Static Checking of Safety Critical Java Annotations

Daniel Tang, Ales Plsek, Jan Vitek
Purdue University

ABSTRACT

The Safety Critical Java Specification intends to support the development of programs that must be certified. The specification includes a number of annotations used to constrain the behavior of programs written against it. This paper describes and motivates the design of these annotations and the rules used to check statically that programs respect their intended semantics. We report on a prototype implementation with the Java Checker Framework and initial experiments annotating a 24KLoc application.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*run-time environments*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*.

General Terms

Languages, Experimentation.

Keywords

Safety Critical Systems, Verification, Annotations, Memory Safety.

1. INTRODUCTION

Safety-critical applications must undergo a rigorous validation and certification process. To facilitate this, software is often written so as to be analyzable and verifiable. Traditional approaches to certification, such as the DO-178B certification for airborne systems in the US [9], ED-12B in Europe, dictate software engineering processes and manual verification, however, there is growing pressure to use automated techniques. This is motivated by the growing size of todays's safety-critical code bases. Thus the challenge is to propose languages, methodologies and tools that can scale to large systems and provide the necessary guarantees of safety and certifiability.

The Safety Critical Java (SCJ) Specification [5] is an emerging standard for developing safety-critical applications in Java. SCJ is based on a subset of the Real-time Specification for Java (RTSJ) [2] and is designed to enable the creation of applications, infrastructures, and libraries that are amenable to certification under safety-critical standards. For the purpose of this work there are three salient features to the SCJ Specification. First, the programs are composed of one or more missions, where each mission is composed of a bounded number of schedulable objects. Second, each schedulable object is given its dedicated memory area that is allocated during initialization phase of the mission. And third, SCJ distinguishes three compliance levels with different cost and difficulty of certification.

The draft SCJ specification defines a set of Java metadata annotations which can be used to annotate user code and API classes. These annotations impose constraints on what code is legal, or compliant with the SCJ specification. The annotations are intended for controlling three aspects of program behavior: (i) *level compliance* – adherence of applications to one of the three SCJ compliance levels, and restriction of APIs used to the level of the application; (ii) *behavioral compliance* – verify compliance of the application to certain temporal and behavioral restrictions; and (iii) *memory safety* – ensure that no memory store can lead to a dangling pointers. The specification prescribes static checking of SCJ applications, and programs that fail to verify must be rejected.

The contributions of this paper are to describe the SCJ annotations and give a list of rules that must be followed when using these annotations. We have implemented a static checker that checks compliance of applications as part of the compilation process using the Java checker framework. Finally, we have conducted a case study to evaluate usability of the annotations on a small SCJ program. The results of this work are available as part of our Open Safety Critical Java, oSCJ, project[1].

It should be noted that the SCJ specification does not require automated verification of SCJ applications. Thus, for instance, checking that the program will not run out of memory is out of scope. Richer properties will have to be verified as part of the certification process using methods and techniques deemed acceptable by the certification body.

JTRES'10 August 19-21, 2010 Prague, Czech Republic

[1]The oSCJ distribution is available at `www.ovmj.net/oscj`.

2. THE SAFETY CRITICAL JAVA SPECIFICATION

The complexity of safety critical software varies greatly; the cost of certification of both the application and the infrastructure is highly sensitive to their complexity, so enabling the construction of applications that employ a more restricted programming model is desirable. The SCJ specification defines three compliance levels to which both implementations and applications may conform. Level 0 provides a simple, frame-based cyclic executive model which is single threaded with a single mission at a time. Level 1 extends this model with support for multi-threading via aperiodic event handlers, multiple concurrent missions, and a fixed-priority preemptive scheduler. Level 2 lifts all restrictions on threads and supports nested missions with managed threads.

SCJ programs are organized into a set of *missions*, which are relatively independent with respect to lifetime and resources needed. It is up to the programmer to decompose the application into missions and define their startup sequence. Missions are launched according to a pre-defined order. A mission has three phases in its lifecycle, which are *initialization*, *execution*, and *cleanup*. After a mission terminates, the next mission is released, if there is one. A mission, is made up of a set of *shedulable objects*, i.e. threads and their scheduling constraints.

2.1 Scoped Memory

The scoped memory model was introduced by the RTSJ and is retained in SCJ. The heap, on the other hand, has been abandoned to make memory usage predictable. The SCJ specification supports three types of memory areas: *immortal memory*, *mission memory*, and *private memory*. As the name suggests, immortal memory is the memory area that spans the lifetime of the application; objects allocated there live for the entire program execution. The other two are scoped memory areas, they will be created and reclaimed at runtime. The scoped memory model requires implementation of runtime scope memory checks verifying each memory write. Each mission has a mission memory, shared by the mission's schedulable objects to store mission-scoped data. Each schedulable object has it's own private memory. It can create nested private memory areas by invoking the `enterPrivateMemory()` method. The private memory is retained for the duration of the schedulable object's `run()` method. The bound on the total size of all private memory instances of any given schedulable are determined at mission startup. The difference between `enterPrivateMemory()` and the RTSJ `enter()` method is that former creates and enters a nested scope only if the current scope does not already have a subscope. This means that scope hierarchy is a stack rather than a tree as in the RTSJ. The `executeInArea()` method is available to switch to a parent scope but this can not be used to create nested scopes higher in the hierarchy. Figure 1 illustrates this. A `PeriodicEventHandler` is performing computations in a loop and at the end of each iteration, a logging message is stored into the logger. In order to prevent memory leaks, the operation of logging is performed in a dedicated memory area. To create this new area, the user first needs to obtain a reference to the current memory area and then call `enterPrivateMemory()`. As input parameters, the user specifies a size of the new area, which must be smaller than the memory reserved for the current schedulable, and a class implementing the functionality — `Logger`, that will be executed inside this memory.

```
class MyHandler extends PeriodicEventHandler {
  public void handleEvent() {
    ...
    ManagedMemory mem = ManagedMemory.
                          getCurrentManagedMemory();
    Logger log = new Logger();
    while(...) {
       ...
       log.setMessage(msg);
       mem.enterPrivateMemory(3000, log);
    }
  }
}

class Logger implements Runnable {...}
```

Figure 1: Enter Private Memory Example

3. SCJ ANNOTATIONS

The SCJ expert group proposed a set of Java metadata annotations which enables tool developers to add type-like information to a SCJ program. The three main groups of annotations are described in the following subsections.

In the remainder of the paper, we differentiate between *user* code and *infrastructure* code. User code is checked by the tool to abide by the restrictions outlined in this chapter. Infrastructure code is verified by the vendor. Infrastructure code includes the `java` and `javax` packages as well as vendor specific libraries.

3.1 Compliance Levels

The SCJ specification defines three levels of compliance. Both application and infrastructure code must adhere to one of these compliance levels. These API visibility annotations are used to prevent client programmers from accessing SCJ API methods that are intended to be internal. Since the SCJ specification spans two packages, package-private visibility is not an option. In addition, elements intended to only be used at certain compliance levels, should be considered invisible to applications and infrastructure code declared to be at a lower compliance level. For this purpose, the specification introduces the `@SCJAllowed` annotation, which can be placed on any class or member, see Figure 2. This annotation has two parameters: `value` and `members`.

annotation	parameters	description
@SCJAllowed	**value** = LEVEL_0 LEVEL_1 LEVEL_2	*value* specifies compliance level of an element.
	SUPPORT	Infrastructure private, can be overridden by user-level code.
	INFRASTRUCTURE	Infrastructure private.
	HIDDEN	An element non-accessible both from user and infrastructure.
	members = true / false *(default)*	if *TRUE,* value is recursively inherited by sub-elements.

Figure 2: Compliance Levels Annotation.

The value parameter may contain the following values ordered from the lowest to the highest level : LEVEL_0 < LEVEL_1 < LEVEL_2 < SUPPORT < INFRASTRUCTURE < HIDDEN. SCJ specifies that an element with a certain level may only be visible by those elements that are at the specified level or higher. Therefore, a method that is annotated with @SCJAllowed(LEVEL_1) may be invoked by a method having an annotation @SCJAllowed(LEVEL_2) but not vice versa.

The elements with level SUPPORT and INFRASTRUCTURE can only occur in infrastructure code that can not be accessed from user code. HIDDEN denotes classes and methods that can not be accessed from user or infrastructure code. @SCJAllowed(HIDDEN) is assigned by default to all unannotated elements. The `members` parameter controls whether or not the annotation applies to its enclosed members. This reduces the annotation burden on the programmer.

3.1.1 Rules

If a class, interface, or member has compliance Level C, it may only be used in code that also has compliance Level C or higher. It is legal for an implementation to not emit code for methods and classes that may not be used at the chosen level of SCJ application, though it may be necessary to provide stubs in certain cases. If a class has a default constructor, the constructor's compliance level is that of the class if the annotation has `members=true`, and `HIDDEN` otherwise. Static initializers have the same compliance level as their defining class, regardless of the members argument. It is illegal for an overriding method to change the compliance level of the overridden method. It is also illegal for a subclass to have a lower compliance level than its superclass. Intuitively, all of enclosed elements of a class or member should have a compliance level greater than or equal to the enclosing element.

3.1.2 Example

Figure 3 illustrates use of the compliance level annotation. The example shows both user and infrastructure fragments of source code. As we can see, all the elements are declared to reside at some level. Class `MyMission` is at Level 0. Every Level 0 mission is composed of one or more periodic handlers; in this case, we define the `MyHandler` class. The handler is, however, declared to be at Level 1, which is an error. Furthermore, `MyMission`'s initialization method attempts to instantiate `MyHandler` and tries to call `PeriodicEventHandler`'s `run()` method. However, the method is annotated as @SCJAllowed(INFRASTRUCTURE)and can be called only from infrastructure code.

Looking at the SCJ infrastructure code, the `PeriodicEventHandler` class implements the `Schedulable` interface, both of which are Level 0 compliant. However, `PeriodicEventHandler` is defined to override `getReleaseParameters()`, originally allowed only at Level 2. This is an illegal attempt to decrease method visibility.

3.2 Behavior Restrictions

This set of annotations deals with behaviors and characteristics of methods. For example, some methods may only be called in a certain mission phase. Others may be restricted from allocation or blocking calls. In both cases, the restricted behavior annotation @SCJRestricted is used. The

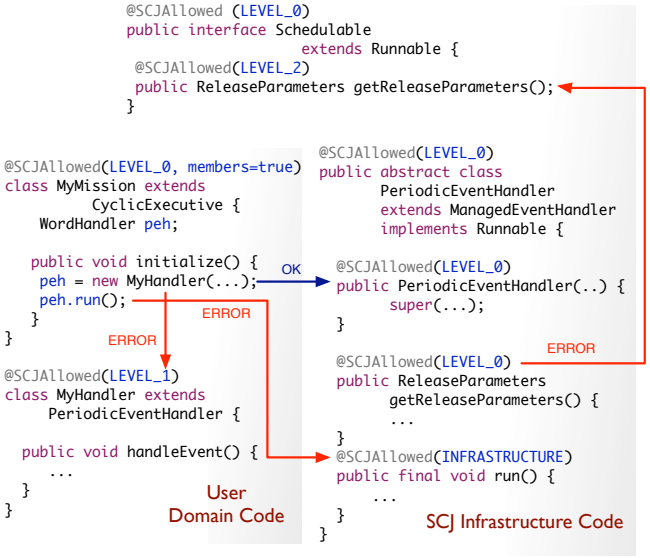

Figure 3: Compliance Levels Annotation Example.

annotation	parameters	description
@SCJRestrict	**mayAllocate** = true *(default)* / false	Specifies that method allocates memory.
	maySelfSuspend = true / false *(default)*	Specifies that the method may perform blocking operation.
	value = INITIALIZATION RUN CLEANUP ALL	Specifies mission context in which a certain method can be executed.

Figure 4: Behavior Restriction Annotations.

annotations and its parameters are defined in Figure 4. The default argument `value` is one of the list of restrictions of type Phase:

- INITIALIZATION denotes a method which can only be called during the initialization phase of a mission,
- CLEANUP denotes a method which may only be called during the clean-up phase,
- RUN denotes a method which may be called during the mission execution phase,
- ALL denotes a method which may be called at any time.

The restrictions `INITIALIZATION`, `RUN`, and `CLEANUP` are mutually exclusive. Furthermore, two other parameters are specified: `mayAllocate` and `maySelfSuspend`:

- mayAllocate=false is annotated on methods that perform no allocation in themselves,
- maySelfSuspend=false denotes a method which is guaranteed to not block.

There is a partial ordering on annotations: `maySelfSuspend=true` < `maySelfSuspend=false`, `mayAllocate=false` < `mayAllocate=true`, `INITIALIZATION` < `ALL`, `RUN` < `ALL`,

CLEANUP < ALL. When overriding a method, a subclass is required to retain the annotation of the overridden method or strengthen it. Annotations that are related by the ordering relation are mutually exclusive.

When no annotation is present or some of the parameters are not specified, default values are mayAllocate=true, maySelfSuspend=false and value=ALL. Native methods have no defaults and thus must be fully annotated.

3.2.1 Rules

Only methods annotated with mayAllocate=true may contain expressions that result in allocation. This includes new expressions, as well as string concatenation and autoboxing of primitive types to objects. Methods annotated mayAllocate=false may only invoke methods annotated mayAllocate=false. A similar rule is necessary for maySelfSuspend=false. In addition, maySelfSuspend=false methods may not have blocking statements, such as synchronized blocks. Methods annotated INITIALIZATION, CLEANUP, or RUN may only invoke other methods with the same annotation or ALL. Methods annotated ALL may only invoke other methods annotated ALL.

3.3 Memory Safety

Memory safety is similar in RTSJ and SCJ: Objects in a certain scope may not reference objects in a child scope. Objects in a child scope are deallocated before objects in a parent scope. Therefore, the specification introduces the following annotations, see Figure 5:

annotation	parameters	description
@DefineScope	*name* - name of the scope *parent* - parenting scope	Defines a scope and its parent.
@Scope	*name* - name of the scope where object lives	For an element specifies in which scope it is allocated.
@RunsIn	*name* - the scope where an element is executed.	Specifies allocation context of a method or class.

Figure 5: Memory Safety Annotations.

@DefineScope(name, parent) is used to statically construct the scope tree. It must be attached to ManagedMemory references to keep track of what scope each reference represents. It is also used to annotate the Runnable passed to enterPrivateMemory() to name the new scope being created. Mission classes should also carry the @DefineScope annotation to name the mission memory and define what the parent scope of the defined mission is. It is not necessary to annotate Mission objects; by default, a scope named after the mission is created, with immortal memory as the parent. Although event handlers create implicit scoped memories, the necessary addition to the scope tree can be inferred from the @Scope and @RunsIn annotation on the event handler class.

During the first pass of checking for memory safety, the scope tree is constructed and checked for well-formedness. That is, there can be no duplicates in the scope tree, or cycles. In addition, every chain of scopes must end at the immortal scope.

@Scope is used to annotate a class to indicate in which scope all objects of the class are allocated. Under our system, all objects of a type are allocated in the same memory area. All methods in the class run in the specified scope by default.

@RunsIn is used to annotate either classes or methods. When annotating a class, it signifies the default allocation context for its methods. When attached to a method, it specifies the context for that particular method, overriding any annotations on its enclosing type.

The @Scope and @RunsIn annotations together define the allocation context for each method in a SCJ program. Annotations on methods take precedence over annotations on classes, with @RunsIn taking precedence over @Scope.

3.3.1 Rules

Verifying memory safety involves two passes over the code; the first pass constructs the scope tree and checks it for errors and the second pass uses the tree constructed in the first pass to do actual checking.

Being the most complex property that our checker verifies, there are many rules that code must follow:

- Objects may only be allocated in the context specified by the annotation on their types.
- Arrays may only be allocated in the same context as that of their element type.
- Variables of a specific type may not be declared in any scope that is a parent of the scope specified by the type. Intuitively, if all objects are allocated in a particular scope, a reference from a parent scope will result in a dangling pointer.
- Static variables must have types with no @Scope annotation or @Scope("immortal").
- Methods retain annotations from methods that they override.
- Invocation of a method is only allowed if its allocation context is the same as the current context or is a parent to it. This is simply to prevent objects from being allocated in the wrong scope.
- Calls to a scope's executeInArea() method can only be made if the scoped memory is a parent of the current context in the scope tree. In addition, the Runnable object passed to the method must have a RunsIn annotation that matches the name of the scoped memory.
- Calls to a scope memory's enterPrivateMemory(size, runnable) method are only valid if the runnable variable definition is annotated with @DefineScope(name="x",parent="y") where x is the memory area being entered and y is a the current allocation context. The @RunsIn annotation of the runnable must be the name of the scope being defined by @DefineScope.
- Calls to a scope's newInstance() or newArray() methods are only valid if the class or element type of the array are annotated to be allocated in target scope.
- Class casting must follow special rules in order to not to lose the scope knowledge associated with each variable. When casting a variable of a certain type, its defined scope must be either the as the scope of the target type or the scope of the target type must be undefined and the current allocation context must be the same as the scope of the variable being casted.

The above rules assume that all classes that undergo verification are annotated. However, in some cases, this may not be the case. In order to handle unannotated classes, we augment the above set of rules:

- Objects of unannotated types may be allocated anywhere. Since we can statically determine the allocation context in any annotated class, the allocation context of an unannotated object can subsequently be determined.
- Objects of unannotated types may not leave the allocation context in which they were instantiated (i.e., may not be passed to a method which has a different allocation context). Allowing an unannotated object to pass to a different scope from the one in which it was created would lose scope information necessary for determining assignability.
- When a method returns an object of an unannotated type, it must be allocated in the method's allocation context.
- Classes that are unannotated may not reference any annotated types. This prevents objects of unannotated types, which can be allocated anywhere, from allocating objects in a scope other than their designated one.

3.3.2 *Example*

Figure 6 gives an example illustrating the use of memory safety annotations. A `MyMission` class declares a scope in which the mission is running by @DefineScope(name="MyMission",parent="immortal"). Furthermore, the mission handler `MyHandler` is defined to be allocated in mission's memory by @Scope("MyMission"), while running in its own handler private memory by @RunsIn("MyHandler"), thus implicitly defining a new memory area. The user is also expected to define a new scope area any time code enters a child scope. This is illustrated by the `MyRunnable` class that is allocated in `MyHandler` private memory while running in its own scope. When instantiating this runnable, we annotate it with @DefineScope to define a new scope area that may be entered using this runnable.

4. IMPLEMENTATION

Static verification is done using the Checker Framework, which is built on top of JSR308 [4]. JSR308 is part of Java 7 and extends the Java 5 annotation system. Annotations can be used in many more places than previously, which can allow programmers to create richer type systems. The Checker Framework exposes a low level AST visitor for low level processing for annotations. All of our verification is done with these AST visitors.

The SCJ specification expects that the metadata annotations will be checked at compile time as well as at load time (or link time if class loading is integrated with the linking). Therefore, the SCJ annotations are retained in the compiled bytecode intermediate forma. Compile-time checking is useful to provide rapid feedback to developers, while load- or link-time checking is essential for ensuring safety. Virtual machines that use an ahead-of-time compilation model are expected to perform the checks when the executable image of the program is assembled. The virtual machine may omit memory access checks for classes that have been successfully checked.

```
@Scope("immortal")
@DefineScope(name="MyMission",
                  parent="immortal")
class MyMission extends CyclicExecutive {
   public void initialize() {
      new MyHandler(...);
   }
}

@Scope("MyMission")
@RunsIn("MyHandler")
class MyHandler extends PeriodicEventHandler {

  public void handleEvent() {
      @DefineScope(name="MyRunnable",parent="MyHandler")
      MyRunnable r = new MyRunnable();
      ManagedMemory.getCurrentManagedMemory().
         enterPrivateMemory(3000, r);
  }
}

@Scope("MyHandler")
@RunsIn("MyRunnable")
class MyRunnable implements Runnable {
    public void run() {
    }
}
```

Figure 6: Example of Memory-Safety Annotations.

5. EXAMPLE AND EVALUATION

Figure 7 presents an example that uses all three types of SCJ annotations. The example shows a Level 0 `MyMission` class and its periodic handler – `MyHandler`. The `MyMission` object is allocated in a scope named similarly and implicitly runs in the same scope. A substantial portion of the class' implementation is dedicated to the `initialize()` method, which creates mission's handler and then uses `enterPrivateMemory()` to perform some initialization tasks in subscopes using `ARunnable` and `BRunnable`.

The figure also highlights several errors detected by the checker. First, the user attempts to allocate instances of classes `A` and `B` in an allocation context that is not consistent with those defined for these classes. In order to instantiate properly the class `B` in `MyMission` class, the user has not other choice then to duplicate the class implementation and provide each class declaration with a different scope annotation, as it is shown in the example where `B` is defined in both packages with corresponding scope annotations.

Furthermore, the usage of both handlers is also illegal. The `ARunnable` class is annotated to be Level 1 and therefore is not accessible in the context of the Level 0 `MyMission`. The `BRunnable` class is defined to run in the BRunnable scope; however, the initialization method defines the instance of `BRunnable` class to run in PrivateMemory.

The `MyHandler` class implements functionality – the `handleEvent()` method – that will be periodically executed throughout the mission. The allocation context of this execution will be MyHandler scope, as the RunsIn annotation upon the `MyHandler` class suggests. Looking at the `handleEvent()` method, we can see that some of the functionality is designated to be executed in child scope memory areas through the `ARunnable` and `BRunnable` classes. However, the checker will detect an error since the `ARunnable` is declared to run in the MyMissionInit memory area, which is not a

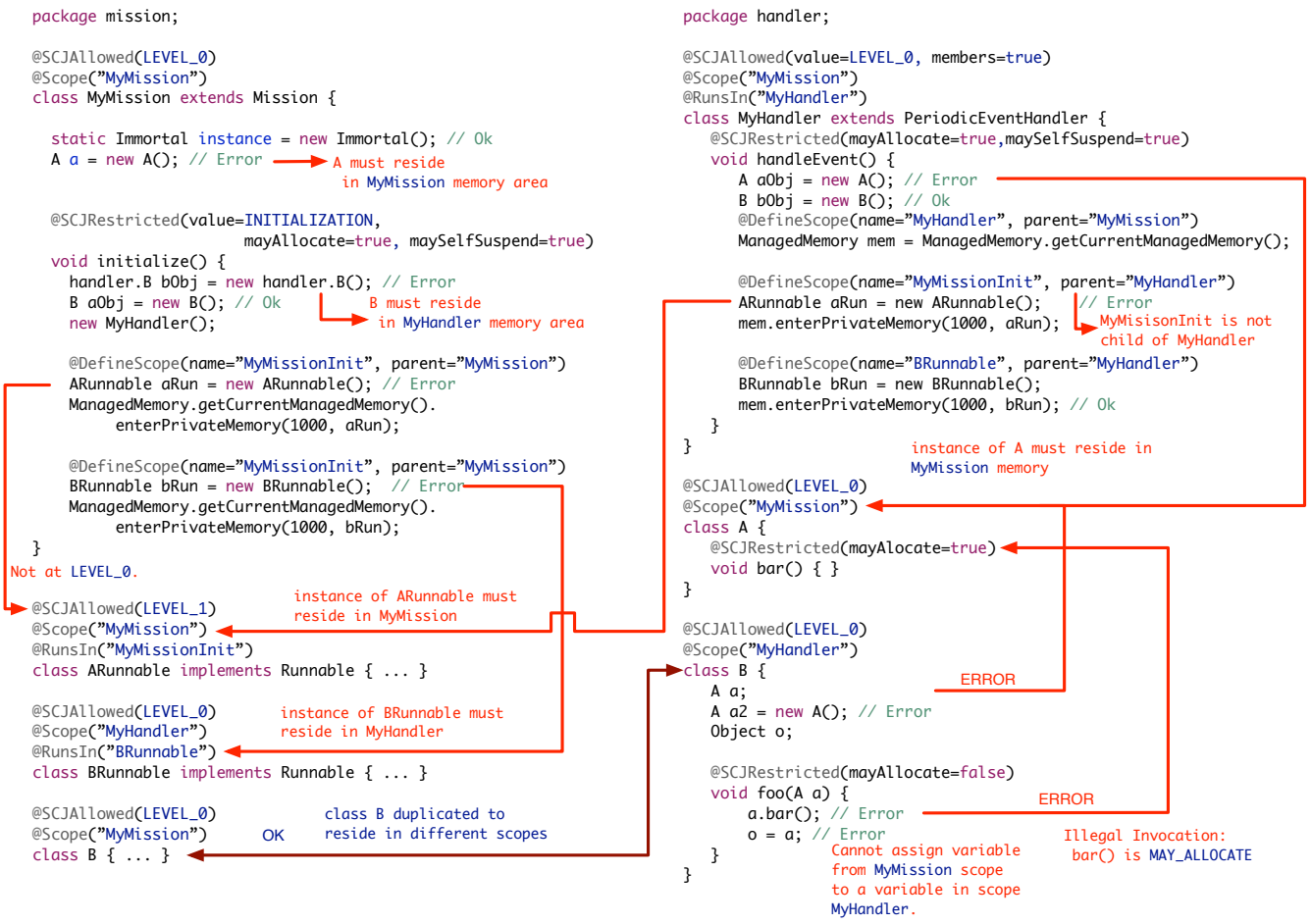

Figure 7: SCJ Annotations; A Full-Scale Example.

child to the MyHandler memory area. Furthermore, the MyHandler and `B` classes try to instantiate the `A` class in illegal contexts. Method `bar()` in `B` further tries to call `a.foo()`, which is annotated as mayAllocate=true whereas the caller method prohibits any allocation. Finally, the assignment `o = a;` is illegal, since the user is assigning a variable from the MyMission scope to a variable in the MyHandler scope. Assigning an instance of an unannotated type into a variable in a different scope would cause lost of scope knowledge related to this instance, which is illegal.

This example, together with roughly more than 100 other examples, which each demonstrate a detection of a specific annotation error, are part of our checker and can be found in oSCJ distribution[2].

5.1 Evaluation

In order to conduct a field test for feasibility, we have implemented and annotated miniCDj as a case study. miniCDj[3] is a SCJ application based on the CDx benchmark [6,8] and simulates a collision-detector algorithm that iteratively computes collision courses of aircraft. From the annotation point of view, miniCDj represents a Level 0 application, where a periodic handler iteratively processes the aircraft movements and computes collision courses in a child scope while keeping and updating position of each aircraft in the mission's memory. The 24kLoC of miniCDj spans over 61 classes, to properly annotate the source code we used 92 annotations (61 level annotations and 31 memory safety annotations). One class (`Vector3d`) needed to be duplicated as it is used in two different allocation contexts.

We have extracted several points highlighting the characteristics of a development process that involve SCJ annotations:

- *Class Duplication* – The most displeasing aspect of assigning each class to a specific scope, already mentioned in [1], is the necessity to duplicate a class that depending on the execution context needs to reside in two different scopes. This can be alleviated by defining the class without any scope annotations.
- *Restrictive* – The annotations are sometimes too restrictives, e.g. for `o = a` observed in Figure 7 which is evaluated as illegal because of the scope knowledge lost.
- *Negligible Engineering Burden* – The effort required to convert a vanilla RTSJ to the code compliant with the SCJ annotations is estimated as relatively small since the developer must be aware of these metadata

[2]Available in `oSCJ/tools/Checker/tests`.
[3]The miniCDj distribution available at `www.ovmj.net/cdx/`.

information (e.g. SCJ functionality needed for given application, allocation context and object life span) prior to the implementation. Furthermore, the effort is facilitated by the checker that can be easily used in the development process and that iteratively reports all violations of the SCJ annotation rules.

6. RELATED WORK

The Aonix PERC Pico virtual machine introduces stack-allocated scopes, an annotation system, and an integrated static analysis system to verify scope safety and analyze memory requirements. The PERC type system [7] introduces annotations indicating the scope area in which a given object is allocated. A byte-code verifier interpreting the annotations proves the absence of scoped memory protocol errors. The PERC Pico annotations do not introduce absolute scopes identifiers. Instead, they emphasize scope relationships (e.g. argument A resides in a scope that "encloses" the scope of argument B). This allows more generic reuse of classes and methods in many different scopes, rather than requiring duplication of classes for each distinct scope context at the cost of a higher annotation burden. The PERC annotations address sizing requirements which are not considered here.

The authors of [3] proposed a type system for Real-Time Java. Although the work is applied to a more general scenario of RTSJ-based applications, it shows that a type system makes it possible to eliminate runtime checks. In comparison to the approach in this paper, the proposed type system provides a richer but a more complex solution.

Scoped Types [1, 10] introduce a type system for RTSJ which ensures that no run-time errors due to memory access checks will occur. Furthermore, Scoped Types capture the runtime hierarchy of scopes and subscopes in the program text by the static hierarchy of Java packages and by two dedicated Java annotations. The authors demonstrates that it is possible to statically maintain the invariants that the RTSJ checks dynamically, yet syntactic overhead upon programmers is small. The solution presented by the authors is a direct ancestor of the approach proposed by this paper.

7. CONCLUSION

The Safety Critical Java Specification (SCJ) developed by JSR-302 expert group is near completion. The goal of this effort is to bring Java to the domain of safety-critical systems. As one of its goals, the specification proposes a metadata annotation system that enforces correct usage of SCJ concepts in applications. This paper provides an overview of these annotations, their semantics, and rules that must be followed by users. We have also developed a checker that statically checks conformance of applications to the specification. Our initial evaluation case study reports that although the annotation model may sometimes be restrictive or require a class duplication, the annotations impose small syntactic overhead upon programmers while allowing to statically check required SCJ properties.

Acknowledgments. This work was partially supported by NSF grants CNS-0938256, CCF-0938255, CCF-0916310 and CCF-0916350. The authors gratefully acknowledge the contributions of the JSR 302 Expert Group to the ideas presented in this paper and thank reviewers for their comments.

8. REFERENCES

[1] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java memory management. *Realtime Systems Journal*, 37(1), 2007.

[2] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[3] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Programming Language Design and Implementation (PLDI)*, 2003.

[4] JSR 308 Expert Group. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

[5] JSR 302. Safety critical Java technology, 2007.

[6] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, and Jan Vitek Ben Titzer and. CDx: A family of real-time Java benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, September 2009.

[7] Kelvin Nilsen. A type system to assure scope safety within safety-critical Java modules. In *International workshop on Java technologies for real-time and embedded systems (JTRES)*, 2006.

[8] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *EuroSys Conference*, 2010.

[9] RTCA and EUROCAE. Software considerations in airborne systems and equipment certification. Radio Technical Commision for Aeronatics(RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B, 1992.

[10] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, 2004.