

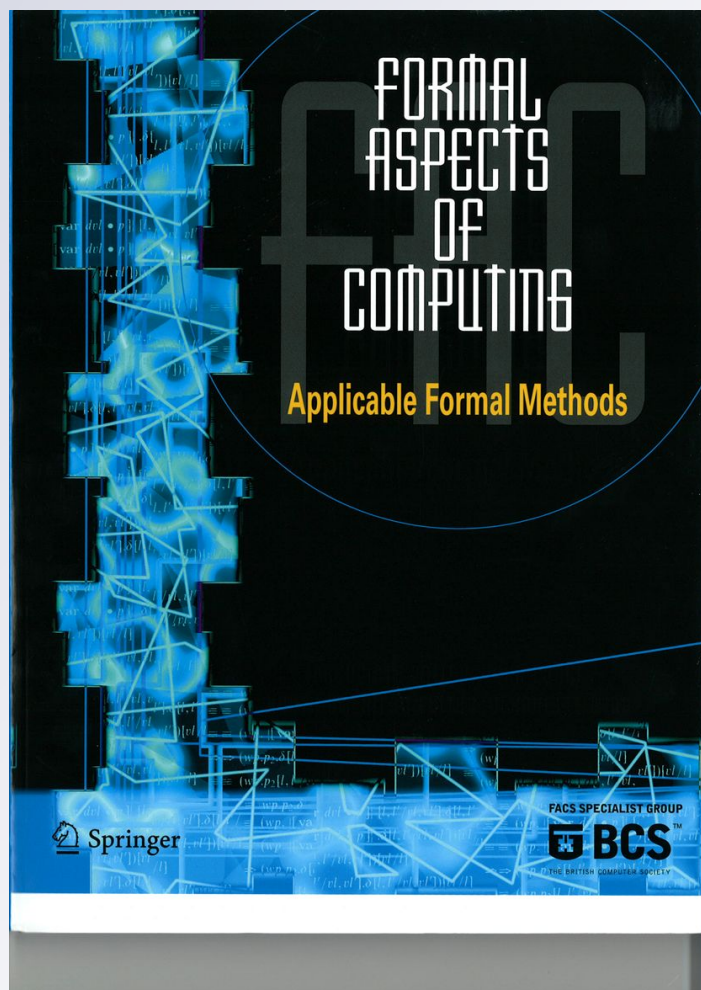
Building a push-button RESOLVE verifier: Progress and challenges

Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather Harton, Wayne Heym, et al.

Formal Aspects of Computing
Applicable Formal Methods

ISSN 0934-5043
Volume 23
Number 5

Form Asp Comp (2011) 23:607-626
DOI 10.1007/s00165-010-0154-3



Your article is protected by copyright and all rights are held exclusively by British Computer Society. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your work, please use the accepted author's version for posting to your own website or your institution's repository. You may further deposit the accepted author's version on a funder's repository at a funder's request, provided it is not made publicly available until 12 months after publication.

Building a push-button RESOLVE verifier: progress and challenges

Murali Sitaraman¹, Bruce Adcock², Jeremy Avigad³, Derek Bronish², Paolo Bucci², David Frazier¹, Harvey M. Friedman^{4,2}, Heather Harton¹, Wayne Heym², Jason Kirschenbaum², Joan Krone⁵, Hampton Smith¹ and Bruce W. Weide²

¹ School of Computing, Clemson University, Clemson, SC 29634, USA

² Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA

³ Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA 15213, USA

⁴ Department of Mathematics, The Ohio State University, Columbus, OH 43210, USA

⁵ Department of Mathematics and Computer Science, Denison University, Granville, OH 43023, USA

Abstract. A central objective of the verifying compiler grand challenge is to develop a push-button verifier that generates proofs of correctness in a syntax-driven fashion similar to the way an ordinary compiler generates machine code. The software developer's role is then to provide suitable specifications and annotated code, but otherwise to have no direct involvement in the verification step. However, the general mathematical developments and results upon which software correctness is based may be established through a separate formal proof process in which proofs might be mechanically checked, but not necessarily automatically generated. While many ideas that could conceivably form the basis for software verification have been known “in principle” for decades, and several tools to support an aspect of verification have been devised, practical fully automated verification of full software behavior remains a grand challenge. This paper explains how RESOLVE takes a step towards addressing this challenge by integrating foundational and practical elements of software engineering, programming languages, and mathematical logic into a coherent framework. Current versions of the RESOLVE verifier generate verification conditions (VCs) for the correctness of component-based software in a modular fashion— one component at a time. The VCs are currently verified using automated capabilities of the Isabelle proof assistant, the SMT solver Z3, a minimalist rewrite prover, and some specialized decision procedures. Initial experiments with the tools and further analytic considerations show both the progress that has been made and the challenges that remain.

Keywords: Languages, Software engineering, Theorem proving, Tools, Verification

1. Introduction

Automation is a fundamental challenge in developing a verifier and eventually realizing the grand challenge of building verified software [Hoa03]. In pursuit of automation, some well-known efforts have focused on verifying particular sets of properties of software. Alternatively, to achieve full behavioral verification, others have restricted the domain of verification to types that are typically “built in” to programming languages, such as integers and

arrays. In the interest of achieving automation, still others have proposed systems that, by design, compromise on soundness and/or completeness.

The objective of our verifying compiler work is to facilitate specification and automated verification of the full behavior of software. It is designed to verify software written in RESOLVE, an integrated specification and programming language with clean semantics that has been especially designed to facilitate modular (hence scalable) verification. The specification language is a mathematical higher-order language and is extensible with new mathematical theories, definitions, and related developments including general-purpose theorems and lemmas. The implementation language is imperative and object-based. The proof system for verification of implementation correctness is both sound and relatively complete. The proof system is also modular: it is possible to verify component-based software, one component at a time, relying on only the specifications of the components reused by the component that is being verified. A key design objective is that beyond providing specifications and implementations with suitable annotations (e.g., loop invariants and progress metrics), software developers should not be involved in proof activities. Proofs of non-trivial theorems used by automated provers, however, might themselves require assistance from mathematical specialists, but that is done “off line” to populate a library of mathematical results to be used in automated verification of VCs arising from programs. This is in contrast to some recent related work (e.g., Jahob [ZKR09]) that expects programmers to suggest explicit proof steps as a part of the program development process.

This paper explains the progress we have made in developing a verifier and related tools, and outlines several challenges that remain. Its focus is on tools, not on the underlying theory (for which relevant background papers are cited); the paper is intended to be somewhat self-contained and hence accessible to a reader who is not familiar with these details. Section 2 of the paper presents the objectives of the verifying compiler through illustrative screenshots of a tool demonstration system for a simple benchmark. It also describes an internal architecture of the verifier that serves to integrate the subordinate tools detailed in subsequent sections. Section 3 explains VC generation using two versions that we have developed to explore issues in generation of VCs and their proofs; for variety, the section employs two distinct examples, both of which illustrate the modular nature of VC generation. The focus of Sect. 4 is on proving these VCs automatically in two different ways. Since the VCs stem from general software specifications and implementations and may involve multiple mathematical theories and higher-order logic formulae, sometimes general-purpose provers are necessary to discharge them. At other times, special-purpose decision procedures may be suitable. So the paper discusses experience with employing Isabelle [NPW02] as an automated prover (as opposed to an interactive proof assistant) and a simplifier and specialized decision procedure we have developed; the toolset also incorporates the SMT solver Z3 [deM08] and a minimalist rewriting prover as alternatives, but this paper does not discuss experiences with them. Section 5 summarizes tool-related issues in mechanically checking proofs of non-trivial theorems invoked in completing automated proofs as in Sect. 4. Section 6 contains a discussion of related automated software verification efforts along with our conclusions.

2. Verifying compiler overview

This section presents an overview of the verifying compiler and a description of the toolset underlying its overall internal organization.

2.1. External vision of push-button verification

To illustrate the goals of the verifier quickly, we consider automated verification benchmark #1 described in [WSH08]. This benchmark requires verification of (total) correctness of an operation to add two integers by repeated incrementing and decrementing. The implementation may be annotated as necessary with invariants for loops and/or progress metrics to show termination.

To underscore the meaning of “push-button” verification, three screenshots of our tool demonstration system illustrate the working of one version of the RESOLVE tools for this simple example, explained in detail in [WSH08]. Figure 1 contains the specification of *Integer* addition within a larger context of the *Integer_Template* concept (or contract) that is not shown. Specification of the *Integer_Template* concept, or any other one selected, may be viewed by clicking the “View” button. Figure 2 shows an iterative realization (or implementation) of the specification. Figure 3 shows that the code has been proved correct by the push-button verifier on clicking the “Verify with RESOLVE” button. (The “Compile” button generates executable code.)

The example illustrates that given supporting specifications and relevant mathematics, the task of the programmer is limited to supplying annotated code for the implementation under consideration for verification. All detailed proof activity is left to the verifier.

The verification process is *modular*, so verification of one implementation never involves any other implementations, merely other specifications. In other words, every implementation is verified correct exactly once in its lifetime. For example, to complete benchmark #1, part 2 [WSH08] where the goal is to verify *Integer* multiplication implemented using repeated addition, only the specification of the *Integer* addition operation (in Fig. 1) is necessary. The implementations of addition are not relevant and are not used in the verification process. The RESOLVE verification system uses the same underlying specification-based proof system for verification of all operations (including those on typically built-in types like *Integers* or pointers).

Concept	Enhancement	Enhancement Realization	Concept Realization
Char_Str Character Integer List Location_Linking	Add_Capability Adding_Capability Alt_Int_Mult_Capability BWW_Int_Mult_Capability Example_Capability	Iterative_Add_to_Realiz Recursive_Add_to_Realiz	
View Create	View Create	View Create	View Create


```

Enhancement Adding_Capability for Integer_Template;
uses Integer_Theory, Std_Integer_Fac;

Operation Add_to(updates i:Integer; evaluates j:Integer);
requires min_int <= i + j and i + j <= max_int and j >= 0;
ensures i = #i + j;
    
```

Generate Code

Generate VCs

Verify with Isabelle

Verify with RESOLVE

Create Facility

Generate Facility

end Adding_Capability;

View: Input Results Java Code VCs Clear Textbox

Fig. 1. Example specification: Integer addition

Verification Demo Math Units

Concept	Enhancement	Concept Realization	Enhancement Realization
Prioritizer_Template.co Location_Linking_Template.co List_Template.co Integer_Template.co Char_Str_Template.co Basic_Map_Template.co	Int_Mult_Capability.en Adding_Capability.en		Recursive_Add_to_Realiz.rb Iterative_Add_to_Realiz.rb
View	View	View Compile	View

Code for Iterative_Add_to_Realiz.rb enhancement realization:

```

Realization Iterative_Add_to_Realiz for Adding_Capability of Integer_Template;
uses Std_Boolean_Fac;

Procedure Add_to(updates i:Integer; evaluates j:Integer);
While (not Is_Zero(j))
  changing i, j;
  maintaining (i + j = #i + #j) and j >= 0;
  decreasing j;
do
  Increment (i);
  Decrement (j);
end;
end Add_to;
end Iterative_Add_to_Realiz;
    
```

Generate Code

Generate Isabelle VCs

Generate Readable VCs

Generate and Verify

Fig. 2. An iterative implementation of addition

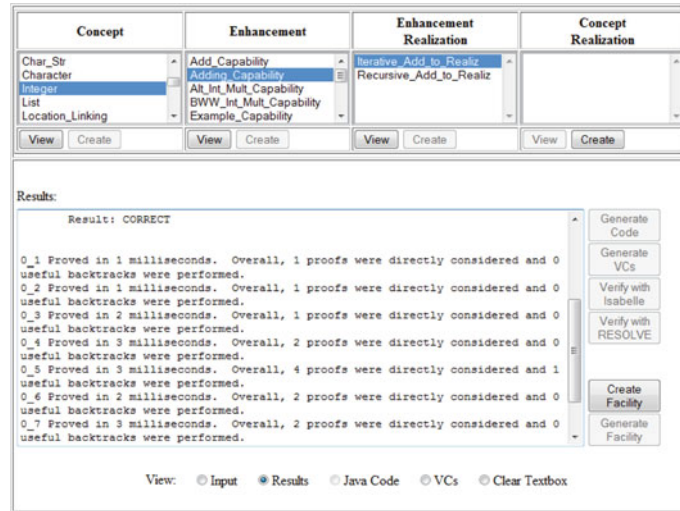


Fig. 3. Net result of push-button verification

2.2. Toolset organization and its usage scenarios

Figure 4 is an illustration of the internal organization of the verifier. In the figure, we depict tangible artifacts in the verified software process by ovals, and tools that facilitate the process by boxes. Two human roles are shown as shaded boxes; it should be noted that both roles might be filled by the same person, or each by teams of people. One role is to write “assertive code” in accordance with specifications of desired software behavior. The other role is to develop mathematical theories, definitions and results relevant to the specifications, along with formal proofs of such purely mathematical results. Proofs may be generated automatically or written manually, but must be validated by a mechanical proof checker. In contrast, proofs of correctness of the code are to be completed automatically with no human intervention.

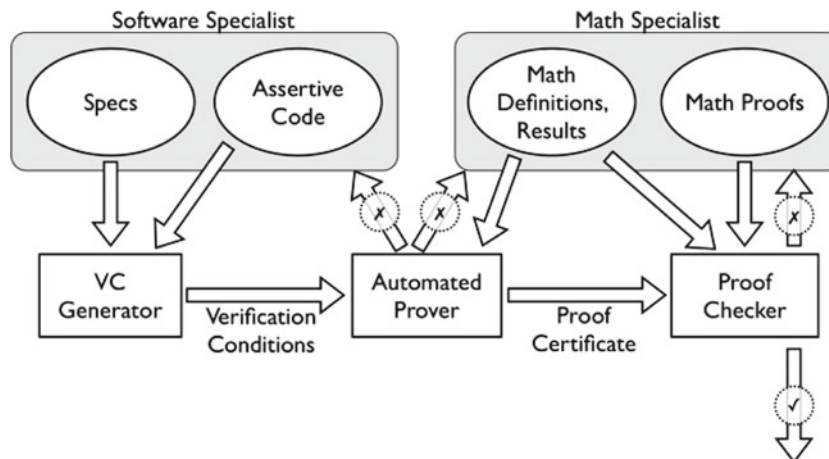


Fig. 4. Verification toolset architecture

A person in the first role—we call her the *software specialist*—writes not only the kind of (executable) code routinely written today, but also embedded formal annotations such as invariants and progress metrics for loops, representation invariants and abstraction relations for new data representations, etc. She debugs and repairs code and/or assertions when an implementation cannot be automatically verified. In addition to understanding specifications of off-the-shelf components used in her code, the software specialist also writes specifications for local operations she needs in client code, and designs, specifies, and implements more general-purpose components for shared component libraries.

When there is difficulty formalizing intended software behavior, a software specialist may consult a person in the second role—the *math specialist*. The math specialist may engage in new theory development or add mathematical definitions and results to existing theories to simplify specifications and code annotations, and may establish various lemmas and theorems about fragments of mathematics to ease the task of the automated prover. For example, when writing code involving text, stacks, queues, lists, and related components (which in RESOLVE are modeled using mathematical strings), it is sometimes necessary to reason about the reversal of a string, as can be seen in Sect. 3.1. The math specialist might define a mathematical function *reverse* from strings to strings, and then write reusable lemmas that might be expected to arise in the verification of programs involving types that are modeled as strings. As an example, consider the lemma $\forall \alpha, \beta(\text{reverse}(\alpha \cdot \beta) = \text{reverse}(\beta) \cdot \text{reverse}(\alpha))$, where \cdot denotes concatenation of strings. Typically, such results would be already included in the mathematical developments and would be reused in verifying a number of components.

A key idea in the verified software paradigm is that the proof of supporting mathematical facts need not be carried out automatically. The math specialist can construct the proof off-line, perhaps with the help of an automated proof assistant such as Isabelle. Because the lemma being proved is assumed throughout the rest of the software verification process, the soundness of the system depends on the validity of this proof; so submitting the proof to an automated proof checker may be appropriate as explained in Sect. 5. The proved lemma then becomes one of many mathematical results included with the definitions used in specifications and becomes available to the automated prover during software verification, where the proof of the lemma is no longer involved.

Some lemmas might require substantial mathematical insight to identify and prove, and cannot be established automatically, in general. So the goal is push-button verification of most programs *given* a sufficiently rich set of theories and developments in the form of lemmas in the automated prover's math library. In the steady state, it is expected that the role of the math specialist becomes not primarily to add more and more lemmas for established theories, but rather to develop additional theories and results that might be helpful in new software development tasks for new application domains.

It is likely the verified software paradigm will involve far more software specialists than math specialists, if only because the work of the latter is inherently far more reusable. Considerable experience to date with this paradigm suggests there will be far fewer definitions and results in the library of mathematical units than there are programs in the software component library: a testament to the power of mathematics in building useful abstractions.

The next three subsections concentrate on the major components of the verifying compiler architecture. Verification condition generation is the topic of Sect. 3, with the automated prover described in Sect. 4 and the proof checker in Sect. 5.

3. Automated generation of verification conditions

A key element of the verifying compiler is its VC (verification condition) generator, which takes relevant specifications and an implementation and produces a set of mathematical assertions that, if proven, guarantee the correctness of the implementation. The proof rules that underlie VC generation and claims of soundness and completeness of those rules are discussed elsewhere [EHO94, HSK08, Hey95, Kro88, SWO97]. At present we are building and experimenting with two versions of the VC generator, one that begins with the goals of the implementation and proceeds to derive VCs one at a time by sweeping backward through the code, and another that leads to a table of known assertions and assertions to be proved and that is effectively neutral with respect to the order in which VCs are generated. The nature of these tools is such that they will allow us to study the impact of inherent differences in the structure of VCs and the impact of different (but logically equivalent) styles of specification writing from an automation perspective. The VC generators are conceptually plug-compatible, meaning one can be used in the place of the other within the toolset architecture. Both share the structural and semantic framework common to RESOLVE [SiW94] (though they differ slightly in terms of the language syntax they accept), so the comments on specification, design, and implementation mentioned in context in Sect. 4.1 (e.g., modularity issues) and Sect. 4.2 (e.g., swapping) are applicable to both approaches. The significant differences are in the proof rules underlying VC generation and the structure of the resulting VCs.

3.1. Goal-directed VC generation

The formal proof rules underlying the goal-directed approach are discussed in [HSK08]. To illustrate the VCs generated by this approach, we have chosen benchmark #3 from [WSH08]. The goal of this benchmark is to specify and implement a piece of code involving a user-defined abstract data type (ADT) *Queue* instead of code that uses structures such as integers or arrays which are usually built into programming languages. Specifically, this benchmark requires verification of the (iterative or recursive) code for a *Queue* reversal operation. In this section, we explain how our VC generator addresses this challenge in a modular fashion, using only the specification of *Queue*, without regard to how a *Queue* is represented and realized. This means that, among other advantages, the proof does not deal with nodes, pointers, and other possibly quirky details of *Queue* representations [KSW06, WEH94]. Those internal details are examined separately when the ADT data representations themselves are verified. This approach makes reasoning about pointers a relatively independent verification problem as opposed to an intertwined problem that must be addressed in specification and verification of all object-based software [Kul04].

Figure 5 contains the specification of a *Queue* flipping operation. This specification enhances, or extends, the specification of *Queue_Template* (given in an Appendix), where *Queues* are conceptualized as mathematical strings of entries, as formalized in *String_Theory*. This specification approach is similar in spirit to other model-based specification languages, such as those summarized in [Win90]. Details of the specification notation may be found in [KSY08]. In general, concepts, such as *Queue_Template* may import one or more (highly reusable) mathematical theories as necessary. The *Queue* type defined in *Queue_Template* is bounded, so the component is parameterized both by the type of entries in the *Queue* and a maximum length. All *Queues* are conceptually constrained to be within this length. *Queue_Template* provides a set of basic, or primary, operations to manipulate *Queue* objects, such as *Enqueue*, *Dequeue*, and *Length*. Its design avoids the need to copy either (shallow) references or (deep) representations [HaW91, KSY08]. In the ensures clause of *Flip*, $\#Q$ denotes the incoming value of Q and *reverse* is a mathematical string operator.

Figure 6 contains a recursive implementation of the *Flip* operation. A recursive procedure must be annotated with a **decreasing** clause that is type-checked to be an ordinal. This progress metric must be shown to decrease with each recursive call in order to show termination. Loops in iterative implementations must be annotated additionally with an invariant through a **maintaining** clause as illustrated in the next subsection.

The goal-directed VC generator can produce outputs in multiple forms, including an Isabelle-friendly version (not shown). Four VCs that arise for the flipping implementation are shown in Fig. 7. Each VC is an implication and it includes the assumptions that may be used to prove the obligation following the implication. The types of variables are shown at the top, where *Str* is the name given for strings in *String_Theory*. In the VCs, “o” denotes string concatenation, $\langle E \rangle$ denotes a string containing one entry E , $|\alpha|$ denotes the length of a string α .

It is important to note that the VCs are purely mathematical assertions. Any program variable (e.g., a *Queue* Q) that appears in a VC stands for its abstract mathematical value (e.g., a mathematical string).

In the VCs, variable names with a prefix of “?” (e.g., $?E$, $?Q$) along with P_Val are generated in the verification process, and they correspond to the values of variables at different states. The assumptions in the VCs arise from constraints on objects (e.g., the constraints $|Q| \leq Max_Length$) and knowledge about the states based on the specifications of called operations or branch conditions.

The goal-directed approach attempts to minimize both the number of VCs that are generated and the number of new variables that are introduced. The generator also performs other simplifications to ease the task of the back-end prover. Eliminating VCs reduces unnecessary effort for the provers and naturally reduces the time for automated proving (though there are no end-to-end performance comparisons of different integrated systems). Variable reduction has a direct effect on the number of proof steps necessary, which is often the determining factor for whether a general-purpose prover can discharge a proof automatically. To prove the correctness of the code the following VCs are generated and proved. Figure 7 shows only four VCs because the fifth one has been simplified, proved, and eliminated by the VC generator itself.

- Termination (VC 1)
- Requires clause of called operation *Dequeue* (simplified to true by VC generator)
- Requires clause of called operation *Enqueue* (VC 2)
- Ensures clause of *Flip* when if condition is true (VC 3)
- Ensures clause of *Flip* when if condition is false (VC 4)

```

Enhancement Flipping_Capability for Queue_Template;
  Operation Flip( updates Q: Queue);
    ensures Q = reverse(#Q);
  end Flipping_Capability;

```

Fig. 5. Specification of a Queue flipping capability

```

Realization Recursive_Flipping_Realiz for
  Flipping_Capability of Queue_Template;
  uses Std_Boolean_Fac;
  Procedure Flip( updates Q: Queue);
    decreasing |Q|;

    Var E: Entry;
    If (Length(Q) /= 0) then
      Dequeue(E,Q);
      Flip(Q);
      Enqueue(E,Q);
    end;
  end Flip;
end Recursive_Flipping_Realiz;

```

Fig. 6. A recursive implementation

```

Free Variables: Max_Length:Z, min_int:Z, max_int:Z,
  Q:String_Theory.Str(Entry), P_val:N, ?E:Entry,
  ?Q:String_Theory.Str(Entry), E:Entry

(((min_int <= 0) and (0 < max_int)) and
(Max_Length > 0)) and
((|Q| <= Max_Length) and (P_val = |Q| and
(|Q| /= 0 and Q = (<?E> o ?Q))))
=====
(|?Q| < |Q|)

(((min_int <= 0) and (0 < max_int)) and
(Max_Length > 0)) and
((|Q| <= Max_Length) and (P_val = |Q| and
(|Q| /= 0 and Q = (<?E> o ?Q))))
=====
(|reverse(?Q)| < Max_Length)

(((min_int <= 0) and (0 < max_int)) and
(Max_Length > 0)) and
((|Q| <= Max_Length) and (P_val = |Q| and
(|Q| /= 0 and Q = (<?E> o ?Q))))
=====
(reverse(?Q) o <?E>) = reverse(Q)

(((min_int <= 0) and (0 < max_int)) and
(Max_Length > 0)) and ((|Q| <= Max_Length) and
(P_val = |Q| and |Q| = 0))
=====
Q = reverse(Q)

```

Fig. 7. VCs generated for recursive flipping realization

The VCs in the figure took a fraction of a second to generate and 0.7 s for Isabelle to prove. In addition to proving similar conditions, for code involving loops, VCs to prove the validity of the (programmer-supplied) loop invariant are also generated [HSK08]. In order to be able to illustrate the whole verification process through alternative methods, only simple examples have been discussed in this paper. It is important to note that, in general, the generated VCs may involve quantifiers and higher-order logic (because the corresponding software specifications involve them). One ongoing research direction is to determine to what extent (and to what benefit) quantifiers can be minimized in specifications through suitable mathematical definitions and theory developments. Kirschenbaum et al. [KHS08], for example, note the difficulty in automating the proof of an invariant because of a universal instantiation problem and explain the use of a higher-order predicate along with development of suitable theorems for automating VCs that arise in a generic sorting case study.

3.2. Tabular VC generation

In this second approach, VCs are generated into a locally defined XML format using a tool we have developed that implements the proof rules described formally in [Hey95] and informally in [SAK00]. The XML serves as an intermediate form that is translated into Unicode for human consumption, into input for Isabelle for automated proof, and into input for our own prover *SplitDecision* (see below) that incorporates special-purpose decision procedures for fragments of mathematical theories that are widely used in RESOLVE, e.g., strings, finite sets, and integers.

As outlined in [WSH08], the tabular method generates one VC for each state in the program where the next statement involves either establishing a pre-condition for the next operation invoked, or a loop invariant or progress metric, or the post-condition for the operation being verified. The VC generator works in two phases. Rote VC generation introduces in the first phase a large number of mathematical variables: one for each program variable in each program state (between consecutive statements). For instance, x_i stands for the value of program variable x in state i ; in a program with n variables and m statements, the VCs constructed this way therefore collectively contain a total of approximately nm distinct mathematical variables at the end of phase one.

The simplification phase of the VC generator then applies a few theory-independent logical restructuring rules to each VC. The most obvious and useful simplification is simply substitution of equals for equals. Many of the nm variables are removed by this step. For instance, the hypotheses in the original VCs often contain clauses such as “ $x_{10} = x_9$ ”, with a substitution removing one of these two variables (and this clause). Any effective back-end automated prover would certainly be able to apply such substitutions to achieve the same effect, but the VC generator does this before handing the VCs to an automated prover so the human-readable output of the VC generator is more concise. Readability is an issue because it may be necessary for software or math specialists to study unproved VCs and tweak their specifications or other assertions (e.g., an invariant). Some other work done in the second phase is intended to help the back-end prover by making structural changes that are expected to be fruitful because of the nature of the proof rules. An example is the use of automatic case analysis. For the VCs generated for components in our library, case analysis can only rarely be performed by a back-end automated prover: the requirement for human advice about appropriate cases seems to be the norm. The VC generator therefore strives to make it unnecessary. The simplification phase divides each VC into appropriate cases based on the path structure of the code from which the VCs arise so the automated prover need not be asked to infer information that was previously known but effectively obfuscated by the initial phase of VC generation.

The net result is that, in the tabular approach, there are normally many VCs even for a relatively short piece of code. On the other hand, each VC tends to be relatively small and apparently “prover-friendly”. Experience to date [KAB09] shows that the vast majority of such VCs can be discharged without difficulty by the automated provers we are using—based on fairly limited mathematical theory libraries (in the case of Isabelle) and limited built-in mathematical knowledge about the theories involved in the specifications (in the case of *SplitDecision*).

To illustrate the tabular reasoning method and at the same time to highlight other aspects of verification—verification of an iterative piece of code that uses objects modeled as mathematical sets instead of strings—this subsection considers an enhancement that allows a client program to form unions and intersections of two sets of items given the *SetTemplate* contract (see Appendix B), rather than using the same example from the previous section. VCs for the Queue *Flip* example using the tabular reasoning method are available at the project website.

The specification of a set union and intersection enhancement is given in Fig. 8 and an implementation is shown in Fig. 9. An important feature of this code is the use of the swap statement (“:=” operator) to swap, or exchange, the values of two variables of the same type. Swapping is a staple of RESOLVE [HaW91] and it replaces the typical variable-to-variable assignment in other languages. Among other things, swapping admits efficient implementation for every type without introducing the need for reference semantics and the associated problems of aliasing that otherwise would significantly complicate modular verification [WeH01]. While there are automated systems today for reasoning about properties of programs with reference behavior as summarized in the related work section, fundamental complications that references and aliasing pose for modular verification of full software behavior remain open issues [Kul04, LLM08, WeH01].

In the code in Fig. 9, swapping is used for two purposes. First, the programmer makes sure that the iteration is over the smaller of the two sets provided as arguments (for efficiency). Then the local *Set* variable *tmp* (initialized to the empty set upon declaration) is used to hold those elements of the incoming t that are also in the incoming s , i.e., those that are in their intersection. At the end of the loop, since the value of *tmp* is the intersection of the original arguments and this value is supposed to be returned in t , these two variables are swapped. This iteration idiom is common in RESOLVE programs that use collection ADTs [WeH91] [WEH94].

```

contract SetUniteAndIntersect enhances SetTemplate
  procedure UniteAndIntersect (updates s: Set,
                               updates t: Set)
    ensures
      s = #s union #t and t = #s intersection #t
  end SetUniteAndIntersect

```

Fig. 8. An operation to form union and intersection of sets

```

realization Iterative2 implements SetUniteAndIntersect

  procedure UniteAndIntersect (updates s: Set,
                               updates t: Set)
    variable ss, ts: Integer
    variable tmp: Set
    ss := Size (s)
    ts := Size (t)
    if IsGreater (ts, ss) then
      s := t
    end if
    loop
      maintains s union t = #s union #t and
        (s intersection t) union tmp =
          (#s intersection #t) union #tmp and
          t intersection tmp = {}
      decreases |t|
    while not IsEmpty (t) do
      variable x: Item
      RemoveAny (t, x)
      if not IsMember (s, x) then
        Add (s, x)
      else
        Add (tmp, x)
      end if
    end loop
    t := tmp
  end UniteAndIntersect

end Iterative2

```

Fig. 9. An implementation of the contract in Fig. 8

Our VC generator produces 38 VCs for this code. They range in apparent complexity through the spectrum of samples shown in Fig. 10, which are copied directly from the human-readable output produced by the tabular VC generator. A few VCs that look like #15 are evidently the most difficult of these to prove, but even in this case the conclusion follows from only the given hypotheses 2 and 7 and a couple general-purpose properties of finite sets. The conclusion of VC #5 follows from the property that the empty set is a right identity for intersection; there are a number of other VCs like this one. VC #10 is typical of a few more VCs: its conclusion is simply one of the hypotheses. The conclusion of VC #35, like a few other VCs of the same nature, follows from the given hypothesis 1 and a couple other general-purpose properties of finite sets.

Both Isabelle and *SplitDecision* prove all 38 VCs from this code automatically. Isabelle does it by expanding union, intersection, and difference into their definitions based on set membership and then relying on its considerable power to simplify Boolean expressions. *SplitDecision* uses a similar approach encoded as structural rewrites (in this case, involving expressions with finite-set operators such as union and cardinality) that simplify each VC as far as possible—and all these VCs are simplified to “true”.

Our web interface to the toolset, the VC generator, and both back-end automated provers currently run on a typical PC running Linux and Apache. The response time on clicking the “Verify” button for this code is quite reasonable: VC generation takes a small fraction of a second, and Isabelle proves all 38 VCs in about 3.7 s while *SplitDecision* proves them all in about 2.0 s of wall-clock time.

To be sure, not all VCs involving strings, finite sets, and integers are automatically discharged for all the examples we have tried. We currently let Isabelle time-out at 1 s on each VC, and occasionally it does so without having completed a proof or refutation. Either Isabelle or *SplitDecision* might also simplify a VC to something else with no further progress in sight. Considerable empirical evidence will need to be obtained in order to generalize comfortably about why some VCs turn out to be difficult; initial work in this direction is reported in [KAB09]. Some known challenges include VCs involving permutations of strings (which arise in the specifications and

Verification Condition #5 (state index: 7, loop invariant)

Prove

$$s_0 \cap \emptyset = \emptyset$$

Given

1. $|t_0| > |s_0|$

Verification Condition #15 (state index: 15, loop invariant)

Prove

$$(s_8 \cup \{x_{10}\}) \cup (t_8 \setminus \{x_{10}\}) = t_0 \cup s_0$$

Given

-
1. $t_8 \neq \emptyset$
 2. $s_8 \cup t_8 = t_0 \cup s_0$
 3. $(s_8 \cap t_8) \cup \text{tmp}_8 = (t_0 \cap s_0) \cup \emptyset$
 4. $t_8 \cap \text{tmp}_8 = \emptyset$
 5. $|t_8| \geq 0$
 6. $\text{is_initial}(x_9)$
 7. $x_{10} \in t_8$
 8. $\text{is_initial}(x_{12})$
 9. $|t_0| > |s_0|$
 10. $x_{10} \notin s_8$

Verification Condition #10 (state index: 9, requires clause)

Prove

$$t_8 \neq \emptyset$$

Given

-
1. $t_8 \neq \emptyset$
 2. $s_8 \cup t_8 = s_0 \cup t_0$
 3. $(s_8 \cap t_8) \cup \text{tmp}_8 = (s_0 \cap t_0) \cup \emptyset$
 4. $t_8 \cap \text{tmp}_8 = \emptyset$
 5. $|t_8| \geq 0$
 6. $\text{is_initial}(x_9)$
 7. $|t_0| \leq |s_0|$

Verification Condition #35 (state index: 17, ensures clause)

Prove

$$s_{16} = s_0 \cup t_0$$

Given

-
1. $s_{16} \cup \emptyset = t_0 \cup s_0$
 2. $(s_{16} \cap \emptyset) \cup \text{tmp}_{16} = (t_0 \cap s_0) \cup \emptyset$
 3. $\emptyset \cap \text{tmp}_{16} = \emptyset$
 4. $|t_0| > |s_0|$

Fig. 10. Some VCs for unite and intersect set implementation

annotations of sorting algorithms), those using a function that maps the entries of a string to the finite set containing those entries, and situations where the proof of a VC involves an algebraic property whose importance was not apparent at the time of theory development (e.g., that concatenation of strings is commutative in arguments to the *is_permutation* predicate). We are confident that these sorts of VCs can be handled effectively with additional library development for Isabelle and additional rules for *SplitDecision*.

Neither the goal-directed approach nor the tabular approach currently generates VCs for all of the features the RESOLVE language. However, no known technical barriers remain for VC generation. For example, one important step is to add VC generation for components that include new data type representations (e.g., a realization of *SetTemplate*). The idea is that the realization is coded with annotations similar to those in Hoare's framework for

proof of data representation [Hoa72, SWO97] (i.e., with a representation invariant and an abstraction relation for each new type).

There are numerous issues that can make it difficult to prove the generated VCs, and they need to be addressed. They concern mathematical theories, their use in specifications, and programming paradigms. One challenge for math specialists is suitable theory development (in terms of both definitions and theorems presenting algebraic properties about them) that is necessary to minimize the usage of both existential and universal quantifiers. For example, though auxiliary variables can be used in RESOLVE to automate verification in the presence of existential quantifiers (e.g., the *Swap_First_Entry* operation in the *Queue_Template* specification in Appendix A) following the lines of [Kin70], suitable string theory development (e.g., with the use of a definition that returns the part of a given string between specified indices and supporting theorems) can dramatically reduce the need for quantifiers [KHS08]. The challenge for specifiers is devising specifications that facilitate ease of mechanization, yet are easy for people to understand. Just as programmers are taught to write “good” code, software developers should learn to write “good” specifications. The challenge for language designers is in taking into account ease of verifiability considerations in addition to efficient performance. Finally the tools should provide meaningful feedback to programmers when there are errors, and assist them in the process of eventually developing verified software.

4. Automated proving of VCs

This section summarizes two complementary approaches for proving generated verification conditions. The first section explains how we import our theory developments to Isabelle and illustrates what can and cannot be proved using only “auto” mode. The second subsection discusses *SplitDecision*, a custom-built tool for proving RESOLVE VCs.

4.1. Automated proving using Isabelle

A natural question about proving VCs using a particular theorem proving tool such as Isabelle is how the VCs should be expressed. We take the approach of creating VCs that are consistent with RESOLVE mathematical theories instead of generating VCs directly in the language of the prover (Isabelle) that use prover-specific theories provided by other Isabelle users or its developers. This decision allows us to remain decoupled from Isabelle: we import RESOLVE mathematical theories for use in the VC proofs by encoding them for Isabelle. There are many other choices for an off-the-shelf automated prover, and while Isabelle has many merits, it is not necessarily the ultimate back-end automated prover for RESOLVE VCs. For the examples used in this paper, we have created both a RESOLVE finite set theory and a RESOLVE string theory within Isabelle. Both theories use types defined in Isabelle as witnesses to the consistency of the theories. However, once these off-line existence proof obligations are dispatched, only lemmas defined in the RESOLVE theories are used by Isabelle to prove VCs.

This approach both reduces the number of errors in the importing process (by providing witnesses to ensure consistency) and ensures that the theorem proving tool “understands” the mathematics in the VCs generated. We also note that the particular method of VC generation (goal directed or tabular) is unimportant to the proof of the VC; both sources of VCs result in assertions that can be processed and proved by Isabelle.

The first examples presented in this paper have specifications in terms of mathematical string theory. These mathematical strings are used as a mathematical model of the behavior of the *Queue* ADT, which is similar to what can be done using sequences in JML [LBR06]. However, specifications and code are two distinct entities in RESOLVE; specifications are statements in mathematics and are not intended to be executable.

Informally, strings over a given type *obj* are intended to have a model that is exactly the elements of *obj**, where * is the Kleene star. However, a theory such as the one in Fig. 11 is self-contained and might have other models; that is, a theory is defined not by exhibiting a particular model, but rather by its axioms. Of course, it is important to know that such a model exists and that we are not describing an inconsistent theory.

A few functions are defined inductively for strings: concatenation, length, and reverse. Figure 11 is a screenshot showing how we state some simple lemmas without proofs. The proofs of these lemmas follow easily (but not automatically) from the axioms and definitions. Once proved with human assistance, though, these lemmas can be used in proofs of VCs, exactly like the axioms and definitions. Ongoing experiments include specification and automated proving with alternative string definitions (e.g., one that returns the part of a given string between specified indices) and supporting theorems.

String Type Signature

$$\begin{aligned} \text{string} &\stackrel{\text{def}}{=} \text{string}(\text{obj}) \\ \Lambda &: \text{string} \\ \text{ext} &: \text{string} \times \text{obj} \longrightarrow \text{string} \end{aligned}$$
String Axioms

1. $\text{ext}(s, x) \neq \Lambda$
2. $\text{ext}(s_1, x_1) = \text{ext}(s_2, x_2) \Rightarrow s_1 = s_2 \wedge x_1 = x_2$
3. $\forall S \in \mathcal{P}(\text{string}) : (\Lambda \in S \wedge \forall x, s : (s \in S \Rightarrow \text{ext}(s, x) \in S)) \Rightarrow S = \text{string}$

Function Definitions

1. $\langle _ \rangle : \text{obj} \longrightarrow \text{string} \stackrel{\text{def}}{=} \langle x \rangle = \text{ext}(\Lambda, x)$
2. $|_| : \text{string} \longrightarrow \mathbb{N} \stackrel{\text{def}}{=} |\Lambda| = 0 \wedge |\text{ext}(s, x)| = |s| + 1$
3. $* : \text{string} \times \text{string} \longrightarrow \text{string} \stackrel{\text{def}}{=} (s * \Lambda = s) \wedge (s_1 * \text{ext}(s_2, x) = \text{ext}(s_1 * s_2, x))$
4. $\text{reverse} : \text{string} \longrightarrow \text{string} \stackrel{\text{def}}{=} \text{reverse}(\Lambda) = \Lambda \wedge \text{reverse}(\text{ext}(s, x)) = \langle x \rangle * \text{reverse}(s)$

Useful Lemmas

1. lemma EmptyNotSingle: $\Lambda \neq \langle x \rangle$
2. lemma IdofEmpty : $\Lambda * \alpha = \alpha$
3. lemma LenofSingle : $|\langle x \rangle| = 1$
4. lemma LenofCat : $|\alpha * \beta| = |\alpha| + |\beta|$
5. lemma AssocCat : $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$
6. lemma ReverseofReverse : $\text{reverse}(\text{reverse}(\alpha)) = \alpha$
7. lemma ReverseofCat : $\text{reverse}(\alpha * \beta) = \text{reverse}(\beta) * \text{reverse}(\alpha)$
8. lemma LenofReverse: $|\text{reverse}(\alpha)| = |\alpha|$

Fig. 11. String theory

It is important to note that addition of new theory developments to Isabelle is outside the “core” prover. No part of Isabelle needs to be changed to add them. It is also important to note that the idea is independent of Isabelle, which is simply serving as a proxy for a general-purpose prover that allows addition of new definitions and results.

First we consider the VCs in Fig. 7 that are generated from the code to flip a queue in Sect. 3.1. The first VC relates to showing termination by showing that the length of the *Queue* passed to the recursive call is strictly smaller than the length of the original *Queue*. Isabelle can prove this easily based on lemmas about the operators involving strings.

Verification Condition #1 from Figure 7 (recursion termination)

$$\begin{aligned} &\text{min_int} \leq 0 \\ \wedge & 0 < \text{max_int} \\ \wedge & \text{Max_Length} > 0 \\ \wedge & |Q| \leq \text{Max_Length} \\ \wedge & \text{P_val} = |Q| \\ \wedge & |Q| \neq 0; \\ \wedge & Q = \langle E1 \rangle * Q1 \end{aligned}$$

$$|Q1| < |Q|$$

The second VC (#3 below) corresponds to the obligation of showing that the ensures clause is true when the if condition holds, i.e., *Queue* length is not 0. It is proven easily by Isabelle based on algebraic properties of reverse and concatenation that are captured in a few oft-used lemmas.

Verification Condition #3 from Figure 7 (ensures clause for recursive case)

$$\begin{array}{l}
 \text{min_int} \leq 0 \\
 \wedge \quad 0 < \text{max_int} \\
 \wedge \quad \text{Max_Length} > 0 \\
 \wedge \quad |Q| \leq \text{Max_Length} \\
 \wedge \quad P_val = |Q| \\
 \wedge \quad |Q| \neq 0; \\
 \wedge \quad Q = \langle E1 \rangle * Q1 \\
 \hline
 \text{reverse}(Q1) * \langle E1 \rangle = \text{reverse}(Q)
 \end{array}$$

Next, we consider proof of VCs arising from the *Set* example in Sect. 3.2. The finite set theory used by RESOLVE provides all of the expected set functions and predicates, such as set membership, union, intersection, set difference, singleton, and subset. It also provides the set cardinality function that represents the size of the set. We note that, without the cardinality function, this theory is decidable [FOS80]. Moreover, such functions also restrict the combinations of decision procedures for other theories (for example, string theory above) because of the potential for “crosstalk” involving the use of natural numbers as the range.

We use two main rules to simplify formulas that involve cardinality: $|a \cup b| = |a| + |b| - |a \cap b|$ and $|a \setminus b| = |a| - |a \cap b|$. Of course the cardinality of a singleton is one and the cardinality of the empty set is 0. These rules have been powerful enough to handle the VCs generated by examples similar to the one in Sect. 3.2 but of course may need to be extended to handle VCs arising from other code in the future. As noted earlier, for technical reasons we do not expect the process of adding new lemmas to Isabelle’s theories or new rules to *SplitDecision* to end with the ability to prove all true VCs. We do, however, expect the frequency with which the math specialist needs to make such improvements to decrease continually as more improvements are made for a particular theory.

As previously noted, for the examples used in the tabular method of generating VCs which involve sets, Isabelle is able to prove all the VCs. Some involve simple manipulations while others require more involved simplifications using the definitions of the set functions and predicates. We present two VCs, along with what the proofs entail.

The first VC exemplifies the interactions between cardinality of finite sets and the definitions of sets, and is shown below. The VC corresponds to proving that the loop body as shown in Fig. 9 actually does decrease the progress metric. The proof is performed automatically by Isabelle via the “auto” tactic. It readily follows by applying various rules that first transform the consequent into $|t_8| - |t_8 \cap \{x_{10}\}| < |t_8|$. Then, an additional rule which simplifies intersections if one of the arguments is a subset of the other argument (note Given 7) is used to further simplify the formula to $|t_8| - |\{x_{10}\}| < |t_8|$. Finally, this formula can be shown to be true using algebra and the fact that $|\{x_{10}\}| = 1$. The proof requires some bookkeeping and use of simple lemmas.

Verification Condition #31 (state index: 15, loop metric)

$$\begin{array}{l}
 \text{Prove} \\
 |t_8 \setminus \{x_{10}\}| < |t_8| \\
 \hline
 \text{Given} \\
 1. \quad t_8 \neq \emptyset \\
 2. \quad s_8 \cup t_8 = t_0 \cup s_0 \\
 3. \quad (s_8 \cap t_8) \cup \text{tmp}_8 = (t_0 \cap s_0) \cup \emptyset \\
 4. \quad t_8 \cap \text{tmp}_8 = \emptyset \\
 5. \quad |t_8| \geq 0 \\
 6. \quad \text{is_initial}(x_9) \\
 7. \quad x_{10} \in t_8 \\
 8. \quad \text{is_initial}(x_{12}) \\
 9. \quad |t_0| > |s_0| \\
 10. \quad x_{10} \notin s_8 \\
 \hline
 \end{array}$$

The VC at the heart of the correctness of the computation is shown next. This VC involves proving that a clause of the loop invariant holds at the end of the loop. In the proof of this VC, instead of using specific facts about cardinality, the proof involves the definitions of set membership with respect to the union, intersection

and set difference functions. For example, an item is in the union of two sets if it is an element of either set. Once the definitions are expanded, it becomes a matter of simple logical manipulations that are performed easily by Isabelle.

Verification Condition #22 (state index: 15, loop invariant)

Prove

$$(s_8 \cap (t_8 \setminus \{x_{10}\})) \cup (tmp_8 \cup \{x_{10}\}) = (s_0 \cap t_0) \cup \emptyset$$

Given

1. $t_8 \neq \emptyset$
2. $s_8 \cup t_8 = s_0 \cup t_0$
3. $(s_8 \cap t_8) \cup tmp_8 = (s_0 \cap t_0) \cup \emptyset$
4. $t_8 \cap tmp_8 = \emptyset$
5. $|t_8| \geq 0$
6. $is_initial(x_9)$
7. $x_{10} \in t_8$
8. $is_initial(x_{14})$
9. $|t_0| \leq |s_0|$
10. $x_{10} \in s_8$

The current theories available for use by Isabelle to prove VCs resulting from RESOLVE programs include finite sets and string theory. Work proceeds on adding more RESOLVE mathematical theories such as tree theory and graph theory to Isabelle. More concrete examples of possible challenges include verifying sorting implementations. These implementations make use of mathematical functions that abstract the notion of sorting, e.g., the statement that every entry in one string is less than or equal to any entry in another string (where “less than or equal to” is a proxy for any total pre-ordering on the actual entry type). These highly generic definitions, while necessary to deal with fully generic code of typical RESOLVE software components, can complicate automated proofs. Furthermore, identification of ways to streamline, augment, and check how RESOLVE mathematical theories are imported to the provers is also ongoing. These theories were developed with ease of specification in mind. In order to support automated verification as well, extra lemmas may be necessary. Moreover, there may be some degree of redundancy in the theories that can be reduced both to aid the specifiers and to aid the automated proof tools.

4.2. Automated proving with SplitDecision

When not using its “auto” mode, Isabelle is an interactive proof assistant. Often a proof assistant will halt its execution and present the user with alternatives for how to proceed, in an effort to reduce the number of “blind alleys” that it might follow in its proof efforts. This dependence on outside intervention is fine for a proof assistant in its primary intended usage scenarios, but it is not necessarily compatible with the desire to have VCs proven completely automatically. Therefore, in parallel with our exploration of Isabelle’s strength in “auto” mode, we are also pursuing the development of a new tool geared specifically for proving VCs from RESOLVE code, called *SplitDecision*.

SplitDecision leverages domain-specific mathematical knowledge along with general-purpose strategies for reducing logical formulae in order to simplify RESOLVE VCs while maintaining logical equivalence. Each action it takes makes progress toward the ultimate goal of a “true” or “false” conclusion, and it never backtracks. It continues to work unidirectionally until truth or falsehood is established, or until no more simplifications are applicable.

The simplifications *SplitDecision* is allowed to perform are defined by specialized decision procedures tuned to handle some of the most-used mathematical theories involved in RESOLVE specifications. The goal of this work is for *SplitDecision* to be able to prove/disprove any RESOLVE VC, or at least reduce it to a form that a proof assistant will need no further guidance to prove/disprove. So far, a decision procedure for an important category of assertions in mathematical string theory [Fri09] has been implemented. A theory of arrays has also been handled similarly and efficiently (e.g., in [Bar02, BrM07, GNR07]).

SplitDecision is an ongoing project. It was originally built partly with the intent of making the VCs simpler for humans to read and (possibly) smaller for a general-purpose prover like Isabelle to start with. It has, instead, turned out to be able to prove outright a large fraction of the VCs we have examined thus far [KAB09].

5. Proof checking

While the VCs that arise from the software specialist's program are to be dispatched automatically, this process is supported by theorems suggested by the math specialist. As has already been discussed, a “false theorem” would compromise the soundness of the entire system and as a result it is important that any putative theorems themselves be verified. However, since these theorems might be of arbitrary complexity, in general, they cannot be verified automatically. To this end, the math specialist may provide explicit proofs in a formalized language that can then be checked by the system. One such proof checking system integrated within the RESOLVE framework is discussed in detail in [SRS08]. Since these theories are organized into libraries around mathematical concepts (e.g., strings) which are reused by many different programming objects (e.g., stacks, queues, and lists), the overhead associated with writing and checking proofs of non-trivial theorems is amortized over time and reuse.

For example, consider the theorem:

$$\forall \alpha, \beta (\text{reverse}(\alpha \cdot \beta) = \text{reverse}(\beta) \cdot \text{reverse}(\alpha))$$

from Sect. 2. This theorem was necessary to complete the proof of the third VC in Fig. 7, which arose from the verification of the *Queue* flipping code. Such theorems are found in a RESOLVE mathematical unit *String_Theory*, called a *précis*, that contains a summary of the results in a theory. Here is what the example theorem looks like in context:

```

Precis String_Theory;
...
  Theorem S1: For_all alpha, beta:
    reverse(alpha o beta) =
      reverse(beta) o reverse(alpha);
...
end String_Theory;

```

The *précis* is an interface to mathematical theory developments and it provides only theorems, such as the one above, so that they can be used by clients such as the automated prover, the math specialist, or the software specialist. Corresponding proofs are provided in a separate *proof unit*, also a part of the RESOLVE framework. A design goal of the RESOLVE proof-writing language is to highlight the separation between results, which are used by consumers like the prover, and corresponding proofs, which are used only to ensure soundness. We note that math specialists are free to use other tools, such as automated provers or proof assistants, to help generate the required proofs.

One primary design challenge in developing a proof checker is balancing the needs of an explicit, formal language with the expectations of trained mathematicians rather than programmers. The current version of the RESOLVE proof checker is described in [SRS08] with an illustrative example.

6. Related work and conclusions

There are many forks in the road of research toward verified software [LAB06] and there are many fundamental questions [Jon03]. Considerable related work lies down the paths *not* taken by our team, all of which offer hope for progress that other researchers will and should continue to pursue. The rationale for each major decision made so far in our efforts is summarized below, along with relevant related work.

First, it is important to establish the scope of software being considered. There are a number of options that fall along a spectrum including (a) concurrent/distributed software with a focus on a limited set of properties to be verified, and (b) full behavioral verification of sequential software. Ultimately, the vision is to be able to specify and verify all software, but our work focuses on the latter type—specifically, on sequential software *components*, because all useful software today is component-based. The work is complementary in principle to most recent work on finite-state model-checking, etc. (which suffers equally from the technical drawback that it cannot address the entire problem, e.g., software systems whose appropriate models are manifestly not finite-state).

Within the context of software components, a major choice involves the goal—indeed, the definition—of “verification”. The question is whether to prove that software satisfies (a) some fixed set of properties, e.g., that there are no null dereferences or that no array indices are out of range [DLN98]; or (b) a partial specification of

behavior, e.g., that adding an element to a queue increases the length of the queue by one (without stipulating anything about, for example, the order of the actual queue elements) [Mey92]; or (c) a complete behavioral specification of what the software is supposed to do and what it is not supposed to do [AiL97, Jac06, LBR06, ORR96, ZKR08]. Our goal is the last one. This “full functional” notion of verification is certainly the most difficult to achieve, but accounting for complete behavior will be required eventually, and this goal subsumes the others.

Another important decision is whether to try to verify software written in (a) some industrial language, e.g., Java or C or C++ or C#, or (b) a clean language such as RESOLVE [Kul04, SiW94] that lends itself to systematic exploration of fundamental research questions related to verified software. A critical point is that RESOLVE has been designed to support *modular* verification based on the principle of design-by-contract [Mey92]: each component is verified in isolation, once and for all, rather than being verified every time it is used in a client program. Modular verification is difficult in the presence of unrestricted uses of references and aliasing, because code in one component may affect the reasoning in another [Kul04, MuP00, WeH01]. The ability to verify one component at a time is necessary for the verified software paradigm to be scalable to practical-size programs. RESOLVE also is rich enough that software components written in it are at least as general and efficient as their counterparts in the above languages, and the RESOLVE discipline and component catalog with coding in C++ has been used successfully to develop commercial software [HBW00]; in addition the code for *SplitDecision* is written in C++ using this style. Translating verified RESOLVE components to Java is the topic of [SHF09]. A summary of objectives and approaches to specification and verification in modern languages along with various pitfalls may be found in [LLM08]. While there are many similarities between the objectives of the RESOLVE system and the discussion in [LLM08], there are notable differences. RESOLVE specifications are not intended for run-time execution and RESOLVE programmers are not expected to be involved in significant theory development.

A final major decision involves mechanical theorem-proving tools used to prove VCs. There are many candidates for such a back-end. SMT solvers (e.g., [BaT07, deM08, DNS05]) are designed to verify (decide) the universal (i.e., quantifier-free) fragment of the union of disjoint theories whose universal fragments are independently decidable. So, for example, to handle a quantifier-free VC that involves some linear arithmetic, reasoning about arrays, generic function symbols, and equality, SMT solvers are ideal. They are remarkably powerful and efficient. On the other hand, they are limited in their capabilities for verifying VCs involving assertions and definitions with quantifiers, against large background theories that are undecidable. Since the goal is general software verification, RESOLVE specifications and hence VCs often involve these complications, though these issues are not immediately apparent from the examples in this paper; an interested reader may consult [KHS08], for additional details. When the generated VCs are within scope, ongoing experimentation includes the use of the SMT solver Z3 for automated verification [deM08].

Unlike SMT solvers, Isabelle is a general-purpose theorem prover for higher-order logic assertions. It is intended to be a proof assistant for professional mathematicians but it can also serve as a fully automated prover. Isabelle offers several technical benefits as it serves now as a proxy for the ultimate automated prover. Like RESOLVE's specification language, it is not limited to first-order logic, but instead supports higher-order logic (Isabelle/HOL [NPW02]). Crucially, Isabelle provides facilities both to define new mathematical theories (with associated lemmas, theorems, and corollaries), and to incorporate new mathematical results into its automated reasoning. This ability to create new theories is integral to the task at hand because it provides the right layer of abstraction between the specifics of how Isabelle performs proofs and the mathematical theories used in RESOLVE specifications. In addition, Isabelle can be configured so that it can send the VCs to a powerful fully automated first-order prover, such as Vampire [RiV02] or E [Sch04]. This means there is no reason for the RESOLVE VC generator to have separate interfaces to such tools; Isabelle itself serves as that interface.

The rest of this section elaborates on closely related verification tools. Like RESOLVE, Spec# focuses on specifying an imperative language and providing compile-time push-button verification. Unlike RESOLVE, Spec# [BLS04, LeM08] is based on an existing commercial language, C#. It does not significantly alter that language to ease verification, though it does offer at least one new feature to ease specification: non-null references. Whereas RESOLVE compiles mathematical assertions directly to VCs, Spec# compiles first to an intermediate language, Boogie [BaC06], intended to generalize the verification process. Spec# also implements dynamic checks in addition to static ones so that specified components may properly interact with unspecified ones. Where RESOLVE uses swapping to avoid aliasing, limiting reasoning about reference behavior to those components that explicitly use a pointer component [Kul04, KSW06], Spec# allows routine aliasing. However, to simplify reasoning and the type-ownership paradigm, it includes immutability at the object rather than class level [LMW08]. In order to simplify verification in the presence of aliased references, it depends instead on the programmer to employ an involved set of annotations on how each reference can be used at a point in the code. Of course, these annotations

themselves may lead to new proof obligations [DaM06]. Dafny [Lei10] is a programming language and verification system, developed in conjunction with the Boogie project [BaC06]. Dafny is a research language that seeks to achieve modular verification in the presence of reference semantics and user-defined generics. Like RESOLVE, Dafny manifests specifications in the form of loop invariants and operation contracts, but because of the necessity of reasoning about the heap, these also involve annotations about the set of objects affected by the code's execution. The need for these frame-related assertions and the lack of direct support for abstract mathematical modeling for new classes complicate modular verification in Dafny. The Dafny project has attempted substantive yet incomplete solutions to the benchmarks listed in [WSH08].

Jahob [ZKR08, ZKR09] is intended to support automatic verification of Java classes that are similar (but not identical) to those in the popular Java collections framework. By adding syntax through which the programmer supplies detailed hints to a back-end theorem prover in a prover-neutral format [ZKR09], far more code can be verified automatically than previously could be done without such hints [ZKR08]. Jahob, like other attempts to verify code in Java and other languages that were not designed to support modular verification, faces difficulties when applied to client code that uses these previously verified classes, because of aliasing that can leak across abstraction boundaries.

ESC/Java2 is a system that applies the Extended Static Checking [DLN98] system to the Java programming language. ESC/Java2 operates on JML annotations within a Java program. Like the RESOLVE system, it strives to statically detect errors in code. Unlike RESOLVE, it does not strive to be sound or relatively complete, but rather to flag a class of potential problems. Like Spec#, ESC/Java2 builds on top of an existing commercial language, and encourages incremental specification rather than full program specification. When combined with a development environment, such as when used with a plug-in for Eclipse, it provides limited push-button verification. Aside from ESC/Java2, there are a number of different verification efforts underway that also use JML to specify Java code [BCC05]. One example is the work in [KCJ08] for translating Java/JML correctness checks into a Hoare-style logic processed by Isabelle. Verification based on JML is also concerned with keeping up with the evolving Java language [Cok08].

Unlike JML and Spec#, some verification tools are based on subsets of popular languages. The focus of SPARK (based on Ada) is safety-critical software [Bar02]; current SPARK tools can verify some of the earlier benchmarks in [WSH08].

The Verisoft system [AHL08] is unique in that it is intended to address the verification of an entire system down to the level of hardware and device drivers. For example, the group's work includes developing a verified compiler for C0, a restricted subset of C [LPP05]. However, full automation of verification is not an explicit goal.

The Coq IDE provides a graphical front-end for working with the Coq proof management system. The user is able to load up a Coq file and incrementally step through portions of the file, getting useful feedback about where and why problems are encountered. Like the RESOLVE system, the Coq IDE provides immediate feedback on the veracity of mathematical statements with the additional feature of helping to establish where exactly the problem lies. Coq is not intended as a programming language but provides useful proving and proof-checking features for a verification system [Coq10].

The Why/Krakatoa/Caduceus platform can produce VCs from annotated Java and C source code [FiM07]. It employs an intermediate programming language for the generation of VCs which can be output in the format of several different third party provers, including Isabelle and Coq.

Software verification is a grand challenge. To address the challenge, we need to automate solutions to a variety of subproblems. This paper illustrates the progress we have made by discussing in detail a toolset that we have developed for software verification in RESOLVE, a language designed to be amenable for verification. But obstacles to automation remain. They range from fine-tuning techniques for mathematical theory and software development to more effective techniques for automating proofs.

Acknowledgments

This material is based upon work supported by the U. S. National Science Foundation under grants CCF-0811748, CCF-0811737, DMS-0700174, DMS-0701187, DMS-0701260, and DUE-0633506, and by a grant from the John Templeton Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the John Templeton Foundation. Our thanks are due to Bill Ogden for his numerous contributions to the foundations of this research.

Appendix A: Specification of Queue_Template

```

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

Type Family Queue is modeled by Str(Entry);
exemplar Q;
constraint |Q| <= Max_Length;
initialization ensures Q = empty_string;

Operation Enqueue( alters E: Entry;
                  updates Q: Queue);
requires |Q| < Max_Length;
ensures Q = #Q o <#E>;

Operation Dequeue( replaces R: Entry;
                  updates Q: Queue);
requires |Q| /= 0;
ensures #Q = <R> o Q;

Operation Swap_First_Entry( updates E: Entry;
                           updates Q: Queue);
requires |Q| /= 0;
ensures there exists Rem: Str(Entry) such that
  #Q = <E> o Rem and Q = <#E> o Rem;

Operation Length(restores Q: Queue): Integer;
ensures Length = (|Q|);

Operation Rem_Capacity(restores Q: Queue): Integer;
ensures Rem_Capacity = (Max_Length - |Q|);

Operation Clear(clears Q: Queue);

end Queue_Template;

```

Appendix B: Specification of SetTemplate

```

contract SetTemplate (type Item)
uses UnboundedIntegerFacility

math subtype SET_MODEL is finite set of Item

type Set is modeled by SET_MODEL
exemplar s
initialization ensures
  s = empty_set

procedure Add (updates s: Set, clears x: Item)
requires x is not in s
ensures s = #s union {x}

procedure Remove (updates s: Set, restores x: Item,
                 replaces x_copy: Item)
requires x is in s
ensures s = #s \ {x} and x_copy = x

procedure RemoveAny (updates s: Set, replaces x: Item)
requires s /= empty_set
ensures x is in #s and s = #s \ {x}

function IsMember (restores s: Set, restores x: Item):
control
ensures IsMember = (x is in s)

function IsEmpty (restores s: Set): control
ensures IsEmpty = (s = empty_set)

function Size (restores s: Set): Integer
ensures Size = |s|

end SetTemplate

```

References

- [AiL97] Aichernig BK, Larson PG (1997) A proof obligation generator for VDM-SL, In: Proc. FME 4, LNCS 1313. Springer, New York
- [AHL08] Alkassar E, Hillebrand M, Leinenbach D, Schirmer NW, Starostin A (2008) The Verisoft approach to system verification. In: Proc. VSTTE 2008. Springer, New York
- [Bar02] Barnes J (2002) High-integrity Ada: the Spark approach. Addison-Wesley, USA
- [BaC06] Barnett M, Chang BYE et al (2006) Boogie: a modular reusable verifier for object-oriented programs. In: de Boer FS, Bonsangue MM, de Roeper W-P (eds) Proc. FMCO 4, LNCS 4111. Springer, pp 364–387
- [BLS04] Barnett M, Leino KR, Schulte W (2004) The Spec# programming system: an overview. In: Burdy L, Huisman M (eds) Construction and analysis of safe, secure and interoperable smart devices international workshop, LNCS 3362. Springer, pp 49–69
- [BaT07] Barrett C, Tinelli C (2007) CVC3. In: Damm W, Hermanns H (eds) Proc. CAV 19, LNCS 4590. Springer, pp 298–302
- [BrM07] Bradley A, Manna Z (2007) The calculus of computation: decision procedures with applications to verification. Springer, New York
- [BCC05] Burdy L, Cheon Y, Cok D, Ernst M, Kiniry J, Leavens GT, Leino KRM, Poll E (2005) An overview of JML tools and applications. STTT 7(3): 212–232
- [Cok08] Cok D (2008) Adapting JML to generic types and Java 1.6. In: Proc. SAVCBS, pp 27–35. <http://www.eecs.ucf.edu/SAVCBS/2008/SAVCBS08-proceedings.pdf>
- [Coq10] The Coq proof assistant reference manual version v8.1. <ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual.pdf>
- [DaM06] Darvas Á, Müller P (2006) Reasoning about method calls in interface specifications. J Object Technol (JOT) 5(5):50–85
- [deM08] de Moura L, Bjørner N (2008) Z3: An efficient SMT solver. In: Proc. TACAS, pp 337–340
- [DLN98] Detlefs DL, Leino KRM, Nelson G, Saxe JB (1998) Extended static checking. Research Report 159, Compaq Systems Research Center
- [DNS05] Detlefs D, Nelson G, Saxe JB (2005) Simplify: a theorem prover for program checking. JACM 52(2):365–473
- [EHO94] Ernst GW, Hookway RJ, Ogden WF (1994) Modular verification of data abstractions with shared realizations. IEEE TSE 20(4):288–307
- [FOS80] Ferro A, Omodeo EG, Schwartz JT (1980) Decision procedures for elementary sublanguages of set theory. I. Multi-level syllogistic and some extensions. Commun Pure Appl Math 33:599–608
- [FiM07] Filliâtre J, Marché C (2007) The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm W, Hermanns H (eds) Proc. CAV 19, LNCS 4590. Springer, New York
- [Fri09] Friedman HM (2009) Deciding statements about strings with applications to program verification. OSU-CISRC-8/09-TR42, Department of Computer Science and Engineering, The Ohio State University
- [GNR07] Ghilardi S, Nicolini E, Ranise S, Zucchelli D (2007) Decision procedures for extensions of the theory of arrays. Ann Math Artif Intell 50(3–4):231–254
- [Har08] Harrison J (2008) Formal proof—theory and practice. Notices of the AMS 55:1395–1406
- [HaW91] Harms DE, Weide BW (1991) Copying and swapping: influences on the design of reusable software components. IEEE TSE 17(5):424–435
- [HSK08] Harton HK, Sitaraman M, Krone J (2008) Formal program verification. In: Wah B (ed) Wiley Encyclopedia of Computer Science and Engineering. Wiley, London
- [Hey95] Heym W (1995) Computer program verification: improvements for human reasoning. Ph.D. thesis, The Ohio State University
- [Hoa72] Hoare CAR (1972) Proof of correctness of data representations. Acta Inform 1:271–281
- [Hoa03] Hoare CAR (2003) The verifying compiler: a grand challenge for computing research. JACM 50:63–69
- [HML07] Hoare CAR, Misra J, Leavens GT, Shankar N (2007) The verified software initiative: a manifesto, <http://qpq.csl.sri.com/vsr/manifeto.pdf/view>. Accessed December 2008
- [HBW00] Hollingsworth JE, Blankenship L, Weide BW (2000) Experience report: using RESOLVE/C++ for commercial software. In: Schneir B (ed) Proc. FSE. ACM, pp 11–19
- [Jac06] Jackson D (2006) Software abstractions: logic, language, and analysis. MIT Press, Cambridge
- [Jon03] Jones CB (2003) The early search for tractable ways of reasoning about programs. IEEE Ann Hist Comput 25(2):26–49
- [KCJ08] Karabotsos G, Chalin P, James PR, Giannas L (2008) Total correctness of recursive functions using JML4 FSPV. In: Proc. SAVCBS, pp 19–27. <http://www.eecs.ucf.edu/SAVCBS/2008/SAVCBS08-proceedings.pdf>
- [Kin70] King JC (1970) A Program Verifier. Ph.D. dissertation, Carnegie Tech, 261 pp
- [KHS08] Kirschenbaum J, Harton HK, Sitaraman M (2008) A case study in automated verification. In: Shankar N (ed) Proc. CAV AFM Workshop
- [KAB09] Kirschenbaum J, Adcock B, Bronish D, Smith H, Harton H, Sitaraman M, Weide BW (2009) Verifying component-based software: deep mathematics or simple bookkeeping? In: Edwards SH, Kulczycki G (eds) Proc. ICSR 11, LNCS 5791. Springer, pp 31–40
- [Kro88] Krone J (1988) The role of verification in software reusability. Ph.D. Thesis, The Ohio State University
- [KSY08] Kulczycki G, Sitaraman M, Yasmin N, Roche K (2008) Formal specification. In: Wah B (ed) Wiley Encyclopedia of Computer Science and Engineering. Wiley, London
- [Kul04] Kulczycki G (2004) Direct reasoning. Ph.D. Dissertation, Clemson University
- [KSW06] Kulczycki G, Sitaraman M, Weide BW, Rountev A (2006) A specification-based approach to reasoning about pointers. ACM SIGSOFT Softw Eng Notes 31:55–62
- [LBR06] Leavens GT, Baker AL, Ruby C (2006) Preliminary design of JML: a behavioral interface specification language for Java. ACM Softw Eng Notes 31:1–38

- [LAB06] Leavens GT, Abrial J-R, Batory D, Butler M, Coglio A, Fislser K, Hehner E, Jones C, Miller D, Peyton-Jones S, Sitaraman M, Smith DR, Stump A (2006) Roadmap for enhanced languages and methods to aid verification. In: Proc. GPCE 5. ACM Press, New York, pp 221–236
- [LLM08] Leavens GT, Leino KRM, Müller P (2008) Specification and verification challenges for sequential object-oriented programs. *Formal Aspects Comput* 19(2):159–189
- [LPP05] Leinenbach D, Paul W, Petrova E (2005) Towards the formal verification of a C0 compiler: code generation and implementation correctness. *SEFM* 3:2–11
- [LMW08] Leino KRM, Müller P, Wallenburg A (2008) Flexible immutability with frozen objects. In: Shankar N (ed) Proc. VSTTE 2008, LNCS 5295. Springer, New York, pp 192–208
- [LeM08] Leino KRM, Müller P (2008) Using the Spec# language, methodology, and tools to write bug-free programs. *LASER 2007/2008 Lecture Notes*, Springer, New York
- [Lei10] Leino KRM (2010) Dafny: An automated program verifier for functional correctness. In: Proc. LPAR 16 (in press)
- [Mey92] Meyer B (1992) Applying design by contract. *Computer* 25:40–51
- [MuP00] Müller P, Poetzsch-Heffter A (2000) Modular specification and verification techniques for object-oriented software components. In: Leavens GT, Sitaraman M (eds) *Foundations of component-based systems*. Cambridge University Press, London
- [NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic. LNCS 2283. Springer, New York
- [ORR96] Owre S, Rajan SP, Rushby JM, Shankar N, Srivas M (1996) PVS: combining specification, proof checking, and model checking. In: Alur R, Henzinger TA (eds) Proc. CAV, LNCS 1102. Springer, Berlin, pp 411–414
- [RiV02] Riazanov A, Voronkov A (2002) The design and implementation of VAMPIRE. *AI Commun* 15:91–110
- [Sch04] Schulz S (2004) System abstract: E 0.81. In: Proc. 2nd IJCAR. LNAI 3097. Springer, Berlin, pp 223–228
- [SiW94] Sitaraman M, Weide BW (1994) Component-based software using RESOLVE. *ACM SIGSOFT Softw Eng Notes* 19:21–67
- [SWO97] Sitaraman M, Weide BW, Ogden WF (1997) On the practical need for abstraction relations to verify abstract data type representations. *IEEE TSE* 23(3):157–170
- [SAK00] Sitaraman M, Atkinson S, Kulczycki G, Weide B, Long T, Bucci P, Pike S, Heym W, Hollingsworth J (2000) Reasoning about software-component behavior. In: Proc. ICSR 6, LNCS 1844. Springer, Berlin, pp 266–283
- [SRS08] Smith H, Roche K, Sitaraman M, Krone J, Ogden WF (2008) Integrating math units and proof checking for specification and verification. In: Proc. SAVCBS, pp 59–66. <http://www.eecs.ucf.edu/SAVCBS/2008/SAVCBS08-proceedings.pdf>
- [SHF09] Smith H, Harton H, Frazier D, Mohan R, Sitaraman M (2009) Generating verified Java components through RESOLVE. In: Edwards SH, Kulczycki G (eds) Proc. ICSR 11, LNCS 5791. Springer, Berlin, pp 11–20
- [WSH08] Weide BW, Sitaraman M, Harton HK, Adcock B, Bucci P, Bronish D, Heym WD, Kirschenbaum J, Frazier D (2008) Incremental benchmarks for software verification tools and techniques. In: Shankar N (ed) Proc. VSTTE 2008, LNCS 5295. Springer, Berlin, pp 84–98
- [WeH01] Weide BW, Heym W (2001) Specification and verification with references. In: Proc. SAVCBS, pp 50–59
- [WEH94] Weide BW, Edwards SH, Harms DE, Lamb DA (1994) Design and specification of iterators using the swapping paradigm. *IEEE TSE* 20(8):631–643
- [Win90] Wing JM (1990) A specifier's introduction to formal methods. *IEEE Comput* 23(9):8–24
- [ZKR08] Zee K, Kuncak V, Rinard M C (2008) Full functional verification of linked data structures. In: Proc. PLDI. ACM Press, New York, pp 349–361
- [ZKR09] Zee K, Kuncak V, Rinard M C (2009) An integrated proof language for imperative programs. In: Proc. PLDI. ACM Press, New York, pp 338–351

Received 20 February 2009

Revised: 20 October 2009

Accepted in revised form 11 March 2010 by T. Margaria, D. Kröning and J. Woodcock

Published online 14 April 2010