Applying Formal Timing Analysis to Satellite Software

Agenda

- Introduction and Motivation
- Schedulability Analysis
- Architecture & Engaged Toolchain
- Lessons Learned
- Missions, Summary & Outlook
Introduction and Motivation – Satellite Software

- Central software item that glues together all subsystems
  - sensors, actuators, mechanisms, power …

- Control loops, including
  - Attitude and Orbit Control
  - Thermal Control

- Telecommand and Telemetry Interfaces
  - Full commandability and visibility

- Robustness
  - Failure Detection, Isolation and Recovery
  - Hardware redundancy

- Payload operation
  - Very large spectrum of software functions and performance needs

→ Large variety of functions with high reliability requirements
Introduction and Motivation – Mission Critical S/W

• Increasingly large projects
  • Complexity, Size, Cost, Time

• Testing software is very important but
  • Impossible to cover all cases

• Introduction of formal methods
  • Absence of runtime errors, Memory / Stack size analysis …
  • Timing Requirements

→ Schedulability Analysis

SA Required for flight software qualification by the [ECSS-E-40-1]
How to perform a Schedulability Analysis?

1) Timing analysis of code snippets

2) Schedulability calculation
   - Tasking Model
   - Mathematical Theory

3) Check with requirements

→ Analysis Model Complexity increases with Software/System Complexity
Static Architecture

Static layered set of interactive components

Model-Driven Approach /w design pattern and partial code generation from UML

→ C language implementation
Real-Time Architecture

Single Core
Fixed Priority Preemptive Multitasking RTOS (RTEMS)
Immediate Priority Ceiling Protocol

Hard Real-Time
- Sporadic functionality
- Periodic functionality

Soft & Non Real-Time
- Majority of Code

Increasing Fixed Priority

Real-Time Tasks
- Loop Task
- Helper Tasks
- Main Task
- Background Tasks

Cyclic Executive of Periodic Actions
Cooperative Scheduling of Event Triggered Actions

Cooperative Non-Preemptive Event-Based Scheduling

Hard-Coded Cycle
- Critical Task Set
- Non-Critical Task Set
### Timing Analysis

**WCET Analysis of Code Snippets**

(subject to hard real-time constraints)

- Static code analysis using AbsInt $a^3$ suite
- Safe upper bounds for the WCET
- Manually added annotations

```c
@staticWCET
exported uint32 myFunction2(uint32 l) {
    uint32 x = 0;
    if (l > 1000) {
        l = 1000;
    } if
    for (uint32 i = 0; i < l; i++) {
        x = x + i;
    } for
    return x;
} myFunction2 (function)
```

Alternatives:
- Rapita Systems
- Bound – T
Schedulability Calculation and Check

Response Time Calculation

- For simple tasking models: MS Excel and VB scripts
- For more elaborate models: In-house DSL

Determined WCET

Timing Requirements

Req. AOCS-210:
Execution of the AOCS step function and its pre- and postprocessing shall be completed within 200ms after the 1Hz activation pulse.

Alternatives:
MAST, Cheddar Symtavision

Calculation, Visualization and Documentation

«CyclicPulse» periodic_10Hz @10.0Hz
«ISR» Periodicals @10.0Hz
«Task» PeriodicAOCOS @2.0Hz

Timeline of Stimulus periodic_10Hz
Experiences and Lessons Learned

- Software Development Lifecycle
- Software Requirements
- Software Architecture
- Software Coding / Analysis Tools
Experiences and Lessons Learned
-- Software Development Lifecycle

“Here is my software,
it’s ready,
it’s tested,
and we can’t change it anymore,

→ please prepare the timing analysis report until tomorrow EOB !”

Ok, that’s a bit exaggerated, but:

It’s about the mindset:
people are focused on function rather than on timing
Experiences and Lessons Learned
-- Software Requirements

Non-functional (timing) requirements are essential.
→ You have to know your needs early to make the correct design decisions
→ You have to be able to verify/validate/test your system in the end

From experience:

• Timing requirements are usually not of high quality and incomplete.
  • Requirement documents
  • User manual and ICDs of used physical devices

→ A complete set of high quality formal timing requirements is essential
Experiences and Lessons Learned
-- Software Architecture

Software Architecture ↔ Computational Model, Mathematical Theory
(Response Time Calculation)

Some common design/implementation pattern are in conflict with assumptions or efficient application of the theory!

- Great source for discussions
- Know the assumptions → And show that they are obeyed!
- Design for Analyzability

→ Design the architecture according to timing needs and keep it simple!

But, we have always done it like this!
Experiences and Lessons Learned
-- Software Architecture

Software Modularity

Common Pattern:
• Define modules according to functionality
• Modules are independent from each other
• Support module reusability
→ Each module defines a set of tasks

Efficient Application of Theory:
• Keep task interaction scenarios simple
→ Limit the set of tasks

Reduce number of tasks and their communication
Experiences and Lessons Learned
-- Software Architecture

Software Modularity

- When first establishing Timing Analysis, we
  - Reduced the number of tasks from 50 to 10 (5 being subject to hard constraints)
  - Reduced the monitors (to protect critical sections) from 50 to 15

- Functionality is assigned to tasks according their timing needs

<table>
<thead>
<tr>
<th>Temporal Characteristics</th>
<th>Aperiodic</th>
<th>Periodic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>TC reception, …</td>
<td>AOCS &amp; Thermal control loops, …</td>
</tr>
<tr>
<td>Soft</td>
<td>TM generation, …</td>
<td>Monitoring of parameter values, …</td>
</tr>
</tbody>
</table>

- Software Modularity (for reuse)
  - Passive modules only according to pattern
  - Specific tasks that invoke operations
Experiences and Lessons Learned
-- Software Architecture

Self-Suspending Tasks

Common Pattern:
• Tasks suspend during resource accesses
• Suspensions may be hidden to provide a “user-friendly” API (Libraries, Legacy Code, Operating System …)
• Suspensions to implement temporal sequences (wait statement)

→ Invisible delays and additional (hidden) ISR invocations
→ Unknown (but hopefully bounded) delays

Assumption of Theory:
• Each task has a single point of suspension

Eliminate task self-suspensions in tasks subject to hard real-time constraints.
Experiences and Lessons Learned

-- Software Architecture

Self-Suspending Tasks

- Introduction of “Helper Tasks” executing self-suspending
  - I/O accesses
  - Low-Level Driver API calls
  - (+ Globally account for “hidden” ISRs)

- Temporal sequences are implemented in event-driven state machines

Real-Time Tasks

Loop Task

Cyclic Executive of Periodic Actions

Helper Tasks

Main Task

Cooperative Scheduling of Event Triggered Actions

Critical Task Set

Non-Critical Task Set
Experiences and Lessons Learned
-- Software Architecture

Data flow constraints

Common Pattern:
- A single task prepares data items for different data sinks
- Some data is required before the task completes

Theory:
- Doesn’t specifically address this need

Augment the analysis model with “virtual” tasks and “internal” deadlines.
Experiences and Lessons Learned

-- Software Architecture

**Data flow constraints**

- Define Internal Deadlines

\[
D_B \leq R_{AB} \\
D_C \leq R_{ABC}
\]

- Calculate response times of task until respective data is prepared
- And compare with respective deadline
Experiences and Lessons Learned

-- Coding Rules

Coding Rules $\leftrightarrow$ (static) WCET Analysis

Mostly WCET Analysis Tool Specific

... details not addressed in presentation

• Great source for discussions
• Know the peculiarities of the WCET tool

Mind-set

• Algorithms: Worst Case Time (WCET) vs Average Time (AET) $\rightarrow$ loops!
• Avoid dynamic structures and data dependent control flow

$\rightarrow$ Code for efficient WCET analyzability!
The presented analysis has been and is carried out successfully in multiple missions:

- Small Geo Platform
  - Hispasat AG1 Telecom Satellite
  - European Data Relay Satellite – C
- Galileo
- Meteosat 3rd Generation

... Method will be applied to all future missions ...
Evaluation and Summary

Is it worth doing Formal Timing Analysis?

→ YES
-- We have to do it as it is part of the Flight Software Qualification
-- We can do it as the technology is available

Is it worth spending all the effort and money?

→ Well, we’ll have to become way more efficient
-- and there are many options to improve in
Experiences

• Engineers (Requirements and Software) are not used to consider timing essential
  → Intensify training, Increase awareness

• Modern WCET analysis tools are complex to use correctly
  → Ease utilization by restricting the programming language (MBSE, pattern …)

• Tools and analysis are not (yet) well integrated in the workflow
  → We want a “Push-Button Solution”

• SA and static WCET analysis provides orthogonal view on the software
  → Nice Side-effect … helps us finding bugs early
Outlook – future processor hardware

Right now:
Leone2 single core @ 65Mhz under maximum load conditions
30% CPU utilization

If more processing power is needed:
• Distributed Systems
  • Increasing Interest by Systems Engineering

• Multi-Core Processors
  • First hardware is on the (Space) Market

→ Flight Software Engineering should get ready to address these upcoming challenges

→ Need input from academia and tool vendors
Thank you!