A Formal Account of the Safety-Critical Java Mission Model

Ana Cavalcanti, Andy Wellings, Jim Woodcock, and Frank Zeyda.

University of York, UK

10 February 2010

Context and Objectives

The HiJaC project tackles the development of methods, techniques and tools to formally reason about **S**afety-**C**ritical **J**ava Applications.

Context

- Five year projected funded by EPSRC.
- 3 RAs and one PhD Student. (Frank Zeyda, Kun Wei, Chris Marriott)
- Grant Holder: Ana Cavalcanti, Jim Woodcock and Andy Wellings.
- Project Partners from Industry:
  → AWE, IBM Canada, Praxis Systems, Sun Microsystems.

Project Objectives

1. Sounds framework for the specification, design, and implementation of verified real-time systems.
2. Make use of the *Circus* family of languages.
3. Integration of paradigms and novel refinement techniques.

Outline

1. The *Circus* Family of Languages
2. The Automotive Cruise Controller
3. Safety-Critical Java
4. Formalisation of the Mission Model (*)
5. Validation in FDR
6. Future Outlook and Conclusions

(*) = Primary contribution of the research reported in this talk.

The *Circus* Family of Languages

Main Features of *Circus*

- Combines elements from Z, CSP and GCL (Dijkstra, Morgan).
- Key concept of *Circus* is that of a process.
- Processes encapsulates state and actions that operate on the state.
  - State is always local to the process.
  - Actual process behaviour is defined by a designated main action.
- Interact with the environment through communication events.
- *Circus* has a refinement calculus and denotational semantics based on the UTP (**U**nifying **T**heories of **P**rogramming).
- *Circus* is in particular suitable for the specification of state-rich, reactive systems.

The result of 10 years of on-going research at York and other universities.

The *Circus* Family of Languages

Circus has been extended in various ways to deal with particular specification problems and semantic issues.

Circus and Friends

- *Circus* Time additionally takes into account time.
  - Provides operators to specify real-time behaviour.
- **Oh***Circus* introduces the notion of classes to model data objects.
  - Process are not first-class citizens but classes are.
- Slotted-*Circus* introduces the notion of time slots akin to the semantic model for Handel-C.
- ArcAngel*C* is a tactic language for *Circus* refinement.

Tools for *Circus*

- A syntax and type checker for *Circus*
  - Implemented within the **C**ommunity **Z** **T**ools (CZT).
- A mechanised semantics in ProofPower-Z.

The Automotive Cruise Controller (Events)

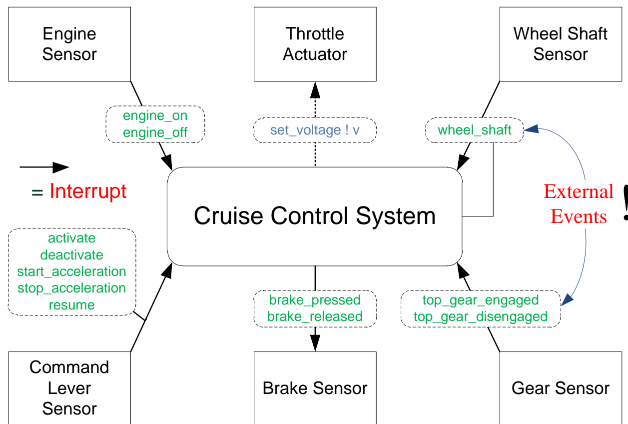


Figure: Overview of Events and Components of the ACCS.

The ACCS reacts to external sensor events (green) and in cruise mode periodically sets the voltage on the throttle actuator (blue).

The Automotive Cruise Controller (System)

Requirements

- Response time to external events.
  - Sensor events have deadlines and priorities associated with them.
- Setting of the voltage when cruising ($set_voltage\,!\,v$).
  - Must happen at least three times per second.
- Comfortable acceleration.
  - Throttle voltage must increase gradually in acceleration mode.
- Estimation of the current speed.
  - Based on the number of rotations of the wheel shaft.
  - Accuracy depends on latency of events and timers.
- Cover functional as well as dynamic aspects of the system.
  - Computation of the throttle voltage is entirely functional.

Implementation

Originally for RTSJ (Andy Welling's book) but we adapted it to SCJ.

The Automotive Cruise Controller (UML)

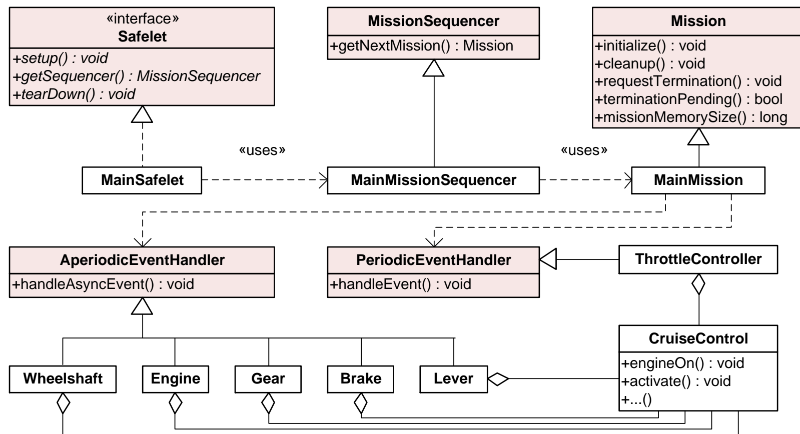


Figure: UML Diagram for the Cruise Controller SCJ Application.

Components in a reddish tint belong to the SCJ framework API.

Safety-Critical Java (SCJ)

Based on the Real-time Specification for Java (RTSJ).

Objectives of SCJ

- Make Java programs amenable to formal analysis and certification.
- Definition of well-defined execution models.
- Cater for different types of program architectures.
  - sequential vs concurrent
  - single- vs multi-processor
  - periodic vs aperiodic tasks
  - event-based vs thread-based
- Predictability of execution behaviour.
  - Restricted thread and memory model, and real-time guarantees.

Most notable features...

- There is no heap. ($\rightarrow$ only immortal and scoped memory areas)
- There is no garbage collector.

Compliance Notion

Level Compliance

We distinguishes Level 0, Level 1 and Level 2 applications.

1. Level 0: only periodic activities, no preemption;
2. Level 1: periodic and aperiodic activities, preemption and concurrency;
3. Level 2: additionally nested missions and no-heap real-time threads.

→ API is more restrictive for lower levels.

Behavioural Compliance

Restricting methods in their behaviours:

- May allocate or not
- May self-suspend or not (use of `wait()` and `synchronize {...}`)

Memory Compliance

Sound use of scoped memory: "no dangling references".

Analysis of SCJ Programs

Static Checking

Purdue University developed tools for static checking of SCJ.

- Establishes level, behavioural and memory compliance.
- Requires **annotation** of the program code.
- Does not consider functional correctness or real-time constraints.

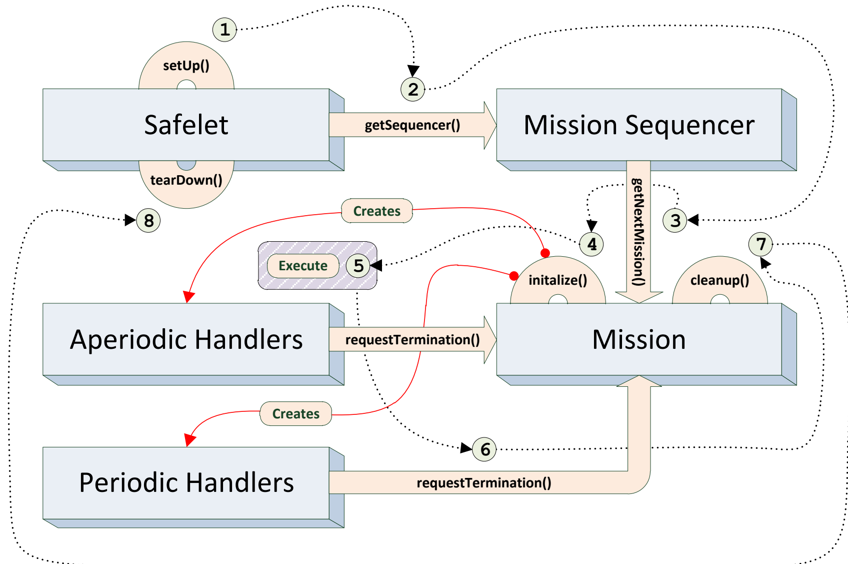
Model Checking and Proof

Various tools exist for Standard Java.

- ESC/Java and ESC/Java2 (**E**xtended **S**tatic **A**nalysis)
- Java PathFinder (model checking execution paths)
- Krakatoa (proof system based on the *WHY* platform)
- ... (Chris Marriott's literature review)

None of these treat all of the following aspects:
functional correctness, concurrency, memory safety and real-time (!).

Life-cycle of a Safelet



Formalisation of the Mission Model

Aspects that we treat...

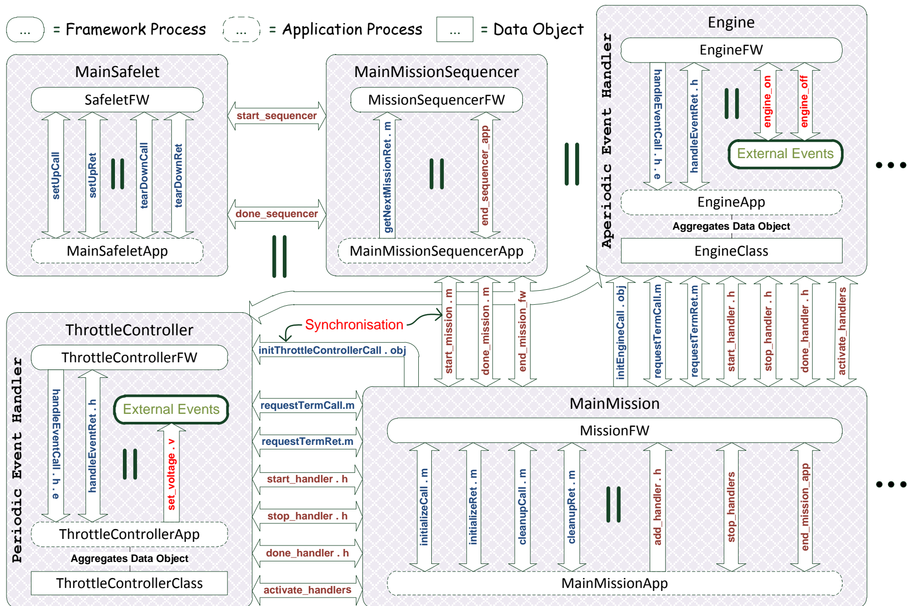
- The entire life-cycle of Safelet execution (Level 1).
- Framework and application behaviour.
- Periodic and aperiodic event handlers.
- Real-time constraints and deadlines.
- External interaction with the environment.

Aspects that we ignore...

- Memory areas and sound use of memory.
  - On-going work by Ana Cavalcanti and Jim Woodcock.
- Level 2 missions.
  - Starting missions within missions. ($\rightarrow$ additional complication)
  - Independent real-time threads.

We use an 'integrated' version of *Circus* to formulate the model.

High-level Specification Architecture



High-level Specification Architecture

Main Concepts

- Separation of framework and application behaviour.
- Components are modelled by a parallel composition.

  $Mission \mathrel{\widehat=} MissionFW \llbracket SyncChan \rrbracket MainMissionApp$

  Active Framework Behaviour $\parallel$ Passive Application Class.
- Framework events realise control mechanisms.
  - *start_sequencer*, *activate_handlers*, *mission_done*, etc.
- Data objects can be attached to application processes.
  - In particular we make use of **Oh***Circus* **classes** here.
- The entire system is the parallel composition of all components.

  $Safelet \parallel MainMissionSequencer \parallel MainMission \parallel Handlers$

At the top level all framework and method call events are hidden.
$\rightarrow$ Only external (sensor and actuator) events are exposed.

The Safelet Component

A Safelet object is the entry point of any SCJ application.

Java code for the MainSafelet class

```
class MainSafelet implements Safelet {
    void setUp() { }
    public MissionSequencer getSequencer() {
      return new MainMissionSequencer();
    }
    void tearDown() { }
}
```

Infrastructure Behaviour

1. Execute the setUp() method of the safelet.
2. Obtain and start the mission sequencer (getSequencer()).
3. Wait for the mission sequencer to finish.
4. Execute the tearDown() method of the safelet.

Formal Model of the Safelet

Framework Process

process *SafeletFW* $\mathrel{\widehat=}$ **begin**

$Setup \mathrel{\widehat=} setUpCall \longrightarrow setUpRet \longrightarrow Skip$

$Execute \mathrel{\widehat=} start_sequencer \longrightarrow done_sequencer \longrightarrow Skip$

$TearDown \mathrel{\widehat=} tearDownCall \longrightarrow tearDownRet \longrightarrow Skip$

$\bullet\ Setup \mathbin{;}\ Execute \mathbin{;}\ TearDown$

end

Application Process

process *MainSafeletApp* $\mathrel{\widehat=}$ **begin**

$setUpMeth \mathrel{\widehat=} setUpCall \longrightarrow Skip \mathbin{;}\ setUpRet \longrightarrow Skip$

$tearDownMeth \mathrel{\widehat=} tearDownCall \longrightarrow Skip \mathbin{;}\ tearDownRet \longrightarrow Skip$

$Methods \mathrel{\widehat=} \mu X \bullet setUpMeth \mathbin{;}\ X$

$\bullet\ Methods \bigtriangleup tearDownMeth$

end

Formal Model of the Safelet

Composite Process

channelset *MainSafeletChan* $==$
$\{\!| \mathit{setUpCall}, \mathit{setUpRet}, \mathit{tearDownCall}, \mathit{tearDownRet} |\!\}$

process *MainSafelet* $\widehat{=}$
$(\mathit{SafeletFW} \llbracket \mathit{MainSafeletChan} \rrbracket \mathit{MainSafeletApp}) \setminus \mathit{MainSafeletChan}$
exposes *start_sequencer*, *end_sequencer*

Observations

- Framework and application process synchronise on channels for method calls.
- Framework channels that control other components are exposed.
  - We introduce a new documentary clause **exposes** above.
- Channels for methods exclusively called by the framework are concealed. ($\rightarrow$ To some extent enforces behavioural compliance.)

The Mission Sequencer Component

A Mission Sequencer object instantiates the missions of the application.

Java code for the MainMissionSequencer class

```
public class MainMissionSequencer extends MissionSequencer {
  private boolean mission_done;

  public MainMissionSequencer() {
    mission_done = false;
  }

  protected Mission getNextMission() {
    if (!mission_done) {
      mission_done = true;
      return new MainMission();
    }
    else {
      return null;
    }
  }
}
```

`getNextMission()` is called by the infrastructure to obtain next mission.

Formal Model of the Mission Sequencer

Framework Process

$$
\begin{array}{l}
\textbf{process } \mathit{MissionSequencerFW} \mathrel{\widehat=} \textbf{begin} \\
\mathit{Start} \mathrel{\widehat=} \mathit{start_sequencer} \longrightarrow \mathit{Skip} \\
\mathit{Execute} \mathrel{\widehat=} \mu X \bullet \mathit{getNextMissionRet}\,?\,\mathit{next} \longrightarrow \\
\quad \textbf{if } \mathit{next} \neq \mathit{null} \longrightarrow \mathit{start_mission}\,.\,\mathit{next} \longrightarrow \mathit{done_mission}\,.\,\mathit{next} \longrightarrow X \\
\quad [\!] \; \mathit{next} = \mathit{null} \longrightarrow \mathit{Skip} \\
\quad \textbf{fi} \\
\mathit{Finish} \mathrel{\widehat=} \mathit{end_mission_fw} \longrightarrow \mathit{end_sequencer_app} \longrightarrow \mathit{done_sequencer} \longrightarrow \mathit{Skip} \\
\bullet\ \mathit{Start} \,;\ \mathit{Execute} \,;\ \mathit{Finish} \\
\textbf{end}
\end{array}
$$

Observations

- Framework process has to terminate the application process.
- Only one channel required for the method call (?)
- Channels *start_mission* and *done_mission* are parametrised by a mission identifier: unique to the class.

Formal Model of the Mission Sequencer

Application Process

process *MainMissionSequencerApp* $\widehat{=}$ **begin**

state *MainMissionSequencerState*

$mission_done : BOOL$

$Init \widehat{=} mission_done := FALSE$

$getNextMissionMeth \widehat{=}$

$\quad \textbf{if } mission_done = FALSE \longrightarrow (mission_done := TRUE;$

$\qquad getNextMissionRet\,!\,MainMissionId \longrightarrow Skip)$

$\quad [\!] \neg (mission_done = FALSE) \longrightarrow getNextMissionRet\,!\,null \longrightarrow Skip$

$\quad \textbf{fi}$

$Methods \widehat{=} \mu X \bullet getNextMissionMeth\ (\Box \ldots)\ ;\ X$

$\bullet\ Init\ ;\ (Methods \triangle end_sequencer_app \longrightarrow Skip)$

end

→ A general pattern emerges for encoding application processes.

→ Traceable to the Java code i.e. `MainMissionSequencer` class.

Formal Model of Missions

Framework Process

process *MissionFW* $\widehat{=}$ **begin**

state *MissionFWState*

$mission : MissionId$
$handlers : \mathbb{F}\ HandlerId$

Init

$MissionFWState'$

$mission' = null \land handlers' = \varnothing$

$AddHandler \mathrel{\widehat{=}} \mathbf{val}\ handler : HandlerId \bullet handlers := handlers \cup \{handler\}$

$StartHandlers \mathrel{\widehat{=}} \mathop{|||}\ h : handlers \bullet start_handler\,.\,h \longrightarrow Skip$

$StopHandlers \mathrel{\widehat{=}} \mathop{|||}\ h : handlers \bullet stop_handler\,.\,h \longrightarrow Skip$

$DoneHandlers \mathrel{\widehat{=}} \mathop{|||}\ h : handlers \bullet done_handler\,.\,h \longrightarrow Skip$ ...

Framework Process – cont'd

$$
\begin{aligned}
&Start \mathrel{\widehat=} Init \; ; \; start_mission \, ? \, m \longrightarrow mission := m \\
&Initialize \mathrel{\widehat=} initializeCall \, . \, mission \longrightarrow Skip; \\
&\quad \mu X \bullet ((add_handler \, ? \, h \longrightarrow AddHandler(h) \; ; \; X) \\
&\qquad \Box \; initializeRet \, . \, mission \longrightarrow Skip) \\
&Execute \mathrel{\widehat=} StartHandlers \; ; \; activate_handlers \longrightarrow \qquad [\text{red} = \text{broadcast}] \\
&\quad (DoneHandlers \; ||| \; stop_handlers \longrightarrow StopHandlers) \\
&Cleanup \mathrel{\widehat=} cleanupCall \, . \, mission \longrightarrow cleanupRet \, . \, mission \longrightarrow Skip \\
&Finish \mathrel{\widehat=} end_mission_app \, . \, mission \longrightarrow done_mission \, . \, mission \longrightarrow Skip \\
&\bullet \; (\mu X \bullet Start \; ; \; Initialize \; ; \; Execute \; ; \; Cleanup \; ; \; Finish \; ; \; X) \\
&\qquad \triangle \; end_mission_fw \longrightarrow Skip \quad \textbf{end}
\end{aligned}
$$

Observations

- Continually offers to execute missions (recursion).
- Controls starting, activation and termination of handlers.

Application Process: `initialize()` method (not Standard *Circus*)

$$
\begin{array}{l}
initializeMeth \mathrel{\widehat=} initializeCall \,.\, \mathbf{MainMissionId} \longrightarrow Skip; \\
\quad (\mathbf{var}\ engine : \mathbf{ref}\ EngineClass; \\
\qquad throttle : \mathbf{ref}\ ThrottleControllerClass;\ \ldots \bullet \\
\qquad engine := ref(\mathbf{new}\ EngineClass); \\
\qquad EngineInitCall\,!\,engine \longrightarrow Skip; \\
\qquad add_handler \,.\, EngineHandlerId \longrightarrow Skip; \\
\qquad throttle := ref(\mathbf{new}\ ThrottleControllerClass); \\
\qquad ThrottleControllerInitCall\,!\,throttle \longrightarrow Skip; \\
\qquad add_handler \,.\, ThrottleControllerHandlerId \longrightarrow Skip\ ;\ \ldots); \\
\quad initializeRet \,.\, \mathbf{MainMissionId} \longrightarrow Skip
\end{array}
$$

Colour Legend

green highlights standard **Oh***Circus* constructs;
red highlights extensions needed for supporting references;
blue is used for framework events;
violet is used for method call and object initialise events.

Application Process: `requestTermination()` method

$$
\begin{array}{l}
requestTerminationMeth \mathrel{\widehat=} \\
\quad requestTerminationCall\,.\,\mathbf{MainMissionId} \longrightarrow \\
\quad\quad \mathbf{if}\ terminating = FALSE \longrightarrow \\
\quad\quad\quad terminating := TRUE\ ;\ stop_handlers \longrightarrow Skip \\
\quad\quad [\!]\ \neg\,(terminating = FALSE) \longrightarrow Skip\ \mathbf{fi}\ ; \\
\quad requestTerminationRet\,.\,\mathbf{MainMissionId} \longrightarrow Skip
\end{array}
$$

Observations

- Implementation is `final` thus cannot be redefined by the user.
- Application process communicates with the framework process via *stop_handlers*.
- Component *terminating* is part of the application process state.

→ Application model not entirely generic here, unless we introduce a notion of (process) inheritance.

Event Handler Components

Event handlers are either called periodically (PEHs) or in response to external events occurring in the system (APEHs).

Java code for the Engine class

```
public class Engine extends AperiodicLongEventHandler {
  private CruiseControl cruise; /* Handler aggregates shared data object. */

  public Engine(CruiseControl cruise) {
    super(...);
    this.cruise = cruise;
  }

  public void handleAsyncLongEvent(long event) {
    switch (event) {
      case ENGINE_ON:
        cruise.engineOn(); break;

      case ENGINE_OFF:
        cruise.engineOff(); break;
    }
  }
}
```

Formal Model of Aperiodic Event Handlers

Framework Process for the `Engine` class

process $EngineFW \mathrel{\widehat=}$ **begin**

state $EngineFWState$
$active : BOOL$ (Do we executes as part of the current mission?)

$Init \mathrel{\widehat=} [EngineFWState' \mid active' = FALSE]$

$StartHandler \mathrel{\widehat=} start_handler \,.\, EngineHandlerId \longrightarrow$
$\quad activate_handlers \longrightarrow active := TRUE$

$DispatchHandler \mathrel{\widehat=} \mu X \bullet (engine_on \longrightarrow$
$\quad handleAsyncEventCall \,.\, EngineHandlerId \,!\, EngineOnEventId \longrightarrow$
$\quad handleAsyncEventRet \,.\, EngineHandlerId \longrightarrow Skip) \;\Box$
$\quad (engine_off \longrightarrow \ldots) \,;\; X$

$StopHandler \mathrel{\widehat=} stop_handler \,.\, EngineHandlerId \longrightarrow active := FALSE$

$\bullet\; (\mu X \bullet Init \,;\; (StartHandler \;\Box\; activate_handlers \longrightarrow Skip);$
$\quad \textbf{if}\; active = TRUE \longrightarrow (DispatchHandler \;\triangle\; StopHandler)$
$\quad [\!]\; active = FALSE \longrightarrow Skip\; \textbf{fi} \,;\; X) \;\triangle\; end_mission_fw \longrightarrow Skip$

end

Formal Model of Periodic Event Handlers

Framework Process for the `ThrottleController` class (extract)

$$ReleaseHandler \mathrel{\widehat=} \mu X \bullet (release_handler\,.\,ThrottleControllerHandlerId \longrightarrow Skip \;\mathbf{deadline}\; 0)\,;\; (\mathbf{wait}\; TCPeriod)\,;\; X$$

$$DispatchHandler \mathrel{\widehat=} \mu X \bullet release_handler\,.\,ThrottleControllerHandlerId \longrightarrow (handleEventCall\,.\,ThrottleControllerHandlerId \longrightarrow Skip \;\mathbf{deadline}\; TCReleaseJitter)\,;\; (handleEventRet\,.\,ThrottleControllerHandlerId \longrightarrow Skip \;\mathbf{deadline}\; TCDeadline)\,;\; X$$

$$PeriodicDispatch \mathrel{\widehat=} (ReleaseHandler \;[\![\, \varnothing \mid \{\!|\, release_handler \,|\!\} \mid \varnothing \,]\!]\; DispatchHandler) \setminus \{\!|\, release_handler \,|\!\}$$

Observations

- An additional event (*release_handler*) models release of the timer.
- We need *Circus* Time to specify the deadline constraints.
  - "**wait** *t*" and "*a* **deadline** *t*" actions.
- Omitted parts of the process are similar:
  → Except we use *PeriodicDispatch* instead of *DispatchHandler*.

Some Conclusions on the Formal Model

- Main challenge is to find the right compromise between
  1. Simplicity of the model, and
  2. faithfulness to the SCJ infrastructure and Java language.

  → This is a non-trivial and painstaking task.
- The formal model has constantly evolved.
  → It is likely to evolve even more in the future.
- It only copes with one object of a class existing at a time.
  → Reuse of class processes is possible though.
- No notion of sound memory utilisation yet.
- Generalisation is possible but comes at a cost. We consider:
  1. Support multiple instances of the same class type;
  2. Favour data objects as identifiers as opposed to explicit ids.
- There is need for a highly-integrated variant of *Circus*.

Validation in FDR

To validate the model, we used the CSP model checker FDR.

How did we use FDR?

- The *Circus* model was manually translated into a CSP model.
- FDR provided everything we needed except for timed constructs.
- State components of *Circus* processes were encapsulated in CSP processes. Some simplifications to keep complexity at bay.

What properties have been checked?

1. Deadlock-freeness of individual and composite components.
   - Does not always hold but can give insight into potential problems.
2. Termination of individual components and composite components.
   - Does not always hold but can give insight into potential problems.
3. Deadlock-freeness of application scenarios.
4. Termination of application scenarios.

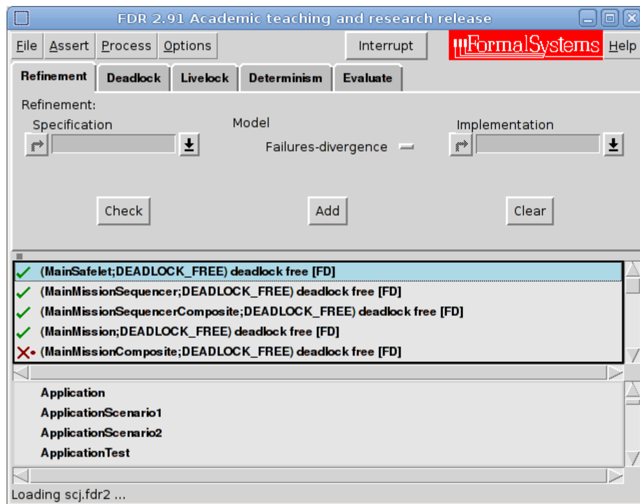
Validation Results

Examples of found Issues

- Unlike *SafeletFW* and *MissionSequencerFW*, the *MissionFW* framework process needs to be externally terminated. Easy one!
- Initially we only had *Call* but no *Ret* channels.
  → In some cases we can get away with it, but for the `setUp()` and `tearDown()` methods of the `Safelet` we need them both.
- Termination could be requested too soon, i.e. before the mission has been fully initialised, resulting handlers not being terminated.
  → Led to a change in using a broadcast event to active handlers.
- Calling `requestTermination()` twice resulted in a deadlock.
  → Corrected by modifying the respective *Circus* action.
- Interrupt may stop a handler in the middle of executing a method.
  - Causes a deadlock because the method never engages in event for the method return. → Revise model replacing the interrupt.

→ Some of these are indeed difficult to spot by review only.

FDR Demonstration



Show a small demo if time.

Future Outlook

Where do we go from here?

- Further exploit model-checking by creating a suite of test scenarios.
  → So far only a few have been specified...
  → Maybe try and use more complex scenarios.
- We need to look at further case studies.
  → Might reveal additional features that need to be considered.
- Formalise the translation of SCJ applications into *Circus* models, and vice versa.
- Automatic generation of the application-specific processes.
  - I proposed an MSc project, hope someone will be brave enough.
- Development our new *Circus*$^{\#}$ language (syntax and semantics).
- A refinement strategy for transforming abstract specifications into concrete models of SCJ programs.

Collaborative effort: Jim, Ana, Andy, Frank, Kun, Chris, >**?**<

Conclusion

- We gave a formal account of the SCJ Mission model.
- To our knowledge it is the first attempt in formalising SCJ.
- We are not aiming to model Java in its entirety, but a conceptual language underlying the paradigms of SCJ.
- The work highlights the need for highly-integrated languages (*Circus*$^{\#}$) and reasoning frameworks.
- The UTP (**U**nifying **T**heories of **P**rogramming) shall provide the basis for achieving integration on a semantics level.
- The wider objective is to develop a "correctness-by-construction" approach for developing verified SCJ programs from specifications.
- Ideally this should all happen in a semi-automatic fashion.

We have a long way to go...

Thank you for your attention.

Any Questions?