Architectural Analysis of Dynamically Reconfigurable Systems

Technical Presentation

Presenter: Dharma Ganesan

Principal Investigator (PI): Dr. Mikael Lindvall, CESE
NASA POC: Sally Godfrey, GSFC
Team members:
Dr. Chris Ackermann, Dr. Arnab Ray, Lyly Yonkwa (CESE)
GMSEC, CFS, RBSP
In collaboration with Fraunhofer Institute for Experimental Software Engineering (IESE)

© 2010 Fraunhofer USA, Inc.
Center for Experimental Software Engineering
Background

• Many NASA projects use flexible architecture styles for
  – creating loosely coupled systems
  – minimizing future software change

• Examples of such systems:
  – Goddard Mission Services Evolution Center (GMSEC)
    • A reusable framework for ground systems
  – Core Flight Software (CFS)
    • A reusable framework for flight systems
Problem

- Increased flexibility of architectural styles decreases analyzability
- Behavior emerges and varies depending on the configuration
- Does the resulting system run according to the intended design?
- What architectural decisions impede or facilitate testing?
Top Down Approach

• Architecture analysis
  – focusing on critical components’ behavior data
  – visualizing architecture relevant events
  – drilling down to details as necessary

• Detect defects and deviations
  – modeling, comparing planned vs. actual behavior

• Architecture and its testability
Currently Targeted Projects: GMSEC and CFS

- Reusable framework for ground and flight systems
- GMSEC and CFS systems are running at FC-MD
- Confirmed defects/violations reported in several papers

Some example results
Analyzing Software Architectures

Dynamic Save

Component A  Component B  Software Bus

Run-time Events difficult to analyze because There are too many low level events

New tool

Component A  Component B

Goal

Push/Pull

New tool can detect architecture relevant events and hide irrelevant information

No static dependencies!  \(\rightarrow\) static analysis is not sufficient

CESE

Fraunhofer

Analyzing Software Architectures

Component A  Component B  Software Bus

No static dependencies!  \(\rightarrow\) static analysis is not sufficient

Dynamic Save

Component A  Component B  Software Bus

Run-time Events difficult to analyze because There are too many low level events

New tool

Component A  Component B

Goal

Push/Pull

New tool can detect architecture relevant events and hide irrelevant information
Analyzing Runtime Events

• Problems
  – different events are of interest
  – events can occur in any order
  – huge number of events
  – range between events might be very large

Solutions: Goal-oriented data collection and a pattern recognition engine
Actual Architecture Recognition

Planned architectural styles:
E.g. Pipe & Filter, Publish Subscribe

Runtime events:
init,timestamp=1264620606308,constructor=v1.MergeFilter,instanceid=obj578ceb
call,timestamp=1264620606317,methodname=java.io.PipedReader.read,callee=obj9ed927,caller=objfa7e74,argument=null

Rules:
Filter:
The constructor name of a Filter contains “Filter”

Push:
The callee of a method call is a “PipedWriter” instance,
the name of the method is “write”,
the caller is an Instance of Filter
This diagram was automatically created by Dynamic SAVE using run time information from GMSEC.

Problem: Much information, but GMSUB component that receives messages missing!
Sample output from new approach

This diagram will be automatically created by the new approach using the same run time information from GMSEC.

Only critical messages visible, all else hidden.

Pattern engine matched pairs of messages and reduced the information significantly!

Unexpected Duplicate message!
Sample output from new approach …

This diagram will be semi-automatically created by the new approach using the same run time information from GMSEC.
Taking message timing delays into account

Timing is off!
Sent before but arrives after!

The slopes indicate message delays that may impact behavior

This diagram will be automatically created by the new approach using the same run time information from GMSEC
We analyze the CFS architecture and its unit testing architecture

Focus of the analysis:
- What architectural decisions impede or facilitate testing?
Some Recommendations for improved testability

• Modules should be programmed to abstract interfaces
  – mock implementations of interfaces for unit testing

• Some internal details of modules should be public – cannot “hide” everything

• Avoid using the same return code of functions for different scenarios
Abstract Interfaces and Testability – CFS example

```c
Software Bus (SB)

<table>
<thead>
<tr>
<th>linux/osapi.c</th>
<th>rtems/osapi.c</th>
<th>vxworks6/osapi.c</th>
<th>Test/ut_osapi_stubs.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 OS_QueuePut(...)</td>
<td>int32 OS_QueuePut(...)</td>
<td>int32 OS_QueuePut(...)</td>
<td>int32 OS_QueuePut(...) {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>\  // Mock Implementation</td>
</tr>
<tr>
<td>sendTo(...)</td>
<td>...</td>
<td>rtems_message_queue_send(...);</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>msgQSend (...)</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
```

```c
int32 OS_QueuePut(...) {
// Mock Implementation
}
```
Open some internal details – CFS example

```c
int32 CFE_ES_LoadLibrary(char *EntryPoint, char *LibName, ...) {
    boolean LibSlotFound = FALSE;
    for (i = 0; i < CFE_ES_MAX_LIBRARIES; i++) {
        if (CFE_ES_Global.LibTable[i].RecordUsed == FALSE) {
            LibSlotFound = TRUE;
            break;
        }
    }
    if (LibSlotFound == FALSE) return CFE_ES_ERR_LOAD_LIB;
}

/* Test for loading more than max number of libraries */
for (j = 0; j < CFE_ES_MAX_LIBRARIES; j++) {
    CFE_ES_Global.LibTable[j].RecordUsed = TRUE;
}
Return = CFE_ES_LoadLibrary("EntryPoint","LibName", ...);
UT_Report(Return == CFE_ES_ERR_LOAD_LIB, "CFE_ES_LoadLibrary",
          "No free library slots");
```
We’re building a new approach that
– helps understand, visualize, and validate
  software systems that use loosely coupled
  architecture styles
– helps evaluating testability of the architecture

Next steps
– refine software tools and method, apply also
  to other NASA systems
Acronyms

- AFRL – Air Force Research Laboratory
- APL – Applied Physics Laboratory
- ARC – Ames Research Center
- CESE – Center for Experimental Software Engineering
- cFE – core Flight Executive
- CFS – Core Flight Software
Acronyms (2)

- CHIPS - Cosmic Hot Interstellar Plasma Spectrometer
- CLARREO - Climate Absolute Radiance and Refractivity Observatory
- COTS – Commercial Off-The-Shelf
- DSILCAS – Distributed System Integrated Lab Communications Adapter Set
- Dyn-SAVE – Dynamic SAVE
Acronyms (3)

- GLAST - Gamma-ray Large Area Space Telescope
- GMSEC – Goddard Mission Services Evolution Center
- GOTS – Government Off-The-Shelf
- GPM - Global Precipitation Measurement
- GSFC – Goddard Space Flight Center
- IV& V – Independent V & V
Acronyms (4)

- JSC – Johnson Space Center
- LADEE - Lunar Atmosphere and Dust Environment Explorer
- LDCM - Landsat Data Continuity Mission
- LRC - Langley Research Center
- LRO - Lunar Reconnaissance Orbiter
- MMOC – Multi-Mission Operations Center
- MMS - Magnetospheric MultiScale
Acronyms (5)

• MSFC - Marshall Space Flight Center
• RBSP – Radiation Belt Storm Probes
• SAVE – Software Architecture Visualization and Evaluation
• SDO – Solar Dynamics Observatory
• TRMM – Tropical Rainfall Measuring Mission
• V & V – Verification and Validation