Verified Software Components

Murali Sitaraman (Clemson)
Bruce W. Weide (Ohio State)

RESOLVE/Reusable Software Research Group
http://www.cs.clemson.edu/group/resolve
http://cse.osu.edu/rsrg

We gratefully acknowledge NSF grants
CCF-0811748, CCF-1161916, CCF-1162331,
CNS-0745846, DUE-0942542, DUE-1022191,
DUE-1022941, and DUE-1022191.
Overview

• Two 90-min sessions (separated by afternoon break):
  1. Formal Specification and “Push-Button” Verification
  2. Proof of Correctness of Data Representation
Overview

- Two 90-min sessions (separated by afternoon break):
  1. Formal Specification and "Push-Button" Verification
  2. Proof of Correctness of Data Representation

Session 2 includes time for discussion of audience-suggested research and education issues; so, think of some...
1. Formal Specification and “Push-Button” Verification

• Context: goals; verifying compilers
  – Activity

• Describing behavior of software: mathematical modeling; design-by-contract
  – Activity

• Iteration: loop invariants
  – Activity
Goal and Approaches

Ultimate goal: push-button verification of (at least) functional correctness.
Goal and Approaches

One approach: **lightweight tools** that can prove certain properties.
Goal and Approaches

Abstract models of some properties of program behavior: ESC/Java, model-checkers, ...
Goal and Approaches

Approach discussed here: **verifying compiler** as in Tony Hoare’s grand challenge.
Abstract models of full program behavior: Dafny, RESOLVE, ...
Verifying Compiler Overview

Program Specs and Code

Math Definitions and Theorems

Verification Condition Generator

Automated Prover

✔

✖
Verifying Compiler Overview

Program Specs and Code

Math Definitions and Theorems

Verification Condition Generator

Automated Prover

Verifying Compiler
Some FAQ Answers Up Front

• Language?
  – Needs to support writing both *formal specifications* and *code for implementations*
Some FAQ Answers Up Front

• Language?
  – Needs to support writing both *formal specifications* and *code for implementations*
  – Needs to support writing new *mathematical developments* (e.g., definitions for use in specification and results for use in verification)
Some FAQ Answers Up Front

• Language?
  – Needs to support writing both formal specifications and code for implementations
  – Needs to support writing new mathematical developments (e.g., definitions for use in specification and results for use in verification)
  – Needs to have rich and clean semantics
Some FAQ Answers Up Front

• Language?
  – Needs to support writing both formal specifications and code for implementations
  – Needs to support writing new mathematical developments (e.g., definitions for use in specification and results for use in verification)
  – Needs to have rich and clean semantics

Claim: RESOLVE meets all these requirements.
Some FAQ Answers Up Front

• Verification?
  – Needs to be *sound*
Some FAQ Answers Up Front

• Verification?
  – Needs to be *sound*
  – Needs to be *modular* (compositional, component-wise) in order to be scalable to realistic programs
Some FAQ Answers Up Front

• Verification?
  – Needs to be \textit{sound}
  – Needs to be \textit{modular} (compositional, component-wise) in order to be scalable to realistic programs
  – Needs to demand \textit{minimal justifications of correctness} from software developers
Some FAQ Answers Up Front

• Verification?
  – Needs to be *sound*
  – Needs to be *modular* (compositional, component-wise) in order to be scalable to realistic programs
  – Needs to demand *minimal justifications of correctness* from software developers

Claim: *RESOLVE* meets all these requirements.
Activity: Key Points Preview

• Code may involve *user-defined types* of the kind familiar in OO software
• Recursion presents no serious problems
• Main ideas involved in formal specification and verification are not particular to a given language
Given `Queue<T>` in Java ... 

- **void** `enqueue(T x)`
  - Adds `x` to the rear of `this`

- `T dequeue()`
  - Removes and returns the entry at the front of `this`

- **boolean** `isEmpty()`
  - Reports whether `this` is empty
... Implement This Method

- **void invert()**
  - Reverses the order of the entries in **this**
... Implement This Method

- **void** `invert()`
  - Reverses the order of the entries in this
Demo

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
**Globally_Bounded_Queue_Template** ➔
Enhancements ➔
**Inverting_Capability** ➔
Realizations ➔
**Recursive_Inverting_Realiz**
Tools: General Points

• Sound
• Incomplete (for many reasons)
  – Inadequate justification
  – Inadequate mathematical developments, which are work in progress
  – Web IDE time out
  – Backend provers are efforts in progress
  – Incompleteness in number theory
Demo

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
Globally_Bounded_Queue_Template ➔
Enhancements ➔
Inverting_Capability ➔
Realizations ➔
Recursive_Inverting_Realiz

Specification of a generic **Queue** interface
http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
Globally_Bounded_Queue
Enhancements ➔
Inverting_Capability ➔
Realizations ➔
Recursive_Inverting_Realiz

Specification of a Queue Invert operation
Demo

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
**Globally_Bounded_Queue_Template** ➔
Enhancements ➔
**Inverting_Capability** ➔
Realizations ➔
**Recursive_Inverting_Realiz**

Recursive implementation of
**Invert** operation
Activity: Key Points Review

• Code may involve *user-defined types* of the kind familiar in OO software
• Recursion presents no serious problems
• Main ideas involved in formal specification and verification are not particular to a given language
Inadequate Specifications

• *Imprecise*: `enqueue` enqueues item to the rear of the queue

• *Implementation-dependent*: `enqueue` does the same thing as `addElement` on a `vector`

• *Metaphorical*: `enqueue` has the same behavior as adding an item to the end of a line
Formal Specifications

• When we use variables of types such as `int` in programming, we think of their values as coming from mathematical domains such as $\mathbb{Z}$ (with constraints)

• Similarly, we need a mathematical domain to conceptualize values of variables of types such as `Queue`
Formal Specifications

• When we use integer operations in programming (e.g., +), we understand them in terms of their math counterparts on \( \mathbb{Z} \)

• Similarly, we will use terms appropriate from the chosen math domain to specify Queue operations
Queue Concept Specification

• Parameterized by an Entry type
• **Mathematical modeling**: Conceptualize Queue values as mathematical strings of entries with appropriate computational constraints
• Specifications of Queue operations use math string notations
Queue Concept Specification

- Parameterized by an Entry type
- **Mathematical modeling**: Conceptualize Queue values as mathematical *strings* of entries with appropriate computational constraints
- Specifications of Queue operations use math string notations

Other options: sets, multisets, functions, …
Queue Concept Specification

- Parameterized by an Entry type
- **Mathematical modeling**: Conceptualize Queue values as mathematical strings of entries with appropriate computational constraints
- Specifications of Queue operations use math string notations

a.k.a. “contract”
String Theory Notation

• General idea: $\Sigma^*$ is strings over $\Sigma$
  - $\text{Str}(\text{Entry})$ is strings over $\text{Entry}$

• Notations
  – Empty_String: $\Lambda$
  – Concatenation: $\alpha \circ \beta$
  – String containing a single entry: $<x>$
  – Length: $|\alpha|$
Design-by-Contract

• Articulated clearly in the 1980s (by Meyer)
• Design-by-contract has become the standard policy governing “separation of concerns” across modern software engineering
• Major difference for verification (as opposed to runtime assertion checking): mathematical modeling of program types
Specification of Operations

• Based on the mathematical model of a program type, such as `Queue`, each operation has:
  – A *requires clause* (*precondition*) that characterizes the responsibility of the client code that calls that operation
  – A *ensures clause* (*postcondition*) that characterizes the responsibility of the code that implements that operation
Specification Example: Key Points Preview

• This contract specification is *generic* (i.e., parameterized)
• Mathematical model for a Queue is a string, and operations are specified using the notations of string theory
• Parameters have *specification modes*
• Specifications are designed to avoid (needless, inefficient) copying
Specification Example

http://resolve.cs.clemson.edu/research

Components ➔

Concepts ➔

Globally_Bounded_Queue_Template
Specification Example: Key Points Review

- This contract specification is *generic* (i.e., parameterized)
- Mathematical model for a *Queue* is a string and operations are specified using the notations of string theory
- Parameters have *specification modes*
- Specifications are designed to avoid (needless, inefficient) copying
Connections to Verification

• From specification to verification:
  – A *requires clause (precondition)* that characterizes the responsibility of the client code that *calls* that operation
  – A *ensures clause (postcondition)* that characterizes the responsibility of the code that *implements* that operation
Connections to Verification

• From specification to verification:
  – A *requires clause (precondition)* that characterizes the responsibility of the client code that *calls* that operation
  – A *ensures clause (postcondition)* that characterizes the responsibility of the code that *implements* that operation

This responsibility results in generating *verification conditions (VCs)* for each call in client code.
Connections to Verification

• From specification to verification:
  – A *requires clause (precondition)* that characterizes the responsibility of the client code that *calls* that operation
  – A *ensures clause (postcondition)* that characterizes the responsibility of the code that *implements* that operation.

This responsibility results in generating VCs at the end of the code that implements an operation.
Verification Example: Key Points Preview

• Verification requires an integrated specification + implementation language

• Verification is *modular* (e.g., uses only specifications of called operations)

• Verification conditions (VCs) are raised not only for the end results, but also to make sure that intervening calls are legit
Verification Example

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
Globally_Bounded_Queue_Template ➔
Enhancements ➔
Inverting_Capability ➔
Realizations ➔
Recursive_Inverting_Realiz
Verification Example: Key Points Review

• Verification requires an integrated specification + implementation language
• Verification is *modular* (e.g., uses only specifications of called operations)
• Verification conditions (VCs) are raised not only for the end results, but also to make sure that intervening calls are legit
Activity:
Key Points Preview

• Even apparently simple code that’s “just about int\text{s}” can be subtle enough to benefit from automated verification

• Main ideas involved in formal specification and verification are not particular to a given language, e.g., RESOLVE
Clock Arithmetic

• The value of $a \mod b$, or $a \mod b$, where $a$ and $b$ are mathematical integers and $b > 0$, is computed by doing clock arithmetic on a clock face with $b$ positions labelled 0 through $b-1$. 
Clock Arithmetic

• The value of $a \mod b$, or $a \mod b$, where $a$ and $b$ are mathematical integers and $b > 0$, is computed by doing clock arithmetic on a clock face with $b$ positions labelled 0 through $b-1$
  – If $a > 0$, the “hand” on the clock starts at 0 and moves $|a|$ positions clockwise
  – Where it ends up is the value of $a \mod b$
Clock Arithmetic

• The value of $a \mod b$, or $a$ modulo $b$, where $a$ and $b$ are mathematical integers and $b > 0$, is computed by doing clock arithmetic on a clock face with $b$ positions labelled 0 through $b-1$.
  – If $a > 0$, the “hand” on the clock starts at 0 and moves $|a|$ positions clockwise.
  – If $a < 0$, it moves $|a|$ counter-clockwise.
  – Where it ends up is the value of $a \mod b$. 
Example: 24-hr Clock
Mod ≠ Remainder

• What is the remainder upon dividing 67 by 24?
Mod ≠ Remainder

• What is the \textit{remainder} upon dividing 67 by 24? It is 19.

• What is the \textit{remainder} upon dividing –67 by 24?
Mod ≠ Remainder

• What is the *remainder* upon dividing $67$ by $24$? It is $19$.

• What is the *remainder* upon dividing $-67$ by $24$? It is $-19$.
  
  – At least most people (and Java) say it is...

• What is $(-67) \mod 24$?
Mod ≠ Remainder

• What is the *remainder* upon dividing 67 by 24? It is 19.
• What is the *remainder* upon dividing –67 by 24? It is –19.
  – At least most people (and Java) say it is...
• What is \((-67) \mod 24\)? It is 5.
Given ...

• The \% operator in Java (which, despite sometimes being called the “mod” operator, actually computes the remainder of integer division), e.g.:
  \[
  67 \% 24 \text{ evaluates to 19}
  \]
  \[
  -67 \% 24 \text{ evaluates to -19}
  \]
... Implement This Method

- \textbf{int} \ clockMod(int \ a, \ int \ b)
- Contract specification:
  \textbf{requires} \ b \ > \ 0
  \textbf{ensures} \ clockMod = a \ mod \ b
... Implement This Method

- `int clockMod(int a, int b)`
- Contract specification:
  - requires \( b > 0 \)
  - ensures `clockMod = a \mod b`

You can do it in one concise line of Java code...
Demo

http://resolveonline.cse.ohio-state.edu

IntegerFacility ➔

ClockMod
Activity: Key Points Review

• Even apparently simple code that’s “just about ints” can be subtle enough to benefit from automated verification

• Main ideas involved in formal specification and verification are not particular to a given language, e.g., RESOLVE
Iteration and Loop Invariants

• Developer must sometimes write a justification of correctness of code
  – Assertion that the developer claims to be true and that will be checked to be true by the verifier (for soundness)
  – Proving and using such an assertion is easier for a mechanical verifier than inventing it
  – Writing it requires education and experience

• A loop invariant is one kind of justification
What Is a Loop Invariant?

• For a typical while loop, an assertion that is true at the beginning and at the end of each iteration, including the first and the last

• More generally: an invariant is a property that is true every time execution reaches a certain point—in the case of a loop invariant, the loop condition test
While Statement Control Flow

Code;
While B
  maintaining Inv;
do
  body;
end;
Confirm Q;

Confirm Q
Checking the Invariant: Part 1

Confirm Inv

While B maintaining Inv;
  do
    body;
  end;
Confirm Q;
Checking the Invariant: Part II

Code;
While B maintaining Inv;
  body;
end;
Confirm Inv;
Assume Inv

B
true
body

29 October 2013
Inductive Proof of the Invariant

Code;
While B maintaining Inv;
do body;
end;
Confirm Q;

Base Case VC: Confirm Inv

Inductive Assumption: Assume Inv

Inductive Case VC: Confirm Inv
Using the Invariant

Code;
While B
    maintaining Inv;
do
    body;
end;
Confirm Q;

Assume Inv

Confirm Q
Identifying a Suitable Invariant

- Pick an assertion such that the assertion and the negation of loop condition together imply the assertion to be confirmed after the loop (i.e., \textbf{not B and Inv} \Rightarrow Q)
- Make sure the assertion is indeed an invariant (see checking the invariant, parts I and II)!
Loop Invariant Example

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
Globally_Bounded_Stack_Template ➔
Enhancements ➔
Flipping_Capability ➔
Realizations ➔
Obvious_Flipping_Realiz
Identifying a Suitable Invariant

• Need to confirm (or prove) the *ensures* clause of Flip at the end of the code:

\[ S = \text{Reverse}(\#S) \]

• Need to confirm the following before the swap statement (\( S :=: \ Temp; \) )

\[ \text{Temp} = \text{Reverse}(\#S) \]

• Identify an invariant, Inv:

\[ S = \text{Empty}\_\text{String and Inv} \Rightarrow \text{Temp} = \text{Reverse}(\#S) \]
# Observing the Loop in Action

<table>
<thead>
<tr>
<th>After Iteration Number...</th>
<th>$S =$</th>
<th>$Temp =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$&lt; 10, 20, 30, 40 &gt;$</td>
<td>$&lt; &gt;$</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Observing the Loop in Action

<table>
<thead>
<tr>
<th>After Iteration Number...</th>
<th>$S =$</th>
<th>$Temp =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$&lt; 10, 20, 30, 40 &gt;$</td>
<td>$&lt; &gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$&lt; 20, 30, 40 &gt;$</td>
<td>$&lt; 10 &gt;$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Observing the Loop in Action

<table>
<thead>
<tr>
<th>After Iteration Number...</th>
<th>( S = )</th>
<th>( Temp = )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(&lt; 10, 20, 30, 40 &gt;)</td>
<td>(&lt; &gt;)</td>
</tr>
<tr>
<td>1</td>
<td>(&lt; 20, 30, 40 &gt;)</td>
<td>(&lt; 10 &gt;)</td>
</tr>
<tr>
<td>2</td>
<td>(&lt; 30, 40 &gt;)</td>
<td>(&lt; 20, 10 &gt;)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Observing the Loop in Action

<table>
<thead>
<tr>
<th>After Iteration Number...</th>
<th>( S = )</th>
<th>( Temp = )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(&lt; 10, 20, 30, 40 &gt;)</td>
<td>(&lt; &gt;)</td>
</tr>
<tr>
<td>1</td>
<td>(&lt; 20, 30, 40 &gt;)</td>
<td>(&lt; 10 &gt;)</td>
</tr>
<tr>
<td>2</td>
<td>(&lt; 30, 40 &gt;)</td>
<td>(&lt; 20, 10 &gt;)</td>
</tr>
<tr>
<td>3</td>
<td>(&lt; 40 &gt;)</td>
<td>(&lt; 30, 20, 10 &gt;)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Observing the Loop in Action

<table>
<thead>
<tr>
<th>After Iteration Number...</th>
<th>$S =$</th>
<th>$Temp =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$&lt; 10, 20, 30, 40 &gt;$</td>
<td>$&lt; &gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$&lt; 20, 30, 40 &gt;$</td>
<td>$&lt; 10 &gt;$</td>
</tr>
<tr>
<td>2</td>
<td>$&lt; 30, 40 &gt;$</td>
<td>$&lt; 20, 10 &gt;$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt; 40 &gt;$</td>
<td>$&lt; 30, 20, 10 &gt;$</td>
</tr>
<tr>
<td>4</td>
<td>$&lt; &gt;$</td>
<td>$&lt; 40, 30, 20, 10 &gt;$</td>
</tr>
</tbody>
</table>
What Doesn’t Change?

• Claim: the concatenation of the reverse of Temp, with S, is always the same: it is equal to the value of S before the first iteration

• Loop invariant (goal driven):

\[ \text{Reverse}(S) \circ \text{Temp} = \text{Reverse}(\#S) \]

• Alternative loop invariant (fact driven):

\[ \#S = \text{Reverse}(\text{Temp}) \circ S \]
Progress Metric for Termination

• Progress metric is a natural number (an ordinal, in general) that is decreasing
• Programmer specifies a progress metric for loop termination and the verifier checks its validity and uses it in its proof
• Suitable progress metric for example:
  
  \text{decreasing } |S|;

• Unsuitable one: \text{decreasing } |\text{Temp}|;
Activity:
Key Points Preview

- Ideas are language-independent
- Programmer supplies an adequate invariant and progress metric for termination to justify loop correctness
- Verifier checks the validity of the above (otherwise it won’t be sound)
- Verifier uses them to complete its proof of code correctness
Given…

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
Globally_Bounded_Queue_Template ➔
Enhancements ➔
Inverting_Capability ➔
Realizations ➔
Invariantless_Iterative_Realiz
... Do This

• Write invariants for loops
  – Write the invariant for the second loop first
Activity:
Key Points Review

• Invariants may involve mathematical models of multiple objects
  – Stacks and queues just happen to have the same string model

• No deep thinking is necessary to complete the proofs given the invariant
Another One?

http://resolve.cs.clemson.edu/research

Components ➔
Concepts ➔
Globally_Bounded_Queue_Template ➔
Enhancements ➔
Remove_Last_Capability ➔
Realizations ➔
Invariantless_Remove_Last_Realiz
Survey Questions

• Please briefly and anonymously answer the following questions on a sheet of paper and turn them in at the front table:

1. Which part of the first session did you find **most** interesting and/or helpful? If you have time: why?

2. Which part of the first session did you find **least** interesting and/or helpful? If you have time: why?
2. Proof of Correctness of Data Representation

- Correctness of data representation: representation invariants and abstraction relations
  - Philosophical discussion
  - Mathematical framework for thinking about it
  - Activity
So, What’s Inside the Computer?

• Consider any popular video game, e.g., Nintendo Wii bowling
• Are there bowling balls and bowling pins inside the game console’s computer?
So, What’s Inside the Computer?

• Consider any popular video game, e.g., Nintendo Wii bowling

• Are there bowling balls and bowling pins inside the game console’s computer?
  – Of course not!

• What’s really inside the computer, then, that makes bowling-like behavior?
So, What’s Inside the Computer?

• Consider any popular video game, e.g., Nintendo Wii bowling.
• Are there bowling balls and bowling pins inside the game console?
  – Of course not!
• What’s really inside the computer, then, that makes bowling-like behavior?

A thought experiment: What dynamic behavior would you see if you had a magical magnifying glass and could see inside the computer at any level of detail while it’s running?
A Possible Metaphor
A Tower of Abstractions

- Bowling pins?
- Vectors?
- Numbers?
- Bits?
- Voltages?
- Electrons?
- ????
A Tower of Abstractions

• Bowling pins?
• Vectors?
• Numbers?
• Bits?
• Voltages?
• Electrons?
• ???

These are all “just” mathematical models, i.e., abstractions used to explain and predict observable behavior.
A Tower of Abstractions

- Bowling pins?
- Vectors?
- Numbers?
- Bits?
- Voltages?
- Electrons?
- ???

Domain: Physics
These models are supposed to match physical reality, and are discarded if they do not; limited by the physical world.
A Tower of Abstractions

- Bowling pins?
- Vectors?
- Numbers?
- Bits?
- Voltages?
- Electrons?
- ???

Domain: *Computing*
These models are entirely artificial (need not match physical reality); limited only by the creativity of the software engineer.
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???

Numbers may be built on top of bits…
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???

Bits may be built on top of voltages…
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???

Voltages may be built on top of (?) electrons…
Interpretation of Representation

• Let’s not take the tower-building metaphor too far!

• A better approach is to think about interpreting a lower-level configuration (a.k.a. a representation) to get a higher-level value
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???

(Configurations of) bits may be interpreted as numbers...
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???

( Configurations of) voltages may be interpreted as bits…
A Tower of Abstractions

- Bowling pins
- Vectors
- Numbers
- Bits
- Voltages
- Electrons
- ???

(Configurations of) electrons may be interpreted as voltages…
Example

http://resolveonline.cse.ohio-state.edu

SetTemplate  ➔
QueueRealization
Two-Level Thinking

*client view*: mathematical model for type, contracts for operations

*implementer view*: data representation for type, algorithms for operations
Two-Level Thinking

This is the **abstract state space**: the set of all possible math model values as seen by a client.
Two-Level Thinking

This is the **concrete state space**: the set of all possible math model values of the data representation.
Two-Level Thinking

This is the interpretation of each concrete value (below) as an abstract value (above).
The Commutative Diagram

**before** an operation call

**after** an operation call
The Commutative Diagram

This is the specified behavior of the operation call (based on its contract).
This is the *actual* behavior of the operation call (based on its code).
What’s Left to Write Down?
The developer must **describe the concrete state space**, i.e., the legitimate values of the data representation: the **representation invariant**.
What’s Left to Write Down?

The developer must *describe the interpretation* for the legitimate values of the data representation: the *abstraction function*.
Justifications of Correctness

• Two key design decisions justify the correctness of a data representation:
  – The representation invariant: Which “configurations” of the data representation can ever arise, i.e., are considered legitimate?
  – The abstraction function: How is a legitimate data representation to be interpreted to get an abstract value?
Consequences

• With the representation invariant and abstraction function supplied by the software developer:
  – The code for each operation can be *written independently* of the others, and the operations will “play well together”
  – The code for each operation can be *verified independently* of the others
Activity:
Key Points Preview

• Each operation body, and its proof of correctness, can be done independently of all the others

• No pointers or references or nodes or links are involved in this representation of QueueTemplate because ListTemplate hides them
Activity

• Representing a Queue using a List
Demo

http://resolveonline.cse.ohio-state.edu

ListTemplate
The Type List

math subtype LIST_MODEL is
  (left: string of Item,
   right: string of Item)

type List is modeled by LIST_MODEL
  exemplar l
  initialization ensures
    l.left = empty_string and
    l.right = empty_string
How To Think About It

• An initial $\textbf{List}$ value:
  
  $(< >, < >)$

• A typical $\textbf{List}$ value:
  
  $(< \circ, \square, < \triangle, \diamond, \bigcirc, \triangle >)$
Insert

\[ s = \bigcirc \quad \bigtriangleup \quad \bigstar \quad \square \quad \bigstar \]

\[ x = \quad \bigotimes \]

\[ \text{Insert} (s, x) \]

\[ s = \bigcirc \quad \bigtriangleup \quad \bigstar \quad \square \quad \bigstar \quad \bigotimes \]

\[ x = \bigcirc \]
Remove

\[ s = \quad \bullet \quad \blacksquare \quad \triangle \quad \diamond \quad \bigcirc \quad \blacktriangle \]

\[ x = \quad \blacktriangle \quad \diamond \quad \bigcirc \quad \blacktriangle \]

\textit{Remove} (s, x)

\[ s = \quad \bullet \quad \blacksquare \quad \triangle \quad \diamond \quad \bigcirc \quad \blacktriangle \]

\[ x = \quad \blacktriangle \quad \diamond \quad \bigcirc \quad \blacktriangle \]
Advance

\[ S = \quad \bigcirc \quad \square \quad \bigcirc \quad \bigcirc \quad \bigcirc \quad \bigtriangleup \]

Advance \((s)\)

\[ S = \quad \bigcirc \quad \square \quad \bigcirc \quad \bigcirc \quad \bigcirc \quad \bigtriangleup \]
$s = \text{Reset}(s)$
$$s = \text{AdvanceToEnd}\ \text{(s)}$$
Activity and Demo

http://resolveonline.cse.ohio-state.edu

QueueTemplate ➔
ListRealization
Activity:
Key Points Review

• Each operation body, and its proof of correctness, can be done independently of all the others

• No pointers or references or nodes or links are involved in this representation of QueueTemplate because ListTemplate hides them
Discussion of Research and Education Issues

• Your suggestions?
Overview of Foundations and Tools

Empirical Studies


VC Generators and Provers

• Harton, H. K., Mechanical and Modular Verification Condition Generation For Object-Based Software, Ph.D. Dissertation, Clemson University, 2011.

Benchmarks


Challenge Problems


References and Linked Structures


Performance Specification and Verification


CS Education

• Numerous papers at educational conferences over the past 20 years including ACM SIGCSE, ACM ITiCSE, and CSEET

Survey Questions

• Please briefly and anonymously answer the following questions on a sheet of paper and turn them in at the front table:
  1. Which part of the second session did you find most interesting and/or helpful? If you have time: why?
  2. Which part of the second session did you find least interesting and/or helpful? If you have time: why?
For More Information...

• http://www.cs.clemson.edu/group/resolve
• murali@clemson.edu

• http://cse.osu.edu/rsrg
• weide.1@osu.edu