

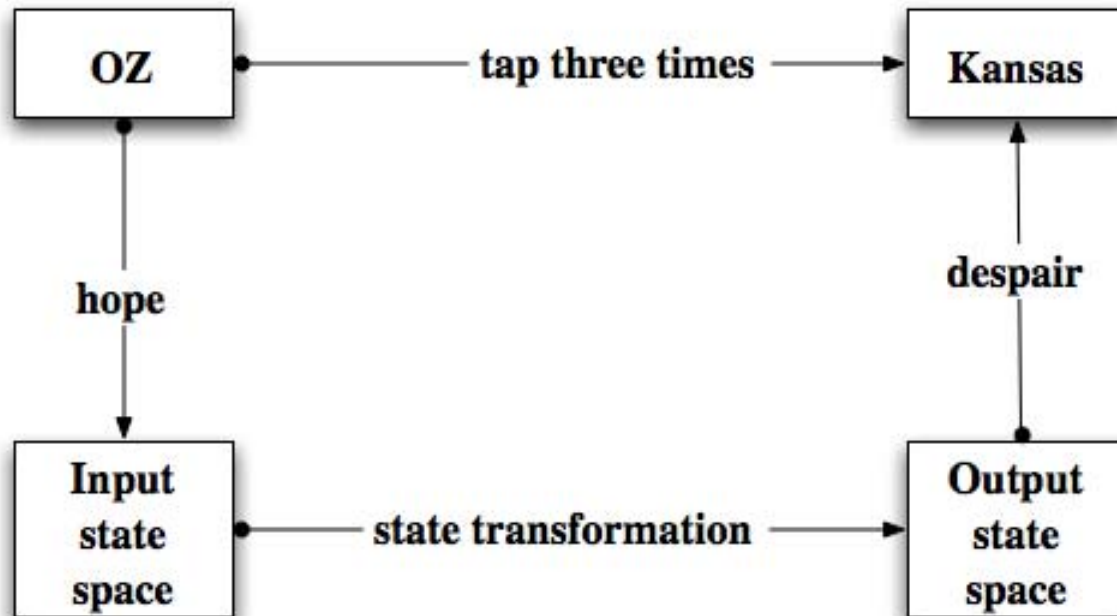
Destiny

Computer aided bottom up code review

Frank Rimplinger NSA

`frankrimlinger@mac.com`

Problem statement



Starting with a fuzzy diagram such as this, can we pass to a commutative diagram in which all four maps are well defined and the original “intent” is preserved?

Mission Critical Software

- Destiny addresses the problem of software assurance with a rigorous modeling process that complements testing and model checking approaches.
- The strength of Destiny analysis is that of mathematical proof. No approximations or simplifications of code behavior are introduced. Single threaded analysis only.
- Serves as intermediary between the analyst, the source code, and the ACL2 theorem-prover.
- Process detects errors and synthesizes a precise notion of “good outcomes” for each analyzed method.

Procedure

- Destiny operates on Java code, leveraging an important Java feature: the concise description of the state of the computer, as modeled by the JVM (Java Virtual Machine).
- Starting with a primitive method M , Destiny guides the analyst to a precise description of the “good” outcomes of M , rendered as far as possible in terms of source code level constructs.
- Destiny forbids other methods dependent on M from passing “bad” states to M .
- Process repeats as necessary to specify good outcomes and forbid bad input until all layers of software have been described.

Comprehension requires context

- Destiny provides the **state** of the computer at any execution point within a method via a point-and-click navigable interactive display.
- The state is a concise description of the heap, the stack of method frames, and the static area.
- Source code level names are preserved.
- The interactive display also provides the constraints on input required to achieve the displayed state.
- Given all this information, the analyst then chooses between three possibilities: a good outcome, a bad outcome, or an error in the code.

Analyst as oracle

- The analyst role as oracle is essential in any verification process.
- Proof-by-construction: front-loaded analyst effort to set up a mechanism that will generate the correct code. Top-down.
- Destiny: back-loaded analyst effort to compare ordinary developer documentation against descriptions of state generated by the tool. Bottom-up.
- The two different points of view complement each other, and may potentially be used together to help detect **oracle error**.

Analyst as pioneer

- There is an **organic** cycle to computer-aided tool development. Each development cycle of Destiny:
- introduces new features to relieve repetitive analyst tasks,
- increases the level of source code complexity that the tool can successfully handle, and
- produces incremental evidence of tool viability for management.
- The role of the analyst as **pioneer** in the tool development cycle is as **critical** as the traditional role as oracle in the verification process.

Example of Destiny process

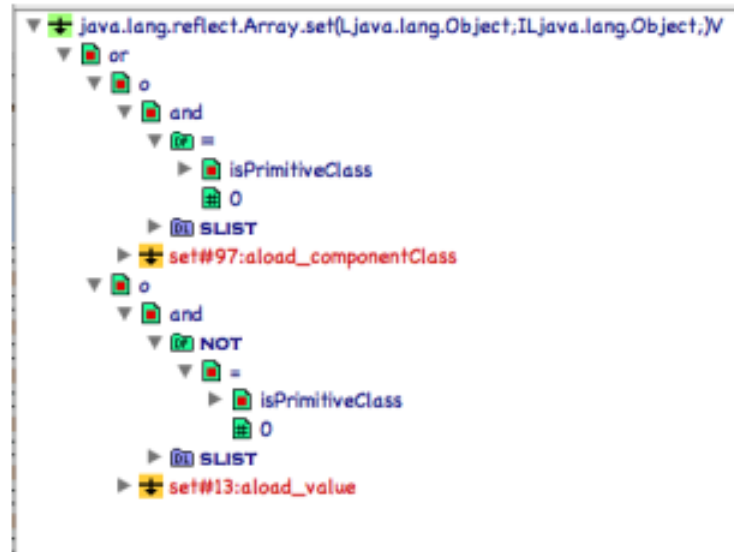
- Destiny presents a method as a series of decision points, where the analyst decides what outcome to work towards and verifies that things “make sense”.
- To see this process in action, consider the Destiny implementation, written in java, of the **typically** native method `java.lang.reflect.Array.set()`


```

: public static void set(Object array,int index, Object value)
:     throws IllegalArgumentException,ArrayIndexOutOfBoundsException{
:
:     Class componentClass=array.getClass().getComponentType();
:     if(componentClass==null){
:         // Array does not represent an array class
:         throw new IllegalArgumentException();
:     }
:     if(componentClass.isPrimitive()){
:         if(value instanceof java.lang.Boolean){
:             setBoolean(array, index, ((Boolean)value).booleanValue());
:         }
:         else if(value instanceof java.lang.Character){
:             setChar(array, index, ((Character)value).charValue());
:         }
:         else if(value instanceof java.lang.Byte){
:             setByte(array, index, ((Byte)value).byteValue());
:         }
:         else if(value instanceof java.lang.Short){
:             setShort(array, index, ((Short)value).shortValue());
:         }
:         else if(value instanceof java.lang.Integer){
:             setInt(array, index, ((Integer)value).intValue());
:         }
:         else if(value instanceof java.lang.Long){
:             setLong(array, index, ((Long)value).longValue());
:         }
:         else if(value instanceof java.lang.Float){
:             setFloat(array, index, ((Float)value).floatValue());
:         }
:         else if(value instanceof java.lang.Double){
:             setDouble(array, index, ((Double)value).doubleValue());
:         }
:         else{
:             // A primitive array requires a primitive value, so no can do
:             throw new IllegalArgumentException();
:         }
:     }
:     else{
:         // OK, we must be storing an object
:         // Check for assignment compatibility
:         if(!componentClass.isAssignableFrom(value.getClass())){
:             throw new IllegalArgumentException();
:         }
:
:         // Check array bounds
:         ArrayExceptions(array,index);
:
:         // assign the object
:         Destiny_Object.setArrayObjectValue(array,index, value);
:     }
: }

```

After some issues involving the calls to `getClass()`, `getComponentType()` and `isPrimitive()` have been resolved, Destiny provides the analyst with a choice. Both these choices lead to good outcomes, so two separate “abstractions” are born at this point.

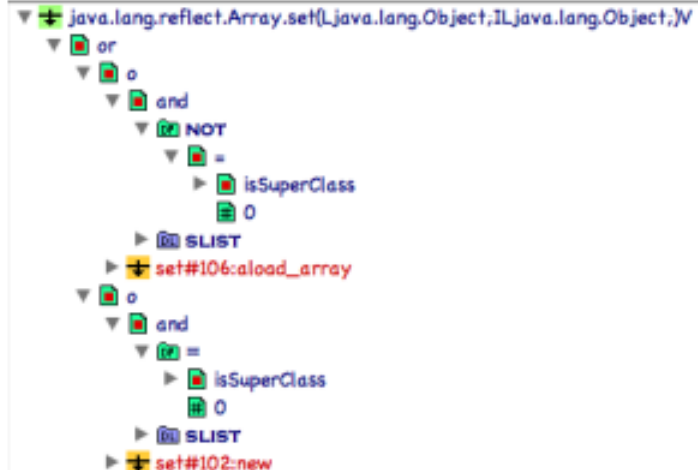


```

: public static void set(Object array,int index, Object value)
:     throws IllegalArgumentException,ArrayIndexOutOfBoundsException{
:
:     Class componentClass=array.getClass().getComponentType();
:     if(componentClass==null){
:         // Array does not represent an array class
:         throw new IllegalArgumentException();
:     }
:     if(componentClass.isPrimitive()){
:         if(value instanceof java.lang.Boolean){
:             setBoolean(array, index, ((Boolean)value).booleanValue());
:         }
:         else if(value instanceof java.lang.Character){
:             setChar(array, index, ((Character)value).charValue());
:         }
:         else if(value instanceof java.lang.Byte){
:             setByte(array, index, ((Byte)value).byteValue());
:         }
:         else if(value instanceof java.lang.Short){
:             setShort(array, index, ((Short)value).shortValue());
:         }
:         else if(value instanceof java.lang.Integer){
:             setInt(array, index, ((Integer)value).intValue());
:         }
:         else if(value instanceof java.lang.Long){
:             setLong(array, index, ((Long)value).longValue());
:         }
:         else if(value instanceof java.lang.Float){
:             setFloat(array, index, ((Float)value).floatValue());
:         }
:         else if(value instanceof java.lang.Double){
:             setDouble(array, index, ((Double)value).doubleValue());
:         }
:         else{
:             // A primitive array requires a primitive value, so no can do
:             throw new IllegalArgumentException();
:         }
:     }
:     else{
:         // OK, we must be storing an object
:         // Check for assignment compatibility
:         if(!componentClass.isAssignableFrom(value.getClass())){
:             throw new IllegalArgumentException();
:         }
:
:         // Check array bounds
:         ArrayExceptions(array,index);
:
:         // assign the object
:         Destiny_Object.setArrayObjectValue(array,index, value);
:     }
: }

```

Choosing the primitive branch leads to eight new abstractions corresponding to the eight primitive types. Choosing the other branch leads to the “assignment compatibility” decision. We avoid the “bad outcome” which will throw an exception, so no new abstractions are born in the non-primitive case.

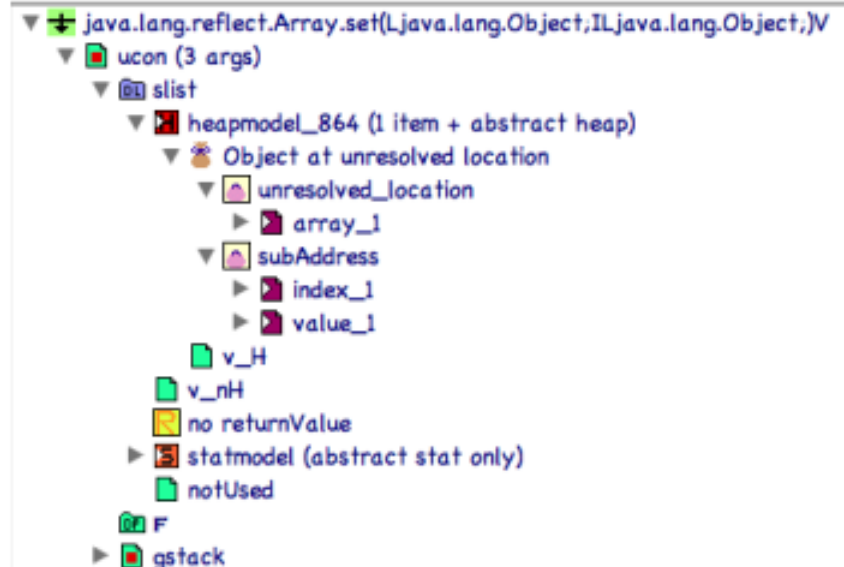


```

: public static void set(Object array,int index, Object value)
:     throws IllegalArgumentException,ArrayIndexOutOfBoundsException{
:
:     Class componentClass=array.getClass().getComponentType();
:     if(componentClass==null){
:         // Array does not represent an array class
:         throw new IllegalArgumentException();
:     }
:     if(componentClass.isPrimitive()){
:         if(value instanceof java.lang.Boolean){
:             setBoolean(array, index, ((Boolean)value).booleanValue());
:         }
:         else if(value instanceof java.lang.Character){
:             setChar(array, index, ((Character)value).charValue());
:         }
:         else if(value instanceof java.lang.Byte){
:             setByte(array, index, ((Byte)value).byteValue());
:         }
:         else if(value instanceof java.lang.Short){
:             setShort(array, index, ((Short)value).shortValue());
:         }
:         else if(value instanceof java.lang.Integer){
:             setInt(array, index, ((Integer)value).intValue());
:         }
:         else if(value instanceof java.lang.Long){
:             setLong(array, index, ((Long)value).longValue());
:         }
:         else if(value instanceof java.lang.Float){
:             setFloat(array, index, ((Float)value).floatValue());
:         }
:         else if(value instanceof java.lang.Double){
:             setDouble(array, index, ((Double)value).doubleValue());
:         }
:         else{
:             // A primitive array requires a primitive value, so no can do
:             throw new IllegalArgumentException();
:         }
:     }
:     else{
:         // OK, we must be storing an object
:         // Check for assignment compatibility
:         if(!componentClass.isAssignableFrom(value.getClass())){
:             throw new IllegalArgumentException();
:         }
:
:         // Check array bounds
:         ArrayExceptions(array,index);
:
:         // assign the object
:         Destiny_Object.setArrayObjectValue(array,index, value);
:     }
: }

```

And it's a wrap! Having made all these decisions, the method simplifies down to the desired good outcome. All this information is stored persistently as an “abstraction rule” for `java.lang.reflect.Array.set()`.



java.lang.reflect.Array.set(Ljava.lang.Object;ILjava.lang.Object;)V

sets array [index]=value

Assumptions:

array calls

java.lang.Object.getClass()Ljava.lang.Class;
array.getClass does not throw invocation exception
getClass float, array case
componentClass is not null
componentClass calls java.lang.Class.isPrimitive()Z
componentClass.isPrimitive()Z does not throw invocation exception

componentClass is not primitive

value calls

java.lang.Object.getClass()Ljava.lang.Class;
getClass float for value, Array case
componentClass calls
java.lang.Class.isAssignableFrom(Ljava.lang.Class;)Z
componentClass.isAssignableFrom does not throw invocation exception
componentClass is assignable from value
value.getClass does not throw invocation exception
array exceptions float for value

The analyst also makes a brief notation for each decision made, and these form a readable “transcript” of the rule. Although not precise, these notes preserve the thought process involved in the abstraction.

Observations

- The analyst-as-oracle perceives that the pictures line up with the source code in a meaningful way.
- The oracle concludes, assuming all the rest of the abstractions for set() simplify correctly to the desired outcome, that this method is **correct**.
- The analyst in the role of pioneer would have many **complaints**, such as:
 - Some sort of template mechanism is required to eliminate the tedium of working through a bunch of almost identical cases
 - The notes that the analyst makes at each decision point should really be automatically generated by an synthetic language process. This would eliminate a very real source of confusion caused by inaccurate notations.
- Feedback ultimately drives a new generation of the tool.
- **Over time, the tool becomes more and more powerful.**
- Each generation requires more \$\$\$ than the last to successfully pursue, so the effort is always to find the next “killer feature.”

Special considerations for loops

- Destiny automatically handles the translation of loops in source code into a potentially hierarchical system of tail recursive functions.
- Destiny cannot reason inductively, but it does guide the analyst to form **conjectures** and **axioms**. These artifacts are parsed into the ACL2 language, where a theorem-prover specialist can prove the conjectures, given the axioms.
- On the Destiny side, the conjectures are *assumed* to be true (unless proven otherwise), and the axioms become part of the hypothesis set of a given good outcome.
- Destiny hides as much detail of the “Destiny JVM implementation” as possible from ACL2, allowing ACL2 to concentrate on the fundamental logic of the situation.

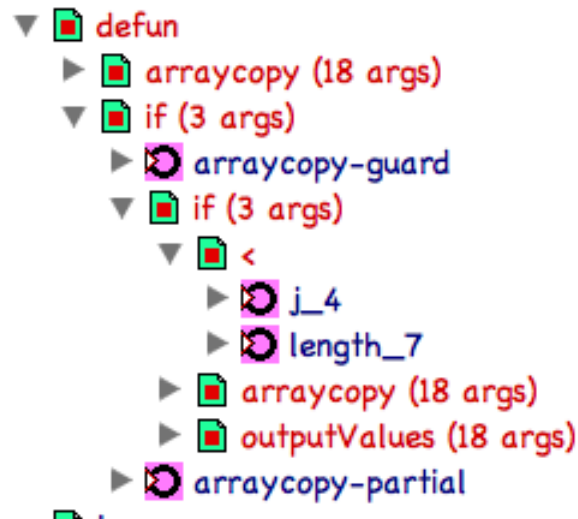
Loop code example

- Destiny implements `java.lang.System.arraycopy()` as java code.
- This method does a lot of error checking, taking into account widening conventions for primitive type arrays, and assignment issues for arrays of references. Moreover, the method also does range checking, and supports a buffered copy in the event of source/destination overlap.
- For this example, we concentrate on the loop inside `arraycopy()` that performs the actual copy of data from one array to the other.










```


















for (int j = 0; j < length; j++){
    Object currentSrcValue=Array.get(sourceArray, currentsrc);
    // item level compatibility for reference type copy
    if(!primitiveCopyFlag){
        if(!destComponentClass.isAssignableFrom(currentSrcValue.getClass())){
            //reference types don't match
            throw new ArrayStoreException();
        }
    }
    Array.set(dest, currentdest,currentSrcValue);++currentsrc;++currentdest;
}

```



Destiny presents the decision points inside the loop as in the first example, but now the results of these decisions become part of the “loop guard”, which must be satisfied each pass through the loop.

- ▼  and (4 args)
 - ▶  dest.Array_Type_Marker is B at 4.1
 - ▶  get float for srcarray, case B at 4.1
 - ▶  primitivecopyflag is true at 4.17.06,1
 - ▼  set float, primitive case at 4.17.06,9
 -  frank
 - ▶  nh
 - ▶  h
 - ▶  stack

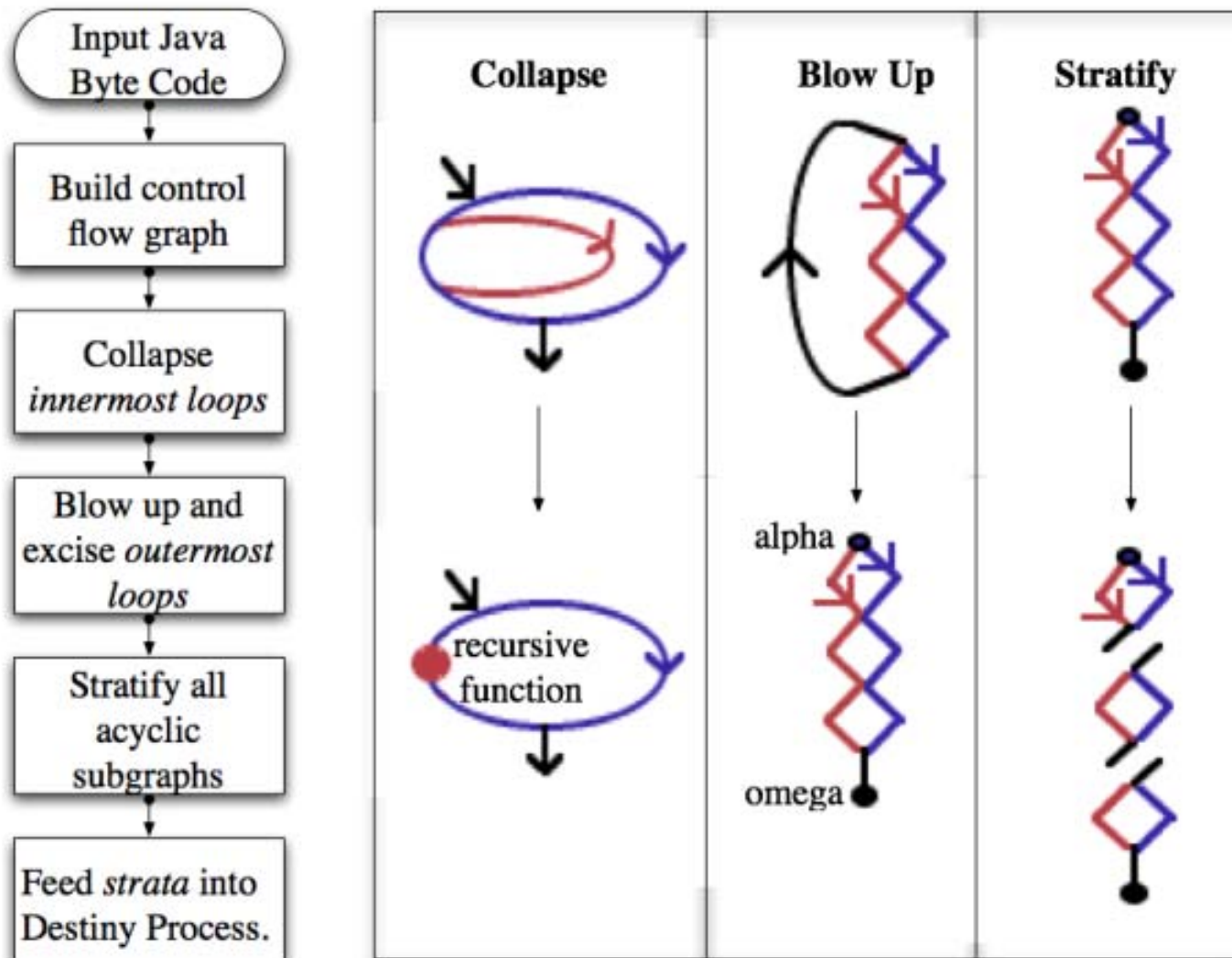
- ▼  arraycopy loop guard as axiom
 - ▼  and (4 args)
 - ▶  dest.Array_Type_Marker is B at 4.1
 - ▶  get float for srcarray, case B at 4.1
 - ▶  primitivecopyflag is true at 4.17.06,1
 - ▼  set float, primitive case at 4.17.06,9
 -  frank
 - ▼  +
 - ▶  v-nh_1
 -  4
 - ▶  pushH (3 args)
 - ▼  pushFrame
 - ▼  storeCat1 (3 args)
 - ▶  currentsrcvalue_5
 -  13
 - ▶  storeCat1 (3 args)
 - ▶  popFrame

Each variable maintains a history of its past values. Input values to the method become primitive symbols in the ACL2 logic. By “lifting” the loop guard to these primitive symbols, an **axiom** is automatically generated for the ACL2 logic. Therefore, to the extent that (an appropriate transformation of) the variables is loop invariant, ACL2 need not be cognizant of the details of the internal structure of the Destiny JVM implementation.

Observations about loops

- Once a conjecture is made, Destiny simplifies the methods based on the assumption that the conjecture is in fact true. A replay facility allows the analyst to recover from bad conjectures, saving whatever work is salvageable.
- The abstraction of `arraycopy()` does not actually assert that this method copies the data from one array to another. Such a conjecture will be generated by a method which calls `arraycopy()`. Calling method variables may access the heap upon the return from `arraycopy()`, whence conjectures involving such variables are indirectly conjectures about `arraycopy()`.
- To assist in generating such conjectures, Destiny automatically *specializes* the generic ACL2 theory for `arraycopy()` based on the input values supplied by the caller. The specialized theory then *augments* the ACL2 theory of the caller.
- By this means, the ACL2 effort achieves re-usability.
- An important conjecture about a loop is that it terminates, given whatever assumptions on the input are reasonable. Of course, the question of termination is logically undecidable, and in practice Destiny is of limited help in determining such assumptions. However, Destiny does guide the user to the *statement* of the appropriate termination conjecture.

Automated Code Subdivision Process



From strata to good outcomes

- The code subdivision process emits “strata”, which are tiny bits of control flow with few or no branch points.
- The code specification process takes as input genuine Java methods, (or minimal co-recursive sets of methods). At the byte code level, even the simplest methods contain many branches, whence the control flow of a method contains many strata.
- The Destiny interactive tool allows the user to efficiently and transparently navigate through the strata of a method and locate the “good paths” that lead to the “good outcomes.”

Strategies to manage complexity

- The rewriter automatically attempts a breadth-first top-down pattern matching strategy. All expressions are simplified at a given height, relative to the stratification. Then the next height is interpreted and the process repeats.
- The “tree viewer” allows a user to navigate any expression representing control flow and/or state transition in a top-down manner, used to target commands moderating the creation of the specification of a method.
- The “graph viewer” allows a user to understand very local behavior of control flow, now used mostly for debugging purposes.
- The “3D graph viewer”, under development, will allow the user to orchestrate rewriting strategies that complement, modify, or generalize the automatic breadth-first rewriting strategy.

On-board technologies

- Destiny technology (loops and strata subdivision)
- Breadth-first top-down conditional rewriter
- Full-blown JVM model
- 2d and 3d interactive rendering of control flow
- Interactive graphic layer for formula manipulation
- Interactive control layer for specification generation
- Recursive function generation
- ACL2 theory and specialization generation
- Persistent storage for rule base and specifications
- Playback facility for regression testing
- Lifetime analysis of heap objects

Status, goals and philosophy

- Project is currently demonstrating “first operational capability”, applied to actual customer code.
- Goal is always more and better automation.
- Rule base oriented strategies allow for great flexibility and experimentation.
- Visualization enables debugging capability and also the conceptualization of new techniques.
- Always on the look-out for opportunities to integrate established technologies, especially model-checking and VCG's.

Who are we?

- Frank Rimlinger, NSA, project manager and chief architect.
- Jon Barrilleaux, JMB Oakland CA, visualization senior scientist.
- Warren Hunt Jr, Prof. of Computer Science at University of TX, consultant
- Robert Krug, University of TX staff member, formal methods support
- Jason Roberts, JMB visualization support (currently unfunded)
- Marc Durant, JMB algorithmic and technical support