Harmonizing Alternative Approaches to Safety-Critical Development with Java

Kelvin Nilsen, Chief Technology Officer Java, Atego Systems

What is “safety-critical Java”?

- Ravenscar Java? (Kwon, Wellings, King: 2002)
- A Profile for Safety-Critical Java? (Schoeberl, Sondegaard, Thomsen, Ravn: 2007)
- Perc Pico? (Aonix, 2007)
- JSR 302? (2012?)
  - In its default configuration?
  - Or with optional annotations?
- Or is it just careful application of traditional (main-stream) Java?

The Problem

- Each alternative approach to safety-critical Java offers different benefits and tradeoffs
- Most approaches are assumed to be universal
  - ➢ The entire application must be implemented according to the constraints of the respective safety-critical Java "definition"
- Question: How to safely integrate software components in baseline JSR-302, annotated JSR-302, Perc Pico, and traditional Java?
  - ➢ The focus is on scope safety (one of the harder problems)
  - ➢ Ignores, for example, schedulability analysis
  - ➢ The approaches probably generalize to other safety-critical Java approaches

JSR-302 Rules

- Temporary objects are allocated in nested scopes (as in RTSJ)
  - The scope model is simpler than RTSJ: avoids scope cycles, fragmentation of backing stores
- Programmers explicitly create, size, and enter scopes (as in RTSJ)
- Reference assignments must be checked to assure that outer-nested objects never refer to inner-nested objects
- JVM vendors and tool providers are “encouraged” to support techniques that allow programmers to guarantee the absence of IllegalAssignmentError exceptions
  - Aside: This probably requires significant restrictions on the structure of source code, and may impose additional “non-standard” annotations

Annotated JSR-302 Rules

- The run-time model is the same as unannotated JSR-302, but static properties declared in annotations are checked at compile time
- The draft JSR-302 specification includes a set of optional annotations
  - Certain classes can be declared to always reside in specific named scopes
  - Unannotated classes are assumed to reside in "current scope"
- "Shortcomings" of the annotated JSR-302 protocol
  - Explicit management of first-class scopes is tedious and error prone
  - Difficult to reuse classes within multiple distinct scopes; may require class replication
  - Hard to implement certain common programming patterns

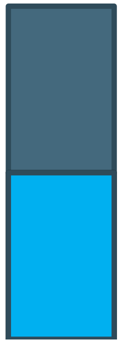
Perc Pico Rules

- Annotations are placed on reference variables rather than classes
  - Default (unannotated) variable refers only to immortal objects
  - @Scoped variable may refer to a “stack-allocated” object
  - @CaptiveScoped variable does not “escape” (is not assigned to a field of a more outer-nested object)
- “Scopes” are implicit rather than first-class objects
- The Perc Pico compiler determines the scope for each allocation based on static analysis of object lifetimes (as guided by annotations)
- Perc Pico annotations also enable automatic calculation of scope sizes
- “Shortcomings” of Perc Pico
  - Not an “international standard”
  - Too “magical”; tendency by programmers to forget that they are still responsible for managing scopes

A Motivating Example

- Initialization of a newly allocated “mission” object requires a computation that depends on instantiation of temporary objects
- Intended stack usage:

Assume stack grows downward



Allocate memory for the “mission”

Return from “constructor”, reclaiming memory of temporary objects

C implementation

```
typedef struct {
    char *digits;
    unsigned char avail_digits;
    unsigned char used_digits;
    unsigned char sign;
} BigInteger;

typedef struct { BigInteger crypto_key; } TheMission;
TheMission tm;
char digits[40];

void initializeMission() {
    BigInteger t1, t2; char digits1[20], digits2[20];
    struct timespec now; longlong seed;

    clock_gettime(CLOCK_REALTIME, &now);
    seed = now.tv_nsec + (longlong) now.tv_sec * 1000000
    tm.crypto_key.digits = digits; tm.avail_digits = 40;
    t1.digits = digits1; t1.avail_digits = 20;
    t2.digits = digits2; t2.avail_digits = 20;
    fillRandomBigInteger(&t1, 128, 24, &seed);
    fillRandomBigInteger(&t2, 128, 24, &seed);
    multiplyBigInteger(&t1, &t2, &(TheMission.crypto_key));
}
```

Lacks encapsulation: digits and other fields are exposed to outside; construction is user's responsibility.

Pointers to stack-allocated objects are shared without any safety net.

Annotated JSR 302 Solution (1 of 3)

```
// Not shown: @Scope(name=“TM”) annotation on definition of TheMission class

@SCJRestricted(INITIALIZATION)
public TheMission() {
    CalculateCryptoKey calculator = new CalculateCryptoKey(this);
    SizeEstimator z = new SizeEstimator();
    z.reserve(AbsoluteTime.class, 1);
    z.reserve(Random.class, 1);
    z.reserve(BigInteger.class, 3);
    z.reserveArray(20, byte.class);
    z.reserveArray(20, byte.class);
    z.reserveArray(40, byte.class);
    z.reserve(CalculateCryptoKey.AssignCryptoKey.class, 1);
    ((ManagedMemory) MemoryArea.getMemoryArea(this)).
        enterPrivateMemory(z.getEstimate(), calculator);
}
```

Annotated JSR 302 Solution (2 of 3)

```
@Scope("TM") @SCJAllowed(members=true)
static class CalculateCryptoKey implements Runnable {
    TrainMission tm;

     @SCJRestricted(INITIALIZATION) public CalculateCryptoKey(TheMission the_mission) {
        tm = the_mission;
    }

    @DefineScope(name="TM.0", parent="TM“) @Scope("TM.0") @SCJAllowed(members=true)
    static class AssignCryptoKey implements Runnable {
        TheMission tm; // resides in scope “TM”
        BigInteger bi; // resides in scope “TM.0”
        AssignCryptoKey(TrainMission tm, BigInteger bi) {
            this.tm = tm; this.bi = bi;
        }

        @RunsIn("TM“)
        public void run() {     // copy bi into the "TM" scope (from the "TM.0" scope)
            tm.crypto_key = bi.multiply(BigInteger.ONE);
        }
    }
```

Annotated JSR 302 Solution (3 of 3)

```
    @RunsIn("TM.0“)
    public void run() {
        AbsoluteTime now = javax.realtime.Clock.getRealtimeClock().getTime();
        Random r = new Random(now.getMilliseconds());
        BigInteger t1, t2, t3;
        t1 = new BigInteger(128, 24, r);
        t2 = new BigInteger(128, 24, r);
        t3 = t1.multiply(t2);
        AssignCryptoKey assigner = new AssignCryptoKey(tm, t3);
        MemoryArea.getMemoryArea(tm).executeInArea(assigner);
    }
}
```

Biggest problem: too much code dedicated to scope management.

Can’t see the forest for the trees.

Annotated JSR 302 Solution (1 of 3)

```
// Not shown: @Scope(name=“TM”) annotation on d

@SCJRestricted(INITIALIZATION)
public TheMission() {
    CalculateCryptoKey calculator = new CalculateC
    SizeEstimator z = new SizeEstimator();
    z.reserve(AbsoluteTime.class, 1);
    z.reserve(Random.class, 1);
    z.reserve(BigInteger.class, 3);
    z.reserveArray(20, byte.class);
    z.reserveArray(20, byte.class);
    z.reserveArray(40, byte.class);
    z.reserve(CalculateCryptoKey.AssignCryptoKey
    ((ManagedMemory) MemoryArea.getMemoryAre
        enterPrivateMemory(z.getEstimate(), calcula

}
```

Another problem: poor encapsulation.

The user of BigInteger needs to know about the byte array used in its implementation.

Also, would prefer to perform these computations at compile time.

And would rather that the SizeEstimator object z reside in a temporary scope rather than in the “TM” scope..

Annotated JSR 302 Solution (2 of 3)

```
@SCJRestricted(INITIALIZATION) @Scope("TM") @SCJAllowed(members=true)
static class CalculateCryptoKey implements Runnable {
    TrainMission tm;

    public CalculateCryptoKey(T
        tm = the_mission;
    }

    @DefineScope(name="TM.0"                              @SCJAllowed(members=true)
    static class AssignCryptoKey
        TheMission tm; // resides
        BigInteger bi; // resides in
        AssignCryptoKey(TrainMission tm, BigInteger bi) {
            this.tm = tm; this.bi = bi;
        }

        @RunsIn("TM“)
        public void run() {      // copy bi into the "TM" scope (from the "TM.0" scope)
            tm.crypto_key = bi.multiply(BigInteger.ONE);
        }
    }
```

And to be completely honest, this code doesn’t even compile right now.

I’m trusting that we’ll be able to “fix” the JSR-302 spec to allow something like this before we’re done.

Perc Pico Solution

```
@StaticAnalyzable
public TheMission() {
                                    e now;
                                    dom();
                                    1, t2;
   now = javax.realtime.Clock.getRealtimeClock().getTime();
   Random r = new Random(now.getMilliseconds());
   assert StaticLimit.InvocationMode("Digits=20");
   t1 = new BigInteger(128, 24, r);
   assert StaticLimit.InvocationMode("Dig
   t2 = new BigInteger(128, 24, r);
   assert StaticLimit.InvocationMode("Dig
   this.crypto_key = t1.multiply(t2);
}
```

Annotation denotes that the compiler will determine the sizes of relevant scopes: a private scope and a constructed scope.

The declaration of BigInteger's constructor requires a pre-condition assertion to size its internal memory whenever invoked from a "static analyzable" context.

Allocations assigned to captive-scoped variables are always taken from the private scope.

Within a constructor, allocations assigned to @Scoped fields are taken from the constructed scope.

Why do I care about this?

- My perception: primary appeals of Java are high-level abstraction, ease of programming, modular composition, software reuse
- There are (at least) three different ways to structure safety-critical Java code, and I am not satisfied that JSR-302 is sufficiently robust to replace all the others
- Atego will support the JSR-302 standard (if we can figure out what it is)
- But we want to also provide customers with the option to use something that we believe to be "much better"
  - Not about vendor lock-in
  - Need 10-fold improvement to disrupt the status quo

Mixing Semantic Models

- How can Atego provide full compliance with JSR-302 without abandoning the strengths of its current Perc Pico technologies?
- The general idea: provide the capability of mixing code implemented according to the distinct semantic models
- But each model has radically different approaches to scope safety
  - JSR-302 assumes no way to instantiate particular classes in any scope other than named scope
  - Perc Pico assumes @CaptiveScoped variables are never assigned to the fields of more outer-nested objects
  - It is not “safe” to allow methods implemented according to one semantic model to invoke methods implemented according to the other

The Relevant Mission API

- Infrastructure invokes a MissionSequencer’s getNextMission() method, which is implemented by the application developer.
- The infrastructure invokes the mission’s initialize() method (application code) to instantiate and “register” the schedulables.
- Upon return, the infastructure starts up the registered schedulables.
- Infrastructure waits for someone to invoke requestTermination(), arranges to terminate the registered schedulables, waits for them to complete current releases.
- Infrastructure invokes the mission’s cleanUp() method.
- How does a mission communicate with outer-nested missions?
  - Shared buffers are passed in to the mission’s constructor by the mission sequencer

Motivating Example

- Suppose an inner mission provides communication services for its sibling missions
- A CommandBuffer object, residing in outer-nested mission memory, provides communication with clients
  - The read() method notifies the communication mission that work is available and waits for the read data to be returned.
  - The write() method notifies the communication mission that work is available and waits for the buffered write data to be transferred.
  - The getWork() method blocks the communication mission’s server thread until a read() or write() request is received.

Suppose Inner-Nested Mission Implemented in Perc Pico

```
public interface CommandBuffer {
    // methods invoked by client threa
    public int read(int socket_id, @Cap
    public void write(int socket_id, @C                         t num_bytes);

    // methods invoked by server threa
    public int getWork();               // returns 1 for read, 2 for write
    @Scoped byte[] getBuffer();
    int num_bytes;                      // how many bytes to read or write
    public void finishWrite();
    public void finishRead(int bytes_fetched);
}
```

Suppose the outer-nested mission is annotated JSR-302.

These annotations are not meaningful in that context.

In Outer-Nested Mission, CommandBuffer Has Different API

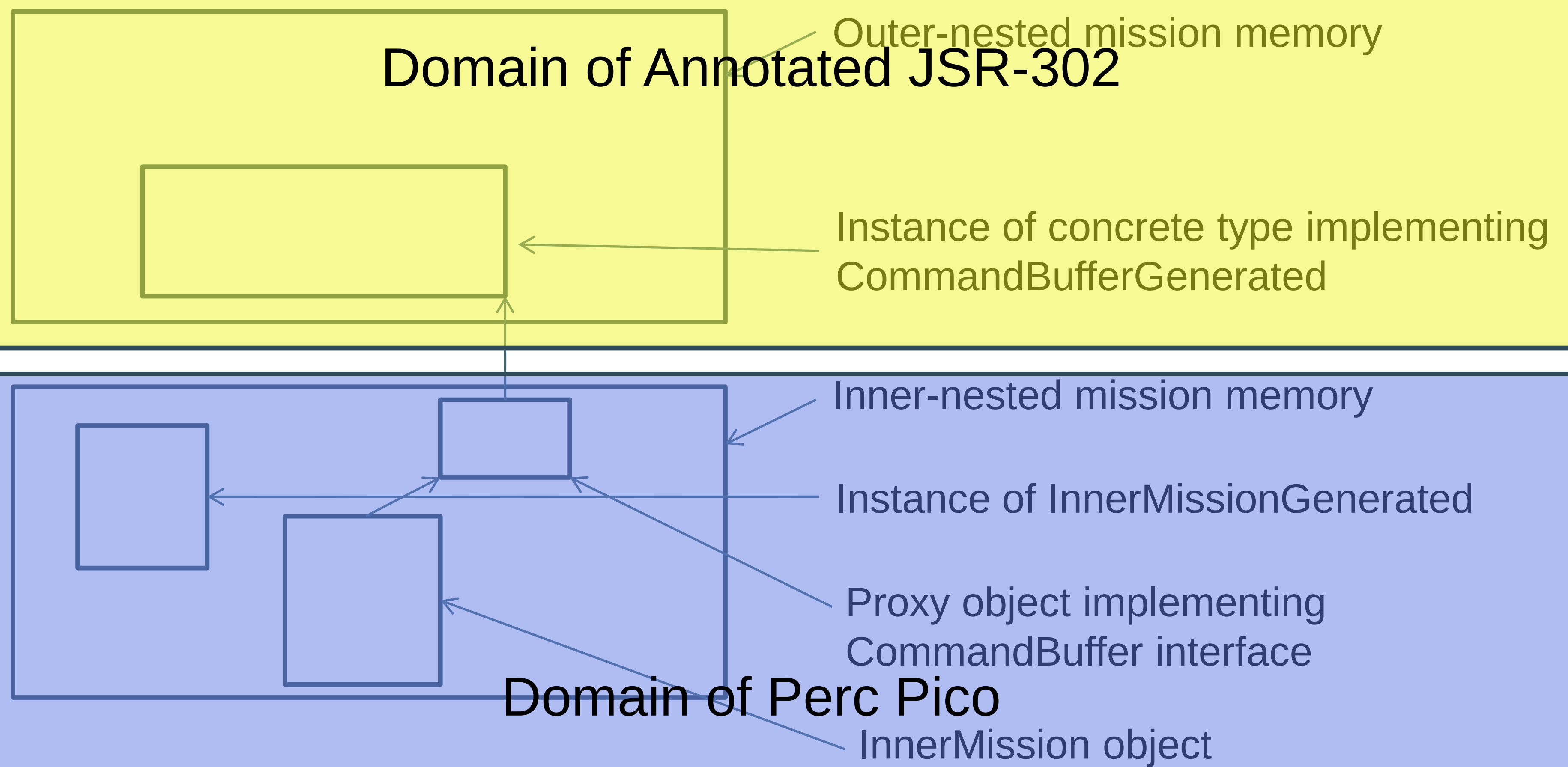
```
public interface CommandBufferGenerated {
    // methods invoked by client threads in sibling missions
    public int read(int socket_id, @Scope(UNKNOWN) byte[]);
    public void write(int socket_id,
                      @Scope(UNKNOWN) byte[], int num_bytes);

    // methods invoked by server thread
    public int getWork();            // returns 1 for read, 2 for write
    @Scope(UNKNOWN) byte[] getBuffer();
    int getNumBytes();               // how many bytes to read or write
    int getSocketId();
    public void finishWrite();
    public void finishRead(int bytes_fetched);
}
```

Wrapper for Inner-Nested Mission

```
public class InnerMissionGenerated extends Mission {
    public
    InnerMissionGenerated(@Scope(UNKNOWN) CommandBufferGenerated) {
        // Auto-generated infrastructure magic here to instantiate
        // the real InnerMission
    }
    // the following method implementations are similarly “magical”
    public void initialize() {
    }
    public void cleanUp() {
    }
    public void requestTermination() {
    }
    public long missionMemorySize() {
    }
}
```

Memory Organization


Domain of Annotated JSR-302
Outer-nested mission memory
Instance of concrete type implementing CommandBufferGenerated
Inner-nested mission memory
Instance of InnerMissionGenerated
Proxy object implementing CommandBuffer interface
Domain of Perc Pico
InnerMission object

Some Disclaimers

- Some uncertainty:
  - What will be the final form of JSR-302 annotations?
  - How will Perc Pico evolve as a result of integration with JSR-302?
- This is an unimplemented conceptual design only at the current time
- This is not a promise that Atego will implement the contemplated capabilities
- If implemented, the final form of the technology may differ from today's description

Summary

- Alternative approaches to safety-critical Java development offer different benefits
- The JSR-302 Mission provides an encapsulation boundary to isolate alternative semantic models
- The proposed integration methodology consists of:
  - Specifying inner-nested mission APIs in terms of interfaces only
  - Submitting the inner-nested mission API to a special proxy generation tool to generate proxy implementations that manage the boundary between alternative semantic models
- Also discussed in paper (but not here): integration with traditional Java