David R. Cok

CEA - LSL
and
Independent consultant

28 June 2018

LESSONS FROM VERIFYING LEGACY JAVA CODE APPLIED TO C++ SPECIFICATION & VERIFICATION

MAIN GOALS OF THIS TALK

- Describe high-level lessons from specifying and verifying several industrial (Java) libraries
- Summarize some outstanding technical/language feature issues from those verification projects
- Present language designs (of some features) for ACSL++, a specification language for C++ that builds on ACSL

ACKNOWLEDGEMENTS

- Long-term development of JML and OpenJML (open source - openjml.org)
  - aicas (equipment grant)
  - NSF CCF0916350 (Leavens & Singleton, at UCF)
  - NSF ACI-1314674 (Cok, at GrammaTech)
  - Amazon (Automated Reasoning Group)

- Specific verification projects
  - mostly under NDA
  - 1 publication with Amazon at VSTTE

- C++ Specification language
  - CEA
  - VESSEDIA (EC project) - vessedia.eu

ACKNOWLEDGEMENTS

- NSF Disclaimer: This material is based upon work supported by the National Science Foundation under the grants listed above. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

- Portions of this work were funded/supported by the VESSEDIA project, funded from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 731453.

The work described here is being published in:

- (accepted) VSTTE 2018: Practical Methods for Reasoning about Java 8’s Functional Programming Features (David R. Cok, Serdar Tasiran)

- (accepted) FTfJP 2018: Reasoning about Functional Programming in Java and C++ (David R. Cok)

- (accepted) FTfJP 2018: Specification Idioms from Industrial Experience (David R. Cok)

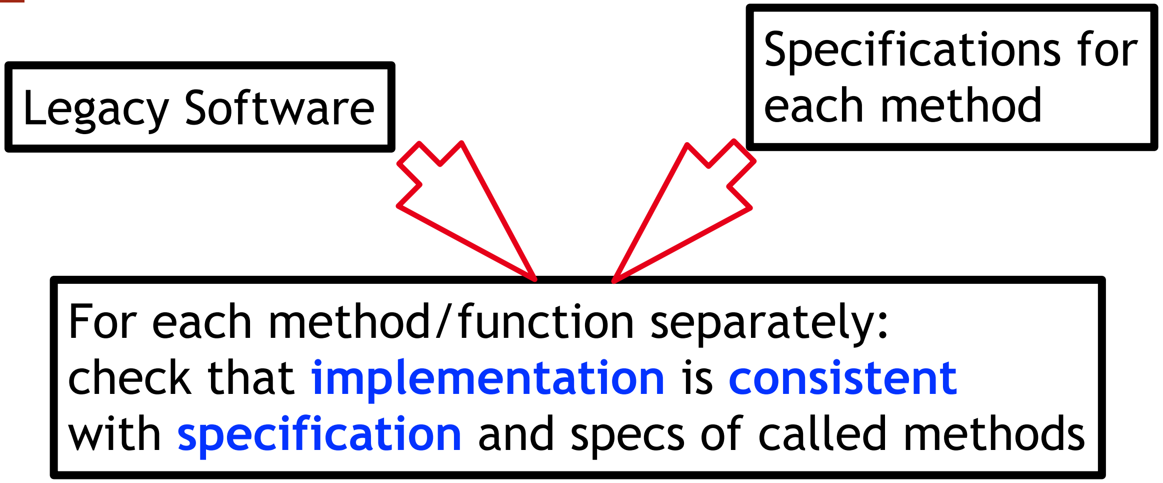
- (invited) ISOLA 2018: Java Automated Deductive Verification in Practice: Lessons from industrial proof-based projects (David R. Cok)

- (invited) ISOLA 2018: Java Verification: Static, Dynamic and In-between (David R. Cok)

- Complementary: ICSE 2018 poster: An Algorithm and Tool to Infer Practical Postconditions (Singleton, Leavens, Rajan,. Cok)

VERIFYING LEGACY JAVA SOFTWARE — HIGH-LEVEL OBSERVATIONS

VERIFICATION PARADIGM


Legacy Software

Specifications for each method

For each method/function separately:
check that **implementation** is **consistent**
with **specification** and specs of called methods

- **Modular** verification
- Requires (quite a bit of) **spec writing** [spec inference on its way]
- A lot of gain from partial work
  - **precluding runtime exceptions**
  - verifying **functional behavior** of critical (but not all) pieces
- But complete modular S&V gives confidence in usefulness and correctness of specs

VERIFICATION PARADIGM - OPENJML

For each method/function separately:
check that implementation is consistent
with specification and specs of called methods

Compiler parse, name resolution and type-checking

Translate into an IR, simplifying and including all assertions to be checked

Translate into a logical language (SMT-LIB)

SMT solver checks each verification condition

(Lots of room for engineering variation and optimization)
(Variation: use interactive provers (e.g. Coq, PVS) instead of automated (SMT))

CURRENT PROJECTS

- 3 current projects to S&V legacy software + 1 about to start
  - Java, some C
  - Java S&V performed using JML and the OpenJML tool

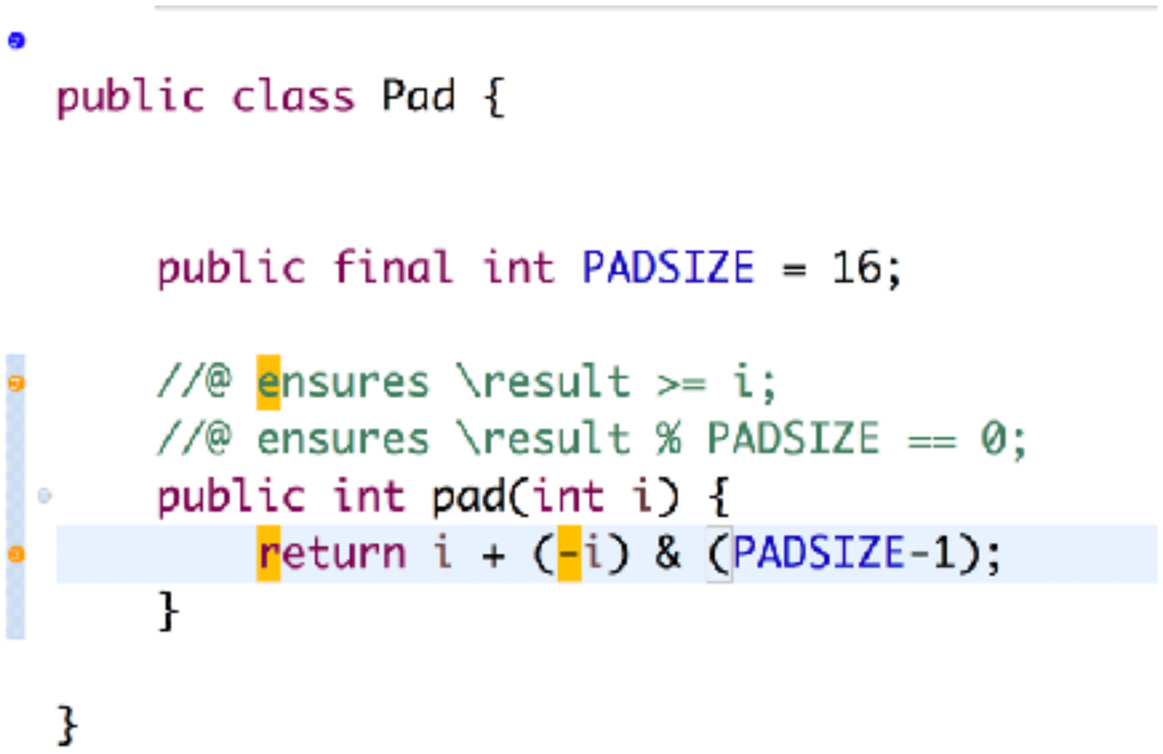
- Legacy:
  - code already written
  - heavily tested
  - specify and verify the code ***as is***

- Important to clients:
  - software is being used
  - software is being actively developed
  - software is safety- or security- or correctness- critical

- Industrial scale:
  - significant abstraction
  - significant amount of code (but manageable in ~10^1 p-m)
  - more struggles with scale, information hiding and data marshaling than intricate algorithms

DEMO - OPENJML

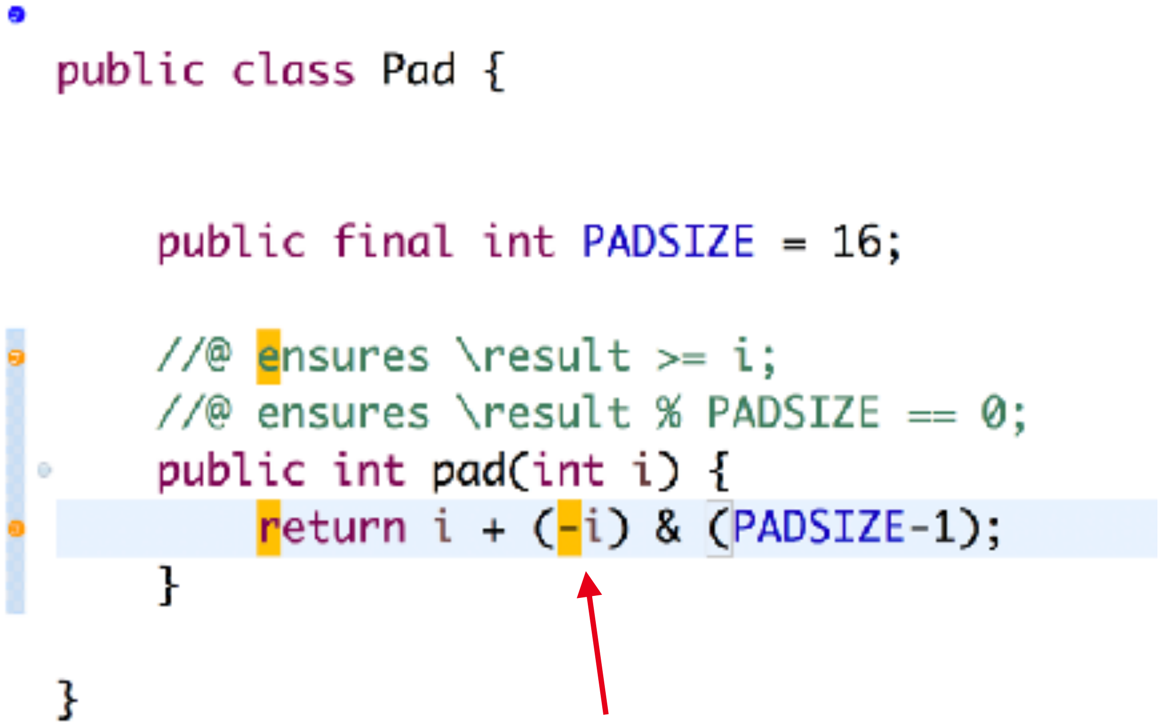
SCREENSHOT OF OPENJML DEMO



```
public class Pad {

    public final int PADSIZE = 16;

    //@ ensures \result >= i;
    //@ ensures \result % PADSIZE == 0;
    public int pad(int i) {
        return i + (-i) & (PADSIZE-1);
    }

}
```

Yellow markers are locations of errors.
Hovering over marker shows error message.
Hovering over variable/expression shows counterexample value

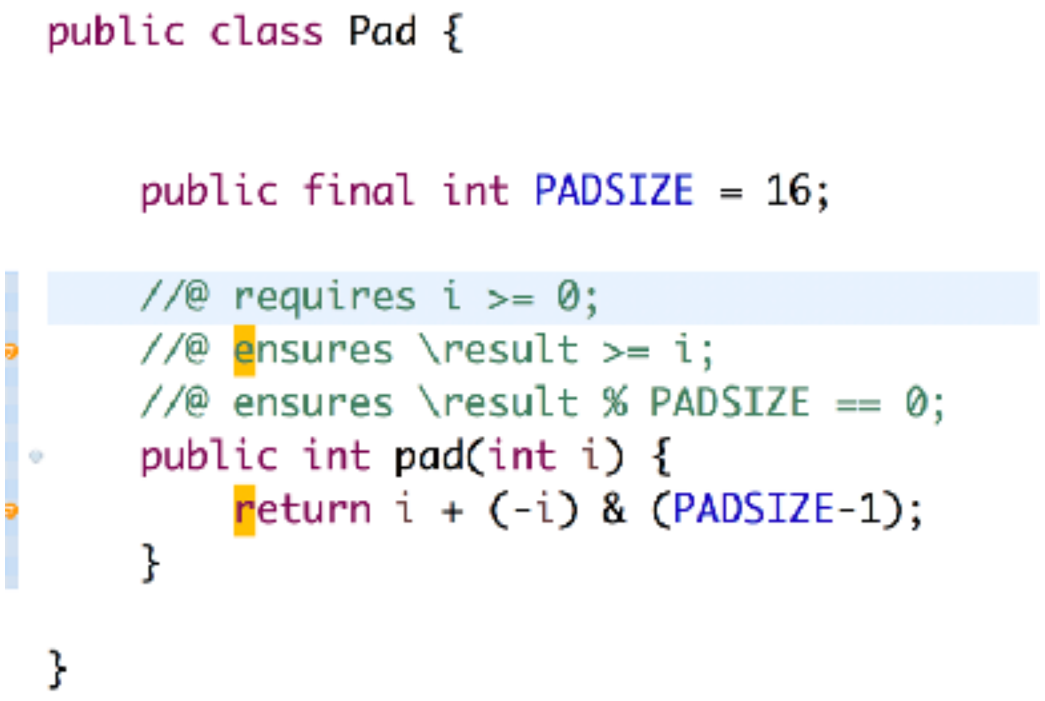
SCREENSHOT OF OPENJML DEMO



```
public class Pad {

    public final int PADSIZE = 16;

    //@ ensures \result >= i;
    //@ ensures \result % PADSIZE == 0;
    public int pad(int i) {
        return i + (-i) & (PADSIZE-1);
    }

}
```

Error is an arithmetic range error
i: Minimum int
-i: Minimum int
:: negation not allowed for the minimum int
So add a precondition

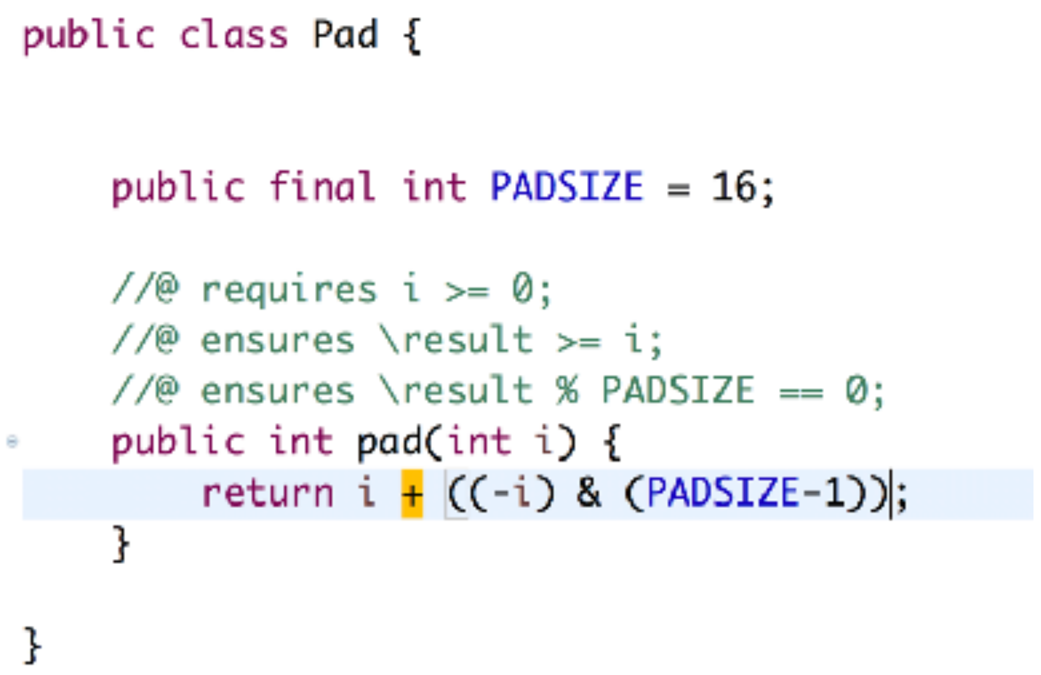
SCREENSHOT OF OPENJML DEMO



```
public class Pad {


    public final int PADSIZE = 16;

    //@ requires i >= 0;
    //@ ensures \result >= i;
    //@ ensures \result % PADSIZE == 0;
    public int pad(int i) {
        return i + (-i) & (PADSIZE-1);
    }

}
```

Precondition added — verification still fails
i: 1610612712
-i: -161061712
Hovering over + : i + (-i) is 0
:: Operator precedence is wrong
:: Add parentheses

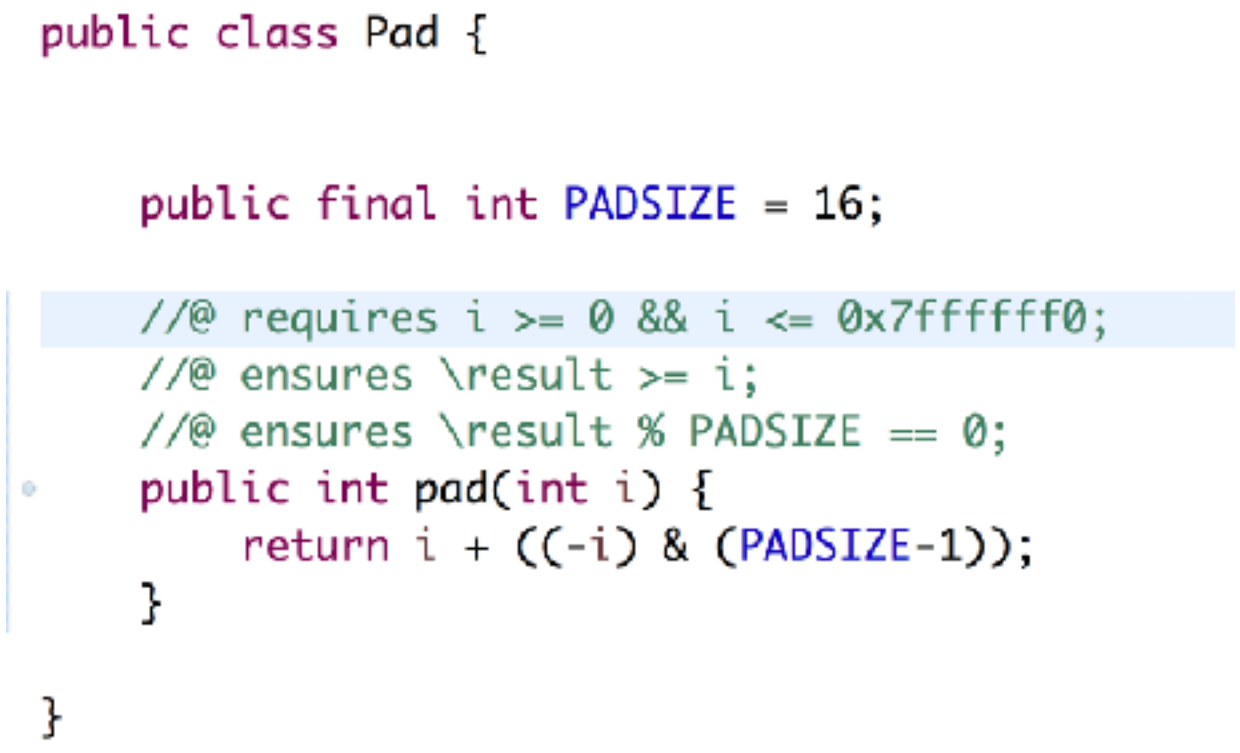
SCREENSHOT OF OPENJML DEMO



```
public class Pad {


    public final int PADSIZE = 16;

    //@ requires i >= 0;
    //@ ensures \result >= i;
    //@ ensures \result % PADSIZE == 0;
    public int pad(int i) {
        return i + ((-i) & (PADSIZE-1));
    }

}
```

Parentheses added — verification still fails
i: #x7ffffffc — close to but not max int
return expression - min integer
:: Padding a number within PADSIZE of the maximum integer causes an overflow
:: Add a precondition limiting i

SCREENSHOT OF OPENJML DEMO



```
public class Pad {


    public final int PADSIZE = 16;

    //@ requires i >= 0 && i <= 0x7ffffff0;
    //@ ensures \result >= i;
    //@ ensures \result % PADSIZE == 0;
    public int pad(int i) {
        return i + ((-i) & (PADSIZE-1));
    }

}
```

Added the additional precondition.
Now verification succeeds.

SCREENSHOT OF OPENJML DEMO

General point of OpenJML Demo:

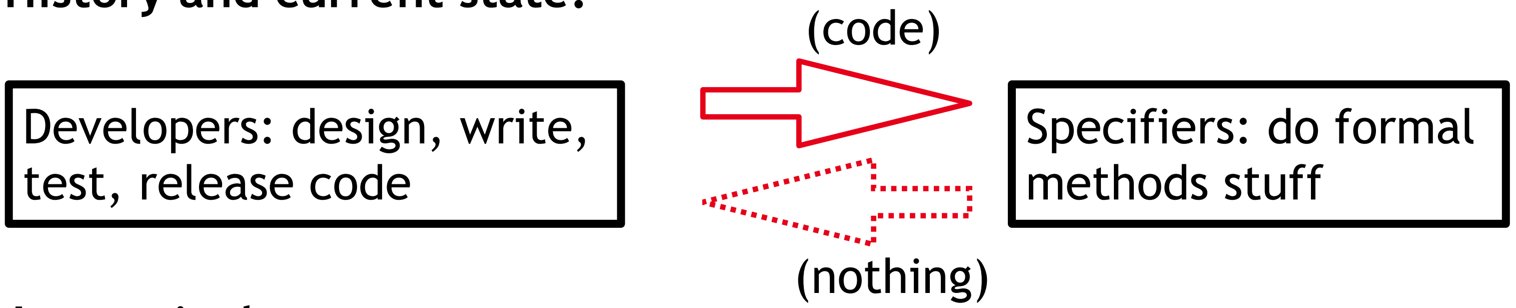
One spends much more time debugging specs/proofs than on successful proofs.

Demo showed one particular bit of functionality - the ability to explore a counterexample: variable values, expression values, also control flow and values in specifications.

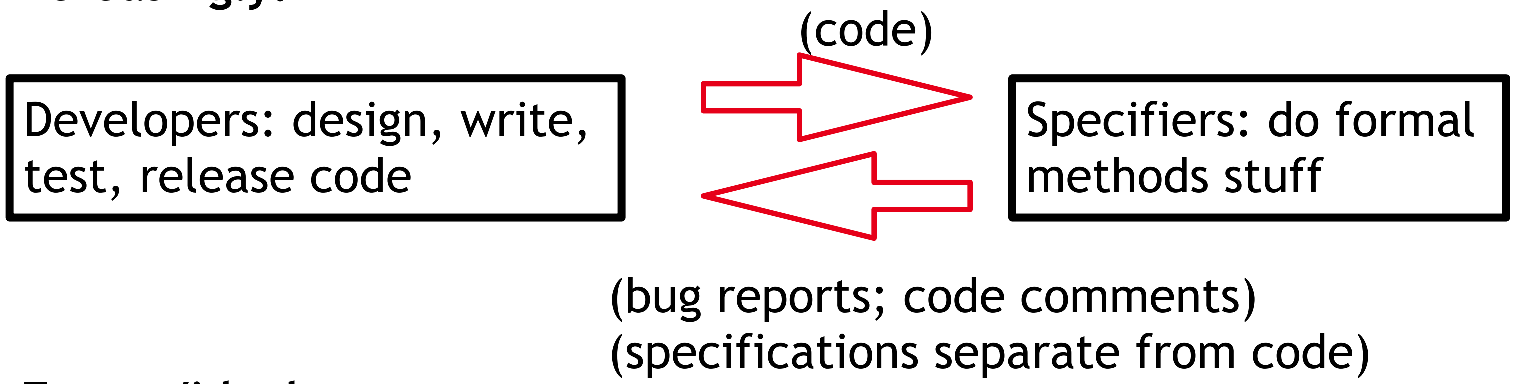
CHALLENGES

CHALLENGE 1: DEVELOPERS AND SPECIFIERS

History and current state:


(code)
Developers: design, write, test, release code
Specifiers: do formal methods stuff
(nothing)

Increasingly:


(code)
Developers: design, write, test, release code
Specifiers: do formal methods stuff
(bug reports; code comments)
(specifications separate from code)

Future/ideal:

Integrated team (perhaps with specializations)
- formal specifications closely associated with code
- verification checks part of continuous integration

CHALLENGE 1: DEVELOPERS AND SPECIFIERS

- **Building trust**
- **Demonstrating value**
- **Tools that are robust**
- **Tools that non-experts can use**
- **Tools that scale to “real” code**
- **Target software that is worth the effort**
- **Specifications that add value to the code, not just clutter it**
  (readable, readily understandable, adds insight, is not duplicative)
  e.g. specifications can be more verbose than the code itself

CHALLENGE 2: SCALE

Projects so far:

- 10^4-10^5 of LOC, about as many lines of specs
- 10^3 methods
- 10^{1+} hours of verification time; $10^{0\text{-}4}$ sec/method
- average of 10^4 lines of SMT/method
- $10^{5\text{-}6}$ individual assertions

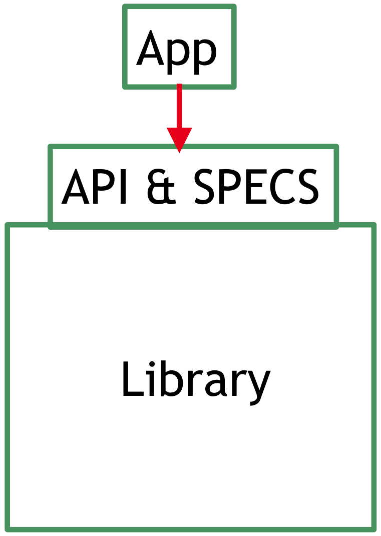
Engineering optimization needed
(along with theoretical focus on soundness and expressiveness)

- preprocessing? (or does the SMT solver do this best?)
- design of SMT translation for optimized solver execution
- breadth of SMT solver capability
- handling quantifiers (cf. Leino and Pit-Claudel)
- defaults and inference (cf. Singleton et al.)
- wisely tracking and using dependencies

CHALLENGE 3: LIBRARY SPECIFICATIONS

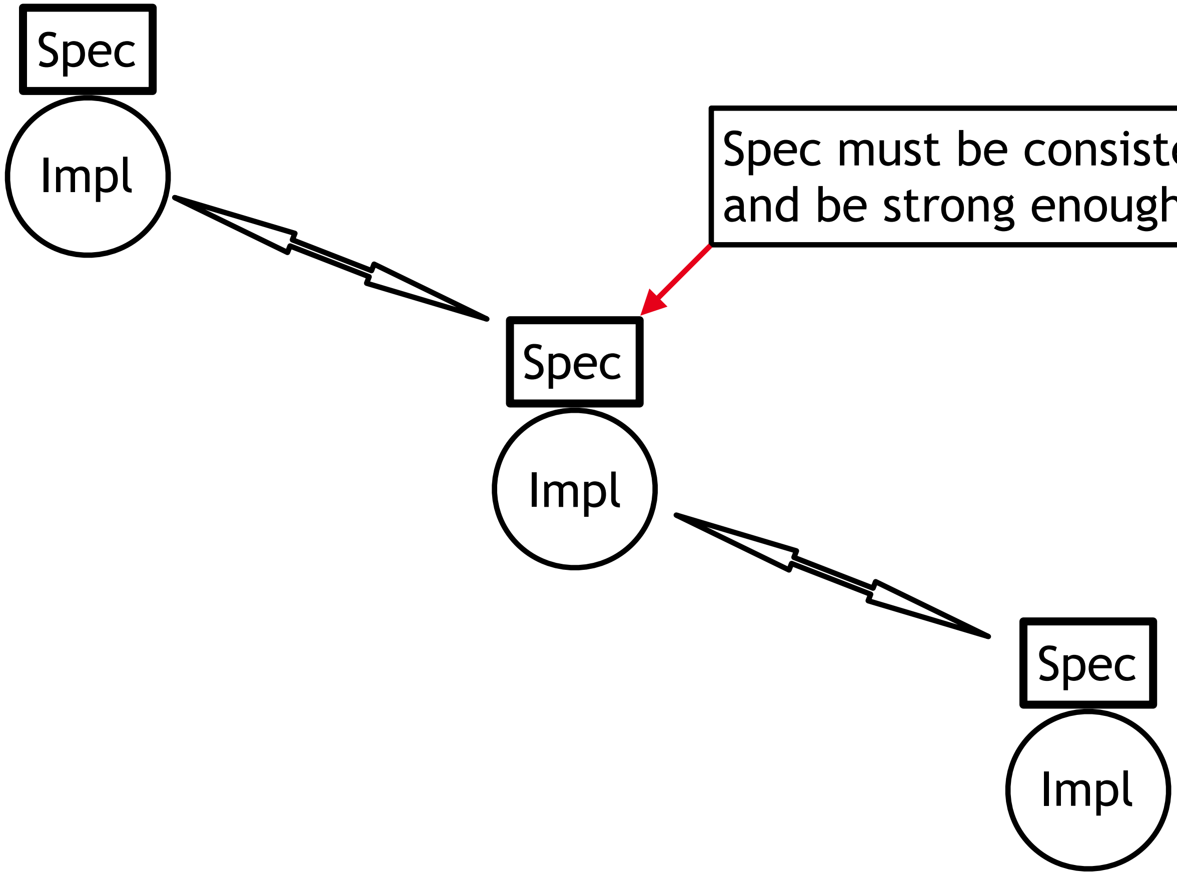
Modern programming languages rely on system libraries to provide many capabilities.

Correspondingly, verification environments need specifications of those libraries.



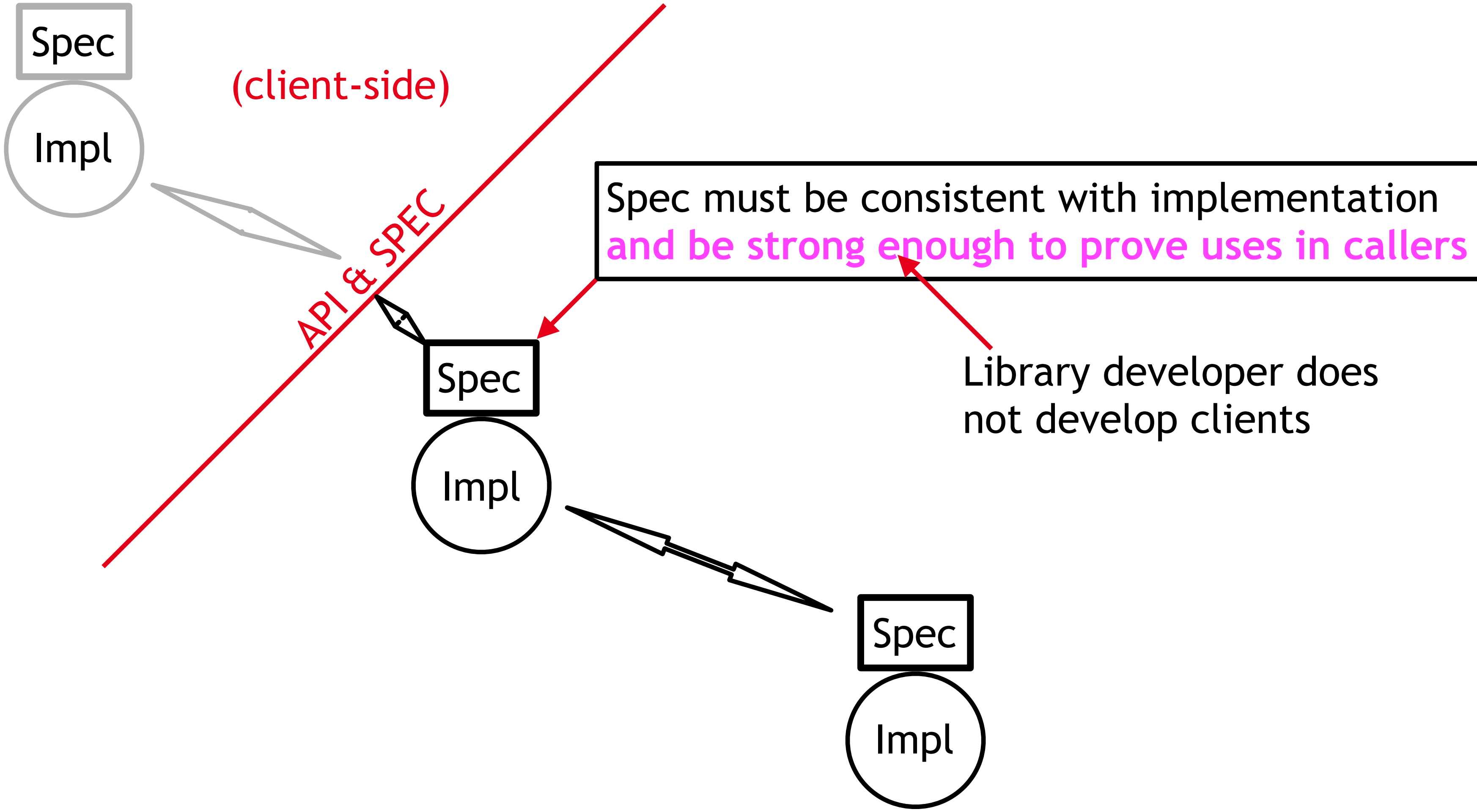
- Specs need to be **written**
- Specs need to be **verified** against library implementations
- Will **one set** of specifications be useful for all purposes?
  - avoiding runtime exceptions vs. verifying functional behavior
  - predominantly bit-vector vs. mathematical arithmetic
  - runtime verification vs. static deductive verification
  - the same set of specs for different tools?

CHALLENGE 4: VERIFYING LIBRARIES (as the software deliverable)



Spec must be consistent with implementation and be strong enough to prove uses in callers

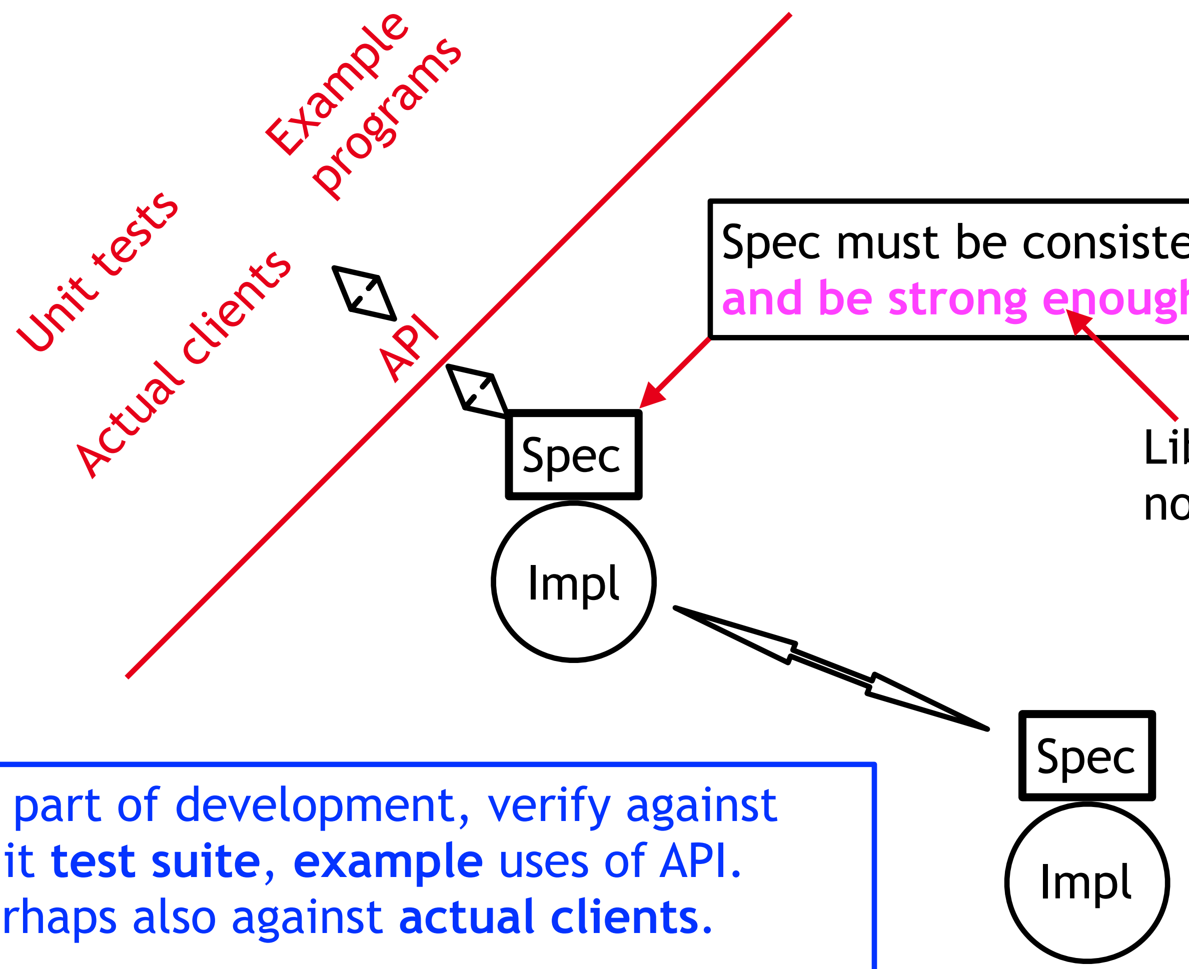
CHALLENGE 4: VERIFYING LIBRARIES



Spec must be consistent with implementation
and be strong enough to prove uses in callers

Library developer does
not develop clients

CHALLENGE 4: VERIFYING LIBRARIES



Spec must be consistent with implementation
and be strong enough to prove uses in callers

Library developer does not develop clients

As part of development, verify against unit **test suite**, **example** uses of API. Perhaps also against **actual clients.**

Need to have tools to check for **coverage** of such API specification testing.

CHALLENGE 5: CONTINUOUS INTEGRATION

As software evolves, specifications and proofs must evolve

Make the verification part of Continuous Integration, along with dynamic unit testing.

Include checks for coverage

It's not hard: just do it (though the verification can be time-consuming)

It would help to have
- ability to replay (instead of re-find) SMT-based proofs
- dependency checking tools to minimize the re-verification needed for a given change
- faster proof tools

CHALLENGE 6: SPECIFICATION QUALITY

Dynamic testing has measures of test coverage

What is the equivalent for specification quality?

CHALLENGE 7: VERIFYING SECURITY

$$\begin{tikzcd}
\boxed{\text{Verifying functional specs}} \arrow[r] & \boxed{\text{Program does what the specs say}} \\
\boxed{\text{Verifying security}} \arrow[r] & \boxed{\text{Program will not do anything else}}
\end{tikzcd}$$

We can specify and prove specific security properties

How do we know which properties we are missing?

CHALLENGE 8: SPECIFICATION LANGUAGE EXPRESSIVENESS & UNDERSTANDABILITY

(Not just technical language feature expressiveness)

Verification tools prove specification and implementation are mutually consistent, not necessarily correct.

Do the specs match human expectations of program behavior?

Specifications must be

concise enough

understandable enough

that non-expert human review can be confident of reasonable completeness and correctness.

Move away from traditional logic-based languages?

DSLs?

Table-based specs?

CHALLENGES SUMMARY

- Developer trust
- Scale
- Library specifications
- Library verification
- Continuous integration
- Quality and completeness of specifications
- Verifying security
- Specification language expressiveness

SPECIFICATION LANGUAGE FEATURES

SPECIFICATION CHALLENGES

(from Leavens, Leino, Müller, Specification and Verification Challenges… 2006)

- Mathematical modeling types
- Reasoning about quantifiers and comprehensions
- Method calls in specifications
  - Frame properties in callbacks [now add: functional programs]
- Specifying effects on static fields
- (Java) lazy class initialization
- Invariants of complex data structures
- Finalizers
- Specifying clients of function objects
- Specifying function objects
- Specifying libraries
- Specification for multiple tools

SPECIFICATION CHALLENGES

(from Leavens, Leino, Müller, Specification and Verification Challenges… 2006)

- Mathematical modeling types
- Reasoning about quantifiers and comprehensions
- Method calls in specifications
  - Frame properties in callbacks [now add: functional programs]
- Specifying effects on static fields
- (Java) lazy class initialization
- Invariants of complex data structures
- Finalizers
- Specifying clients of function objects
- Specifying function objects
- Specifying libraries
- Specification for multiple tools

All of these are still challenges in practical application

ADDITIONAL SPECIFICATION CHALLENGES

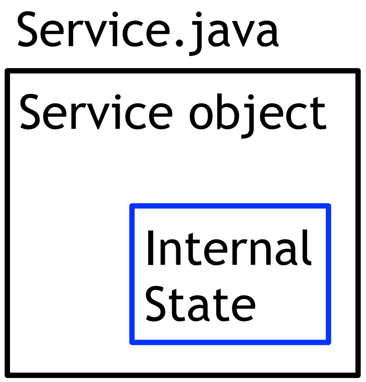
- Abstraction and refinement
- Management of invariants
- Hidden state and observational purity
- Usability: understanding and debugging proof failures
- Concurrency

ABSTRACTION AND REFINEMENT

- Crucial to modeling or developing large-scale systems
- Not much of an issue in verifying specific algorithms

OBSERVATIONAL PURITY

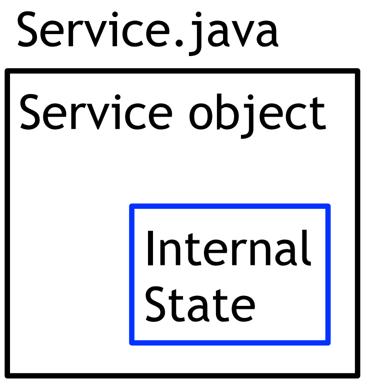
OBSERVATIONAL PURITY
- HIDE IMPLEMENTATION DETAILS WITH DATAGROUPS



```
//@ assigns state;
void set(Object o) { … }
```

```
//@ model public JMLDataGroup  state;

private int size; //@ in state;
```

OBSERVATIONAL PURITY - HIDE IMPLEMENTATION DETAILS WITH DATAGROUPS



```
//@ assigns state;
void set(Object o) { … }
```

```
//@ assigns s.state;
void foo(Service s) {
    s.set(…);
}
```

```
//@ model public JMLDataGroup  state;

private int size; //@ in state;
```

But frame conditions keep bubbling out to transitive callers even ones that need not know that Service is even being used.

Need to have abstraction layers of frame conditions

OBSERVATIONAL PURITY - IGNORED STATE

In some cases the internal state does not affect the rest of the program:

- A cache
- Log output

Can we omit changes to some internal state from frame conditions?

OBSERVATIONAL PURITY
- IGNORED STATE: CACHE

```
class Cache {

public:
    //@ model JMLDataGroup state;

private:
    boolean isCached = false; //@ in state;
    int value; //@ in state;

public:

   //@ assigns state;
   int get() {
     if (!isCached) {
        value = computeValue();
        isCached = true;
     }
     return value;
   }

}
```

OBSERVATIONAL PURITY
- IGNORED STATE: CACHE

```
class Cache {

public:
    //@ model JMLDataGroup state;

private:
    boolean isCached = false; //@ in state;
    int value; //@ in state;

public:

   //@ assigns ~~state~~ \nothing;
   int get() {
     if (!isCached) {
        value = computeValue();
        isCached = true;
     }
     return value;
   }
}
```

OBSERVATIONAL PURITY
- IGNORED STATE: CACHE

```
class Cache {

public:
     //@ model JMLDataGroup state;

private:
     boolean isCached = false; //@ in state;
     int value; //@ in state;

public:
    //@ assigns ~~state~~ \nothing;
    int get() {
      if (!isCached) {
         value = computeValue();
         isCached = true;
      }
      return value;
    }

    //@ pure
    boolean isCached() {
      return isCached;
    }
}
```

OBSERVATIONAL PURITY - IGNORED STATE: CACHE

```
class Cache {

public:
     //@ model JMLDataGroup state;

private:
     boolean isCached = false; //@ in state;
     int value; //@ in state;

public:
    //@ assigns ~~state~~ \nothing;
    int get() {
      if (!isCached) {
         value = computeValue();
         isCached = true;
      }
      return value;
    }

    //@ pure
    boolean isCached() {
      return isCached;
    }
}
```

Observes internal state

OBSERVATIONAL PURITY - IGNORED STATE: CACHE

```
class Cache {

public:
     //@ model JMLDataGroup state;

private:
     boolean isCached = false; //@ in state;
     int value; //@ in state;

public:
    //@ assigns ~~state~~ \nothing;
    int get() {
      if (!isCached) {
         value = computeValue();
         isCached = true;
      }
      return value;
    }

    //@ pure
    boolean isCached() {
      return isCached;
    }
}
```

```
isCached();
get();
isCached();
```

If **get()** does not modify program state then the two **isCached()** calls must return the same value.

If state is observed, it may not be ignored!

[Proof obligations are similar to assuring information flow properties]

Observes internal state

ACSL++

ACSL++ GOALS

- ACSL++: A specification language for C++ programs
  - Part of VESSEDIA — vessedia.eu
- Create a design document for ACSL++
  - Build on ACSL - http://frama-c.com
  - Build on STANCE - http://www.stance-project.eu
  - Leverage experience of other specification languages, O-O and otherwise, e.g., JML, Spec#, Dafny, SPARK, …
  - Also leverage experience with industrial scale projects
- Then, implement, as possible and time permits
  - Expansion of Frama-C's frama-clang plug-in: C++ -> C -> Frama-C

C++ ISSUES

- Simple items
    - Namespaces
    - Classes (aggregates)
    - Templates
    - Exceptions
    - Default values of formal parameters
    - Attributes
    - Enums
    - Defensive programming
    - Pure functions
    - Types
    - Invariants
    - Conversions and casts, implicit and explicit
- New concepts
    - Inheritance and abstraction, access control
    - Changes to hidden state
    - Functional programming
- Misc
    - Access control

TEMPLATES

C++ TEMPLATES

- C++'s generic programming mechanism: here no operations required of T other than copying

```
template <class T> class Stack {                      Stack<int> s;
    Stack<T> push(const T& item);
    void pop();
    T& top();
}
```

- But often there are implicit comparison or arithmetic or other operations (could even require specific methods)

```
template <class T> class List {
    void sort();      <————————————— Needs comparison
    double average(); <——————————— Needs arithmetic
}
```

- C++ has no means (other than documentation or shared knowledge) to specify what operations are needed.

- Type problems are not discovered until **compilation** if a template is **instantiated** with inappropriate types.

- Similarly: no place to put **specifications** of operations needed for template parameters.
  So one can't reason about templates apart from specific instantiations.

C++ CONCEPTS

- ***Concepts*** is a proposal (perhaps for C++20) to include a constraint language for templates.
  An example taken from the (very draft) proposal

```
template <class T>
 concept C = requires(T a, T b, const T c, const T d) {
        c == d; // #1
        a = std::move(b); // #2
        a = c; // #3
};
```

- The requires construct lists syntax that elements of a type must satisfy for the type to be considered to adhere to the concept C

- Another example: type T and U that can be compared. == and != must produce results convertible to Boolean.
  These don't explicitly say
  - t == u and u == t give the same result
  - t == u and t != u give opposite boolean results

  (though there are proposals for some portion of such functionality.)

```
template <class T, class U>
concept __WeaklyEqualityComparableWith =
   requires(const remove_reference_t<T>& t,
        const remove_reference_t<U>& u) {
    t == u; requires Boolean<decltype(t == u)>;
    t != u; requires Boolean<decltype(t != u)>;
   u == t; requires Boolean<decltype(u == t)>;
   u != t; requires Boolean<decltype(u != t)>;
};
```

C++ CONCEPTS
WHERE TO PUT THE SPECIFICATIONS?

- (Early) Draft design
  - piggyback on the Concepts idea
  - include (in ACSL++ annotations) declarations with conventional method specifications
  - If there are generic specifications and specifications in a specialization:
    both sets of specs apply, as if they were additional behaviors

```
template <class T>
concept EqualityComparable =
   requires(const remove_reference_t<T>& t,
         const remove_reference_t<T>& u) {
    t == u; requires Boolean<decltype(t == u)>;
    t != u; requires Boolean<decltype(t != u)>;
   u == t; requires Boolean<decltype(u == t)>;
   u != t; requires Boolean<decltype(u != t)>;

 /*@ behavior neq
    @     requires \true;
    @     ensures (t!= u) == !(t == u);
    @     throws {...} \false;
    @ boolean operator!=(const T& t, const T& u);
    @*/
};
```

C++ CONCEPTS
VERIFICATION CONDITIONS

- When writing a template
  - use concept names to characterize template arguments
  - implementations of template functions can be verified using the specs of methods from the concept

- When using (instantiating) a template
  - The actual template argument must obey the declared concept of the formal template argument

FUNCTIONAL PROGRAMMING IN C++ (and Java)

FUNCTIONAL PROGRAMMING IN C++

- Lambda expressions

  ```
  auto addone = [] (int i ) { return i+1; }
  ```

  effectively a class with an operator() method

- Implicit iteration (combinators)

  ```
  transform(v.begin(), v.end(), v.begin(), addone);
  ```

- Computing (composing) function objects - properties of output function depend on input functions

  ```
  auto fg = compose(f, g);
  ```

FUNCTIONAL PROGRAMMING IN C++
LAMBDA EXPRESSIONS

```
        auto addone = [] (int i ) { return i+1; }
```

- Properties of a function object:
    - Pre/frame/post/footprint conditions
    - Define a specification type that holds those pre/frame/post conditions

```
/*@ class Increment {
        requires i < INT_MAX;
        assigns \nothing;
        ensures \result == (i+1);
        int operator() (int i);
} @*/
```

```
        auto /*@{ Increment }@*/ addone = [] (int i ) { return i+1; }
```

- Clients of *addone* can see *addone*'s specification in Increment.

FUNCTIONAL PROGRAMMING IN C++
INLINING

```
set(Supplier<T> s) {
   this.x = s.get();
}


set( () -> t )
```

Small function; source is available

Actual argument is a literal

FUNCTIONAL PROGRAMMING IN C++
INLINING

```
set(Supplier<T> s) {
   this.x = s.get();
}
```

Small function; source is available

Actual argument is a literal

```
set( () -> t )
```

Inline substitution and evaluation:

```
this.x = t;
```

FUNCTIONAL PROGRAMMING IN C++
INLINING

```
set(Supplier<T> s) {
   this.x = s.get();
}
```

Small function; source is available

Actual argument is a literal

```
set( () -> t )
```

Inline substitution and evaluation:

```
this.x = t;
```

Requires source of called methods
Breaks modularity

FUNCTIONAL PROGRAMMING IN C++
MODEL METHODS

Instead of the source itself, specify a method using a ‘model method’ (abstraction/summary of the method’s implementation)

```
//@ behavior { this.x = s.get(); }
set(Supplier<T> s);
```

- Abstraction/summary of the method’s body
- Sometimes is a duplicate of it
- Preserves modularity

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

```
vector<int> v;
vector<int> w;
transform(v.begin(),v.end(),w.begin(),addone);
```

- Non-functional

```
vector<int> v;
/*@  loop_invariant 0 <= i && i <= v.size();
     loop_invariant  (\forall int j; (0 <= j && j < i) ==> w[j] == v[j] + 1);
 @*/
for (int i = 0; i < v.size(); ++i)  {
  w[i] = addone(v[i]);
}
```

If the function argument is a function literal (such as a lambda expression), then it can be inlined along with a loop implementing *transform* and analyzed as a traditional loop.

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

```
vector<int> v;
vector<int> w;
transform(v.begin(),v.end(),w.begin(),addone);
```

- Non-functional

```
vector<int> v;
/*@  loop_invariant 0 <= i && i <= v.size();
     loop_invariant  (\forall int j; (0 <= j && j < i) ==> w[j] == v[j] + 1);
 @*/
for (int i = 0; i < v.size(); ++i)  {
   w[i] = ((v[i]) + 1);
}
```

If the function argument is a function literal (such as a lambda expression), then it can be inlined along with a loop implementing *transform* and analyzed as a traditional loop.

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

```
vector<int> v;
vector<int> w;
transform(v.begin(),v.end(),w.begin(),addone);
```

- Non-functional

```
vector<int> v;
vector<int> w;
auto iter = v.begin();
auto out = w.begin();
/*@  loop_invariant 0 <= \count && \count <= std::distance(v.begin(),v.end());
     loop_invariant  (\forall int j; (0 <= j && j < \count) ==> w[j] == v[j] + 1);
     loop_invariant  (\forall int j; (0 <= j && j < \count) ==> *(w.begin()+j) == 1 + *(v.begin()+j));
 @*/
while (iter != v.end()) {
  *out = addone(*iter);
   ++iter; ++out;
}
```

These specs combine user information:
    the effect of addone
and library information:
    the structure of the loop

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

- there is no loop in user code to which to attach loop invariants
- the loop in the library code is separated from any details of user code

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

```
vector<int> v;
vector<int> w;
transform(v.begin(),v.end(),w.begin(),addone);
```

Possibly:

```
/*@ ensures \forall int i; 0 <= i && i < rend-rbegin ==>
                                        *(wbegin+i) == f( *(rbegin+i) );
@*/
transform(rbegin,rend,wbegin,f)
```

But this only works if **f** is a **nicely pure function**, without other dependence (and besides it is not a logic function)

Instead, we need to be able to combine
- a representation of the combinator's actions
- and client information about the effect of the argument

That is - the spec of **transform** needs information about the specification of **f**

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

In java.util.stream.Stream:

```
  /*@ public normal_behavior
    @   requires true;
    @   {
    @     //@ loop_invariant i == \count && 0 <= i && i <= _length;
    @     //@ decreases this._length - i;
    @     for (int i=0; i < this._length; i++) {
    @        consumer.accept(this.values[i]);
    @     }
    @   }
    @*/
   void forEachOrdered(java.util.function.Consumer<? super T> consumer);
```

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

```
In client code:

public class Test {

      …

      public void foo() {

            …

            ii = 0;
            //@ loop_invariant Test.ii == \count;
            //@ loop_invariant (\forall int j; j>=0 && j<\count; arr[j] == st.value[j]);
            //@ loop_modifies Test.ii, Test.arr[*];
            //@ inlined_loop;
            st.forEachOrdered(v -> putAtI(v)); // arr[ii] = v; ii++;
            //@ assert Test.ii == st.count();

            //@ assert arr[4] == 5;
            //@ assert (\forall int j; j>=0 && j<arr.length; arr[j] == st.value[j]);
      }
}
```

FUNCTIONAL PROGRAMMING IN C++
IMPLICIT ITERATION

In client code:

```
public class Test {

        …

        public void foo() {

                …

                ii = 0;

                //@ loop_invariant Test.ii == \count;
                //@ loop_invariant (\forall int j; j>=0 && j<\count; arr[j] == st.value[j]);
                //@ loop_modifies Test.ii, Test.arr[*];
                //@ loop_invariant i == \count && 0 <= i && i <= _length;
                //@ decreases _length - i;
                for (int i=0; i < _length; i++) {
                   (v -> putAtI(v)).accept(st.values[i]); --------->> putAtI(st.values[i])
                }

                //st.forEachOrdered(v -> putAtI(v)); // arr[ii] = v; ii++;

                //@ assert Test.ii == st.count();

                //@ assert arr[4] == 5;
                //@ assert (\forall int j; j>=0 && j<arr.length; arr[j] == st.value[j]);
        }
}
```

FUNCTIONAL PROGRAMMING
SIMPLE CASES

In our first Java S&V project using FP, nearly all uses of FP could be handled by a combination of

- specification interfaces
- inlining function literals
- model programs
- combining user and library specs for implicit iteration

Did not need to handle the full generality of FP

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
choice(bool b, ... ftrue, ... ffalse) {
  return [](int i){ return b ? ftrue(i) : ffalse(i); }
}
```

What should we write as the specifications of **choice?**

We need to be able to express the specification of the result in terms of specifications of arguments.

[Kassios, Müller, 2011: Modular Specification and Verification of Delegation with SMT solvers]

- No need for 2nd order functions or reasoning
- Translation can be handled by SMT solvers

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

[Just working with unary functions int -> int]

Precondition of f(i):

pre(f, i) **= …**

Postcondition of f(i) returning r:

post(f, r, i) **= …**

Frame condition of f(i):

writes(f, i) **= …**

Reads footprint of f(i):

reads(f, i) **= …**

If we use pre(f) that returns a function, which is the precondition, then we start needing to manipulate functions in SMT

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
compose(... f, ... g) {
  return [](int i){ return f(g(i)); }
}
```

ensures forall int i: **pre**(\result,i) = (**pre**(g,i) &&
    (forall int t : **post**(g,t,i) ==> **pre**(f,t)));

ensures forall int r,i : **post**(\result,r,i) =
    (exists int t: **post**(g,t,i) && **post**(f,r,t));

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, addone);
int k;
int kk = h(k);
assert k == kk;
```

```
assume forall int r,i : post(h,r,i) =
        (exists int t: post(addone,t,i) && post(subone,r,t) );

assume post(h,kk,k);
assert k == kk;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, addone);
int k;
int kk = h(k);
assert k == kk;
```

```
forall int r i: post(addone, r, i) = (r == i+1)
forall int r i: post(subone, r, i) = (r == i -1)
```

```
assume forall int r,i : post(h,r,i) =
        (exists int t: post(addone,t,i) && post(subone,r,t) );

assume post(h,kk,k);
assert k == kk;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, addone);
int k;
int kk = h(k);
assert k == kk;
```

```
forall int r i: post(addone, r, i) = (r == i+1)
forall int r i: post(subone, r, i) = (r == i -1)
```

```
assume forall int r,i : post(h,r,i) =
        (exists int t: (t==i+1) && (r== t-1) );

assume post(h,kk,k);
assert k == kk;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, addone);
int k;
int kk = h(k);
assert k == kk;
```

```
assume forall int r,i : post(h,r,i) =
            (r == i );

assume post(h,kk,k);
assert k == kk;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, addone);
int k;
int kk = h(k);
assert k == kk;
```

```
assume forall int r,i : post(h,r,i) =
                (r == i );

   post(h,kk,k) =
                (kk == k );

assume post(h,kk,k);
assert k == kk;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, bump);
int k;
int kk = h(k);
assert k == kk;
```

```
forall int r i: post(bump, r, i) = (r > i)
forall int r i: post(subone, r, i) = (r == i -1)
```

```
assume forall int r,i : post(h,r,i) =
            (exists int t: post(bump,t,i) && post(subone,r,t) );

assume post(h,kk,k);
assert …;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, bump);
int k;
int kk = h(k);
assert ...;
```

```
forall int r i: post(bump, r, i) = (r > i)
forall int r i: post(subone, r, i) = (r == i -1)
```

```
assume forall int r,i : post(h,r,i) =
          (exists int t: (t>i) && (r==t-1) );

assume post(h,kk,k);
assert …;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, bump);
int k;
int kk = h(k);
assert ...;
```

```
assume forall int r,i : post(h,r,i) =
          ( r+1 > i);

assume post(h,kk,k);
assert ...;
```

FUNCTIONAL PROGRAMMING IN C++
COMPOSITION

```
h = compose(subone, bump);
int k;
int kk = h(k);
assert ...;
```

```
assume forall int r,i : post(h,r,i) =
                ( r+1 > i);

assume post(h,kk,k);
assert   kk >= k ;
```

FUNCTIONAL PROGRAMMING IN C++

- Hand translations to SMT have a lot of quantification:
  - provable conjectures prove very quickly
  - invalid conjectures timeout on Z3, OK on CVC4

- **pre** and **post** will have different signatures for each function signature

- this same technique can be adopted for C function pointers
  (and for function objects implemented with Java anonymous classes)

- There are also rules for frame conditions and various other details

FUNCTIONAL PROGRAMMING IN C++
BACK TO IMPLICIT ITERATION

- Want to write a stand-alone specification for **transform(in_begin, in_end, out_begin, f);**

```
in = in_begin;
out = out_begin;

//@ loop_invariant (\forall int j; 0<=j && j<\count;
       … accumulation of effects of \count iterations … );

for (int \count; 0 <= \count && \count < in_end-in_begin) {
   *out = f(*in); // note other implicit inputs and side effects
   in++; out++;
}

//@ ensures (\forall int j; 0<=j && j<in_end-in_begin;
       … accumulation of effects of all iterations … );
```

- The specifications need to incorporate the recurrence solution of the inductive formula corresponding to each iteration.

FUNCTIONAL PROGRAMMING IN C++
BACK TO IMPLICIT ITERATION

- Can do that for some cases:

```
/*@ behavior pure:
      assumes \separated( in_begin..in_end-1, out_begin..out_begin+(in_end-1-in_begin) );
      assumes (\forall int j; 0<=j<in_end-in_begin ==> \writes(f,in_begin[j]) == \empty);
      assumes (\forall int j; 0<=j<in_end-in_begin ==> \reads(f,in_begin[j]) == \empty);
      requires (\forall int j; 0<=j<in_end-in_begin ==> \pre(f,in_begin[j]));
      assigns out_begin[0 .. in_end-1-in_begin];
      ensures (\forall int j; 0<=j<in_end-in_begin ==> \post(f,out_begin[j],in_begin[j]));

    behavior pure_inplace:
      assumes in_begin == out_begin;
      assumes (\forall int j; 0<=j<in_end-in_begin ==> \writes(f,in_begin[j]) == \empty);
      assumes  … reads footprint is separated from out range …
      requires (\forall int j; 0<=j<in_end-in_begin ==> \pre(f,in_begin[j]));
      assigns out_begin[0 .. in_end-1-in_begin];
      ensures (\forall int j; 0<=j<in_end-in_begin ==> \post(f,out_begin[j],\old(in_begin[j])));

      …
@*/
transform(in_begin, in_end, out_begin, f);
```

FUNCTIONAL PROGRAMMING IN C++
BACK TO IMPLICIT ITERATION

- Input might read past output:

```
/*@
    behavior write_ahead:
      assumes … reads footprint, out range, writes footprint are separated …
      assumes out_begin == in_begin+1;
      requires (\forall int j; 0<=j<in_end-in_begin ==> \pre(f, … output value at j …));
      assigns \union(0,in_end-in_begin-1, \writes(f, _ )), out_begin[0 .. in_end-1-in_begin];
      ensures (\forall int j; 0<=j<in_end-in_begin ==> \post(f, out_begin[j], in_begin[j]));

@*/
transform(in_begin, in_end, out_begin, f);
```

Updated value (out_begin[j-1])

```
e.g.,

int a[100];
a[0] = 0;
transform(&a[0], &a[99], &a[1], [](int i){ return i+1; }

produces

a = 0,1,2,3,4,5,6,7,8,9,…
```

FUNCTIONAL PROGRAMMING IN C++
BACK TO IMPLICIT ITERATION

- But still…
  - The general case is not expressible
  - Having a plethora of special cases is not ideal
  - However, common cases can be specified
- … This aspect is still a work in progress

CONCLUSION

- S&V is being accomplished at industrial scale
  - Still takes experts
  - Still takes significant effort
  - But are producing results of value (not just research endeavors)
  - Starting to be accepted by developers
- There are S&V project management issues to solve
- There are (encoding and reasoning) tool enhancement issues to solve
- There are new specification language issues brought about by the combination of imperative and functional programming
- **Theory and tool advancements go best hand-in-hand with practical applications.**

Contacts :

Email : {david.cok@cea.fr}
Email : {david.r.cok@gmail.com}

ORIGINAL ABSTRACT

- Title: Lessons from verifying legacy Java code for C++ specification & verification
- David R. Cok

- Legacy code is typically written with no regard for specification and verification. Consequently verification tools applied to legacy code must support most language features and be able to scale to the size and scope of industrial software. This talk will use case study examples from a verification project targeting industrial Java code to demonstrate both how verification of features in Java 8 was achieved and what challenges still remain for such projects. Then we will illustrate how the lessons learned from that project are informing the design of a specification language for C++.

WORKSHOP TOPIC

- The theme for this year will be "Sound Open Source Static Analysis for Security", with sessions on "analysis of legacy code", "use in new developments" and "accountable software quality". So we plan to invite speakers across critical industries (railway, OS/networks, avionics, nuclear) and research labs (Sandia Labs, NASA, Galois) to present their use of either Frama-C or SPARK in these contexts. For your one-hour keynote, what we have in mind is an overview of where sound static analysis (mostly based on deductive verification, but could be also abstract interpretation) can provably help with the development of higher quality / more secure software, based on your extensive experience with the use of formal methods for security in an industrial context. Of course, let us know if you'd like to present something completely different, that may be even better.
-