Architecture Abstraction Plugin  
User Documentation

Julien Delange [jdelange@sei.cmu.edu](mailto:jdelange@sei.cmu.edu)

# Introduction

Using architecture abstractions help to understand the underlying software architecture and find potential defects. AADL is a language to design software architecture that is appropriate to analyze it and find potential defect and issues. We analyze common architecture patterns that could potentially increase software complexity and make the software design and implementation more difficult.

In the following, we identify several architecture patterns that increase software complexity but also incur rework. We then present out dedicated analysis plugin to analyze architectures, find defects and generate reports to suggest how to get rid of them and improve the software architecture.

# Architecture Patterns

## Pattern1: Global Variables across tasks

One common issues is the use of global variables between tasks. Having global variables increase the tasks dependencies with an outside element but also increase the variable scope, enabling potential side effects from other components.

The suggested improvement consists in using communication ports (data ports, event data ports)

* **Severity**: important
* **Category**: communication flow
* **Benefits**: increase reusability, avoid side-effect from other component, ease validation/verification

|  |  |
| --- | --- |
| Task 1  Task 2  Shared Data | Task 1  Task 2 |
| Architecture Pattern | Proposed Change |

## Pattern2: Unprotected global variables (use of semaphores or mutexes)

As for the previous architecture pattern, one issue is to have a shared data with multiple writers that is not protected against mutual exclusion. Having appropriate protection mechanism from mutual exclusion is of primary important when using global variables. We then detect when data does not use a protection mechanism and suggest to use one.

* **Severity**: warning
* **Category**: communication flow
* **Benefits**: avoid unwanted concurrent data modification

## Pattern3: Periodic harmonic tasks in the same process

When having two tasks in the same process that have the same period, it can be a good idea to refactor the process and merge the task into a single one. This would then reduce the number of tasks in the system, avoid execution costs related to scheduling and context switches and ultimately, make the system verification easier.

* **Severity**: recommendation
* **Category**: design
* **Benefits**: performance and easier verification

|  |  |
| --- | --- |
| Task 1 (100ms)  Task 2 (100ms)  Process 1  Task 3 (250ms) | Task 1 + 2 (100ms)  Process 1  Task 3 (250ms) |
| Architecture Pattern | Proposed Change |

## Pattern4: Generic Data Types

Using generic data types might introduce safety issue (assigning a bad value – e.g. out of range - to a variable). A good software engineering practice consists in defining specific data types for the architecture.

* **Severity**: important
* **Category**: design
* **Benefits**: lower tests, increase software verification

## Pattern5: Communication queues (queueing ports)

If a system does not put correct dimensions in queue size when using queueing mechanism, it would then lead to data loss. Instead, the architecture must dimension the data queues according to sender/receiver periods.

|  |  |
| --- | --- |
| Task 1  (100ms)  Task 2 (200ms)  Queue size = 1 | Task 1  (100ms)  Task 2 (200ms)  Queue size = 2 |
| Architecture Pattern | Proposed Change |

* **Severity**: important
* **Category**: communication flow
* **Benefits**: determinism, no loss of data

## Pattern6: Communication rates (sampling port)

While the previous example focus on queueing communication, the following will detail the architecture pattern related to sampling ports (data ports). When using such ports, if no data loss is acceptable, the receiver period must be smaller than the sender. It will then ensure that no data from the sender is lost.

|  |  |
| --- | --- |
| Task 1  (100ms)  Task 2 (200ms) | Task 1  (100ms)  Task 2 (<100ms) |
| Architecture Pattern | Proposed Change |

* **Severity**: important
* **Category**: communication flow
* **Benefits**: determinism, no loss of data

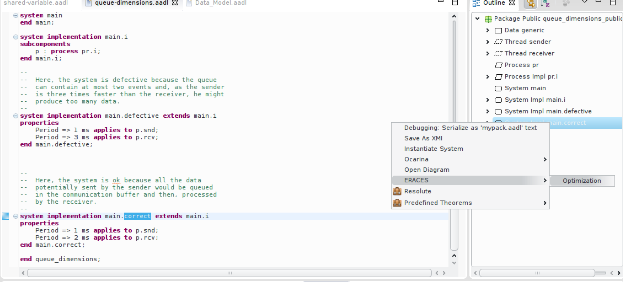
## Pattern7: Naming Policy

The naming policy of variables and communication ports can show bad design decisions. For example, this is common to communicate the state of a components through multiple variables named “is\_XXX” (for example, is\_open and is\_closed when designing a door). But this should be captured with a single enumeration types that can have different values. The naming policy can then help to detect design defect (communication the states in multiple variables, sending a command using several Boolean flags, etc.). Fixing these issues would then reduce the number of variables and makes software analysis and verification better.

* **Severity**: important
* **Category**: design
* **Benefits**: fewer tests

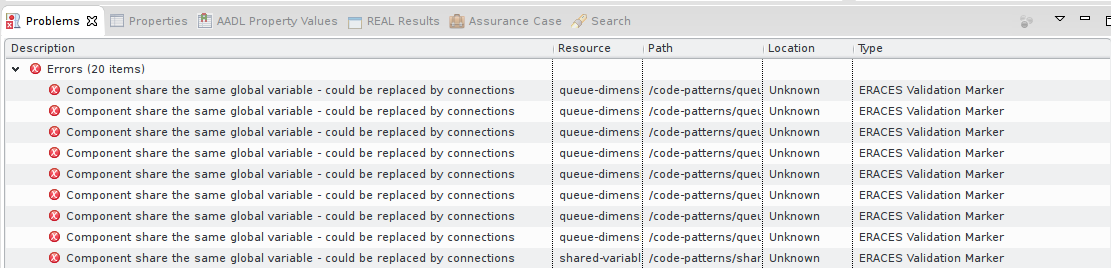
# ERACES Architecture Analysis Plugin

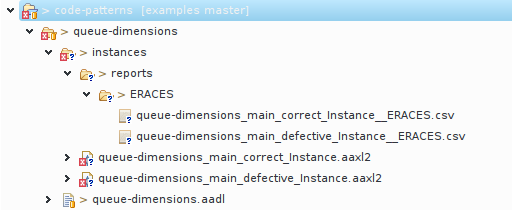
An implementation of an analysis plugin has been done to validate all the rules. To use it, please select the system instance to analyze, right click on it and select the ERACES Optimization option.



When the analysis is completed, it reports the errors using two mechanisms:

* **Error markers**: in the Eclipse view, markers are added to the system instace under analysis
* **Analysis Report**: in the instances/reports/ directory, a new report is generated. It summarizes all the different errors found during the analysis. The report is written in a CSV file that can be opened with any spreadsheet program (Excel, Libreoffice, etc.)





# Copyright and Acknowledgments

Copyright 2015 Carnegie Mellon University  
  
This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.  
  
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.  
  
This report was prepared for the  
SEI Administrative Agent  
AFLCMC/PZM  
20 Schilling Circle, Bldg 1305, 3rd floor  
Hanscom AFB, MA 01731-2125  
  
NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.  
  
This material has been approved for public release and unlimited distribution except as restricted below.  
  
Internal use:\* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.  
  
External use:\* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.  
  
\* These restrictions do not apply to U.S. government entities.  
  
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.  
  
DM-0002673