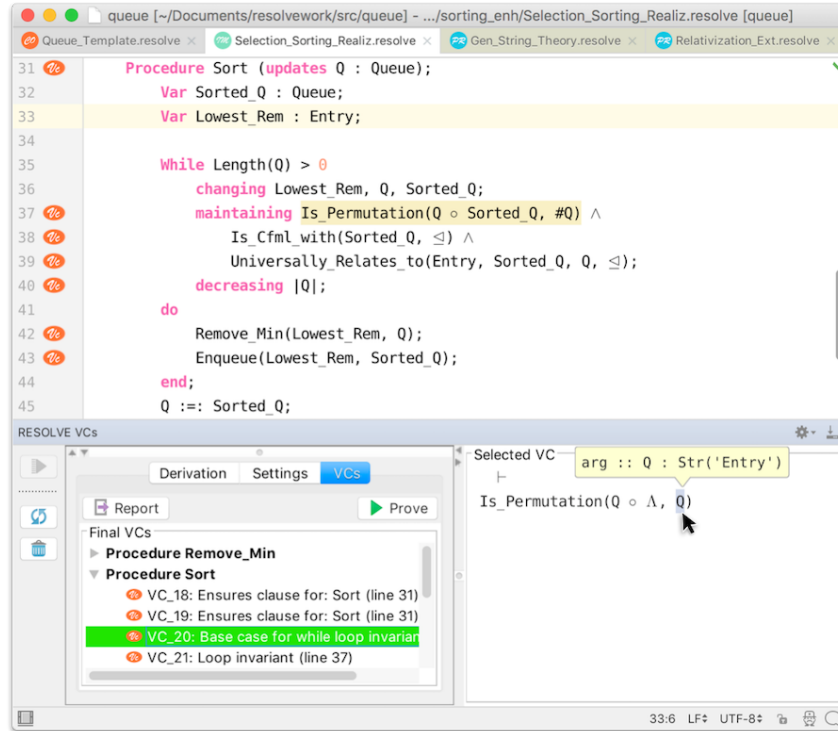



Figure 6: Selection sorting impl. in RESOLVE Studio; viewing the VC for the loop invariant’s base case

entry in each iteration via the local procedure `Remove_Min`, then enqueues it onto a temporary queue, `Sorted_Q` (which holds the entries ordered thus far). The loop is annotated in several different parts: a **changing** clause indicates the variables being updated in the loop (which can help simplify certain styles of invariants), a **decreasing** clause (a progress metric for proving termination), and a loop invariant communicated through the **maintaining** clause. The invariant, shown in Fig. 6, is summarized below:

- The first conjunct states that the concatenation of elements in the (temporary) `Sorted_Q` and `Q` constitute the entirety of the elements being sorted.
- The second conjunct states that the `Sorted_Q`’s elements are ordered w.r.t. the \leq relation.
- The last conjunct states that every element in `Sorted_Q` is related by \leq to every element in `Q`. In other words: that all entries in `Sorted_Q` “precede” the remaining entries in `Q`.

6.3 Verification

The verification process is started by invoking the “Generate VCs” action. Once VCs are generated, clicking any of the  buttons opens a tool menu wherein users can select any VCs arising from that particular construct or line of code (Fig. 7, leftmost).

Once a VC is selected, the IDE automatically navigates to the VC in question in the tool window, displaying the goal and givens. Selecting any particular VC in the “Final VCs” window also highlights the assertion, statement, or construct that generated it within the code. Pressing the prove button invokes the verifier, updating the badges with a checkmark, warning, or a timeout/failure icon.

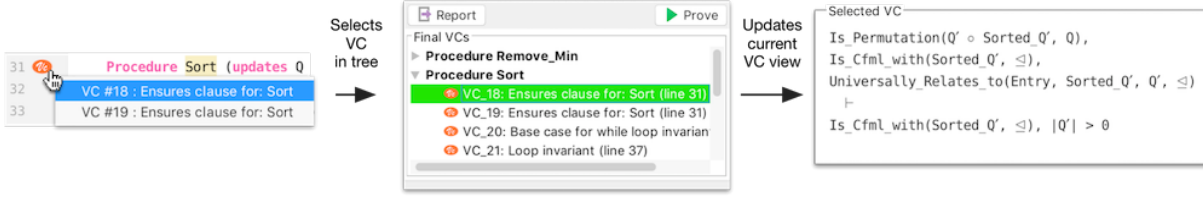


Figure 7: VC selection process in RESOLVE Studio

Advanced Feature: VC Derivation Tracing. RESOLVE Studio also allows experienced users and researchers the ability to trace interactively through the derivation of VCs, starting from initial assertive code, to final VCs (Fig. 8) below.

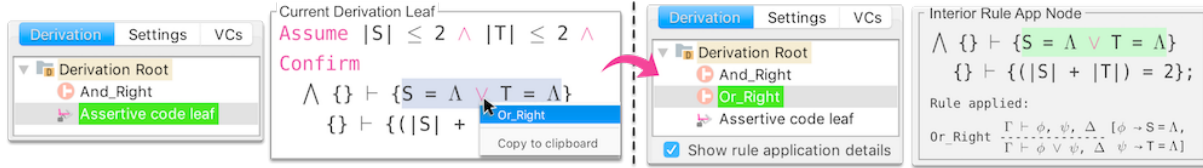


Figure 8: Interactive VC derivation tracing and rule application from example 5.1

We anticipate this being of use to math specialists such as theory developers who seek to add theory-specific simplification rules (e.g., simplifying $S \circ \Lambda$ to S) and observe their affect when applied during derivations to further ‘clean’ resulting VCs and ease their comprehension.

7 Experimental Evaluation

We have tested our revised VC generation technique on a modest battery of extension operations⁸ for a variety of concepts drawn from RESOLVE’s current component library including typical operations for lists, stacks, and queues.

Component	Non-Parsimonious			Parsimonious		
	#VCs	#VCs _{A≥5}	#VCs _{A≥10}	#VCs	#VCs _{A≥5}	#VCs _{A≥10}
List Search	35	35	35	35	16	0
List Reverse (R)	8	8	8	8	0	0
Queue Append (R)	8	8	8	8	0	0
Queue Selection Sort	36	36	36	36	16	0
Stack Copy	24	24	24	24	5	0

Table 1: Comparison of VC generation schemes: non-parsimonious vs parsimonious.

Table 1 compares VCs generated without and with the parsimonious scheme. For each, the first column contains the total number of VCs generated (#VCs), the number of VCs generated with five or more antecedents (#VCs_{A≥5}), and the number of VCs generated with ten or more antecedents (#VCs_{A≥10}).

⁸Recursive implementations are labeled with an (R)

Note that with the non-parsimonious scheme, all VCs contain ten or more antecedents, while under the parsimonious scheme none contain more than ten. Since this paper has been written, the parsimonious scheme detailed has also been incorporated into the compiler backing the web-IDE, meaning students have already benefited from the reduced size of each VC. For a sense of how much shorter the VCs are, consider the (unprovable) VC from our initial example in Fig. 2: $\{1 \leq |\langle e''' \rangle \circ S'|\} \vdash \{S' = \langle e''' \rangle \circ S'\}$.

These experimental results indicate that the parsimonious scheme produces fewer givens for humans to consider when debugging failed VCs. This can be viewed as a measure of *usefulness* for VC comprehension overall, as well as a means of improving *ease of use* when combined with our existing F-IDEs.

8 Related Work

Since VC generation and auto-active program verification are vast topics, this section focuses primarily on work in generating simpler, debuggable VCs presented in an IDE or otherwise. A direct comparison of our work with other efforts is somewhat hindered by the simple fact that others typically “outsource” the process of generating VCs to Intermediate Verification Languages (IVLs) such as Boogie [18]. Outsourcing has its benefits, namely: shifting the burden of VC generation (which can be non-trivial to implement in general) to a separate, reusable tool that multiple languages can target. Notable disadvantages however include high potential for “impedance mismatches” when translating between IVLs [3], or (more commonly) when translating the constructs of the rich ‘high level’ specification language into the ‘lower level’ representation employed by a particular IVL [32]—or vice versa [10]. These mismatches in turn can complicate error reporting efforts, including VC feedback on failed verification attempts. Some related systems are summarized below.

Dafny [19] integrates its toolchain into an IDE for Visual Studio including the Boogie Verification Debugger [18], which translates Z3 generated counter examples into a form suitable for human consumption. AutoProof [31], which also uses Boogie and Z3 for the verification of Eiffel programs, employs “two step” verification to broadly interpret (as opposed to on a per-VC basis) the reasons for verification failures using a combination of traditional modular verification and approximation (such as unrolling). For IDE support, push-button verification and a host of other functionality is provided in the Eiffel Verification Environment.

Why3 [7] is another popular autoactive tool that employs its own IVL (WhyML), to target a number of SMT solvers for automated proof as well as some traditionally interactive systems (such as Coq) for proofs requiring manual steps. The language comes with a verification environment (called WhyIDE) that gives users the ability to browse goals in the current session and perform common transformations such as, for example, splitting a goal into separate conjuncts (which can then be sent to different provers).

The KeY framework [2], which targets full functional verification of Java programs, perhaps gives users the most control over how VCs are structured and ultimately dispatched. As opposed to being reliant on any particular IVL, the language instead employs its own in-house prover that attempts to automatically verify a VC via repeated application of first order sequent reduction rules (among others). In cases where automation fails, the system can also serve as an interactive proof assistant that allows users to systematically apply ‘tactics’ (i.e., *tactics*) to the current proof state—manually guiding the system towards the goal.

9 Conclusions and Future Work

We have discussed the design of two F-IDEs that aim to provide users with feedback suitable for reasoning about code correctness when verification fails. Following a technical overview of our revised proof system, including rules for deriving “parsimonious” VCs, we have demonstrated our presentation of VCs in both the context of the web-IDE (in use for several years) and a newer F-IDE named RESOLVE Studio. A critical avenue for future work remains the ability to efficiently persist verification results to ease usability and to support scalable development of larger, more intricate case studies. Though we have extensive positive feedback from students who have used our online reasoning tools for which the VC generator detailed in this paper now forms the backbone [8, 9, 25], we anticipate additional studies on student usage and debugging of VCs in our second experimental environment, RESOLVE Studio.

Acknowledgments

We thank the RSRG research groups at Clemson and Ohio State who have contributed to this work. Particular thanks are due to Bill Ogden and Joan Krone for their insights on the proof rules detailed.

References

- [1] Bruce Adcock (2010): *Working Towards the Verified Software Process*. PhD thesis, The Ohio State University.
- [2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt & Mattias Ulbrich (2016): *Deductive Software Verification - The KeY Book - From Theory to Practice*. LNCS 10001, Springer, Cham, doi:10.1007/978-3-319-49812-6.
- [3] Michael Ameri & Carlo A. Furia (2016): *Why Just Boogie?* In Erika Ábrahám & Marieke Huisman, editors: *IFM 2016*, Springer International Publishing, Cham, pp. 79–95, doi:10.1007/978-3-319-33693-0_6.
- [4] Charles T. Cook, Svetlana Drachova-Strang, Yu-Shan Sun, Murali Sitaraman, Jeffrey C. Carver & Joseph E. Hollingsworth (2013): *Specification and reasoning in SE projects using a Web IDE*. In: *26th International Conference on Software Engineering Education and Training, CSEE&T 2013, San Francisco, CA, USA, May 19-21, 2013*, pp. 229–238, doi:10.1109/CSEET.2013.6595254.
- [5] Charles T. Cook, Heather Harton, Hampton Smith & Murali Sitaraman (2012): *Specification Engineering and Modular Verification Using a Web-integrated Verifying Compiler*. In: *ICSE*, ACM, pp. 1379–1382, doi:10.1109/ICSE.2012.6227243.
- [6] Charles T. Cook, Heather K. Harton, Hampton Smith & Murali Sitaraman (2012): *Specification engineering and modular verification using a web-integrated verifying compiler*. In Martin Glinz, Gail C. Murphy & Mauro Pezzè, editors: *ICSE 2012*, IEEE Computer Society, pp. 1379–1382, doi:10.1109/ICSE.2012.6227243.
- [7] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: *ESOP 2013*, LNCS 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.
- [8] Megan Fowler, Eileen T. Kraemer, Murali Sitaraman & Joseph E. Hollingsworth (2021): *Tool-Aided Loop Invariant Development: Insights into Student Conceptions and Difficulties*. In Carsten Schulte, Brett A. Becker, Monica Divitini & Erik Barendsen, editors: *ITiCSE 2021*, ACM, pp. 387–393, doi:10.1145/3430665.3456351.
- [9] Megan Fowler, Eileen T. Kraemer, Yu-Shan Sun, Murali Sitaraman, Jason O. Hallstrom & Joseph E. Hollingsworth (2020): *Tool-Aided Assessment of Difficulties in Learning Formal Design-by-Contract Assertions*. In Jürgen Mottok, editor: *ECSEE 2020*, ACM, pp. 52–60, doi:10.1145/3396802.3396807.

- [10] Carlo A. Furia, Christopher M. Poskitt & Julian Tschannen (2015): *The AutoProof Verifier: Usability by Non-Experts and on Standard Code*. In Catherine Dubois, Paolo Masci & Dominique Méry, editors: *F-IDE 2015*, pp. 42–55, doi:10.4204/EPTCS.187.4.
- [11] Smith Hampton (2013): *Engineering Specifications and Mathematics for Verified Software*. PhD Dissertation, Clemson University.
- [12] Douglas E. Harms & Bruce W. Weide (1991): *Copying and Swapping: Influences on the Design of Reusable Software Components*. *IEEE Trans. Softw. Eng.* 17(5), pp. 424–435, doi:10.1109/32.90445.
- [13] Heather Harton (2011): *Mechanical and Modular Verification Condition Generation for Object-Based Software*. PhD Dissertation, Clemson University.
- [14] Wayne D. Heym, Paolo A. G. Sivilotti, Paolo Bucci, Murali Sitaraman, Kevin Plis, Joseph E. Hollingsworth, Joan Krone & Nigamanth Sridhar (2017): *Integrating Components, Contracts, and Reasoning in CS Curricula with RESOLVE: Experiences at Multiple Institutions*. In Hironori Washizaki & Nancy Mead, editors: *CSEET&T, IEEE*, pp. 202–211, doi:10.1109/CSEET.2017.40.
- [15] Nabil M. Kabbani, Daniel Welch, Caleb Priester, Stephen Schaub, Blair Durkee, Yu-Shan Sun & Murali Sitaraman (2015): *Formal Reasoning Using an Iterative Approach with an Integrated Web IDE*. In: *F-IDE 2015*, pp. 56–71, doi:10.4204/EPTCS.187.5.
- [16] Jason Kirschenbaum, Bruce M. Adcock, Derek Bronish, Hampton Smith, Heather K. Harton, Murali Sitaraman & Bruce W. Weide (2009): *Verifying Component-Based Software: Deep Mathematics or Simple Book-keeping?* In Stephen H. Edwards & Gregory Kulczycki, editors: *ICSR 2009, LNCS 5791*, Springer, Berlin, Heidelberg, pp. 31–40, doi:10.1007/978-3-642-04211-9_4.
- [17] Eileen T. Kraemer, Murali Sitaraman & Joseph E. Hollingsworth (2018): *An Activity-Based Undergraduate Software Engineering Course to Engage Students and Encourage Learning*. In: *Proceedings of the 3rd European Conference of Software Engineering Education, ECSEE 2018, Seon Monastery, Bavaria, Germany, June 14-15, 2018*, pp. 18–25, doi:10.1145/3209087.3209100.
- [18] Claire Le Goues, K. Rustan M. Leino & Michał Moskal (2011): *The Boogie Verification Debugger (Tool Paper)*. In Gilles Barthe, Alberto Pardo & Gerardo Schneider, editors: *SEFM 2011, LNCS*, Springer, Berlin, Heidelberg, pp. 407–414, doi:10.1007/978-3-642-24690-6_28.
- [19] K. Rustan M. Leino (2013): *Developing verified programs with Dafny*. In David Notkin, Betty H. C. Cheng & Klaus Pohl, editors: *ICSE 2013, IEEE*, pp. 1488–1490, doi:10.1109/ICSE.2013.6606754.
- [20] K. Rustan M. Leino & Michal Moskal (2010): *Usable Auto-Active Verification*. In Tom Ball, Lenore Zuck & Natarajan Shankar, editors: *UV 2010*. Available at <http://fm.cs1.sri.com/UV10/>.
- [21] K. Rustan M. Leino & Clément Pit-Claudel (2016): *Trigger Selection Strategies to Stabilize Program Verifiers*. In Swarat Chaudhuri & Azadeh Farzan, editors: *CAV 2016, LNCS 9779*, Springer, pp. 361–381, doi:10.1007/978-3-319-41528-4_20.
- [22] Nicodemus M. J. Mbwambo (2017): *A Well-Designed, Tree-Based, Generic Map Component to Challenge the Process Towards Automated Verification*. Master’s Thesis, Clemson University.
- [23] Bertrand Meyer (1992): *Applying “Design by Contract”*. *Computer* 25(10), pp. 40–51, doi:10.1109/2.161279.
- [24] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS 2008, LNCS 4963*, Springer, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.
- [25] C. Priester, Y. S. Sun & M. Sitaraman (2016): *Tool-Assisted Loop Invariant Development and Analysis*. In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pp. 66–70, doi:10.1109/CSEET.2016.28.
- [26] Kalyan C. Regula, Hampton Smith, Heather Keown, Jason O. Hallstrom, Nigamanth Sridhar & Murali Sitaraman (2012): *A Case Study in Verification of Embedded Network Software*. In Alwyn Goodloe & Suzette Person, editors: *NFM 2012, LNCS 7226*, Springer, pp. 433–448, doi:10.1007/978-3-642-28891-3_38.

- [27] Iman Saleh (2015): *Formalizing Data-Centric Web Services*. Web-Scale Workflow and Analytics, Springer, doi:10.1007/978-3-319-24678-9.
- [28] Murali Sitaraman, Bruce M. Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather K. Harton, Wayne D. Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith & Bruce W. Weide (2011): *Building a Push-Button RESOLVE Verifier: Progress and Challenges*. *Formal Aspects of Computing* 23(5), pp. 607–626, doi:10.1007/s00165-010-0154-3.
- [29] Yu-Shan Sun (2018): *Towards Automated Verification of Object-Based Software with Reference Behavior*. PhD Dissertation, Clemson University.
- [30] Aditi Tagore, Diego Zaccai & Bruce W. Weide (2012): *Automatically Proving Thousands of Verification Conditions Using an SMT Solver: An Empirical Study*. In: *NFM 2012*, pp. 195–209, doi:10.1007/978-3-642-28891-3_20.
- [31] Julian Tschannen, Carlo A. Furia, Martin Nordio & Nadia Polikarpova (2015): *AutoProof: Auto-Active Functional Verification of Object-Oriented Programs*. In Christel Baier & Cesare Tinelli, editors: *TACAS 2015*, LNCS 9035, Springer, Berlin, Heidelberg, pp. 566–580, doi:10.1007/978-3-662-46681-0_53.
- [32] Mark Utting, David J. Pearce & Lindsay Groves (2017): *Making Whiley Boogie!* In Nadia Polikarpova & Steve Schneider, editors: *IFM 2017*, LNCS 10510, Springer, pp. 69–84, doi:10.1007/978-3-319-66845-1_5.
- [33] Daniel Welch (2019): *Scaling Up Automated Verification: A Case Study and a Formalization IDE for Building High Integrity Software*. PhD Dissertation, Clemson University.
- [34] Daniel Welch & Murali Sitaraman (2017): *Engineering and Employing Reusable Software Components for Modular Verification*. In Goetz Botterweck & Claudia Werner, editors: *ICSR 2017*, LNCS 10221, Springer, pp. 139–154, doi:10.1007/978-3-319-56856-0_10.