Reusable Libraries for Safety-Critical Java

Juan Ricardo Rios
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: jrri@dtu.dk

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: masca@dtu.dk

***Abstract*—The large collection of Java class libraries is a main factor of the success of Java. However, these libraries assume that a garbage-collected heap is used. Safety-critical Java uses scope-based memory areas instead of a garbage-collected heap. Therefore, the Java class libraries are problematic to use in safety-critical Java.**

We have identified common programming patterns in the Java class libraries that make them unsuitable for safety-critical Java. We propose ways to improve the libraries to avoid the impact of the identified problematic patterns. We illustrate these changes by implementing a total of five scope-safe classes from commonly used libraries.

***Index Terms*—Real-time systems, Java, Safety-critical systems, Safety-critical Java**

I. Introduction

As real-time systems grow in complexity, Java becomes appealing for their development, resulting in the appearance of real-time Java profiles. One of these profiles, the safety-critical Java technology specification (SCJ) [18], has been developed for certifiable safety-critical systems. SCJ is a specialization of the Real-Time Specification for Java (RTSJ) [4] and it has inherited the concept of scoped memories, i.e., regions of memory that are not managed by a garbage collector (GC).

The scoped memory model of SCJ is one of its most important features and perhaps the most difficult to use correctly. References between objects have to follow a strict set of rules. This memory model complicates the use of standard Java class libraries (JCL) in application development, because these libraries were developed under the assumptions that they run on a system with a GC and can have unrestricted references between objects. Therefore, it is important to find ways to modify standard libraries, so they work well within the SCJ memory model.

This paper explores some of the most common programming patterns and idioms present in OpenJDK6's standard JCL. We identify patterns and idioms that are problematic for SCJ applications and we present different ways to mitigate the impact of these patterns and idioms. In addition, we create and test five scope-safe classes representative of three of the most commonly used libraries: `java.util`, `java.lang`, and `java.io`. Focus has been on minimizing the need for changes and modifications with respect to the original implementation.

This paper is organized in eight sections. Section II introduces the basic concepts and the programming model of SCJ. Section III presents related work. Section IV presents a study of problematic programming patterns and idioms in the JCL. Section V analyses the issue of obtaining re-usable libraries by modifying or rewriting portions of the JCL. Section VI gives details on our implementation of scope-safe representative classes of the `util`, `lang` and `io` packages. Section VII discusses further observations based on our analysis and implementation. Section VIII concludes the paper.

II. Safety-Critical Java

The expert group for safety-critical Java develops the SCJ specification [18] within the Java community process (JCP) under specification request number JSR 302. The SCJ specification is a subset of the RTSJ [4]. To cover different levels of complexity, SCJ defines three compliance levels: level 0 is a single-threaded cyclic executive, level 1 a single mission with a preemptive scheduler, and level 2 allows nested missions and usage of an adapted version of RTSJ's `NoHeapRealtimeThread`. In SCJ, concurrent tasks are called managed schedulables. With respect to memory areas, all three levels support immortal memory, mission memory, and thread private scopes. Level 2 allows nested mission memories.

A. Missions and Scheduling

SCJ defines the concept of a mission. A mission consists of a set of managed schedulables and a mission memory.[1] In a level 1 application, which is the focus of this work, managed schedulables are called handlers. The number of handlers for a mission is fixed. Handlers are either periodic or event-triggered. Either an application or an interrupt handler can trigger the event for an event-triggered handler.

Handlers use the mission memory to share data. A mission has three phases: initialization, execution, and cleanup. The SCJ implementation creates the mission memory before executing the initialization phase. Within the initialization phase the application allocates shared data in mission memory, and creates and registers all handlers. The SCJ implementation *starts* all handlers on the transition to the execution phase. The application cannot create and register new handlers in the execution phase. In the execution phase handlers allocate temporary objects in handler-private memory. Allocation in mission memory is not prohibited; however, it is strongly discouraged. Objects allocated in immortal or mission memory

[1]SCJ Level 2 also allows managed threads.

during the mission phase are not reclaimed within the mission phase. If the lifetime of those objects is not the application or mission lifetime then this is a memory leak and the application may run out of memory. After the cleanup phase, the SCJ implementation clears the mission memory and a new mission can be started.

An SCJ application does not contain a `main()` method. Instead, a class that implements the `Safelet` interface, one or more classes that extend `Mission`, and at least one class that extends the `MissionSequencer` represent the SCJ application.

B. Memory Model

SCJ defines three memory areas: immortal memory, mission memory, and anonymous private scope memories. Immortal memory is, like in the RTSJ, for objects that live for the whole application, which might consist of several missions. Mission memory represents a scoped memory that exists for the lifetime of a mission and is the main memory for data exchange between handlers. Each handler has an initial private scope, that the infrastructure enters on each release and exits at the end of each release. The handler can enter nested private scopes. These private scopes are anonymous, as the application code has no access to a `ScopedMemory` object that *might* represent this private memory.

III. Related Work

Because safety-critical Java is new, the efforts to develop libraries for real-time Java profiles have focused on the RTSJ. However, previous work on the RTSJ can be related to SCJ as these studies have aimed at eliminating scoped-memory related errors.

In [8], the authors describe some of the challenges faced while implementing IBM's WebSphere, a RTSJ-compliant commercial Java virtual machine. One of the challenges here is integration with the existing JCL. Integration is challenging because objects are allocated in the memory area where the current thread runs, thus making all classes in the JCL potentially unsafe for shared use between different types of threads.[2] WebSphere provides a small subset of classes that are safe for shared use between RTSJ threads. In the context of the WebSphere JVM, "safe" means free of throwing `MemoryAccessError` exceptions, i.e., errors that are thrown when attempting to refer to an object in an inaccessible memory area.

Automatic identification of no-heap safe classes is the topic of [9]. Dibble presents a taxonomy to classify existing classes according to the degree with which they can be shared by all types of RTSJ threads. This taxonomy is based on the existence of static or instance variables that can store references to objects in heap and no-heap memory areas. A list of potentially unsafe classes, obtained through static analysis, is also presented. Dibble's analysis identifies all classes that have non-final static reference fields as unsafe.

[2]RTSJ defines `RealtimeThread` and `NoHeapRealtimeThread` in addition to standard Java threads.

The Javolution project [7] is one of the first attempts to produce an extended library of reusable, time deterministic, no-heap safe classes. In that project, issues such as sharing objects between scopes are eliminated. In addition, extra steps, e.g., explicitly switching between memory areas, are handled automatically. Javolution allows dynamic resizing of collections and allocates the required extra storage from immortal memory. Allocations in immortal memory can become a memory leak when elements are removed from the collection. Javolution's dynamic memory allocation and reliance on exception handling makes static program and worst-case execution time analysis difficult.

In [13] the development of reusable libraries targeted for real-time Java is presented. The authors present classes that can be used as drop-in replacements for three types of collection classes: `List`, `Set`, and `Map`. The authors' approach is based on recycling objects from a fixed-size pool of objects. As a consequence, operations such as insertion or deletion can be bounded and unpredictable resizing operations are avoided. Nevertheless, because the elements of a collection must be mutable, users have to provide their own way in which elements can change state. This work focuses on known execution times and memory consumption rather than on ensuring scope-safety.

The focus of the mentioned studies has been on performance; scope-safety is treated by avoiding assignments to heap memory; or provide safe subsets of classes with non-deterministic behavior. In contrast, our work concentrates on scope-safety by analyzing design patterns and idioms. We also explore deterministic behavior, both for execution time and memory consumption.

IV. Standard Java Libraries Under the SCJ Memory Model

The standard JCL was not developed to be used in the SCJ memory model. The JCL is based on a system where objects are allocated to the heap and are automatically deallocated by a GC. Furthermore, objects can refer to each other unrestrictedly. These assumptions are no longer valid in SCJ as its memory model eliminates the heap and the GC. In addition, as objects may have different lifetimes, scope allocations restrict how objects can refer to each other.

As a result, the different programming idioms and patterns used in the JCL can lead to memory leaks and illegal reference assignments. The remainder of this section presents a more detailed study of programming idioms and patterns present in three of the most commonly used JCL: `java.io`, `java.lang` and `java.util`.

A. Lazy Initialization

This pattern delays the initialization of a field that contains a reference to an object until the field is used for the first time. This pattern is used for two purposes: 1) to save memory in case the field is never needed, and 2) to break circularities in class initialization [3]. The problem with this pattern is that the object referred to by the lazily initialized field will be created in the scope of the first handler that uses it. This scope and the scope where the object with the lazy initialized field was allocated may not be the same.

As an example, consider the `keySet()` and `values()` methods in the `AbstractMap` class of the `java.util` package. These methods provide different views of the objects contained in a particular map implementation such as `HashMap` or `TreeMap`; they return a `Set` and a `Collection` object respectively. To ensure referential integrity, a call to these methods should be done from a scope that encloses the scope of an `AbstractMap` subclass instance.

The singleton pattern is as well a version of lazy initialization. The creation of singleton objects is prone to breaking referential integrity in SCJ. The singleton pattern under the scoped memory model of RTSJ has been analyzed in [5]. The solution, which can also be applied to SCJ, consists of explicitly allocating the singleton instance in immortal memory by using the available immortal memory API methods.

B. Dynamic Resizing

When a structure grows beyond its current capacity, it needs a size adjustment to accommodate new elements. Resizing involves creation of a new and larger array to accommodate the previous elements and the new ones. The old array is dereferenced leading to a memory leak. The new array, created in the scope of the caller, may eventually be referenced from an object in a different scope, thereby potentially creating an illegal reference assignment. This situation is illustrated in Figure 1a, where a method adds an object to a full collection. The objects in the figure are annotated with the scope where they are allocated: MM stands for the shared mission memory and PM for a private memory.

This method is called from a private memory, PM, while the object to be added lives in mission memory, MM. In Figure 1a, the container array ends in a scope that will be inaccessible to other handlers, even though it is perfectly legal for all handlers to access the elements of the array. This is referred to as polluted containers in [8].

An example of this situation is found in the `Vector` class. When an element is added, the `ensureCapacity` method is used to resize the collection if necessary. In case there is no resizing, the element to be added must be allocated in the same scope or in an outer nested scope from where the `Vector` object is allocated.

A similar situation occurs with some of the methods in the `StringBuffer` and `StringBuilder` classes. Concatenation and append operations may need resizing while character replacement operations will always create new character arrays.

C. Objects Used in Mixed Contexts

Modification of JCL objects shared between handlers requires special care because most of the methods may create new objects in the scope of the caller. Consider for instance the following two examples:

- The `addElement(Object obj)` method of the `Vector` class. The already existing object to be added to the collection should reside in the same or in an outer nested scope as the `Vector` object. As long as the addition of the new element does not exceed the capacity of the collection, the method can be called from within any scope.
- The `add(E e)` method of the `LinkedList` class. This method will create a wrapper object. This wrapper object is used to store book keeping information (e.g., references to the next and previous elements in the list) and a reference to the actual element to be added. There are two requirements in this case: 1) call the method from the same memory, or an outer nested scoped memory, as the one in which the `LinkedList` object is allocated, and 2) to ensure the element is added in the same scope or an outer nested scope from where the method is called. Neglecting to do this may result in contamination of the container array with an object allocated in a scope that is inaccessible to other handlers. This situation is referred to as polluted objects in [8] and is illustrated in Figure 1b.

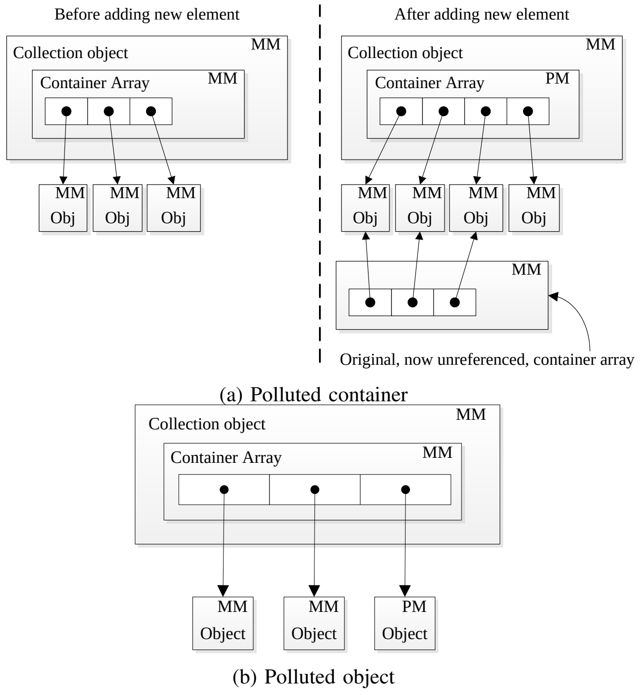


Fig. 1. Effects of using shared objects from different scopes. Scoped memories from where objects are allocated are represented as MM for mission memory and PM for private memory

D. Iterators

It is common to use an iterator pattern in collection classes to traverse and access elements of a container. This pattern is used in base classes such as `AbstractMap` and `AbstractList`. Iterator patterns require the allocation of `Iterator` objects that may lead to memory leaks if used within the shared memory areas (e.g., during initialization phase) or if used repeatedly. Iterators also require additional synchronization considerations by the application developer.

E. Loop Bounds

Although not related to scope-safety, the methods in software libraries intended for real-time systems must have predictable execution times. Unbounded loops are a concern for real-time systems as worst-case execution time (WCET) analysis tools cannot automatically extract loop bounds. For correct analysis, loops must be annotated manually with bounds.

F. Exceptions

Throwing exceptions may involve the creation of an exception object in the current allocation context. Even if the scope has been sized to include the effects of any thrown exception (that is, not only considering the normal execution path), there is still the risk of ending up with illegal assignments if the exception is thrown in an inner scope and propagated to be handled in an outer scope. Furthermore, it is desirable to avoid exception propagation as this may introduce program paths that are complicated to analyze [18].

G. Use of java.lang.reflect

The use of the reflection API can be dangerous in real-time systems. In addition to "loosing all the benefits of compile-time type checking, including exception checking" [3], private fields and methods can be accessed. In SCJ, the use of the reflection API is severely restricted. However, in JCL a number of methods, like `toArray()`, use the reflection API.

V. Reusable Libraries in SCJ

Real-time and safety-critical systems are typically less dynamic and more restricted than non real-time or non safety-critical systems. Such characteristics allow for certain restrictions and modifications to be made in library code in order to achieve the following characteristics:

- **Maintain referential integrity.** Referential integrity concerns the avoidance of throwing illegal assignment exceptions. It is important that SCJ library classes are aware of the scoped memory where method arguments, returned results, and objects allocated in methods reside.
- **Predictable memory consumption.** The size of a scoped memory area has to be provided when the area is created. It is therefore important to know how much memory will be allocated in the specific scoped memory. Libraries with predictable memory consumption help to size scoped memory areas in such a way that allocation demands can be met at all times during the execution of a program.
- **Predictable worst-case execution time.** Predictable execution time in library code is essential for calculating the WCET of an application. In turn, WCET values are used as input for the schedulability analysis.

The SCJ specification provides a list of class libraries for safety-critical applications defined with respect to the JDK 1.6. This set of core libraries is kept as small as possible by restricting the use of certain methods and fields. The goal is to reduce size and complexity to decrease the certification costs of applications. A summary of these classes is presented in Table I and Table II.

TABLE I
JAVA.IO CLASSES ALLOWED BY SCJ. NOT SHOWING EXCEPTION CLASSES.

Class Name	Relation to JDK 1.6	Reusability type
Closeable	Same	N/A
DataInput	Same	N/A
DataOutput	Same	N/A
Flushable	Same	N/A
Serializable	Same	N/A
DataInputStream	Same	Instance unsafe
DataOutputStream	Same	Instance unsafe
FilterOutputStream	Same	Instance safe
InputStream	Same	Instance safe
OutputStream	Same	Instance safe
PrintStream	Same	Instance unsafe

TABLE II
JAVA.LANG CLASSES ALLOWED BY SCJ. NOT SHOWING EXCEPTION CLASSES.

Class Name	Relation to JDK 1.6	Reusability type
Appendable	Same	N/A
CharSequence	Same	N/A
Comparable	Same	N/A
Runnable	Same	N/A
Boolean	Same	Instance safe
Byte	Same	Instance safe
Character	Restricted	Instance safe
Class	Restricted	Instance unsafe
Double	Same	Instance safe
Enum	Restricted	Instance safe
Float	Same	Instance safe
Integer	Same	Instance safe
Long	Same	Instance safe
Math	Same	Instance unsafe
Number	Same	Instance safe
Object	Restricted	Instance safe
Short	Same	Instance safe
StackTraceElement	Same	Instance safe
StrictMath	Same	Instance unsafe
String	Restricted	Instance safe
StringBuilder	Restricted	Instance unsafe
System	Restricted	Instance unsafe
Thread	Restricted	Instance unsafe
Thread.UncaughtExceptionHandler	Same definition	N/A
Throwable	Restricted	Instance unsafe
Void	Same	Instance safe

Table I lists classes from the `java.io` package and Table II, from the `java.lang` package. In this work, the subset of already defined classes is used as a starting point either to modify or to create new safe classes, such as the ones developed for the `java.util` package. For this package only the `Iterator` interface is provided in the SCJ specification.

A. Analysis of Standard Java Class Libraries

An analysis of the classes defined in JSR-302, using the OpenJDK's (version 6) source code, was performed to:

- Classify a standard implementation of the classes allowed by JSR-302 according to the taxonomy described in [9]. This classification is performed to provide an estimate of the degree of reusability of unmodified classes.

- Locate the points where memory allocations take place. In order to provide bounds on memory consumption it is important to know how memory is being used and, whenever possible, to provide rules, restrictions, or modifications that prevent unbounded memory allocations.

In [9], classes are cataloged according to whether or not they can be considered as no-heap safe. No-heap safe means that a particular class can be used concurrently by both *heap* and *no-heap* threads without the risk of storing references to heap-allocated objects.

To adapt this classification to SCJ, we abandon the concept of *heap* threads, as SCJ does not allow the use of heap memory. In addition, what is referred to as code executed by *no-heap* threads in [9] translates into periodic or aperiodic event handlers. Our classification is for scope safety (i.e. without the risk of generating illegal references) and has the following two categories:

- **Instance safe:** A class instance can be shared by different event handlers or allocated in a handler's private memory and used in a nested private memory without the risk of generating illegal references. Few classes are expected to fall into this category, as they need to have only final reference fields.
- **Instance unsafe:** Event handlers cannot safely share instances of this class.

The results of this classification are shown in Table I and Table II. The following rules were used for classification:

1) A class is instance safe if all of its reference fields are declared as final.
2) A class with non-final reference fields assigned only at class initialization (execution of its `<clinit>` method) can be considered to be instance safe.
3) A class with non-final reference fields or with methods that perform array reference assignments are instance unsafe.
4) A class inherits its superclass classification. For example, if class B extends class A and class A is instance unsafe, then class B is also instance unsafe. An unsafe class B can, however, have a safe class A as parent.

During the analysis of the JCL classes, we noted all allocation places and reference assignments. Once a problematic part in the code was located, we proceeded to implement our solutions, which combine techniques such as restricting the size of different structures, changing between scopes, running specific code in nested scopes, and recycling of objects by memory pooling.

B. Mitigating the Effects of Design Patterns and Programming Idioms on Scoped Memory

This section presents a number of recommendations on how to avoid the problems associated with the previously described design patterns and programming idioms.

1) Lazy Initialization: Illegal references can be avoided by creating the lazy object either in immortal memory or in the same scope as the object containing the lazy initialized field. One possible solution is to execute the object creation code in class initializers, which will execute in immortal memory, as it is done in [2]. This approach works for objects that should be accessed during the whole application lifetime, such as a `Properties` object.

For objects that are only used by specific missions, another approach is to create the lazy initialized object when the instance of the class containing the field is created, i.e., as part of the object's constructor. This is the approach followed in the implementation of our libraries.

Another solution would be to change the allocation context to the memory area where the object with the lazy initialized field is allocated. The lazy object can then be safely created.

2) Dynamic Resizing and Elimination of Unintended References: The fundamental problem here is that for every expansion of a data structure, a new storage element (usually an array of objects) is created in the context of the caller while the previous storage element gets dereferenced, producing a memory leak.

One option to avoid structures to dynamically resize is to limit the maximum amount of elements the structure can hold. This seems too restrictive, but it is likely that most data structures for hard real-time systems are big enough to hold all of the intended elements.

Another option is to change the allocation context before the expansion to guarantee that the storage element is created in the same memory area as the data structure. However, memory leaks created by de-referencing the old storage element cannot be avoided.

The approach we use in our libraries is to limit the maximum amount of elements a structure can hold and to recycle objects from a pool of objects. Objects are allocated from the pool when they are needed and returned to the pool when removed from the data structure. In this way we avoid the risk of creating the storage element in another region and the memory leaks associated with removing or replacing elements.

3) Objects Used in Mixed Contexts: This is perhaps one of the most difficult issues to address in library code since there is no easy way to ensure that caller-allocated results or arguments to methods reside in appropriate scopes.

Illegal references come from field or array stores, either to new objects or to objects referenced by arguments passed to methods. Arguments must reside in the same scope as the `this` argument,[3] or in an outer scope (e.g., for setter methods). One option for ensuring that library code enforces this requirement is to provide dynamic guards (see the memory annotations appendix of [18]). A dynamic guard is a conditional statement used to test that the parenting relationship of the scopes in which arguments reside is appropriate to avoid illegal references.

Another option is to change the allocation context to execute a method (or part of it) in the scope of the `this` argument. This is particularly useful if the method requires the creation of auxiliary objects as in a `HashMap` or a `LinkedList`,

[3] `this` is a reference to the object whose method is being called.

where additional objects are created to store bookkeeping information.

For our libraries we reuse objects from pools and require that instances of library classes are created in the same or an inner nested scope as the pool.

4) Iterators: One option for the use of this pattern is to pre-allocate single iterator objects when collection objects are created, as in [13]. However, we consider that not much is gained by pre-allocating single iterator objects per collection, because this solution only works on single-threaded applications. As described by the authors of [13], if a handler requests use of the iterator object after another handler has gained access to the single iterator object, then the iterator object's state is reset causing runtime errors.

The use of iterators may become problematic if used multiple times. In this case, a better idea is to use iterators within nested private memory areas.

5) Loop Bounds: Most of the programming idioms, used for loops in the JCL, are not friendly for our current WCET analysis tool, WCA [23]. The exit condition in loops may depend on boolean flags or on a value that the data flow analysis in the tool is not able to propagate (e.g., internal manipulation of an array size). The most common case was loops in which the stop condition is a boolean check for a `null` element. In our libraries, loops are limited to a maximum number specified as an argument in the instance constructor. This argument can be propagated by the data flow analysis of the WCA tool provided it is not modified inside the library code. To enforce this restriction, such arguments are declared final.

6) Exceptions: Safe exception handling in SCJ requires the observation of the following rule:

- Propagation of exceptions to a scope different from the one in which it was originally allocated causes a `ThrowBoundaryError` exception. In this way, scoped memory errors such as illegal reference assignments are avoided [18].

According to SCJ's specification, there are no special requirements for the allocation of exception objects. These objects can be created in the current scope with the `new` keyword; in a different scope after a change of allocation context; or they can be pre-allocated. Our libraries pre-allocate exceptions in immortal memory and when necessary and possible, library exceptions are created with a constant string message describing the cause of the error. Unnecessary memory allocations that come from concatenation of strings are thus eliminated.

VI. Implementation

In this section we describe the implementation of five representative classes of the standard Java libraries. The implemented classes were adapted for use in safety-critical Java in accordance with the solutions outlined in Section V-B. Of the five classes, three are defined in the safety-critical Java specification (`AbstractStringBuilder`, `StringBuilder`, and `DataInputStream`) while the other two (`Vector` and `HashMap`) are not part of the safety-critical Java specification. Nevertheless, we consider them to be important in the development of reusable software components.

A. *AbstractStringBuilder and StringBuilder*

Within these classes, memory consumption is related to the size of the character array backing those types of objects. To provide bounds on memory consumption, we limit the maximum number of characters any of those classes can hold. This maximum length can, e.g., be the size of log messages or to the size of a text to be displayed. This decision can further be supported by considering that safety-critical programs typically do not incur in extensive text processing or file manipulations [18].

Limiting the maximum number of characters also has the following two benefits: 1) Resizing operations are not needed and 2) we can have bounds on methods that iterate over the character elements (through annotations, see Section V-B5).

B. *DataInputStream*

The `java.io` package contains classes to perform input and output operations in Java. We focus on the `DataInputStream` class because the additional classes in this package defined in JSR-302 (see Table I) are only wrapper classes.

Memory allocations within classes in this package come from re-sizable arrays that are used for temporary processing or to perform buffered reads and writes. Methods that perform temporary processing can be executed inside nested scopes. In this way, array resizing is allowed if needed and any temporary array will be collected when leaving the nested scope. As an example, Figure 2 shows our modified version of the `readUTF(DataInput in)` method (lines 4–12) from the `DataInputStream` class. This method reads a representation of a character string encoded in modified UTF-8 format[4] and uses two arrays of up to 65,535 bytes for temporary processing. The `readUtfHelper` inner class encapsulates in its `run()` method (lines 21–26) the code of the original `readUTF` method and executes it in a nested private memory. The modified version needs an additional parameter to set the size of the nested scope because the memory consumption of objects is implementation dependent. The `run()` method also handles the additional scope change needed to return the resulting string object into the context of the caller (lines 29–34). The scope change is made using the SCJ's `executeInOuterArea` method, which moves the current allocation context one level up in the scope stack.

Resizing operations can also be avoided by using working arrays and buffers of size equal in size to the worst-case expected length. The drawback of this approach is that arrays that are only needed for a few methods will be created for every instance and will most likely be poorly utilized.

[4]See http://docs.oracle.com/javase/6/docs/api/java/io/DataInput.html#modified-utf-8 for a description of the modified UTF-8 format

```
public class DataInputStream ... {
  /* Other methods of DataInputStream class */
  ...
  public static  final  String  readUTF(DataInput in, long size)...  {
    ReadUtfHelper readUtfHelper = new ReadUtfHelper();
    readUtfHelper.in = in;
    ManagedMemory.enterPrivateMemory(size, readUtfHelper);

    /* Return String  lives  in  the context of  the  caller   */
    return readUtfHelper.retString;
  }
}

class ReadUtfHelper implements Runnable {
  String  retString;
  DataInput in;

  @Override
  public void run()  {
    try  {
      /* Begin of  original  code of readUTF method */
      int  utflen  = in.readUnsignedShort();
      byte[]  bytearr  = new byte[utflen];
      final  char[]  chararr  = new char[utflen];
      ...
      /* End of  original  code of readUTF method */

      /* Return a String object  in  the scope of the  caller   */
      ManagedMemory.executeInOuterArea(new Runnable() {
        @Override
        public void run()  {
          retString  = new String(chararr,  0,  count);
        }
      });
    } catch (IOException e)  {...}
  }
}
```

Fig. 2. Example of a method modified to run in a nested private scope.

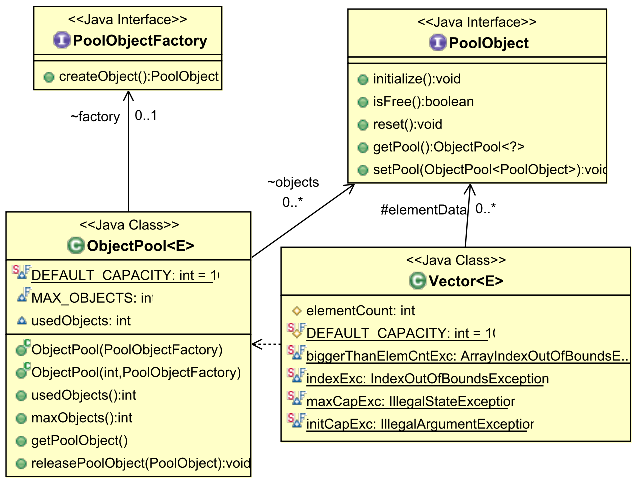
C. Vector and HashMap

These two classes are representative for the `java.util` package. The safety-critical Java specification only provides the definition for the `Iterator` interface. However, as these classes are useful for building reusable software components, we decided to provide some implementation examples for this package.

The modified `Vector` class, illustrated in Figure 3, shows how, through the use of object pooling [17], a solution for most of the problems mentioned in Section IV can be provided.

Our classes are restricted to store only elements belonging to a pool of pre-allocated objects. When elements are removed or replaced, they are returned to their corresponding pool, their state is reset, and they are marked as available for reuse. To add an element, one must first obtain a free object from the pool of pre-allocated objects and then add it to the `Vector`.

An `ObjectPool` instance is created with a fixed number of objects. The number of objects is passed as a parameter in the constructor (if omitted, a default value is used). The elements belonging to the pool are created when the `ObjectPool` is instantiated and a `PoolObjectFactory` provides a strategy to define how they will be created (through the `createObject()` method). Retrieving a free object from the pool is done by calling the `getPoolObject()` method which in turn will



<<Java Interface>>
PoolObjectFactory
createObject():PoolObject
~factory
0..1
<<Java Interface>>
PoolObject
initialize():void
isFree():boolean
reset():void
getPool():ObjectPool<?>
setPool(ObjectPool<PoolObject>):void
~objects
0..*
#elementData
0..*
<<Java Class>>
ObjectPool<E>
DEFAULT_CAPACITY: int = 1
MAX_OBJECTS: int
usedObjects: int
ObjectPool(PoolObjectFactory)
ObjectPool(int,PoolObjectFactory)
usedObjects():int
maxObjects():int
getPoolObject()
releasePoolObject(PoolObject):void
<<Java Class>>
Vector<E>
elementCount: int
DEFAULT_CAPACITY: int = 1
biggerThanElemCntExc: ArrayIndexOutOfBoundsE...
indexExc: IndexOutOfBoundsException
maxCapExc: IllegalStateException
initCapExc: IllegalArgumentException

Fig. 3. Class hierarchy of the modified `Vector` class.

call an initialization hook, the `initiaize()` pool object's method, from the returned free object. When returning an object to the pool, the `releasePoolObject()` method calls the termination hook method, `reset()`, of the object being returned. It is important to note that the `ObjectPool` as well as the `Vector` can only have elements of the `PoolObject` type or subtype. This restriction is enforced through generics. Method `getPool()` returns a reference to the pool an object belongs to (it is possible that a `Vector` contains elements from different pools).

Memory allocations in this class are a result of resizing the storage element, throwing new exceptions, and from the use of iterators. Resizing is avoided by fixing the size of the internal storage element. The fixed-sized storage element together with the fixed-size pool of objects implies that the maximum number of elements that the collection can store must be known in advance. Creating new exception objects is avoided by pre-allocating them in immortal memory during class initialization.

The `HashMap` class presents additional complexity due to a double object-pool management, one for the objects representing entries in the bucket list (implementations of `Map.Entry`) and one for the `Map` objects that are to be added to the hash map.

D. Comparison with JCL

Table III shows a comparison between our modified classes and the original JDK 6 implementation. We compare the lines of code (LoC), number of fields (NoF), methods (NoM), modified methods (NoMM) and constructors (NoC). For the number of methods, numbers in parenthesis represents the number of methods belonging to the public API. For the number of modified methods, the two numbers represent respectively the total number of modified methods and the number of new methods needed for extra functionality. The number in parenthesis indicates the modified methods of the public API.

TABLE III
Comparison of the implemented classes with JDK's implementation. LoC = Lines of code, NoF = Number of fields, NoM = Number of methods, NoMM = Number of modified methods, and NoC = Number of constructors. Number of public methods is in parenthesis. For the NoMM in the implemented classes, the numbers represent modified methods and additional methods respectively.

Class Name	**JDK 1.6**				**Implemented Classes**				
	LoC	NoF	NoM	NoC[1]	LoC	NoF	NoM	NoMM	NoC[1]
java.lang.AbstractStringBuilder	447	2	52 (50)	2	237	6	29 (26)	11 (11) / 0	2
java.lang.StringBuilder	189	1	38 (35)	4	119	1	19 (18)	1 (0) / 0	4
java.io.DataInputStream	212	4	18 (18)	1	108	5	17 (17)	1 (1) / 0	1
java.io.DataInputStream$ReadUtfHelper[2]	–	–	–	–	77	2	1	0 / 1	1
java.io.DataInputStream$1[3]	–	–	–	–	6	0	1	0 / 1	0
java.util.Vector	322	4	48 (45)	4	228	7	36 (35)	16 (15) / 1	2
java.util.Vector$1[4]	14	1	2	0	14	1	2	0 / 0	0
java.util.Vector$Itr	33	3	4	0	33	3	4	0 / 0	0
java.util.Vector$ListItr	43	0	6	1	43	0	6	0 / 0	1
java.util.HashMap	356	10	36 (13)	4	269	22	24 (11)	11 (5) / 1	2
java.util.HashMap$Entry	48	4	8	1	57	5	9	0 / 1	1
java.util.HashMap$EntryIterator	5	0	1	0	5	0	1	0 / 0	0
java.util.HashMap$EntrySet	21	0	5	0	24	0	5	1 / 0	0
java.util.HashMap$HashIterator	41	4	3	1	43	4	3	0 / 1	1
java.util.HashMap$KeyIterator	5	0	1	0	5	0	1	0 / 0	0
java.util.HashMap$KeySet	17	0	5	0	27	0	5	1 / 0	0
java.util.HashMap$ValueIterator	5	0	1	0	5	0	1	0 / 0	0
java.util.HashMap$Values	14	0	4	0	14	0	4	0 / 0	0

[1] Zero means only the default implicit constructor. [2] Not in JDK6, encapsulates SCJ functionality.
[3] Not in JDK6. Anonymous `Runnable` class. [4] Anonymous `Enumeration` class.

None of the original interfaces implemented by the modified classes were changed to keep compatibility as close to standard Java applications as possible. However, a number of interfaces implemented by the original JDK 6 classes are not allowed by SCJ. For example, the `Cloneable` interface is not allowed because of its weak definition. Therefore, the `clone()` method will throw a `CloneNotSupportedException`. The `Serializable` interface is left as part of the implemented classes for compatibility with standard Java, but its inclusion or exclusion has no effects on a SCJ application.

Some of the constructors had to be eliminated due to their underlying algorithm. For example, the `Vector(Collection<? extends E> c)` is omitted as the size of the final array element created cannot be guaranteed. This is because the constructor relies on the size of the `Collection` parameter, which can be modified while the constructor is still executing. As a result of modifying the collection, array objects of varying size can be created and it is not known which one will be returned to be used as the storage element for the `Vector`. Therefore, methods operating on collections are committed. Constructors that require parameters for resizing, and methods for resizing and ensuring capacity are also omitted. Similar restrictions for constructors and resizing methods applies for `HashMap`.

The reduction in the number of methods and lines of code is a consequence of: (1) the elimination of methods that are no longer needed (e.g., resizing methods) and (2) the reduced number of methods allowed by the SCJ profile. The increase in the number of fields in the implemented classes is because pre-allocated exceptions are included as class variables.

E. Testing

To check that the implemented classes are functionally correct, a set of test cases were developed using the standard JUnit Java framework. The test cases allow checking of the majority of methods, except for those that involve parts of the SCJ API. For methods including parts of the SCJ API, the Java processor JOP [22] was used. JOP provides an implementation of Level 0 and Level 1 of the safety-critical Java profile [24].

To test that there are no reference assignment errors, we used the private memory analyzer tool described in [6] together with the reference assignment check facility of JOP. Memory consumption is checked by measuring the amount of memory used by the different methods in the libraries. Memory measurements are only carried out on methods that have memory allocations identified by the analysis in Section V-A. Our synthetic test-bench for this part of the testing is a SCJ application with shared data structures in mission memory that are accessed from a set of `PeriodicEventHandlers` (PEH).

As a final step, two additional, more complex applications were tested. First, the new `java.util` collection classes were used as drop-in replacements for the shared data structures in the parallel miniCDj benchmark [25]. The miniCDj benchmark is a SCJ version of the benchmark described in [16]. miniCDj implements an air traffic controller simulator that generates artificial radar frames containing airplane positions. The frames are processed to detect possible collisions. For the parallel version, one PEH generates the radar frames and a selectable number of `AperiodicEventHandlers` process them.

The second test uses a SCJ version of a watchdog application running on top of the Cubesat space protocol (CSP) [1].

CSP is a network-layer protocol designed at Aalborg University that is used by small space-research satellites called Cubesats. The watchdog application has one PEH that sends packets to a set of nodes and one PEH functions as a router. An interrupt service routine adds incoming packets into the router's queue. For our experiments, one of the CSP nodes was an on-board satellite computer used in commercial Cubesats. The application has three main data structures that handle packets, connections and sockets. We replaced the data structure used to handle packets with our `Vector` implementation. Functionality was not affected nor were any scope-related issues introduced. An interesting result was a reduction of almost 7% of the use in immortal memory; in the original implementation, the router PEH needs additional packet-managing structures.

VII. Discussion

A. Loop Bounds for Library Code

To automatically find loop bounds with our WCET tool, some loop exit conditions had to be changed. For example, a loop like: `for(Entry e = tab[i]; e != null; e = e.next)` was modified as:

```
Entry e = tab[i];
for (int i = 0; i < entries.length; i++){
  if(e == null) break;
  ...
  e = e.next;}
```

The worst-case iteration count, i.e. `entries.length`, can be propagated in the data flow analysis. The `break` statement avoids iterations being performed when they are not necessary.

A different approach that can be adopted is the use of standard Java annotations to pass symbolic information about loop bounds, as proposed in [12], where loops can be bounded with annotations of the form:

```
@LoopBound(max=elementCount)
for (int i = 0; i < elementCount; i++){...}
```

where the maximum number of iterations (*elementCount* in the example code above) is obtained from an annotation attached to the declaration of the class instance implementing the method with the loop. However, such non-standard annotations require the use of a modified Java compiler.

Annotations for loop bounds in `String` and `StringBuilder` objects are more difficult to handle in the same way. These objects can be created explicitly (with the `new` keyword) or when calling the `toString()` method, when declaring constant strings, as a result of concatenation with the "+" operator, etc. In these cases there is no easy way to propagate information about the internal character array size to the internal methods containing loops.

B. Programming Idioms and Patterns for WCET Analysis

Another issue found when performing WCET analysis of library code was the use of overridden implementations of `Object.equals()`. The problem here is that the call graph generated during the analysis contains cycles, and the WCA tool is not able to handle this type of recursion. Cycles in the call graph were observed in classes that use the `java.util.Map.Entry` inner class. Testing for equality between two entries requires testing for equality between the pair of key-value mappings contained in the entry. The key and value objects will call their own implementation of the `equals()` method. One possible solution could be to restrict the types of objects that can be stored as key-value mappings, and to implement a different form of equality that avoids the use of an overridden form of `Object.equals`. This is, however, too restrictive on the types of key-value objects that can be used in a map.

The delegation pattern also generates cycles in the call graphs. For example, the methods of the `AbstractList.Sublist` inner class make calls to the "real" implementation of a `List` passed as argument to the constructor. To avoid cycles, the annotation system proposed in [14] can be used to tighten the number of possible receiver types of a method invocation.

C. Application Developer Considerations

Many of the problems outlined in Section IV can be solved by modifications of the Java class libraries source code. Such modifications can minimize, but cannot completely eliminate, the occurrence of scoped memory protocol errors in a complete application. For example, it is the responsibility of the application developer to use caller allocated results or arguments to methods correctly, so as to avoid illegal assignments. We presented patterns for SCJ memory usage in [20]. Furthermore, a typing system based on annotations, such as the one described in [19], can be useful in this case. This typing system adds extra information about the scope of the different elements of an application, which can later be retrieved to perform static analysis. In addition, which restrictions exist for passed arguments and returned results should be considered when overriding library method implementations.

D. Certification Issues in Library Code

As a final remark, it is important to note that the use of reusable components and libraries for the development of safety-critical systems presents additional challenges, e.g., unused code from a library introduces code that is not traceable to requirements (dead and/or deactivated code). Certification standards, such as DO-178C [21], expect that code not associated to requirements is either eliminated or that requirements for the code are developed [11], [10]. Therefore, the benefits of re-usability should overcome the increased certification effort and associated costs.

Another type of issue is the encapsulation of data, as this complicates robustness testing [11], i.e., "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [15]. Robustness tests will not be able to access library-private data.

E. Stability Between Releases of the JDK

By comparing the source code between JDK6 and JDK7 of the modified classes in this work, we found very few changes. The changes concentrate in methods to ensure capacity, the use of the enhanced for-loop and generics. Those minimal changes made our libraries relatively stable between JDK's releases.

VIII. Conclusion

Standard Java class libraries were not developed to be used under the scoped memory model of safety-critical Java. They rely on automatic memory deallocation by a GC and objects that can reference to each other in an unrestricted manner. These assumptions are no longer valid in SCJ as its memory model eliminates the heap and the GC, and restricts the way in which objects can refer to each other.

We have analyzed and identified common problematic patterns present in three of the most used Java libraries: `java.lang`, `java.util`, and `java.io`. We have also provided modifications to the mentioned libraries that mitigate the problems introduced by the identified patterns.

Safety-critical systems are more restricted and less dynamic than non safety-critical systems. These characteristics allows for restrictions to be made for the implementation of reusable libraries for SCJ. We have adapted representative sample classes as a first step to developing reusable libraries for SCJ. The provided classes have predictable memory consumption, are WCET analyzable, and maintain referential integrity between objects created and used internally by the classes.

Acknowledgments

We thank Anders P. Ravn and Wolfgang Puffitsch for their insights and comments on this work.

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and has received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

Source Access

The modified classes presented in this work are part of the JOP distribution and implementation of SCJ. They can be downloaded from https://github.com/jop-devel/jop and are located in the `java/target/src/paper/libs/` directory.

References

[1] J. L. Andersen, M. Todberg, A. E. Dalsgaard, and R. R. Hansen. Worst-case memory consumption analysis for SCJ. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 2–10, New York, NY, USA, 2013. ACM.

[2] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 249–258, New York, NY, USA, 2007. ACM.

[3] J. Bloch. *Effective Java*. Addison-Wesley, Upper Saddle River, NJ, 2008.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] A. Corsaro and C. Santoro. Design Patterns for RTSJ Application Development. In R. Meersman, Z. Tari, and A. Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 394–405. Springer Berlin / Heidelberg, 2004.

[6] A. E. Dalsgaard, R. R. Hansen, and M. Schoeberl. Private memory allocation analysis for safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 917, 2012.

[7] J.-M. Dautelle. Javolution. http://javolution.org/, September 2012.

[8] M. Dawson. Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 241–247, may 2008.

[9] P. Dibble. No-Heap Safe Classes. http://www.rtsj.org/docs/noheapSafe1/Noheapsafeclasses4.html, August 2004. Retrieved on April 11, 2013.

[10] M. R. Elliott and P. Heller. Object-oriented software considerations in airborne systems and equipment certification. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 85–96, New York, NY, USA, 2010. ACM.

[11] Federal Aviation Administration. *Handbook for Object-Oriented Technology in Aviation (OOTiA)*. Federal Aviation Administration (FAA), Washington, D.C., USA, October 2004.

[12] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 47–57, St. Louis, MO, United States, April 2008. IEEE Computer Society.

[13] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. Toward Libraries for Real-Time Java. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 458–462, may 2008.

[14] E.-S. Hu, G. Bernat, and A. Wellings. Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems. In *Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, pages 109–116, 2002.

[15] IEEE. *IEEE-STD-610 ANSI/IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology*. IEEE, Feb. 1991.

[16] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. A family of real-time Java benchmarks. *Concurrency and Computation: Practice and Experience*, 23(14):1679–1700, Sept. 2011.

[17] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*. Wiley Software Patterns Series. Wiley, 2005.

[18] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-Critical Java Technology Specification, Public draft, 2013.

[19] K. Nilsen. A type system to assure scope safety within safety-critical Java modules. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 97–106, New York, NY, USA, 2006. ACM.

[20] J. R. Rios, K. Nilsen, and M. Schoeberl. Patterns for safety-critical Java memory usage. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 1–8, Copenhagen, DK, October 2012. ACM.

[21] RTCA. DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification, 2011.

[22] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[23] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40(6):507–542, May 2010.

[24] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 54–61, Copenhagen, DK, October 2012. ACM.

[25] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei. Refinement of the Parallel CDx. Technical report, University of York, Department of Computer Science, York, UK, 2012.