Automatically Proving Thousands of Verification Conditions Using an SMT Solver: An Empirical Study

Aditi Tagore, **Diego Zaccai**, Bruce W. Weide
The RESOLVE Software Research Group
Department of Computer Science and Engineering
The Ohio State University

Research supported by the NSF under grants DMS-0701260, CCF-0811737, DUE-0942542, and ECCS-0931669

The Study

- Empirical study using an SMT solver to automatically prove Verification Conditions (VCs)
  - 4028 VCs studied
  - bit.ly/nfmVC
- We believe all these VCs are valid (from correct code)

The Study

- VCs from about 50 RESOLVE software components
  - About 2000 lines of code
  - Arithmetic algorithms
  - Sorting of items with arbitrary orders
  - Client manipulations and data representations for stacks, queues, lists, sets, etc.

The Study

- Specifications based on standard design-by-contract principles and annotations, but very few other assertions
- Strict separation between code and the mathematics used to write contracts
- Approach does not assume that programmers have any knowledge about intricacies of a theorem-prover

Contributions

- First automated proof of a sorting algorithm where client supplies both:
  - Type to be sorted, *and*
  - Ordering (any total pre-order)

Contributions

- Highlight the importance of universal algebraic lemmas to an automated prover, in lieu of expanded definitions of user-defined mathematical symbols
  - Not only as a way of providing information but also to hide unnecessary complexity

User-Defined Symbols

- Necessary for any specification language
- Math functions or predicates in RESOLVE can be defined
  - Explicitly
  - Implicitly
- Restrictions on template parameters raise similar issues

A Simple Example

- Definition body:

```
definition IS_ODD (
    n: integer
): boolean
is
    there exists k: integer
        (n = 2 * k + 1)
```

A Simple Example

- Sample use in a contract:

```
procedure Halve (
    updates i: Integer
)
    requires
        not IS_ODD(i)
    ensures
        i * i = #i
```

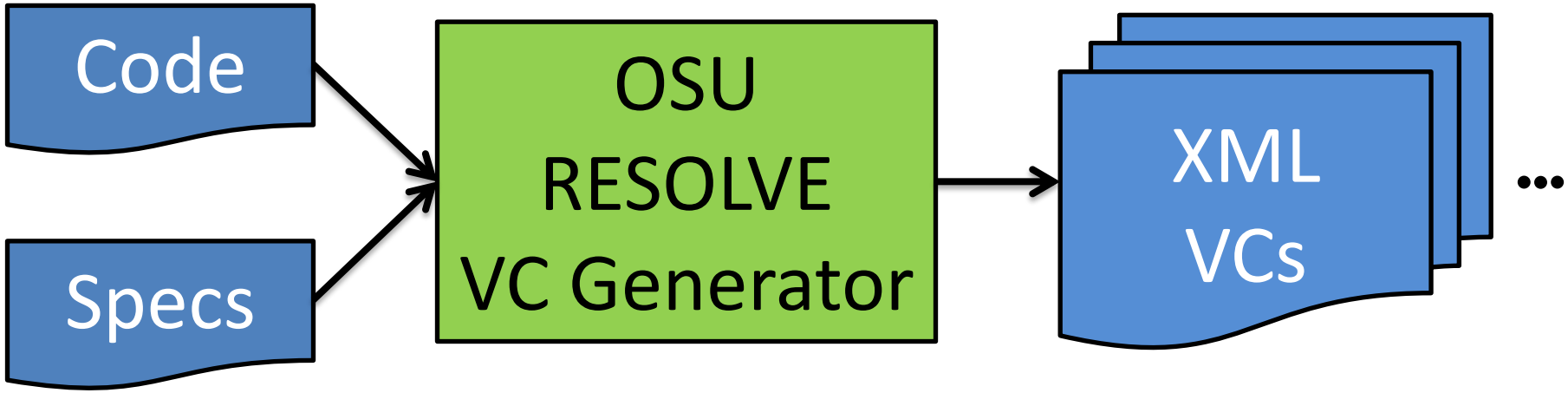
A Simple Example

- Some algebraic properties:

for all x : **integer**

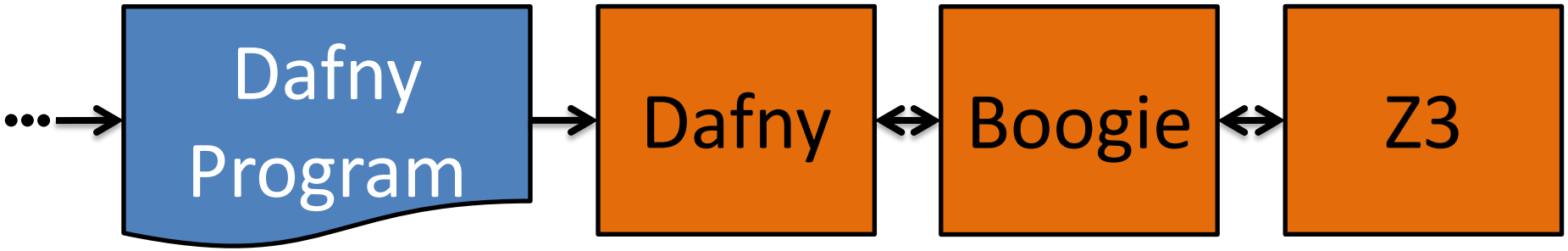
- IS_ODD(x) ⇒ **not** IS_ODD(x + 1)
- **not** `IS_ODD(x)` ⇒ IS_ODD(x + 1)
- **not** `IS_ODD(x+x)`

The Tool Chain

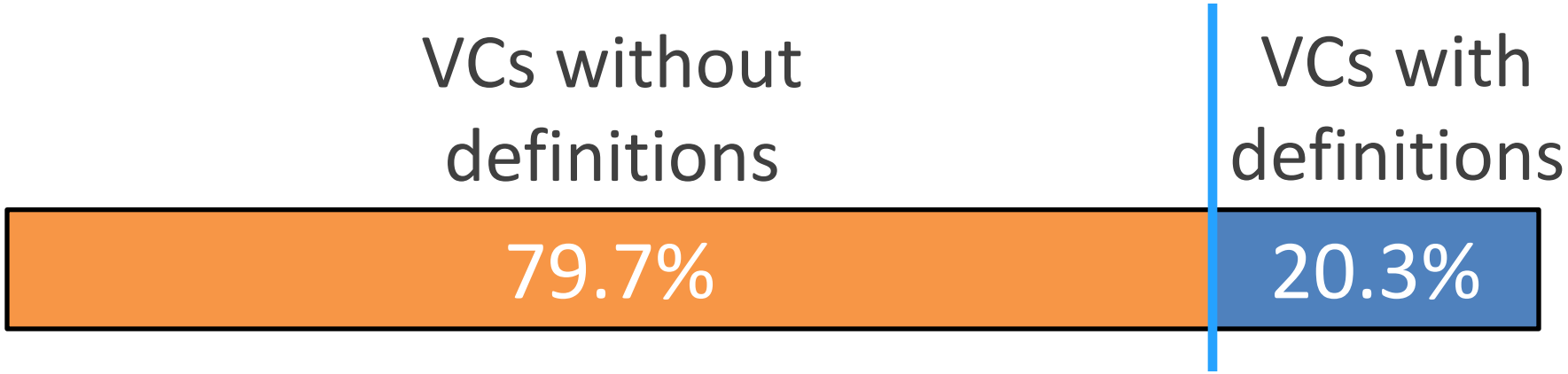


The Tool Chain

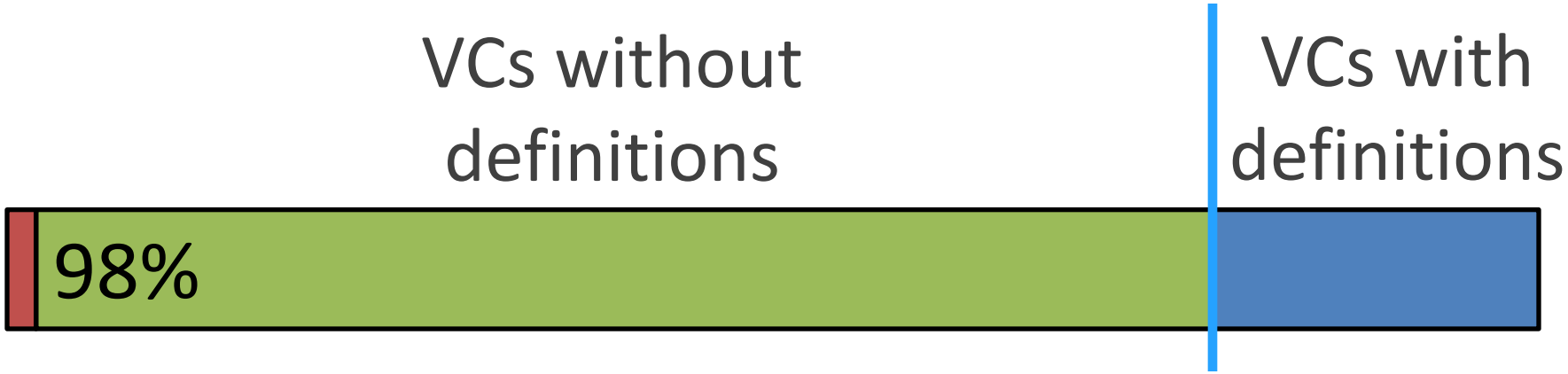
The Tool Chain



Two Categories of VCs



VCs With No User-Defined Symbols



First Attempt

- Treat user-defined symbols as uninterpreted symbols
- This is the only step of interest for VCs without user-defined symbols

First Attempt - Sample VC

Prove:

ARE_PERM (‹› ∘ q14 ∘ ‹x6› ∘ q24,
q10 ∘ ‹› ∘ ‹›)

Given:

‹x6› ∘ ‹› ≠ ‹›

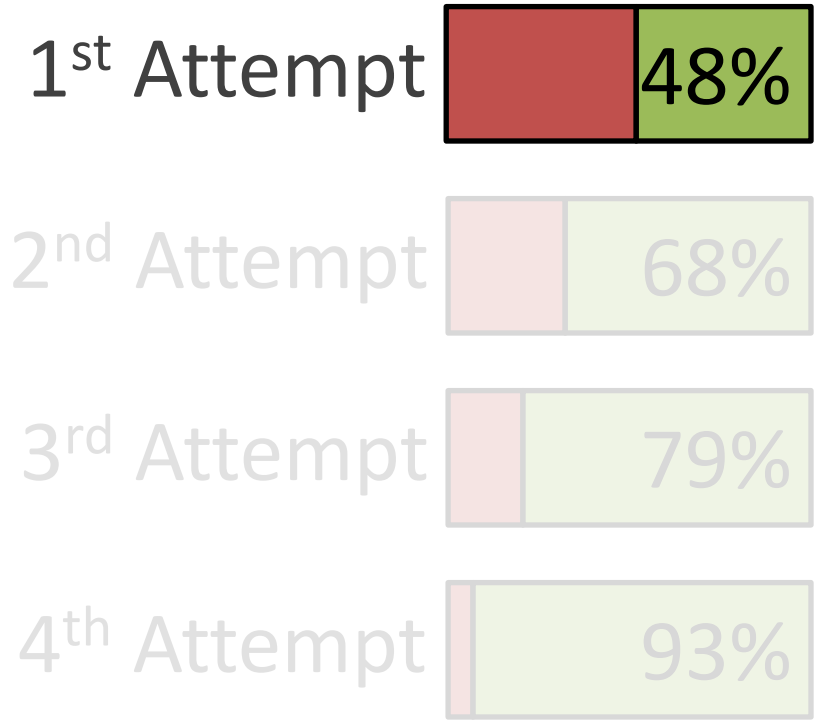
∧ …

∧ ‹x6› ∘ ‹› = ‹›

First Attempt - Results

- Can only prove VCs whose truth is unrelated to the user-defined symbols
- Only 48% of VCs with user-defined symbols are proven

Results for VCs with definitions



Second Attempt

- Provide to the prover the bodies of definitions of user-defined symbols

Second Attempt - Example Body

```
definition ARE_PERMUTATIONS (
    s1: string of Item,
    s2: string of Item
): boolean
is
    for all i: Item
        (OCCURS_COUNT(s1, i)=
         OCCURS_COUNT(s2, i))
```

Second Attempt - Sample VC

Prove:

ARE_PERM(‹› ∘ q10 ∘ q23 ∘ ‹q2Item3›,
‹› ∘ q10 ∘ q23 ∘ ‹q2Item3›)

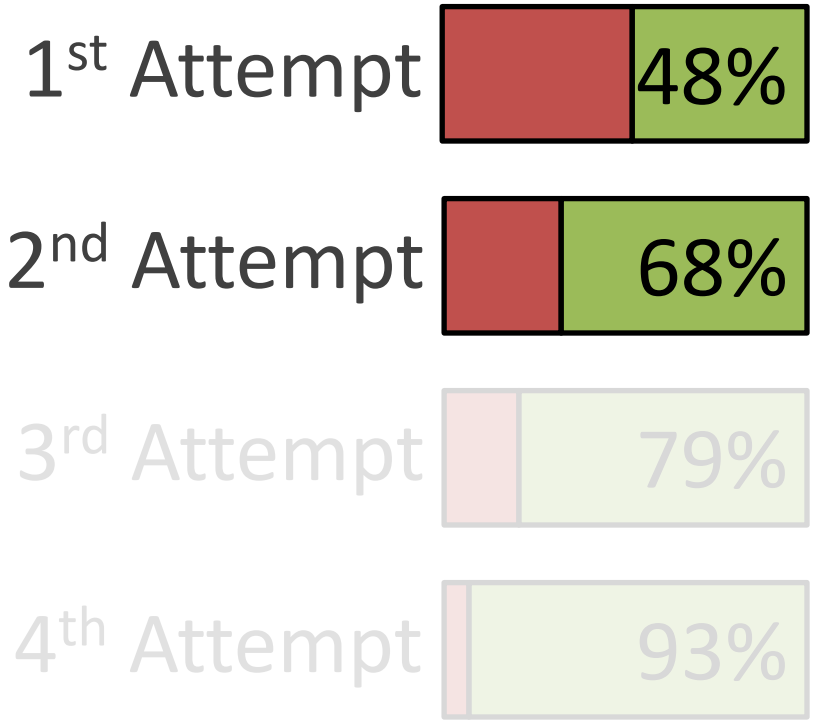
Given:

…

Second Attempt - Results

- Great improvement as Z3 can now reason about the provided symbols
- Still not enough; only 68% of VCs are proven

Results for VCs with definitions



Third Attempt

- Provide both the definition bodies and some universal algebraic lemmas
  - Most lemmas are evident from the development of the mathematical definitions, not their use in contracts
  - Few are not mathematically interesting, and instead arose from attempting to prove VCs

Third Attempt - Sample Lemmas

for all a,b,c: **string of** Item

- ARE_PERM (a, a)
- ARE_PERM (a, b) **and** ARE_PERM (b, c)
  ⇒ ARE_PERM (a, c)
- ARE_PERM (a, b) ⇒ ARE_PERM (b, a)
- ARE_PERM (a ∘ b, b ∘ a)
- …

Third Attempt - Sample VC

Prove:

ARE_PERM (
  tmp4 ∘ ‹q2Item4› ∘ q24 ∘ q16 ∘ ‹q1Item6›,
    ‹› ∘ q10 ∘ q23 ∘ ‹q2Item3›)

Given:

ARE_PERM (
  tmp4 ∘ ‹q1Item6› ∘ q16 ∘ q24 ∘ ‹q2Item4›,
    ‹› ∘ q10 ∘ q23 ∘ ‹q2Item3›) ∧ …

Third Attempt - Sample VC

Prove:

ARE_PERM (
  tmp4 ∘ ‹q2Item4› ∘ q24 ∘ q16 ∘ ‹q1Item6›,
    ‹› ∘ q10 ∘ q23 ∘ ‹q2Item3›)

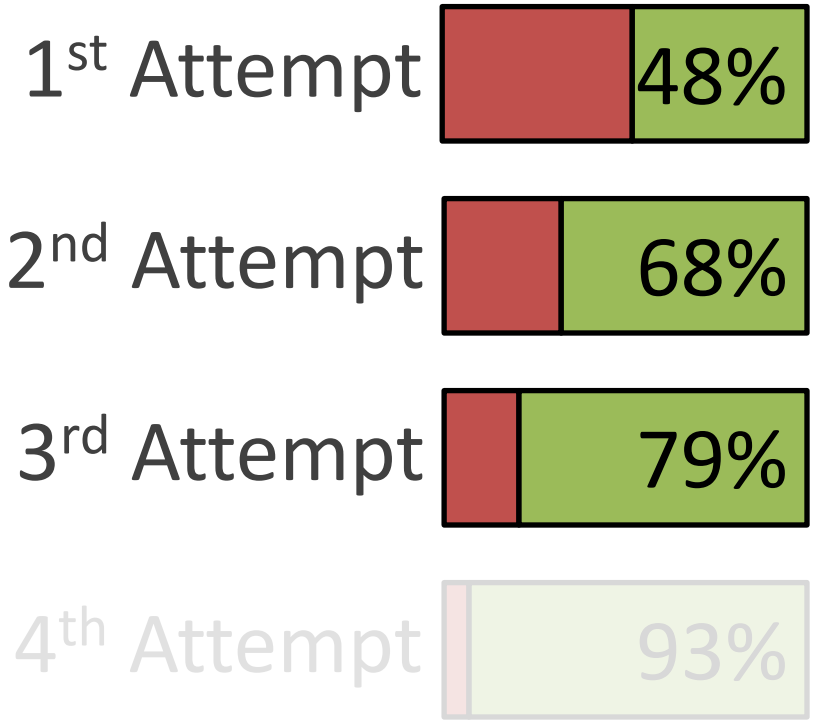
Given:

ARE_PERM (
  tmp4 ∘ ‹q1Item6› ∘ q16 ∘ q24 ∘ ‹q2Item4›,
    ‹› ∘ q10 ∘ q23 ∘ ‹q2Item3›) ∧ …

Third Attempt - Results

- The lemmas provide useful simplification rules, and about 79% of VCs are proven

Results for VCs with definitions



Fourth Attempt

- Keep universal algebraic lemmas from the third attempt
- *Remove* the bodies of definitions of user-defined symbols

Fourth Attempt - Intuition

- Removing the definitions diminishes the complexity of the quantifier structure of a VC
  - Preserves encapsulation of definitions
  - Advantages of user-defined symbols are no longer restricted to the human readers

Fourth Attempt - Sample VC

Prove:

IS_NONDEC (‹q2Item4› ∘ q24)

Given:

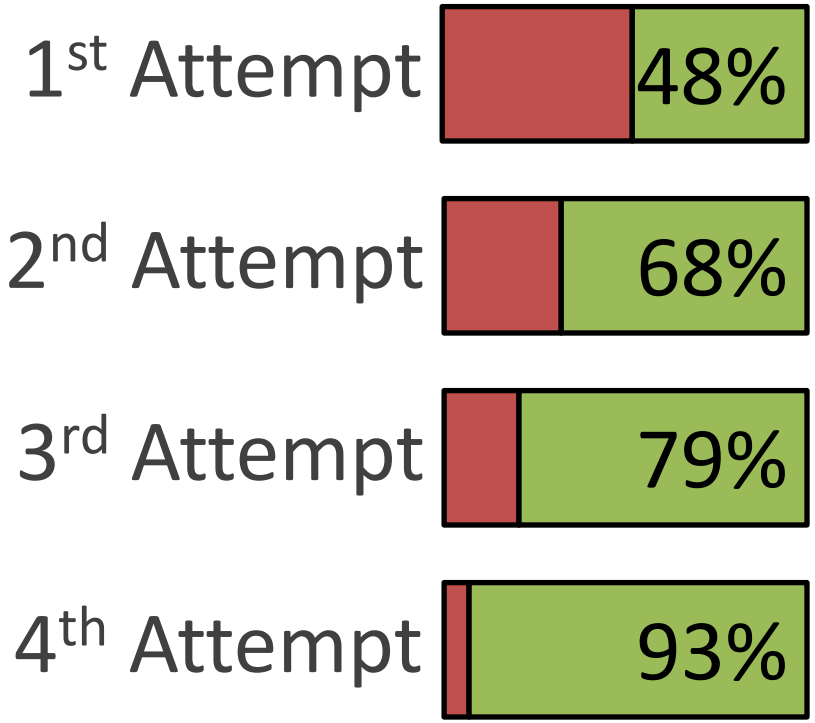
IS_NONDEC (tmp4 ∘ ‹q2Item4› ∘ q24)

∧ …

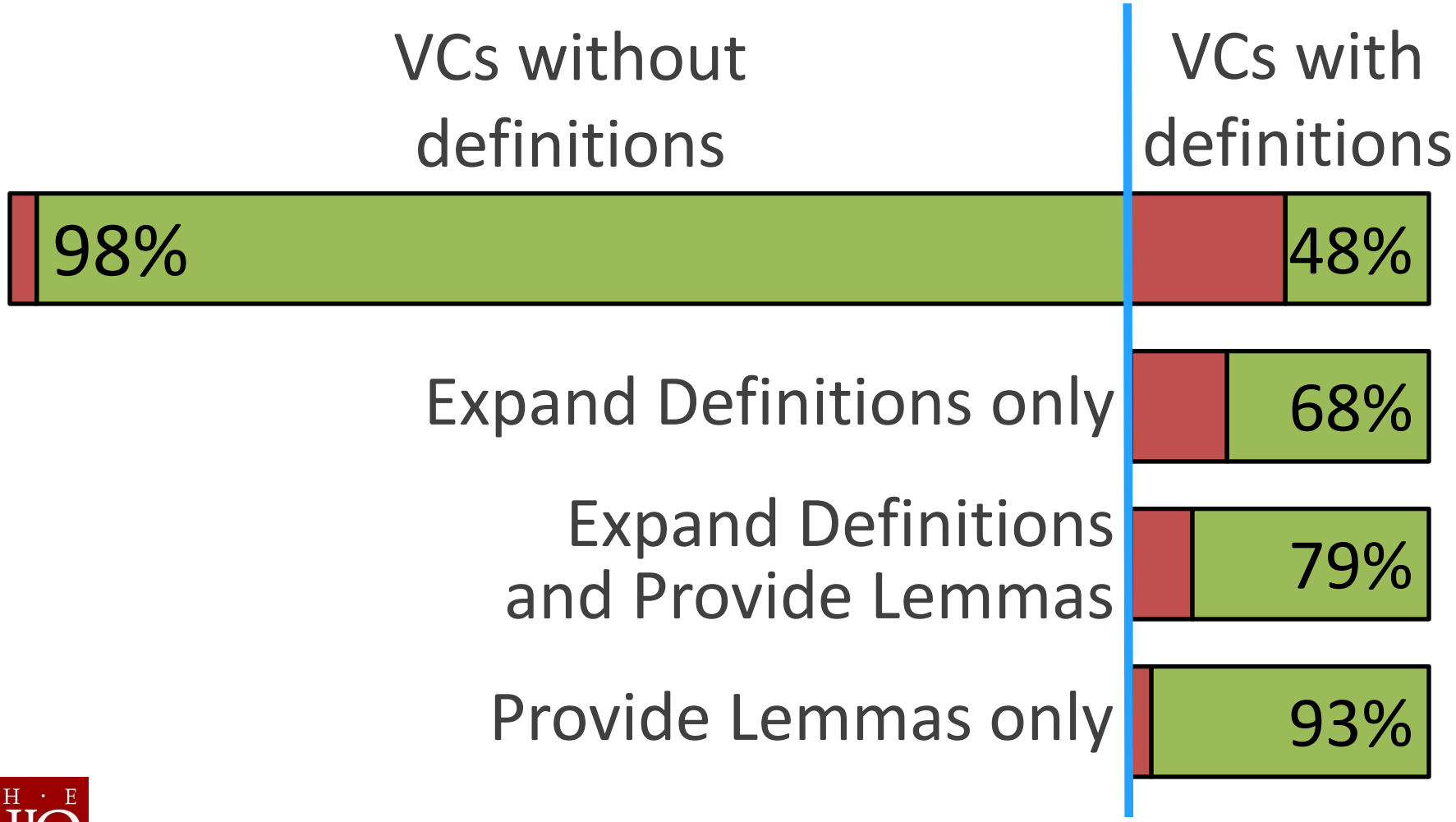
Fourth Attempt - Results

- About 93% of VCs with user-defined symbols are now proven

Results for VCs with definitions



Result of Proof Attempts



Conclusion

- First automated proof of a sorting algorithm where client supplies both:
  - Type to be sorted, *and*
  - Ordering (any total pre-order)

Conclusion

- Highly reusable universal algebraic lemmas support automated proofs of VCs with user-defined symbols

Conclusion

- Supplying universal algebraic lemmas about user defined mathematical functions and predicates is, in general, a better way than expanding definitions to support automated proofs of VCs

Conclusion

```
definition ARE_PERMUTATIONS (
    s1: string of Item,
    s2: string of Item
): boolean
is
    for all i: Item
        (OCCURS_COUNT(s1, i)=
         OCCURS_COUNT(s2, i))
```

Conclusion

- Benefits of universal algebraic lemmas
  - Better than expanding definitions

    Do not reintroduce the quantifiers and other complexities they were designed bury
  - Preserve encapsulation of definitions

    Benefits both humans (writers and readers of specifications) and provers

Future Work

- Study the potential for dependence of these results on specification and programming language features
  - Conjecture: basic conclusion applies to other programming and specification languages, other VC generation techniques, etc.

Questions?

Aditi Tagore, **Diego Zaccai**, Bruce W. Weide
The RESOLVE Software Research Group
Department of Computer Science and Engineering
The Ohio State University