

Proposed Draft Extensions for Mission-Critical Java

Kelvin Nilsen, Ph.D., CTO, Aonix North America

The Open Group's Real-Time and Embedded Forum has been working for the past several years to define a standard for development of hard real-time safety-critical software using the Java programming language. Various participants in this effort have wondered whether the proposed safety-critical Java standard has any relevance to the development of hard real-time mission-critical software. At July's Boston meetings, it was requested of Kelvin Nilsen that he provide a description of his ideas for combining hard real-time code written using a generalization of the safety-critical standard with traditional Java components running in soft real-time or non-real-time virtual machine. This document is the response to that request.

During over two decades of providing developer tool support for safety-critical and mission-critical development, Aonix has recognized a common development pattern for the development of hard real-time mission-critical software components such as missile guidance, radars, sonar, and rocket launch systems. Developers of hard real-time mission-critical software often prefer to use many of the same core software components that are used by safety-critical developers because they know they can trust that these components have satisfied stringent certification requirements. These developers of mission-critical software usually choose not to license the certification artifacts that are available from Aonix to facilitate DO-178B certification and usually do not follow all of the very restrictive development guidelines that are typically imposed on developers of safety-critical software.

In general, we have found that both the safety-critical and the mission-critical communities encourage this practice. The safety-critical community feels that it is essential that the tools and components they use be shared with a much broader community than the very small niche of safety-critical developers. This is because they do not want to pay the full cost of developing and maintaining software that is so specialized that it only serves their needs. They find that tools which serve a broader audience often provide more functionality and more robust operation for lower costs. This is one reason that many safety-critical development efforts have recently migrated away from Ada to C and C++, even though Ada is clearly a superior technical solution for the development of safety-critical code.

Mission-critical developers are attracted to safety-critical development environments and software components provided that:

- They can use this software without paying a premium for safety certification, and
- The tools do not overly restrict their ability to address the special needs of mission-critical development.

At Aonix, we have addressed the first of these two issues by pricing our safety-critical development tools and components competitively with mission-critical development offerings available from other suppliers. For a separate licensing fee, we provide the certification artifacts (test data, inspection reports, traceability analysis, and audit trails) that are required to pass a safety certification audit.

The second of these issues is addressed by supplementing the very restrictive capabilities of a safety-certified run-time environment with enhancements that more effectively address the more general needs of mis-

sion-critical development. Some specific capabilities that are often required by mission-critical developers, but forbidden in safety-certified projects include:

- Dynamic memory allocation and deallocation
- Dynamic loading and unloading of functional components
- Asynchronous transfer of control, timeouts, and abortion of running components
- Access to larger and more complex software components, such as network communication stacks and file systems. These components would often be omitted from safety-certified systems because obtaining safety certification of these components would be cost prohibitive.
- Communication and coordination with other software systems that have not been safety certified.

This proposal considers an extension to the draft safety-critical Java standard to make it possible to very efficiently and reliably combine hard real-time safety-critical components with traditional Java components running on either a traditional Java virtual machine such as Hot Spot or a real-time virtual machine such as PERC or Jamaica.

In these sorts of mission-critical configurations, hard real-time components can be implemented using the safety-critical Java conventions. These hard real-time components will typically run with footprint and throughput efficiency that is very close (within 5-10%) of optimized C. This represents a three-fold improvement over typical optimized Java performance and footprint. Note that when “safety-critical components” are combined with a traditional Java virtual machine, the combined technology is no longer safety certified. This is because the Java virtual machine and standard libraries are much too large and complex to be certified to the demanding standards of DO-178B. However, there are many important mission-critical needs that can be addressed by this configuration, such as:

- Portable and very efficient device drivers (possibly, but not necessarily, having hard real-time constraints) can be implemented using the safety-critical Java programming notations.
- Compared with the use of JNI, interfaces to legacy (“native”) components written in other languages are much more efficient and much safer if implemented using the safety-critical Java environment as an intermediary between traditional Java and the native code.
- Performance-critical code such as Fourier analysis and matrix manipulation can be provided much more efficiently as safety-critical Java components than as traditional Java code or as legacy code interfaced to Java through JNI.

Selective Sharing of Control with Traditional Java Components

The recommended approach for providing efficient and reliable integration of hard real-time components with traditional Java components makes use of a restrictive form of object sharing between the hard real-time and traditional Java domains. The shared objects always reside in the hard real-time domain and do not participate in garbage collection. Since these are hard real-time objects, they will never be subject to relocation. This greatly simplifies the implementation and improves execution efficiency.

We describe this object sharing as “restrictive” because we restrict access to the hard real-time object from the traditional Java domain. In particular, the traditional Java domain can not see any of the instance or static variables associated with the object. Furthermore, it cannot see the object’s regular methods. It can only see methods that have been specially designated as traditional Java methods. These traditional Java methods, which cannot be seen or invoked by the safety-critical threads, are analogous to operating system entry points for application software. In this scenario, writing safety-critical Java code is similar to making

modifications to an operating system kernel. As with traditional operating system design, greater trust is placed in the implementers of the lower-level (operating system) software, and great care is taken to ensure that errors or malicious intent of application software not compromise the lower level components. In traditional operating system design, invocation of kernel services generally crosses a memory protection barrier, and hardware memory management units assure that application code cannot see or modify kernel code and data structures. The restrictive object sharing architecture supports the same abstraction guarantees, but it does so using static byte-code verification techniques which allow much more efficient integration of the hard real-time and non-real-time software. The performance benefits of this sort of architecture have been proven, for example, in studies conducted by Calton Pu on the Synthesis Kernel.

For examples of the `@TraditionalJavaMethod` annotation, see lines 9 and 14 of Figure 1. This simple

```
[1] package samples;
[2]
[3] import javax.jmc.TraditionalJavaMethod;
[4]
[5] public class Thermostat {
[6]     private float measured_temperature;
[7]     private float desired_temperature;
[8]
[9]     public final @TraditionalJavaMethod @ScopedPure synchronized
[10]         float getTemperature() {
[11]         return measured_temperature;
[12]     }
[13]
[14]     public final @TraditionalJavaMethod @ScopedPure synchronized
[15]         void setTemperature(float f) {
[16]         desired_temperature = f;
[17]     }
[18]
[19]     public final synchronized void updateTemperature(float f) {
[20]         measured_temperature = f;
[21]     }
[22]
[23]     public final synchronized float checkThermostat() {
[24]         return desired_temperature;
[25]     }
[26] }
```

Figure 1: Annotated Source code for class `Thermostat`

module represents an interface between a thermostat module, written as a hard real-time component, and traditional Java code which might need the ability to set and get the temperature.

Sample Implementation of Device Driver

In this section, we describe a simple interrupt-handling device driver software component, written entirely in Java. Figure 2 provides the declarations of class constants and instance variables for the interrupt handler. Figure 3 presents the implementations of the three hard real-time methods relevant to implementation of the interrupt handler.

```
[1] package samples.device;
[2]
[3] import javax.jsc.*;
[4]
[5] class Interrupt extends javax.jsc.InterruptHandler implements javax.jsc.Atomic {
[6]
[7]     static final int INTERRUPT_PRIORITY = 128;
[8]     static final int InterruptNo = 5;
[9]     static final int DataPortAddr = 0x3f0;
[10]    static final int StatusPortAddr = 0x3f4;
[11]    static final int ControlPortAddr = 0x3f8;
[12]    static final byte RcvReady = (byte) 0x08;
[13]    static final byte ResetPort = (byte) 0xff;
[14]    static final int BUFFER_LENGTH = 1024;
[15]
[16]    final public byte shared_buffers[][];
[17]
[18]    IOPort8IO data_xfer;
[19]    IOPort8I status;
[20]    IOPort8O control;
[21]
[22]    public int write_buffer_count;
[23]    public int read_buffer_count;
[24]    public int write_buffer_index;
[25]    boolean reader_waiting;
[26]
```

Figure 2: Constant and Instant Variable Declarations for **Interrupt** class

The `ceilingPriority()` method shown on lines 1 through 3 of Figure 3 is required because this class implements the `javax.jsc.Atomic` interface. For classes that implement this interface, programmers are required to provide this method in order to establish a syntactic marker within the body of the class that can be used to establish static properties regarding the object's synchronization behavior. When an instance of this class is constructed, the synchronization behavior is governed by the default monitor control policy as with traditional RTSJ, with a run-time check at the time this object is instantiated to ensure that the dynamic properties are consistent with the declared static properties. Within the safety-critical profile, programmers agree to follow the convention that each instantiation of a class will use a monitor control policy that is consistent with the syntactic markers represented by the class's implementation of the `Atomic` interface and the `ceilingPriority()` method. Consistency checking is performed at the time the class is instantiated.

The constructor shown on lines 5 through 26 of Figure 3 instantiates the three I/O ports required for operation of the interrupt handler. For any given hardware configuration, certain ranges of memory and I/O address space will be eligible to be treated as I/O ports that are accessible from hard real-time Java components. Range checking to assure that this hard real-time Java component has permission to access the requested I/O ports is performed at the time these I/O ports are instantiated. Once instantiated, no additional checking is required when reading or writing the I/O ports. Besides instantiating the necessary I/O ports, the `Interrupt` constructor also allocates memory to represent a pair of input buffers. Note that this buffer pair is allocated in `ImmortalMemory` because the `shared_buffers` variable is not declared with the `@Scoped` attribute.

```

[1] public @Ceiling(INTERRUPT_PRIORITY) int ceilingPriority() {
[2]     return INTERRUPT_PRIORITY;
[3] }
[4]
[5] public Interrupt() throws EmbeddedConflictException {
[6]     super(new PriorityParameters(INTERRUPT_PRIORITY),
[7]           // no deadline and no miss handler
[8]           new AperiodicParameters(null, null),
[9]           // allow no allocation in default context or in immortal area for the thread
[10]          new MemoryParameters(OL, OL),
[11]          // treat this as a daemon thread
[12]          true);
[13]
[14]     // The following three createIOPort() requests throw EmbeddedConflictException
[15]     // if permission is not granted to create the requested IO Ports. Arguments are:
[16]     //     address, memory-mapped?, port width, readable?, writeable?
[17]     data_xfer = (IOPort8IO) IOPort.createIOPort(DataPortAddr, true, 8, true, true);
[18]     status = (IOPort8I) IOPort.createIOPort(StatusPortAddr, true, 8, true, false);
[19]     control = (IOPort8O) IOPort.createIOPort(ControlPortAddr, true, 8, false, true);
[20]
[21]     write_buffer_count = read_buffer_count = write_buffer_index = 0;
[22]
[23]     shared_buffers = new byte [2][];
[24]     shared_buffers[0] = new byte[BUFFER_LENGTH];
[25]     shared_buffers[1] = new byte[BUFFER_LENGTH];
[26] }
[27]
[28] // This is the interrupt handling code
[29] public synchronized void handleAsyncEvent() {
[30]     byte b;
[31]
[32]     if (status.readByte() == RcvReady) {
[33]         b = data_xfer.readByte();
[34]         if (write_buffer_count < BUFFER_LENGTH)
[35]             shared_buffers[write_buffer_index][write_buffer_count++] = b;
[36]         // else, buffer overflow is ignored
[37]
[38]         if (reader_waiting)
[39]             this.notify();
[40]     }
[41]     control.writeByte(ResetPort);
[42] }

```

Figure 3: Hard Real-Time Methods for Interrupt Handling Code

Lines 29 through 42 of Figure 3 provide the actual interrupt handling code. This method is declared as `synchronized` because, while it is running, all other interrupts of equal or lower priority are forbidden from running. Because this class is declared to implement the `Atomic` interface, the safety-critical Java byte-code verifier assures that the body of every `synchronized` method is execution-time bounded. This restricts the set of services that can be invoked from within an interrupt handler. The byte-code verifier prohibits interrupt handling code from invoking services that might block. The byte-code verifier also assures that

only objects that implement the `Atomic` interface can set their ceiling priority to ranges that correspond to hardware-dispatched interrupt handling.

In Figure 4, we present the two traditional Java methods that serve to enable efficient streaming of bytes from the hard real-time interrupt handler to the traditional Java domain. Lines 1 through 15 provide the implementation of the `getReadBuffer()` method. This method returns a reference to a hard real-time array of bytes that were fetched by the interrupt handler from the I/O port. Note that the reference value returned from this method will be represented within the traditional Java domain by a proxy object of type `javax.rtpoxy.ByteArray`. Note also that the traditional Java thread will block within this method until at least one byte is available in the shared buffer. Lines 17 through 19 provide the implementation of `readBufferLength()`, which represents the number of bytes which can be fetched from the buffer returned from the `getReadBuffer()` method.

```
[1]    public @TraditionalJavaMethod @ScopedPure final synchronized
[2]           byte[] getReadBuffer() throws java.lang.InterruptedException {
[3]        byte read_buffer[];
[4]
[5]        while (write_buffer_count == 0) {
[6]            reader_waiting = true;
[7]            this.wait();
[8]        }
[9]        reader_waiting = false;
[10]       read_buffer = shared_buffers[write_buffer_index];
[11]       write_buffer_index = (write_buffer_index + 1) % 2;
[12]       read_buffer_count = write_buffer_count;
[13]       write_buffer_count = 0;
[14]       return read_buffer;
[15]    }
[16]
[17]    public @TraditionalJavaMethod @ScopedPure final int readBufferLength() {
[18]        return read_buffer_count;
[19]    }
[20] }
```

Figure 4: Traditional Java Methods Associated with Interrupt Handling Code

To set up this interrupt handler within the hard real-time domain, the real-time programmer executes code such as is illustrated in Figure 5. Note the invocation of the `javax.jmc.Registry.publish()` on line 19. The `javax.jmc` package represents a hypothetical mission-critical package that complements the capabilities of the safety-critical package. The `publish()` method makes the `Interrupt` object visible to the traditional Java domain.

In order to establish a communication channel between the traditional Java environment and the hard real-time domain, the traditional Java environment must instantiate a `JavaDevice` object, as represented by the code provided in Figure 6. Note that the constructor for `JavaDevice` looks up the proxy object known by the name "ByteStream", throwing an `IllegalStateException` if the registry does not currently hold any object by that name. Having instantiated a `JavaDevice` object, the traditional Java domain easily fetches bytes streamed from the device driver by invoking the `getByte()` method. Note that the typical control path through the `getByte()` method simply extracts a single byte from the `input_buffer`. This requires no synchronization with the hard real-time domain. This protocol is both much more efficient and much more

```
[1] package samples.device;
[2]
[3] import javax.jsc.*;
[4] import javax.jmc.Registry;
[5] import javax.jmc.TraditionalJavaShared;
[6]
[7] public class Main {
[8]     private static final int INTERRUPT_NO = 9;
[9]
[10]     public @TraditionalJavaShared static void main(String args[]) {
[11]         Interrupt handler;
[12]         InterruptEvent vector;
[13]
[14]         try {
[15]             MonitorControl.setMonitorControl(PriorityCeilingEmulation.
[16]                 instance(Interrupt.INTERRUPT_PRIORITY));
[17]             handler = new Interrupt();
[18]             vector = new InterruptEvent(INTERRUPT_NO, handler);
[19]             Registry.instance().publish("ByteStream", handler);
[20]             vector.arm();
[21]             doOtherStuff();
[22]         } catch (Throwable t) {
[23]             ;
[24]         } finally {
[25]             vector.disarm();
[26]             Registry.instance().unpublish(handler);
[27]             Registry.instance().awaitClearRegistry();
[28]         }
[29]     }
[30] }
```

Figure 5: Code to Set Up Interrupt Handler in Hard Real-Time Domain

reliable than using the Java Native Interface (JNI) to integrate high-level Java code with low-level device driver code.

Using Nested Scopes for Composition of Modular Software Systems

The proposed dynamic memory allocation abstractions for the safety-critical and hard real-time mission-critical RTSJ profiles require that memory for particular modules be allocated and deallocated in LIFO order, with certain modules being contained entirely within other modules. In general, higher layer soft real-time software is much more dynamic and lower level hard real-time software tends to be much more static. However, even the hard real-time components occasionally need to be able to reconfigure their use of memory.

There is design tension between the need to support very efficient and reliable operation, and the desire to support flexibility and dynamic reconfiguration in the field. Many mission-critical systems are required to support non-stop operation. It may be difficult or cost prohibitive to shut down the system each time software reconfiguration is required. Consider, for example, some common needs for flexibility in field-deployed systems:

```
[1] public class JavaDevice {
[2]     javax.rtpoxy.packages.samples.device.Interrupt input_port;
[3]     javax.rtpoxy.ByteArray input_buffer;
[4]     int input_index;
[5]     int input_length;
[6]
[7]     // Throws IllegalStateException if the Registry does not hold
[8]     // any object identified by the name "ByteStream"
[9]     public JavaDevice() throws IllegalStateException {
[10]
[11]         input_port = (javax.rtpoxy.packages.samples.device.Interrupt)
[12]             javax.rtpoxy.Root.lookup("ByteStream");
[13]
[14]         // initialize these variables to force first read to get new buffer
[15]         input_index = input_length = 0;
[16]     }
[17]
[18]     public final byte getByte() {
[19]         if (input_index >= input_length) {
[20]             input_buffer = input_port.getReadBuffer();
[21]             input_index = 0;
[22]             input_length = input_port.readBufferLength();
[23]         }
[24]         return input_buffer.atGet(input_index++);
[25]     }
[26] }
```

Figure 6: Traditional Java Code to Establish Communication Between Hard and Soft Real-Time Domains

- An error is discovered in a network protocol stack and a new version of this software must be installed to replace the faulty module.
- An orbiting communications satellite is required to support new services that were not anticipated or fully standardized when the satellite was originally launched. This likely requires the installation of new control-plane and management-plane functionality.
- A hardware failure in a line card of a highly available telecommunication server requires that the line card be hot-swapped with a new replacement. Since the replacement hardware has a different ROM revision number than the original hardware, certain shelf controller device drivers must be updated.
- A diagnostic computer is hooked up to a running telecommunication switch. The diagnostic system installs certain temporary software components into the telecommunication switch in order to gather the required diagnostic information.
- A sensor in a rocket ship fails. If the sensor has only partially failed, it may be possible to remedy this situation by replacing the sensor's device driver with a new driver that simply filters and recalibrates sensor readings. In other cases, it may be possible to replace the failed sensor's device driver with a software module that approximates the desired sensor data by assimilating information from alternative sources. For example, if one of four redundant sensors fails, the failed sensor readings might be replaced with the average reading from the other three.
- Due to limited communication bandwidth between UAV (Unmanned aerial vehicle) or orbiting satellite or deep-space probe or ASW (Anti-Submarine Warfare) buoy and "central control", certain assim-

ilation and fusion of sensor data must be performed by remotely deployed devices. There is simply not enough bandwidth to transmit all gathered sensor information back to headquarters. In some cases, such as with deep-space probes, the long round-trip communication delay precludes the use of closed loop feedback-control algorithms that include earth involvement in the closed loop. In all of these cases, it may be necessary to upload custom-tailored information processing or control loop algorithms into the remotely deployed systems in order to achieve maximum utilization of the limited communication bandwidth based on specific needs and circumstances.

- A communication engineer plugs a new sensor into a software-controlled field radio so that new sensor information can be transmitted to the command and control center. This new sensor, which likely requires installation of certain device driver software into the intelligent radio system, might provide seismic, sonar, radar, visual, or laser-based weapons targeting data.

Within the soft real-time domain, a real-time implementation of J2SE provides excellent support for dynamic reconfiguration of software. New classes can be loaded. Obsolete classes can be unloaded. Defragmenting real-time garbage collection efficiently and reliably manages the provisioning of memory resources to support the newly installed functionality.

It is more difficult to maintain compliance with high performance and hard real-time constraints in the face of software reconfiguration. System architects must carefully organize their hard real-time architecture so as to facilitate modular replacement and enhancement of particular software components within the architecture.

The first step is to minimize dependencies between individual components. Consider, for example, an idealized architecture for an aircraft mission control system. This particular example is contrived to illustrate the mixing of hard and soft real-time components, and of components that are considered safety-critical with those that are more in the realm of the mission-critical domain. In a real system, there would be stronger partitioning and stronger isolation of concerns between these hypothetical components. The necessary modules might consist of the following:

1. Interface to global positioning hardware
2. Interface to air traffic control communication subsystem
3. Interface to meteorological communication
4. Interface to aircraft sensors (fuel gauges, temperature, pressure, wind speed)
5. Module to provide graphical user interface
6. Module to maintain map data base
7. Module to maintain and assimilate relevant air traffic control information
8. Module to maintain and assimilate relevant meteorological information
9. Module to maintain current flight plan
10. Module to track flight progress against current plan
11. Module to support what-if scenario analysis (Do I have sufficient fuel to divert to alternative destination?)
12. Module to support revision to or replacement of the current flight plan
13. Module to perform auto-pilot functions

In considering how to architect this particular software system, the architect is likely to begin by characterizing the criticality and timeliness requirements associated with each of the components in the system. Suppose the architect’s analysis characterizes the individual components according to the chart illustrated in Figure 7. Note that we have divided the Graphical User Interface (component 5) into two components,

Hard Real Time	9:Flight Plan 10:Flight Tracking 3:Weather Comm	1:GPS 4:Aircraft Sensors 2:ATC Comm 5a:GUI HRT 7a:ATC Separation Control 13:Auto Pilot
	7b:ATC Flow Model 8:Weather Model 6:Map Data 5b:GUI SRT 11:What-if? Analysis 12:Flight Plan Maintenance	
Soft Real Time		
	Mission Critical (DO-178B Level C)	Safety Critical (DO-178B Level A)

Figure 7: Criticality and Timeliness Requirements for Architectural Components

one representing the safety-critical glass-cockpit information display that is required to manually fly the airplane, and the other representing more general purpose interactive display for use by a flight or navigation engineer who might be responsible for making changes to the planned mission. Note also that the stream of digital data represented by the Air Traffic Control radio is communicated both to the airplane minimum separation enforcement module (7a) and to the flow control module (7b). Air Traffic Flow Control is provisioned in this model as a mission-critical function.

For purposes of this discussion, assume that all of the software is to be implemented on the same microprocessor running a partitioned ARINC-653 kernel. The six level-A components are all implemented within one partition and all of the level-C components are implemented in a different partition. Figure 8 illustrates the dependencies between the various components. An arrow from one component to another indicates that the first depends on the second. Note that the dependency analyses in these figures describe software maintenance dependencies. When we say, for example, that the “Flight Tracking” module depends on the “Flight Plan Interface”, we are saying that any changes to the “Flight Plan Interface” may require changes to the “Flight Tracking” module. Ultimately, this represents our best guess as to future software maintenance requirements. In general, we are more confident that we can hold interfaces stable while evolving the implementations of certain modules that communicate with each other by way of these interfaces.

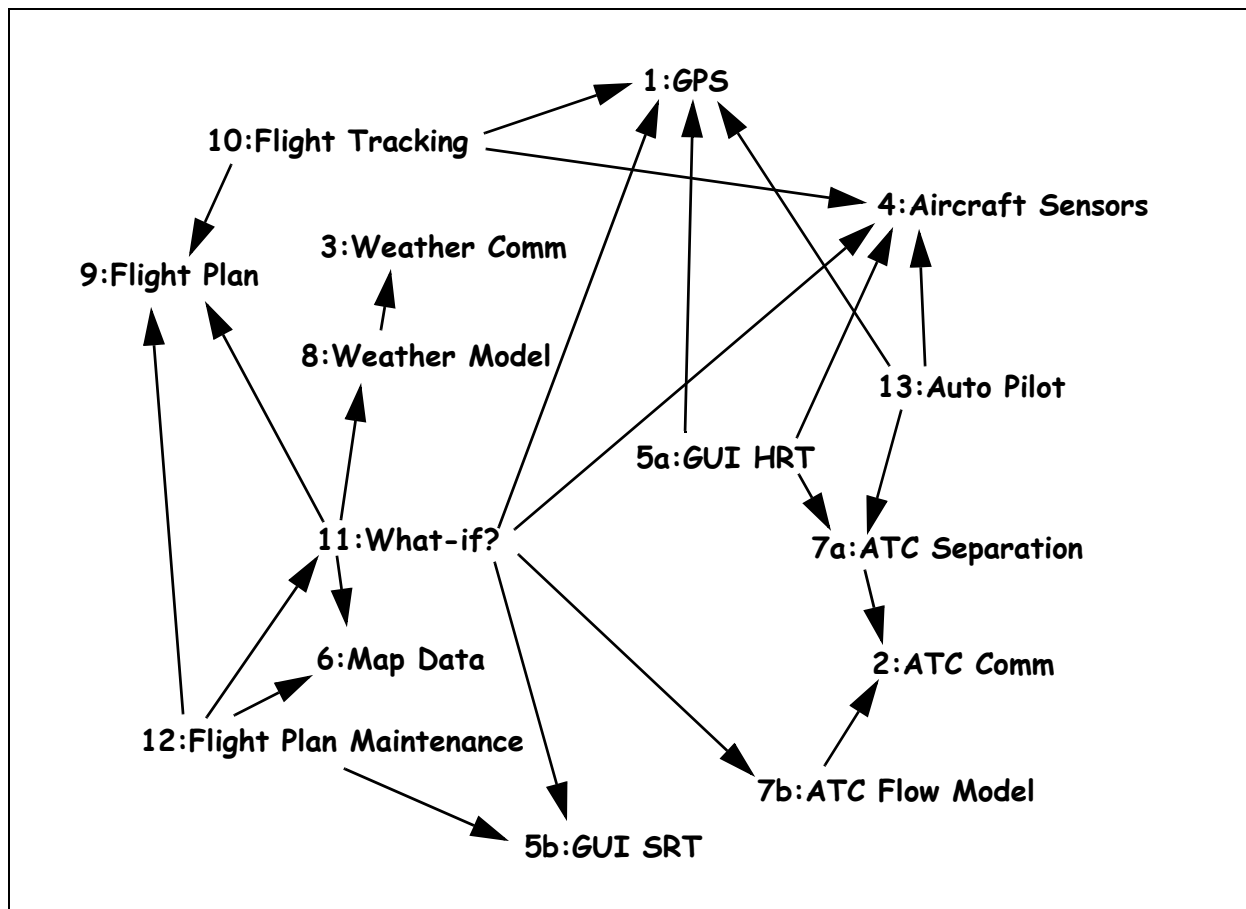


Figure 8: Inherent Dependency Analysis Between Components

The dependency analysis illustrated in Figure 8 represents inherent source-code maintenance dependencies. However, it ignores partitioning, flexibility, and maintainability issues. To address these issues, it is necessary to introduce standard interfaces and shared buffers into the dependency model. A dependency analysis of the enhanced model is illustrated in Figure 9.

In Figure 9, we have color coded the objects to emphasize their distinct realms of responsibility. All of the components residing in the safety-critical domain are colored red. Blue objects reside in the hard real-time mission-critical domain, and black objects reside in the soft real-time (traditional Java) domain. We use green to represent ARINC 653 ports that connect safety-critical to mission-critical components. Further, we assume that only hard real-time components can communicate directly with these ARINC 653 ports. Thus, whenever a traditional Java component requires access to ARINC 653 port data, an intermediary hard real-time component serves to shuttle the data between the respective domains.

Given this organization of software components, we would first divide the software between safety-critical and mission-critical functionality. Then, we would carefully organize the memory partitions for each of the two hard real-time execution domains. The safety-critical partition might be arranged as shown in Figure 10. The suggested organization of the mission-critical partition is shown in Figure 11.

Note in Figure 10 that there are three layers of nested scopes. The level-0, or outermost scope, holds the internal interfaces to key components. These interfaces are placed in the outer-most scope to make them visible to all of the inner-nested scopes that represent the individual components in the system. Scope lev-

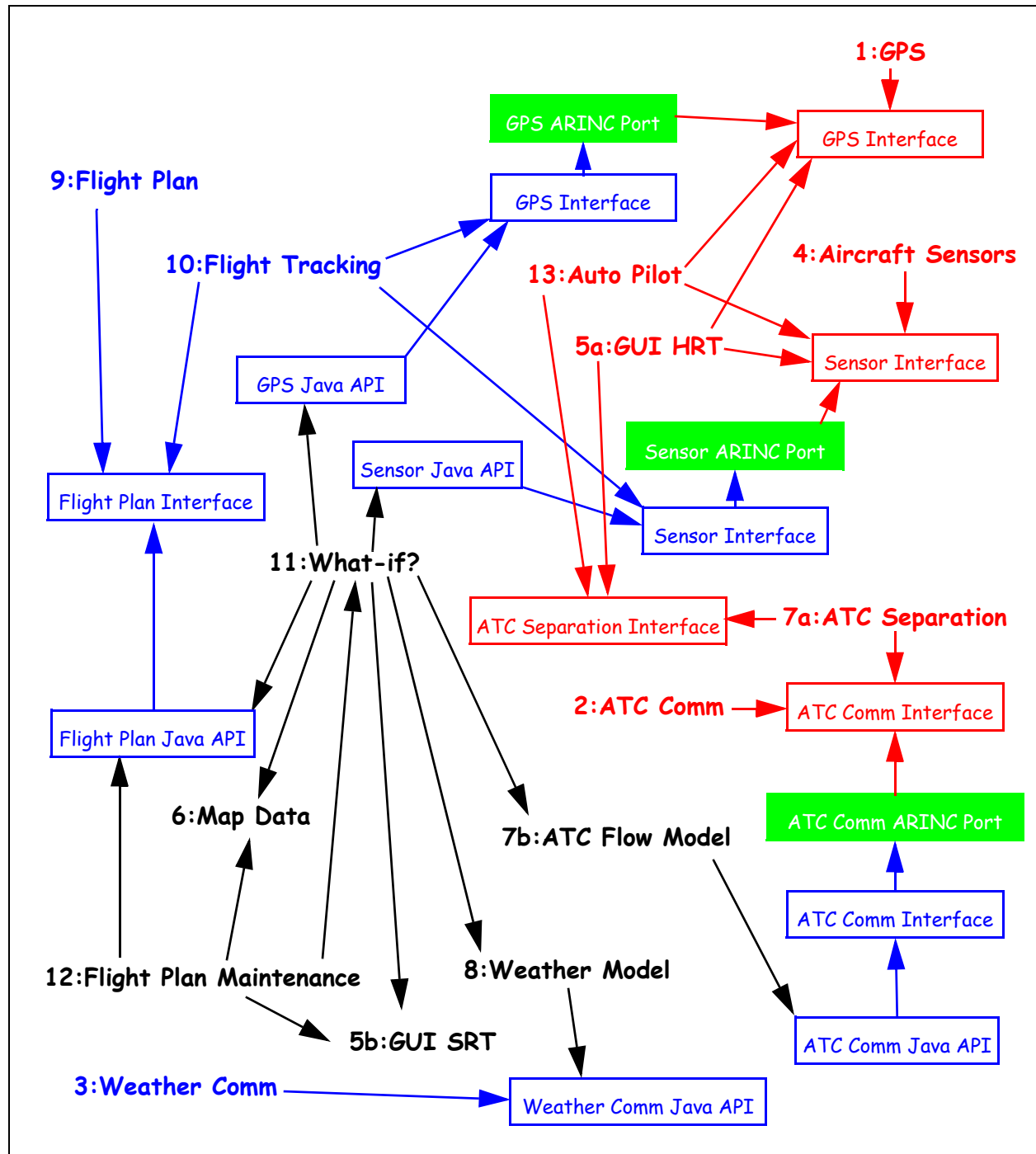


Figure 9: Dependency Analysis of Modules and Interfaces

els 0 and 1 both hold collections of objects. In the third scope level, we allocate multiple independent scopes, one to represent each of the threads that comprises the functional modules of the safety-critical partition.

The memory organization illustrated in Figure 11 suggests the use of four nested scope levels within the mission-critical partition. Java code to initialize the mission-critical partition is provided in Figures 12 through 17. Figure 12 provides the code to initialize the Level0 data structures. Note that the four variables

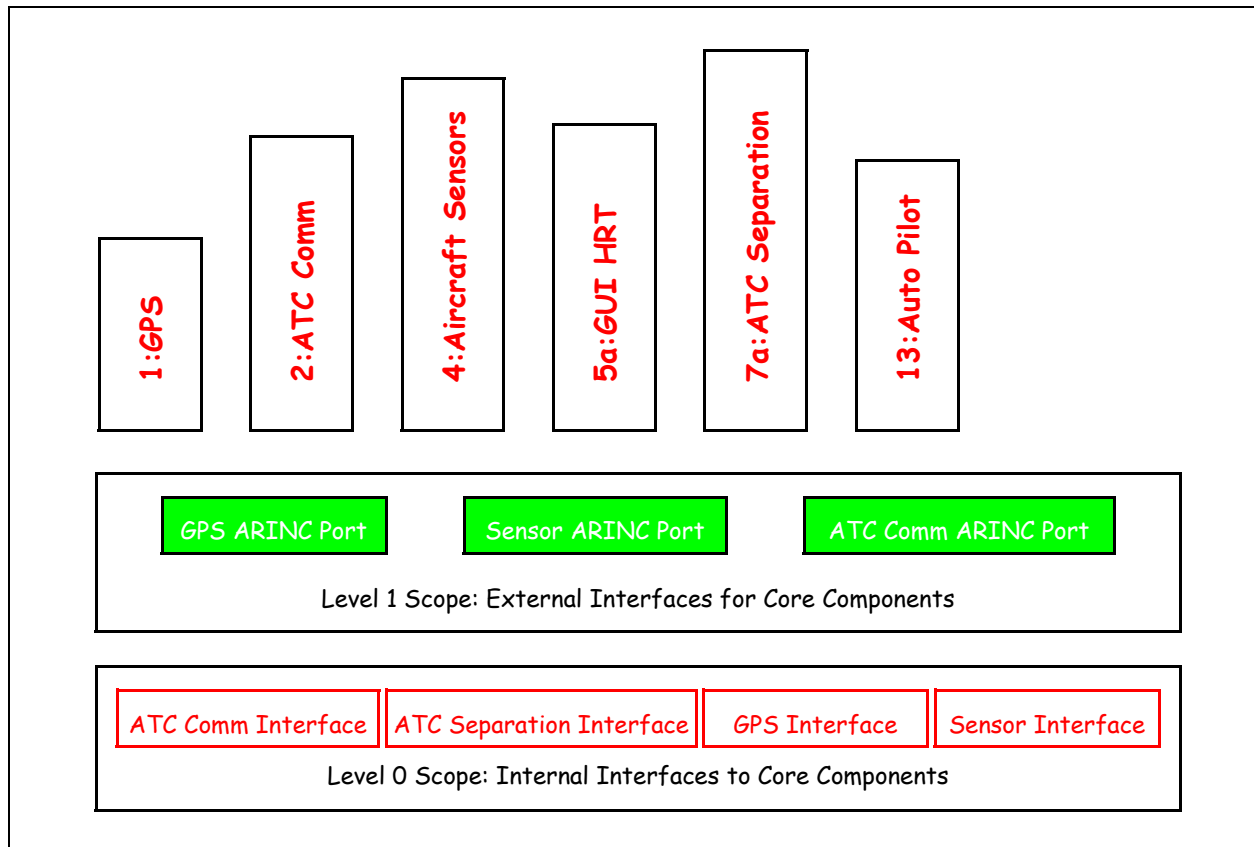


Figure 10: Organization of Safety-Critical Memory Partition

atc, sensor, gps, and fp are all declared with the @Scoped attribute. This indicates the programmer's intent that these variables may refer to objects allocated within the same scope as this Level0 object or in more outer-nested scopes. All of the assignments to these variables are made within this object's constructor. Thus, no run-time assignment checks are required when these variables are initialized since all of the objects are allocated within the same scope.

The inner_scope variable represents all of the memory dedicated to inner-nested scopes. The RTSJ rules of referential integrity prohibit the Level0 scope from holding any references to objects allocated within the inner-nested scopes. ThreadStack is a proposed new mission-critical abstraction designed to make possible the creation of scopes within scopes. The ThreadStack object itself is allocated within the Level0 scope. But any objects allocated within the ThreadStack, including the thread that is spawned on line 25 of Figure 12, are allocated within inner-nested scopes and cannot be seen from within the Level0 object.

The proposed rules for use of the ThreadStack abstraction within mission-critical code require that each invocation of the spawn() method be paired with a finally invocation of the same ThreadStack object's join() method, as shown, for example, on lines 23 through 28 of Figure 12 and lines 33 through 47 of Figure 13. This is to be enforced by the byte-code verifier.

The arguments to the spawn() invocation are:

1. A @Scoped Class object which must derive from NoHeapRealtimeThread. The spawn() method assures that this Class argument derives from NoHeapRealtimeThread, and further requires that the

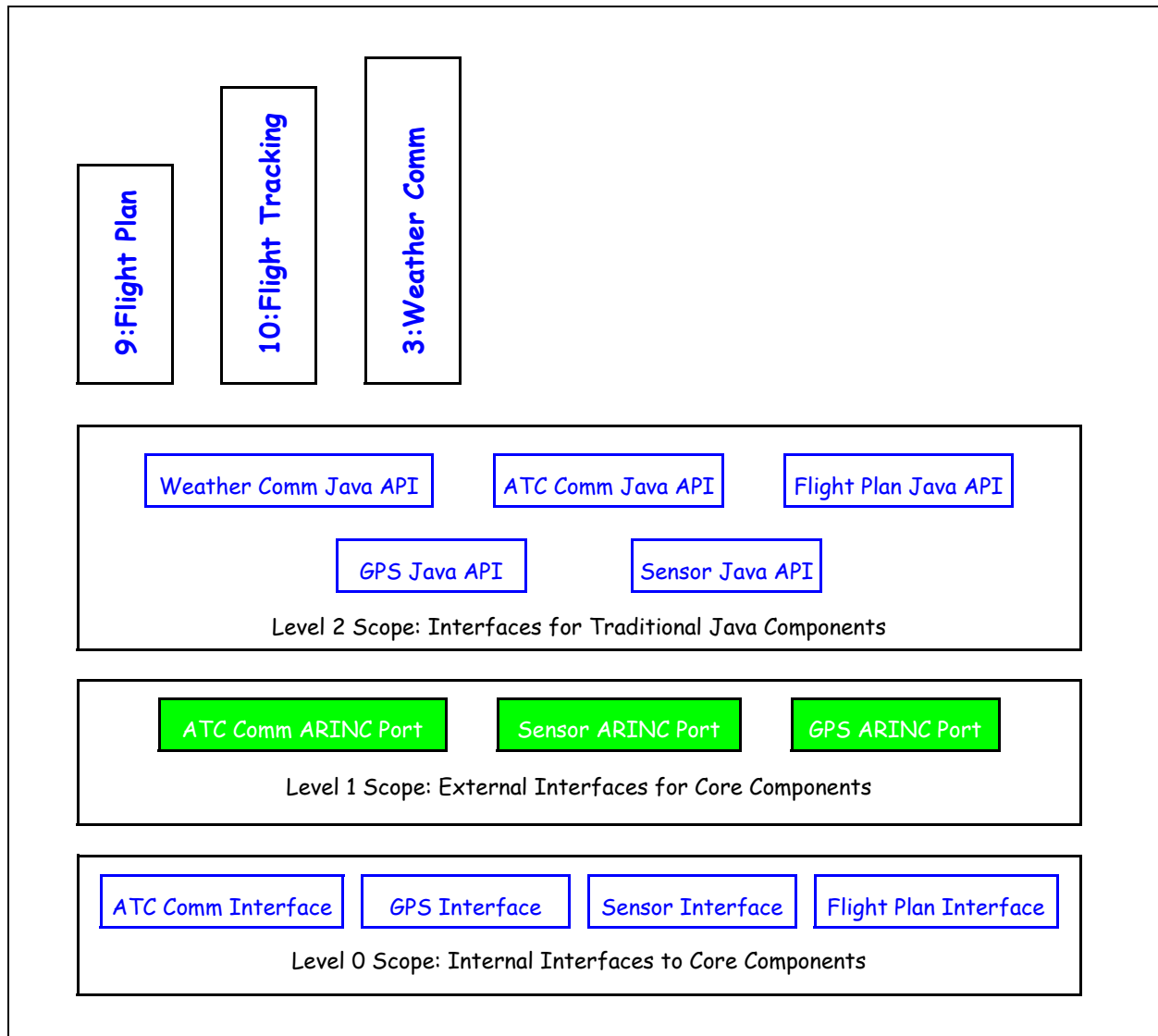


Figure 11: Organization of Hard Real-Time Mission-Critical Partition

class has a constructor that takes as arguments instances of `Object`, `SchedulingParameters`, and `MemoryArea`.

2. A `@Scoped Object` which will be passed as an argument to the constructor of the newly spawned thread. In this case, we are passing a reference to the `Level0` object itself, as this makes it possible for inner-nested scopes to gain access to the shared `atc`, `sensor`, `gps`, `fp`, and `inner_scope` variables.
3. A `@Scoped SchedulingParameters` value which governs the scheduling of the newly spawned thread.

The three parameters are all declared to have the `@Scoped` attribute because the spawned thread promises to keep these references within its scope and not copy them into global locations. Another form of the `spawn()` method omits the second (`Object`) argument. Use this second form to instantiate the primordial `Level0` object.

Figure 13 provides initialization of the the `Level1` class. The `Level1` constructor is automatically invoked from within the `spawn()` invocation on line 25 of Figure 12. Note that the first argument of the constructor

```

[1] public class Level0 extends NoHeapRealtimeThread {
[2]     private final static int SIZE_OF_INNER_SCOPES = 4096 * 9;
[3]     private final static int NESTED_PRIORITY_LEVEL = 18;
[4]
[5]     @Scoped ATC_CommInterface atc;
[6]     @Scoped SensorInterface sensor;
[7]     @Scoped GPS_Interface gps;
[8]     @Scoped FlightPlanInterface fp;
[9]     @Scoped ThreadStack inner_scope;
[10]
[11]     public @ScopedPure Level0(SchedulingParameters parms, MemoryArea area) {
[12]         super(parms, area);
[13]         atc = new ATC_CommInterface();
[14]         sensor = new SensorInterface();
[15]         gps = new GPS_Interface();
[16]         fp = new FlightPlanInterface();
[17]         inner_scope = new ThreadStack(SIZE_OF_INNER_SCOPES);
[18]     }
[19]
[20]     public @ScopedThis void run() {
[21]         PriorityParameters parms;
[22]
[23]         try {
[24]             parms = new PriorityParameters(NESTED_PRIORITY_LEVEL);
[25]             inner_scope.spawn(Level1.class, this, parms);
[26]         } finally {
[27]             inner_scope.join();
[28]         }
[29]     }
[30] }

```

Figure 12: Code to Initialize the **Level0** Class

is copied from the second argument to `spawn()`. We coerce this argument to a `Level0` reference, thereby obtaining access to the various outer-nested objects that are used by the Level-1 objects instantiated within this constructor. Note in line 22, for example, that construction of the level-1 `GPS_ARINC_PORT` object takes as an argument a reference to the level-0 `GPS_Interface` object that is represented by the `gps` instance variable within the `Level0` object.

Since each of the instance variables initialized by the constructor is annotated with the `@Scoped` attribute, all of the objects constructed within this constructor are allocated within the same `ThreadStack` memory scope that holds this `Level1` object. This means all of the memory for all of these objects can be instantly reclaimed when the `Level1 run()` method terminates. If any of these instance variables had not been labeled with the `@Scoped` annotation, the corresponding constructor would have been forced to allocate the memory for that object out of `ImmortalMemory` rather than using the temporary memory represented by the enclosing `ThreadStack` context.

The `run()` method is shown in Figure 15. Note the use of the `@TraditionalJavaShared` annotation on line 1. This signifies that the objects allocated within this method's private scope may be shared with traditional Java components. Each of the `javax.jmc.Registry.publish()` invocations (shown on lines 31 through 35) checks to ensure that the scope containing the published object is declared with the `@TraditionalJava-`

```

[1] public class Level1 extends NoHeapRealtimeThread {
[2]     private final static int THREAD_STACK_SIZE = 4096;
[3]     private final static int SIZE_OF_INNER_SCOPES = 4096 * 4;
[4]     private final static int NESTED_PRIORITY_LEVEL = 18;
[5]     private final static int GPS_PRIORITY_LEVEL = 30;
[6]     private final static int ATC_PRIORITY_LEVEL = 24;
[7]     private final static int SENSOR_PRIORITY_LEVEL = 24;
[8]
[9]     @Scoped GPS_ARINC_port gps_port;
[10]    @Scoped ATC_ARINC_port atc_port;
[11]    @Scoped Sensor_ARINC_port sensor_port;
[12]    @Scoped ThreadStack gps_memory;
[13]    @Scoped ThreadStack atc_memory;
[14]    @Scoped ThreadStack sensor_memory;
[15]    @Scoped ThreadStack inner_scope;
[16]    @Scoped Level0 outer_context;
[17]
[18]    public @ScopedPure Level1(Object outer_object,
[19]                               SchedulingParameters parms, MemoryArea area) {
[20]        super(parms, area);
[21]        outer_context = (Level0) outer_object;
[22]        gps_port = new GPS_ARINC_port(outer_context.gps);
[23]        atc_port = new ATC_ARINC_port(outer_context.atc);
[24]        sensor_port = new Sensor_ARINC_port(outer_context.sensor);
[25]        gps_memory = new ThreadStack(THREAD_STACK_SIZE);
[26]        atc_memory = new ThreadStack(THREAD_STACK_SIZE);
[27]        sensor_memory = new ThreadStack(THREAD_STACK_SIZE);
[28]        inner_scope = new ThreadStack(SIZE_OF_INNER_SCOPES);
[29]    }
[30]
[31]    public @ScopedThis void run() {
[32]        PriorityParameters parms;
[33]        try {
[34]            parms = new PriorityParameters(GPS_PRIORITY_LEVEL);
[35]            gps_memory.spawn(GPS_ARINC_thread.class, gps_port, parms);
[36]            parms = new PriorityParameters(ATC_PRIORITY_LEVEL);
[37]            atc_memory.spawn(ATC_ARINC_thread.class, atc_port, parms);
[38]            parms = new PriorityParameters(SENSOR_PRIORITY_LEVEL);
[39]            sensor_memory.spawn(Sensor_ARINC_thread.class, sensor_port, parms);
[40]            parms = new PriorityParameters(NESTED_PRIORITY_LEVEL);
[41]            inner_scope.spawn(Level2.class, this, parms);
[42]        } finally {
[43]            inner_scope.join();
[44]            gps_memory.join();
[45]            atc_memory.join();
[46]            sensor_memory.join();
[47]        }
[48]    }
[49] }

```

Figure 13: Code to Initialize the **Level1** Class

```

[1] public class Level2 extends NoHeapRealtimeThread {
[2]     private final static int THREAD_STACK_SIZE = 4096;
[3]     private final static int FPT_PRIORITY_LEVEL = 12;
[4]     private final static int FTT_PRIORITY_LEVEL = 24;
[5]     private final static int WCT_PRIORITY_LEVEL = 4;
[6]
[7]     private static final int SHUTDOWN = 0;
[8]     private static final int REPLACE_FLIGHT_PLAN_MODULE = 1;
[9]     private static final int REPLACE_FLIGHT_TRACKING_MODULE = 2;
[10]    private static final int REPLACE_WEATHER_COMM_MODULE = 3;
[11]    private static final int IDLE = 4;
[12]
[13]    @Scoped ThreadStack fpt_memory, ftt_memory, wct_memory;
[14]
[15]    @Scoped Level1 outer_context;
[16]
[17]    private int request = IDLE;
[18]
[19]    @Scoped Class replace_class;
[20]    @Scoped Object replace_arg;
[21]    @Scoped SchedulingParameters replace_parms;
[22]
[23]    public @ScopedPure Level2(Object outer_object,
[24]                               SchedulingParameters parms, MemoryArea area) {
[25]        super(parms, area);
[26]        outer_context = (Level1) outer_object;
[27]
[28]        fpt_memory = new ThreadStack(THREAD_STACK_SIZE);
[29]        ftt_memory = new ThreadStack(THREAD_STACK_SIZE);
[30]        wct_memory = new ThreadStack(THREAD_STACK_SIZE);
[31]    }
[32]

```

Figure 14: Variable Declarations and Constructor for Level2 Class

`aShared` attribute. If not, it throws an `IllegalArgumentException` and refuses to expose the object to the traditional Java domain. Further, the byte-code verifier assures that any method that is declared with the `@TraditionalJavaShared` attribute includes an invocation of `javax.jmc.Registry.awaitClearRegistry()` as the last statement in the `finally` block of code associated with the `try` statement that surrounds the body of code that comprises this method's body. See line 47. The `awaitClearRegistry()` invocation blocks the current thread until the traditional Java environment no longer holds any proxies for objects residing in this method's private scope.

The `Level2` class introduces the ability to selectively replace certain mission-critical program modules without shutting down the entire system. This somewhat simplistic code illustrates the flexibility and capabilities of the proposed mission-critical composition hierarchy. It does not necessarily represent the ideal architecture for dynamic reconfiguration of mission-critical software components.

The `doMaintenance()` method is shown in Figure 16. This has the responsibility for dynamically reconfiguring certain Level-2 modules within a running mission-critical system. The `request` instance variable is set separately to request that a particular module be replaced (See Figure 17). The `joinAndSpawn()` method

```
[1] public @TraditionalJavaShared @ScopedThis void run() {
[2]     @Scoped Java_Weather_COMM weather = null;
[3]     @Scoped Java_ATC_COMM atc = null;
[4]     @Scoped Java_FlightPlan fp = null;
[5]     @Scoped Java_GPS gps = null;
[6]     @Scoped Java_SensorInterface sensor = null;
[7]     Object [] flight_tracking_args;
[8]     PriorityParameters parms;
[9]     javax.jmc.Registry registry = javax.jmc.Registry.instance();
[10]    try {
[11]        weather = new Java_Weather_COMM();
[12]        atc = new Java_ATC_COMM(this.outer_context.outer_context.atc);
[13]        fp = new Java_FlightPlan(this.outer_context.outer_context.fp);
[14]        gps = new Java_GPS(this.outer_context.outer_context.gps);
[15]        sensor = new Java_SensorInterface(this.outer_context.outer_context.sensor);
[16]
[17]        parms = new PriorityParameters(FPT_PRIORITY_LEVEL);
[18]        fpt_memory.spawn(FlightPlanThread.class,
[19]                        outer_context.outer_context.fp, parms);
[20]
[21]        parms = new PriorityParameters(FTT_PRIORITY_LEVEL);
[22]        flight_tracking_args = new Object[3];
[23]        flight_tracking_args[0] = outer_context.outer_context.fp;
[24]        flight_tracking_args[1] = outer_context.outer_context.gps;
[25]        flight_tracking_args[2] = outer_context.outer_context.sensor;
[26]        ftt_memory.spawn(FlightTrackingThread.class, flight_tracking_args, parms);
[27]
[28]        parms = new PriorityParameters(WCT_PRIORITY_LEVEL);
[29]        wct_memory.spawn(WeatherCommThread.class, weather, parms);
[30]
[31]        registry.publish("Weather Data", weather);
[32]        registry.publish("ATC Flow Control", atc);
[33]        registry.publish("Flight Plan", fp);
[34]        registry.publish("Global Positioning", gps);
[35]        registry.publish("Sensor Data", sensor);
[36]
[37]        this.doMaintenance();
[38]    } finally {
[39]        registry.unpublish(weather);
[40]        registry.unpublish(atc);
[41]        registry.unpublish(fp);
[42]        registry.unpublish(gps);
[43]        registry.unpublish(sensor);
[44]        fpt_memory.join();
[45]        ftt_memory.join();
[46]        wct_memory.join();
[47]        Registry.instance().awaitClearRegistry();
[48]    }
```

Figure 15: The run() Method for Level2 Class

```

[1]    @ScopedThis synchronized void doMaintenance() {
[2]        while (request != SHUTDOWN) {
[3]            switch (request) {
[4]
[5]                case REPLACE_FLIGHT_PLAN_MODULE: {
[6]                    fpt_memory.joinAndSpawn(replace_class, replace_arg, replace_parms);
[7]                    request = IDLE;
[8]                    this.notifyAll();
[9]                    break;
[10]                }
[11]
[12]                case REPLACE_FLIGHT_TRACKING_MODULE: {
[13]                    ftt_memory.joinAndSpawn(replace_class, replace_arg, replace_parms);
[14]                    request = IDLE;
[15]                    this.notifyAll();
[16]                    break;
[17]                }
[18]
[19]                case REPLACE_WEATHER_COMM_MODULE: {
[20]                    wct_memory.joinAndSpawn(replace_class, replace_arg, replace_parms);
[21]                    request = IDLE;
[22]                    this.notifyAll();
[23]                    break;
[24]                }
[25]
[26]                case IDLE:
[27]                default:
[28]
[29]                }
[30]            this.wait();
[31]        }
[32]    }

```

Figure 16: The `doMaintenance()` method for `Level2` Class

first waits for the originally spawned thread to terminate and then instantiates a new thread in its place, overwriting the memory that had previously been reserved for the first thread. This illustration does not show the code that arranges for the original thread to terminate its execution. That must be handled elsewhere.

Remember that the byte-code verifier assures that every `spawn()` invocation is accompanied by a `join()` within an accompanying `finally` block of code. It is important to maintain this invariant in order to assure that program control does not leave any context before all of the references to that context have been abandoned. In preserving this same invariant, the `joinAndSpawn()` method enforces the following:

- This `ThreadStack` must have successfully spawned a thread which has not yet been joined by the parent thread. Otherwise, `joinAndSpawn()` does not wait to join the subordinate thread and instead throws `IllegalStateException`.
- Any `@Scoped` variables passed as arguments to the `joinAndSpawn()` method must belong to scopes that belong to the same scope or to scopes more outer-nested than the scope that hosted the original `spawn()` invocation. This is necessary because that scope contains the `join()` operation that preserves

```

[1]  @AllowCheckedScopedLinks @ScopedThis synchronized void
[2]      changeWeatherCommModule(@Scoped Class new_class,
[3]                              @Scoped Object new_arg,
[4]                              @Scoped SchedulingParameters new_parms) {
[5]      while (request != IDLE)
[6]          this.wait();
[7]      replace_class = new_class;
[8]      replace_arg = new_arg;
[9]      replace_parms = new_parms;
[10]     request = REPLACE_WEATHER_COMM_MODULE;
[11]     this.notifyAll();
[12] }
[13]
[14] @AllowCheckedScopedLinks @ScopedThis synchronized void
[15]     changeFlightTrackingModule(@Scoped Class new_class,
[16]                               @Scoped Object new_arg,
[17]                               @Scoped SchedulingParameters new_parms) {
[18]     while (request != IDLE)
[19]         this.wait();
[20]     replace_class = new_class;
[21]     replace_arg = new_arg;
[22]     replace_parms = new_parms;
[23]     request = REPLACE_FLIGHT_TRACKING_MODULE;
[24]     this.notifyAll();
[25] }
[26]
[27] @AllowCheckedScopedLinks @ScopedThis synchronized void
[28]     changeFlightPlanModule(@Scoped Class new_class,
[29]                            @Scoped Object new_arg,
[30]                            @Scoped SchedulingParameters new_parms) {
[31]     while (request != IDLE)
[32]         this.wait();
[33]     replace_class = new_class;
[34]     replace_arg = new_arg;
[35]     replace_parms = new_parms;
[36]     request = REPLACE_FLIGHT_PLAN_MODULE;
[37]     this.notifyAll();
[38] }
[39] }

```

Figure 17: **Level2** Methods to Hot-Swap Certain Modules

the outer scope until the inner-nested thread has terminated. Otherwise, `joinAndSpawn()` does not wait to join the subordinate thread and instead throws `IllegalArgumentException`.

Figure 17 provides the code for three methods that independently replace certain running modules within the level-3 component. All of the arguments to these three methods are declared with the `@Scoped` annotation. This means these arguments may refer to objects residing within temporary memory scopes. Because static analysis may not be able to determine all of the dynamic contexts within which these methods might be invoked, a run-time check is required on assignment to the `replace_class`, `replace_arg`, and `replace_parms` instance variables. This is why each of the methods has the `@AllowCheckedScopedLinks`

annotation. If the author of these components preferred to avoid this run-time check, he could remove the `@AllowCheckedScopedLinks` annotation and the `@Scoped` annotations on each of the arguments. This would force users of these services to pass references to `ImmortalMemory` objects as arguments.

Conclusions

Though it may be more difficult to program high-performance, hard real-time code in Java than to write traditional Java code, certain components of most mission-critical systems must be implemented using technologies that are more efficient and more deterministic than traditional Java. At the same time, development and maintenance of these low-level components using portable stylized Java rather than assembler, C, or C++ will yield significant productivity improvements and cost savings. We have every reason to believe that the two-fold developer productivity benefits and five- to ten-fold software maintenance benefits that Java has exhibited over C++ in traditional information technology markets can be matched in the deeply embedded mission-critical and safety-critical markets as well.

By carefully partitioning functionality between high and low-level software, it is possible to leverage the best strengths of Java within each respective programming domain. Low-level Java technologies are most appropriate for implementing low-level device drivers, including interrupt handlers, safe and efficient interfaces to legacy (“native”) code, and certain performance-critical components such as Fast Fourier transforms.

Appendix A: The Traditional Java API¹

The flight mission planning example above helps to motivate the need for and design of the traditional Java API. This section describes the API in more thorough detail. When a hard real-time object is published to the traditional Java world, the system creates a traditional Java object to serve as a proxy for the hard real-time object within the traditional Java domain. When a traditional Java thread invokes a method of a proxy object, the resulting interaction with the hard real-time environment may require the use of a proxy thread that is scheduled according to the rules of the hard real-time environment.

Declaration of Traditional Java Methods. A method that is declared with the `@TraditionalJavaMethod` annotation is designed to be invoked by a traditional Java thread running in an environment that is cooperating with the hard real-time environment. Consider, for example, the source code for the hard real-time class named `Thermostat`, which is shown in Figure 1 on page 3. In this class, the `getTemperature()` and `setTemperature()` methods are both identified as traditional Java methods, meaning these methods are to be invoked only by traditional Java threads. When compiled and linked into a hard real-time Java execution environment, the `updateTemperature()` and `checkThermostat()` methods are regular methods for invocation by other hard real-time threads.

The Registry. The class `javax.jmc.Registry` provides mechanisms for use by hard real-time components that desire to expose certain services to the traditional Java domain. The `Registry` only allows objects that were allocated within scopes that are associated with methods declared to have the `@TraditionalJavaShared` attribute. In theory, a hard real-time Java environment can concurrently share objects with multiple independent Java virtual machine environments. The `@TraditionalJavaShared` annotation takes an optional argument, named `jvm_id`, which specifies the name of the traditional Java virtual machine with which a given scope is allowed to share its allocated objects.

Having instantiated a hard real-time object to be shared with the traditional Java environment, the hard real-time programmer publishes this object so that it can be seen by traditional Java threads by invoking the `publish()` method of `javax.jmc.Registry`, passing as arguments the unique string name by which this object will be identified within the registry and a reference to the object. Traditional Java components will obtain references to this object's proxy by invoking the `javax.rtpoxy.Root.lookup()` method within the traditional Java environment, passing the string name that identifies the object as the sole argument to the `lookup()` method.

The following services are supported by the `javax.jmc.Registry` abstraction:

```
public static Registry instance() throws IllegalStateException
```

Returns a reference to the primordial instance of `Registry`, or throws `IllegalStateException` if the primordial instance has not yet been instantiated.

```
public static Registry instance(String name) throws IllegalStateException
```

Returns a reference to the instance of `Registry` that corresponds to the connected Java virtual machine that is identified by `name`, or throws `IllegalStateException` if no `Registry` has been instantiated to represent an interface to the named virtual machine.

1. The concepts described in this document were presented to the Open Group's Real-Time and Embedded Forum on Oct. 21 in New Orleans, LA. During the discussion that followed this presentation, it was suggested that we consider using RMI to connect the non-real-time and hard real-time domains rather than the *ad hoc* techniques described in this proposal. This suggestion is currently under consideration.

`public Registry(int num_proxy_stacks, int stack_size) throws IllegalStateException`
Constructs the primordial instance of `Registry` having an initial pool of `num_proxy_stacks` `ThreadStack` objects, each having the specified `stack_size`. This method throws `IllegalStateException` if the primordial instance has already been instantiated. This method throws `OutOfMemoryError` error if there is not sufficient memory to instantiate the `ThreadStack` objects.

`public Registry(String name, int num_proxy_stacks, int stack_size) throws IllegalStateException`
Constructs an instance of `Registry` to represent the interface to the Java virtual machine identified by `name`, allocating an initial pool of `num_proxy_stacks` `ThreadStack` objects, each having the specified `stack_size`. This method throws `IllegalStateException` if a `Registry` instance has already been instantiated with this same `name`. This method throws `OutOfMemoryError` error if there is not sufficient memory to instantiate the `ThreadStack` objects.

`public final void publish(@Scoped String name, @Scoped Object obj) throws IllegalArgumentException, IllegalStateException`
Publish `obj` so it is accessible from the traditional Java domain. Both `name` and `obj` may be scope-allocated. A run-time check assures that `obj` is allocated within a scope that is declared with the `@TraditionalJavaShared` annotation and this `@TraditionalJavaShared` scope is consistent with this `Registry` object, and that the scope of `name` is compatible with assignment to `obj` (i.e. that `name` resides in the same or more outer nested scope as `obj`). This method throws `IllegalArgumentException` if the scope containing `obj` was not annotated with the `@TraditionalJavaShared` annotation or if the scope containing `name` is not properly nested. This method throws `IllegalStateException` if there already exists a published symbol with the same name. The byte-code verifier assures that every scope that is declared with the `@TraditionalJavaShared` attribute includes an invocation of `awaitClearRegistry()`, as described below.

`public final void unpublish(@Scoped Object obj) throws IllegalStateException`
Remove the hard real-time object `obj` from the public registry so that the traditional Java environment can no longer look it up. Note that the traditional Java environment may still have a way to see this object even if it's no longer in the public registry. For example, a traditional Java thread may have looked up the object before it was unpublished and the corresponding proxy object might retain a reference to the object even after the object was removed from the registry. This method throws `IllegalStateException` if the object is not currently found in the registry.

`public final void unpublish(@Scoped String name) throws IllegalStateException`
Remove the hard real-time object known symbolically by `name` from the public registry so that the traditional Java environment can no longer look it up. Note that the traditional Java environment may still have a way to see this object even if it's no longer in the public registry. For example, a traditional Java thread may have looked up the object before it was unpublished and the corresponding proxy object might retain a reference to the object even after the object was removed from the registry. This method throws `IllegalStateException` if the object is not currently found in the registry.

`public boolean addProxyThreadStacks(int num_stacks)`
Add `num_stacks` `ThreadStack` objects to the pool of proxy threads associated with this `Registry` object. The size of each thread's `ThreadStack` is the same size as was specified in the constructor for this `Registry` object. Returns `true` if the stacks were successfully added, `false` if the stacks were not added because, for example, there was not sufficient memory.

```
public final void awaitClearRegistry()
```

This routine is normally called from a method that has declared itself to have the `@TraditionalJavaShared` attribute. If the caller method does not have this attribute, the invocation returns immediately as there can be no objects from this scope visible to the traditional Java domain. If the caller does have the `@TraditionalJavaShared` attribute, the method returns only after all of the proxy objects that were shared from this method's allocation scope have all been removed from the traditional Java domain. In general, this requires that the traditional Java garbage collector has reclaimed all of the proxy objects and the finalizer code for those proxy objects has unregistered the objects from the shared registry. Note that removal of objects from a shared Java registry should be a very rare event. Thus, we are willing to invest some "time and energy" in making sure there is a clean separation of concerns before destroying the corresponding hard real-time allocation context.

Traditional Java Proxies for Hard Real-Time Objects. The proxy for a hard real-time instance of `java.lang.Object` is represented in the traditional Java domain by an instance of `javax.rtpoxy.Root`. Arrays from the hard real-time environment are represented within the traditional Java domain by proxy objects of one of the following classes: `BooleanArray`, `ByteArray`, `ShortArray`, `CharArray`, `IntArray`, `LongArray`, `FloatArray`, `DoubleArray`, `RefArray`. All of these classes are defined within the `javax.rtpoxy` package and each extends `javax.rtpoxy.Root`. Each array class supports `length()`, `atGet()`, and `atPut()` methods. For `RefArray`, `length()` always returns zero and the `atGet()` and `atPut()` methods always throw `ArrayIndexOutOfBoundsException` because we do not allow the traditional Java environment to directly fetch or store reference values within the hard real-time environment.

All other classes defined within the hard real-time domain and shared with the traditional Java environment extend `javax.rtpoxy.Root` (either directly or indirectly), using a copy of the hierarchy that derives from `java.lang.Object` within the hard real-time domain. Classes defined in the default package of the hard real-time environment are treated as members of the `javax.rtpoxy.packages` package within the traditional Java environment. Hard real-time classes defined in named packages are treated as subpackages of `javax.rtpoxy.packages` within the traditional Java environment.

Preprocessing of Traditional Java Methods. To use the `Thermostat` class shown in Figure 1 on page 3 as an interface between traditional Java and hard real-time Java components, apply the `rtpoxyc` (real-time proxy compiler) program to this file, producing as output one class file that runs in the traditional Java domain and another that runs in the hard real-time domain. The deployment process is illustrated in Figure 18. The source code of Figure 1 is translated into class files representing the program components shown in Figures 19 and 20 respectively.

When the `Thermostat` class is loaded by the hard real-time class loader, the two methods that are tagged with the `@TraditionalJavaMethod` annotation are treated specially. These two methods are invisible to the hard real-time domain. Insofar as the hard real-time class hierarchy is concerned, they might as well not be present at all. Instead, these methods are inserted into the body of the proxy class definition that represents instances of this class within the traditional Java environment.

Since the Java specification requires that stores and fetches to `float` values be atomic, programmers might be tempted to remove the `synchronized` qualifier from the methods of this class. This would be an error. It turns out that in the absence of the `synchronized` qualifier, the optimization rules for Java compilers would permit the contents of the shared `measured_temperature` and `desired_temperature` variables to be cached in machine registers within the independent contexts of the respective hard real-time and traditional Java threads. Note also that all of the synchronization is performed within the context of the hard real-time execution environment. If a traditional Java thread synchronizes on a proxy object, that will acquire a lock

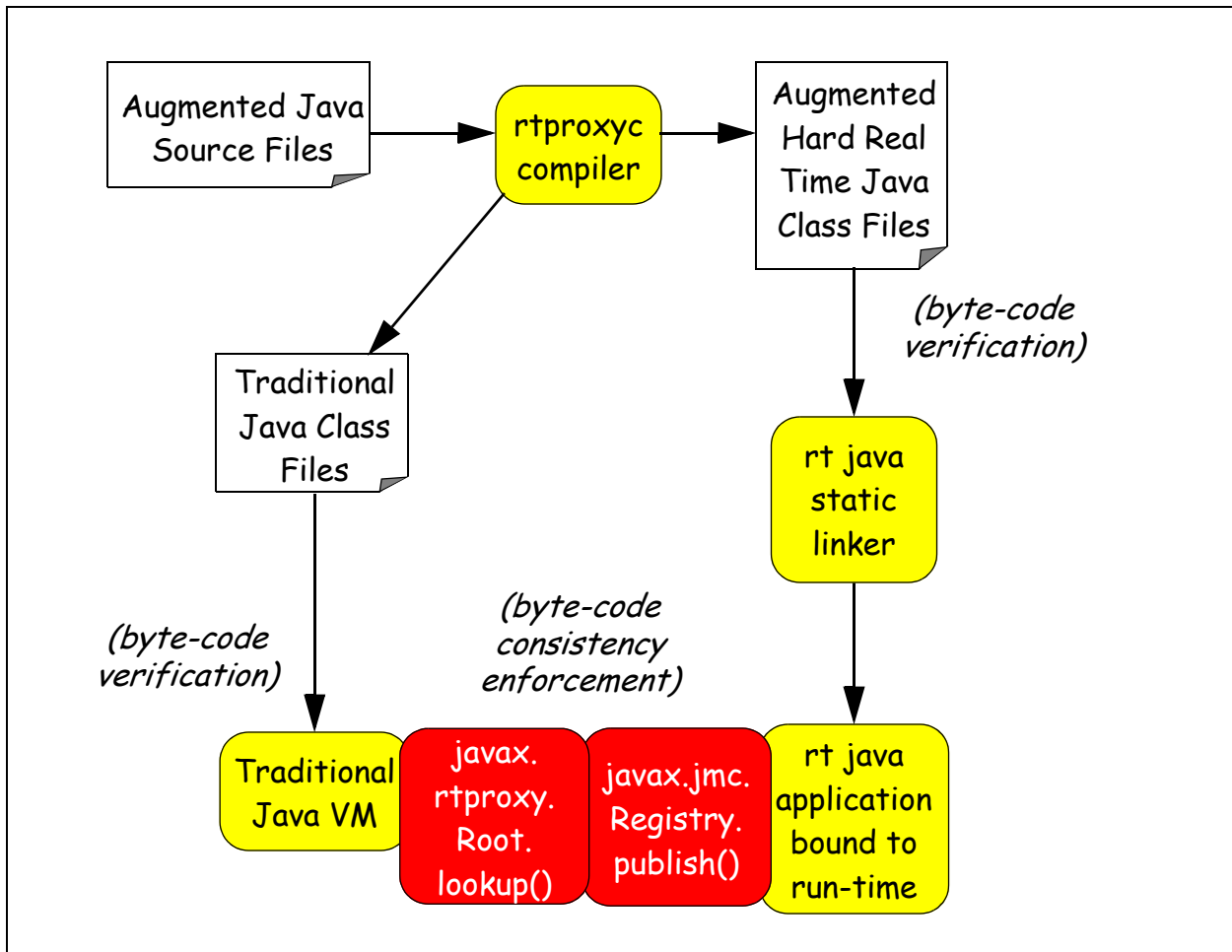


Figure 18: Deployment of Traditional Java Interfaces Between Hard Real-Time and Traditional Java

```
[1] package samples;
[2]
[3] public class Thermostat {
[4]     private float measured_temperature;
[5]     private float desired_temperature;
[6]
[7]     public final synchronized void updateTemperature(float f) {
[8]         measured_temperature = f;
[9]     }
[10]
[11]     public final synchronized float checkThermostat() {
[12]         return desired_temperature;
[13]     }
[14] }
```

Figure 19: Hard Real-Time View of Thermostat class

associated with the proxy object within the traditional Java environment. It will have no effect on any locks within the hard real-time domain. The only way to acquire a hard real-time lock is to invoke a traditional

```
[1] package javax.rtpoxy.packages.samples;
[2]
[3] public class Thermostat extends javax.rtpoxy.Root {
[4]     private long handle_to_rt_object;
[5]
[6]     /* Note that these methods are not synchronized even though they were marked
[7]      * synchronized in original source code. That's because the synchronization
[8]      * must be performed according to the rules of the hard real-time environment,
[9]      * so it must be implemented within the native method.
[10]    */
[11]     public final native float getTemperature() {
[12]         try {
[13]             javax.rtpoxy.Libraries.synchronize(handle_to_rt_object);
[14]             // fetch float value at offset 0 within hard real-time object
[15]             f = javax.rtpoxy.Libraries.getFloat(handle_to_rt_object, 0);
[16]         } finally {
[17]             javax.rtpoxy.Libraries.unsynchronize(handle_to_rt_object);
[18]         }
[19]         return f;
[20]     }
[21]
[22]     public final void setTemperature(float f) {
[23]         try {
[24]             javax.rtpoxy.Libraries.synchronize(handle_to_rt_object);
[25]             // store float value at offset 4 within hard real-time object
[26]             javax.rtpoxy.Libraries.setFloat(handle_to_rt_object, 4, f);
[27]         } finally {
[28]             javax.rtpoxy.Libraries.unsynchronize(handle_to_rt_object);
[29]         }
[30]     }
[31] }
```

Figure 20: Traditional Java View of **Thermostat** class

Java method and allow that method to acquire the lock using whatever protocols are appropriate for a given implementation of the hard real-time run-time environment.

The implementations of the two traditional Java methods in Figure 20 have been transformed from the original Java source code in order to provide access to the hard real-time object from within the traditional Java environment. The implementation of the `javax.rtpoxy.Libraries` classes may use JNI code, or might use implementation-specific optimizations such as special proxy-aware JIT compilation and in-lining techniques. The methods of the `javax.rtpoxy.Libraries` class have the ability to reach into the hard real-time domain in order to read and modify objects residing in that domain. These are privileged operations that should not be exposed to arbitrary Java components. In situations where a trusted JIT compiler integrates the implementation of these methods, the JIT compiler assures that these libraries are only invoked from within classes that derive from `javax.rtpoxy.Root`. When these library services are invoked “out of line”, a security check on the stack backtrace is performed to assure that the calling method resides within a class that derives from `javax.rtpoxy.Root`. Any other invocations of these library services represent security violations and are prohibited. A special exception, not specified in the current draft, is thrown.

For purposes of enforcing consistency between hard real-time components and the traditional Java components, the hard real-time class file components include as encoded attributes a representation of the traditional Java methods that were extracted from the original source file. In addition to enforcing the verification rules of traditional Java and the special verification rules of the hard real-time Java profile, the byte-code verifier for the hard real-time class file assures that these traditional Java methods comply with all of the special restrictions imposed by this draft specification on the bodies of such methods. Each time a new proxy class is loaded into the traditional Java domain, we perform a consistency check to assure that the methods of the traditional Java proxy object implement exactly the same functionality that is specified in the corresponding hard real-time class file's attribute encodings.

Execution of Traditional Java Methods. Proxy objects within the traditional Java domain are created each time a new hard real-time object is published by way of the registry and each time a reference to a new hard real-time object is returned from a traditional Java method. Every proxy is either an instance of `javax.rtpoxy.Root` or some class that extends this class.

When a reference value is returned or thrown from a traditional Java method, it is replaced within the traditional Java environment by its corresponding proxy. If no proxy exists, one is created at the time the value is returned. Reference values communicated from the hard real-time domain to the traditional Java domain must either reside within `ImmortalMemory` or within scopes that were declared with the `@TraditionalJavaShared` annotation. When a traditional Java method returns or throws a reference, a run-time check is performed at the interface to the traditional Java domain to assure that the reference corresponds to an object that can be shared with the traditional java environment, as indicated by the corresponding `@TraditionalJavaShared` annotation. It is the hard real-time programmer's responsibility to assure this requirement is satisfied. If a violation is detected, the traditional Java method terminates by throwing an `IllegalStateException`.

When a proxy object is passed as an argument to a traditional Java method, it is replaced with a reference to the proxy object's hard real-time referent while the method executes in the hard real-time domain. If a traditional Java method takes one or more reference arguments, it is necessary to assure that all of the reference arguments and the hard-real-time equivalent of the proxy's `this` are all scope compatible. This check consists of identifying which of the reference arguments is most deeply nested and then assuring that all of the other references correspond to scopes that are more outer nested on the same stack as the most deeply nested scope. If this condition is not satisfied, the traditional Java method invocation aborts, throwing an instance of `IllegalArgumentException` to the traditional Java thread that attempted to perform this invocation.

The constructor for `javax.rtpoxy.Root` includes a check to enforce that the only thread allowed to instantiate an instance of `javax.rtpoxy.Root` or of any of its subclasses is the special thread that is responsible for maintaining the relationship between the traditional Java domain and the hard real-time environment. This guarantees that any instances of `javax.rtpoxy.Root` and instances of any of its subclasses are true proxies for hard real-time objects residing in the hard real-time environment.

By applying this technique to assure that every proxy object was instantiated under our direction, and the implementation of its methods was derived from code written by the hard real-time developer, and the code has successfully fulfilled the stringent requirements of the hard real-time Java byte code verifier, there is no need for additional run-time checks on the invocation of the proxy object's traditional Java methods, except when a traditional Java method expects one or more reference arguments.

These conventions guarantee that the hard real-time domain never holds a reference to a traditional Java object, entirely eliminating the risk that hard real-time threads might need to coordinate with execution of

the garbage collector. This avoids the problems that preclude synchronization between hard real-time and non-real-time threads in the standard RTSJ.

Proxy Thread Management. When a traditional Java thread invokes a traditional Java method, the body of the method is executed at the priority of the Java thread that is executing. This thread is free to detach from the traditional Java virtual machine while it is executing the hard real-time Java method because it is guaranteed that no garbage collected objects will be accessed by the thread while it is executing this code.

If the traditional Java method performs synchronization, the priority inversion avoidance implementation is under the control of the hard real-time scheduler. If the synchronization lock is governed by priority ceiling emulation, the thread will immediately elevate its priority to the ceiling associated with that lock. If the lock is controlled by priority inheritance, this thread becomes eligible to inherit the higher priorities of the hard real-time threads. In both cases, it is important to recognize that even though the traditional Java thread holds a hard real-time lock, it only does so under the conditions that:

1. It has already detached from the JVM's scheduler and is not interacting with the JVM's garbage collector, and
2. It is only executing code that was written by the hard real-time developer.

Whenever a traditional Java method performs synchronization, the traditional Java thread is replaced by a proxy thread running within the hard real-time domain. Memory for the proxy thread is provided from the pool of proxy `ThreadStack` objects maintained by `javax.jmc.Registry`. The proxy's stack is instantiated as if a new thread had been spawned from the scope the contains the mostly deeply nested argument passed to the traditional Java method. In the case that the traditional Java method invocation does not pass any reference arguments, the allocation context of `this` is by default the mostly deeply nested context. This assures that the rules of referential integrity will allow the proxy thread's local variables and any objects allocated on the proxy thread's stack to make reference to objects that are contained within outer-nested scopes.

Byte Code Verification

The following additional byte code verification checks are required to support the capabilities described in this proposal:

1. The byte-code verifier assures that every scope that is declared with the `@TraditionalJavaShared` attribute includes an invocation of `awaitClearRegistry()` as the last statement in the `finally` clause corresponding to the `try` clause that contains all executable code except the clean up code associated with the method. The clean up code, all of which must appear in the `finally` clause, consists only of:
 - a. Invocations of `ThreadStack.join()`,
 - b. Invocations of `Registry.unpublish()`, and
 - c. A single invocation of `Registry.instance().awaitClearRegistry()`. If the enclosing method's `@TraditionalJavaShared` annotation specified the name of a particular Java virtual machine, the byte-code verifier assures that the same name is supplied as an argument to this `instance()` invocation.
2. Any invocation of `ThreadStack.spawn()` is contained within a `try` statement, and the corresponding `finally` statement includes a matching invocation of `ThreadStack.join()` for the same `ThreadStack` object.
3. Any method invoked from within a traditional Java method must be declared with the `@TraditionalJavaMethod` attribute.

4. Every traditional Java method must be declared with the `@ScopedPure` annotation and must not have the `@CallerAllocatedResult` annotation.
5. A traditional Java method only accesses instance fields of its own object (`this`), static variables associated with its own class, and the elements of arrays that are referenced from its local, instance, or class static variables. It is not allowed to access instance fields of other objects or static fields of other classes.