Enforcing Static Program Properties in Safety-Critical Java Software Components

Dr. Kelvin Nilsen
Aonix

Using the Java language for the development of safety-critical code requires even more enforcement of static properties than is enforced by the traditional Java platform. This article examines style guidelines and describes development tools that enforce the guidelines in order to enable cost-effective certification of Java application code to DO-178B Level A and similar safety certification standards. These approaches eliminate the need for garbage collection, support safe and efficient modular composition of independently developed software components, and enable automatic analysis of an application's worst-case memory and CPU-time requirements.

Software deployed in safety-critical systems must achieve the highest standards of quality, must exhibit a high level of determinism, and must rely on minimal run time services to facilitate proof of the run time environment's correctness [1, 2]. Because of the rigorous certification requirements associated with safety-critical software, developers of safety-critical systems generally adopt styles of programming that are easier to certify but may be more difficult to program. For example, the safety-critical Java standard (JSR-302) offers stack memory allocation as an alternative to standard edition Java's garbage-collected heap [3, 4]. Although the algorithms for allocating and deallocating stack memory are very simple, efficient, and deterministic, reliance on stack memory introduces a different kind of problem. In particular, dangling pointers may be introduced if pointers to stack-allocated objects live longer than the objects they refer to. Certification of a safety-critical system needs to prove the absence of dangling pointers in addition to proving that each memory allocation request will be satisfied in a predictable amount of time.

The Real-Time Specification for Java (RTSJ) [5] eliminates dangling pointers to stack-allocated objects by enforcing the rule that no object allocated in an outer-nested stack frame may hold a pointer to any object allocated in an inner-nested stack frame. Enforcement of this rule is performed with a run time check every time an object's field is overwritten. According to this rule, a seemingly harmless statement like:

```
anObject.aField = aValue;
```

will throw an IllegalAssignmentError exception if aValue resides in a scope that is more inner-nested than the scope that holds anObject. Certification of a safety-critical application must prove that no assignments abort with this exception.

In the vernacular of computer science, a *static property* is a property that can be determined by analysis of a software program without (or before) running the program. For safety-critical software, all of the properties that are critical to its safe operation should be static properties. By the time the software is running as part of a safety-critical system, it is too late for a run time check to detect that a critical property has been violated.

There are two common approaches to verification of static properties. The approach most familiar to software engineers is a programming language type system [6, 7]. The second approach, broadly characterized as the use of static analysis, augments the analysis performed by the programming language type system. Table 1 highlights some of the differences between the two.

This article describes the implementation of a type system that enforces properties that are important to the development of safety-critical code using the Java language. The approaches are somewhat unique in that our implementation of the safety-critical Java type system borrows certain techniques that are more traditionally used in static analyzers. These techniques are not usually used in the implementation of the safety-critical Java type system. We make these techniques more efficient and precise by restricting the data-flow analysis to one method at a time. Among the important properties that are analyzed by the safety-critical type system, it can enforce that:

1. A method is written in a restrictive style allowing special development tools to automatically analyze the CPU time and the stack memory required to execute the method.
2. A method is known not to block its execution awaiting some condition that is to be satisfied by some other thread or by an external event.
3. A method does not copy its incoming arguments into state variables that would possibly persist beyond execution of the method itself.
4. The objects referenced from certain incoming arguments to a method are known to reside in a scope that encloses (surrounds) the scopes containing objects referenced from certain other arguments.

All of these properties are important attributes of real-time software systems. Property 1 is important when a programmer wants to know if it is appropriate to invoke a particular method from a context, such as a hardware interrupt handler or a hard real-time task that requires a reliable upper bound on the amount of CPU time and memory required to execute the method. Property 2 must be verified for methods that are invoked while holding certain kinds of priority ceiling locks. When temporary objects are passed as arguments to a method, property 3 establishes an assurance that a dangling pointer will not result as a side effect of the operations performed within the invoked method. Finally, the knowledge represented by property 4 makes it possible for a method to safely establish pointers from certain temporary objects (the ones that are known to have shorter lifetimes) to certain other temporary objects (those known to have longer lifetimes).

Safety-Critical Type Declarations

The safety-critical type system uses the meta-data annotation system introduced with Java 5.0 to associate safety-critical properties with specific software components[1]. Programmers use the type system of standard edition Java to specify, for example, that a particular method's argument is of HighResolutionTime. Using annotations to augment the standard edition type system, safety-critical Java developers use the safety-critical type system's @Scoped annotation to clarify, for exam-

Characteristic	Type System	Static Analyzer
Identification of Static Properties	The programmer inserts declarations to identify intent and the type system verifies that the code is consistent with the declared intent.	The static analyzer infers the intent from context and usage. The static analyzer may infer different intentions for the same code when used in different contexts. The static properties that will be inferred by the static analyzer are not easily recognized by the human review of source code.
Enforcement	By refusing to translate programs that contain type system errors, the compiler enforces the type system.	Programmers may decide not to run the static analyzer or may ignore its recommendations.
Precision	The programming language specification must precisely characterize exactly what constitutes a legal program.	The characterization of what will be understood by a static analyzer is much less precise. One vendor's static analyzer may reach very different conclusions than another's. Static analyzers may produce *false negatives*, stating that certain lines of code may violate desired properties even though an intelligent human analysis would prove that the code does not represent a problem. Static analyzers may produce *false positives*, concluding that a desirable property holds true when really it does not. This typically occurs when humans misconfigure the analysis in an attempt to reduce false negatives.
Efficiency	Because the compiler runs so frequently, type systems generally restrict themselves to properties that are easily and efficiently verified.	Whereas a compiler generally runs in seconds, a static analyzer often requires hours. Rather than focusing attention on each method or class in isolation, the typical static analyzer attempts to discover all of the contexts from which each method might be invoked, and it propagates static information known about each context into the execution of the method within that context.
Expressive Power	To facilitate efficient enforcement, the *vocabulary* for speaking about types is limited.	In theory, static analyzers can distinguish many more subtle nuances than a type system and can treat particular program components as having different properties when invoked from different contexts.

Table 1: *Comparison of the Two Common Approaches to Static Property Verification*

ple, that the argument may have been allocated in stack memory. The following method declaration illustrates this usage:

```
void setDeadline(@Scoped
HighResolutionTime newDeadline);
```

The remainder of this section describes some of the annotations that are available to safety-critical developers using the safety-critical type system.

Resource Limitations

To indicate that a particular method must be implemented using a subset of the full Java language—that can be automatically analyzed by development tools to determine the worst-case CPU time and stack memory requirements—its declaration is accompanied by a @StaticAnalyzable annotation, as in the following code:

```
@StaticAnalyzable
void handleAsyncEvent() {
    // method body
}
```

The safety-critical Java type system enforces that all overriding methods also be @StaticAnalyzable. Furthermore, the type system enforces that any methods invoked from within a @StaticAnalyzable method are also declared @StaticAnalyzable, and it requires that the programmer provide special assertions to limit iteration counts and recursion depths, and to bound the sizes of any arrays or strings allocated within the method.

Non-Blocking Behavior

A special form of the @StaticAnalyzable annotation allows developers to state a requirement that the implementation of a particular method does not perform any blocking operations. Java annotations have associated attributes, with default values for each attribute. One of the attributes of @StaticAnalyzable is named enforce_non_blocking. Its default value is *true*. To specify that blocking is allowed, developers can override the default value, as in the following method declaration:

```
@StaticAnalyzable(enforce_non_blocking
= {false})
void waitForInput();
```

When declared (as shown), the stack memory usage and the total CPU time consumed by this method are bounded. However, since the method may block waiting for input, analysis of how long the method will execute depends on understanding when the input will become available.

Captive-Scoped Arguments

Certain incoming method arguments may be declared as @CaptiveScoped, meaning that the method promises to hold copies of those argument values only within its local variables or passed to other methods as @CaptiveScoped arguments. @CaptiveScoped arguments can never be copied to instance or static fields. Thus, the invoker of a method knows that it can safely reclaim the memory associated with temporary objects, which are passed as @CaptiveScoped arguments as soon as the invoked method returns. The following declaration demonstrates use of the @CaptiveScoped annotation:

```
@CaptiveScopedThis void
reserve(@CaptiveScoped SizeEstimator
sizeIncrement);
```

Nesting Relationships of Stack-Allocated Arguments

In certain situations, the safety-critical Java type system understands that incoming temporary arguments have certain relative lifespan orderings. For example, the @Scoped arguments to an instance method of a reentrant-scope object are known to have a lifetime that is at least as long as the reentrant scope object itself. The safety-critical Java system organizes memory as a hierarchy of scopes. If one object is known to live as long as another, we say the first *encloses* the second. This terminology derives from the hierarchy of scopes within which the two objects reside. Outer-nested scopes

enclose inner-nested scopes. The scopes are organized as a stack, so objects residing in outer-nested scopes live longer than objects residing in inner-nested scopes. Consider the following method declaration:

```
// Assume this method is associated with
// a @ReentrantScope class
@ScopedThis put(@Scoped Object
anObject);
```

At the invocation point of this method, the safety-critical Java type system enforces that the value assigned to the incoming anObject argument encloses the value assigned to the implicit this argument. Thus, within the implementation of the put() method, it is safe to assign anObject to a field of this. Since anObject lives at least as long as this, no dangling pointer will result when anObject's memory is reclaimed.

Data-Flow Analysis

Data-flow analysis consists of analyzing the flow of information within a software module. Traditionally, type systems do not perform data-flow analysis. Rather, data-flow analysis is an advanced technique performed by static analyzers that examine the flow of information throughout the entire program, including flow between methods. A unique characteristic of the data-flow analysis performed by the safety-critical type system is that it restricts its attention to the Java byte code one method at a time. This allows it to operate more efficiently, and it establishes a foundation upon which the results of static analysis can be fully deterministic, without false positives or false negatives, and without ambiguity from one vendor's implementation to the next. The following are the steps that comprise the data-flow analysis performed during enforcement of the safety-critical type system. Further detail on data-flow analysis techniques is available in reference [6].

1. The first step is to divide the method's code into independent basic blocks, with directed edges representing the possible control flows from one basic block to the next. A basic block is a sequence of instructions that is executed sequentially, without branches into or out of the code sequence.
2. For each basic block, identify the attribute information that is introduced by execution of that block (the *gen-set*) and the attributes that are superseded by execution of that block (the *kill-set*). In many analyses, the *gen-sets* and the *kill-sets* are described algorithmically rather than with discrete enumerations of elements.
3. Define the *join* functions for attribute information.
   a. For feed-forward attribute analysis, the *join* function is applied everywhere multiple control paths enter a given basic block from predecessor basic blocks.
   b. For feed-backward attribute analysis, the *join* function is applied everywhere multiple control paths leave a given basic block to the successor basic blocks.
4. For feed-forward attribute analysis, identify the initial set of attribute information based on safety-critical Java type annotations associated with the method's declaration.
5. For feed-backward attribute analysis, identify the initial set of attribute information based on safety-critical Java type annotations associated with the method's declaration.
6. Repeat until the inner loop executes without any further changes to previously computed attribute information. For each basic block in the method:
   a. For feed-forward attributes, compute the block's attribute information by joining the attribute information available from all predecessor blocks, removing the attribute information that is superseded by this block's *kill-set*, and adding the attribute information represented by this block's *gen-set*.
   b. For feed-backward attributes, compute the block's attribute information by merging the attribute information available from all successor basic blocks, removing the attribute information that is superseded by this block's *kill-set*, and adding the attribute information represented by this block's *gen-set*.
   c. For convenience in discussing execution of this algorithm, we speak of each basic block's *in-set* and *out-set*. The *in-set* represents the *join* of information flows into this basic block. The *out-set* represents the result of applying this block's *gen-set* and *kill-set* information to the *in-set* information. For feed-forward attribute analysis, the *in-set* information is associated with the start of the block and the *out-set* information is associated with the end of the block. For feed-backward attribute analysis, the *in-set* information is associated with the end of the block and the *out-set* information is associated with the beginning of the block.

"A unique characteristic of the data-flow analysis performed by the safety-critical type system is that it restricts its attention to the Java byte code one method at a time. This allows it to operate more efficiently ..."

Data-flow analysis problems guarantee termination by operating on a finite universe of possible attribute values. Thus, there is a maximum size for each *in-set* and *out-set*. Each iteration of the algorithm either leaves the *in-set* and *out-set* sizes unchanged, or at least one set expands. If the set sizes are unchanged, the algorithm has terminated.

Example Analysis of a Feed-Backward Attribute

Because of limitations built into the standard edition Java annotation system, it is not possible for developers to annotate their local variables. Thus, the safety-critical Java type system infers type information by examining how the variables are used within the method. If a local variable's value is ever assigned to a field variable or passed as an argument to a formal parameter that is declared @Captive Scoped, then the local variable must be treated as a *captive-scoped* variable. This situation is recognized by feed-backward analysis of data-flow, as illustrated by the following example.

Assume the following external method declarations:

```
// Within class java.lang.Object
@CallerAllocatedResult
@CaptiveScopedThis String toString();

// Within the same class as the following
// method
@Scoped static Object staticField;
static void print(@CaptiveScoped String
arg);
```

Now, consider analysis of the following method:

```
[1] @Scoped static Object staticField;
[2] static void method() {
[3]     Object anObject = new Object();
[4]     print(anObject.toString());
[5]     staticField = anObject;
[6] }
```

This method allocates an Object, invokes the toString() method on this Object, printing the resulting String, and then assigning the value of anObject to the staticField variable. I will describe the analysis that allows the compiler to determine that the String returned from the Object.toString() method invocation on line 4 is allocated in this method's local scope and discarded upon return from the method. The same analysis detects that the Object allocated at line 3 must be allocated within the corresponding ClassLoader scope because the assignment on line 5 makes this Object reachable from the class variable named staticField.

The first step is to divide this method into basic blocks, computing the *kill-set* and *gen-set* for each. The results of this step are represented in Figure 1.

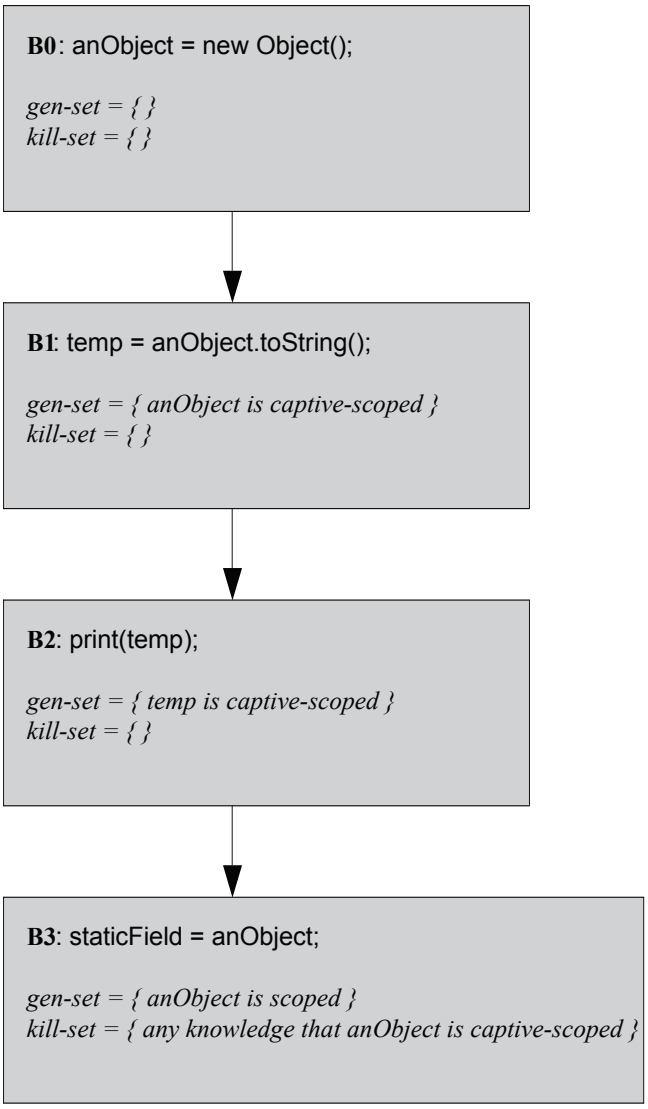


Figure 1: *Basic Blocks Related By Control-Flow Edges*

Based on the information available in block **B1**, it appears that anObject is *captive-scoped*. This is because we invoke the toString() method on anObject, and this method is declared @CaptiveScopedThis. Based on the information available within block **B2**, it appears that the synthesized temp variable is *captive-scoped*. Note that the print() method expects a single *captive-scoped* String argument. In block **B3**, we discover that anObject must be a scoped variable because it is assigned to staticField, which is declared @Scoped. The safety-critical type system treats *captive-scoped* as a specialization of *scoped*. If a given variable is treated in different contexts as both *captive-scoped* and *scoped*, it concludes that the variable must be *scoped*. This is similar to the notion of *widening* in traditional type systems, which allow a single-precision floating point value to be assigned to a double-precision floating point value, but would not allow a double-precision floating point value to be assigned to a single-precision variable without an explicit type coercion. For this reason, the *kill-set* for **B3** removes the *captive-scoped* association for anObject.

For this attribute analysis, the *join* function represents the most conservative classification indicated by all subsequent uses of the variable. This same *join* behavior is applied when propagating usage information through a basic block. If one future usage indicates a variable is *scoped* when another indicates that it is *captive-scoped*, we treat the variable as *scoped* because that is the more conservative treatment. If any future usage indicates that a variable is not *scoped*, then we must treat the variable as *unscoped* even if certain other future usages treat the variable as *scoped* or *captive-scoped*. The *unscoped* attribute is the most conservative. A variable with the *unscoped* attribute is only allowed to reference *immortal* objects, which are allocated in the heap rather than stack memory. The RTSJ identifies such objects as immortal because there is no garbage collection and no command to reclaim the memory for an object previously allocated within the heap.

There is no specific scoping information represented by the annotations associated with this method's return result. Assume that we process the basic blocks in ascending numeric order. Remember that since we are performing a feed-backward analysis, the *in-set* is associated with the end of each block, and the *out-set* is associated with the start. After the first iteration through the basic blocks, we have the following information:

B0:*in-set = { }*
out-set = { }

B1:*in-set = { }*
out-set = { anObject is captive-scoped }

B2:*in-set = { }*
out-set = { temp is captive-scoped }

B3:*in-set = { }*
out-set = { anObject is scoped }

Propagating all of the available data-flow information to all basic blocks requires several additional iterations. After the second iteration, the data-flow associated with each block is the following:

B0:*in-set = { anObject is captive-scoped }*
out-set = { anObject is captive-scoped }

B1:*in-set = { temp is captive-scoped }*

out-set = { anObject is captive-scoped, temp is captive-scoped }

B2: *in-set = { anObject is scoped }*
out-set = { temp is captive-scoped, anObject is scoped }

B3: *in-set = { }*
out-set = { anObject is scoped }

After the third iteration, we have:

B0: *in-set = { anObject is captive-scoped, temp is captive-scoped }*
out-set = { anObject is captive-scoped, temp is captive-scoped }

B1: *in-set = { anObject is scoped, temp is captive-scoped }*
out-set = { anObject is scoped, temp is captive-scoped }

B2: *in-set = { anObject is scoped }*
out-set = { temp is captive-scoped, anObject is scoped }

B3: *in-set = { }*
out-set = { anObject is scoped }

We need one more iteration to reach a fixed point. After, the fourth iteration, we discover the following:

B0: *in-set = { anObject is scoped, temp is captive-scoped }*
out-set = { anObject is scoped, temp is captive-scoped }

B1: *in-set = { anObject is scoped, temp is captive-scoped }*
out-set = { anObject is scoped, temp is captive-scoped }

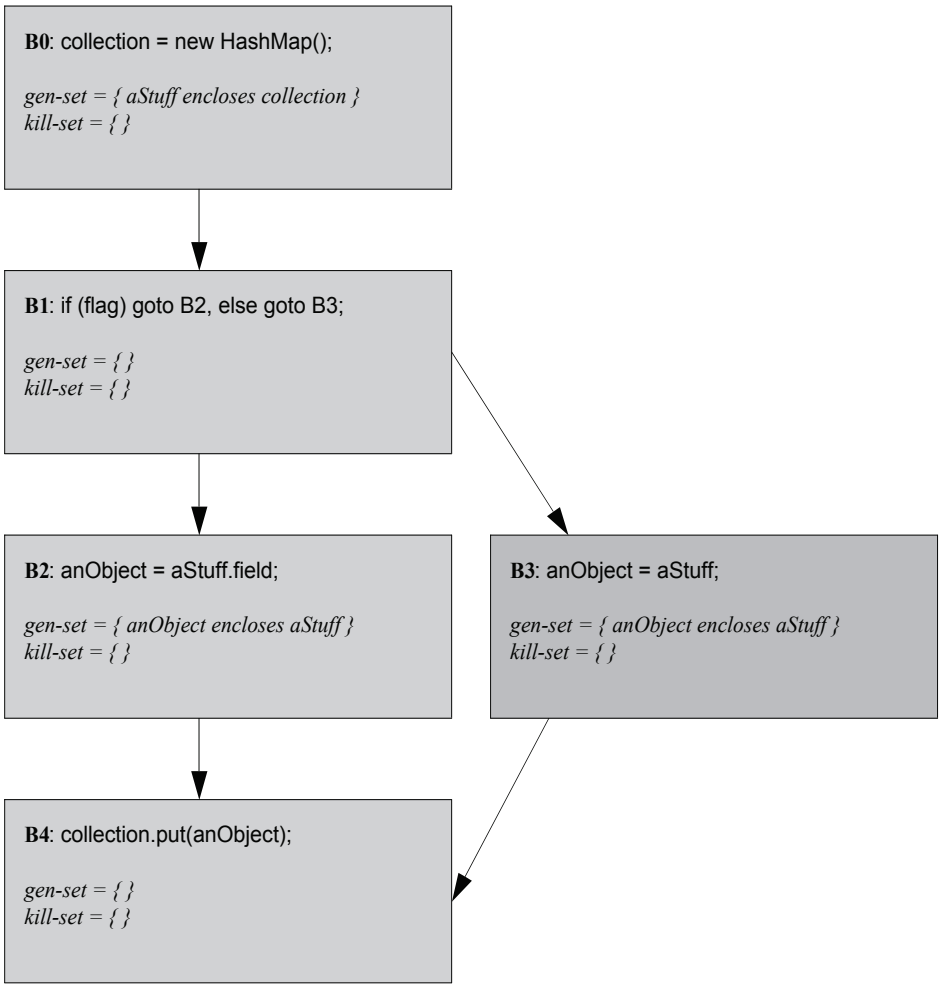
B2: *in-set = { anObject is scoped }*
out-set = { temp is captive-scoped, anObject is scoped }

B3: *in-set = { }*
out-set = { anObject is scoped }

If we were to iterate one more time through the basic blocks, there would be no further changes to our calculations of *in-sets* and *out-sets*. Thus, the data-flow analysis is done.

The safety-critical type system uses this information to determine that the *captive-scoped* temp String created implicitly at line 4 can be allocated in this method's local stack frame memory. Likewise, it determines that the *scoped* Object created at line 3 must be allocated in this class' ClassLoader scope because it must be referenced from one of the class's static variables.

Figure 2: *Basic Blocks Related By Control-Flow Edges*



Example Analysis of a Feed-Forward Attribute

The question of whether one object resides in a scope that is nested external to the scope that holds another object is answered by a feed-forward analysis of data flow. We use the term *encloses* to describe this notion. For purposes of illustration, assume that the feed-backward analysis has already determined that the new HashMap object allocated at line 3 is to be allocated in this method's local stack frame memory. That knowledge becomes an input to the analysis described in the following program fragment:

```
[1] static void buildCollection(Boolean
    Flag, @Scoped Stuff aStuff) {
[2]     Object anObject;
[3]     HashSet collection = new
        HashMap();
[4]     if (flag)
[5]         an Object = aStuff.field;
[6]     else
[7]         an Object = aStuff;
[8]     collection.add(anObject);
[9] }
```

In this code, the new HashSet object created at line 3 is allocated in this method's local stack frame. Depending on the value of the incoming flag argument, we insert into the HashSet either a reference to the object named by the aStuff argument, or the object named by the field variable associated with the aStuff argument. The Stuff declaration (not shown) defines an instance field named field of type @Scoped Object. The HashSet class is declared with the @ReentrantScope annotation, and its add() method declares its single argument to be @Scoped. Given these declarations, the safety-critical Java compiler is required to prove that the argument to the HashSet.add() method resides in a scope that encloses the scope of the HashSet object itself. This is required because the add() method is going to create a reference from the HashSet object to the Object that is inserted into the set. The remainder of this section describes the analysis performed by the compiler to establish this relationship.

The first step is to divide this method

into basic blocks, computing the *kill-set* and *gen-set* for each. The output of this step is represented in Figure 2.

Note that block **B1** generates the knowledge that aStuff encloses collection. This is because any incoming *scoped* arguments necessarily reside in scopes that surround all locally allocated objects. Also note that block **B2** generates the knowledge that anObject encloses aStuff. This is because any field fetched from an object must necessarily reside in a scope that is visible from the object (i.e., that encloses the object). Otherwise, the code that originally assigned the field would have been disallowed. Similarly, block **B3** generates the same knowledge because, by definition, every scope encloses itself.

For this analysis, the *join* function computed for node N is the intersection of all *out-sets* associated with the predecessors of node N. In other words, the only information we know about the enclosure relationships between objects is information known on all incoming paths. If the information is only known upon exit from one of several predecessors to this block, the information may be true—but is not necessarily true upon entry to this particular basic block.

There is no specific object relationship information represented by the annotations associated with this method's incoming arguments; consequently, the *in-set* for block **B0** is empty. Assume that we process the basic blocks in ascending numeric order. After the first iteration through the basic blocks, we have the following information:

B0: *in-set = { }*
out-set = { aStuff encloses collection }

B1: *in-set = { aStuff encloses collection }*
out-set = { aStuff encloses collection }

B2: *in-set = { aStuff encloses collection }*
out-set = { aStuff encloses collection, anObject encloses aStuff }

B3: *in-set = { aStuff encloses collection }*
out-set = { aStuff encloses collection, anObject encloses aStuff }

B4: *in-set = { aStuff encloses collection, anObject encloses aStuff }*
out-set = { aStuff encloses collection, anObject encloses aStuff }

If we iterate one more time through the basic blocks, there will be no further changes to our calculations of *in-sets* and *out-sets*. Thus, the data-flow analysis is done.

Block **B4** consists of the statement collection.put(anObject). The annotated API description for HashMap.put(), which is not shown, requires for every invocation that the argument to put() enclose the HashMap object that is the target of the put() invocation. The type system applies the transitive property on the relationships available within the *in-set* for block **B4**, thereby validating that the requirement for relative nesting of incoming arguments is satisfied. In other words, the safety-critical type system has proven that the invocation of HashMap.put() at line 8 is a legal invocation.

Conclusion

Standard edition Java provides the infrastructure that is required to augment the type system to speak of static properties that are relevant to safety-critical development. The augmented type system can be implemented by tools that run in combination with standard edition Java development tools by analyzing byte code. The benefits of this approach include leveraging mainstream economies of scale for much of the software and expertise associated with safety-critical development, exploiting the improved programming language features of Java in comparison with legacy languages like Ada, C, and C++, and providing an enhanced type system that focuses on properties of concern to safety-critical developers. All of this translates to reduced costs, improved longevity, and increased functionality for safety-critical software.◆

References

1. RTCA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification." 1 Dec., 1992.
2. Besnard, J., et al., Eds. Developing Software for Safety Critical Systems. Institute of Electrical and Electronics Engineers, July 1998.
3. Gosling, James, et al. The Java Language Specification. 3rd ed. Prentice Hall PTR, June 2005.
4. JSR-302: Safety-Critical Java Technology. Java Community Process <http://jcp.org/en/jsr/detail?id=302>.
5. Bollella, Gregory, et al. The Real-Time Specification for Java. Addison-Wesley Longman, 2000.
6. Aho, Alfred V., et al. Compilers: Principles, Techniques, and Tools. 2nd ed. Addison Wesley, Oct. 2007.
7. Nilsen, Kelvin. A Type System to Assure Scope Safety Within Safety-Critical Java Modules. Proc. of the 4th Annual Workshop on Java Technologies for Real-Time and Embedded Systems, ACM. Paris, France: 11-13 Oct. 2006.
8. The PERC Pico User Manual. Aonix. 19 Apr. 2008 <http://research.aonix.com/jsc/pico-manual.4-19-08. pdf>.

Note

1. The JSR-302 expert group of the Java Community Process is developing a specification for safety-critical development with the Java language. The team of experts, including the author of this article, has identified a number of static properties that should be assured for any Java software deployed in safety-critical systems, but has chosen not to standardize the mechanisms by which these properties are assured. This article describes the annotation system implemented for this purpose in a commercial product offered by the author's company [7, 8]. Because of space limitations, this article provides only an overview of the complete annotation system.

About the Author

Kelvin Nilsen, Ph.D., is the chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including ObjectAda compilers, development environment, libraries, and commercial off-the-shelf safety certification support. Nilsen's seminal research on the topic of real-time Java led to the founding of NewMonics (subsequently purchased by Aonix in 2003), a leader in advanced real-time virtual machine technologies to support real-time execution of Java programs. Nilsen has a bachelor's degree in physics from Brigham Young University as well as master's and doctorate degrees in computer science from the University of Arizona.

Aonix
5930 Cornerstone Court West
STE 250
San Diego, CA 92121
Phone: (801) 756-4821
Fax: (801) 756-4839
E-mail: kelvin@aonix.com