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

Yu-Shan Sun and Daniel Welch and Murali Sitaraman

Technical Report RSRG-19-01
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

January 2019

Copyright © 2019 by the authors. All rights reserved.

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

Yu-Shan Sun, Daniel Welch, and Murali Sitaraman

School of Computing, Clemson University
Clemson, SC 29634, USA
{yushans,dtwelch,msitara}@clemson.edu

Abstract. This paper presents two distinct Formal Integrated Development Environments (F-IDEs) focusing on how they aid human reasoning when verification fails. Both environments are modular and facilitate reasoning about the full behavior of object-based code. The first environment, henceforth referred to as the web-IDE, has been used for several years to teach aspects of formal specification and verification, including why verification conditions (VCs) arise and how to use them in reasoning when verification fails. The second F-IDE, named RESOLVE Studio, is a more fully-fledged environment with features consistent with modern environments and it allows sophisticated developers or researchers to go beyond reasoning about code and study theory developments and proof steps of code correctness. RESOLVE Studio is backed by an alternative sequent-based VC generator and has received limited experimentation for theory development in a graduate course. Both environments, and the VC generation methods detailed in this paper are based on RESOLVE—an integrated specification and programming language. But the principles of alternative VC generation methods, IDE features, and the observations about their impact on novices and experienced developers 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* [15] specification and verification languages such as Why3 [6], Dafny [14], RESOLVE [19], and AutoProof [22]—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 VC details differently. For example, Dafny tends to report assertion failures through Z3 [17] generated counter examples, 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 support to users of all experience levels when verification fails.

Admittedly when VCs are correct (meaning, both necessary and sufficient) and do not gratuitously complicate the task of the underlying provers, 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 that when verification does fail, they represent a useful starting point for both novices and experts alike when attempting to identify which fixes are needed at the source code or specification level (which is why they must be human readable). The second reason is a more direct consequence of teaching beginning software engineering students code verification principles. That is—even when verification succeeds—students should be able to examine the mathematical details of an actual VC, understand where it came from and why it was provable (e.g., making connections to concepts already learned in discrete math courses). Correspondingly, easing the human reasoning process (through simpler, smaller VCs) and better supporting student inquiry into what happens when a program is verified are primary motivations for the technical VC generation enhancements and front-end tools discussed in this paper.

In this paper we present two F-IDEs that serve as the primary means of viewing VCs, proving, and debugging code that has failed to verify. The language targeted by the environments is RESOLVE [19]—an imperative, object-based integrated specification and programming language. RESOLVE has been utilized in graduate and undergraduate computer science education for nearly two decades, and over 25,000 students (a population that continues to grow) have experimented with versions of the language its tools [5,10,11].

The first of the two environments, the web-IDE (shown in Fig. 1, top), runs a version of the RESOLVE compiler on the back-end and supports persistent user login settings along with project templates that can be uploaded and filled in—features which have proven useful in the classroom setting.[1] As this particular IDE has been in use for almost 10 years, over 500 students have employed it for both reasoning activities and software engineering projects [4,18,12].

The second, newer F-IDE is a desktop based environment named RESOLVE Studio (shown in Fig. 1, bottom). Built on top of the JetBrains IDE platform,[2] this environment combines the usual modern IDE amenities users have come to expect (such as contextual reference completions for keywords, contracts, and code) with a newer version of the RESOLVE compiler. In particular, this newer version of the compiler provides a correspondingly revised VC generator that eliminates excess givens for each VC and permits simplification rules (drawn RESOLVE's collection of rich mathematical theory developments) to be incrementally applied during the VC derivation process. Additionally, in an effort to help guide the design and application of such rules, RESOLVE Studio also permits users to interactively derive

[1] The web-IDE and RESOLVE's current component library is located at: https://resolve.cs.clemson.edu/teaching/

[2] https://www.jetbrains.com/

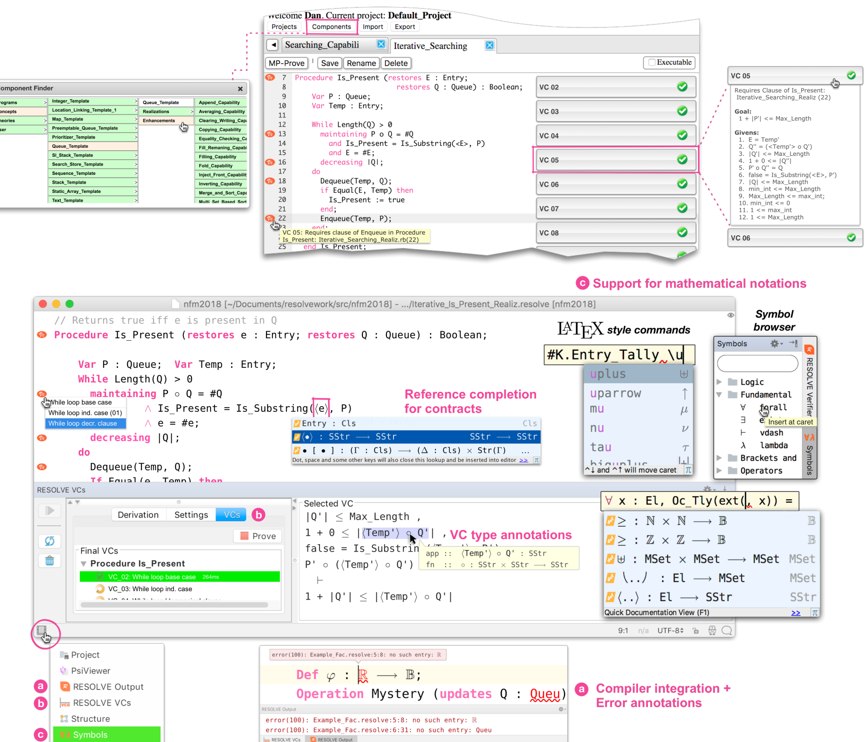

Fig. 1: RESOLVE's web-based F-IDE (top), and RESOLVE Studio (bottom).

VCs from Hoare style triples. Among other benefits of the derivations are benefits for researchers who may design new or modify existing RESOLVE theory developments. This environment has seen a (limited) usage in a graduate level programming languages course to develop, experiment, and typecheck some modest mathematical theories.

Contributions and Outline. The contributions are the following. The first is a general demonstration of how formal reasoning about software correctness can be supported in the context of two F-IDEs—each with features and design decisions that are targeted towards different audiences with different levels of experience—a luxury few current system afford. The second (more theoretical) contribution is a set of modified Hoare-style proof rules for generating sequent-based VCs in a "parsimonious" manner that omits irrelevant givens, along with a prototype implementation of this revised VC generation technique integrated into RESOLVE Studio. The intent of such simplified VCs is to ease automation as well as improve human reasoning and VC comprehension.

The rest of the paper is organized as follows. Sects. 2 and 3 contain (respectively) some background information on RESOLVE and an illustration of the feedback cur-

rently provided to students through the web-IDE (with an emphasis on notable deficiencies with such feedback). Sect. 4 provides the theoretical treatment of our revised proof rules, while Sect. 5 showcases a prototype implementation of our revised VC generation scheme in action within RESOLVE Studio. Sect. 6 contains an experimental evaluation of our revised VC generator, while Sect. 7 reviews some related work. Sect. 8 wraps up with a summary, conclusions, and future work.

2 RESOLVE Background

RESOLVE [19] is an imperative, object-based programming and specification framework designed to support modular verification of sequential code. Formal specifications in RESOLVE are *model-based*. In particular, this means that each programmatic type carries with it a corresponding abstract mathematical model that is specified in terms of one or more pre-existing (or user-defined) mathematical theories. Operations to manipulate objects of the type are specified strictly in the mathematical terms of such a model. Further, to facilitate modular reasoning, the language enforces a strict separation between the abstract state expressed in interface specifications and executable, realization level code. Thus, realizations 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).

To give a brief idea of RESOLVE's flavor of mathematical models, a bounded queue for instance could be modeled as a finite string of generic entries[3] where the model itself and the functional contracts for its operations employ the usual theory-defined string operators such as length ($|\bullet|$), concatenation ($\circ$), empty string (Λ), string formation ($\langle\bullet\rangle$), and others.[4] And while certainly new, more specific theories can be defined and used, we encourage reuse of existing theories (such as strings) whenever possible.

Readers interested in a more complete description of RESOLVE should consult [8,9,19,20] or one of the numerous case studies [16,21,24] carried out using the language.

3 The Need for Simpler VCs: Examining a Failed VC

While we have seen benefits of even modest enhancements made to our presentation of VCs (such as, for example, showing "more relevant" givens before others and annotating the source of VCs next to the line(s) on which they arise—illustrated in Fig. 1) the complexity of debugging VCs for beginners is considerably increased when they include additional baggage—as is the case here, where the goal carries a large, obfuscating, number of irrelevant givens.

[3] That is, a sequence of values such as $\langle 1,6,2,1\rangle$.

[4] Since the web-IDE currently accepts only ASCII syntax, these operators can also be written as `|*|`, `o`, `Empty_String`, and `<..>`, respectively.

```
 5 Oper Do_Nothing (restores S : Stack);
 6   requires 1 <= |S|;
VC 06: Ensures clause of of Procedure Do_Nothing:
NFM_Ex.rb(5)
 8
 9   Var e : Entry;
10   Var T, U : Stack;
11
12   Pop(e, S);
13   Push(e, T);
14
15   Pop(e, T);
16   Push(e, U);
17
18   Pop(e, U);
19   //Push(e, S);
20 end Do_Nothing;
```

```
VC 06

Ensures Clause of Do_Nothing (from
  'restores' mode parameter 'S'):
  NFM_Ex.rb(5)

Goal:
 S' = S

Givens:
 1. U'' = (<e'> o U')
 2. |U'| <= Max_Depth
 3. U'' = <e''> o U
 4. |<e''> o U| <= Max_Depth
 5. T'' = <e''> o T'
 6. |T'| <= Max_Depth
 7. T'' = <e'''> o T
 8. |<e'''> o T| <= Max_Depth
 9. S = <e'''> o S'
10. |S'| <= Max_Depth
11. U = Empty_String
12. T = Empty_String
13. Entry.Is_Initial(e)
14. 1 <= |S|
15. |S| <= Max_Depth
16. min_int <= Max_Depth
17. Max_Depth <= max_int
18. min_int <= 0
19. 1 <= max_int
20. 1 <= Last_Char_Num
21. 1 <= Max_Char_Str_Len
22. 1 <= Max_Depth
```

Fig. 2: A failed VC in the web-IDE with many extraneous givens.

For illustration purposes, Fig. 2 shows a particularly concocted bad piece of trivial code that fails to verify. Here, verification was initiated by pressing the **MP-Prove** button shown in Fig. 1. VCs are generated for preconditions of invoked operations in addition to the postcondition of the code that is being proved. The process is completed when all VCs are verified or the system times out. Since many student programs often contain errors and fail to verify, in order to provide them quick feedback, the timeout has been set to be minimal on the web-IDE, at the risk of not being able to discharge some otherwise provable VC. Also, once at least three VCs fail to prove, the other VCs are not attempted. Hovering over the VC badges in the gutter shows the details of the relevant VC (including its source). The VC that has failed in this case is merely attempting to prove that the given stack `S` has been restored as per the parameter "mode" `restores` on `S`, which adds the implicit postcondition `S = #S` (where `#S` denotes the *incoming* `S`).

In the next section we provide a formal presentation of the changes made to RESOLVE's proof system to improve the simplicity (and hence) debuggability of VCs such as the one shown in Fig. 2.

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 complete description of the proof system may be found elsewhere [9,20].

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 chosen notation, as assertive code may include any number of preceding statements or assertions such as **Assume** P;

$$\mathcal{C} \backslash\ c; \quad \texttt{Confirm}\ Q;$$

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

The remainder of this section provides the mathematical and syntactic preliminaries necessary for a formal discussion of our revised proof rules, a brief review of the sequent calculus, and finally a formalization of some key rules that drive the generation of *parsimonious* VCs. Lastly, on a typographical note: we'll 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 higher order specification language is organized into several constituent parts. First, a set of type symbols $\mathcal{T}$, which for the purposes of simplicity we assume contains a sort $T : SSet$, where $SSet$ denotes the proper class of all sets.

- A set $\mathcal{X}$ of typed constant symbols $x, y : T \in \mathcal{X}$ where x, y could also have any other types in $\mathcal{T}$.
- A set $\mathcal{F}$ of typed function symbols $f, g, .., h : T_1 \times .. \times T_n \longrightarrow T$ which range over $\mathcal{F}$. Here $T_1 \times .. \times T_n$ represents the domain, while the (non-subscripted) T represents the co-domain. We denote the arity of a given function symbol f as $\mathsf{ar}(f)$.
- Lastly, a set $\mathcal{P}$ of typed predicate symbols $P : T_1 \times .. \times T_n \longrightarrow \mathbb{B}$; which also includes a reserved binary predicate for equality $(=)$ as well as nullary predicate symbols for *true* and *false*.

These sets, when combined, constitute the language's vocabulary $\mathcal{V} = \mathcal{X} \cup \mathcal{F} \cup \mathcal{P}$; where $\mathcal{V}$ can be enriched via the definition of new constants, functions, and predicates in mathematical theories. With these categories fixed, we now define formulas (which denote truth values) and terms (which serve as the fundamental building blocks of formulas).

Definition 1. *The set of formulas and terms of our specification language over vocabulary $\mathcal{V}$ is given by the following abstract syntax.*

$$\mathbf{Form}_{\mathcal{V}} \ni \phi, \psi ::= P(t_1, .., t_{\mathsf{ar}(P)}) \mid true \mid false \mid \neg\phi \mid \phi \circ \psi \mid \mathcal{Q}x_1, .., x_n, \phi \mid t$$

$$\mathbf{Term}_{\mathcal{V}} \ni t, y ::= t_0(t_1, .., t_{\mathsf{ar}(t_0)}) \mid t.y \mid \cdots \mid \mathcal{B}x_1, .., x_n, t \mid (\psi) \mid x$$

where $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$*,* $\mathcal{Q} \in \{\forall, \exists\}$*, and a variable binder symbol* $\mathcal{B} \in \{\lambda, \otimes, \ldots\}$*.*

Specifically, formulas consist of the usual binary connectives and quantifiers while terms permit (respectively) function application,[5], field selectors, arbitrary variable binders, and vars/constants x. As usual, formulas can generally be considered terms that are assigned type $\mathbb{B}$.

Next, we establish our syntax for assertive code fragments.

[5] Outfix and infix style applications are also accepted, though we omit these for the sake of brevity.

Definition 2. *A fragment of assertive code consists of zero or more program statements interleaved with assertive statements of the form:*

$$\mathbf{Stmt}_{\mathcal{V}} \ni s ::= \texttt{Assume}\ \phi; \mid \texttt{Confirm}\ \phi; \mid \texttt{Stipulate}\ \phi; \mid v := t; \mid \cdots \mid id(y_1,..,y_n);$$
$$\mathbf{AsrtCode} \ni a ::= \ s*\ \ \texttt{Confirm} \bigwedge seq+; \qquad \mathbf{Seqnt} \ni seq ::= \Gamma \vdash \Delta$$

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

The statement production rule admits verification language specific statements (including `Assume`, `Confirm`, and `Stipulate` clauses—which we elaborate on further in Sect. 4.4) as well as strictly programmatic ones such as function assignment ($v := t$) and procedure calls ($id(y_1,..,y_n)$). We omit syntax for any remaining programmatic statements for brevity.

Lastly, since the formalization of our parsimonious proof rules in the proceeding section rely on the notion of free variables, we define a "free variable" function $\mathsf{FV} : \mathbf{Term}_{\mathcal{V}} \longrightarrow \wp(\mathbf{Term}_{\mathcal{V}})$:

Definition 3. *The set of free variables of some term θ, denoted $\mathsf{FV}(\theta)$, is defined inductively as follows:*

$$\mathsf{FV}(x) = \{x\} \qquad \mathsf{FV}(t.y) = \mathsf{FV}(t_y) \qquad \mathsf{FV}(t_0(t_1,..,t_n)) = \bigcup_{i=0}^{n} \mathsf{FV}(t_i)$$
$$\mathsf{FV}(\mathcal{Q} \cup \mathcal{B}\, x_1,..,x_n,\phi) = \mathsf{FV}(\phi) \sim \{x_1,..,x_n\}$$

In addition to the FV function defined above, for convenience we also assume the presence of a separate version, $\mathsf{FV}_{\vdash}$, that takes a sequent (as opposed to a term) and produces the collection of free variables across the wffs in Γ and Δ.

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 $\mathbf{G}_1,\cdots,\mathbf{G}_n$, which, when broken apart, constitute the final VCs produced by our goal-directed program proof process. Each **G**entzen sequent in this conjunction has the form $\mathbf{G} \equiv \varphi_1,\cdots,\varphi_m \vdash \psi_1,\cdots,\psi_n$ where m and n are non-negative integers and $\varphi_1,..,\varphi_m$, $\psi_1,..,\psi_n$ denote sets of wffs within a given sequent's *antecedent* and *succeedent*, respectively. The wffs within these sets are added from different specification contexts as dictated by our proof rules. Semantically, each sequent **G** adheres to the usual interpretation, $\bigwedge_{i=1}^{m} \varphi_i \vdash \bigvee_{j=1}^{n} \psi_j$ where the universal conjunction of all wffs on the left entails ($\vdash$) the universal disjunction of all wffs on the right.

In an effort 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 our final `Confirm` until they contain only atomic formulas (i.e., until the logical operators $\wedge$, $\vee$, $\neg$, and $\Longrightarrow$ are eliminated).

$$\textsf{notLeft}\ \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta} \qquad \textsf{notRight}\ \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta} \qquad \textsf{andLeft}\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}$$

$$\textsf{andRight}\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \qquad \textsf{orRight}\ \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}$$

$$\textsf{orLeft}\ \frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta}$$

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

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 high level in Fig. 4. In particular, 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. Following the application of each statement rule, we then apply sequent reduction rules and others (such as rules for handling equality or theory-specific ones).

After eliminating all statements, the conjuncted sequents in the final **`Confirm`** are broken apart and sent off to RESOLVE's in-house congruence closure prover for verification [8,11]. While the particulars of the prover are not directly relevant to this paper, it's worth noting that we've also experimented extensively with specialized decision procedures [1] as well as SMT solvers [21] based on Z3 [17].

All of the proof rules (including the domain specific ones) are formally written in the following style:

$$\textsf{ruleName}\ \frac{H_1 \ \cdots \ H_n}{C}$$

where the conclusion C appearing below the line can be deduced from the one or more hypotheses H_n appearing above the line. Throughout the presentation of the rules we explain each new piece of notation as it is introduced.

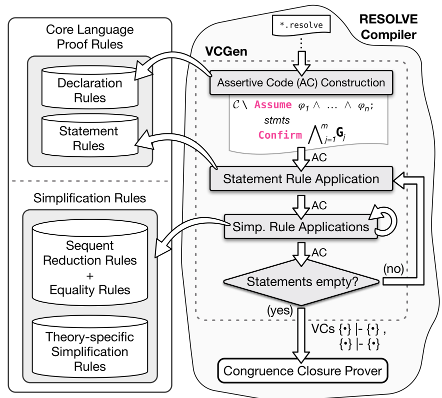

Fig. 4: RESOLVE's VC generation scheme.

4.4 Proof Rules for Forming Parsimonious VCs

Elimination of unnecessary givens and the generation *parsimonious* VCs is incorporated into the backwards derivation process used to generate the VCs. Note that it

is not possible to delay these optimizations as "post-processing" step as we would lose key contextual information about the formulas that make up the final VC (such which construct they came from). The remainder of this section discusses the core rules we use to drive this process.

Confirm Rule. A `Confirm` clause, which adds a sequent with a single goal to the final sequent conjunction, represents an intermediate type of assertion that typically arises from the application of other (larger) proof rules. For example, application of the call rule generates a `Confirm` clause asserting that the operation's (argument-specialized) precondition holds.

$$\textsf{Confirm}\ \frac{\mathcal{C}\setminus c;\quad \texttt{Confirm}\ \bigwedge \Psi \cup \{\vdash \varphi\};}{\mathcal{C}\setminus c;\quad \texttt{Confirm}\ \varphi;\quad \texttt{Confirm}\ \bigwedge \Psi;}$$

Here and in subsequent rules we employ the turnstile $\vdash$ as a shorthand sequent constructor. For example, $\vdash \varphi$ in the Confirm rule above 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 $\Gamma_s, \varphi \vdash \Delta_s$ where Γ_s and Δ_s refer to sets of existing wffs in the antecedent and succedent of s, respectively.

Example. 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 ⋀∅;
```

Application of the Confirm rule (followed by an application of andRight then orRight) to the resulting sequent in the final confirm yields:

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

Parsimonious Assume Rule. `Assume` clauses add a number of constituent assertions (i.e., givens) to the antecedent of each sequent in the final `Confirm`. The key to the new parsimonious rule is to add only givens that are relevant to the formulas already present in a given sequent.

Previous, more conservative variants of our VC generator simply added each conjunct of an encountered assume to the list of givens for each VC—resulting in the blowup in VC size discussed in Sect. 3. Below is our parsimonious variant of the original rule.

$$\textsf{ParsimoniousAssume}\ \frac{\mathcal{C}\setminus c;\quad \texttt{Confirm}\ \bigwedge \sigma(\Psi, \varphi)}{\mathcal{C}\setminus c;\quad \texttt{Assume}\ \varphi;\quad \texttt{Confirm}\ \bigwedge \Psi;}$$

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 to the set of existing sequents Ψ in the final `Confirm` and the formula φ that is being assumed—i.e., $\sigma(\Psi, \varphi)$. We formally define this selection function as follows:

$$\sigma : ((S : \wp(\mathbf{Sqnt}_{\mathcal{V}})) \times (\psi : \mathbf{Form}_{\mathcal{V}})) \longrightarrow \wp(\mathbf{Sqnt}_{\mathcal{V}}) \triangleq \{s : S \mid \forall \phi : \langle\!\langle \psi \rangle\!\rangle, \text{ if } \mathsf{FVC}(\langle\!\langle \psi \rangle\!\rangle, \phi) \cap \mathsf{FV}_{\vdash}(s) \neq \emptyset \text{ then } \Gamma_s, \phi \vdash \Delta_s \text{ else } \Gamma_s \vdash \Delta_s\}.$$

Specifically, σ takes a set of existing sequents S along with a formula ψ and produces a new set of sequents. Here the $\langle\!\langle \bullet \rangle\!\rangle$ operator appearing in the comprehension merely splits ψ into a set of its constituent conjuncts.

The heart of the rule is the conditional in the comprehension's body that tests whether or not the transitive closure of all free variables of ϕ appearing across of the collection of conjuncts of ψ (computed via a free-var closure function $\mathsf{FVC}(\langle\!\langle \psi \rangle\!\rangle, \phi)$) intersects with the free variables obtained from the sequent s. If the intersection is non-empty, ϕ is added to the set of existing antecedents of sequent s, otherwise s remains unchanged.

- *Aside: Computing the Free Variable Closure:* $\mathsf{FVC} : (T : \wp(\mathbf{Term}_{\mathcal{V}})) \times (\varphi : \mathbf{Term}_{\mathcal{V}}) \longrightarrow \wp(\mathbf{Form}_{\mathcal{V}})$ takes a set of terms T along with a source term φ and adds each $t \in T$ as a vertex into an undirected graph G; where edges are added between two terms iff their free variable sets intersect. The function produces the set of free variables of φ and any terms "reachable" from φ.

While the parsimonious assume rule's benefits will not be immediately evident in our small running example, its impact on the number of givens in real VCs will be examined further in Sect. 6.

Handling Top Level Equality Terms. We also introduce a separate 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 $t_1 = t_2$, where $\mathsf{isVar}(t_1)$ and $t_1 \in \bigcup_i \mathsf{FV}(\varphi_i) \vee t_1 \in \bigcup_j \mathsf{FV}(\Delta_j)$.

$$\mathsf{ApplyEqLeft}\ \frac{\mathcal{C} \setminus c; \quad \bigwedge \varphi_i[t_1 \rightsquigarrow t_2] \vdash \Delta_j[t_1 \rightsquigarrow t_2] \text{ where } i \leq n, j \leq m}{\mathcal{C} \setminus c; \quad \bigwedge \varphi_1, \cdots, \varphi_n, t_1 = t_2 \vdash \Delta_1, \cdots, \Delta_m}$$

Based on our abstract syntax, the isVar meta-predicate merely holds whenever the term it receives is a non-literal symbol (such as v) or a selector term (such $x.y.z$).[6]

Returning to our running example, since ApplyEqLeft stipulates that $t_1 = t_2$ appears as a top level formula in the antecedent (both of which are currently empty), the

[6] If a selector chain concludes with an application, such as $x.f(z)$, then $x.f$ is merely a subterm that represents the name of the function that is being applied—the application itself is the root of the overall term.

rule cannot yet be applied. Thus our only option is to apply the ParsimoniousAssume rule which yields:

```
Confirm ⋀
  {|S| ≤ 2, |T| ≤ 2, S = Λ, T = ⟨1⟩ ∘ ⟨2⟩} ⊢ {S = Λ, T = Λ}
  {|S| ≤ 2, |T| ≤ 2, S = Λ, T = ⟨1⟩ ∘ ⟨2⟩} ⊢ {|S| + |T| = 2};
```

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:

```
Confirm ⋀
  {|Λ| ≤ 2, |⟨1⟩ ∘ ⟨2⟩| ≤ 2} ⊢ {Λ = Λ, (⟨1⟩ ∘ ⟨2⟩) = Λ}
  {|Λ| ≤ 2, |⟨1⟩ ∘ ⟨2⟩| ≤ 2} ⊢ {|Λ| + |⟨1⟩ ∘ ⟨2⟩| = 2};
```

The first sequent is easily provable via reflexivity, while the second follows directly from basic theorems and corollaries available in string theory. Note that while we could conceivably apply additional rules (such as reflexivity) to "close/prove" sequents on the fly during VC generation, we nevertheless choose to let the prover itself dispatch all sequents (even the trivial ones) not only to bolster trustworthiness, but to also ensure that the results for each VC/sequent are dispatched in a consistent manner and at a similar time for the benefit of reporting purposes (i.e., after the "prove" button is pressed).

Stipulate Assume Rule. The process of removing "irrelevant" givens requires care, as it can have subtle, negative impacts on verifier feedback. Consider, for example, a scenario in which unrelated givens contradict each other—as they do below.

```
If I = 0 then
  If I ≠ 0 then K := 1; J := 2; Confirm K = J; end;
end;
```

The VC generator will consider all nominal paths through the code above. Thus, even though the code within the inner conditional is not reachable, the verifier will nevertheless unsuccessfully attempt to establish $\vdash 1 = 2$. Note that because I does not overlap with symbols in this sequent, the conditionals—by the design of our parsimonious rules—will not be added as antecedents. Though the code indeed will not prove, this is still not advantageous from a feedback perspective, as student code often contains such contractions (though perhaps ones that are less obvious). So to address this, we employ a so-called **Stipulate** clause for all path conditions. The rule itself is simple: it unconditionally adds the formula stipulated to a sequent's antecedent (regardless of whether or not there are overlapping symbols). Thus, when the sequent reaches the verifier, the contradiction lurking in the antecedent can be detected and reported to the user accordingly.

[7] Note that beyond the traditional rules, we broadly categorize any rule involving the $\vdash$ meta operator as a *sequent* reduction rule

5 Sequent-Based VC Generation in RESOLVE Studio

In this section, using a (slightly) modified version of the example developed in Sect. 4.4, we provide an overview of the VC reporting, simplification, and derivation tracing features provided by our second environment, RESOLVE Studio.

Verification Process. While the verification process and feedback is fundamentally similar to the web-IDE's reporting, this environment incorporates both the revised rules presented in the previous section and makes some key workflow changes. First, RESOLVE Studio splits the verification process into two steps: (1) generation of VCs, and (2) initiation of RESOLVE's "push button" verifier. Fig. 5 and Fig. 1 show the first and second step of this process in action, respectively.

```
5   // Code to generate the ex. from Sect. 4.4
6   Operation Simple (updates S, T : Stack);
7    requires
8      |S| ≤ 2 ∧ |T| ≤ 2 ∧
9      S = Λ ∧ T = (⟨1⟩ ∘
10   ensures
11     (S = Λ ∨ T = Λ) ∧
12     (|S| + |T| = 2);
13  Procedure
14    //skip;
15  end Simple;
```

Fig. 5: Opening the VC derivation view.

Upon clicking the **Derive VCs** button in the contextual popup menu, a tool window opens along the bottom of the IDE that lists all assertive code blocks generated from the current module that must first be fully derived, then verified. Since the current example consists of only a single operation, the **VC Producing Constructs** view contains only a single item (identified with the icon). The **Derivation** tab window immediately below tracks the derivation steps of the currently selected construct as a tree, while the large pane on the right is updated with the contents of the currently selected tree node (namely, a partially derived Hoare triple).

At this point, since only assertive code has been formed—not final VCs—the environment gives users the option of either automatically deriving simplified VCs for all assertive blocks[8] (by pressing the button) or some specifically selected subset thereof. The **Settings** tab merely allows users to tweak the (global) VC generation and proof strategy which includes (for example) bounding the maximum number of simplification rules, setting the prover's timeout, and other granular details about how proof rules should be applied during derivations.

Once the button is pressed, the derivation tree is updated in real-time, where each interior node represents the application of a particular rule. For example, the derivation tree that results from the assertive code in Sect. 4.4 is shown in Fig. 6.

Here, each node has an accompanying icon indicating the type of rule applied. In particular, the S denotes the application of a *statement* proof rule, while indicates

[8] Which is the default, implicit selection.

the application of a *sequent* based reduction rule. While there is a third category of rule—theory-based rules—we defer discussion of these until the end of this section. Lastly, when a derivation is completed, its tree is then "closed" (denoted via the icon) and the resulting VCs are added to the tabbed **VCs** window (illustrated in Fig. 1, bottom). In this tab, VCs are grouped under the construct that produced them, and are connected to locations in the actual source file via the badges appearing the editor's gutter. And since multiple VCs can arise on a single line of code, RESOLVE Studio allows users to click on the badge and select a particular VC; doing so automatically navigates them to the relevant entry in the **VCs** tab.

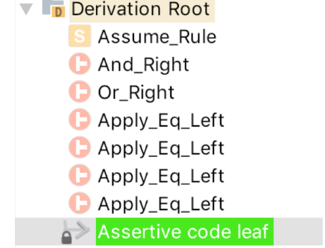

Fig. 6: A completed derivation tree.

Advanced Feature: *Interactive Derivation Tracing*. To help both researchers and advanced learners gain a better understanding of the workings of RESOLVE's proof system, the environment also permits users to interactively derive VCs (Fig. 7).

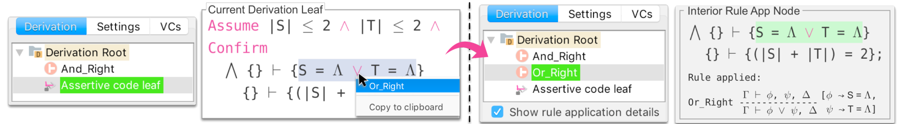

Fig. 7: Interactive VC derivation tracer.

Clicking on a (sub)term within the final confirm produces a contextual popup with applicable rules. Clicking a rule applies it, highlights the site of the application in green, and shows the rule itself along with its instantiation details (Fig. 7, right).

Advanced Feature: *Theory-Based Simplification Rules*. To further reduce the size and complexity of the terms that make up our VCs, we've also added support for user defined simplification rules. Such rules are declared within theories, and are preceded by the `Simp` keyword which "registers" the rule with the system.[9] A typical simplification rule, for example, could read: `Simp:` $\forall T : SStr, \forall s : Str(T), s \circ \Lambda = s$. Where the rule always terminates in an equality where the left side is more complicated than the right. More than just simplifying VCs, this feature can also be used to help spur development of new theories and new rules within them.

6 Experimental Evaluation

We've tested our revised VC generation technique on a modest battery of extension operations[10] for a variety of ADTs drawn from RESOLVE's current compo-

[9] Note: this is inspired by the behavior of Isabelle's "`simp`" command.

[10] Recursive implementations are labeled using the subscript: (*Recur*)

nent library including linked-lists, stacks, queues, and trees. Table 1 compares VCs generated with and without parsimonious schemes. For each scheme, column 1 contains the total number of VCs generated (#*VCs*), number of VCs with 5 or more antecedents ($\#VCs_{A\geq5}$), and number of VCs with 10 or more antecedents ($\#VCs_{A\geq10}$).

Table 1: Comparison of VC generation schemes

Component	(a) #*VCs*	(a) $\#VCs_{A\geq5}$	(a) $\#VCs_{A\geq10}$	(b) #*VCs*	(b) $\#VCs_{A\geq5}$	(b) $\#VCs_{A\geq10}$
`List Search`	35	35	35	35	16	0
`List Reverse`$_{(Recur)}$	8	8	8	8	0	0
`Queue Append`$_{(Recur)}$	8	8	8	8	0	0
`Stack Copying`	24	24	24	24	5	0

(a) Without parsimonious (b) With parsimonious

Notice that for the non-parsimonious scheme, all the VCs contain 10 or more antecedents. In contrast, the parsimonious scheme vastly reduced the number of antecedents for all VCs. A simple version of our ParsimoniousAssume rule has been patched into the existing compiler and has already received limited testing in the classroom, where students have already benefitted from the reduced, smaller VCs. Using the parsimonious scheme, the failed VC from Fig. 2 generates the following reduced sequent: $\{\texttt{S} = \langle \texttt{e}'''''\rangle \circ \texttt{S}', 1 \leq |\texttt{S}|\} \vdash \{\texttt{S}' = \texttt{S}\}$

7 Related Work

Since the principles of VC generation are well established in the literature—and play an integral role in most (if not all) modern auto-active program verifiers—the focus of this section is primarily on efforts that have made progress towards generating simpler, debuggable VCs through an IDE or otherwise.

Before getting into specific efforts, its worth noting that a direct comparison to the work detailed in this paper is somewhat hindered by the simple fact that most efforts "outsource" the process of generating VCs to Intermediate Verification Languages (IVLs) such as Boogie [13]. Such an approach naturally 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 [23]—or vice versa [7]. These mismatches in turn can complicate error reporting efforts, including VC feedback on failed verification attempts.

To help cope with this, many efforts have turned to IDEs and other tooling to help make verification and—specifically—the interpretation of verification results more usable and practical. For example, Dafny [14] integrates its toolchain into an IDE for Visual Studio which includes the Boogie Verification Debugger [13], which

translates Z3 generated counter examples into a form suitable for human consumption. AutoProof [22], which also uses Boogie and Z3 for the verification of Eiffel programs, employs so-called "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). In terms of IDE support, push-button verification along with a host of other functionality is provided through the Eiffel Verification Environment.

Why3 [6] is another popular autoactive tool that employs its own IVL (WhyML), to target a number of SMT solvers as well as some traditionally interactive systems (such as Coq) for automated proving. The language comes with a verification environment (called WhyIDE) that gives users the ability 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).

Lastly, the KeY framework [2], which targets 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 'taclets' (i.e., *tactics*) to the current proof state—manually guiding the system towards the goal.

8 Conclusions and Future Work

We have presented two F-IDEs designed to provide users with feedback suitable for reasoning about code correctness when verification fails. Additionally, we provided a technical overview of our revised proof system, including rules for deriving "parsimonious" VCs that omit irrelevant givens. We demonstrated our revised VC generator in the context of RESOLVE Studio, and showed several new approaches of using it to interact with RESOLVE's proof system.

Immediate directions for future work include updating the web-IDE to support the newer version of the compiler. Another direction involves adding new verification usability features to RESOLVE Studio (e.g., highlighting formulas that a particular VC originates from in the editor) as well as usage of the environment to assist in the development of larger component-based systems.

Acknowledgments

We thank the members of our research groups at Clemson and Ohio State who have contributed to this work. Particular thanks is due to Bill Ogden and Joan Krone for their valuable insights and feedback in formulating the proof rules appearing in this work. This research is funded in part by NSF grants CCF-1161916, DUE-1022941 and DUE-1611714.

References

1. Adcock, B.: Working Towards the Verified Software Process. PhD thesis, The Ohio State University (2010)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer, Cham (2016), http://dx.doi.org/10.1007/978-3-319-49812-6
3. Ameri, M., Furia, C.A.: Why just Boogie? In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. pp. 79–95. Springer International Publishing, Cham (2016)
4. Cook, C.T., Drachova-Strang, S., Sun, Y., Sitaraman, M., Carver, J.C., Hollingsworth, J.E.: 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 (2013), http://dx.doi.org/10.1109/CSEET.2013.6595254
5. Cook, C.T., Harton, H., Smith, H., Sitaraman, M.: Specification Engineering and Modular Verification Using a Web-integrated Verifying Compiler. In: ICSE. pp. 1379–1382. ACM (2012), http://dl.acm.org/citation.cfm?id=2337223.2337423
6. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer (2013), https://doi.org/10.1007/978-3-642-37036-6_8
7. Furia, C.A., Poskitt, C.M., Tschannen, J.: The AutoProof verifier: Usability by non-experts and on standard code. In: Dubois, C., Masci, P., Méry, D. (eds.) F-IDE 2015 (2015), https://doi.org/10.4204/EPTCS.187.4
8. Hampton, S.: Engineering Specifications and Mathematics for Verified Software. PhD Dissertation, Clemson University (2013)
9. Harton, H.: Mechanical and Modular Verification Condition Generation for Object-Based Software. PhD Dissertation, Clemson University (2011)
10. Heym, W.D., Sivilotti, P.A.G., Bucci, P., Sitaraman, M., Plis, K., Hollingsworth, J.E., Krone, J., Sridhar, N.: Integrating components, contracts, and reasoning in CS curricula with RESOLVE: experiences at multiple institutions. In: Washizaki, H., Mead, N. (eds.) CSEE&T. pp. 202–211. IEEE (2017), https://doi.org/10.1109/CSEET.2017.40
11. Kabbani, N.M., Welch, D., Priester, C., Schaub, S., Durkee, B., Sun, Y., Sitaraman, M.: Formal reasoning using an iterative approach with an integrated web IDE. In: F-IDE 2015. pp. 56–71 (2015), https://doi.org/10.4204/EPTCS.187.5
12. Kraemer, E.T., Sitaraman, M., Hollingsworth, J.E.: 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, Seeon Monastery, Bavaria, Germany, June 14-15, 2018. pp. 18–25 (2018), https://doi.org/10.1145/3209087.3209100
13. Le Goues, C., Leino, K.R.M., Moskal, M.: The boogie verification debugger (tool paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. pp. 407–414. LNCS, Springer, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-24690-6_28
14. Leino, K.R.M.: Developing verified programs with Dafny. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) ICSE 2013. pp. 1488–1490. IEEE (2013), https://doi.org/10.1109/ICSE.2013.6606754
15. Leino, K.R.M., Moskał, M.: Usable auto-active verification. In: Ball, T., Zuck, L., Shankar, N. (eds.) UV 2010 (2010), http://fm.csl.sri.com/UV10/

16. Mbwambo, N.M.J.: A Well-Designed, Tree-Based, Generic Map Component to Challenge the Process Towards Automated Verification. Master's Thesis, Clemson University (2017)
17. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78800-3_24
18. Priester, C., Sun, Y., Sitaraman, M.: Tool-Assisted Loop Invariant Development and Analysis. In: 29th IEEE International Conference on Software Engineering Education and Training, CSEET 2016, Dallas, TX, USA, April 5-6, 2016. pp. 66–70 (2016), https://doi.org/10.1109/CSEET.2016.28
19. Sitaraman, M., Adcock, B.M., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H.M., Harton, H.K., Heym, W.D., Kirschenbaum, J., Krone, J., Smith, H., Weide, B.W.: Building a push-button RESOLVE verifier: Progress and challenges. Formal Aspects of Computing 23(5), 607–626 (2011), https://doi.org/10.1007/s00165-010-0154-3
20. Sun, Y.: Towards Automated Verification of Object-Based Software with Reference Behavior. PhD Dissertation, Clemson University (2018)
21. Tagore, A., Zaccai, D., Weide, B.W.: Automatically proving thousands of verification conditions using an SMT solver: An empirical study. In: NFM 2012. pp. 195–209 (2012), https://doi.org/10.1007/978-3-642-28891-3_20
22. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: Autoproof: Auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Berlin, Heidelberg (2015), https://doi.org/10.1007/978-3-662-46681-0_53
23. Utting, M., Pearce, D.J., Groves, L.: Making Whiley Boogie! In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 69–84. Springer (2017), https://doi.org/10.1007/978-3-319-66845-1_5
24. Welch, D., Sitaraman, M.: Engineering and employing reusable software components for modular verification. In: Botterweck, G., Werner, C. (eds.) ICSR 2017. LNCS, vol. 10221, pp. 139–154. Springer (2017), https://doi.org/10.1007/978-3-319-56856-0_10