Translation of Java⋆ to Real-Time Java Using Aspects ⋆⋆

Extended Abstract

Morgan Deters, Nicholas Leidenfrost, and Ron K. Cytron⋆⋆⋆

Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA

Abstract. The Real-Time Specification for Java [1] (RTSJ) introduces the concept of nested-scope memory areas to Java. This design allows a programmer to allocate objects in areas that are ignored by the garbage collector. Unfortunately, the specification of scoped memory areas currently involves the introduction of unwieldy, application-specific constructs that can ruin the reusability of the affected software.

We propose the use of aspects [2], in particular the AspectJ [3] system, to transform a Java program into a scope-aware RTSJ program automatically. Moreover, we have developed analysis that automatically determines storage scopes, in response to information provided by an instrumented form of the application at hand. That instrumentation is also accomplished using aspects. Here we present our ongoing work in using aspects to detect and specify memory scopes automatically in Java programs.

1 Motivation

One major roadblock to the widespread acceptance of Java as a language for real-time and embedded systems is its reliance at runtime on an asynchronous garbage collector. Such a collector may preempt threads running in a real-time environment or take control of the system during memory allocation requests. The collector can then take an unbounded amount of time to complete its task, introducing unacceptable unpredictability into the system. To address these concerns and provide greater programmatic control over memory allocation and usage, the Real-Time Specification for Java [1] (RTSJ) introduces to Java the use of structural *memory areas* through the `MemoryArea` type, the subtyping of which can admit multiple strategies for allocating and deallocating storage.

One of the key memory strategies of RTSJ is provided with the `ScopeMemory` type, which allows for memory areas to be associated with a particular

⋆ A registered trademark of Sun Microsystems

⋆⋆ Funded by the National Science Foundation under grant 0081214 and by DARPA under contract F33615-00-C-1697

⋆⋆⋆ Contact author: mdeters@cs.WUSTL.edu

scope of thread execution. When an execution scope is exited, objects in the memory area are released without the need for a garbage collector. However, the unit of a thread is not necessarily the most natural or useful level at which to create and manipulate memory areas. Consider the introduction of a `Cache` type into a system (Figure 1(a)). A singleton `Cache` object is constructed when the cache is first accessed. In a corresponding RTSJ design, we want this object to be allocated in `ImmortalMemory`, because the singleton `Cache` is around for the length of the program. Placing it in `ImmortalMemory` keeps the garbage collector from scanning or marking it. This would be a waste of time since we know that it will never become collectible: the scope of the `Cache` object is global. To realize this RTSJ design, the programmer replaces occurrences of `new Cache` with invocations of `newInstance` on the desired memory area, as shown in Figure 1(b).

If class `Cache` had a more complicated design that supported multiple instances, this recoding of `new` instructions to `newInstance` invocations would need to be pushed into the user's code at *every* site that a `Cache` object was constructed. This breaks the encapsulation of the `Cache` type. Memory instructions are sprinkled throughout user code rather than being factored out into a separate, decoupled memory strategy. The resulting code will not be easily reusable.

Figure 1(c) shows a better approach. With the `CacheMemory` aspect, we can write the `Cache` class just as we did in the Java implementation; the aspect assumes the responsibility of placing it into the correct type of memory area. Further, if we extended `Cache` to support multiple instantiations, then `CacheMemory` could weave into user code, allowing the user to construct a `Cache` instance naturally, without being responsible for its memory requirements.

We propose an automatic system for translating Java code into `MemoryArea`-aware RTSJ code by determining the necessary RTSJ memory scopes required to describe it. By employing *probing aspects*, we determine where scopes can be used and develop a graph from which we can compute provably legal scope hierarchies. A scope hierarchy is selected, runtime execution points are chosen for opening and closing scopes, and an aspect is generated to enforce the structure on the original program's execution. Objects in the modified program are located in scoped memory areas rather than in the garbage-collected heap.

2 Scope Determination

As specified by RTSJ, a thread becomes associated with a scope when that thread calls `enter` on the scope. The scope then resumes the thread by calling its `run` method. Any number of threads can enter a scope, and that scope can be deleted only when all threads that have entered it have exited their `run`

```
class Cache {
 protected static Cache singleton;
 public static Cache instance() {
  if(singleton == null)
   try {
    singleton = new Cache();
   } catch(Exception e) { ... }
  return singleton;
 }
 // etc.
}
```

(a)

```
class Cache {
 protected static Cache singleton;
 public static Cache instance() {
  if(singleton == null)
   try {
    singleton = (Cache)
     ImmortalMemory.instance().
      newInstance( Cache.class );
   } catch(Exception e) { ... }
  return singleton;
 }
 // etc.
}
```

(b)

```
aspect CacheMemory {
 around() returns Cache : calls(Cache.new(..)) {
  ConstructorCallJoinPoint ccjp =
   (ConstructorCallJoinPoint) thisJoinPoint;
  ConstructorSignature cs =
   (ConstructorSignature) ccjp.getSignature();

  return (Cache)( ImmortalMemory.instance().
                  newInstance( Cache.class )
 }
}
```

(c)

Fig. 1. (a) A partial Java implementation of a `Cache` type. **(b)** A partial RTSJ implementation of a `Cache` type in `ImmortalMemory`. **(c)** An AspectJ aspect used to rewrite `Cache` instantiations to be in `ImmortalMemory`. RTSJ `Cache` objects can now be constructed via `new`, just like objects in Java.

method; detection of this condition is accomplished by reference-counting the scope. Any memory allocated via a simple `new` instruction is allocated in the memory area currently associated with the allocating thread. Explicit `MemoryArea` allocation instructions are also permitted via the `newInstance` method, as in the `Cache` example of Figure 1.

`ScopeMemory` scopes can be nested, and it makes sense to design nesting relationships when small and perhaps iterative pieces of code produce a lot of garbage during their computation—this garbage can then be cleaned up all at once, on exit of the inner scope, without the need for a garbage collector. Programmers and program analysis tools tend to associate notions of storage scope with method scope. Thus, it may be desirable for a `ScopeMemory` to be associated with a particular scope of execution—a method, for example. The scope could then be deleted when the method exits. RTSJ avoids the improper collection of objects that are still reachable by mandating that object references may only point to objects within the same scope or outward from inner scopes

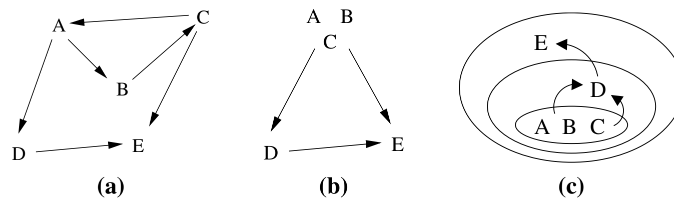

Fig. 2. A *doesReference graph* for a program P **(a)** before and **(b)** after grouping strongly connected objects. An arrow from A to B indicates that object A stores a reference to object B. **(c)** shows one possible scoping structure, where E is in the outermost scope, D is in a sub-scope, and A, B, and C are in a sub-scope of D's scope. Object references in RTSJ may only point outward to enclosing scopes.

to enclosing scopes; scopes are at least as long-lived as those that nest within them, so these outward-pointing references are considered safe.

If a Java program were simply moved to an RTSJ platform, then by default all objects would be allocated in the garbage-collected heap, which offers no guarantees for real-time activities. To generate an RTSJ scope hierarchy out of Java's flat memory model, we work backwards, tracking which objects reference which others, to build a set of scope nesting structures that do not violate the object referencing regulations of RTSJ.

We have developed a *reference-probing aspect* to determine these legal RTSJ `ScopeMemory` assignments. The probe defines as join points [2] the object instantiations and assignments of interest to us and builds a *doesReference graph* to track which objects refer to which other objects. The *doesReference graph* may contain strongly connected objects; these objects are grouped together as they must share a scope. This collapses the more general graph into a directed acyclic graph (DAG).

For example, if object A references object B, the DAG contains an edge from A to B. Then B's scope must be at least as long-lived as A's. There are two legal scoping hierarchies in this instance: that where B is in an outer scope and A is in an inner scope, and that where A and B are both in the same scope. A simple (but nonoptimal) algorithm for determining suitable scopes is to perform a topological traversal of the DAG, which corresponds to a right-to-left preorder traversal of any depth-first spanning tree of the DAG. Figure 2 shows an example illustrating this procedure.

3 Join Point Discovery

The join points in the original program at which we need to inject instructions to enter these scopes must also be determined. Consider Figure 3, a possible

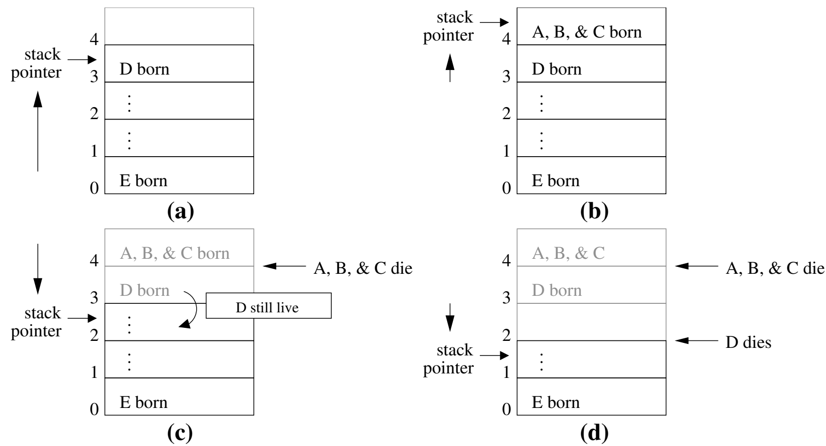

Fig. 3. A view into the execution stack of program P at different points in P's execution. **(a)** E is born first, in frame 0. In frame 3, D is allocated, then **(b)** A, B, and C (which refer back to D) are allocated in frame 4. A, B, and C are not returned or thrown from the execution scope that generated them, and we know from Figure 2 that there are no references to them, so they must die upon exiting their birth stack frame—this indicates that we can close their associated memory scope when frame 4 pops. **(c)** However, D escapes the execution scope of its birth (it was returned or thrown), and it is still live in frame 2. **(d)** Therefore, D's scope must not close until frame 2 is popped and D is known to be dead.

execution stack for the program P whose reference behavior is described by Figure 2. First, E is constructed, then in some later stack frame D is constructed (and stores a reference to E). If we can determine that D is always dead at the time a particular stack frame pops, we can close its scope at that point. This is demonstrated in the figure.

We can reason about the frame events of Figure 3 by engineering advice on method and constructor join points. To determine the points at which objects are dead, we weave a *liveness-probing aspect* into program P. This aspect inspects method and constructor executions to determine when objects are born and when they become unreachable.

By combining the *doesReference* scope information harvested by the *reference-probing* aspect and the frame birth and death data from the *liveness-probing aspect*, we can discover a scoping hierarchy that respects both RTSJ's requirements on object references and the observed execution flow of P.

4 Conclusion

Our approach has the following advantages:

1. *The memory concerns of the system are described and enforced in a modular fashion.* The memory concerns are described through the use of aspects, rather than sprinkling memory instructions throughout the code via `newInstance` invocations. This is a particularly important issue for objects in real-time Java packages that want or need to manage their own memory concerns under RTSJ. Without aspects, the user would have the responsibility of placing packaged objects in the correct type of memory area.
2. *Automation of* `ScopeMemory` *detection and management lowers development costs.* This dynamic analysis approach can be used to find a memory-efficient scoping structure, and the resulting, automatically generated memory aspect is easily tested by weaving it in to user code. Human-readable descriptions of object behavior can be generated that allow a useful view into the system and point out inefficiencies or unintended design consequences in the system.
3. *The introduction of aspect code into the target program introduces real-time predictability.* Because the scoping hierarchies are computed and the necessary join points are discovered offline, the aspect that enforces the runtime use of RTSJ scoped memory areas consists mainly of a table lookup. This can be done in bounded time, and we expect the translated program's performance to be more predictable (and suitable to real-time environments) than the original program.
4. *User source files are unchanged.* The real-time modifications are completely described in separate aspect source code; our aspect weaves into the user's source in order to modify it. For large source code trees, disk space requirements can be dramatically smaller. Additionally, one source code tree is sufficient for both Java and `ScopeMemory`-enhanced RTSJ code.

In summary, we have proposed a mechanism for automating the creation of RTSJ memory scopes. The expression of those scopes is accomplished via aspects, as is the offline dynamic analysis to determine the scopes. At present we have pieces in place to perform the analysis and to create scopes that are respectful of object references from program runs. Our future plans call for investigating tradeoffs between various scope nesting structures, in terms of footprint and the overhead incurred for managing the scopes.

References

1. Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
2. Gregor Kiczales. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.
3. The AspectJ Organization. Aspect-Oriented Programming for Java. `www.aspectj.org`, 2001.