A Formal Model of the Safety-Critical Java Level 2 Paradigm

Matt Luckcuck Ana Cavalcanti Andy Wellings

University of York,
UK

iFM, June 2016

Outline

Outline

- Java in Safety-Critical Systems
- Safety-Critical Java
  - Safety-Critical Java Level 2
- *Circus*
- Modelling Approach
- Summary and Next Steps

Java in Safety-Critical Systems

Java

- Java not traditionally associated with safety-critical programs
- More abstraction, less control. . .
    - Garbage collection
    - Poor scheduling control

"The intrinsic safety of the standard language is irrelevant, **it is how safe the use of the language can be made that matters**" – Hatton *Safer C (1995)*

Java in Safety-Critical Systems

Java

- Interesting for safety-critical systems:
  - Strong typing
  - Precise definition
  - Widely understood
  - Language features e.g. exception handling
- Long standing effort to improve Java. . .
  - Java Community Process's Java Specification Requests (JSR)

Java in Safety-Critical Systems

Real-Time Specification for Java (RTSJ)

- Java Community Process: JSR 1
- RTSJ addresses some of the Java's problems. . .
  - Region-based memory
  - Control memory usage
  - Better scheduling control
- Complex for safety-critical programs

Safety-Critical Java

SCJ Overview

- International effort lead by The Open Group
- Java Community Process: JSR 302
- Builds on RTSJ
- Aimed at applications that must be certified
- Embeds a new, simpler programming paradigm
- $\sim$ 112 pages of language specification...
    - $\sim$ 36 classes and interfaces
    - Does not cover verification

Safety-Critical Java

SCJ Overview

- Requires a real-time virtual machine
- Real-time abstractions from the RTSJ
- Restricted hierarchical programming structure
- Region-based hierarchical memory
- Fixed priority scheduler with Priority Ceiling Emulation

Safety-Critical Java

Tools

- SCJ has specific tools for. . .
  - Memory Safety
  - Memory Consumption
  - Execution Time
  - Schedulability
  - Program Verification

Safety-Critical Java

Compliance Levels

- Level 0:
  - Single processor
  - Cyclic executive
- Level 1:
  - Introduce concurrency
  - More release patterns
- Level 2:
  - Highly concurrent
  - Multi-processor
  - Complicated release patterns
  - Suspension

Safety-Critical Java

SCJ API

- `Safelet`: controls the program and starts the Mission Sequencer
- `MissionSequencer`: instantiates and starts a sequence of Missions
- `Mission`: controls a set of tasks, represented by subclasses of Managed Schedulable
- `ManagedSchedulable`: super-type of all four tasks:
    - `PeriodicEventHandler`
    - `AperiodicEventHandler`
    - `OneShotEventHandler`
    - `ManagedThread`

Safety-Critical Java

Mission Phases

1. Initialize: creates and registers schedulables
2. Execute: simultaneously activate mission's schedulables
3. Cleanup: reset data structures

SCJ Level 2

SCJ Level 2 Features

- Access to suspension features
- Access to all Managed Schedulables. . .
    - Uniquely: `ManagedThread` and `MissionSequencer`
- Schedulable Mission Sequencers allow multiple Missions to be active. . .
    - One active Mission per Mission Sequencer
    - Schedulables from any running Mission may preempt, based on their priorities
    - No assumption of schedulable from a particular mission having priority

Modelling Approach

This work. . .

- Models the Safety-Critical Java (SCJ) Level 2 paradigm using *Circus*
- Agnostic of Java
- Limited treatment of some Exceptions
- First formal semantics of SCJ Level 2
- Builds on a model of SCJ Level 1. . .
  - Level 2 features
  - API changes
- Model ignores. . .
  - Scheduling
  - Resources (E.g. Memory)

Model Benefits

Top-Down

Target for refinement-based development of SCJ programs

- Refinement from abstract to concrete specifications. . .
    - Concrete specifications that capture the SCJ paradigm
- Correctness by construction

Bottom-Up

Translation from SCJ code to model

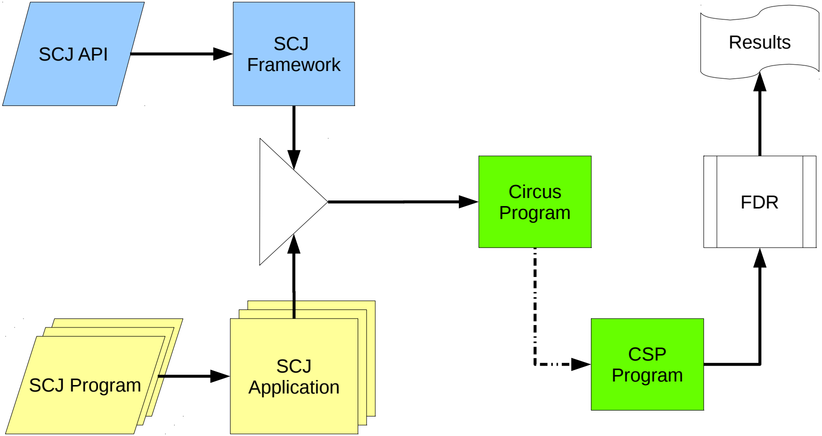
- Catches certain program errors. . .
    - Deadlock
    - Divergence
    - Exceptions

Modelling Approach

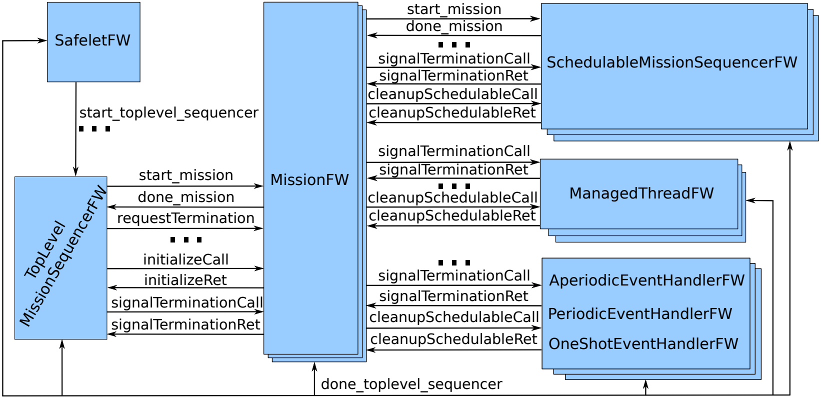
Circus Language

- Combination of **Z** and **CSP**
  - Captures both State and Behaviour
- Organised around Processes
  - State component (**Z**) to hold variables
  - Actions (**Z** and **CSP**) to perform behaviours
  - Main action specifies overall behaviour
- Communication through **CSP** channels

Modelling Approach



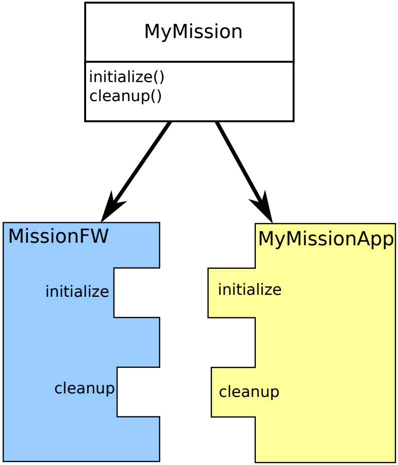
Modelling Approach



Modelling Approach

Framework:

- Generic
- API classes



Application:

- Specific
- Program behaviour

Modelling Approach

Exceptions

- Modelled by an event followed by **Chaos**
    - Built-in process that diverges
- Only for paradigm misuse
- Coverage:
    - Thread interrupt
    - Incorrect method parameter
    - Suspension without a lock
    - Locking an object with a lower priority
    - Registering schedulable twice

Synchronisation and Suspension

Java Synchronisation and Suspension

- SCJ restrictions:
  - Only `synchronized` methods
  - Threads queue in eligibility order
  - Most eligible waiting thread:
    - Highest priority thread...
    - That has been waiting for the longest time
- Suspension is achieved with `Object.wait()` and `Object.notify()...`
  - May only be called on `this`

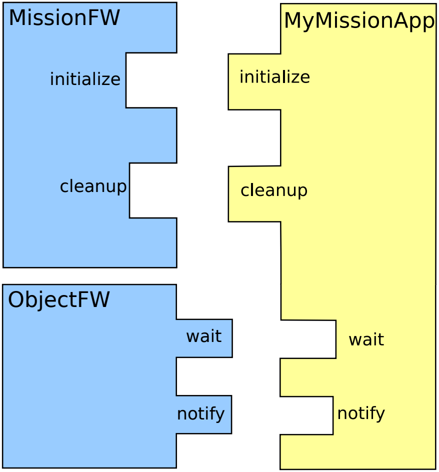
Synchronisation and Suspension

Our Model

Extra processes to model synchronisation and suspension. . .

- *ObjectFW*:
    - Object used as a lock
    - Stores threads waiting on this Object
    - Controls threads trying to lock this Object
- *ThreadFW*:
    - Schedulable calling a `synchronized` method
    - Tracks priority and interrupted status

Synchronisation and Suspension



Evaluation

Confidence

- Close correspondence with the SCJ API
- Builds on the Level 1 model...
  - Level 1 model has been validated against the API
- Our modelling effort simplified SCJ termination protocol...
  - Adopted in v0.96

Evaluation

Translation

- Informal translation strategy, which provides semantics to our model
- 10 hand-translated examples covering different release patterns, synchronisation, and schedulable mission sequencers
- Prototype tool, $\mathrm{T}^{\underline{\mathrm{ight}}}\mathrm{R}^{\underline{\mathrm{ope}}}$, to produce models from code:

Readers–Writers	6 classes	$\sim$1.2 seconds
Aircraft	25 classes	$\sim$2.3 seconds

Evaluation

Animation and Model Checking

- Translated models `CSPm` to use FDR3...
    - Animate the Framework to compare to SCJ API and running programs
    - Model Check the program specifications to ensure deadlock- and divergence-freedom

Summary and Further Work

Summary

- Model SCJ Level 2 paradigm as **Framework** and **Application**
- Model of SCJ Level 2 contributes to . . .
  - **Top-down** development as a refinement target
  - **Bottom-up** development as verification tool
- Translation Strategy to generate application models
- Models correspond closely to SCJ programs
- Validated our models by translating them to `CSPm` and using FDR3 to animate and model check

Next Steps

- Formalise translation strategy
- Improve TightRope to translate all our example applications

Thank you for listening.