Patterns for Safety-Critical Java Memory Usage

Juan Ricardo Rios
Department of Informatics and Mathematical Modeling
Technical University of Denmark
jrri@imm.dtu.dk

Kelvin Nilsen
Atego Systems, Inc.
kelvin.nilsen@atego.com

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

Scoped memories are introduced in real-time Java profiles in order to make object allocation and deallocation time and space predictable. However, explicit scoping requires care from programmers when dealing with temporary objects, passing scope-allocated objects as arguments to methods, and returning scope-allocated objects from methods. To simplify the correct usage of scopes, programming patterns may be helpful. We present patterns for simple subroutines, sequences of subroutine calls, and nested calls, where the patterns avoid memory leaks and unnecessary copying of values. The patterns are illustrated by implementations in the safety-critical Java profile.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.2 [**Language Classifications**]: Programming Languages—*Object-oriented languages*

Keywords

Real-time systems, real-time Java, safety-critical systems

1. INTRODUCTION

Java, as a programming language and as a large collection of standard libraries, has proven to be a useful tool to increase programmer efficiency both for the development and maintenance of software [8]. The benefits provided by Java come from its high level object-oriented programming features and the use of an automatic garbage collector (GC).

In recent years, the use of Java for systems with real-time constraints has been enabled trough the definition of real-time Java profiles (e.g. [3], [12], and [5]). In those profiles, standard Java is restricted and extended with additional classes. One of such extensions is a real-time memory management strategy which avoids the use of an automatic GC, as some applications have real-time requirements hard to achieve with current garbage collection technology [15]. Memory allocations are performed in specific regions called scoped memories where objects are collectively deallocated at scope exit. In safety-critical Java (SCJ), memory management is the explicit responsibility of the application developer who has to be aware of the allocation context of objects. Maintaining referential integrity and the movement of data between memory areas are two of the most important issues introduced by the scoped model [13].

In a nutshell, moving data between scopes require a creative way of using the available SCJ memory API features. This API contains specific methods used together with a reference to a target memory to change the allocation context and safely create the returned objects.

In this paper we look into the expressiveness of the SCJ memory model and explore patterns how to use it. We present a collection of memory use patterns with increasing complexity that aim at helping in the development of SCJ applications. The focus is on how to pass arguments and return objects between private memories. As a result of our experiments, we report on two possible effects that may compromise the intended safety of the specification: the possibility to break the linear hierarchy in the scope parenting relationship and leaking memory references between private handlers. As a solution we suggest an API simplification that avoids references to memory areas at all.

The remainder of this paper is organized as follows: Section 2 presents work previously done regarding scoped memory use patterns. Section 3 gives an introduction to SCJ's memory model and the relevant API for this work. Memory use patterns specific to SCJ are the topic of Section 4. Discussion and observations based on experiments with SCJ memory patterns are presented in Section 5. Concluding remarks are presented in Section 6.

2. RELATED WORK

Benowitz and Niessner introduced patterns used for periodic activities as well as scope aware factories [1]. Returning objects are allocated in immortal memory using memory pools (fixed size objects) and memory blocks (byte arrays to allocate varying size objects). Memory pools in regions other than immortal memory are explored in [2]. This work describes the communication between several components participating in a real-time control loop. Each component is defined in its own scoped memory and has its own pool in a scoped memory other than immortal. Communication between components is performed by copying values. Recycling objects is an appealing solution to avoid running out

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark

of immortal memory. Nevertheless, in this work we explore other possibilities that include the use of SCJ's mission and private memories.

Pizlo et al. investigated design patterns for the RTSJ [14]. The authors document several design patterns for the effective use of scoped memory regions. From their work only the Scoped Run Loop pattern has a direct equivalent in SCJ. The rest are either RTSJ specific (e.g. Wedge pattern), use features not available in SCJ (e.g. Portals), or introduce violations to the reference assignment rules (e.g. Handoff pattern).

The patterns catalog in [1] is extended with another patterns collection in [4]. For this collection, the Memory Tunnel pattern is particularly interesting. It is used to move data between threads executing in different memory areas. It has however, the problem that it forces a "safe" violation of the reference assignment rules. It relies on a memory tunnel structure which is constructed using native methods to bypass the scope constraints regarding references. As SCJ is intended for certification under standards such as DO-178B, this behavior is likely to reduce the chances of passing any certification. Thus, we use a different approach to move data between scopes that goes through mission memory.

Kwon and Wellings propose to map memory areas to Java methods in a user-controlled fashion in [11]. Motivated by the overhead incurred when checking the single parent rule and cross-scope reference assignments [3], they propose a model that reduces the need for such checks. This is achieved by associating a memory region with annotated methods. Memory areas are entered when the method is invoked, effectively changing the allocation context. Objects created within this context are collected when the method returns. References to objects created outside the current method can be passed as parameters or as local objects in the enclosing object that the method belongs to. References to objects in a callee method are not allowed. If there is the need to return objects after a method is executed, it is done by specifying an additional parameter that can define where to store a returned object. The approach uses RTSJ scoped memories in a restrictive style that is very similar to SCJ private memories. The annotation facility hides the complexities of using the memory API to encapsulate methods and to return objects.

The Lifecycle Memory Managed Periodic Worker Threads pattern (LMMPWT) described in [6] presents an RTSJ framework where a group of no-heap periodic threads cooperate together to complete a task. This pattern focuses on object lifetime management and does not require an explicit understanding of scopes. Lifetimes assigned to objects are divided in four categories that can be directly related to the lifetime of immortal, mission, private and nested memory allocated objects in SCJ. Movement of data between scopes and creation of objects in arbitrary memory areas is done with an encapsulated use of the memory API. The drawback of this implementation is that it relies heavily on reflection for manipulation of objects. Note that the `java.lang.reflect` package is not part of the SCJ specification.

In [19], the authors introduce the concept of scoped types as a way to encapsulate scoped objects. Scope and portal classes are defined and associated with their defining packages. Nested scopes are in turn associated with nested packages. Accessibility of a scoped class is restricted to instances of classes allocated in the same or nested scopes. This work is later extended in [16] where the authors document a number of programming idioms to manipulate scopes.

3. SCJ MEMORY MODEL

SCJ uses a restricted version of the scoped memory model introduced by RTSJ [3]. In SCJ's memory model, the garbage collected heap memory is not needed as the use of `RealtimeThread` is not allowed. There are three different scoped memory areas with different requirements on the lifetime of objects they can hold, namely immortal memory, mission memory, and private scopes. Objects allocated in immortal memory remain valid until the JVM finishes, objects in mission memory are valid for the duration of the mission, and objects in initial private scopes are valid for the duration of one release of a handler. Immortal and mission memory areas are shared between handlers executing concurrently and private scopes can be entered only by a single handler.

Private memories can also have nested scopes to provide space for temporary computations. Nested scopes are created within the backing store reservation space of an event handler's private memory and can be entered only from the memory area where it was created. As an example, the computation of a Fast Fourier Transform (FFT) from a set of samples may create many objects that won't be needed after the result is computed. Any algorithm that consists of multiple processing phases, with the output of each phase serving as the input to the next, is a good candidate for the use of nested private memories to hold the temporary data used within each phase. JPEG encoding is an example of such an algorithm.

3.1 SCJ Memory API

Memory areas are constructed as instances of the `ImmortalMemory`, `MissionMemory` and `PrivateMemory` classes. The latter two extend a common SCJ class, `ManagedMemory`. Unlike the RTSJ, it is not possible to explicitly create instances of `MemoryArea` objects. The SCJ infrastructure instantiates `MissionMemory` and `PrivateMemory` objects in response to certain user requests.

Memory area objects represent an allocation context but the objects themselves are created outside the allocation context they represent e.g., the object representing the initial mission memory resides in immortal memory.

Memory areas are entered implicitly by the infrastructure, using `ManagedMemory.enterPrivateMemory()` to move into an inner nested scope, or with `executeInArea()` to enter an outer nested scope.

Objects are created in the allocation context where the private handler is executing but they can also be created in a different memory area with the help of `newInstance()` and `newArray()` methods.

It is possible to obtain references to memory areas with the methods `getMemoryArea(Object object)` and `getCurrentManagedMemory()`. Such references can be used in combination with the previously described methods to move to an explicit memory area or to indicate where an object is to be created.

4. SCOPED MEMORY USAGE PATTERNS

Patterns can be used to document expertise, provide solutions to recurring design problems, and can lead to more flexible and reusable software. In [9], a design pattern is defined as a solution to a problem in a context.

Based on experience, the main problems with the SCJ are establishing abstractions that allow safe and reliable integration of independently developed software components, abstractions that provide an assurance that integration of components does not introduce memory leaks and illegal references.

This work looks into the expressiveness of the SCJ memory model and explores patterns relating to the use of this memory model. The following section presents several scoped memory usage patterns that aim at helping in the development of SCJ applications. The focus is on how to pass arguments into and return objects from methods that change the allocation context to a nested memory.

4.1 The Basic Pattern

The organization of the event handlers and the initial private memory in SCJ can be considered as a basic, minimalistic pattern. It works quite well when the event handler will not return results or need arguments as parameters (e.g. no feedback is expected when sending control signals to actuators). In this case, it is not necessary to preserve results for the next time the memory area is activated. Implementation of this pattern is straightforward. A managed event handler overrides the `handleAsyncEvent()` method. Temporary objects are allocated in the initial private memory and the memory area is recycled at the end of the release. This pattern is called Scoped Run Loop Pattern in [14]. The difference between the SCJ pattern and the RTSJ pattern is that the memory area does not need to be explicitly created and entered, as this is all handled by the SCJ implementation.

4.2 Loop Pattern

Since private memory space is a scarce resource, applications should carefully use the memory to avoid running out of space. One way to recycle memory is to use nested private scopes. A nested private scope can be entered and exited several times during a release. The object representing the nested private scope is reused in the implementation to avoid generating garbage in the initial private memory [17]. This nested private memory is entered by creating a `Runnable` and change the allocation context with `enterPrivateMemory()`.

An example scenario that benefits from temporary allocation within a nested private memory is a loop body that has no dependency on variables declared within the method and produces no results that need to be visible following execution of the loop body. This is shown in Figure 1.

In this example, all the objects created by the `Runnable w` executed in the nested memory will be collected when the `enterPrivateMemory()` method returns. One can imagine the instance of the `Worker` class to be applying an encryption algorithm to a block of static data that should be transmitted periodically. The encryption algorithm can then generate garbage as a result of temporal computations which will be collected at the end of every release.

Two important issues need to be considered. First, avoidance of illegal references and second, the possibility to introduce memory leaks. Memory leaks can be introduced

```
class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent() {

    Worker w = new Worker();

    for(int i = 0; i < BLOCK_SIZE; i++){
      ManagedMemory.enterPrivateMemory(256, w);
    }
  }
}
```

Figure 1: The loop pattern

if on every iteration the loop allocates objects in an outer memory, for example, in immortal or mission memory.

It is important to emphasize that this additional private memory is only useful when entered several times, usually in a loop. If it would be entered only once per release, one can stay within the primary private memory – the backing store consumption would be the same.

4.3 Execute with Primitive Return Value

More interesting is the case when one wants to pass parameters and/or preserve results within iterations of a scoped loop or handlers/Runnables executing in its own private memory.

Naïve approaches may attempt to use static fields to pass information between scopes. This is limiting, because static fields are only allowed to refer to objects residing in immortal memory; thus, this precludes the passing of scope-allocated parameters. Further, this approach is not thread safe, because multiple threads endeavoring to call the same subroutine would overwrite each other's parameters.

References to an object allocated in mission memory (or any other outer scope) used to pass arguments and return results need to be brought into the Runnable. However, Runnable's `run()` method has no parameters.

As an alternative, we can create a class that implements `Runnable` and use input and output objects to pass and return information. References to those objects are passed in the constructor of that class. Upon return, the caller copies the value from the Runnable's field. Clearly, a drawback when using this approach is that only primitive values can be copied as the return value. Figure 2 shows an example.

4.4 Returning a Newly Allocated Object

It might be the case that we want to run part of the execution in a nested private memory but we need to create objects that will be used later. References to objects created in inner scopes cannot be passed to outer contexts, as they will be reclaimed when leaving the inner scope. The key point here is that objects created while executing in an inner scope need to be created in an outer scope. SCJ's API offers the following two options to create objects outside the current allocation context:

- Using `executeInArea()` to explicitly change the allocation context, or
- Using `newArray()` or `newInstance()` to create the objects without explicitly changing allocation context

```
public class Worker implements Runnable {

    SimpleIn in; SimpleOut out;

  public Worker(SimpleIn in, SimpleOut out) {
      this.in = in;
      this.out = out;
  }

  public void run() {
    // Use this.in, work and generate garbage
    T temp = new T();

    // assign primitive values to out
    out.result = temp.x;
  }
}

class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent() {
    // allocate input and output parameters
    // fill input arguments
    Runnable r = new Worker(in, out);
    ManagedMemory.enterPrivateMemory(256, r);
    // now we can use out.result
  }
}
```

Figure 2: Execute with return.

```
class Worker implements Runnable {

  RetObject rObj;

  public void run() {

    // Do some work...
    MemoryArea mem = MemoryArea.getMemoryArea(this);
    rObj = mem.newInstance(RetObject.class);

  }
};

class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent(){

    Worker w = new Worker();
    ManagedMemory.enterPrivateMemory(256,w);

    // Use returned object and fields
    w.rObj...
  }
}
```

Figure 3: Use of `newInstance` to create an object in an upper scope.

For both options we need a reference to the memory area where the object is to be created and a way to reach the new object. Figure 3 shows an example of how to use nested memory scopes with return objects together with `newInstance()`.

In this example, a reference to the memory is obtained via the `getMemoryArea` method. The return object is allocated in the scope where the caller (`this`) is allocated, but it can be done in any outer scope. The newly created object can be accessed through the reference that is part of the `Worker` instance.

The main restriction of this approach is that it is limited to the use of the parameterless constructor of the returnable object. This is a consequence of the requirements imposed on the `newInstance()` method in SCJ [12]. For more complex classes `executeInArea()` needs to be used.

4.5 Scoped Methods

One can create a nice abstraction to hide the complexities of parameter passing, returning results and switching between memory areas by combining the patterns described above into a scoped method. Ideally, a scoped method would be an expression of the form `ret = f(params)` that can be executed in a specific memory area, have input parameters, and can return values or references to objects.

For an SCJ application developer, executing code in a specific memory area is achieved through the use of `enterPrivateMemory()` to go into a nested private memory or by using `executeInArea()` to move into an upper nested scope. Since both methods take a `Runnable` object as argument, the active part of a scoped method should be coded within the `run()` method of a `Runnable` object.

To provide the functionality of parameter passing and returning results, we use a parameter object whose fields contain the method arguments, a reference to the memory area where the returning object shall be allocated, and a field to store the reference to the returned object (so it can be accessed when the method finishes). Figure 4 shows an example of a scoped method that executes in a nested private memory.

4.6 Runnable Factory

Creating all the code above for a single scoped method might be cumbersome. Typically, one would like to execute more than one method in an application. Hiding this complexity can be done through a `Runnable` factory whose methods have a Runnable return type. The application's method code is implemented in the run method of this returned `Runnable`. The factory object itself can be instantiated in the handler's private memory or any other shared memory. Parameters can be passed when calling the factory method. Figure 5 illustrates the concept.

The example shows an *auxiliary object*, `auxObjIn`, used to pass arguments and return values by way of the `readTemperature()` factory method. This object has a field used to store a reference to an *arbitrary object*, `resArbObj`, that should be returned. In this case, the memory area where the result object is to be saved, is obtained from the auxiliary object via the `getMemoryArea()` static method.

The drawback with this particular implementation is that the memory area where the returned object is allocated will be restricted to the memory area of the auxiliary object. If the returned object needs to be saved in a different context then an additional field holding a reference to the destination memory area can be included in the auxiliary object.

One can go even further and hide the `enterPrivateMemory()` within a real method and also return the reference to the outer scope allocated object.

```
public class ParamObject {

  ManagedMemory mem; // Memory reference
  param_X...       // Method parameters
  ReturnObject rObj;  // Return reference
}

public class Method implements Runnable {

  ParamObject params;

  Method(ParamObject params){
    this.params = params;
  }

  public void run() {

    // Use parameters, do work, create garbage

    // Change context, create return object
    params.mem.executeInArea(new Runnable() {
    public void run() {
      ReturnObject rObject = new ReturnObject();

      // Update return object fields
      params.retObject = rObject;}});
  }
}

class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent(){

    // Created in the scope where handler executes
    ParamObject pObj = new ParamObject();

    // This object simulates the method with
    // parameters that returns an object
    Method myMethod = new Method(pObj);
    ManagedMemory.enterPrivateMemory(256,myMethod);

    // Now the returned object can be used
  }
}
```

Figure 4: Scoped method with parameters and a return object.

```
public class RunnableFactory implements IRunnable{

  @Override
  public Runnable readTemperature(final int i,
                                  final AuxObj auxObjIn) {
    return new Runnable() {
      @Override
      public void run() {
        // Do work, here we can use input parameters ...
        // log() is common to all runnables ...
        log();

        // Overwrite primitives OK...
        // Change execution context ... another runnable ...
        MemoryArea.getMemoryArea(auxObjIn).
                                executeInArea(new Runnable(){
        @Override
        public void run() {
          ArbObj resArbObj = new ArbObj();
          resArbObj.a = 50;
          auxObjIn.arbObj = resArbObj;}});
        }
      };
  }

  @Override
  public Runnable otherFactoryMethod() {
    ...
  }
}

class MyHandler extends PeriodicEventHandler {

  public void handleAsyncEvent() {

    RunnableFactory factory = new RunnableFactory();
    AuxObj auxObj = new AuxObj();

    ManagedMemory.enterPrivateMemory(256,
        factory.readTemperature(5, auxObj));

    ManagedMemory.enterPrivateMemory(512,
        factory.otherFactoryMethod());
  }
}
```

Figure 5: Runnable factory.

4.7 Producer/Consumer

Control systems are often composed of producer and consumer processes that run in their own thread of control. The exchange of information between a producer and a consumer can involve data structures or objects, rather than primitive types. In [14] and in [4] solutions are proposed for RTSJ. Such solutions involve the use of portal objects (not part of SCJ), shared scopes, or the introduction of *safe* violations to the reference assignment rules.

Communication between handlers goes through objects located in the shared memory areas, mission and/or immortal memory. Objects in those areas are not reclaimed until termination of the mission or the JVM respectively. Thus we need to reuse objects in those shared memory areas. In [7], a solution is devised using a memory pool of immortal objects. SCJ allows the use of non-immortal objects to do manual pooling. The pool of objects can be collected when the mission finishes.

5. DISCUSSION

During our exploration of SCJ style scopes, we found some issues that can be avoided by changes in the SCJ API, in standard Java libraries, and/or allowing true stack allocation of objects.

5.1 Simplifying Allocation Context Change

The SCJ specification tries to avoid having references to arbitrary memory areas to forbid a handler to enter or execute code outside its private memory. However, the initial private memory area object is allocated in mission memory. A reference can be leaked by the event handler *owning* this memory area via a shared object in mission memory. A different handler can try to use `executeInArea()`, `newInstance()` and `newArray()` with this memory reference. This is not an issue but, as defined by RTSJ, requires checking at runtime that the targeted memory area is in the current thread's scope stack.

To avoid the need for these run-time checks, and thus to avoid the possibility that the exception will be thrown, we propose a change in the current API with the following modifications:

- A new static method, `executeInOuter()`, as a substitute for `executeInArea()`
- Hide `getCurrentManagedMemory()`, `getMemoryArea()`, `executeInArea()`, `newInstance()` and `newArray()` from the application developer

According to the RTSJ scope rules it is only possible in SCJ to change the allocation context to an outer memory. Therefore, we propose to make this explicit with a static method `executeInOuter()` on `ManagedMemory`. This concept complements the static method to enter a nested private memory. For the `executeInOuter()` method, two variants are considered:

- `executeInOuter(Object obj, Runnable logic)`, that will cause the calling schedulable object to change the allocation context to the outer memory area where object *obj* is allocated
- `executeInOuter(Runnable logic)` which changes the allocation context one level out

The new API with these modifications will be semantically equivalent to the original API but with the added benefit of being free from memory reference leaks and the mentioned run-time checks.[1]

5.2 Scope Aware Java Libraries

Besides scope aware patterns, one can also think on scope aware Java libraries. Consider for example the case of doing string concatenation in a temporary scope. We allocate a `StringBuilder` object in a temporary scope. Then, `executeInArea()` into the outer scope to execute `StringBuilder.toString()` to preserve the value in the outer scope. The issue is on how `toString()` is implemented. If it reuses the buffer from the `StringBuilder` it will end up in a wrong pointer assignment. If it does create a new array and copies the `StringBuilder` buffer then it is fine. The JDK version in JOP does a copy as well as OpenJDK's implementation.

Another example can be found in the data structures of `java.util` library (e.g. `LinkedList`, `Vector`, `Stack`, `HashMap`). The issue is the mechanism used to free the memory no longer needed by elements removed from the structures. When an element is removed, the reference to the removed element is set to `null` to help the GC collect the no longer needed object. Use of such structures in mission or immortal memory will cause a memory leak. A solution is proposed in [10], where elements of the data structure are recycled from a pool of elements.

A more detailed study into programming patterns present in the current Java class libraries is an interesting topic for future work. The question to answer would be if it is possible to obtain re-usable code by rewriting portions of those libraries to be scope-safe.

[1]Just before the deadline for the final version of this paper the SCJ expert group accepted our proposal. The static methods to change the allocation context to an outer memory area will be on `ManagedMemory` and are: `executeInOuterArea(Runnable r)` and `executeInAreaOf(Object o, Runnable r)`.

5.3 Stack Allocation

Note that in order to pass reference arguments into a private memory scope, it is necessary to allocate a Runnable object that is nested at the same level as or in a more inner-nested scope level than the scopes that contain the referenced arguments. This is an impediment to encapsulation. It adds difficulty to the tasks of integrating independently developed safety-critical Java software components and of evolving system functionality during common software maintenance activities.

Good software design requires software engineers to encapsulate data and control within clear boundaries of abstraction. These principles of abstraction demand that a programmer who intends to allocate temporary objects must take responsibility according to the conventions of the underlying programming language technology for managing the temporary memory that is consumed by those temporary objects.

A problem with the draft JSR-302 specification is that the choice to introduce a private memory scope from within a particular method is not a purely private concern. The caller needs to know that a callee is going to perform private allocations because the caller needs to set aside memory within its scope to hold the representation of the Runnable object that is used to pass arguments into the private memory scope. Further, the caller needs to know the size of the `Runnable` object. If subsequent software maintenance activities require changes to the structure of the `Runnable` object, it becomes the software maintainers job to identify all of the contexts from which the method is called and adjust their computations of scope sizes appropriately. And if a callee that performs local memory allocations is invoked from within a loop, each iteration of the loop may allocate yet another `Runnable` object within the caller's default memory allocation area. This represents a form of memory leak. Note that the problem of managing memory for temporary objects is different than the problem of managing memory for objects that are intended to persist beyond the lifetime of a particular method invocation. In that case, the caller of the method must take responsibility for managing the memory for the persistent objects. That is a different problem, not addressed by the solution described in this section.

It is important to note that the motivation for these proposed refinements is improved abstraction. The performance benefits are minimal, as the design of the restrictive scoped memory subset supported by JSR-302 is already sufficient to allow private memory scopes to be allocated on the run-time stack. Better separation of concerns, supporting improved abstraction of software components, could be realized if the JSR-302 specification were amended to allow entry into a private scope within which argument and local variables can be accessed without necessitating the creation of a Runnable object within the outer-nested scope. This mechanism resembles stack allocation as it is possible in C, or which optimizing JVMs might perform when references to the objects does not escape from the method. One possible solution is illustrated by the code fragment in Figure 6

The `openPrivateMemoryArea()` method is only allowed to be invoked as the first operation within a try statement that has an accompanying finally clause which contains only an invocation of `closePrivateMemory()` with no code following the finally clause within the method. An additional restriction is that this pattern is only allowed within methods that are

```
public void Method(arg1, arg2, arg3, ...) {
 try {
   ManagedMemory.openPrivateMemoryArea(size);

   // body of method can make arbitrary
   // reference to arg1, arg2, ...
   // and can allocate in inner-nested
   // private memory area
 }
 finally {
   ManagedMemory.closePrivateMemory();
 }
 // No code is allowed to follow the finally
 // statement within this method,
 // and the finally statement is not allowed
 // to have anything other than
 // the closePrivateMemory() invocation.
}
```

Figure 6: Coding style for possible stack allocation in SCJ.

declared to have void or primitive return types. Otherwise, a programmer might accidentally endeavor to return a locally allocated object to the caller's context.

The authors acknowledge that it would be difficult for the JSR-302 group to adopt this solution because there is no equivalent capability in the current RTSJ standard and there is a strong desire to maintain full compatibility between the JSR-302 and RTSJ standards. Perhaps the JSR-282 group will see the value of this style of interaction and add similar capabilities to the JSR-282 standard in order to allow compatible support for the necessary JSR-302 enhancements.

5.4 Creating a Temporary Memory from executeInArea

A common requirement in scoped-memory software systems is to allocate temporary objects as part of a computation carried out in a more outer-nested scope. Consider the following use cases:

1. A `String` object that is to be returned from a method must be constructed in an outer-nested scope so that it can be referenced from objects residing in outer-nested scopes. Construction of the `String` object requires instantiation and manipulation of a temporary `StringBuilder` object to perform catenation of values supplied as a method's input parameters.

2. A constructor for a complex object needs to allocate temporary objects in order to compute the values to be stored into certain instance fields of the constructed object. The constructor itself generally executes in the scope of the object to be constructed, which may not be the inner-most private memory area.

3. A method to modify a data structure that resides in an outer-nested memory area (e.g. `HashMap.put(key, value)`) needs to execute in the outer-nested memory area that holds the data structure in order to allocate objects that can be referenced from the data structure. In some cases, such as when adding a new entry to an existing hash table requires that the size of the hash table be expanded, this method must allocate temporary objects.

As currently drafted, the JSR-302 specification only allows invocation of `enterPrivateMemory()` if the current memory area is the inner-most memory area associated with the current thread. This makes it difficult to address these common use-case scenarios.

We propose the following refinement to JSR-302. Consider the case that private memory P2 nests within private memory P1. Suppose further that from within allocation context P2, the program invokes `executeInArea()`, making P1 the current allocation context. If the program now invokes `enterPrivateMemory()`, a new private memory area P3 is created which physically nests within P2 (is stacked on top of P2) even though it is conceptually private to the current execution context (P1).

However, despite being a safe use case in SCJ, this pattern is disallowed from RTSJ due to the restrictions imposed by the schedulable object's scope stack. Once the allocation context is changed to a scoped memory area, the scope stack will contain only that memory area and the ones below it [3, 14] making the current allocation context the innermost scoped memory (top of the scope stack). Thus, a new `enterPrivateMemory()` will create a sibling of P2, since the previous `enterPrivateMemory()` has not returned yet and a new P3 area will be created and pushed to the scope stack.

There are good use cases that motivate the creation of temporary memories after the change of allocation context with `executeInArea()`; implementing the new private memory area as nested within the last entered private memory (the one from where `executeInArea()` was called from) of this thread seems reasonable.

5.5 IllegalStateException

The description of `enterPrivateMemory()` in the specification is incomplete. Currently, it is documented to throw `IllegalStateException` if invoked during initialization phase. It should also state that the exception will be thrown if invoked when the current allocation context is the result of an `executeInArea()` invocation. This further restriction is needed to keep each private memory area with at most one child.

6. CONCLUSIONS

Safety-critical Java defines three memory areas: immortal memory, mission memory, and private memories. Private memories are used for temporal storage for the lifetime of a single release. Private memories can be nested. Usage of these memories and returning results from an inner nested memory is not trivial. In this paper we analyzed possible usage patterns of the scoped memory as defined in safety-critical Java aiming at helping in the development of applications.

Temporary storage offered by nested private memories can be exploited with the use of scoped loops which are entered more than once per handler release. If results are to be preserved for the next loop iteration, one can think on updating global objects in immortal memory. This is limiting and not thread safe because static fields are only allowed to refer to other immortal objects and multiple threads may overwrite each other's parameters. A different approach is the use of auxiliary objects to return results and pass arguments from and into the `Runnable` required by the `enterPrivateMemory()` and `executeInArea()` methods. Returning objects from private or nested private memory areas is accomplished by an

allocation context change and by saving the reference to the newly allocated object in a field of an auxiliary object. The complexities of creating auxiliary objects, passing references and saving results can be hidden by means of encapsulated methods.

During our exploration of SCJ style scopes, we found some issues that can be avoided by changes in the SCJ API, in standard Java libraries, and/or allowing true stack allocation of objects. We have proposed a simplification of memory allocation context change by using static methods to switch to an outer memory area. This mechanism fits well with the usage of a static method to create and enter a nested private memory.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers for their comments and insights into this work and Anders P. Ravn for discussions on SCJ scope usage during CJ4ES meetings. This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

8. SOURCE ACCESS

The examples presented in this work are part of the JOP distribution and implementation of SCJ [18]. It can be downloaded from `https://github.com/jop-devel/jop`. The source files of the example patterns are located in the `java/target/src/paper/scopeuse/` directory.

9. REFERENCES

[1] E. Benowitz and A. Niessner. A patterns catalog for RTSJ software designs. In R. Meersman and Z. Tari, editors, *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 497–507. Springer Berlin / Heidelberg, 2003.

[2] G. Bollella, T. Canham, V. Carson, V. Champlin, D. Dvorak, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Programming with non-heap memory in the real time specification for Java. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 361–369, New York, NY, USA, 2003. ACM.

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The real-time specification for Java 1.0.2.

[4] 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.

[5] J.-M. Dautelle. Validating Java for safety-critical applications. In *AIAA Space 2005 Conference*, 2005.

[6] M. Dawson. Real-time Java, Part 6: Simplifying real-time Java development. http://www.ibm.com/developerworks/java/library/j-rtj6/, July 2007. Last accessed on July 6, 2012.

[7] D. L. Dvorak and W. K. Reinholtz. Hard real-time: C++ versus RTSJ. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 268–274, New York, NY, USA, 2004. ACM.

[8] D. Flanagan. *Java In A Nutshell, 5th Edition.* O'Reilly Media, Inc., 2005.

[9] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, 1994.

[10] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. Toward libraries for real-time Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, pages 458–462, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[11] J. Kwon and A. Wellings. Memory management based on method invocation in RTSJ. *Lecture Notes in Computer Science 3292 Proceedings of the OTM 2004 Workshops Workshop on Java Technologies for RealTime and Embedded Systems JTRES*, 3292:333–345, 2004.

[12] 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, 2011.

[13] A. F. Niessner and E. G. Benowitz. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *OTM Workshops*, pages 508–519, 2003.

[14] F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: design patterns and semantics. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 101–110, may 2004.

[15] A. Plsek. *SOLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems.* These, Université des Sciences et Technologie de Lille - Lille I, Sept. 2009.

[16] A. Potanin, J. Noble, T. Zhao, and J. Vitek. A high integrity profile for memory safe programming in real-time Java. *The 3rd workshop on Java Technologies for Real-time and Embedded Systems, San Diego, CA, USA*, 2005.

[17] M. Schoeberl. Memory management for safety-critical Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, pages 47–53, York, UK, September 2011. ACM.

[18] 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)*, Copenhagen, DK, October 2012. ACM.

[19] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 241 – 251, dec. 2004.