Mission Modes for Safety Critical Java

Martin Schoeberl

Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Abstract. Java is now considered as a language for the domain of safety critical applications. A restricted version of the Real-Time Specification for Java (RTSJ) is currently under development within the Java Specification Request (JSR) 302. The application model follows the Ravenscar Ada approach with a fixed number of threads during the *mission* phase. This static approach simplifies certification against safety critical standards such as DO-178B. In this paper we extend this restrictive model by mission modes. Mission modes are intended to cover different modes of a real-time application during runtime without a complete restart. Mission modes are still simpler to analyze with respect to WCET and schedulability than the full dynamic RTSJ model. Furthermore our approach to thread stopping during a mode change provides a clean coordination between the runtime system and the application threads.

1 Introduction

The Real-Time Specification for Java (RTSJ) [1] was a first and successful attempt to enable Java based real-time application development. The RTSJ is a quite dynamic environment reflecting the dynamic nature of Java and aimed at soft real-time applications. Soon after the RTSJ was published, suggestions to restrict the RTSJ for high-integrity applications have been proposed [2, 3]. The restrictions are based on the Ravenscar profile for Ada [4].

The basic idea of the proposed profiles is to divide the application into an *initialization* phase and a *mission* phase. During initialization all threads are created and data structures for communication are allocated. In the mission phase a fixed number of threads are scheduled. The profile disallows garbage collection to provide time predictable scheduling of the real-time threads. Only a restricted version (no sharing between threads) of scoped memory areas is allowed for simple dynamic memory management.

1.1 A Safety Critical Java Profile

The Java Specification Request (JSR) 302 builds on those profiles to define a standard for safety critical Java [5]. The intention of this JSR is to provide a profile that supports programming of applications that can be validated against safety critical standards such as DO-178B level A [6]. In [7] we have defined a simple safety critical Java profile based on the former work. Our extension includes, besides a more natural way to define the scheduling requirements by deadlines instead of priorities, a clean way to shutdown the application.

The profile contains *periodic* and *sporadic* threads. The periodic threads represent the main application logic. Sporadic threads handle software and hardware events (i.e. interrupts). Figure1 shows the simplified classes that represent periodic and sporadic threads.

```
package javax.safetycritical;

public abstract class PeriodicThread
        extends RealtimeThread {

    public PeriodicThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    public PeriodicThread(RelativeTime period)
}

public abstract class SporadicThread
        extends RealtimeThread {

    public SporadicThread(String event,
        RelativeTime minInterval,
        RelativeTime deadline, int memSize)

    public SporadicThread(String event,
        RelativeTime minInterarrival)

    public void fire()
}
```

Fig. 1. Classes for periodic and sporadic real-time events

Both classes extend RealtimeThread (shown in Figure 2) that contains properties common to periodic and sporadic threads. Note that RealtimeThread is not the RTSJ version of a RealtimeThread. We do not inherit from java.lang.Thread to avoid problematic constructs (e.g. sleep()). Furthermore, our thread abstraction does not contain a start() method. All threads are started together at mission start. We extend the profile in this paper to start threads also on a mode change.

The run() method is different from an RTSJ or java.lang thread. It is abstract to enforce overriding in a sub-class. Method run() has to return a boolean value. This value indicates that a periodic task is ready for shutdown or for stopping during a mode change.

In contrast to the RTSJ scheduling parameters are defined by time values (deadlines) instead of priorities. Deadlines represent application requirements more natural

```
package javax.safetycritical;

public abstract class RealtimeThread {

    protected RealtimeThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    protected RealtimeThread(String event,
        RelativeTime minInterval,
        RelativeTime deadline, int memSize)

    abstract protected boolean run();

    protected boolean cleanup() {
        return true;
    }
}
```

Fig. 2. The base class for all schedulable entities

than priorities. An implementation on top of a priority based scheduler can map the deadlines to priority values by a deadline monotonic order [8]. This mapping is part of the runtime system and does not have to be done by the application developer. The distinction between an initialization and a mission phase (with static threads) simplifies the generation of the mapping. The mapping is performed only once at the start of the mission. Furthermore, the definition of deadlines as the main scheduling parameter allows switching to an EDF scheduler in the middleware without changing the application logic.

1.2 Mission Modes

Although a static approach to the development of safety critical applications simplifies (or even enables) certification, the model of a single mission with a fixed number of threads is quite restrictive. Safety critical applications often consist of different modes during runtime, e.g. take off, cruse, and landing of an airplane. During those different modes different tasks have to be performed. In this paper we evaluate different forms to implement mode changes: At the application level, with dynamic thread creation, and with static mission modes. The main idea is to provide a restricted form of dynamic application change without the hard to analyze dynamic creation and stopping of real-time threads. Just stopping a real-time task can be a dangerous action. We build on [7] to enable a coordinated stopping of real-time tasks during a mode change.

2 Mode Changes

Several safety critical applications have different modes of operation. Imagine a part of an avionic application: takeoff, cruse, and landing are typical modes where different tasks are necessary. Several tasks will run during all modes and several are only needed during a specific mode. When a mode change occurs the continuous tasks should not be disturbed by the mode change.

Tasks that are *stopped* shall have a chance to shutdown in a clean way, i.e. run till all actuators are in a safe position. Furthermore the tasks shall be able to perform some form of cleanup. In [7] we introduced the coordinated shutdown for the whole real-time application. We use the proposed mechanism for the shutdown of threads that belong to a specific mode in this proposal.

Switching between different modes does not only represent different tasks to be executed but also reusing of resources. We can distinguish between CPU time and memory resources that can be reused. With respect to reuse three forms of mode change frameworks are possible:

1. Mode changes at the application level do not reuse memory; CPU budgets are reused, but hard to analyze
2. Dynamic created threads can reuse memory with mission scoped memory and CPU resources
3. Predefined modes do not reuse memory. CPU budgets are reused and simple to analyze

2.1 Application Level

A simple form of mode changes can be implemented at the application level – a form of *poor man's* mode changes. All threads that are needed in all modes of the mission are started. The actual application logic is only executed at the modes needed as following example shows:

```
public void run() {

    if (State.mode==State.TAKE_OFF) {
        takeOffTask();
    }
}
```

In this example the run() method represents a periodic task. This form fits well for the static approach to start all threads during mission start. However, it leaves the handling of different missions at the application programming and complicates clean mission changes. Furthermore it complicates WCET analysis and incorporation of those WCET values into schedulability analysis for different modes.

2.2 Dynamic Threads

A real-time system that allows dynamic creation of threads during the mission phase can easily reuse CPU budgets and memory. This is the model that the RTSJ proposes.

However, this dynamic thread creation is hard to analyze and will hamper certification of the safety critical application.

A more restrictive model will group threads belonging to missions or mission phases. Those groups can share a scoped memory that can be recycled when all threads of the group are not needed anymore. However, no coordinated shutdown of threads that belong to a mission phase is part of the RTSJ framework.

2.3 Predefined Modes

Our approach is to define different mission modes at the initialization phase. All threads are still created in this phase and no dynamic thread creation during mission phase is necessary. Each mode contains the list of threads that have to run in this mode. Therefore we can still build all the scheduling tables before the mission starts. The static assignment of threads to mission modes also simplifies schedulability analysis.

We do not reuse any memory that is used for communication between threads for different modes. All those structures are still allocated at the initialization phase. However, threads itself can use scoped memories for intermediate data structures during their execution. We assume that the amount of memory that is used for thread communication is not that high and we would gain little from reusing part of it for different modes.

3 Implementation

The different modes are represented by a simple class that contains the list of threads belonging to the mode:

```
public class MissionMode {

    public MissionMode(RealtimeThread rt[]) {
        // immutable MissionMode object
    }
}
```

Class MissionMode provides just the constructor with the list of real-time threads to implement immutable mode objects. An immutable mode guarantees that the mode cannot be changed at runtime.

The class RealtimeSystem represents the real-time runtime system and is shown in Figure 3. Method start() performs the mission start. Compared to [7] it contains now the mission mode as a parameter. That mode is the one which is used as the first one. Method changeMod() performs the change to the new mission mode. Method stop() performs shutdown of the whole real-time application.

3.1 Shutdown

Before we discuss mode changes we give a brief description of the shutdown process. The same mechanism is used to shutdown individual threads for a mode change.

```
package javax.safetycritical;

public class RealtimeSystem {

    /**
     * This class is uninstantiable.
     */
    private RealtimeSystem()

    public static void start(MissionMode m)

    public static void changeMode(MissionMode m)

    public static void stop()

    public static boolean modeChangePending()

    public static int currentTimeMicros()
}
```

Fig. 3. The representation of the real-time system with mission modes

In [7] we provide an additional phase for the real-time application: *Shutdown*. This phase is intended to provide a safe termination of the real-time system. All threads have a chance to bring actuators into a safe position.

The shutdown phase is initiated similar to the start of the mission phase, by invoking stop() from RealtimeSystem. However, we cannot simply stop all threads, but need a form of cooperation. All real-time threads return a boolean value from the run() method. This value indicates: *I'm ready for a shutdown*. When a thread is in a critical operation, where a shutdown is not allowed, the thread just returns false to delay the shutdown process. The runtime system waits for *all* threads to be ready for shutdown before actually performing the shutdown.

During the shutdown the cleanup() method is scheduled periodically (with the same period) *instead* of the run() method. The cleanup method itself also returns a boolean value to signal *shutdown finished* with true. In that case the thread is not scheduled anymore.

3.2 The Mode Change

When switching from mode A to mode B all threads that belong to both modes just continue to be scheduled. Threads part of mode A and not part of mode B have to be stopped. We reuse our approach to a clean shutdown of periodic threads, as described before, to perform the mode change. All threads that have to be stopped go through the same phases as during a shutdown. That means that the application logic of a single thread does not need to be changed when extending the single mode profile to a multi-mode system.

A current mode change can be queried by the application threads with RealtimeSystem.modeChangePending(). An application task should query this state before performing a long lasting state change where the application thread cannot be stopped. This query is not mandatory. However, it can help to perform the mode change in less time.

When all to-be-stopped threads have performed their cleanup function the threads that are part of mode *B* and not part of mode *A* are *added* according to their release parameters to the schedule table. As this table is known at the initialization phase it is easily built in advance and this *adding* is just a switch between different tables. Performing this switch is the last step in the mode change.

3.3 An Example

Figure 4 shows a simple example using the mission modes. We define three modes: takeoff, cruise, and landing. For each mode we have a periodic task that performs the operation. Only thread takeoff is shown in the example. Furthermore, the task watchdog is part of all three modes.

For each mode we create a MissionMode object and add all threads that belong to the mission with the constructor during the initialization phase. The mission is started with the takeoff mode with RealtimeSystem.start(modeTakeOff). When this phase is finished a mode change is triggered by the takeoff thread with changeMode(modeCruise). During this mode change takeoff is stopped and cruise is started. The thread watchdog just continues to run during the mode change and in the new mode.

4 Discussion

The example showed that the API for different application modes is quite simple and intuitive to use. The proposed solution provides some form of dynamic application change during runtime within the static framework for safety critical Java.

4.1 Analysis

The restricted form of dynamic application change with different modes simplifies WCET and schedulability analysis. For each mode all threads that will be scheduled are known in advance. During the mode change all not anymore used threads are stopped first before new threads from the new mode are started. In that case we do not need additional schedulability analysis for the mode changes.

During stopping of a thread the cleanup() method is invoked instead of the run() method at the same period. We only have to use the larger WCET value from the two methods for schedulability analysis.

The analysis of different modes is slightly more complex than the analysis of a single mission phase. However, it is still simpler than fully dynamic thread creation in the mission phase. We assume that the proposed modes provide enough dynamics in safety critical applications without hampering certification.

```
public class MissionExample {

    static MissionMode modeTakeOff;
    static MissionMode modeCruise;
    static MissionMode modeLand;

    public static void main(String[] args) {

        PeriodicThread watchdog = new PeriodicThread(
                new RelativeTime(1000, 0)) {

            protected boolean run() {
                // do the watchdog work
                return true;
            }
        };

        PeriodicThread takeoff = new PeriodicThread(
                new RelativeTime(100, 0)) {

            boolean finished;
            protected boolean run() {
                doWork();
                if (finished) {
                    RealtimeSystem.changeMode(modeCruise);
                }
                return true;
            }
            protected boolean cleanup() {
                // we need no cleanup as we triggered
                // the mode change
                return true;
            }
            private void doWork() {
                // the periodic work
                // sets finished to true when done
            }
        };

        RealtimeThread mto [] = { watchdog, takeoff };
        modeTakeOff = new MissionMode(mto);
        RealtimeThread mcr [] = { watchdog, cruise };
        modeCruise = new MissionMode(mcr);
        RealtimeThread mld [] = { watchdog, land };
        modeTakeOff = new MissionMode(mld);

        RealtimeSystem.start(modeTakeOff);

    }
}
```

Fig. 4. An example of an application with three mission modes

4.2 Runtime Overhead

A runtime system that provides a single mission phase with statically created threads can be implemented very efficient. During the start of the mission all relevant scheduling parameters (e.g. priorities ordered deadline monotonic) can be calculated. As a result a single scheduling table can be built. As this table does not change during runtime an efficient array instead of a list can be used. For a full dynamic system a list of threads that can be changed during runtime has to be used. Therefore, scheduling decisions at runtime are more complex.

The proposed approach of static modes is slightly more complex than the single mission solution. However, as all modes and the resulting scheduling tables are known before the mission start the tables can still be built in advance. At the end of the mode change (when the new threads are released) just the scheduling table has to be set to the precalculated one for the new mode. Scheduling decisions during runtime are as complex as for the single mission system.

The mode change itself, with stopping some threads and scheduling their cleanup method, is as complex as the shutdown process in the former proposal.

5 Conclusion

In this paper we have proposed an enhancement of safety critical Java to cover different modes during the runtime of a real-time application. The intention is to keep the system still simple in order to certify it according to standards such as DO-178B level A [6]. The WCET and schedulability analysis for a single mode is identical to the analysis of a single mission. Our proposal for mode changes also includes a coordinated shutdown of threads that are not used anymore in a new mode.

We believe that the slightly more complex analysis is outweighed by the benefits from reusing CPU budgets for different modes and providing a simple framework for the application to perform those mode changes. As a next step we will evaluate the proposal with a real-world example. When this evaluation is positive we will suggest this framework to the JSR 302 expert group for inclusion in the future standard of Safety Critical Java Technology.

References

1. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M.: The Real-Time Specification for Java. Java Series. Addison-Wesley (June 2000)
2. Puschner, P., Wellings, A.J.: A profile for high integrity real-time Java programs. In: 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC). (2001)
3. Kwon, J., Wellings, A., King, S.: Ravenscar-Java: A high integrity profile for real-time Java. In: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, ACM Press (2002) 131–140
4. Burns, A., Dobbing, B., Romanski, G.: The ravenscar tasking profile for high integrity real-time programs. In: Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies, Springer-Verlag (1998) 263–275

5. Java Expert Group: Java specification request JSR 302: Safety critical java technology. Available at http://jcp.org/en/jsr/detail?id=302
6. RTCA/DO-178B: Software considerations in airborne systems and equipment certification. (December 1992)
7. Schoeberl, M., Sondergaard, H., Thomsen, B., Ravn, A.P.: A profile for safety critical java. In: 10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2007). (May 2007)
8. Audsley, N.C., Burns, A., Richardson, M.F., Wellings, A.J.: Hard real-time scheduling: The deadline monotonic approach. In: Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atalanta (1991)