MSL Code Analysis

Jake Cox

Agenda

- Planning
- Syntactic Analysis
  - in-house static analysis
  - autocoders
- Semantic Analysis
  - asynchronous inter-process communication (IPC) mechanisms
  - continued improvement of the semantic analysis and associated IPC challenges.
  - augmented code analysis work (workstation test set, Doxygen, profilers)

This presentation discusses MSL IV&V challenges, approaches, successes and lessons learned associated with analysis of MSL code.

Summary of Results

- Each of the two techniques revealed issues that the other technique could not
- Requirement centric approach focused the code reviews
- Auto generated code contained far fewer errors than hand generated code

IV&V analysis of MSL Autogenerated Code

- ***The trust we have with mature autocoders cannot be assumed with an in-house product, and require syntactic as well as semantic IV&V ← We are establishing specific means for analysis on current build 6.1, complete by 7.0***
  - IV&V typical analysis for mature autocoders uses extensive design analysis of the input model, and semantic code analysis. Mature tools/generators are generally free of syntactic errors.
- MSL project is using 8 in-house autocoders, 6 are “first use” on MSL
  - 75% MSL code is generated from these autocoders

Category	MSL Autocoder	Comments
State Machine Generators		Utilizes a MagicDraw graphical input, with legacy.
		Uses a textual input and legacy from technology project. Output is C.
Parameters, cmd/tlm, etc		Each autocoder performs or supports one of the following five functions: Parameter handling, Commanding, Telemetry, Event Reporting, Data Product Generation These autocoders utilize actual xml or xml-like inputs. Outputs are c-code, xml files, some csv files are are used for FSW code, and/or Ground interfacing databases, and/or inputs to other autocoders.
Instrument Interface	Uniquely, code from this autocoder can be hand modified. This code is treated as hand code. Output is C code.	

Comparing Human and Autogenerated Code

	Similarities	Differences
IV&V Code Analysis Objectives	Both Human and Auto-generated code is evaluated to ensure code is error free, answers the three questions, and implements the specification	
Semantic Analysis	Semantic errors can be introduced through an incorrect design that is correctly coded IV&V analysis occurs by mapping and analysis of requirements/ design to code A correct design can be incorrectly coded	Autogenerated code: readability standards are not necessarily enforced and code reviews are not typically used. Code generator outputs can be complex and/or cryptic Autocode generator inputs (incorrect spec development) or generator can introduce semantic errors
Syntactic Analysis	Static code analyzers are an excellent means to detect syntactic errors, and often are indicators of semantic errors	For mature code-generators (e.g. Simulink RTW), the resulting code is often free of syntactical errors. This may not be the case with less mature analyzers Autogeneraters are consistent in ways that a human is not. An autogenerator would repeat a mistake with similar input (facilitates identifying defects in the generated code, and possibly the generator itself) Complexity introduced with the use of multiple generators

Syntatic Analysis

Syntactic Analysis

- Overview
- Process
  - Preparation
  - Tools
  - Analysis
- Results
- Auto-generated Code

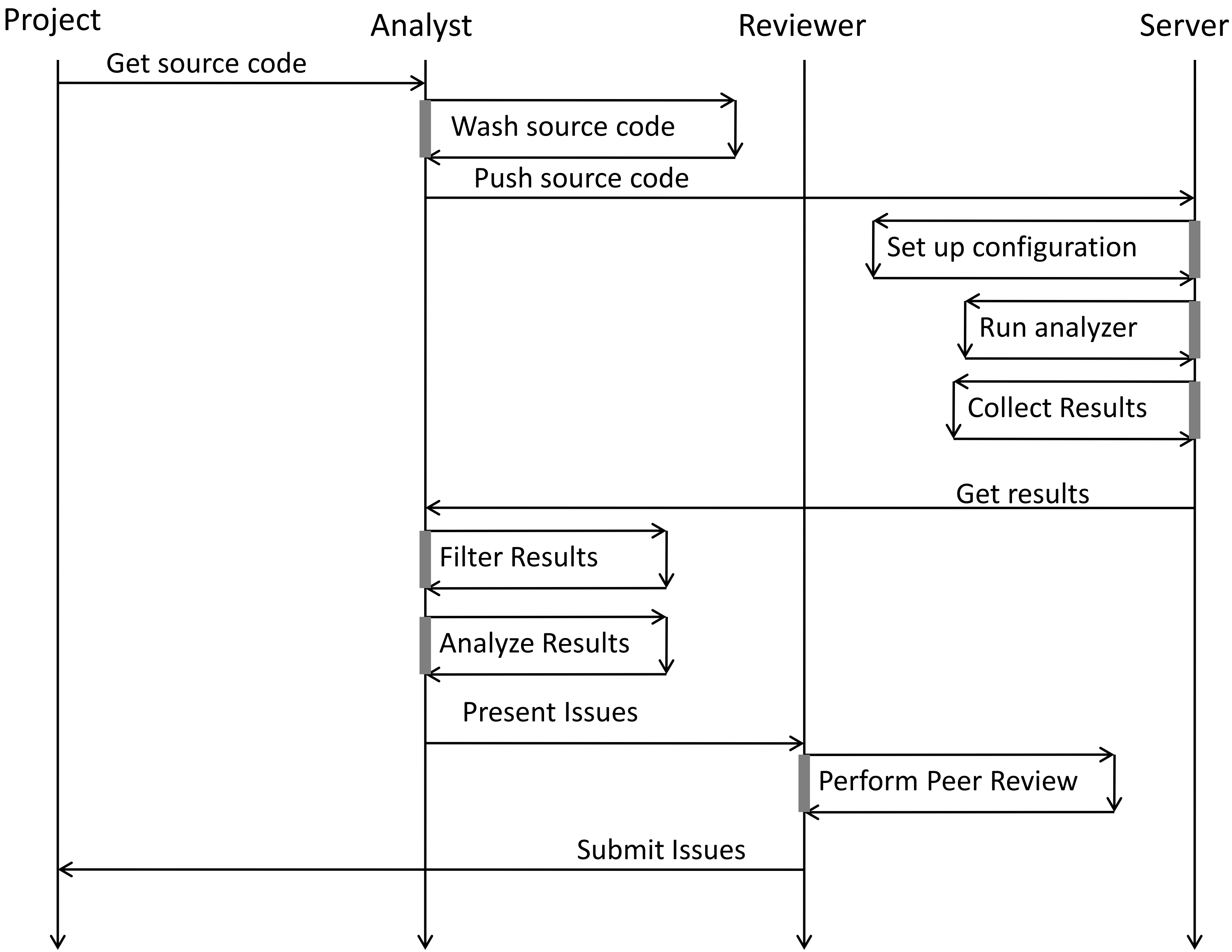
Overview

Tool based code analysis is a way to syntactically verify source code from the projects.

- Insure language correctness
- Syntactical Correctness is the base layer of verification
- Examples
  - A string is large than the buffer
  - Pointer variable checked for null after use
  - Assigned values that are not used later
- Requires strong knowledge of the language

Process

- Receive source code from the project
- Prepare it for running through the tool
- Run the tool to get warnings
- Filter the results
- Analyze the warnings
- Generate Issues


Project
Analyst
Reviewer
Server
Get source code
Wash source code
Push source code
Set up configuration
Run analyzer
Collect Results
Get results
Filter Results
Analyze Results
Present Issues
Perform Peer Review
Submit Issues

Prepare Source for the Tool

- Identify Header paths
- Identify Global defines
- Identify Real-time OS Target

Tools

- Code Sonar
  - requires a full build
- Flexelint
  - Simple to set up
  - One source file at a time
- Inspect
  - More difficult to set up
  - One result file

Tool Setup (Configuration File)

- The tools need to know
  - Where are the header files?
  - What variables/macros have been defined?
  - Where is the source?

Both Flexelint and Inspect allow these to be defined in a configuration file.

Filter the Results

- The tools produce more warnings than can be reasonably analyzed.
- Many warnings will rarely if ever be true issues.
- These warnings are removed from the result-set prior to analysis.

Tool Results

- Initial results.
  - 1566 KB; InSpect / SDO gce
  - 6769 lines
- After filtering.
  - 285 KB
  - 1581 lines

Analyze Each Warning
(The long slog)

- Import into Excel which allows sorting
  - Category
  - Text of the source
  - File name and line number
  - Description

  Requires good record keeping on work accomplished.

Peer Review and Write Issues

- Some projects complete analysis before and then peer review the entire set of results prior to writing issues.
- Others have written issues as they are found and have peer review prior to submission.

Syntactic Analysis of Build 7

- ~25 issues
- ~1.3 million lines of code

- 29,997 Klocwork Inspect warnings

Pointer not checked for null before being used in Module.

In the process_dma_in_context () function of module_prot.h, on line 198, the pointer handler is used without being first checked for null. On line 209 in the same function, handler is checked for null. Thus it is de-referenced and then later checked to determine if it is valid.

```
196  inline static I32 process_dma_in_context(ModuleHandler *handler)
197  {
198      CmdBuffer *cmd = handler-> dma_buf[buf_id];
.
.
.
209  FSW_ASSERT( handler != NULL );
```

Possible call of strlen() on a null string

In the function parse_params() in global.c, on line 7700 it is possible that strlen() will be called on a string set to null.

```
7453   Type * parse_params ( … )
…
7509      for (gindex = 0; gindex < number_of_groups; gindex++) {
…
7642         for (pindex = 0; pindex < parms_entries; pindex++) {
…
7700            if (len >= (I32) strlen (pentry->pname) &&
7701               strncmp (pentry->pname,
7702                                   name,
7703                                   strlen (pentry->pname)) == 0 ) {
```

Interesting Loss of Precision

```
bar.c

int8 foo();

int16 bar() {
        int16 var = foo();

        return var;
}
```

```
foo.c

int16 foo() {
        return 0x3799;
}
```

What is the value of var?

Hint: it is not 0x3799

Raw Results – Static Analysis, Inspect Tool

Errors from InSpect
Sorted by autocode generator, or handcoded

When errors put into a pivot table, the grouping of error types by autocoder are apparent.

Overall, initial results indicated MSL code is very clean.

Hand-code

Warning Type	AC1	AC2	AC3	AC4	Handwritten	AC5	AC6	AC7	AC8
ABR	36				412			34	
ABV.STACK					12				
ASSIGCOND.GEN					2				
FUNCRET.GEN					4	3			
IF_CYCLE	2				11	2		5	
IF_DEF_IN_HEADER_DECL					148				
IF_DEF_IN_HEADER_EX					183			1	
IF_DUPL_HEADER					2				
IF_MISS_DECL	612		80		2242	21	14	343	
IF_MULTI_KIND	65	13	28	762	1936				
IF_ONLY_DECL	139				87				
INC_CONTEXT	87	12	11		337	1		51	
INC_EXTRA	153	5	4	4	437	1	9	121	
INCONSISTENT.LABEL					4				
IPAR					13				
NPD.CHECK.CALL.MIGHT					14				
NPD.CHECK.MIGHT					269		61		
NPD.CHECK.MUST					1113			34	
NPD.FUNC.MIGHT					1				
NPD.FUNC.MUST					12				
NPD.GEN.MIGHT					1				
NPD.GEN.MUST					2				
PRECISION.LOSS					504		1		
RETVOID.GEN					2				
RNPD.CALL					10				
RNPD.DEREF					13				
SV.FMT_STR.BAD_SCAN_FORMAT					1				
SV.FMTSTR.GENERIC					3				
SV.INCORRECT_RESOURCE_HANDLING					2				
SV.STR_PAR.UNDESIRED_STRING_PARAMETER					2				
SV.STRBO.BOUND_COPY					108			68	
SV.STRBO.BOUND_SPRINTF					4				
SV.STRBO.UNBOUND_COPY					2				
SV.STRBO.UNBOUND_SPRINTF					2				
SV.TAINTED.INDEX_ACCESS					1				
SV.TAINTED.LOOP_BOUND					1				
SV.TOCTOU.FILE_ACCESS								17	
UNINIT.CTOR.MUST					1				
UNINIT.STACK.MIGHT					13				
UNINIT.STACK.MUST					260			34	
UNREACH.BREAK					10		96		
UNREACH.GEN	443				255		55	36	63
UNREACH.RETURN					1				
UNREACH.RETURN0					40				
VA_UNUSED.GEN	569				160	19	20	33	
VA_UNUSED.INIT					251			443	
VA_UNUSED.INITCONST	50	35			227		46	122	

Analysis Results – Autogenerated code

- All the warnings associated with auto-coded files were found to be false positives
  - break statements following returns and thus are unreachable
  - debugging print statements wrapped in if statements that are always false
  - unused initial values of zero
  - variable increments at the end of loops
  - duplicate definitions from test stubs
  - etc.
- Results were encouraging, and allayed some initial apprehension about the in-house autocoders used on MSL

Summary of Results (Syntatic)

- Very good understanding of the language is required to distinguish real issues
- Static analyzers can find issues that human readers practically could not
- Auto generated code had predictable warning patterns
- Auto generated code contained far fewer errors than hand generated code

Semantic Analysis

Semantic Analysis

- Overview
- Problems
- Successes
- Improvements

Semantic Analysis Overview

- Find requirement implementation in the code.
- Insure the implementation is:
  - Correct
  - Complete
  - Consistent with the design documentation
- Forces additional concentration on requirements
- Allows the opportunity to find errors of meaning

Problems with Semantic Analysis

- Complex Architecture; particularly with multi-step requirements and those involving timing.
- No external or internal mapping between requirements and software source lower than the module level.
- Some requirements contain too little information to search for them. {FSW shall maintain user-specified values (defined data items) for use in sequence logic commands.}

MSL Software IPC Architecture

- Multiple modules running as tasks
- Inter-task communications through messaging
- Messaging service is asynchronous
- Messages can contain callback functions or functions for further processing
- Callback functions are wrapped in a function object
- Tasks can message themselves

MSL Software IPC Cnt

Task CC

```
void init()
{
    ...
    cc_ipc_handle = ipc_create(...);
    taskCreate( cc_task, ....);
    sss_subscribe_notification(
                                        entry_for_sssTask,...);
    ...
}
```

```
entry_for_sssTask( void * arg)
{
    ...
    ipc_send( cc_ipc_handle, arg);
    ...
}
```

```
cc_task()
{
    while( 1)
    {
                ipc_recv( cc_ipc_handle, &msg);
                dispatch_msg_from_ss( msg);
    }
}
```

Task SSS

```
void sss_subscribe_notification(
                                        subscriber_func_adr,...)
{
    remember subscriber_func_adr;
}
```

```
sss_task()
{
    ...
    if( time to send msg to subscriber)
                (*subscriber_func_adr)( msg);
    ...
    ...
}
```

MSL Software IPC Cnt

CC informs SSS how to send a message to itself.

Task CC

```
void init()
{
    ...
    cc_ipc_handle = ipc_create(...);
    taskCreate( cc_task, ....);
    sss_subscribe_notification(
                                    entry_for_sssTask,...);
    ...
}
```

```
entry_for_sssTask( void * arg)
{
    ...
    ipc_send( cc_ipc_handle, arg);
    ...
}
```

```
cc_task()
{
    while( 1)
    {
                ipc_recv( cc_ipc_handle, &msg);
                dispatch_msg_from_ss( msg);
    }
}
```

Task SSS

```
void sss_subscribe_notification(
                                    subscriber_func_adr,...)
{
    remember subscriber_func_adr;
}
```

```
sss_task()
{
    ...
    if( time to send msg to subscriber)
                (*subscriber_func_adr)( msg);
    ...
    ...
}
```

MSL Software IPC Cnt

Task CC

```
void init()
{
    ...
    cc_ipc_handle = ipc_create(...);
    taskCreate( cc_task, ....);
    sss_subscribe_notification(
                            entry_for_sssTask,...);
    ...
}
```

```
entry_for_sssTask( void * arg)
{
    ...
    ipc_send( cc_ipc_handle, arg);
    ...
}
```

```
cc_task()
{
    while( 1)
    {
            ipc_recv( cc_ipc_handle, &msg);
            dispatch_msg_from_ss( msg);
    }
}
```

Task SSS

```
void sss_subscribe_notification(
                            subscriber_func_adr,...)
{
    remember subscriber_func_adr;
}
```

```
sss_task()
{
    ...
    if( time to send msg to subscriber)
            (*subscriber_func_adr)( msg);
    ...
    ...
}
```

SSS informs CC that it would like to send it a message

MSL Software IPC Cnt

Task CC

```
void init()
{
    ...
    cc_ipc_handle = ipc_create(...);
    taskCreate( cc_task, ....);
    sss_subscribe_notification(
                        entry_for_sssTask,...);
    ...
}
```

```
entry_for_sssTask( void * arg)
{
    ...
    ipc_send( cc_ipc_handle, arg);
    ...
}
```

```
cc_task()
{
    while( 1)
    {
            ipc_recv( cc_ipc_handle, &msg);
            dispatch_msg_from_ss( msg);
    }
}
```

Task SSS

```
void sss_subscribe_notification(
                        subscriber_func_adr,...)
{
    remember subscriber_func_adr;
}
```

```
sss_task()

e to send msg to subscriber)
            (*subscriber_func_adr)( msg);
    ...
}
```

CC places the message from SSS on its queue.

MSL Software IPC Cnt

Task CC

```
void init()
{
    ...
    cc_ipc_handle = ipc_create(...);
    taskCreate( cc_task, ....);
    sss_subscribe_notification(
                                  entry_for_sssTask,...);
    ...
}
```

```
entry_for_sssTask( void * arg)
{
    ...
    ipc_send( cc_ipc_handle, arg);
    ...
}
```

```
cc_task()
{
    while( 1)
    {
            ipc_recv( cc_ipc_handle, &msg);
            dispatch_msg_from_ss( msg);
    }
}
```

Task SSS

```
void sss_subscribe_notification(
                                  subscriber_func_adr,...)
{
    remember subscriber_func_adr;
}
```

```
sss_task()
{
    ...
    if( time to send msg to subscriber)
            (*subscriber_func_adr)( msg);
    ...
    ...
}
```

CC receives the message from the queue and processes it.

Results

400	Total number of requirements analyzed
13%	Issue hit rate
9%	Percentage of hard to trace requirements

Table showing the results from the
Build 7 Semantic Analysis

CMD-0568 and CMD-0569 Telemetry not Implemented in the BRR module

Requirement FSW-BRR-5.8 is defined in the Communication FDD Rev. B as follows: FSW shall report the current value of the line error counter in the CMD-0568 telemetry packet.

Requirement FSW-BRR-5.9 is defined in the Communication FDD Rev. B as follows: FSW shall report the current value of the consecutive error counter in the CMD-0569 telemetry packet.

However, the FSW code that implements these requirements was not found in the BRR module.

Mismatch between FSW-ELE-1.21 and its implementation.

FSW-ELE-1.21: In response to a MOD_MODE command, the FSW shall command the specified transmission to **FedEx** or **Postal**.

```
void sdst_set_mod_mode (…)
{
    SdstModModeValType mod_mode_val;
...
    switch (mod_mode) {
      case 1_Day:
          mod_mode_val = 1Day;
          break;
      case Postal:
          mod_mode_val = Postal;
          break;
      case 2_Day:
          mod_mode_val = 2Day;
          break;
    }
}
```

Process Improvement Initiatives

- Deferring requirement not found issues until the test database can be consulted
- Using Doxygen as an analysis tool
- Profiling at the unit test level to aid tracing

Summary of Results (Semantic)

- Domain knowledge of spacecraft flight software is required
- Knowledge of the specific flight software architecture is also required
- Requirement centric approach focused the code reviews

Thanks to the MSL Code Team

- Randall Hintz
- Jeff Zemerick
- Mike Choppa
- Ken Ritchie
- Pradip Maitra
- Rich Kowalski

Questions