Chapter 1

Java Metadata Annotations

Last edited by Jan Vitek, Ales Plsek, Date: 2011/07/15 19:21:08 .

This chapter describes Java Metadata annotations used by the SCJ. Java Metadata annotations enable developers to add additional typing information to a Java program, thereby enabling more detailed functional and non functional analyses, both for ensuring program consistency and for aiding the runtime system to produce more efficient code. These metadata annotations provide a basis for additional checks for ensuring the correctness and efficiency of safety critical Java programs. They are retained in the compiled bytecode intermediate format and are thus available for performing validation at class load-time. One interest in using metadata annotations is to ensure memory safety, thus preventing several exceptions from being thrown at runtime. They are also used for enforcement of compliance levels and restricting the behavior of certain methods.

The specification differentiates 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.

1.1 Semantics and Requirements

The SCJ annotations address the following three groups of properties.

- *Compliance Levels*—The SCJ specification defines three levels of compliance. Both application and infrastructure code must adhere to one of these compliance levels. Consequently, a code belonging to a certain level may access only the code that is at the same or higher level. This ensures that an SCJ application is consistent with respect to the specified SCJ level.

- *Behavioral Restrictions*—Since the execution of the missions are implemented as a sequence of specific phases (initialization, execution, cleanup), the application must clearly distinguish between these phases. Furthermore, it is illegal to access SCJ functionality that is not provided for current execution phase of a mission.
- *Memory Safety*—To ease certification and improve the safety of developed software, SCJ provides annotations that may be used to analyze the memory management of a program prior to system execution.

1.2 Annotations for Enforcing Compliance Levels

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 more package names (e.g. `javax.realtime` and `javax.safetycritical`), package-private visibility is not an option.

The SCJ specification specifies three compliance levels which applications and implementations may conform to. Each level specifies restrictions on what APIs may be used, with lower levels being strictly more restrictive than higher levels. The `@SCJAllowed()` metadata annotation is introduced to indicate the compliance level of classes and members. The `@SCJAllowed()` annotation is summarized in Tab 1.1 and takes two arguments.

Annotation	Argument	Values	Description
@SCJAllowed	value	**LEVEL_0**	User-level.
		LEVEL_1	
		LEVEL_2	
		SUPPORT	User-level, accessed by library.
		INFRASTRUCTURE	Library private.
		HIDDEN	Non-accessible.
	members	TRUE	Inherit value by sub-elements.
		FALSE	

Table 1.1: Compliance LEVEL annotation. Default values in bold.

1. The default argument of type `Level` specifies the level of the annotation target. The options are `LEVEL_0`, `LEVEL_1`, `LEVEL_2`, `SUPPORT`, `INFRASTRUCTURE` and `HIDDEN`.
    - `LEVEL 0, 1 or 2` specify that an element may only be visible by those elements that are at the specified level or higher. Therefore, a method that is `@SCJAllowed(LEVEL_2)` may invoke a method that is `@SCJAllowed(LEVEL_1)`, but not vice versa. In addition, a method annotated

with a certain level may not have a higher level than a method that it overrides.

- SUPPORT specifies a user-level method that can be invoked only by the infrastructure code, the annotation cannot be used to specify a level of a class. SUPPORT method cannot be invoked by other SUPPORT methods. SUPPORT method can invoke other user-level methods up to the level specified by its enclosing class.
- INFRASTRUCTURE specifies that a method is API private. Therefore, methods outside of javax.realtime and javax.safetycritical packages may not invoke methods that have this annotation.
- HIDDEN denotes classes and methods that are hidden and can not be accessed both from the user and infrastructure code. No element with this annotation can be accessed from the SCJ application or infrastructure.

The default value when no value is specified is LEVEL_0. When no annotation applies to a class or member, it takes on value HIDDEN. The ordering on annotations is LEVEL_0 < LEVEL_1 < LEVEL_2 < SUPPORT <INFRASTRUCTURE < HIDDEN.

2. The second argument, members, determines whether or not the specified compliance level recurses to nested members and classes. The default value is false.

Overriding the Default Level for User Classes

By default, any infrastructure and user code has the level set to HIDDEN. The default value for the user-level code can be overriden by a command-line argument -Alevel= passed to the checker. The possible values of the argument are -Alevel=0, -Alevel=1, and -Alevel=2, corresponding to LEVEL_0, LEVEL_1, and LEVEL_2 respectively.

Compliance Level Reasoning

The compliance level of a class or member is the first of the following:

1. The level specified on its own @SCJAllowed() annotation, if it exists,

2. The level of the closest outer element with an @SCJAllowed() annotation, if members = true,

3. HIDDEN.

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 an SCJ application, though it may be necessary to provide stubs in certain cases.

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. Each element must either correctly override the @SCJAllowed annotation of the parent or restate the parent's annotation. Intuitively, all of enclosed elements of a class or member should have a compliance level greater than or equal to the enclosing element.

Methods annotated HIDDEN or INFRASTRUCTURE may not be overridden in user code. Methods annotated SUPPORT must be overridden by the user and the SUPPORT annotation must be restated.

Static initializers have the same compliance level as their defining class, regardless of the members argument.

Class Constructor Rules

For a class that is annotated @SCJAllowed, all constructors have to be annotated @SCJAllowed as well.

If a class has a default constructor, the constructor's compliance level is that of the class if the annotation has members = true, or HIDDEN otherwise.

Other Rules

The exceptions thrown by a method must be visible at the compliance level of that method.

1.3 Annotations for Restricting Behavior

The following set of annotations is provided to express 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 SCJRestricted annotation has three attributes: mayAllocate, maySelfSuspend, and value. The first two are boolean and the last takes an element of the Phase enumeration.

When mayAllocate is true, the annotated method is allowed to perform allocation or call methods that are also annotated @SCJRestricted(mayAllocate = true). If a method is @SCJRestricted(mayAllocate = false), then all method that override it must be @SCJRestricted(mayAllocate = false) as well. Only methods that are annotated @SCJRestricted(mayAllocate = true) may contain expressions that result in allocation (e.g. at the source level new expressions, string concatenation, and autoboxing). The default value is true.

When maySelfSuspend is true, the annotated method may take an action that caused it to block. If a method is marked @SCJRestricted(maySelfSuspend = false, then neither it nor any method it calls may take an action causing it to block. The default value is true.

A method annotated with value set to anything other than ALL in SCJRestricted, then the method may only be called in the given phase.

The @SCJRestricted annotation may be set on a class, interface, or enumeration, in which case it changes the default values for the methods on that class, interface, or enumeration.

1.4 Annotations for Memory Safety

1.4.1 Definitions of Memory Safety Annotations

The three SCJ annotations for memory safety, summarized in Table 1.2, are as follows.

Scope Tree

A SCJ application contains a finite set of scoped areas, each scoped area has a name and a parent. Scope names must be unique. The scopes and their parent relation must define a well formed *scope tree* rooted at IMMORTAL, the distinguished parent of all scopes.

@DefineScope Annotation

@DefineScope annotation is used to define the *scope tree*, it has two arguments, the symbolic name of the new scope and of its parent scope. The annotation can and must be used only on declaration of classes that have an associated scope (for instance, subclasses of the MissionSequencer and Schedulable classes). For classes implementing the Runnable interface:

Annotation	Where	Arguments	Description
@DefineScope	Any	*Name*	Define a new scope.
@Scope	Class	*Name*	Instances are in named scope.
		CALLER	Can be instantiated anywhere.
	Field	*Name*	Object allocated in named scope.
		UNKNOWN	Allocated in unknown scope.
		THIS	Allocated enclosing class' scope.
	Method	*Name*	Returns object in named scoped.
		UNKNOWN	Returns object in unknown scope.
		CALLER	Returns object in caller's scope.
		THIS	Returns object in receiver's scope.
	Variable	*Name*	Object allocated in named scope.
		UNKNOWN	Object in an unknown scope.
		CALLER	Object in caller's scope.
		THIS	Object in receiver's scope.
@RunsIn	Method	*Name*	Method runs in named scope.
		CALLER	Runs in caller's scope.
		THIS	Runs in receiver's scope.

Table 1.2: Annotation summary. Default values in bold.

1. when used for enterPrivateMemory(), the class must be also annotated with @DefineScope.
2. when used for executeInArea(), the class must be annotated with @DefineScope which refers to an already existing scope and mirrors the @DefineScope annotation used to define this scope.

Furthermore, the @DefineScope annotation must be added to variable declarations holding ScopedMemory objects. The annotation has the form @DefineScope(name="A", parent="B") where A is the symbolic name of the scope represented by the object and B is the name of the direct ancestor of the scope.

@Scope Annotation

@Scope annotations can be attached to class declarations to constrain the scope in which all instances of that class are allocated. The annotation has the form @Scope("A") where A is the name of a scope introduced by @DefineScope. All methods in the class run in the specified scope by default.

Annotating a field, local or argument declaration constrains the object referenced by that field to be in a particular scope.

Lastly, annotating a method declaration constrains the value returned by that method.

Inner classes that are static are independent from the @Scope annotation on the enclosing classes, non-static inner classes must preserve and restate the @Scope annotation of the enclosing class.

Scope IMMORTAL, CALLER, THIS, and UNKNOWN

The special scope name IMMORTAL is used to denote the singleton instance of ImmortalMemory.

The CALLER, THIS and UNKNOWN scope values can be used in @Scope annotations to increase code reuse. A reference that is annotated CALLER is allocated in the same scope as the allocation context (more on the allocation context in Section 1.4.2). Classes may be annotated CALLER to denote that instances of the class may be allocated in any scope.

References annotated THIS point to objects allocated in the same scope as the receiver (i.e. the value of this) of the current method.

Lastly, UNKNOWN is used to denote unconstrained references for which no static information is available.

@RunsIn Annotation

The @RunsIn annotation can be annotated on a method, it specifies the context for that particular method, overriding any annotations on its enclosing type. This can be used, for example, to annotate event handlers, which always execute its event handling code in a different scope from which it was allocated. This annotation follows the same form as @Scope.

An argument of CALLER indicates that the method is scope polymorphic and that it can be invoked from any scope. In this case, the arguments, local variables, and return value are by default assumed to be CALLER. If the method arguments or returned value are of a type that has a scope annotation, then this information is used by the Checker to verify the method. If a variable is labeled @Scope(UNKNOWN), the only methods that may be invoked on it are methods that are labeled @RunsIn(CALLER). @RunsIn(THIS) denotes a method which runs in the same scope as the receiver.

Default Annotation Values

For class declarations, the default value is @Scope(CALLER). This is also the annotation on Object. This means that when annotations are omitted classes can be allocated in any scope (and thus are not tied to a particular scope). Local variables

Class	Constructor	Constructor Parameters	Field
@Scope(Name)	@RunsIn(Name) @Scope(Name)	@Scope(Name)	@Scope(Name)
@Scope(CALLER)	@RunsIn(CALLER) @Scope(CALLER)	@Scope(THIS)[a]	@Scope(THIS)

Class	Method	Method Parameters	Local Variable
@Scope(Name)	@RunsIn(Name) @Scope(Name)	@Scope(Name)	@Scope(Name)
	@RunsIn(CALLER) @Scope(CALLER)	@Scope(CALLER)	@Scope(CALLER)
@Scope(CALLER)	@RunsIn(THIS) @Scope(THIS)	@Scope(THIS)[b]	@Scope(THIS)[c]
	@RunsIn(CALLER) @Scope(CALLER)	@Scope(CALLER)	@Scope(CALLER)[d]

---

[a] Where THIS refers to the enclosing class, a parameter from caller's scope is expected to be passed in.

[b] Where THIS refers to the enclosing class, at the caller's side the scope of the parameter must be the same as the scope of the method invocation receiver.

[c] Because the enclosing method is @RunsIn(THIS).

[d] Because the enclosing method is @RunsIn(CALLER).

Table 1.3: Summary of default annotations for a class annotated with a named scope and a class annotated as CALLER.

and arguments default to CALLER as well. For fields, we assume by default that they infer to the same scope as the object that holds them, i.e. their default is THIS. Instance methods have a default @RunsIn(THIS) annotation. The Table 1.3 summarizes the values of default annotations for all the source-code elements. Consider the following:

- For @Scope(Name) classes:
  - The unannotated fields and method/constructor parameters of unannotated types are by default @Scope(Name).
  - Constructors are automatically annotated @RunsIn(Name).
- For @Scope(CALLER) classes:
  - Constructors are automatically annotated @RunsIn(CALLER). This is the only case when the @Scope(CALLER) annotation of the class has an effect on its body, in fact the class' @Scope(CALLER) annotation is considered only during its instantiation.

- The unannotated fields and method/constructor parameters of unannotated types are by default @Scope(THIS).

Note on the notation: The Table 1.3 includes the cases where the class annotation is @Scope(Name). This not only means that the given annotation has a value of a named scope but that this same value must match all the named scope values for the corresponding lines of the table. For example, if the class is annotated @Scope ("S1"), where S1 is a name of a scope, then the default annotations on the class constructors are @Scope ("S1") and @RunsIn ("S1"). The similar notation is adopted in the remainder of this chapter, for every table containing a scope of a value Name, the same scope value must match all the occurrences of the Name on the given line.

Static Fields and Methods

The static constructors are treated as implicitly annotated @RunsIn(IMMORTAL). Static fields are treated as annotated @Scope(IMMORTAL). Thus, static variables follow the same rules as if they were explicitly annotated with IMMORTAL. Every static field must have types that are annotated @Scope(IMMORTAL) or are unannotated.

Static methods are treated as being annotated CALLER.

Strings constants are immutable and therefore are treated as CALLER.

Overriding annotations

The following rules apply for overriding of the memory safety annotations:

1. Class annotation overriding rules:
    (a) @DefineScope annotation cannot be overriden nor restated.
    (b) Subclasses must preserve the @Scope annotation. A subclass of a class annotated with a named scope must retain the exact same scope name. A subclass of a class annotated CALLER may override this with a named scope.
    (c) s, if the class that the method belongs to is annotated @Scope(s)
2. Method annotation overriding rules: Any @RunsIn annotation may be overriden. Further rules apply to upcasting of types that have overriden a @RunsIn annotation, see Section 1.4.5.

1.4.2 Allocation Context

An allocation context of a method is a scope and its value is the first of:

1. CALLER, if the method is static,
2. s, if the method is annotated @RunsIn(s),
3. CALLER, if the method is annotated @RunsIn(CALLER),
4. s, if the method is annotated @RunsIn(THIS) and if the class that the method belongs to is annotated @Scope(s),
5. s if the method has no annotation and the class that the method belongs to is annotated @Scope(s),
6. THIS if the method is annotated @RunsIn(THIS) and if the class that the method belongs to is annotated @Scope(CALLER) or has no annotation,
7. THIS.

For any given expression, its allocation context is the allocation context of the enclosing method.

1.4.3 Dynamic Guards

Dynamic guards are equivalent of dynamic type checks. They are used to recover the static scope information lost when a variable is cast to UNKNOWN. A dynamic guard is a conditional statement that tests the value of one of two pre-defined methods, allocatedInSame() or allocatedInParent() or, to test the scopes of a pair of references. If the test succeeds, the check assumes that the relationship between the variables holds. The parameters to a dynamic guard are local variables which must be final to prevent an assignment violating the assumption. The following example illustrates the use of dynamic guards.

```
void method(@Scope(UNKNOWN) final List unk, final List cur) {
  if (ManagedMemory.allocatedInSame(unk, cur)) {
    cur.tail = unk;
  }
}
```

The method takes two arguments, one List allocated in an unknown scope, and the other allocated in the THIS scope. Without the guard the assignment statement would not be valid, since the relation between the objects' scopes can not be validated statically. The guard allows the checker to assume that the objects are allocated in the same scope and thus the method is deemed valid. Note that the parameters to allocatedInSame() and allocatedInParent() must be final, so that the variables cannot be modified to violate the assumption.

1.4.4 Scope Concretization

The value of polymorphic annotations such as THIS and CALLER can be inferred from the allocation context in certain cases. A concretization function translates THIS or CALLER to a named scope where possible. For instance a variable annotated THIS takes the scope of the enclosing class (if the class has a named scope). An object returned from a method annotated CALLER is concretized to the value of the calling method's @RunsIn which, if it is THIS, can be concretized to the enclosing class' scope. and to a class that is enclosing the method corresponding to the given allocation context. Therefore, let:

A scope concretization function *conc(S,C,AC)* if a function of of three parameters where :

- S is a scope value,
- AC is the scope of a given allocation context,
- C is the scope of a class enclosing the given allocation context.

and returns one of the following:

- UNKNOWN if S has a value UNKNOWN,
- Name, where Name is some named scope, if either
    - S represents the value Name,
    - S is THIS and C is Name.
- THIS if S is THIS and C is CALLER.
- Name, where Name is some named scope, and S is CALLER and AC is Name.
- CALLER if S is CALLER and AC is CALLER,
- *conc(THIS,C,AC)* if S is CALLER and AC is THIS.

Note that :

- The concretization function does not necessarily yield a named scope.
- While CALLER can be concretized to THIS, the THIS scope can never be concretized to CALLER.
- Concretization of the method's @RunsIn and @Scope annotations is automatically handled by the default annotations rules presented in Section 1.4.1. The THIS scope is concretized to Name if the enclosing class has a @Scope annotation, otherwise stays THIS. The CALLER scopes on methods cannot be further concretized.

Equality of two scopes

We say that **two scopes are equal** if they are identical after concretization. The equality can be also denoted by the == operator.

1.4.5 Scope of an Expression

Every expression must have a scope, if the scope of an expression cannot be determined, the expression is deemed invalid.

The discussion in this section is based on the scope concretization rules presented in Sec.1.4.4 and thus all the scope values discussed are already concretized to their most concrete value (i.e. scopes THIS and CALLER cannot be further concretized to a named scope).

1.4.2.1 Simple expressions

To determine a scope of a simple expression, we list all the possible cases in Tab. 1.5. For a simple expression, the final scope of an expression is then determined as a concretization function applied to a corresponding valid scope value of the basic expression.

Simple Expression	Result Scope
static expr.	IMMORTAL
enum types	IMMORTAL
string concatenation	*conc(*CALLER*)*
string literal	*conc(*CALLER*)*
`this.` or `super.`	*conc(*THIS*)*
local variable	Name/*conc(*THIS/CALLER*)*/UNKNOWN

Table 1.4: Scope of a basic expression.

Considering the table, note that:

- **Local Variables:**
  - Local variables, unlike fields and parameters, may have no particular scope associated with them when they are declared and are of a type that is unannotated. We therefore bind the variable to the scope of the right-hand side expression of its first assignment. In the following example

```
Integer myInt = new Integer();
```

if the containing method is @RunsIn(CALLER), myInt is bound to @Scope(CALLER) while the variable itself is still in lexical scope. In other words, it is as if myInt had an explicit @Scope(CALLER) annotation on its declaration.

- Once a scope is associated with a given variable, it cannot be changed. For example, it would be illegal to have the following assignment in the method body once myInt was already bound to @Scope(CALLER):

```
myInt = Integer.MAX_INT;
```

- **String Concatenation:** The concatenation of the two operand strings results in a new string with a scope value of *conc(*CALLER*)*. The scopes of the operand strings do not have any influence on the scope of the resulting string.

1.4.2.2 Field access

Consider a field access expression e1.f, let:

- S1 be the scope of expression e1,
- S2 be the scope of the field f.

Then, **the scope of an expression** e1.f is S and all its possible values are listed in Tab. 1.6.

S1	S2	S
THIS	THIS	THIS
Name1	Name2	Name2
Name	THIS	Name
CALLER	THIS	CALLER
any	UNKNOWN	UNKNOWN
UNKNOWN	Name	Name
UNKNOWN	THIS	UNKNOWN

Table 1.5: Scope of a field access expression.

1.4.2.3 Assignment expressions

Consider assignment expression e1= e2, let :

- S1 be the scope of expression e1, and
- S2 be the scope of expression e2.

Then this assignment expression is valid iff one of the following holds:

1. S1 == S2, or
2. S1 == UNKNOWN, or
3. If the expression e1 is in a form e3.f where
    - e3 is an expression and f is a field, and
    - S3 is the scope of the field access expression e3.f.

    Then the assignment is valid iff:

    (a) S3 == S2, or
    (b) f is UNKNOWN and the expression is protected by the dynamic guard MemoryArea.allocatedInParent(x.f,y), or
    (c) e1.f is THIS and the expression is protected by the dynamic guard MemoryArea.allocatedInSame(x.f,y).

1.4.2.4 Cast expression

A cast expression (C) e may refine the scope of an expression from an object annotated with CALLER, THIS, or UNKNOWN to a named scope. For example, casting a variable declared @Scope(UNKNOWN)Object to C entails that the scope of expression will be that of C. Casts are restricted so that no scope information is lost.

Therefore, consider a cast expression:

```
(A) e;
```

Let:

- the class A be declared as @Scope(S1) class A {...},
- the class B be declared as @Scope(S2) class B extends A {...},
- the type of the expression e be B,
- AC be the scope of the allocation context of the method enclosing the cast expression.

then, the cast expression is valid iff one of the following applies:

- S1 == S2,
- S1 == CALLER and S2 == AC,

A scope of this cast expression is *conc(S1)*.

@RunsIn overriding rule: The following rule related to overriding of the @RunsIn annotation applies for casts:

- Cast is forbidden if the subtype overrides the @RunsIn annotation on a method of the supertype and the method is not annotated SUPPORT.

1.4.2.5 Method invocation

Consider a method invocation e1.m(...,e2,...), let:

- AC be the scope of the allocation context of the caller,
- ACM be the scope of the allocation context of the invoked method m(),
- T be the scope of the expression e1,
- A be the scope of the expression e2,
- P be the concretized scope of the formal parameter from the method's m() declaration corresponding to the actual argument expression e2,
- SM be the concretized value of the @Scope annotation of the method m(),
- S be the scope of this method invocation expression.

Then, such a method invocation is valid iff I. and II. are valid, and the scope of a method invocation is then determined by III.:

I. Method scope check: one of the following must be valid:

1. The method m is *static*, or

2. The scope ACM is parent to the scope AC and the method m() is annotated @SCJRestricted(mayAllocate=false), or

3. One of the valid cases listed in the Table 1.7 applies.

ACM	T	AC
CALLER	any	any
Name	any	Name
THIS	Name	Name
THIS	THIS	THIS
THIS	CALLER	THIS
THIS	CALLER	CALLER

Table 1.6: Valid method invocation

Note the following:

(a) The cases where the ACM is CALLER or Name are trivial to resolve.

(b) The only non-trivial case is when the ACM == THIS and it cannot be further concretized since its enclosing class is CALLER. In this case, T == THIS or CALLER and the following applies:

   i. if AC == CALLER, then:

      A. If T == THIS then this is invalid method call.

   B. If T == CALLER then this is valid because AC == T == CALLER == THIS.

ii. If AC == THIS then this method call is valid since T == THIS == CALLER.

II. Method parameter check: assignment of method parameters must be valid, therefore, one of the cases listed in Tab. 1.8 must apply.

P	A	AC	T
Name	Name	any	any
THIS	Name	any	Name
THIS	THIS	THIS	CALLER
THIS	CALLER	CALLER	CALLER
THIS	Name	Name	CALLER
THIS	THIS	any	THIS
CALLER	CALLER	CALLER	any
CALLER	Name	Name	any
CALLER	THIS	THIS	any
UNKNOWN	any	any	any
THIS	any	any	UNKNOWN[a]

---

[a] Must be guarded by a dynamic guard.

Table 1.7: Valid parameter assignment

III. Scope of a method invocation expression: For a valid method invocation expression, all the possible scope values S of such an expression are listed in Tab. 1.9.

SM	T	AC	S
Name	any	any	Name
THIS	Name	Name	Name
THIS	CALLER	CALLER	CALLER
THIS	THIS or CALLER	THIS	THIS
CALLER	any	CALLER	CALLER
CALLER	any	Name	Name
CALLER	any	THIS	THIS
UNKNOWN	any	any	UNKNOWN

Table 1.8: Scope of a method invocation expression

1.4.2.5 Allocation expression

Consider an allocation expression new C(y), let:

- A is the scope of an expression y,
- P is a concretized scope of a formal parameter from the constructor declaration corresponding to the actual argument expression y,
- AC is the scope of the allocation context of the method enclosing the allocation expression.
- S is the scope of the class C.

Then this allocation expression is valid iff the following holds:

1. One of the following must hold:
    - AC == S,
    - S == CALLER (the class C can be instantiated anywhere).
2. Constructor parameter assignment must be corresponding to one of the valid cases listed in Tab. 1.10.

P	A	AC
Name	Name	any
THIS or CALLER	Name	Name
THIS or CALLER	CALLER	CALLER
THIS or CALLER	THIS	THIS
UNKNOWN	any	any

Table 1.9: Valid parameter assignments in constructors.

and, **the scope of an allocation expression** is conc(S).

Further, the following rules apply in general for any field or variable declaration:

- A variable or field declaration, C x, is valid if the allocation context is the same or a child of the @Scope of C. Consequently, classes with no explicit @Scope annotation cannot reference classes which are bound to named scopes, since THIS may represent a parent scope.
- By default, the allocation context of an array T[] is the same as that of its element class, T.
- Primitive arrays are considered to be labeled THIS. The default can be overriden by adding a @Scope annotation to an array variable declaration.

1.4.6 Additional rules and restrictions

The SCJ memory safety annotation system further dictates a following set of rules specific to SCJ API methods.

```
@Scope("M") @DefineScope(name="H", parent="M")
class Handler extends PeriodicEventHandler {

  @RunsIn("H") @SCJAllowed(SUPPORT) void handleAsyncEvent() {
    @Scope(IMMORTAL) @DefineScope(name="M", parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    ...
    @DefineScope(name=IMMORTAL,parent=IMMORTAL) @Scope(IMMORTAL)
    ImmortalMemory imm = (ImmortalMemory) ImmortalMemory.instance;
  }
}
```

Figure 1.1: Annotating ManagedMemory object example.

MissionSequencer and Mission

The MissionSequencer must be annotated with @DefineScope, its getNextMission() method has a @RunsIn annotation corresponding to this newly defined scope. Every Mission associated with a particular MissionSequencer is instantiated in this scope and it must have a @Scope annotation corresponding to that scope. Further, MissionSequencer must have @Scope annotation corresponding to the parent scope defined by the @DefineScope annotation.

Schedulables

Each Schedulable must be annotated with a @DefineScope and @Scope annotation. There can be only one instance of a Schedulable class per Mission.

MemoryArea Object Annotation

The annotation system requires every object representing a memory area to be annotated with @DefineScope and @Scope annotations. The annotations allow the checker to statically determine the scope name of the memory area represented by the object. This information is needed whenever the object is used to invoke MemoryArea and ManagedMemory API methods, such as newInstance() or executeInArea() and enterPrivateMemory().

The example in Fig. 1.1 demonstrates a correct annotation of a ManagedMemory object m. The example shows a periodic event handler instantiated in memory M that runs in memory H. Inside the handleAsyncEvent method we retrieve a ManagedMemory object representing the scope M. As we can see, the variable declaration is annotated with @Scope annotation, expressing in which scope the memory area object is allocated – in this case it is the IMMORTAL memory. Further, the @DefineScope annotation is used to declare which scope is represented by this instance.

EnterPrivateMemory() and ExecuteInArea() methods

Calls to a scope's executeInArea() method can only be made if the scoped area is a parent of the allocation context. In addition, the Runnable object passed to the method must have a @RunsIn annotation that matches the name of the scoped area. This is a purposeful limitation of what SCJ allows, since the system does not know what the scope stack is at any given point in the program.

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 allocation context. The @RunsIn annotation of the runnable's run() method must be the name of the scope being defined by @DefineScope. The enterPrivateMemory() method cannot be invoked if the allocation context is CALLER.

newInstance()

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 or not annotated at all. Similarly, calls to newArrayInArea() are only legal if the element type is annotated to be in the same scope as the first parameter or not annotated at all. The expression

ImmortalMemory.instance().newArray(**byte.class**, 10)

should therefore have the scope IMMORTAL. An invocation MemoryArea.newArrayInArea(o, byte.class, 10) is equivalent to calling MemoryArea.getMemoryArea(o).newArray(byte.class, 10). In this case, we derive the scope of the expression from the scope of o.

getCurrent*() methods.

The getCurrent* methods are static methods provided by SCJ API that allow applications to access objects specific to the SCJ infrastructure. The getCurrent*() methods are:

- ManagedMemory.getCurrentManagedMemory(),
- RealtimeThread.getCurrentMemoryArea(),
- MemoryArea.getMemoryArea(),
- Mission.getCurrentMission(),
- MissionManager.getCurrentMissionManager(), and
- Scheduler.getCurrentScheduler().

Typically, an object returned by such a call is allocated in some upper scope; however, there is no annotation present on the type of the object. To explicitly express that the allocation scope of returned object is unknown, the getCurrent*() methods are annotated with @RunsIn(CALLER) and the returned type of such a method call is @Scope(UNKNOWN).

1.4.7 Validation

The first step to validation of these annotations requires the construction of a reachable class set (RCS), this is the set of all classes that may be manipulated by a SCJ schedulable object. The RCS is constructed by starting with all classes that are annotated @Scope and adding all classes that may be instantiated from run() methods and methods called from run() methods. Therefore, to ensure a successful verification of memory safety annotations, all the source files should be accessible and passed into the checker at the same time.

We say that a SCJ application is valid if it contains only valid expressions according to the rules described in Sec. 1.4.5 and Sec. 1.4.6.

Disabling Verification of Scope Safety Annotations

The verification of scope safety annotations can be disabled by a compilation parameter -AnoScopeChecks passed to the checker. In this case, only the level compliance annotations and behavior restricting annotations are verified.

1.5 Level Considerations

These annotations apply to all levels.

1.6 API

1.6.1 Class javax.safetycritical.annotate.SCJRestricted

Declaration

@Retention(**CLASS**)
@Target({ TYPE, FIELD, METHOD, CONSTRUCTOR })
public @interface SCJRestricted

Methods

public Restrict[] value() **default** {ANY_TIME}
public boolean mayAllocate() **default** true
public boolean maySelfSuspend() **default** false;

Declaration

This annotation distinguishes methods that may be called only from a certain context (e.g. CleanUp) or method that may be restricted to execute no memory allocation or blocking.

1.6.2 Class javax.safetycritical.annotate.SCJAllowed

Declaration

```
@Retention(CLASS)
@Target( { TYPE, FIELD, METHOD, CONSTRUCTOR })
public @interface SCJAllowed
```

Description

This annotation distinguishes methods, classes, and fields that may be accessed from within safety-critical Java programs. In some implementations of the safety-critical Java specification, elements which are not declared with this annotation (and are therefore not allowed in safety-critical application software) are present within the declared class hierarchy. These are necessary for full compatibility with standard edition Java, the Real-Time Specification for Java, and/or for use by the implementation of infrastructure software. The value field equals LEVEL_0 for elements that may be used within safety-critical Java applications targeting levels 0, 1, or 2. The value field equals LEVEL_1 for elements that may be used within safety-critical Java applications targeting levels 1 or 2. The value field equals LEVEL_2 for elements that may be used within safety-critical Java applications targeting level 2. Absence of this annotation on a given Class, Field, Method, or Constructor declaration indicates that the corresponding element may not be accessed from within a compliant safety-critical Java application.

Methods

public Level value() **default** LEVEL_0

1.6.3 Class javax.safetycritical.annotate.Level

Declaration

public enum Level

LEVEL_0

LEVEL_1
LEVEL_2
SUPPORT
INFRASTRUCTURE
HIDDEN

Description

Provides a set of possible values for the @SCJAllowed annotation's argument *level*.

1.6.4 Class javax.safetycritical.annotate.Phase

Declaration

public enum Phase

INITIALIZE
EXECUTION
CLEANUP
ANY_TIME

Description

Provides a set of possible values for the @SCJRestricted annotation value.

1.7 Rationale and Examples

It is expected 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). 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.

The scoped memory area classes extend Java to provide an API for circumventing the need for garbage collection. In Java, the type system guarantees that every access to an object is valid, the garbage collector only recycles objects that are not reachable. Since scoped memory is not garbage collected, it would be possible for the application to retain a reference to a scoped-allocated object, and access the memory after the scope was reclaimed. This could lead to memory corruption and crash the entire virtual machine. In order to ensure memory safety, the RTSJ mandates a number of runtime checks on operations such as memory reads and writes as well as calls to scoped memory enter() and executeInArea(). Exceptions will be thrown if the program performs an operation that may lead to an unsafe memory access.

Practical experience with the RTSJ has shown that memory access rules are difficult to get right because the allocation context is implicit and programmers are not used to reasoning in terms of the relative position of objects in the scope hierarchy. In a safety-critical context, these exceptions must never be thrown as they are likely to lead to application failures. Validated programs are guaranteed to never throw any of the following exceptions:

- IllegalAssignmentError occurs when an assignment may result in a dangling pointer. In other words, it occurs when an attempt is made to store a reference to an object where the reference is below the memory area in the scope stack.
- ScopedCycleException is thrown when an invocation of enter() on a scope would result in a violation of the single parent rule, which basically states that a scoped memory may only be entered from the same parent scope while it is active.
- InaccessibleAreaException is thrown when an attempt is made to access a memory area that is not on the scope stack (e.g., calling executeInArea() on it).

1.7.1 Compliance Level Annotation Example

The following example illustrates application of the compliance level annotation on a simple example. The example shows both user and infrastructure fragments of source code, demonstrating the application of the compliance level annotations.

```
@SCJAllowed(LEVEL_0, members=true)
class MyMission extends CyclicExecutive {

    WordHandler peh;

   @SCJAllowed(SUPPORT) public void initialize() {
    peh = new MyHandler(...);            // ERROR
    peh.run();                   // ERROR
   }
}
```

As we can see, all the elements of the example are declared to reside in a specific compliance level. At the user domain, we define class MyMission that is declared to be 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 a MyHandler object and consequently tries to execute its functionality by calling PeriodicEventHandler's run() method. However, the method is annotated as @SCJAllowed(INFRASTRUCTURE), which indicates that it can be called only from the SCJ infrastructure code.

```
@SCJAllowed (LEVEL_0)
public interface Schedulable extends SCJRunnable {

 @SCJAllowed(LEVEL_2)
 public ReleaseParameters getReleaseParameters();
}

@SCJAllowed(LEVEL_1)
class MyHandler extends PeriodicEventHandler {

  @SCJAllowed(SUPPORT) public void handleAsyncEvent() {...}
}

@SCJAllowed(LEVEL_0)
public abstract class PeriodicEventHandler extends ManagedEventHandler {

  @SCJAllowed(LEVEL_0) public PeriodicEventHandler(..) {...}

  @SCJAllowed(LEVEL_0)     // ERROR
  public ReleaseParameters getReleaseParameters() {...}

  @SCJAllowed(INFRASTRUCTURE) public final void run() {...}
}
```

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

1.7.2 Memory Safety Annotations Example

The following user-level code snippet illustrated application of memory safety annotations. The example shows a user-domain source code fragment that defines a MyMission class, where we explicitly declare a scope in which the mission is running by @DefineScope(name="M",parent=IMMORTAL). Furthermore, mission's handler MyHandler is defined to be allocated in mission's memory by @Scope("M"), while running in its own handler's private memory by @RunsIn("H"), defined by according @DefineScope annotation.

```
@Scope(IMMORTAL) @DefineScope(name="M", parent=IMMORTAL)
@SCJAllowed(members=true) class MyMission extends CyclicExecutive {

   @SCJAllowed(SUPPORT) public void initialize() {
      new MyHandler(...);
   }
}
```

```
@DefineScope(name="M", parent=IMMORTAL) @Scope("M")
@SCJAllowed(members=true) class CDMission extends Mission {

  @SCJAllowed(SUPPORT) @RunsIn("M") void initialize() {
    new Handler().register();
    MIRun run = new MIRun();
    @Scope(IMMORTAL) @DefineScope(name="M", parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    m.enterPrivateMemory(2000, run);
  }
}

@SCJAllowed(members=true)
@Scope("M") @DefineScope(name="MI", parent="M")
class MIRun implements SCJRunnable {
  @SCJAllowed(SUPPORT) @RunsIn("MI") void run() {...}
}
```

Figure 1.2: CDx mission implementation.

```
@Scope("M") @DefineScope(name="H", parent="M")
@SCJAllowed(members=true) class MyHandler extends PeriodicEventHandler {

  @SCJAllowed(SUPPORT) @RunsIn("H") public void handleAsyncEvent() {
     ManagedMemory.getCurrentManagedMemory().
         enterPrivateMemory(3000, new Run());
  }
}

@Scope("H") @DefineScope(name="R", parent="H")
@SCJAllowed(members=true) class Run implements SCJRunnable {

    @SCJAllowed(SUPPORT) @RunsIn("R") public void run() {...}
}
```

The user is also expected to define a new scope area any time code enters a child scope. This is illustrated by the Run class that is allocated in MyHandler private memory while running in its own scope. Note the annotations on the Run class, the @DefineScope is used to define a new scope entered by the runnable, furthermore, the @RunsIn annotation specifies the allocation context of the run() method. Notice that the memory areas form a scope tree with the immortal scope in root.

1.7.3 A Large-Scale Example

In this section we present a Collision Detector (CDx) example and illustrate the use of the memory safety annotations. The classes are written with a minimum number of annotations, though the figures hides much of the logic which has no annotations at all.

```
@DefineScope(name="H", parent="M") @SCJAllowed(members=true)
@Scope("M") class Handler extends PeriodicEventHandler {

  Table st;

  @SCJAllowed(SUPPORT) @RunsIn("H") void handleAsyncEvent() {
    Sign s = ... ;
        @Scope("M") V3d old_pos = st.get(s);
        if (old_pos == null) {
          @Scope("M") Sign n_s = mkSign(s);
          st.put(n_s);
        } else ...
  }

  @RunsIn("H") @Scope("M") Sign mkSign(@Scope("M") Sign s) {
    @Scope(IMMORTAL) @DefineScope(name="M",parent="IMMORTAL")
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(s);

    @Scope("M") Sign n_s = ManagedMemory.newInstance(Sign.class);
    n_s.b = (byte[]) MemoryArea.newArrayInArea(s, byte.class, s.length);
    for (int i : s.b.length) n_s.b[i] = s.b[i];
    return n_s
  }
}
```

Figure 1.3: CDx Handler implementation.

The example consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The main computation is executed in a private memory area, as the CDx algorithm is executed periodically; data is recorded in a mission memory area. However, since the CDx algorithm relies on positions in the current and previous frame for each iteration, a dedicated data structure, implemented in the Table class, must be used to keep track of the previous positions of each airplane so that the periodic task may reference it. Each aircraft is uniquely represented by its Sign and the Table maintains a mapping between a Sign and a V3d object that represents current position of the aircraft. Since the state table is needed during the lifetime of the mission, placing it inside the persistent memory is the ideal solution.

First, a code snippet implementing the Collision Detector mission is presented in Fig. 1.2. The CDMission class 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 the mission's handler and then shows how the enterPrivateMemory() method is used to perform some initialization tasks in a sub-scope using the MIRun class. The ManagedMemory variable m is annotated with @DefineScope and @Scope to correctly define which scope is represented by this object. Further, notice the use of @DefineScope to define a new MI scope that will be used as a private memory for the runnable.

```
@SCJAllowed(members=true) @Scope("M") class Table {

  final HashMap map;
  V3d vectors [];
  int counter = 0;
  final VRun r = new VRun();

  @RunsIn(CALLER) @Scope(THIS) V3d get(Sign s) {
    return (V3d) map.get(s);
  }

  @RunsIn(CALLER) void put(final @Scope(UNKNOWN) Sign s) {
    if (ManagedMemory.allocatedInSame(r,s)) r.s = s;
    @Scope(IMMORTAL) @DefineScope(name="M",parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    m.executeInArea(r);
  }
}

@SCJAllowed(members=true) @Scope("M") class VRun implements SCJRunnable {

  Sign s;

  @SCJAllowed(SUPPORT) @RunsIn("M") void run() {
    if (map.get(s) != null) return;
    V3d v = vectors[counter++];
    map.put(s,v);
  }
}
```

Figure 1.4: CDx Table implementation.

The Handler class, presented in Fig. 1.3, implements functionality that will be periodically executed throughout the mission in the handleAsyncEvent() method. The class is allocated in the M memory, defined by the @Scope annotation. The allocation context of its execution is the "H" scope, as the @RunsIn annotations upon the Handler's methods suggest.

Consider the handleAsyncEvent() method, which implements a communication with the Table object allocated in the scope M, thus crossing scope boundaries. The Table methods are annotated as @RunsIn(CALLER) and @Scope(THIS) to enable this cross-scope communication. Consequently, the V3d object returned from a @RunsIn(CALLER)get() method is inferred to reside in @Scope("M"). For a newly detected aircraft, the Sign object is allocated in the M memory and inserted into the Table. This is implemented by the mkSign() method that retrieves an object representing the scope M and uses the newInstance() and newArrayInArea() methods to instantiate and initialize a new Sing object.

The implementation of the Table is presented in Fig. 1.4. The figure further shows a graphical representation of memory areas in the system together with objects allocated in each of the areas. The immortal memory contains only an object representing an instance of the MissionMemory. The mission memory area contains the two schedulable objects of the application – Mission and Handler, an instance representing PrivateMemory, and objects allocated by the application itself – the Table, a hashmap holding V3d and Sign instances, and runnable objects used to switch allocation context between memory areas. The private memory holds temporary allocated Sign objects.

The Table class, presented in Fig. 1.4 on the left side, implements several @RunsIn(CALLER) methods that are called from the Handler. The put() method was modified to meet the restrictions of the annotation system, the argument is UNKNOWN because themethod can potentially be called from any subscope. In the method, a dynamic guard is used to guarantee that the Sign object being passed as an argument is allocated in the same scope as the Table. After passing the dynamic guard, the Sign can be stored into a field of the VectorRunnable object. This runnable is consequently used to change the allocation context by being passed to the executeInArea(). Inside the runnable, the Sign is then stored into the map that is managed by the Table class. After calling executeInArea(), the allocation context is changed to M and the object s can be stored into the map. Finally, a proper HashMap implementation annotated with @RunsIn(CALLER) annotations is necessary to complement the Table implementation.

Bibliography