Formal Reasoning Using an Iterative Approach with an Integrated Web IDE

Nabil M. Kabbani, Daniel Welch, Caleb Priester, Stephen Schaub, Blair Durkee, Yu-Shan Sun, and Murali Sitaraman

RESOLVE Software Research Group (RSRG)
Clemson University
http://www.cs.clemson.edu/group/resolve

We gratefully acknowledge NSF grants CCF-0811748, CCF-1161916, and DUE-1022941.
Iterative Reasoning Process

• User writes code and makes assertions about values of variables in the code
• Alternatively, user writes specifications and code to meet those specifications
• IDE reports if code and assertions match
• If matching fails, user rewrites assertions or code, and repeats process
• Classroom Experimentation
Iterative Reasoning Process

- User writes code and makes assertions about values of variables in the code
- Alternatively, user writes specifications and code to meet those specifications
- IDE reports if code and assertions match
  - What is behind the IDE?
  - What feedback do users get?
Verifying Compiler beneath the IDE

• The RESOLVE verifying compiler supports IDE actions
• The compiler is capable of automatically verifying correctness of practical programs
• Aimed at addressing Tony Hoare’s seminal verifying compiler ‘grand challenge’
  – A future where no software is considered complete until formally verified
RESOLVE Language Characteristics

• Support for both **formal specifications** and **implementation code** (separable interfaces)
  – **Formal specifications**: Syntactic slots for pre/post conditions, loop invariants, etc.
  – Separating specs and code for component reuse

• Support for **Clean Semantics**
  – Must be able to account for (and restrict) harmful referencing, aliasing, and mutation
  – Permit explicit pointer usage where unavoidable
RESOLVE Language Characteristics

• Allows for creation and use of new **mathematical theories**
  – Unlikely that *all* software can be adequately represented with only a handful of mathematical models
  – Though always continue to encourage reuse of existing ones when possible
Verification System Characteristics

• Differs from systems based on SMT solvers because there are no special solves; prover works well with new theories and combinations

• Differs from proof assistants because programmers are not involved in proof development
Iterative Reasoning Process

• User writes code and makes assertions about values of variables in the code
• Alternatively, user writes specifications and code to meet those specifications
• IDE reports if code and assertions match
  – What feedback do users get?
Demo

• User writes code and makes assertions about values of variables in the code (Demo 1)
• Alternatively, user writes specifications and code to meet those specifications (Demo 2)
• IDE reports if code and assertions match
  – What feedback do users get?
Iterative Reasoning Process

• User writes code and makes assertions about values of variables in the code
• Alternatively, user writes specifications and code to meet those specifications
• IDE reports if code and assertions match
• If matching fails, user rewrites assertions or code, and repeats process
  – Process should scale up for object-based software
Verifying Object-Based Realizations

- VCs (Verification Conditions) are necessary and sufficient conditions to prove implementation correctness

- VCs arise from multiple sources:
  - Ensures clause of an operation’s implementation under verification
  - Requires clause of a called operation
  - Establishing correctness of programmer-supplied assertions (e.g., loop and representation invariants)
  - Proof of termination
Demo 3: Specifying Invariants for Object-Based Software

• Specifications
• Implementations
• Loops and invariants
• Iterative invariant development
Iterative Reasoning Process

• User writes code and makes assertions about values of variables in the code
• Alternatively, user writes specifications and code to meet those specifications
• IDE reports if code and assertions match
• If matching fails, user rewrites assertions or code, and repeats process
• Classroom Experimentation
  – Software engineering class projects
Experimentation

- IDE used in multiple sections of a required third year software engineering course for majors
- Students develop verified software using the IDE and ideas discussed
- Examples of student work (Demo 4)
- Discussion of feature additions and improvements to the IDE
Formal Reasoning Using an Iterative Approach with an Integrated Web IDE

Nabil M. Kabbani, Daniel Welch, Caleb Priester, Stephen Schaub, Blair Durkee, Yu-Shan Sun, and Murali Sitaraman

RESOLVE Software Research Group (RSRG)
Clemson University
http://www.cs.clemson.edu/group/resolve

Additional Slides: RESOLVE Notation Details

We gratefully acknowledge NSF grants CCF-0811748, CCF-1161916, and DUE-1022941.
Specifying a Queue

• An ADT contract (or, Concept) in RESOLVE is where most specification occurs
  – Everything in a concept is a mathematical abstraction.
  – A mathematical conceptualization of the ADT or type we wish to implement

• Somewhat analogous to a Java interface with formal specifications.
Specifying a Queue

• Queue_Template is parameterized by:
  – A generic type ‘Entry’
  – A mathematical integer, ‘Max_Length’
  – ‘evaluates’ is a specification parameter mode
    • Indicates what effect the operation (or module) will have on a parameter
Specifying a Queue

• Uses notations defined in String_Theory
• The **Family** clause introduces Queues as a collection of abstract types modeled by Str(Entry), i.e., strings of entries

```plaintext
uses 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;
```
String_Theory

• Not built in; defined in a math unit
• Idea:
  – $\Sigma^*$ is strings over $\Sigma$
  – $\text{Str(Entry)}$ is strings over Entry
• Notations
  – Concatenation: $\alpha \circ \beta$
  – String containing a single entry: $<x>$
  – Length: $|\alpha|$
Specifying a Queue

• The **exemplar** introduces a sample Queue
• The **constraint** restricts which strings are valid Queues.
• The **initialization** ensures that initially a queue is guaranteed to be empty (i.e. a **constructor**)

```plaintext
uses String_Theory;
requires Max_LENGTH > 0;

Type Family Queue is modeled by Str(Entry);
exemplar Q;
constraint |Q| \leq Max_LENGTH;
initialization ensures Q = empty_string;
```
Specifying a Queue: Operation Enqueue

- Each **operation** carries pre/post conditions
  - **Requires/Ensures**: Strictly mathematical assertions. ‘Q’ refers to the mathematical value of a Queue
  - Ensures in English:
    - The resulting Queue is simply the incoming Queue, Q, concatenated with the incoming entry E.

```plaintext
Operation Enqueue(alters E: Entry; updates Q: Queue);
  requires |Q| < Max_Length;
  ensures Q = Q o <?E>;
```
Realizing a Queue

• Once a **concept** has been specified, an **realization** of that **concept** is needed

• Recall: **concepts** are implementation independent abstract specifications
Realizing a Queue:
Circular_Array_Realiz

- A record is the programmatic representation of the conceptual Queue (defined in the concept)
Realizing a Queue: Circular_Array_Realiz

• A **convention** imposes restrictions on our **record**-based Queue representation
  – Must hold after every operation

```plaintext
Realization Circular_Array_Realiz for Queue_Template;
Type Queue = Record
  Contents: Array 0..Max_Length - 1 of Entry;
  Front, Length: Integer;
end;
convention
  0 <= Q.Front < Max_Length and
  0 <= Q.Length <= Max_Length;
correspondence
  Conc.Q = (Concatenation i: Integer
            where Q.Front <= i <= Q.Front + Q.Length - 1,
            <Q.Contents(i mod Max_Length)>>);
```
Realizing a Queue: Circular_Array_Realiz

• The **Correspondence** clause relates internal representation values to their abstract values defined in the **concept**