Scoped Types and Aspects for Real-Time Java Memory Management

Chris Andreae[3], Yvonne Coady[1], Celina Gibbs[1],
James Noble[3], Jan Vitek[4], Tian Zhao[2]

(1) University of Victoria, CA. (2) University of Wisconsin–Milwaukee, USA.
(3) Victoria University of Wellington, NZ. (4) Purdue University, USA.

Abstract. Real-time systems are notoriously difficult to design and implement, and, as many real-time problems are safety-critical, their solutions must be reliable as well as efficient and correct. While higher-level programming models (such as the Real-Time Specification for Java) permit real-time programmers to use language features that most programmers take for granted (objects, type checking, dynamic dispatch, and memory safety) the compromises required for real-time execution, especially concerning memory allocation, can create as many problems as they solve. This paper presents Scoped Types and Aspects for Real-Time Systems (STARS) a novel programming model for real-time systems. Scoped Types give programmers a clear model of their programs' memory use, and, being statically checkable, prevent the run-time memory errors that bedevil the RTSJ. Adopting the integrated Scoped Types and Aspects approach can significantly improve both the quality and performance of a real-time Java systems, resulting in simpler systems that are reliable, efficient, and correct.

1 Introduction

The Real-Time Specification for Java (RTSJ) introduces abstractions for managing resources, such as non-garbage collected regions of memory [8]. For instance, in the RTSJ, a series of *scoped memory* classes lets programmers manage memory explicitly: creating nested memory regions, allocating objects into those regions, and destroying regions when they are no longer needed. In a hard real-time system, programmers must use these classes, so that their programs can bypass Java's garbage collector and its associated predictability and performance penalties. But these abstractions are far from abstract. The RTSJ forces programmers to face more low-level details about the behavior of their system than ever before — such as how scoped memory objects correspond to allocated regions, which objects are allocated in those regions, how those regions are ordered — and then deals with mistakes by throwing dynamic errors at runtime. The difficulty of managing the inherent complexity associated with real-time concerns ultimately compromises the development, maintenance and evolution of safety critical code bases and increases the likelihood of fatal errors at runtime.

This paper introduces Scoped Types and Aspects for Real-Time Systems (STARS), a novel approach for programming real-time systems that shields developers from many accidental complexities that have proven to be problematic in practice. Scoped Types

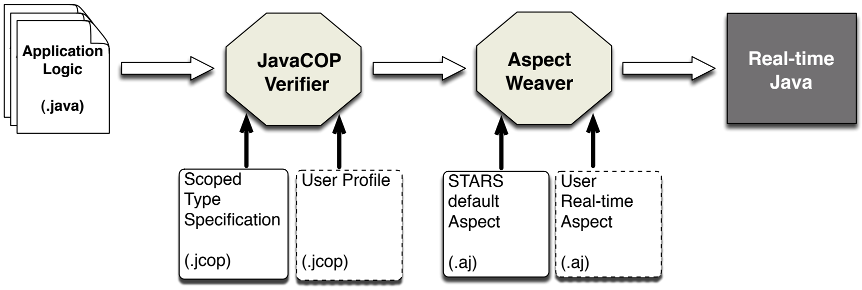


Fig. 1. *Overview of STARS. Application logic is written according to the Scoped Types discipline. The* JAVACOP *verifier uses scoped types rules (and possibly some user-defined application-specific constraints) to validate the program. Then, an aspect weaver combines the application logic with the real-time behavior. The result is a real-time Java program that can be executed on any STARS-compliant virtual machine.*

use a program's package hierarchy to represent the structure of its memory use, making clear where objects are allocated and thus where they are accessible. Real-Time Aspects then weave in allocation policies and implementation-dependent code — separating real-time concerns further from the base program. Finally, Scoped Types' correctness guarantees, combined with the Aspect-oriented implementation, removes the need for memory checks or garbage collection at runtime, increasing the resulting system's performance and reliability. Overall, STARS is a methodology that guides real-time development and provides much needed tool support for the verification and the modularization of real-time programs.

Fig. 1 illustrates the STARS methodology. Programmers start by writing application logic in Java with no calls to the RTSJ APIs. The code is then verified against a set of consistency rules — STARS provides a set of rules dealing with memory management; users may extend these rules with application-specific restrictions. If the program type checks, the aspects implementing the intended real-time semantics of the program can be woven into the code. The end result is a Real-time Java program, which can be run in any real-time JVM that supports the STARS API.

The paper thus makes the following contributions:

1. **Scoped Types.** We use a lightweight pluggable type system to model hierarchical memory regions. Scoped Types are based on familiar Java concepts like packages, classes, and objects, can be explained with a few informal rules, and require no changes to Java syntax.
2. **Static verification** via the JAVACOP pluggable types checker [1]. We have encoded Scoped Types into a set of JAVACOP rules used to validate source code. We also show how to extend the built-in rules with application-specific constraints.
3. **Aspect-based real-time development.** We show how an aspect-oriented approach can help decoupling real-time memory management concerns from the main application logic.

4. **Implementation in a real-time JVM.** We demonstrate viability of STARS with an implementation in the Ovm framework [5]. Only minor changes (18 lines of code in all) were needed to support STARS.
5. **Empirical evaluation.** We conducted a case study to show the impact STARS has on both code quality and performance in a 20 KLoc hard real-time application. Refactoring RTSJ code to a STARS program proved easy and the resulting program enjoyed a 28% performance improvement over the RTSJ equivalent.

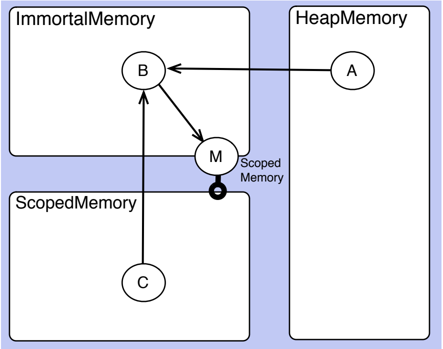
The paper proceeds as follows. After a survey of background and previous work, Section 2 presents an overview of the STARS programming model while Section 3 overviews the current STARS prototype implementations. Section 4 follows with a case study using STARS in the implementation of a real-time collision detection system. Finally we conclude with discussion and future work.

1.1 Background: The Challenges of Real-Time Memory Management

The Real-time Specification for Java (RTSJ) provides real-time extensions to Java that have shown to be effective in the construction of large-scale systems [5, 19, 33]. Two key benefits of the RTSJ are first, that it allows programmers to write real-time programs in a type-safe language, thus reducing opportunities for catastrophic failures; and second, that it allows hard-, soft- and non-real-time tasks to interoperate in the same execution environment. To achieve this second benefit, the RTSJ adopts a mixed-mode memory model in which garbage collection is used for non-real time activities, while manually allocated regions are used for real-time tasks. Though convenient, the interaction of these two memory management disciplines causes significant complexity, and consequently is often the culprit behind many runtime memory errors.

The problem, in the case of real-time tasks, is that storage for an allocation request (i.e. `new`) must be serviced differently from standard Java allocation. In order to handle real-time requests, the RTSJ extends the Java memory management model to include dynamically checked regions known as *scoped memory areas* (or also memory scopes), represented by subclasses of `ScopedMemory`. A scoped memory area is an allocation context which provides a pool of memory for threads executing in it. Individual objects allocated in a scoped memory area cannot be deallocated, instead, an entire scoped memory area is torn down as soon as all threads exit that scope. The RTSJ defines two distinguished scopes for *immortal* and *heap* memory, respectively for objects with unbounded lifetimes and objects that must be garbage collected. Fig. 2 illustrates the allowed reference pattern in RTSJ. Two new kinds of threads are also introduced: *real-time* threads which may access scoped memory areas; and *no heap real-time* threads, which in addition are protected from garbage collection pauses, but which cause dynamic errors if they attempt to access heap allocated objects.

Scoped memory areas provide methods `enter(Runnable)` and `executeInArea(Runnable)` that permit application code to execute within a scope, allocating and accessing objects within that scope. Using nested calls, a thread may enter or execute runnables in multiple scopes, dynamically building up the scope hierarchy. The differences between these two methods are quite subtle [8]: basically, `enter` must be used to associate a scope with a thread, whereas `executeInArea` (temporarily)



1. HeapMemory is garbage collected memory with no timeliness guarantees.
2. ImmortalMemory is not subject to reclamation.
3. ScopedMemory can be reclaimed in a single step if no thread is active in the area.
4. Immortal data can be referenced from any region. Scoped data can only be referenced from same scope or a nested scope. Violations lead to dynamic `IllegalAssignmentError`s.
5. `NoHeapRealtimeThread` cannot load heap references.

A, B, C, and M are objects and their positions indicate where they allocated. M represents the scoped memory where C is allocated. The arrows indicate allowed references between objects.

Fig. 2. *Memory Management in the Real-time Specification for Java.*

changes a thread's active scope to a scope it has previously `enter`ed. Misuse of these methods can cause dynamic errors, e.g. a `ScopedCycleException` is thrown when a user tries to `enter` a `ScopedMemory` that is already accessible. Reference counting on `enter`s ensures that all the objects allocated in a scope are finalized and reclaimed when the last thread leaves that scope.

Real-time developers must take these memory scopes and threading models into account during the design of a real-time system. Scoped memory areas can be nested to form a dynamic, tree-shaped hierarchy, where child memory areas have strictly shorter lifetimes than their parents. Because the hierarchy is established dynamically, memory areas can move around within the hierarchy as the program runs. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime. This means that heap memory and immortal memory cannot hold references to objects allocated in scoped memory, nor can a scoped memory area hold a reference to an object allocated in an inner (more deeply nested) scope. Once again, errors are only detected at runtime and cause runtime exceptions.

Given that safety and reliability are two goals of most real-time systems, the fact that these safety rules are checked *dynamically* seems, in retrospect, to be an odd choice. The only guarantee that RTSJ gives to a programmer is that their programs will fail in a controlled manner: if a dynamic assignment into a dynamically changing scope hierarchy trips a dynamic check, the program will crash with an `IllegalAssignmentError`.

Expressive power. The flip side of the coin is that static memory safety will reduce the expressive power of RTSJ programs. Any static discipline, by its necessarily conservative nature, will rule out some perfectly safe programs. The technique described in this article is no different. In many application contexts, the added safety and decrease in debugging/testing effort will be well worth it. In terms of expressive power, there is an intuitive ordering between the approaches to memory management. Real-time garbage collection is more expressive than scoped memory as there are no restriction on valid pointers. Scoped memory is strictly more expressive than STARS as STARS is implemented by imposing restrictions on the use of the scoped memory API.

It is interesting that the `ScopedMemory` write-barriers are pessimistic in the sense that they prevent valid references from being stored on the ground that they could be used after they become stale. But a stale reference is only dangerous if it is used. One can imagine a real-time system where two sibling scopes are known to the programmer to have equal lifetimes, it would thus be perfectly safe to let them refer to each other as the cross-scope references would not be used after they become stale. Yet, there is no way to do this with the RTSJ as it stands. One could envision a more expressive memory model where reference are coupled with time stamps and read barriers are used to ensure validity.

The tension between expressiveness and correctness can show up in other subtle ways. When programming with the RTSJ, especially on a large project, the best programming practices in large teams can end up being more restrictive than a static type system – just to be on the safe side. Consider for example the changes that had to be made to the standard library class `Vector` for it to be RTSJ-safe:

```
 protected final MemoryArea
       thisArea = MemoryArea.getMemoryArea(this);

 public synchronized void trimToSize() {
   Object[] newArray = null;
   try {
      newArray = (Object[]) thisArea.newArray(Object.class,
                                              elementCount);
   } catch (IllegalAccessException iae) {
      throw new InternalError("Not possible " + iae);
   }
   System.arraycopy(elementData, 0, newArray, 0, elementCount);
   elementData = newArray;
}
```

The non-RTSJ version of this class, when it needs to resize or copy an instance, simply allocates a new `Object` array of the right size. In the RTSJ version of the code, this is not safe as one does not know for sure where the code is currently executing. Allocating the object in the current memory area may result in an violation of the scoped memory rules (this would occur if the current scope is a child of the scope in which the `Vector` was allocated).

To be scope safe, the designer of the RTSJ version of `Vector` decided to allocate all internal array objects in the memory area where the original `Vector` object was allocated. This is not the only solution, but it is at least safe. The implication is that what could have a been a fast straightforward call to a linear-time memory allocator turns into a reflective allocation. Furthermore, to reduce the cost of the operation, the `Vector` instance caches the memory area where the instance was allocated in a field of the object.

There are two conclusions to be drawn from this small example. Firstly, even without a static typing discipline, programmers will have to take steps to make sure that code is scope-safe. In a large project this steps may impose stringent restrictions on expressive power. Secondly, reusability of libraries, while desirable, is questionable. The `trimToSize()` method is almost unusable in a scoped context as it is not able to deallocate the original array (the class, like most of Java code, was written with the assumption of a garbage collector reclaiming unreachable objects).

1.2 Related Work: Programming with Scoped Memory

The Real-time Specification for Java [8] was published in 2000. In later paper Bollella and Reinholtz argued for the need for scoped memory [9]. Readers interested in a discussion are encouraged to consults the paper by Wellings and Puschner [38]. Design patterns for programming with scoped memory have been investigated by several authors [7, 31, 32]. The design and implementation of RTSJ virtual machines have been documented in [11, 17, 2]. Several papers have reported on experience with programming with RTSJ [10, 29, 34, 19, 2].

Beebee and Rinard provided one of the early implementations of the RTSJ memory management extensions [6]. They found it “close to impossible” to develop error-free real-time Java programs without some help from debugging tools or static analysis. The difficulty of programming with RTSJ motivated Kwon, Wellings and King to propose Ravenscar-Java [27], which mandates a simplified computational model. Their goal was to decrease the likelihood of catastrophic errors in mission critical systems. Further work along these lines transparently associates scoped memory areas with methods, avoiding the need for explicit manipulation of memory areas [26]. Limitations of this approach include the fact that memory areas cannot be multi-threaded.

In contrast, systems like Islands [23], Ownership Types [30], and their successors restrict the scope of references to enable modular reasoning. The idea of using ownership types for the safety of region-based memory was first proposed by Boyapati et al. [12], and required changes to the Java syntax and explicit type annotations. Research in type-safe memory memory management, message-based communication, process scheduling and the file system interface management for Cyclone, a dialect of C, has shown that it is possible to prevent dangling pointers even in low-level codes [21]. The RTSJ is more challenging than Cyclone as scopes can be accessed concurrently and are first-class values.

Scoped types are one of the latest developments in the general area of type systems for controlled sharing of references [39]. This paper builds on Scoped Types and proposes a practical programming model targeting the separation of policy and mechanism within real-time applications. The key insight of Scoped Types is the necessity to make the nested scope structure of the program explicit: basically, every time the programmer writes an allocation expression of the form `new Object()`, the object’s type shows where the object fits into the scope structure of the program. It is not essential to know which particular scope it will be allocated in, but rather the object’s hierarchical relationship to other objects. This ensures that when an assignment expression, e.g. `obj.f=new F()`, is encountered, Scoped Types can statically (albeit conservatively) ensure that the assignment will not breach the program’s scope structure.

Deters and Cytron investigated a technique for inferring memory regions for RTSJ programs in [18]. They use traces to identify the lifetime and visibility requirements of objects. Running their tool they obtain very fine grain information about lifetime of objects. They found scope hierarchies that were 80 levels deep. The limitation of that study was that it did not take multi-threading into account and looked at non-real-time codes such as the `javac` compiler. Furthermore, using dynamic traces is always problematic because the results are dependent on the input to the program. Another approach is to try to infer scopes by static analysis as has been done in [16, 28, 20]. The difficulty with static analysis-based approach is that the results are inherently conservative. Small changes in the source code can cause an analysis to move an allocation site to `ImmortalMemory`. When this happens, there is no feedback to the programmer. Thus, it is likely that the testing burden will increase. As none of the published papers has been run on real-time programs, the applicability of these techniques remains an open question.

Spoonhower et al. proposed a new abstraction called Eventrons [35] for memory-safe real-time programming. The role of Eventrons is to circumvent interference with the garbage collector by using objects that are not managed by the collector. The difference between Eventrons and the RTSJ are that Eventrons are statically safe. An Eventron is scheduled periodically and is allowed to preempt the garbage collector. The invariant that must be maintained for this to be safe is that all objects manipulated by the Eventron must be pinned down so that they are not moved by the collector. Furthermore, during its execution, an Eventron must not access objects that are in the garbage collected heap as these may be in an inconsistent state. To ensure safety, Eventrons impose restrictions on the programming model; there can be no allocation within an Eventron, an Eventron is not allowed to store into a reference field and finally it is not allowed to perform blocking operations. The constraints are enforced by a run-time data-sensitive inter-procedural analysis. Eventrons are thus a more restrictive model than STARS, but represent a very interesting point in the design space. They have been incorporated into IBM's product RTSJVM under the name of XRT.

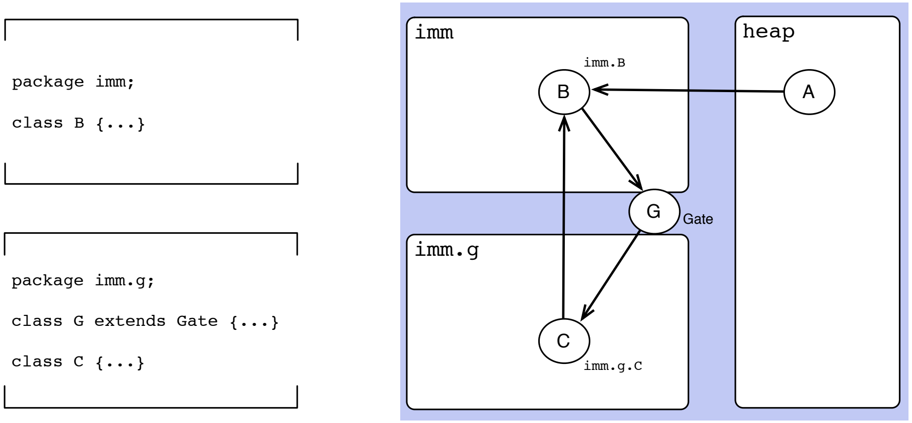
Safety-Critical Java. Under the Java community process JSR-302, a new standard for safety-critical application is currently being developed. This standard will restrict the RTSJ programming model to simplify the task of verification and validation of safety-critical codes. Like STARS, JSR-302 will propose a simplified programming model. One of the current candidates for this specification is [22]. Among the recommendations that are being considered are a simplification of the memory model and of the threading API. We expect that STARS could be adapted to match the safety-critical standard and provide additional static guarantees.

2 The STARS Programming Model

STARS guides the design and implementation of real-time systems with a simple, explicit programming model. As the STARS name suggests, this is made up of two parts, Scoped Types, and Aspects. First, Scoped Types ensure that the relative memory location of any object is obvious in the program text. We use nested packages to define a *static* scope hierarchy in the program's code; a pluggable type checker ensures programs respect this hierarchy; at runtime, the dynamic scope structure simply instantiates this static hierarchy. Second, we use Aspect-Oriented Programming to partially decouple the real-time parts of STARS programs from their application logic. Aspects are used as declarative specifications of the real-time policies of the applications (the size of scoped memory areas or scheduling parameters of real time threads), but also to link Scoped Types to their implementations within a real-time VM.

The main points of the STARS programming model are illustrated in Fig. 3. The main abstraction is the *scoped package*. A scoped package is the static manifestation of an RTSJ scoped memory area. Classes defined within a scoped package are either *gates* or *scoped classes*. Every instance of a gate class has its own unique scoped memory area, and every instance of a scoped class will be allocated in the memory area belonging to a gate object in the same package. Because gate classes can have multiple instances, each scoped package can correspond to multiple scoped memory areas at runtime (one for each gate instance), just as a Java class can correspond to multiple instances. Then, the dynamic structure of the nested memory areas is modeled by the static structure of the nested scoped packages, in just the same way that the dynamic structure of a program's objects is modeled by the static structure of the program's class diagram.

Scoped types are allowed to refer to types defined in an ancestor package, just as in RTSJ, objects allocated in a scope are allowed to refer to an ancestor scope: the converse is forbidden. The root of the hierarchy is the package `imm`, corresponding to RTSJ's immortal memory. There will be as many scoped memory areas nested inside the immortal memory area as there are instances of the gate classes defined in `imm`'s immediate subpackages.



```
package imm;

class B {...}
```

```
package imm.g;

class G extends Gate {...}

class C {...}
```

Fig. 3. *The STARS Programming Model. Each runtime scope has a corresponding Java package. Objects defined in a package are always allocated in a corresponding scope. A scope's gate is allocated in its parent scope.* `G` *is a gate class that extends a helper class* `Gate`*. (*`Gate` *is an abstract class described in Sec. 2.3) Just as with the RTSJ's scoped memory, references from a parent scope to a child scope are forbidden. Reference from scopes into the heap are also disallowed.*

STARS does impact the structure of Real-time Java programs. By giving an additional meaning to the `package` construct, we *de facto* extend the language. This form of overloading of language constructs has the same rationale as the definition of the RTSJ itself — namely to extend a language without changing its syntax, compiler, or intermediate format. In practice, STARS changes the way packages are used: rather than grouping classes on the basis of some logical criteria, we group them by lifetime and function. In our experience, this decomposition is natural as RTSJ programmers must think in terms of scopes and locations in their design. Thus it is not surprising to see that classes that end up allocated in the same scope are closely coupled, and so grouping them in the same package is not unrealistic. We argue that this package structure is a small price to pay for STARS' static guarantees, and for the clarity it brings to programs' real-time, memory dependent code.

2.1 Scoped Types: Static Constraints

The following Scoped Types rules ensure static correctness of STARS programs. In this rules, we assume that a scoped package contains exactly one *gate class* and zero or more scoped classes or interfaces (the *scoped types*). By convention, the gate is named with the package's name with the first letter capitalized. The descendant relation on packages is a partial order on packages defined by package nesting. The distinguished package `imm` is the root of the scope hierarchy. In the following we use `S` and `G` to denote respectively scoped and gate types, we use `C` to refer to any class. We use `p` to refer to the fully qualified name of a package. We refer to types not defined in a scoped package as *heap types*.

Rule 1 (Scoped Types).

1. *Any direct or indirect subpackages of* `imm` *are scoped.*
2. *Any type not defined in a scoped package is a heap type.*
3. *The type of a gate class* `p.G` *defined within a scoped package* `p` *is a gate type.*
4. *The type of any non-gate interface or class* `p.S` *defined within a scoped package* `p` *is a scoped type. The type of an array with elements of scoped type is a scoped type.*

Rule 2 (Visibility).

1. *An expression of scoped type* `p.S` *is visible in any type defined in* `p` *or any of its subpackages.*
2. *An expression of gate type* `p.G` *is visible in any type defined in the immediate super-package of* `p`*. An exception to this rule is the local variable* `this` *which can be used within a gate class.*
3. *An expression of a type defined in* `imm` *is visible in any types.*
4. *An expression of heap type is only visible in other heap types.*

The visibility rule encodes the essence of the RTSJ access rules. An object can be referenced from its defining memory area (denoted statically by a package), or from a memory area with shorter lifetime (a nested package). Gate classes are treated differently, as they are handles used from a parent scope to access a memory area. They must only be accessible to the code defined in the parent scope. The reason other types in the same scope package cannot refer to a gate is that we must avoid confusion between gates of the same type; a parent can instantiate many gates of the same type and the contents of these gates must be kept separate. Even though a gate's type is not visible in its own class, a single exception is made so that a gate object can refer to itself through the `this` pointer (because we know which gate "`this`" is).

Rule 3 (Widening). *An expression of a scoped type* `p.S` *can be widened only to another scoped type in* `p`*. An expression of a gate type* `p.G` *cannot be widened to any other types.*

Rule 3 is traditional in confined type systems where types are used to enforce structural properties on the object graph. Preventing types from being be cast to arbitrary supertypes (in particular `Object`) makes it possible to verify Rule 2 statically.

Rule 4 (Method Inheritance). *An invocation of some method* `m` *on an expression of scoped type* `p.S` *where* `p` *is a scoped package is valid if* `m` *is defined in a class* `p.S′` *in the same package. An invocation of a method* `m` *on an expression of gate type* `p.G` *is valid only if* `m` *is defined in* `p.G`*.*

Rule 4 prevents a more subtle form of reference leak: within an inherited method, the receiver (i.e. `this`) is implicitly cast to the method's defining class — this could lead to a leak if one were to invoke a method inherited from a heap class.

Rule 5 (Constructor Invocation). *The constructor of a scoped class* `p.S` *can only be invoked by methods defined in* `p`*.*

Rule 5 prevents a subpackage from invoking `new` on a class that is allocated in a different area than the currently executing object. This rule is not strictly necessary, as an implementation could potentially reflect upon the static type of the object to dynamically obtain the proper scope. In our prototype, we use factory methods to create objects.

Rule 6 (Static Reference Fields). *A type* `p.S` *defined in a scoped package* `p` *is not allow to declare static reference fields.*

A static variable would be accessible by different instances of the same class allocated in different scopes.

Correctness The fact that a package can only have one parent package trivially ensures that the RTSJ single parent rule will hold. Moreover, a scope-allocated object o may only reference objects allocated in the scope of o, or scopes with a longer lifetime, preventing any RTSJ `IllegalAssignmentError`. For example, suppose that the assignment `o.f=o′` is in the scope `s`, where `o` and `o′` have types `p.C` and `p′.C′` respectively. If `p.C` is a scoped type, then the rules above ensure that `o` and `o′` can only be allocated in `s` or its outer scopes. By Rules 2 and 3, the type of the field `f` is defined in `p′`, which is visible to `p.C`. Thus, the package `p′` is the same as or a super-package of `p` and consequently `o′` must be allocated in the scope of `o` or its outer scope. The same is true if `p.C` is a gate type, in which case `o` either represents `s` or a direct descendant of `s`. A formal soundness argument can be found in Sec. 5.

2.2 Specification files

It is not always convenient to tie scoped types to the package structure. Sometimes, for software engineering reasons, it may be necessary to treat class defined in another package *as if* they were defined in a scoped package. This can be done by simply adding a specification file, which map classes to their corresponding gates. The format of this file is a sequence of predicates `scopedIn(className,packageName)`.

```
declare scopedIn(StateTable, imm.runner);
declare scopedIn(Vector3d, imm.runner);
```

Fig. 4. *A specification file in JavaCop notation could declare that some classes, e.g.* `Vector3d` *and* `StateTable`*, are to be treated as scoped within a package, e.g.* `imm.runner`.

Fig. 4 gives an example of a specification file. For all practical purposes, the classes are type checked as part of the package they are declared to be scoped in. This means they have to obey to all the type rules of the scoped type system. In order to maintain correctness of the approach a class can only be be bound to one package in a given application. If this was not the case, the type system would not be able to prevent references from leaking from one context to the other.

One advantage of using specification files is that one class can be used in different scopes in different applications. On the other hand, there is a certain loss of clarity in the code as the allocation context of the class is not immediately visible.

2.3 Using Aspects

Though the design of memory management in a real-time system may be clear, typically, its implementation will be unclear, because it is inherently tangled throughout the code. For this reason we propose an aspect-oriented approach for modularizing scope management. This part of STARS is implemented using a subset of the Aspect-Oriented Programming features provided by AspectJ [25].[1] After a program has been statically verified, aspects are composed with the Java base-level application. The aspects weave necessary elements of the RTSJ API into the system. This translation depends upon the program following the Scoped Type discipline: if the rules are broken, the resulting program will no longer obey the RTSJ scoped memory discipline, and then either fail

[1] For performance, predictability and safety reasons we recommend users to stay away from dynamic features such as *cflow* and features that require instance-based aspect instantiation such as *perthis* and *pertarget*.

at runtime with just the kind of an exception we aim to prevent; or worse, if running on a virtual machine that omits runtime checks, fail in some unchecked manner.

In this paper, we focus on the interplay between threads and memory and show that with a simple API, it is possible to isolate many common real-time idioms. Fig. 5 shows the key features of the STARS interface. The package `scope` contains two classes. Class `STARS` provides a static method `waitForNextPeriod()` and an instance method `runInThread()`. The `STARS` subclass `Gate` has a private field that holds a reference to a memory area. This is field is not accessible to user code and it's use will be detailed next. The annotation `@WidenScoped` is described in Sec. 3.3.

```java
package scope;

public abstract class STARS {
  static public boolean waitForNextPeriod() { ... }
  public @WidenScoped void runInThread(Runnable r) {}
  ...
}

public abstract class Gate extends STARS {
  private MemoryArea mem;
}
```

Fig. 5. *STARS Interface. The* `scope` *package contains two classes,* `STARS` *and* `Gate`*, and an abstract aspect* `ScopedAspect`*. Every gate class inherits from* `Gate` *and has access to two methods* `waitForNextPeriod()` *and* `runInThread()`.

The API is intentionally simple. Gate classes must extend `scope.Gate`, which gives access to two methods: `waitForNextPeriod()` is used to block a thread until its next release event. This is only meaningful for periodic tasks. The second method, `runInThread(Runnable)`, is used to start a new real-time thread. The single argument of `runInThread` is an instance of class that implements the `Runnable` interface. The method has no behavior. The intended behavior, namely executing the `run()` method of the argument in a new real-time thread, will be implemented in an aspect. The real-time properties of the thread are left unbound in the Java-level code and will be specified at the aspect level.

The STARS API includes a further component, a set of predefined AspectJ pointcut. Fig. 6 shows the main pointcuts: `NewGate(g)` which corresponds to the creation of a gate object `g`, `GateCall(g)` which corresponds to a call of a method of a subclass of gate (excluding methods inherited from STARS), and `RunInThread(r,g)` which describes an invocation of the STARS method `runInThread()` where the receiver is `g` and the logic is `r`.

Associating Memory Areas with Gates: Specifying memory area parameters is done by declaring an `after` advice to the initialization of a newly allocated gate. This must be done in a `privileged` aspect to allow access to the private `Gate.mem` field. The advantage of using a privileged aspect is that the field need not be made `public`, this is important as it prevents Java-level code from modifying or even reading that field. As the type of `mem` is the abstract class `ScopedMemory`, the advice must specify one of its subclasses, `LTMemory` or `VTMemory`, to provide linear or variable time allocation of objects. At the same time, the advice must select an initial and maximal size for the area. The following code fragment is an example of an advice:

```
after(Gate g): NewGate(g) && execution(MyGate.new(..)){
  g.mem = new LTMemory( size );
}
```

This is an advice for an application class called `MyGate`. The developer has chosen a `LTMemory` area and has specified a maximum size for the area.

The code can get more involved when `SizeEstimator`s are used to determine the proper size of the area. The following code fragment, Fig. 7, shows an advice that creates two instances of the `SizeEstimator` class. The `initial` estimator reserves space for 100 instances of `Vector3d` and 20 instances of `Aircraft`. The second estimator is used for the maximum size of the memory area. In this example, `LTMemory` is used to ensure linear time allocation.

Creating Threads: With STARS, threads are weaved in by aspects. The Java-level code does not need to include thread creation statements or calls to related RTSJ APIs. Instead the Java-level code will have a call to `runInThread(r)` where `r` is a runnable which contains a `run()` method. This runnable is the logic that will be run by the real-time thread.

As before, we use an advice, in this case an `around` advice, to weave the thread creation code. The example of Fig. 8 shows how to bind a `RealtimeThread` to

```
privileged abstract aspect ScopedAspect {
  abstract pointcut InScope();
  pointcut NewGate(Gate g) : execution(Gate+.new(..))
                             && target(g)
                             && InScope();
  pointcut GateCall(Gate g) :
                           execution(public void Gate+.*(..))
                           && this(g);
  pointcut RunInThread(Runnable r, STARS g) :
                        execution(void STARS+.runInThread(..))
                        && target(g)
                        && args(r);
  ...
}
```

Fig. 6. *STARS Interface. Every STARS aspect extends* `ScopedAspect`*, must define pointcut* `InScope` *and has access to a number of predefined pointcuts.*

```
after(Gate g): NewGate(g) && execution(MyGate.new(..)){
  SizeEstimator initial = new SizeEstimator();
  inital.reserve(Vector3d.class, 100);
  inital.reserve(Aircraft.class, 20);
  SizeEstimator maximum = new SizeEstimator( initial, 3);
  g.mem = new VTMemory( initial, maximum );
}
```

Fig. 7. *This advice binds a* `RealTimeThread` *to each invocation of* `runInThread`.

a particular call of `runInThread`. The arguments to the advice are a gate `g` and a runnable `r`. The code creates several objects, instances of `PriorityParameters`, `RelativeTime`, and `PeriodicParameters`, which are used to configure the `RealtimeThread`. When the thread is started, it will run in the memory area denoted by `g.mem` and execute the `run()` method of `r`.

```
void around(STARS g, Runnable r): RunInThread(r, g){
   PriorityParameters priority =
      new PriorityParameters(PRIORITY);
   RelativeTime time = new RelativeTime(PERIOD, 0);
   PeriodicParameters period =
      new PeriodicParameters(null, time, null, null, null);
   Thread t = new RealtimeThread(priority, period,
          null, ((Gate) g).mem, null, r);
   t.start();
}
```

Fig. 8. *This advice binds a* `RealTimeThread` *to each invocation of* `runInThread`.

We now show an example, Fig. 9, of Java-level code that can be advised by the STARS aspect of Fig. 8 and Fig. 7. The following code fragment creates an instance of some application subclass of `Gate`, the class `MyGate` used in the example of Fig. 7. The advice of Fig. 7 will create the instance `VTMemory` after the creation of the instance of `MyGate`. The body of `runInThread()` (which is empty) will never be executed, instead the newly created runnable object will be passed in as argument to the `RealtimeThread` created in Fig. 8. The behavior of this simple program is to periodically print a message on the console. The program will terminate with an out of memory exception when the `VTMemory` area fills up with `StringBuffer` and `String` objects allocated as a side effect of string concatenation.

Periodic real-time threads are not the only way to arrange for the execution of real-time activities in the RTSJ. Fig. 10 shows how to bind a runnable to an an asynchronous event handler. This is done, again, by an around advice attached to calls to the `runInThread()` method of a `Gate`. There is a difference though, in the case

```
Gate gate = new MyGate();
gate.runInThread(new Runnable() {
   public void run() {
       while(true) {
          println("iteration " + i++);
          STARS.waitForNextPeriod();
       }
});
...
```

Fig. 9. *Application example.*

of an event handler the Java-level code should not use the `waitForNextPeriod()` method. The way to deals with this is to use AspectJ to report calls to the method as compile-time errors as shown in Fig. 10.

```
void around(STARS g, Runnable r): RunInThread(r, g){
  final Runnable run = r;
  BoundAsyncEventHandler handler =
     new BoundAsyncEventHandler(priority, release, null,
        ((Gate)g).mem, null, false, r)
     {
        Runnable logic = run;
        public void handleAsyncEvent() {
           logic.run();
        }
     };
   PeriodicTimer timer =
      new PeriodicTimer(start, period, handler);
}

declare error : call(STARS.waitForNextPeriod());
```

Fig. 10. *This advices binds* `runInThread` *calls to an asynchronous event handler and turns calls to* `waitForNextPeriod` *into compile-time errors.*

Memory and threading are not the only real-time concerns that can be modularized with aspects. Tsang, Clarke and Baniassad have looked at some of the other uses of AspectJ in RTSJ programs [36]. Furthermore, calls to the RTSJ API are not the only parts of a system relevant from a real-time point of view. For instance, the choice of data structures or algorithm can be crucial for the timing properties of system. An example of such use of aspects can be found in the work of Wang et al. [37].

3 The STARS Prototype Implementation

The STARS prototype has two software components — a checker, which takes plain Java code that is supposed to conform to the Scoped Types discipline, and verifies that it does in fact follow the discipline, and an series of AspectJ aspects that weaves in the necessary low-level API calls to run on a real-time virtual machine.

3.1 Checking the Scoped Types Discipline

We must ensure that only programs that follow the scoped types discipline are accepted by the system: this is why we begin by passing our programs through a checker that enforces the discipline. Rather than implement a checker from scratch, we have employed the JAVACOP "pluggable types" checker [1]. Pluggable types [13] are a relatively recent idea, developed as extensions of soft type systems [15] or as a generalization of the ideas behind the Strongtalk type system [14]. The key idea is that pluggable types layer a new static type system over an existing (statically or dynamically typed) language, allowing programmers to have greater guarantees about their programs' behaviour, but without the expense of implementing entirely new type systems or programming languages. JAVACOP is a pluggable type checker for Java programs — using JAVACOP, pluggable type systems are designed by a series of syntax-directed rules that are layered on top of the standard Java syntax and type system and then checked when the program is compiled. STARS is a pluggable type system, and so it is relatively straightforward to check with JAVACOP. The design and implementation of JAVACOP is described in [1].

The JAVACOP specification of the Scoped Type discipline is approximately 300 lines of code. Essentially, we provide two kinds of facts to JAVACOP to describe Scoped Types. First we define which classes must be considered scoped or gate types; and then we restrict the code of those classes according to the Scoped Type rules.

Defining Scoped Types is relatively easy. Any class declared within the `imm` package or any subpackage is either a scoped type or a gate. Declaring a scoped type in the JAVACOP rule language is straightforward: a class or interface is scoped if it is in a scoped package and is not a gate. A gate is a class declared within a scoped package and with a name that case-insensitively matches that of the package. Array types are handled separately: an array is scoped if its element types are scoped.

```
declare gateNamed(ClassSymbol s){
  require(s.packge.name.equalsIgnoreCase(s.name));
}
declare scoped(Type t){
  require(!t.isArray);
  require(!gateNamed(t.getSymbol));
  require(scopedPackage(t.getSymbol.packge));
}
declare scoped(Type t){
  require(t.isArray && scoped(t.elemtype));
}
declare gate(Type t){
  require(!t.isArray);
  require(gateNamed(t.getSymbol));
  require(scopedPackage(t.getSymbol.packge));
}
```

The rule that enforces visibility constraints is only slightly more complex. The following rule matches on a class definition (line 1) and ensure that all types of all syntax tree nodes found within that definition (line 2) meet the constraints of Scoped Types. A number of types and syntactic contexts, such as Strings and inheritance declarations, are deemed “safe” (`safeNodes` on line 3, definition omitted) and can be used in any context. Lines 4-5 ensure that top level gates are only visible in the heap. Lines 7-8 ensure that a gate is only visible in its parent package. Lines 10-11 ensure that the visibility of a scoped type is limited to its defining package and subpackages. Lines 13-16 apply if `c` is defined within a scoped package and ensure that types used within a scoped package are visible.

```
1  rule scopedTypesVisibilityDefn1(ClassDef c){
2   forall(Tree t : c){
3    where(t.type != null && !safeNode(t)){
4     where(topLevelGate(t.type)){
5      require(!scopedPackage(c.sym.packge)):
6       warning(t,"Top level gate visible only in heap"); }
7     where(innerGate(t.type)){
8      require(t.type.getSymbol.packge.owner == c.sym.packge):
9       warning(t,"gate visible only in immediate superpackage"); }
10    where(scoped(t.type)){
11     require(t.type.getSymbol.packge.isTransOwner(c.sym.packge)):
12      warning(t,"type visible only in same or subpackage"); }
13    where(scoped(c.sym.type)){
14     require(scopedPackage(t.type.getSymbol.packge) ||
15         specialPackage(t.type.getSymbol.packge)    ||
16         visibleInScopedOverride(t)):
17       warning(t,"Type not visible in scoped package."); }
18   }
19  }
20 }
```

We restrict widening of scoped types with the following rule. It states that if we are trying to widen a scoped type, then the target must be declared in the same scoped package, and if the type is a gate widening disallowed altogether. The `safeWideningLocation` predicate is an escape hatch that allows annotations that override the default rules.

```
rule scopedTypesCastingDef2(a <: b @ pos){
 where(!safeWideningLocation(pos)){
  where(scoped(a)){
    require(a.getSymbol.packge == b.getSymbol.packge) :
     warning(pos,"Illegal scoped type widening."); }
  where(gate(a)){
     require(b.isSameType(a)) :
       warning(pos,"May not widen gate."); }
 }
}
```

JAVACOP allows users to extend the Scoped Types specification with additional restrictions. It is thus possible to use JAVACOP to restrict the set of allowed programs further. The prototype implementation has one restriction, though, it does not support AspectJ syntax. JAVACOP is thus not able to validate the implementation of aspects. As long as aspects remain simple and declarative, this will not be a problem.

3.2 Instrumentation and Virtual Machine Support

The implementation of STARS relies on a small number of changes to a real-time Java virtual machine. In our case, we needed only add 18 lines to the Ovm framework [2] and 105 of lines of AspectJ to provide the needed functionality.

The added functionality consists of the addition of three new methods to the abstract class `MemoryArea`. These methods expose different parts of the implementation of the `MemoryArea.enter()`. The `STARSenter()` method increments the reference count associated to the area, changes allocation context and returns an opaque reference to the VM's representation of the allocation context before the change. `STARSexit()` leaves a memory area, possibly reclaiming its contents and restores the previous allocation context passed in as argument. `STARSrethrow()` is used to leave a memory area with an exception. Three methods of the class `LibraryImports` which mediates between the user domain and the VM's executive were made public. They are: `setCurrentArea()` to change the allocation context, `getCurrentArea()` to obtain the allocation context for the current thread, and `areaOf()` to obtain the area in which an object was allocated. All of these methods operate on opaque references.

```
Opaque MemoryArea.STARSenter();
void   MemoryArea.STARSrethrow(Opaque,Throwable);
void   MemoryArea.STARSexit(Opaque area);

static Opaque LibraryImports.setCurrentArea(Opaque area);
static Opaque LibraryImports.getCurrentArea();
static Opaque LibraryImports.areaOf(Object ref);
```

We show two key advices from the `ScopedAspect` introduced in Figure 5. The first advice executes before the instance initializer of any scoped class or array (lines 1-4). This advice obtains the area of `o` – which is the object performing the allocation – and sets the allocation context to that area. The reasoning is that if we are executing a `new` then the target class must be visible. We thus ensure that it is co-located.

```
1  before(Object o): AllocInScope(o) {
2   return LibraryImports
3           .setCurrentArea(LibraryImports.areaOf(o));
4  }
```

We use the second advice to modify the behaviour of any call to a gate (recall that these can only originate from the immediate parent package). This `around` advice uses the memory region field of the gate to change allocation context. When the method returns we restore the previous area.

```
void around(Gate g) : GateCall(g) {
 Opaque x = g.mem.STARSenter();
  try {
   try {
    proceed(g);
   } catch(Throwable e) { g.mem.STARSrethrow(x, e); }
  } finally { g.mem.STARSexit(x); }
 }
```

3.3 Extensions and Restrictions

We have found that, for practical reasons, a small numbers of adjustments needed to be made to the core of the scoped type system.

Intrinsics. Some important features of the standard Java libraries are presented as static methods on JDK classes. Invoking static methods from a scoped package, and especially ones that are not defined in the current package, is illegal. This is too restrictive and we relaxed the JavaCOP specification to allow calls to static methods in the following classes `System`, `Double`, `Float`, `Integer`, `Long`, `Math`, and `Number`. Moreover, we have chosen to permit the use of `java.lang.String` in scoped packages. Whether this is wise is debatable – for debugging purposes it is certainly useful to be able to construct messages, but it opens up an opportunity for runtime memory errors. It is conceivable that the JavaCOP rules will be tightened in the future to better track the use of scope allocated strings.

Exceptions. All subclasses of `Throwable` are allowed in a scoped package. This is safe within the confines of standard use of exceptions. If an exception is allocated and thrown within a scoped package, it is either caught by a handler within that package or escape out of the memory area. In which case it will be caught by the around advice at the gate boundary and `STARSrethrow` will allocate a RTSJ `ThrowBoundaryError` object in the parent scope and rethrow the newly allocated error. One drawback of this rule is that a memory error could occur if a programmer managed to return/assign a scope-allocated error object to a parent area. Luckily there is a simple solution that catches most reasonable use-cases. We define a JavaCOP rule that allows exceptions to be created only if they are within a `throw` statement.

```
declare treeVisInScoped(Tree t){
 require(NewClass n, Throw th;
       n <- env.tree && th<-env.next.tree){
  require(th.expr == n);
```

```
   require(t == n.clazz);
 }
}
```

Annotations. We found that in rare cases it may be necessary to let users override the scoped type system — typically where (library) code is clearly correct, but where it fails the conservative Scoped Types checker. For this we provide two Java 5 annotations that are recognized by the JavaCOP rules. `@WidenScoped` permits to declare that an expression which performs an otherwise illegal widening is deemed safe. `@MakeVisible` takes a type and makes it visible within a class or method.

Reflection. In the current implementation we assume that reflection is not used to manipulate scoped types. But a better solution would be to have reflection enforce the STARS semantics. This can be achieved by making the implementation of reflection scope-aware. Of course, whether reflection should be used in a hard real-time system, considering its impact on compiler analysis and optimization is open for discussion.

Native methods. Native methods are an issue for safety. This is nothing new, even normal Java virtual machines depend on the correctness of the implementation of native methods for type safety. We take the approach that native methods are disallowed unless explicitly permitted in a JavaCOP specification.

Finalizers. While the STARS prototype allows finalizers, we advocate that they should not be used in scoped packages. This because there is a well-known pathological case where a `NoHeapRealtimeThread` can end up blocking for the garbage collector due to the interplay of finalization and access to scope by `RealtimeThread`s. This constraint is not part of the basic set of JavaCOP rules. Instead we add it as a user-defined extension to the rule set. This is done by the following rule:

```
rule nofinalizers(MethodDef m){
 where(m.name.equals("finalize") && m.params.length == 0){
   require(ClassSymbol c; c <- m.sym.owner) {
    require(!scopedPackage(c.packge)):
      warning(m,"Scoped class may not define a finalizer");
   }
 }
}
```

4 Case Study: A Real-time Collision Detector

We conducted a case study to demonstrate the relative benefits of STARS. The software system used in this experiment is modeling a real-time *collision detector* (or CD). The collision detector algorithm is about 25K Loc and was originally written with the Real-time Specification for Java. As a proof-of-concept for our proposal, we refactored the CD to abide by the scoped type discipline and to use aspects.

The architecture of the STARS version of the CD is given in Fig. 11. The application has three threads, a plain Java thread running in the heap to generate simulated workloads, a 5Hz thread whose job is to communicate results of the algorithm to an output device and finally a 10Hz `NoHeapRealtimeThread` which periodically acquires a data frame with positions of aircrafts from simulated sensors. The system must detect collision before they happen. The numbers of planes, airports, and nature of flight restrictions are variables to the system.

The refactoring was done in three stages. First, we designed a scope structure for the program based on the `ScopedMemory` areas used in the CD. Second, we moved classes amongst packages so that the STARS-CD package structure matched the scope structure. Third, we removed or replaced explicit RTSJ memory management idioms with equivalent constructs of our model.

Fig. 12 compares the package structure of the two versions. In the original CD the packages `atc` and `command` were responsible of computing trajectories based on a user-defined specification. They were not affected by the refactoring. Package `detector` contained all of the RTSJ code as well the program's `main()`. Finally `util` contained a number of general purpose utility classes. We split the code in the `detector` package in four groups. The package `heap` contains code that runs in the heap–this is the main and the data reporting thread. The package `imm` contains classes that will be allocated in immortal memory and thus never reclaimed. Below immor-

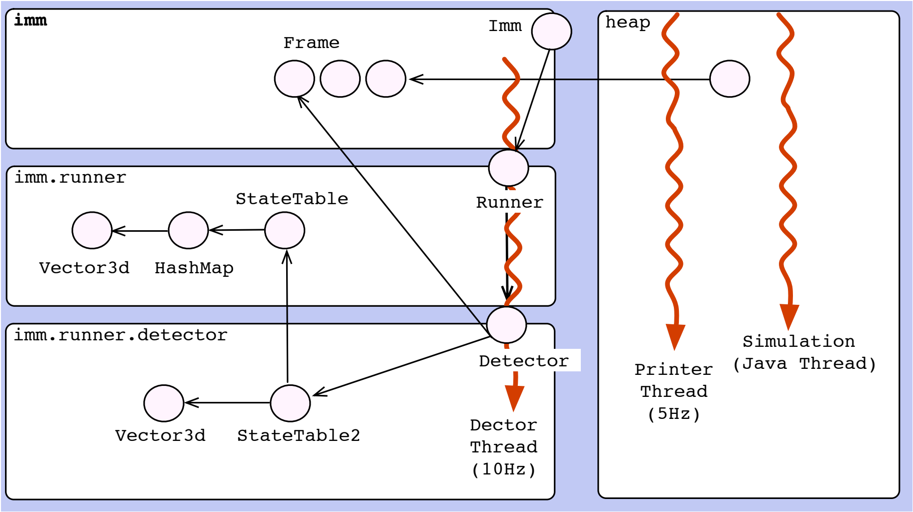


Fig. 11. *Collision Detector. The CD uses two scoped memory areas. Two threads run in the heap: the first simulates a workload, the second communicate with an output device. The memory hierarchy consists of* `imm` *(immortal memory) for the simulated workload,* `imm.runner` *for persistent data, and* `imm.runner.detector` *for frame specific data.*

tal memory there is one scope that contains the persistent state of the application, we defined a package `imm.runner` for this. The main computation is done in the last package, `imm.runner.detector`. This is the largest real-time package which contains classes that are allocated and reclaimed for each period.

The entire code of the real-time aspect for the CD is given in Fig. 13. This aspect simply declares the memory area types for the `imm.runner` and `imm.runner.detector` gates. Then it gives an around advice that specifies that the thread used by the CD algorithm is a `NoHeapRealtimeThread` and gives appropriate scheduling and priority parameters.

The overall size of the Scoped CD has increased because we had to duplicate some of the utility collection classes. This duplication is due to our static constraints. A number of collection classes were used in the `imm.runner` package to represent persistent state, and in the `imm.runner.detector` package to compute collisions. While we could have avoided the duplication by fairly simple changes to the algorithm and the use of problem specific collections, our goal was to look at the ‘worst-case’ scenario, so we tried to make as few changes to the original CD as possible. The methodology used to duplicate collection classes is straightforward: we define a scoped replacement for the `Object` class and replace all occurrences of `Object` in the libraries with the scoped variant. There were some other minor changes, but these were also fairly straightforward.

4.1 Patterns and Idioms

RTSJ programmers have adopted or developed a number of programming idioms to manipulate scopes. After changing the structure of the original CD, we need to convert these idioms into corresponding idioms that abide by our rules. In almost every case, the resulting code was simpler and more general, because it could directly manipulate standard Java objects rather than having to create and manage special RTSJ scope metaobjects explicitly.

CD packages	classes	**Scoped CD packages**	classes
	per package		per package
`atc`	989	`atc`	989
`command`	21198	`command`	21198
`util`	927	`util`	927
`detector`	1041		
		`heap`	105
		`imm`	120
		`imm.runner` (1)	162
		`imm.runner.detector` (5)	1587
		`collections` (22)	8322

Fig. 12. *Package structure of the CD (left) and the STARS CD (right). Number in parenthesis indicate the classes that had to be duplicated to abide by the scoped type constraints.*

```
privileged aspect CDAspect extends ScopedAspect{

 before(Gate g): NewGate(g) && execution(Runner.new(..)){
   g.mem = new LTMemory(Constants.SIZE*2,Constants.SIZE*2);
 }

 before(Gate g): NewGate(g) && execution(Detector.new(..)){
   g.mem = new LTMemory(Constants.SIZE);
 }

 void around(STARS g, Runnable r): RunInThread(r, g){
    Thread t = new NoHeapRealtimeThread(
        new PriorityParameters(Constants.PRIORITY),
        new PeriodicParameters(null,
                new RelativeTime(Constants.PERIOD, 0),
                null, null, null),
        null, ((Gate) g).mem, null, r);
    t.start();
 }
}
```

Fig. 13. *Real-time Aspect for the CD. The aspect specifies the characteristics of memory areas as well as that of the real-time thread used by the application. The CD logic does not refer to any of the RTSJ APIs.*

Scoped Run Loop. At the core of the CD is an instance of the ScopedRunLoop pattern identified in [32]. The `Runner` class creates a `Detector` and periodically executes the detector's `run()` method within a scope. Fig. 14 shows both the RTSJ version and the STARS version. In the RTSJ version, the runner is a `NoHeapRealtimeThread` which has in its `run()` method code to create a new scoped memory (lines 11-12) and a run loop which repeatedly enters the scope passing a detector as argument (lines 17-18).

In the STARS version, `Runner` and `Detector` are gates to nested packages. Thus the call to `run()` on line 16 will enter the memory area associated with the detector. Objects allocated while executing the method are allocated in this area. When the method returns these objects will be reclaimed. Fig. 15 illustrates how a `Runner` is started. In the RTSJ version a scoped memory area is explicitly created (lines 2-3) and the real-time arguments are provided (lines 6-11). In the STARS version most of this is implicit due to the fact that a runner is a gate and the use of the `runInThread()` method which is advised to create a new thread. What should be noted here is that, in this particular example, STARS separates the real-time support from the non-real-time code.

Multiscoped Object. A multiscoped object is an object which is used in several allocation contexts as defined in [32]. In the RTSJ CD the `StateTable` class keeps persistent state and is allocated in the area that is not reclaimed on each period. This table has one entry per plane holding the plane's call sign and its last known position. There is also a method `createMotions()` invoked from the transient scope. The class appears in Fig. 16.

This code is particularly tricky because the state table object is allocated in the persistent area and the method `createMotions()` is executed in the transient area

```
public class Runner extends
       NoHeapRealtimeThread {

 public Runner(
         PriorityParameters r,
         PeriodicParameters p,
         MemoryArea m) {
  super(r, p, m);
 }
 public void run() {
  final LTMemory cdmem =
   new LTMemory(CDSIZE,CDIZE);
  StateTable st =
   new StateTable();
  Detector cd =
   new Detector(st, SIZE);
   while (waitForNextPeriod())
    cdmem.enter(cd);
 }
}
```

```
public class Runner
       extends Gate {







 public void run() {
  StateTable st =
    new StateTable();
  Detector cd =
    new Detector(st, SIZE);
  while (waitForNextPeriod())
    cd.run();
 }
}
```

Fig. 14. *Scoped Run Loop Example. The Runner class: RTSJ version (on the left) and Scoped version (on the right).*

```
public void run() {
 LTMemory memory =
  new LTMemory(MSZ, MSZ);
 NoHeapRealtimeThread rt =
 new Runner(new PriorityParameters(P),
  new  PeriodicParameters(null,
   new RelativeTime(PER,0),
   new RelativeTime(5,0),
   new RelativeTime(50,0),
   null,null),
   memory);
 rt.start();
}
```

```
public void run() {
  Runner rt =
   new Runner();
  runInThread(rt);
}
```

Fig. 15. *Starting up. The* `imm.Imm.run()` *method: RTSJ version (left-hand side) and Scoped version (right-hand side).*

(when called by the `Detector`). The object referred to by `pos` (line 8) is transient and one must be careful not to store it in the parent scope. When a new plane is detected, `old` is null (line 11) and a new position vector must be added to the state table. The complication is that at that point the allocation context is that of the transient area, but the `HashMap` was allocated in the persistent scope (line 2). So we must temporarily change allocation context. This is done by defining an inner class whose sole purpose is to create a new vector and add it to the hash map (lines 23-39). The context switch is performed in lines 15-17 by first obtaining the area in which the StateTable was allocated, and finally executing the `Putter` in that area (line 17). This code is a good example of the intricacy of RTSJ programming.

```
 1 class StateTable {
 2  HashMap prev = new HashMap();
 3  Putter putter = new Putter();
 4 
 5  List createMotions(Frame f) {
 6   List ret = new LinkedList();
 7   for (...) {
 8    Vector3d pos = new Vector3d();
 9    Aircraft craft = iter.next(newpos);
10    ...
11    Vector3d old = (Vector3d) prev.get(craft);
12    if (old == null) {
13     putter.c = craft;
14     putter.v = pos;
15     MemoryArea current =
16       MemoryArea.getMemoryArea(this);
17     mem.executeInArea(putter);
18    }
19   }
20   return ret;
21  }
22 
23  class Putter implements Runnable {
24   Aircraft c;
25   Vector3d v;
26   public void run() {
27     prev.put(c, new Vector3d(v));
28   }
29  }
30 }
```

Fig. 16. *RTSJ StateTable. This is an example of a RTSJ multiscoped object – an instance of class allocated in one scope but with some of its methods executing in a child scope. Inspection of the code does not reveal in which scope* `createMotions()` *will be run. It is thus incumbent on the programmer to make sure that the method will behave correctly in any context.*

The scoped solution given in Fig. 17 makes things more explicit. The `StateTable` class is split in two. One class, `imm.runner.StateTable`, for persistent state and a second class, `imm.runner.detector.StateTable2` that has the update method. This split makes the allocation context explicit. A `StateTable2` has a reference to the persistent state table. The `createMotions()` method is split in two parts, one that runs in the transient area (lines 23-30) and the other that performs the update to the persistent data (lines 8-14).

Since our type system does not permit references to subpackages the arguments to `StateTable.put()` are primitive. The most displeasing aspect of the refactoring is that we had to duplicate the `Vector3d` class - there are now two identical versions - in

```
1  package imm.runner;
2  public class Vector3d { ... }
3
4  public class StateTable {
5   HashMap prev = new HashMap();
6
7   void put(Aircraft craft, float x, float y, float z) {
8    Vector3d old = prev.get(craft);
9    if (old==null)
10     prev.put(craft, new Vector3d(x, y, z));
11   else
12     old.set(x, y, z);
13  }
14 }
15
16 package imm.runner.detector;
17 class Vector3d { ... }
18
19 class StateTable2 {
20  StateTable table;
21
22  List createMotions(Frame f) {
23   List ret = new LinkedList();
24   for (...) {
25    Vector3d pos = new Vector3d();
26    ...
27    table.put(craft, pos.x, pos.y, pos.z);
28   }
29   return ret;
30  }
31 }
```

Fig. 17. *STARS StateTable. With scoped types the table is split in two. This makes the allocation context for data and methods explicit.*

each `imm.runner` and `imm.runner.detector`. We are considering extensions to the type system to remedy this situation.

4.2 Performance Evaluation

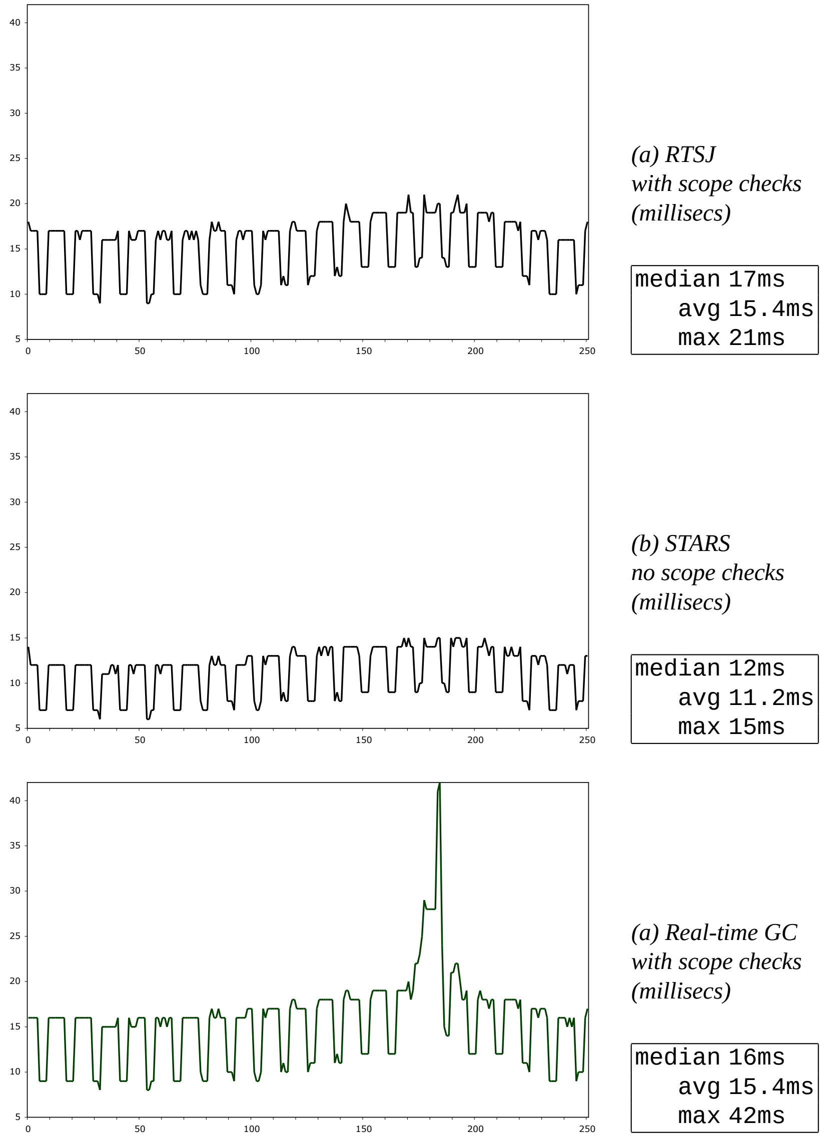
We now compare the performance of three versions of the CD: with the RTSJ, with STARS, and with a real-time garbage collector. The latter was obtained by ignoring the STARS annotations, with all objects allocated in the heap. The benchmarks were run on an AMD Athlon(TM) XP1900+ running at 1.6GHz, with 1GB of memory. The operating system is Real-time Linux with a kernel release number of 2.4.7- timesys-3.1.214. We rely on AspectJ 1.5 as our weaver. We use the Ovm virtual machine framework [5] with ahead-of-time compilation ("engine=j2c, build=run"). The GCC 4.0.1 compiler is used for native code generation. The STARS VM was built with dynamic read and write barriers turned off. The application consists of three threads, 10Hz, 5Hz, and plain Java. Priority preemptive scheduling is performed by the RTSJVM.

Fig. 18 shows the difference in running time between the three versions of the CD. The x-axis plots input frames while the x-axis gives the time the application took to process that frame of input. The per-frame processing time depends on the relative positions of the planes. Thus some frames are processed faster than others. For instance all graphs show a slight increase in processing time for frames 190-200, this is normal application behavior.

For each configuration of the virtual machine, we list median, average and maximum per-frame processing times. Minimum times are not meaningful as they are dominated by virtual machine-specific overheads. With Scoped Memory, the maximum processing time is 21 ms, considerably higher than with STARS which has only a 15 ms worst case time. The extra costs come from the read and write barriers that must be run by the virtual machine. Clearly these overheads are application specific – reference intensive computations will pay a higher price than computations that mostly manipulate primitive values.

The real-time GC number are interesting. The collector used in this experiment is a time-based collector modeled on the Metronome [3, 4]. The collector kicks in when the program's available memory is less than a specified threshold. Once the collector is running, it will interrupt the application for at most 1ms at a time. But, the processing of a frame can be interrupted numerous times. This explains the peak at 42 ms. During the processing of one frame, the application was interrupted every other ms by the collector thus doubling processing time. It is noteworthy that the average median processing time is higher with RTGC than with STARS, this is due to the additional barriers required by the collector.

The results suggest that STARS outperforms both RTSJ and Real-time GC. On average, STARS is about 28% faster per frame than RTSJ and RTGC. This means that the overhead of before advice attached to every allocation is negligible. This is only a single data point, we feel that more aggressive barrier elimination could reduce the overhead of RTSJ programs and that the performance of our RTGC is likely not yet optimal. Nevertheless, the data presented here suggested that there is a potentially significant performance benefit in adopting STARS.



(a) RTSJ
with scope checks
(millisecs)

median 17ms
avg 15.4ms
max 21ms

(b) STARS
no scope checks
(millisecs)

median 12ms
avg 11.2ms
max 15ms

(a) Real-time GC
with scope checks
(millisecs)

median 16ms
avg 15.4ms
max 42ms

Fig. 18. *Performance Evaluation. Comparing the performance of the collision detection implemented with (a) RTSJ, (b) STARS and (c) Java with a real-time garbage collector. We measure the time needed to process one frame of input by a 10Hz high-priority thread. The x-axis shows input frames and the y-axis processing time in milliseconds. The RTGC spikes at 43ms when the GC kicks in. No deadlines are missed. The average per frame processing time of STARS is 28% less than that of RTSJ and RTGC. Variations in processing time are due to the nature of the algorithm.*

5 The SJ calculus

To gain confidence in the programming model underlying our proposal, we introduce the SJ calculus, a sparse imperative and concurrent object calculus, modeled after Featherweight Java [24], in which scopes are first-class values. SJ formalizes the type confinement rules of Scoped Type in terms of a type system. Our proof of type soundness gives us the guarantee that confinement cannot be breached during execution of a well-typed program. We can then proceed to prove that the shape of the scope hierarchy is restricted to tree. And, finally, that deallocation of a scope will not result in dangling references. SJ is a simple object calculus, to keep the semantics concise we have omitted some features that are not essential to the main results. These features include static methods, synchronization, access modifiers, and down-cast expressions. Some specific features related to scoped memory such as the size and the type of the memory area (linear or variable allocation time) are also omitted.

5.1 Syntax and Types

The syntax of the SJ calculus, Figure 19, draws on our previous work [40]. The formalism and syntax is based on the Featherweight Java (FJ) system which has been widely adopted as a vehicle of language research. SJ has four kinds of class: *immortal classes*, *scoped classes*, *gate classes*, and *heap classes*. Classes belong to packages, which can be nested in an arbitrary package hierarchy. Immortal classes are in the package `imm`; mixtures of scoped and gate classes are in the *scoped packages*, which are subpackages of `imm`; and the heap classes are in any other packages. A class in a scoped package is a gate class if its short name matches the name of the package (case insensitive) and any other classes in the package are scoped classes. We add an assignment expression and an expression for creating a new thread of control.

We take metavariables $\mathtt{C}, \mathtt{D}$ to range over classes, $\mathtt{M}$ to range over methods, $\mathtt{K}$ over constructors, and $\mathtt{f}$ and $\mathtt{x}$ to range over fields and variables (including parameters and the pseudo variable `this`), respectively. We also use $\mathtt{P}$ for package names, $\mathtt{e}$ for expressions and ℓ for memory references. We use over-bar to represent a finite ordered sequence, for instance, $\overline{\mathtt{f}}$ represents $\mathtt{f}_1\,\mathtt{f}_2\,\ldots\,\mathtt{f}_\mathtt{n}$. The term $\overline{\mathtt{l}}\,.\,\overline{\mathtt{l}'}$ denotes sequence concatenation. The calculus has a call-by-value semantics. The expression $[{}^{\mathtt{v}}/_{\mathtt{v}_\mathtt{i}}]\overline{\mathtt{v}}$ yields a sequence identical to $\overline{\mathtt{v}}$ except in the ith field which is set to $\mathtt{v}$. We use the usual dot notation to represent nested packages. That is, the package `p.q` is a subpackage of `p`.

$$\mathtt{L} ::= \mathtt{class}\;\mathtt{P.C} \;\triangleleft\; \mathtt{D}\,\{\,\overline{\mathtt{C}}\,\overline{\mathtt{f}};\,\overline{\mathtt{M}}\,\}$$

$$\mathtt{M} ::= \mathtt{C}\;\mathtt{m}(\overline{\mathtt{C}}\,\overline{\mathtt{x}})\,\{\,\mathtt{return}\;\mathtt{e};\;\}$$

$$\mathtt{e} ::= \mathtt{x} \;\mid\; \mathtt{e.f} \;\mid\; \mathtt{e.m}(\overline{\mathtt{e}}) \;\mid\; \mathtt{new}\;\mathtt{C}() \;\mid\; \mathtt{e.f} := \mathtt{e} \;\mid\; \mathtt{spawn}\;\mathtt{e} \;\mid\; \mathtt{v}$$

$$\mathtt{P} ::= \mathtt{imm.P}' \;\mid\; \mathtt{P}' \qquad \mathtt{P}' ::= \mathtt{p} \;\mid\; \mathtt{p.P}' \qquad \mathtt{v} ::= \ell \;\mid\; \mathtt{null}$$

Fig. 19. *Syntax of the SJ calculus.*

Allocation Scope:

Given $\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}),\ \mathit{allocScope}_\sigma(\ell) = \ell$ if $\mathtt{C}$ is a gate, $\mathit{allocScope}_\sigma(\ell) = \ell'$ otherwise.

Deallocation:

$$\mathit{deallocate}(\sigma, \ell_0) = \{(\ell, \mathtt{C}^{\ell'}(\overline{\mathtt{v}})) \in \sigma \ \mid\ \ell' \neq \ell_0\}$$

Evaluation context:

$$\begin{array}{c} E[\circ] ::= \circ \ \mid\ E[\circ].\mathtt{m}(\overline{\mathtt{e}}) \ \mid\ \mathtt{v}.\mathtt{m}(\ldots, \mathtt{v}_{\mathtt{i}-1}, E[\circ], \mathtt{e}_{\mathtt{i}+1} \ldots) \\ \mid\ \ E[\circ].\mathtt{f}_\mathtt{i} \ \mid\ E[\circ].\mathtt{f}_\mathtt{i} := \mathtt{e} \ \mid\ \mathtt{v}.\mathtt{f}_\mathtt{i} := E[\circ] \end{array}$$

Scope reference counts:

$$\begin{array}{c} \mathit{refcount}(\ell, t[\,\overline{\ell\,\mathtt{e}}\,] \mid P') = \mathit{count}_\ell(\bar{\ell}) + \mathit{refcount}(\ell, P') \\ \mathit{refcount}(\ell, \emptyset) = 0 \qquad \mathit{count}_\ell(\emptyset) = 0 \\ \mathit{count}_\ell(\bar{\ell}\,.\,\ell) = 1 + \mathit{count}_\ell(\bar{\ell}) \qquad \mathit{count}_\ell(\bar{\ell}\,.\,\ell') = \mathit{count}_\ell(\bar{\ell}) \end{array}$$

Fig. 20. *Auxiliary definitions.*

$$\frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\,\overline{\mathtt{f}})}{\sigma, \ell_0\ \ell.\mathtt{f}_\mathtt{i} \ \longrightarrow\ \sigma, \ell_0\ \mathtt{v}_\mathtt{i}} \qquad \text{(R-FIELD)}$$

$$\frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\,\overline{\mathtt{f}}) \quad \sigma' = \sigma[\ell \mapsto \mathtt{C}^{\ell'}([{}^{\mathtt{v}}/_{\mathtt{v}_\mathtt{i}}\overline{\mathtt{v}}])]}{\sigma, \ell_0\ \ell.\mathtt{f}_\mathtt{i} := \mathtt{v} \ \longrightarrow\ \sigma', \ell_0\ \mathtt{v}} \ \text{(R-UPDATE)}$$

$$\frac{\begin{array}{c}\text{if } \mathtt{C} \text{ is immortal type, } \ell' = \ell_{\mathtt{imm}},\ \text{else if it is heap type, } \ell' = \ell_{\mathtt{heap}} \\ \text{else } \ell' = \mathit{allocScope}_\sigma(\ell_0) \quad \ell \text{ fresh} \quad \sigma' = \sigma[\ell \mapsto \mathtt{C}^{\ell'}(\overline{\mathtt{null}})]\end{array}}{\sigma, \ell_0\ \mathtt{new}\ \mathtt{C}() \ \longrightarrow\ \sigma', \ell_0\ \ell} \qquad \text{(R-NEW)}$$

$$\frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}') \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}{\sigma, \ell_0\ \ell.\mathtt{m}(\overline{\mathtt{v}}) \ \longrightarrow\ \sigma, \ell\ [{}^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}},\ {}^{\ell}/_{\mathtt{this}}]\mathtt{e}} \qquad \text{(R-INVK)}$$

Fig. 21. *Expression evaluation.*

The presentation of the calculus inherits some of the syntactic oddities of FJ, so $\overline{\mathtt{e}}\,\mathtt{e}$ is a short hand for $\mathtt{e}_1\ \ldots \mathtt{e}_\mathtt{n}\,\mathtt{e}$, and $\mathtt{m}(\overline{\mathtt{C}}\,\overline{\mathtt{x}})$ stands for $\mathtt{m}(\mathtt{C}_1\,\mathtt{x}_1, \ldots, \mathtt{C}_\mathtt{n}\,\mathtt{x}_\mathtt{n})$.

5.2 Semantics

In SJ, each gate object represents a distinct scoped memory area and when no thread is executing methods in the gate object, all of the objects that were allocated within the

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]\,] \quad P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}']\,] \\ \mathtt{e} \neq \ell'.\mathtt{m}(\overline{\mathtt{v}}) \quad \sigma, \ell\ \mathtt{e}\ \rightarrow\ \sigma', \ell\ \mathtt{e}' \end{array}}{\sigma, P \Rightarrow \sigma', P'} \qquad (\text{G-S}\textsc{tep})$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]\,] \quad P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]\,.\,\ell'\,\mathtt{e}'\,] \\ \mathtt{e} = \ell'.\mathtt{m}(\overline{\mathtt{v}}) \quad \sigma, \ell\ \mathtt{e}\ \ \rightarrow\ \sigma, \ell'\ \mathtt{e}' \end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad (\text{G-E}\textsc{nter})$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]\,.\,\ell'\,\mathtt{v}\,] \quad P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{v}]\,] \quad \sigma(\ell') = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}}) \\ \text{if } \mathtt{C} \text{ is not a gate or } \mathit{refcount}(\ell', P') \neq 0, \text{ then } \sigma' = \sigma, \\ \text{else } \sigma'' = \mathit{deallocate}(\sigma, \ell') \quad \sigma' = \sigma''[\ell' \mapsto \mathtt{C}^{\ell_0}(\overline{\mathtt{null}})] \end{array}}{\sigma, P \Rightarrow \sigma', P'} \ (\text{G-R}\textsc{eturn})$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,] \quad P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\ell^{\mathtt{th}}]\,] \ \mid\ t'[\,\overline{\ell\,\ell^{\mathtt{th}}}\,.\,\ell\,\mathtt{e}'\,] \\ \mathtt{e} = E[\mathtt{spawn}\,\mathtt{e}'] \qquad t' \text{ fresh} \end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad (\text{G-S}\textsc{pawn})$$

Fig. 22. *Computation rules.*

associated scope are reclaimed. While the package hierarchy imposes a static structure on scopes, gate objects allow multiple scope instances to be created at runtime. The main restriction imposed by SJ is that a gate can only allocate objects of scoped classes belonging to the same package and gates defined in immediate subpackages. When this restriction is combined with confinement invariants that prevents gate objects leaking from their parent package, we obtain the key property for scoped memory management, namely the restriction that threads enter scopes in the same order as the nesting relation of the packages containing the gate classes.

As in Featherweight Java, the semantics assumes the existence of a class table containing the definitions of all classes. We had to add a store σ and a collection of threads $\overline{P}$ labeled by distinct identifiers $\overline{t}$. Objects are of the form $\mathtt{C}^{\ell}(\overline{\mathtt{v}})$, where $\mathtt{C}$ is a class, $\overline{\mathtt{v}}$ are the values of the fields, and ℓ is the object representing the memory scope in which it was allocated. The store σ is a sequence, $\overline{\mathtt{C}^{\ell}(\overline{\mathtt{v}})}$, of objects, each denoted by a distinct label $\ell_{\mathtt{i}}$. We assume that σ contains an object $\ell_{\mathtt{heap}}$ representing heap memory and an object $\ell_{\mathtt{imm}}$ representing immortal memory. Fig. 20 defines a number of auxiliaries relations. The partial function $\mathit{allocScope}_{\sigma}(\ell)$ retrieves the allocation scope when the current receiver object is ℓ. That is, the allocation scope is ℓ if $\sigma(\ell)$ is a gate object, and otherwise, it is the scope where $\sigma(\ell)$ is allocated. The function $\mathit{deallocate}(\sigma, \ell_0)$ remove objects allocated in the scope corresponding to ℓ_0 from the store σ. An evaluation context (Fig. 20) is an expression $E[\circ]$ with a hole that can be filled in with another expression of proper type. An expression $\mathtt{e}_0$ is written as $E[\mathtt{e}]$ only if $\mathtt{e}$ is in the form of $\mathtt{v}$, $\mathtt{v.f}$, $\mathtt{v.f} := \mathtt{v}'$, $\mathtt{v.m}(\overline{\mathtt{v}})$, or $\mathtt{spawn}\,\mathtt{e}$. For a non-value expression $\mathtt{e}_0$, there exists an unique evaluation context $E[\circ]$ such that $\mathtt{e}_0 = E[\mathtt{e}]$ and $\mathtt{e}$ is not a value. Evaluation contexts do not include the body of `spawn`. The partial function $\mathit{refcount}(\ell, P)$ returns the number of times that the threads of program P have entered the scope corresponding to the gate object ℓ.

The dynamics semantics of SJ is split in two: expression evaluation rules given in Fig. 21 and the computation rules in Fig. 22.

Expression rules. These evaluation rules consider only operations performed within a single thread. The evaluation relation has the form $\sigma, \ell\ \texttt{e}\ \rightarrow\ \sigma', \ell'\ \texttt{e}'$ where σ is the initial store, ℓ is the reference to object currently executing, and $\texttt{e}$ the expression to evaluate. The reduction rules field select (R-FIELD), field update (R-UPDATE), and method invocation (R-INVK) are not surprising, where $\mathit{mbody}(\texttt{m}, \texttt{C})$ returns parameters $\overline{\texttt{x}}$ and method body $\texttt{e}$ of $\texttt{m}$ when it is invoked on an object of the type $\texttt{C}$. The instantiation rule (R-NEW) finds the right scope to allocate the object about to be created. If the new object type $\texttt{C}$ is heap type, then the scope is $\ell_{\texttt{heap}}$, if $\texttt{C}$ is immortal type, then the scope is $\ell_{\texttt{imm}}$, otherwise, the allocation scope is the current scope of the thread. The fields of $\texttt{C}$ must be initialized to null. Finally the store is updated with a fresh reference ℓ bound to the newly allocated object. The helper function $\mathit{init}(\texttt{C})$ is defined in Figure 25.

Computation rules. The computation rules are of the form $\sigma, P \Rightarrow \sigma', P'$ where σ is a store and P is a set of threads. Each thread $t[\,\overline{\ell\,\texttt{e}}\,]$ in P has a distinct label t and a runtime call stack which is a list of receiver-expression pairs $\ell, \texttt{e}$.

Rule (G-STEP) is simple, it picks one thread for execution and evaluates the expression $E[\texttt{e}]$ on the top of the thread's stack. Note that this rule applies when $\texttt{e}$ is not a method invocation.

Rule (G-ENTER) evaluates a thread $t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,]$ containing a method call $\texttt{e} = E[\texttt{e}_0]$ and $\texttt{e}_0 = \ell'.\texttt{m}(\overline{\texttt{v}})$. It creates a new stack frame for the body of the method, $\texttt{e}'$, and the result is a stack $\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}_0]\,.\,\ell'\,\texttt{e}'$.

If a thread's stack has the form $\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}]\,.\,\ell'\,\texttt{v}$, then by Rule (G-RETURN), the thread can pop the stack frame and continue execution with $\texttt{v}$ as the resulted value of the method call $\texttt{e}$. Note that in $E[\texttt{e}]$, if $\texttt{e}$ is not a value, then the evaluation context E is unique. Thus, the replacement is unambiguous. In addition, if the receiver ℓ' is a gate object and no thread is using that gate (*i.e.* $\mathit{refcount}(\ell', P) = 0$), then by Rule (G-RETURN), the objects allocated in the scope corresponding to ℓ' are deallocated and the fields of ℓ' are reset to their initial values.

Rule (G-SPAWN) evaluates a thread $t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,]$ containing a spawn expression $\texttt{e} = E[\texttt{spawn}\,\texttt{e}_0]$. The value of the spawn expression in $\texttt{e}$ is the distinguished $\ell^{\texttt{th}}$ which is a unique, global reference to an object of class `Thread` and we assume that $\ell^{\texttt{th}}$ is allocated in immortal memory. A new thread t' is created to evaluate $\ell\,\texttt{e}_0$. The new thread is started with a call stack $\overline{\ell\,\ell^{\texttt{th}}}$ that matches the call stack of the original thread t to ensure that scope reference counts are accurate.

5.3 Type Rules

The typing rules are shown in Figure 23 and 24. and the related auxiliary functions are defined in Figure 25. Some auxiliary functions used in typing rules are defined as follows: $\mathit{fields}(\texttt{C})$ returns the list of field declarations in the class $\texttt{C}$ (including the inherited fields) in the form of $\overline{\texttt{C}\,\texttt{f}}$; $\mathit{mdef}(\texttt{m}, \texttt{C})$ returns the defining class of the method

$$\Gamma, \Sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-}\textsc{Var}\text{)}$$

$$\Gamma, \Sigma \vdash \ell : \Sigma(\ell) \qquad \text{(T-}\textsc{Loc}\text{)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\,\overline{\mathtt{f}})}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} : \mathtt{C}_\mathtt{i}} \qquad \text{(T-}\textsc{Field}\text{)}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{C}_0' \quad \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0') = \overline{\mathtt{C}} \to \mathtt{C} \\ \Gamma, \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}} \quad \mathtt{C}_0 \preceq \mathtt{C}_0'\end{array}}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-}\textsc{Invk}\text{)}$$

$$\Gamma, \Sigma \vdash \mathtt{new}\ \mathtt{C}() : \mathtt{C} \qquad \text{(T-}\textsc{New}\text{)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = (\overline{\mathtt{C}}\,\overline{\mathtt{f}}) \quad \Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathtt{C} \preceq \mathtt{C}_\mathtt{i}}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} = \mathtt{e} : \mathtt{C}_\mathtt{i}} \qquad \text{(T-}\textsc{Update}\text{)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{Thread}}{\Gamma, \Sigma \vdash \mathtt{spawn}\,\mathtt{e} : \mathtt{Thread}} \qquad \text{(T-}\textsc{Spawn}\text{)}$$

Fig. 23. *Expression typing.*

$\mathtt{m}$ by searching the class hierarchy upward from $\mathtt{C}$; $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ returns the type signature $\overline{\mathtt{C}} \to \mathtt{C}'$ of the method $\mathtt{m}$ called on the type $\mathtt{C}$, where $\overline{\mathtt{C}}$, $\mathtt{C}'$ are the parameter and return types. The type judgments are of the form $\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C}$, where Γ is the type environment of variables and Σ is the type environment of object labels.

The subtyping relation $<:$ is a reflexive and transitive closure of the relation that $\mathtt{C} <: \mathtt{C}'$ if the class $\mathtt{C}$ extends the class $\mathtt{C}'$. We define the partial order $\preceq$ on types to limit the variables that can refer to scoped objects and gates; $\mathtt{C} \preceq \mathtt{C}'$ is defined if $\mathtt{C} <: \mathtt{C}'$, and in addition, if $\mathtt{C}$ is a scoped type, then $\mathtt{C}'$ is a scoped type in the same package, and if $\mathtt{C}$ is a gate type, then $\mathtt{C} = \mathtt{C}'$. If $\mathtt{C} \preceq \mathtt{C}'$, then we say that $\mathtt{C}$ is a *scope-safe* subtype of $\mathtt{C}'$ and the widening of a reference from the type $\mathtt{C}$ to $\mathtt{C}'$ is *scope-safe*.

By Rules (T-Update) and (T-Invk), the reference widening in the field assignments and parameters passing is *scope-safe*. Rule (T-Store) of the form $\Sigma \vdash \sigma$ says that object store σ is well typed, if the type environment Σ has the same domain as σ and for each object label ℓ in the domain of σ, $\Sigma(\ell)$ is equal to the type of $\sigma(\ell)$ and $\sigma(\ell)$ must also be well-typed. If $\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})$, then by Rule (T-StoreLoc), an object $\mathtt{C}^{\ell'}(\overline{\mathtt{v}})$ is well-typed, if the types of $\overline{\mathtt{v}}$ are *scope-safe* subtypes of the field types.

If a method in the class $\mathtt{C}_0$ is well-typed, then the method body $\mathtt{e}$ is well-typed by the expression typing rules, the type of the method body is a *scope-safe* subtype of the return type, and in addition, the method body must be visible in $\mathtt{C}_0$ as defined by the judgment $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)$ by the expression visibility rules in Figure 26. The predicate $\mathit{override}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{C}} \to \mathtt{C})$ is true if either the method $\mathtt{m}$ is not accessible in $\mathtt{C}_0$ or the type signature returned by $\mathit{mdef}(\mathtt{m}, \mathtt{C}_0)$ is the same as $\overline{\mathtt{C}} \to \mathtt{C}$. In the typing rule for class (T-Class), we require that in a class $\mathtt{C}$, and the types of the fields and the base class must be visible in $\mathtt{C}$. Also, all methods in a class must be well-typed by Rule

Store Typing:

$$\frac{\begin{array}{c}\mathrm{dom}(\Sigma) = \mathrm{dom}(\sigma) \\ \forall \ell \in \mathrm{dom}(\sigma)\ .\ \Sigma \vdash \sigma(\ell)\ \wedge\ \Sigma(\ell) = \texttt{C} \text{ if } \sigma(\ell) = \texttt{C}^{\ell_0}(\overline{\texttt{v}})\end{array}}{\Sigma \vdash \sigma} \qquad \text{(T-S}\textsc{tore}\text{)}$$

$$\frac{\begin{array}{c} \mathit{fields}(\texttt{C}) = (\overline{\texttt{C}}\ \overline{\texttt{f}}) \quad \emptyset, \Sigma \vdash \overline{\texttt{v}} : \overline{\texttt{D}} \quad \overline{\texttt{D}} \preceq \overline{\texttt{C}} \\ \text{if } \texttt{C} \text{ is heap type, then } \ell' = \ell_{\texttt{heap}} \text{ else if } \texttt{C} \text{ is immortal type,} \\ \text{then } \ell' = \ell_{\texttt{imm}} \text{ else } \texttt{C} \text{ is either scoped in } \Sigma(\ell)\text{'s package} \\ \text{or a gate in its immediate subpackage} \end{array}}{\Sigma \vdash \texttt{C}^{\ell}\ (\overline{\texttt{v}})} \qquad \text{(T-S}\textsc{tore}\text{L}\textsc{oc}\text{)}$$

Method typing:

$$\frac{\begin{array}{c} \Gamma = \overline{\texttt{x}} : \overline{\texttt{C}}, \texttt{this} : \texttt{C}_0 \quad \mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{C}} \rightarrow \texttt{C}) \quad \Gamma, \emptyset \vdash \texttt{e} : \texttt{C}' \\ \texttt{C}' \preceq \texttt{C} \quad \Gamma \vdash \mathit{visible}(\texttt{e}, \texttt{C}) \end{array}}{\texttt{C}\ \texttt{m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\, \{\, \texttt{return}\, \texttt{e};\, \}\ \texttt{OK IN}\ \texttt{C}_0 \lhd \texttt{D}} \qquad \text{(T-M}\textsc{ethod}\text{)}$$

Class typing:

$$\frac{\overline{\texttt{M}}\ \texttt{OK IN C} \lhd \texttt{D} \quad \mathit{visible}(\overline{\texttt{CD}}, \texttt{C}) \quad \overline{\texttt{D}} \preceq \overline{\texttt{C}}}{\texttt{class P.C} \lhd \texttt{D}\, \{\, \overline{\texttt{C}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\, \}\ \texttt{OK}} \qquad \text{(T-C}\textsc{lass}\text{)}$$

Fig. 24. *Type rules of store, method, and class.*

Initializer look-up:

$$\mathit{init}(\mathtt{Object}) = ()$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{P.C} \vartriangleleft \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \quad \mathit{init}(\mathtt{D}) = \overline{\mathtt{new}\ \mathtt{D}'()} \\ \mathtt{K} = \mathtt{C}()\ \{\mathtt{super}();\ \mathtt{this}.\overline{\mathtt{f}} := \overline{\mathtt{new}\ \mathtt{C}'()};\ \}\end{array}}{\mathit{init}(\mathtt{C}) = \overline{\mathtt{new}\ \mathtt{D}'()},\ \overline{\mathtt{new}\ \mathtt{C}'()}}$$

Field look-up:

$$\mathit{fields}(\mathtt{Object}) = ()$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{P.C} \vartriangleleft \mathtt{C}'\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \quad \mathit{fields}(\mathtt{C}') = (\overline{\mathtt{C}}'\ \overline{\mathtt{g}})}{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}'\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})}$$

Method definition lookup:

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{P.C} \vartriangleleft \mathtt{C}'\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \quad \mathtt{m}\ \text{is defined in}\ \overline{\mathtt{M}}}{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{C}}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{P.C} \vartriangleleft \mathtt{C}'\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \quad \mathtt{m}\ \text{is not defined in}\ \overline{\mathtt{M}}}{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathit{mdef}(\mathtt{m},\ \mathtt{C}')}$$

Method type lookup:

$$\frac{\begin{array}{c}\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{C}' \quad \mathit{CT}(\mathtt{C}') = \mathtt{class}\ \mathtt{P.C}' \vartriangleleft \mathtt{C}''\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \\ \mathtt{D}\ \mathtt{m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e};\ \} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{D}} \to \mathtt{D}}$$

Method body look-up:

$$\frac{\begin{array}{c}\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{C}' \quad \mathit{CT}(\mathtt{C}') = \mathtt{class}\ \mathtt{P.C}' \vartriangleleft \mathtt{C}''\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \\ \mathtt{D}\ \mathtt{m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e};\ \} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}$$

Valid method overriding

$$\frac{\begin{array}{c}m\ \text{not accessible in}\ \mathtt{C}_0\ \text{or} \\ \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{D} \quad \mathtt{C}, \overline{\mathtt{C}} = (\mathtt{D}, \overline{\mathtt{D}})\end{array}}{\mathit{override}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{C}} \to \mathtt{C})}$$

Fig. 25. *Auxiliary functions.*

(T-METHOD). Note that in (T-CLASS) we abuse notation by writing $visible(\overline{\mathtt{C}},\mathtt{C})$ to assert that all types in the $\overline{\mathtt{C}}$ are visible in C.

Type visibility:

$$visible(\mathtt{C},\mathtt{C}_0) \text{ iff } \begin{cases} \text{either } \mathtt{C} \text{ is a scoped type and} \\ \quad \text{in the same or the super-package of } \mathtt{C}_0 \\ \text{or } \mathtt{C} \text{ is a gate type and} \\ \quad \text{in the immediate subpackage of } \mathtt{C}_0 \end{cases}$$

Static expression visibility:

$$\Gamma \vdash visible(\mathtt{this},\mathtt{C}_0) \qquad \dfrac{\begin{array}{c} \Gamma,\emptyset \vdash \mathtt{e} : \mathtt{C} \qquad visible(\mathtt{C},\mathtt{C}_0) \\ \forall \mathtt{e}' \in subexp(\mathtt{e})\ .\ \Gamma \vdash visible(\mathtt{e}',\mathtt{C}_0) \end{array}}{\Gamma \vdash visible(\mathtt{e},\mathtt{C}_0)}$$

$$subexp(\mathtt{e}) = \begin{cases} \emptyset & \text{if } \mathtt{e} \ = \ \mathtt{x} \mid \mathtt{v} \mid \mathtt{new}\ \mathtt{C}() \\ \{\mathtt{e}_0,\overline{\mathtt{e}}\} & \text{if } \mathtt{e} \ = \ \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \\ \{\mathtt{e}_0,\mathtt{e}_1\} & \text{if } \mathtt{e} \ = \ \mathtt{e}_0.\mathtt{f}_\mathtt{i} := \mathtt{e}_1 \\ \{\mathtt{e}_0\} & \text{if } \mathtt{e} \ = \ \mathtt{e}_0.\mathtt{f}_\mathtt{i} \mid \mathtt{spawn}\,\mathtt{e}_0 \mid \mathtt{reset}\ \mathtt{e}_0 \end{cases}$$

Fig. 26. *Type and expression visibility.*

Visibility of types and expressions. The static constraints in our model are mostly to restrict widening of references, and also to limit the accessibility of expressions by their types. For example, an expression of scoped type C is only visible in the defining package of C and its subpackages. We define a relation on types – $visible(\mathtt{C},\mathtt{C}_0)$ (type C is *visible to* type $\mathtt{C}_0$), which encodes the SJ access control rules:

- a scoped type C defined in package p is visible to the classes defined in p and its subpackages;
- a gate type C in package p is only visible to the classes defined in the immediate superpackage of p
- an immortal class type in imm is visible to any classes; and
- a heap class type is only visible to other heap classes.

One slightly surprising implication of this definition is that a gate type is not visible in its own class definition. Thus a gate class C does not contain code that refers to itself with the exception, as we shall see later, of the pseudo variable this which may indeed be used to access fields and methods from within the gate class.

We illustrate the visibility relation of types with the table in Figure 27, which shows when the types in the leftmost column is visible in the classes in the first row.

We check the method body to determine whether type visibility constraints are violated in a class. In Rule (T-METHOD), the judgment $\Gamma \vdash visible(\mathtt{e},\mathtt{C}_0)$ holds if e of type C is visible in a class $\mathtt{C}_0$, which means that either $\mathtt{e} = \mathtt{this}$ or the type C is visible in the class $\mathtt{C}_0$ (i.e. $visible(\mathtt{C},\mathtt{C}_0)$), and if $\mathtt{e} = \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ and C is a scoped type,

	`p.GateP`	`p.ScopedP`	`p.q.GateQ`	`p.q.ScopedQ`
`p.GateP`				
`p.ScopedP`	visible	visible	visible	visible
`p.q.GateQ`	visible	visible		
`p.q.ScopedP`			visible	visible

Fig. 27. *The types* `p.GateP`, `p.ScopedP` *are the gate and a scoped type in the package* `p` *and* `p.q.GateQ`, `p.q.ScopedQ` *are the gate and scoped type in the package* `p.q`. *The package* `p.q` *is a subpackage of* `p`. *The table entries indicate whether the types in the leftmost column is visible in the classes in first row.*

then `C` and $\texttt{C}_0$ are in the same package, and in addition, all the subexpressions of `e` are visible in $\texttt{C}_0$. We make an exception for `this` because even though a gate type is visible only to the classes of its immediate super-package, a gate object must be able to use the variable `this` for accessing its fields and calling its methods. For any scoped class, the type of the variable `this` are always visible in its class. The restriction also limits creating new objects of the scoped classes in the current package and gate classes in the immediate subpackages.

5.4 Properties.

The purpose of our model is to simplify the allocation of objects in scoped memory areas. Thus, we would like to statically guarantee the properties that during the evaluation of a real-time program,

- the nesting structure of scopes remains a tree, and
- deallocation of scoped memory areas does not create dangling pointers.

In RTSJ, the nesting structure of scopes is determined by how threads enter scopes. In our model, the scope structure is fixed by how the gate objects representing the scopes are created. That is, if a scope a is represented by a gate object created in the scope b, then a must be directly contained in b; moreover, the gate object representing a is defined in the immediate subpackage of the gate object representing b. Thus, our type system guarantees that the scopes represented by the gate objects always form a tree. It also ensures that the threads in a program will preserve such a scope tree such that each thread either enters the scopes already entered by the thread or enters a new scope directly contained in the current scope of the thread. Thus, even though a scope stack of a thread may grow indefinitely (e.g. the thread reenters the scopes already on stack), the nesting structure of scopes resembles the nesting structure of the scoped packages and always remains a tree.

To ensure that deallocation of scoped memory area does not create dangling pointers, we require that a scoped object can only access objects with the same or longer lifetime, while a gate object can in addition access the objects allocated in the scope that it represents. Thus, when the last thread in a scope exits, the objects in that scope can be deallocated. The references to the deallocated objects are no longer accessible in the program because they are only accessible to the scoped objects with the same or shorter lifetime and to the gate of the scope, but those scoped objects have already been deallocated and the methods of the gate object are not being invoked. The above accessibility constraints are enforced by SJ's type rules, where a scoped type is accessible in the classes of its defining package and the subpackages. It is possible for two instances of the same class to be allocated in two sibling scopes (they share some parent scope). To prevent such objects from accessing each other, we limit the access to a gate object to itself and the classes in its immediate super-package. Consequently, an object may only gain access to the gate of its own scope and the gates of its immediate nested scopes and thus, it cannot reference objects in its sibling scope.

Our proof strategy for the above properties is to show that the safety invariant that we define below is preserved in each reduction step. We also show that subject reduction preserves the typing of programs and if a program is well-typed, safe, not terminated, then it always makes progress. This implies that in a well-typed and safe program, the deallocation of scoped memory area will not create dangling pointers since a program with dangling pointers may fail to make progress,

Safety Invariant We say that an object o can *safely access* o' if either o' has longer lifetime than o or o is the gate of the scope where o' is allocated. A store σ is safe if for each label ℓ defined in σ, the object $\sigma(\ell)$ can *safely access* the objects referenced in its fields. A stack frame ℓ $\mathtt{e}$ is safe if the object $\sigma(\ell)$ can *safely access* every object referenced in $\mathtt{e}$. A program σ, P is *safe* if

- σ is safe and each frame in the call stack of each thread in P is *safe*,
- for each gate object ℓ, if $\mathit{refcount}(\ell, P) = 0$, then objects allocated in the scope represented by ℓ are not in σ, and
- for each frame $\ell\,\mathtt{e}$ in the call stack of each thread in P, the allocation scope of ℓ is either heap, immortal scope, or a scope represented by the receiver object of a previous frame in the call stack.

Well-typed program A program σ, P is well-typed if $\exists \Sigma$ such that $\Sigma \vdash \sigma$ and the call stack of each thread in P is well-typed.

Given Σ, the call stack $\overline{\ell\,\mathtt{e}}\,.\,\ell\;\mathtt{e}\,.\,\ell'\;\mathtt{e}'$ is well-typed if

- $\ell'\;\mathtt{e}'$ is well-typed, $\overline{\ell\,\mathtt{e}}\,.\,\ell\;\mathtt{e}$ is well-typed, and
- either $\mathtt{e} = \ell^{\mathtt{th}}$, or $\mathtt{e} = E[\mathtt{e}_0]$, $\mathtt{e}_0$ is not a value, and if $\emptyset; \Sigma \vdash \mathtt{e}' : \mathtt{C}'$ and $\emptyset; \Sigma \vdash \mathtt{e}_0 : \mathtt{C}$, then $\mathtt{C}' \preceq \mathtt{C}$.

Given Σ, ℓ $\mathtt{e}$ is well-typed if ℓ is well-typed and $\exists \mathtt{C}$ such that $\emptyset; \Sigma \vdash \mathtt{e} : \mathtt{C}$ and $\mathit{visible}(\mathtt{e}, \ell)$ (defined below) is true.

Given Σ, $\Sigma(\ell) = \mathtt{C}_0$, and $\emptyset; \Sigma \vdash \mathtt{e} : \mathtt{C}$, the constraint $\mathit{visible}(\mathtt{e}, \ell)$ is true if

- either $\mathtt{e} = \ell$ or $\mathit{visible}(\mathtt{C}, \mathtt{C}_0)$,
- if $\mathtt{e} = \mathtt{new}\ \mathtt{C}()$ and $\mathtt{C}$ is a scoped type, then $\mathtt{C}$ and $\mathtt{C}_0$ are in the same package, and
- for each subexpression $\mathtt{e}'$ of $\mathtt{e}$, $\mathit{visible}(\mathtt{e}', \ell)$ is true.

An expression $\mathtt{e}$ contains a null pointer exception if it has the form

$$\mathtt{null.m}(\overline{\mathtt{v}}) \;\mid\; \mathtt{null.f} \;\mid\; \mathtt{null.f} := \mathtt{v}$$

Lemma 1 (Subject Reduction). *If σ, P is well-typed, safe, and $\sigma, P \Rightarrow \sigma', P'$, then σ', P' is well-typed and safe.*

We say that a thread in P is terminated if it has the form $t[\,\ell_0\,\mathtt{v}\,]$ or it contains a null pointer exception.

Lemma 2 (Progress). *If σ, P is well-typed and not all threads in P have terminated, then there exists σ', P' such that $\sigma, P \Rightarrow \sigma', P'$.*

We say that an irreducible program σ, P is stuck if P contains a non-terminated thread. In Theorem 1, we prove that if a program is well-typed, then it will not get stuck. As usual, $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$.

Theorem 1. *If σ, P is well-typed, safe, and $\sigma, P \Rightarrow^* \sigma', P'$, then σ', P' is not stuck.*

The proofs of the key results are in appendix.

6 Discussion and Future Work

The combination of Scoped Types with Aspects is a promising means of structuring policy with its corresponding mechanism. When a real-time program is in this form, we can get the benefit of high level abstractions along with increased flexibility of their key mechanisms as aspects. The approach further allows for flexible combinations of lightweight static verification. The prototype implementation of STARS shows that the benefits of our approach can be obtained using mostly off-the-shelf technologies, in particular, existing aspect-oriented languages and static verifiers, and with only minimal changes to a real-time Java virtual machine. There is also potential for significant performance improvements. In our benchmark we have seen that a STARS program may run 28% faster than the corresponding RTSJ program.

This work has illustrated how aspects can extract and localize real-time memory management concerns. In our case study the real-time memory management and threading specific portion of the application could be extracted as a simple declarative aspect.

One of the advantages of STARS is its truly lightweight type system. So lightweight, in fact, that one only needs make a judicious choice of package names to denote nesting of memory regions. The attraction is that no changes are needed in the language and tool chain, and that the rules are simple to explain. We do not attempt to sweep the costs of adopting STARS under the rug. As we have seen in the case study, there are cases where we had to change interfaces from objects to primitive types, thus forfeiting some of the software engineering benefits of Java. We were forced to duplicate the code of some common libraries in order to abide by the rules of scoped types. While there are clear software engineering drawbacks to code duplication, the actual refactoring effort in importing those classes was small. With adequate tool support the entire refactoring effort took less than a day. The hard part involved discovering and disentangling the scope structure of the programs that we were trying to refactor.

The benefits in terms of correctness can not be overemphasized. Every single practitioner we have met has remarked on the difficulty of programming with RTSJ-style scoped memory. In our own work we have encountered numerous faults due to incorrect scope usage. As a reaction against this complexity many RTSJ users are asking for real-time garbage collection. But RTGC is not suited for all applications. In the context of safety critical systems a number of institutions are investigating restricted real-time 'profiles' in which the flexibility of scoped memory is drastically curtailed [22]. But even in those proposals, there are no static correctness guarantees. Considering the cost of failure, the effort of adopting a static discipline such as the one proposed here is well justified.

We see several areas for future work. One direction is to increase the expressiveness of the STARS API to support different kinds of real-time systems and experiment with more applications to further validate the approach. Another issue to be addressed is to extend JAVACOP to support AspectJ syntax. In the current system, we are not checking aspects for memory errors. This is acceptable as long as aspects remain simple and declarative, but real-time aspects may become more complex as we extend STARS, and their static verification will become a more pressing concern. Finally we want to investigate extensions to the type system to reduce, or eliminate, the need for code duplication.

References

1. Chris Andreae, James Noble, Shane Markstrum, and Todd D. Millstein. A framework for implementing pluggable type systems. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–74. ACM, 2006.
2. Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 2006.
3. David F. Bacon, Perry Chang, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 285–298, January 2003.
4. David F. Bacon, Perry Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 466–478, 2003.
5. Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbuster, Edward Pla, and David Holmes. A real-time Java virtual machine for avionics. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*. IEEE Computer Society, 2006.
6. William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.
7. Edward G. Benowitz and Albert F. Niessner. A patterns catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 497–507, 2003.
8. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

9. Greg Bollella and Krik Reinholtz. Scoped memory. In *Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC02)*, 2002.
10. Gregory Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk 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*. ACM, 2003.
11. Gregory Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Frédéric Parain. Mackinac: Making hotspot real-time. In *Eigth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 45–54. IEEE Computer Society, 2005.
12. Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation*, June 2003.
13. Gilad Bracha. Pluggable type systems. In *OOPSLA 2004 Workshop on Revival of Dynamic Languages*, 2004.
14. Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *In Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, September 1993.
15. Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM Press, 1991.
16. Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management, ISMM*, pages 85–96. ACM, 2004.
17. Angelo Corsaro and Doug Schmidt. The design and performance of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.
18. Morgan Deters and Ron Cytron. Automated discovery of scoped memory regions for real-time Java. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 25–35, Berlin, June 2002. ACM Press.
19. Daniel Dvorak, Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004), 12-14 May 2004, Vienna, Austria*, pages 15–22, Silver Spring, MD 20910, USA, 2004. IEEE Computer Society Press.
20. Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in java. *Electr. Notes Theor. Comput. Sci*, 113:105–121, 2005.
21. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM Conference on Programming Language Design and Implementation (PLDI'03)*, pages 282–293, Berlin, Germany, June 2002.
22. HIJA. European High Integrity Java Project. `www.hija.info.`, 2006.
23. John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991.

24. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
25. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
26. Jagun Kwon and Andy Wellings. Memory management based on method invocation in RTSJ. In *OTM Workshops 2004, LNCS 3292, pp. 33–345*, 2004.
27. Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Joint ACM Java Grande/ISCOPE Conference*, November 2002.
28. C. Nakhli, C. Rippert, G. Salagnac, and S. Yovine. Efficient region-based memory management for resource-limited real-time embedded systems. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, Nantes, France, July 2006.
29. Albert F. Niessner and Edward G. Benowitz. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 508–519, 2003.
30. James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of the 12th Eurpean Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.
31. James Noble and Charles Weir. *Small Memory Software: Patterns for systems with limited memor y*. Addison-Wesley, 2000.
32. Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04)*, Vienna, Austria, May 2004.
33. David C. Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the 3rd International Symposium on Distributed Objects and Applications, DOA 2001, 17-20 September 2001, Rome, Italy*, pages 3–4, Silver Spring, MD 20910, USA, 2001. IEEE Computer Society Press.
34. David C. Sharp, Edward Pla, Kenn R. Luecke, and Ricardo J. Hassan II. Evaluating Real-Time Java for mission-critical large-scale embedded systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), May 27-30, 2003, Toronto, Canada*, pages 30–36, Silver Spring, MD 20910, USA, 2003. IEEE Computer Society Press.
35. Daniel Spoonhower, Joshua Auerbach, David F. Bacon, Perry Cheng, and David Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proceedings of the conference on Programming language design and implementation (PLDI)*, pages 283–294, 2006.
36. Shiu Lun Tsang, Siobhán Clarke, and Elisa Baniassad. An evaluation of aspect-oriented programming for java-based real-time systems development. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 291–300. IEEE Computer Society, 2004.
37. Nanbor Wang, Christopher D. Gill, Douglas C. Schmidt, and Venkita Subramonian. Configuring real-time aspects in component middleware. In *CoopIS, DOA, and ODBASE, OTM Confederated International Conferences,*, volume 3291, pages 1520–1537. Springer, 2004.
38. Andy J Wellings and Peter P. Puschner. Evaluating the expressive power of the Real-Time Specification for Java. *Real-Time Systems*, 24(3):319–359, 2003.
39. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS04)*, Lisbon, Portugal, December 2004.

40. Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1), January 2006.

Appendix

This appendix contains proofs of the main results of this article.

Lemma 3. *If* $\sigma, \ell_0\ \mathtt{e} \to \sigma', \ell'_0\ \mathtt{e}'$, $\exists \Sigma$ *such that* σ *and* $\ell_0\ \mathtt{e}$ *are well-typed and safe, and* $\emptyset; \Sigma \vdash \mathtt{e} : \mathtt{C}$*, then* $\exists \Sigma'$ *such that* σ' *and* $\ell'_0\ \mathtt{e}'$ *are well-typed and safe,* $\emptyset; \Sigma' \vdash \mathtt{e}' : \mathtt{C}'$*, and* $\mathtt{C}' \preceq \mathtt{C}$*.*

Proof. 1. Suppose $\mathtt{e} = \ell.\mathtt{f_i}$ and $\mathtt{e}' = \mathtt{v_i}$, where $\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})$, $\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})$. Since σ is well-typed, the type of $\mathtt{v_i}$ has to be a scope-safe subtype of $\mathtt{C_i}$. By the assumption that $\ell_0\ \mathtt{e}$ is well-typed, it follows that $\mathit{visible}(\mathtt{e}, \ell_0)$. It means that the type of $\ell.\mathtt{f_i} - \mathtt{C}$ must be visible to the class of ℓ_0. By Rule (T-Field), $\mathtt{C} = \mathtt{C_i}$. Since the type of $\mathtt{v_i}$ is a scope-safe subtype of $\mathtt{C_i}$, the type of $\mathtt{v_i}$ is visible to the class of ℓ_0. Thus, we have $\mathit{visible}(\mathtt{e}', \ell_0)$.

By assumption, it is safe for ℓ_0 to access ℓ and it is safe for ℓ to access $\mathtt{v_i}$. If ℓ is a scoped object, then ℓ_0 can safely access $\mathtt{v_i}$ because $\mathtt{v_i}$ must be in the same or an outer scope of ℓ. If ℓ is a gate object, then there are three possibilities: (1) $\ell_0 = \ell$. (2) ℓ_0 is in a scope directly enclosing the scope of ℓ and $\mathtt{v_i}$ is in the outer scope of ℓ. In this case, ℓ_0 can safely access $\mathtt{v_i}$ since the latter is in the same or an outer scope of ℓ_0. (3) ℓ_0 is in a scope directly enclosing the scope of ℓ and $\mathtt{v_i}$ is in the scope of ℓ. This case can be ruled out by the fact that the type of $\mathtt{v_i}$ must be visible to the class of ℓ_0.

2. Suppose $\mathtt{e} = \ell.\mathtt{f_i} := \mathtt{v}$ and $\mathtt{e}' = \mathtt{v}$, where $\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})$, $\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})$, and $\sigma' = \sigma[\ell \mapsto \mathtt{C}^{\ell'}([{}^{\mathtt{v}}/_{\mathtt{v_i}}]\overline{\mathtt{v}})]$. By assumption and Rule (T-Update), $\mathtt{v}$ is well-typed and its type is a scope-safe subtype of $\mathtt{C}_i$. Thus, σ' is well-typed. Also by assumption, ℓ_0 can safely access ℓ and $\mathtt{v}$. Based on Rule (T-Class), $\mathtt{C}_i$ must be visible to the class ℓ and hence, the type of $\mathtt{v}$ must be visible to the class ℓ as well. Therefore, $\mathtt{v}$ must be in the same or an outer scope of ℓ and σ' is safe.

3. Suppose $\mathtt{e} = \mathtt{new\ C}()$, $\mathtt{e}' = \ell$, and $\sigma' = \sigma[\ell \mapsto \mathtt{C}^{\ell'}(\overline{\mathtt{null}})]$, where $\ell' = \ell_{\mathtt{heap}}$ if $\mathtt{C}$ is a heap type, $\ell' = \ell_{\mathtt{imm}}$ if $\mathtt{C}$ is an immortal class type, and otherwise $\ell' = \mathit{allocScope}_\sigma(\ell_0)$. By assumption, we have $\mathit{visible}(\mathtt{e}, \ell_0)$, and hence $\mathtt{C}$ must be in the package of the class of ℓ_0 or a gate in its subpackage. If $\mathtt{C}$ is not a heap or immortal class type, then ℓ' is the gate representing the allocation scope of ℓ_0, and hence, $\mathtt{C}$ must either be scoped in the package of the class of ℓ' or a gate in its subpackage, Thus, $\exists \Sigma'$ such that σ' is well-typed and σ' is safe.

4. Suppose $\mathtt{e} = \ell.\mathtt{m}(\overline{\mathtt{v}})$ and $\mathtt{e}' = [{}^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}},\ {}^{\ell}/_{\mathtt{this}}]\mathtt{e}_0$, where $\mathit{mbody}(\mathtt{m},\ \mathtt{C}_0) = (\overline{\mathtt{x}},\ \mathtt{e}_0)$ and $\sigma(\ell) = \mathtt{C}_0^{\ell'}(\overline{\mathtt{v}}')$. We need to show that $\ell\ \mathtt{e}'$ is well-typed and safe. Since $\mathtt{e}$ and the method $\mathtt{m}$ are well-typed, by Rules (T-Method) and (T-Invk), it is straightforward to show by induction that $\emptyset; \Sigma \vdash \mathtt{e}' : \mathtt{C}'$ and $\mathtt{C}' \preceq \mathtt{C}$. The object labels in $\mathtt{e}'$ are $\overline{\mathtt{v}}$ and ℓ. Suppose $\mathtt{m}$ is defined in class $\mathtt{C}'_0$ where $\mathtt{C}_0 <: \mathtt{C}'_0$. From Rule (T-Method), the parameter types of $\mathtt{m}$ are all visible to $\mathtt{C}'_0$. By Rule (T-Invk), $\mathtt{C}_0 \preceq \mathtt{C}'_0$, which means that $\mathtt{C}_0$ and $\mathtt{C}'_0$ are the same class or in the same package. Thus, the parameter types of $\mathtt{m}$ are visible to $\mathtt{C}_0$ as well. Again, by Rule (T-Invk), the type of $\mathtt{v_i}$ is scope-safe subtype of the type of the parameter $\mathtt{x_i}$ for all $\mathtt{i}$. Thus, the types of $\overline{\mathtt{v}}$ are visible to $\mathtt{C}_0$. Consequently, for each object label ℓ' in $\mathtt{e}'$, either it is ℓ or its type is visible to the type of ℓ. Therefore, we have $\mathit{visible}(\mathtt{e}', \ell)$ and $\ell\ \mathtt{e}'$ is well-typed. By assumption, ℓ_0 can safely access $\overline{\mathtt{v}}$ and ℓ. From earlier argument, the types of $\overline{\mathtt{v}}$ are visible to the type of ℓ. Thus, $\overline{\mathtt{v}}$ can only be allocated in the same or outer scopes of ℓ and ℓ can safely access $\overline{\mathtt{v}}$. Therefore $\ell\ \mathtt{e}'$ is safe as well.

Proof of Lemma 1 (Subject Reduction) *If σ, P is well-typed, safe, and $\sigma, P \Rightarrow \sigma', P'$, then σ', P' is well-typed and safe.*

Proof. 1. Suppose $P = P'' \mid t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]]$ $P' = P'' \mid t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}']]$, where $\mathtt{e} \neq \ell'.\mathtt{m}(\overline{\mathtt{v}})$ and $\sigma, \ell\ \mathtt{e} \rightarrow \sigma', \ell\ \mathtt{e}'$. From Lemma 3, σ' and $\ell\ \mathtt{e}'$ are well-typed and safe, and the type of $\mathtt{e}'$ is a scope-safe subtype of the type of $\mathtt{e}$. It is straightforward to show by induction that $E[\mathtt{e}']$ is well-typed and its type is a scope-safe subtype of $E[\mathtt{e}]$. Thus, $\ell\,E[\mathtt{e}']$ is well-typed and safe, and consequently σ', P' is well-typed and safe.

2. Suppose $P = P'' \mid t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]]$ and $P' = P'' \mid t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]\,.\,\ell'\,\mathtt{e}']$, where $\mathtt{e} = \ell'.\mathtt{m}(\overline{\mathtt{v}})$ and $\sigma, \ell\ \mathtt{e}\ \ \rightarrow \sigma, \ell'\ \mathtt{e}'$. From Lemma 3, $\ell'\ \mathtt{e}'$ is well-typed, safe, and the type of $\mathtt{e}'$ is a scope-safe subtype of the type of $\mathtt{e}$. Thus, σ, P' is well-typed. If the type of ℓ' is not a heap or immortal type, then the allocation scope of ℓ' must be represented by a gate object. Since σ, P is safe, ℓ can safely access ℓ' and ℓ' must be allocated in the same or outer scope of ℓ'. Thus, by induction, the allocation scope of ℓ' is represented by the receiver object of a frame in the call stack of P. Therefore, σ, P' is safe.

3. Suppose $P = P'' \mid t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{e}]\,.\,\ell'\,\mathtt{v}]$ and $P' = P'' \mid t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{v}]]$, where $\sigma(\ell') = \mathtt{C}^{\ell''}(\overline{\mathtt{v}})$. Since P is well-typed, either $E[\mathtt{e}] = \ell^{\mathtt{th}}$, or $\mathtt{e}$ is not a value and the type of $\mathtt{v}$ is a scope-safe subtype of the type of $\mathtt{e}$. Suppose $\mathtt{e}$ is not a value. Thus, by induction, it can be shown that $E[\mathtt{v}]$ is well-typed and its type is a scope-safe subtype of the type of $E[\mathtt{e}]$. Since the type $\mathtt{e}$ is visible to the type of ℓ, the type of $\mathtt{v}$ is visible to the type of ℓ as well. Moreover, the object ℓ' can safely access $\mathtt{v}$ and ℓ can safely access ℓ'. Thus, ℓ can safely access $\mathtt{v}$ and as a result, we have $\mathit{visible}(E[\mathtt{v}], \ell)$ and $\ell\,E[\mathtt{v}]$ is well-typed.

If $\mathtt{C}$ is a gate class type and $\mathit{refcount}(\ell', P') = 0$, then objects in the scope represented by ℓ' will be deallocated and the objects in the fields of ℓ' will be reset to their initial value. Suppose that the resulted store is σ'. Suppose $\sigma'' = \mathit{deallocate}(\sigma, \ell')$, $\sigma' = \sigma''[\ell' \mapsto \mathtt{C}^{\ell''}(\overline{\mathtt{null}})]$. Since σ'' is a subset of σ, it is apparently safe. Thus, σ' is safe. Since σ, P is safe and if the reference count of ℓ' is zero, then the objects allocated in the scope of ℓ' are not in σ', σ', P' is safe.

To show that $\exists \Sigma'$ such that P' is well-typed and $\Sigma' \vdash \sigma'$, we let $\Sigma'(\ell_0) = \mathtt{C}_0$ for each ℓ_0 in the domain of σ' if the type of $\sigma'(\ell_0)$ is $\mathtt{C}_0$. We only need to show that the objects deallocated from σ are not accessible in σ' and P'. The objects removed from σ are allocated in the scope represented by ℓ'. Since σ is safe, for each object ℓ_0, the values referenced in its fields are allocated in the same or outer scopes of ℓ_0. σ' is σ with objects allocated in the scope of ℓ' removed and the fields of ℓ' set to null. The objects allocated in ℓ' are not accessible in the fields of objects in σ' either. Suppose the opposite is true: there exists ℓ_0 in σ' having a field holding objects allocated in ℓ'. Then, ℓ_0 must be allocated in an inner scope of ℓ' – ℓ_0'. Since σ', P' is safe, $\mathit{refcount}(\ell_0', P') \neq 0$ (because otherwise, no objects in σ' is allocated in ℓ_0'). However, if $\mathit{refcount}(\ell_0', P') \neq 0$, then ℓ' must be the receiver object of a stack frame in a thread preceding another frame with ℓ_0' as the receiver object. Then, $\mathit{refcount}(\ell', P') \neq 0$, which is a contradiction. Therefore, no objects of σ' holds references to the objects allocated in ℓ' and σ' is safe. The objects allocated in the scope of ℓ' are not accessible to the threads in P'. Suppose the opposite is true: an object allocated in ℓ' appears in a stack frame of a thread in P'. Then the receiver object of the frame should have be allocated in the same or an inner scope of ℓ'. This means that ℓ' must be the receiver of a frame on the call stack as well, which is a contradiction to the assumption that the reference count of ℓ' is zero in P'. Thus, σ', P' is well-typed.

4. Suppose $P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,]$ and $P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\ell^{\texttt{th}}]\,] \;\mid\; t'[\,\overline{\ell\,\ell^{\texttt{th}}}\,.\,\ell\,\texttt{e}'\,]$, where $\texttt{e} = E[\texttt{spawn}\,\texttt{e}']$. It is clear that σ, P' is well-typed and safe.

Lemma 4. *If* $\texttt{e}$ *has the form* $\ell.\texttt{f}_{\texttt{i}}$, $\ell.\texttt{f}_{\texttt{i}} := \texttt{v}$, $\texttt{new C}()$, *or* $\ell.\texttt{m}(\overline{\texttt{v}})$, *and* $\exists\sigma, \Sigma, \ell_0$ *such that* σ *and* $\ell_0\ \texttt{e}$ *are well-typed then* $\exists\sigma', \texttt{e}', \ell'$ *such that* $\sigma, \ell_0\ \texttt{e} \rightarrow \sigma', \ell'\ \texttt{e}'$.

Proof. 1. Suppose $\texttt{e} = \ell.\texttt{f}_{\texttt{i}}$. Since σ and $\ell_0\ \texttt{e}$ are well-typed, ℓ is defined in σ and the fields of its type contains the field $\texttt{f}_{\texttt{i}}$. Thus, if this field holds value $\texttt{v}_{\texttt{i}}$, then $\texttt{e}' = \texttt{v}_{\texttt{i}}$ and $\sigma, \ell_0\ \texttt{e} \rightarrow \sigma, \ell_0\ \texttt{e}'$.

2. Suppose $\texttt{e} = \ell.\texttt{f}_{\texttt{i}} := \texttt{v}$. Since σ and $\ell_0\ \texttt{e}$ are well-typed, ℓ is defined in σ and the fields of its type contains $\texttt{f}_{\texttt{i}}$. Thus, $\sigma, \ell_0\ \texttt{e} \rightarrow \sigma', \ell_0\ \texttt{v}$, where σ' is σ with the field $\texttt{f}_{\texttt{i}}$ if ℓ in σ replaced by $\texttt{v}$.

3. Suppose $\texttt{e} = \texttt{new C}()$. Since σ and $\ell_0, \texttt{e}$ are well-typed, ℓ_0 is well-typed. Thus, ℓ' is either $\ell_{\texttt{imm}}$, $\ell_{\texttt{heap}}$, or $\mathit{allocScope}_\sigma(\ell_0,)$. Let ℓ be a fresh label so that $\sigma' = \sigma[\ell \mapsto \texttt{C}^{\ell'}(\overline{\texttt{null}})]$. Then, $\sigma, \ell_0\ \texttt{e} \rightarrow \sigma', \ell_0\ \ell$.

4. Suppose $\texttt{e} = \ell.\texttt{m}(\overline{\texttt{v}})$. Since σ and $\ell_0, \texttt{e}$ are well-typed, ℓ is defined in σ and method $\texttt{m}$ is defined on object ℓ. If the method body and parameters of $\texttt{m}$ are $\texttt{e}_0$ and $\overline{\texttt{x}}$, then $\sigma, \ell_0\ \texttt{e} \rightarrow \sigma, \ell\ [{}^{\overline{\texttt{v}}}/_{\overline{\texttt{x}}},\ {}^{\ell}/_{\texttt{this}}]\texttt{e}_0$.

Proof of Lemma 2 (Progress) *If* σ, P *is well-typed and not all threads in* P *have terminated, then there exists* σ', P' *such that* $\sigma, P \Rightarrow \sigma', P'$.

Proof. Since not all threads in P have terminated, there exists a thread in P with a call stack of the form $\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}]$ or $\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}]\,.\,\ell'\,\texttt{v}$, which does not contain null pointer exception.

1. Suppose $P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}]\,]$ where $\texttt{e} \neq \ell'.\texttt{m}(\overline{\texttt{v}})$ and $\texttt{e} \neq \texttt{spawn}\,\texttt{e}'$. Since $E[\texttt{e}]$ does not contain null pointer exception. $\texttt{e}$ must have the form $\ell'.\texttt{f}_{\texttt{i}}$, $\ell'.\texttt{f}_{\texttt{i}} := \texttt{v}$, or $\texttt{new C}()$. Since σ, P is well-typed, σ and $\ell\,\texttt{e}$ are well-typed. Thus, by Lemma 4, $\exists\sigma', \texttt{e}'$ such that $\sigma, \ell\ \texttt{e} \rightarrow \sigma', \ell\ \texttt{e}'$. Let $P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}']\,]$. Then $\sigma, P \Rightarrow \sigma', P'$ by Rule (G-Step).

2. Suppose $P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}]\,]$ and $\texttt{e} = \ell'.\texttt{m}(\overline{\texttt{v}})$. Since σ, P is well-typed, σ and $\ell\,\texttt{e}$ are well-typed. Thus, by Lemma 4, $\exists\sigma, \texttt{e}'$ such that $\sigma, \ell\ \texttt{e} \rightarrow \sigma, \ell\ 'e'$. Let $P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{e}]\,.\,\ell'\,\texttt{e}'\,]$. Then $\sigma, P \Rightarrow \sigma, P'$ by Rule (G-Enter).

3. Suppose $P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,.\,\ell'\,\texttt{v}\,]$ Since σ, P is well-typed, $\texttt{e}$ is either $\ell^{\texttt{th}}$ or has the form $E[\texttt{e}]'$, where $\texttt{e}'$ is not a value. Let $P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\texttt{v}]\,]$. If the type of ℓ' is not a gate or the reference count of ℓ' in P' is not zero, then let $\sigma' = \sigma$. Otherwise, let σ' be σ with all objects in allocated in ℓ' removed and all fields of ℓ' set to null. Then, $\sigma, P \Rightarrow \sigma', P'$ by Rule (G-Return).

4. Suppose $P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,]$ and $\texttt{e} = E[\texttt{spawn}\,\texttt{e}']$. Let $P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,E[\ell^{\texttt{th}}]\,] \;\mid\; t'[\,\overline{\ell\,\ell^{\texttt{th}}}\,.\,\ell\,\texttt{e}'\,]$, where t' is a fresh thread label. Then $\sigma, P \Rightarrow \sigma, P'$ by Rule (G-Spawn).