Modeling, Automatic Fault Simulation, and Statistical Verification in ESMoL
(pieces of a work in progress)

Joseph Porter, Graham Hemingway,
Gabor Simko, Nicholas Kottenstette, Harmon Nine,
Chris VanBuskirk, Prof. Gabor Karsai, and
Prof. Janos Sztipanovits

Vanderbilt University School of Engineering
Department of Electrical Engineering and Computer Science
Institute for Software Integrated Systems (ISIS)
1025 16th Ave S, Suite 102
Nashville, TN 37212
The Embedded Systems Modeling Language (ESMoL) and its associated modeling suite provide tools to design, analyze, and implement high-confidence distributed embedded software applications. We perform software component modeling in ESMoL by first importing blocks from a Simulink control design model into a graphical design environment. These blocks specify the functions which will be implemented as software components. ESMoL adds component interface definitions as well as hardware platform models and distributed deployment concepts. The current release of the tool suite includes integrated scheduling analysis, platform-specific simulation, and code generation for time-triggered platforms. We will describe an extension to ESMoL to support fault modeling and simulation. Modeling concepts include fault behavior along with the description of testing scenarios to ensure requirements are met under fault conditions. Our work integrates with statistical model checking tools to assess the potential impact of error conditions on specified formal correctness properties.
• For a simple software PID controller...

Start with a few requirements:

• Controller performance:
  • For reference command changes, the control loop shall settle to steady-state (±5%) within 500ms.

• Software timing performance:
  • The control algorithm shall execute within 20ms of the arrival of an input data sample.
Test & Configuration

Explosion Problem

- For a simple software PID controller...

Where can things go wrong?

- First, assume perfect dynamics

- That reduces the possibilities down to:
  - Processor
    - Intermittently unavailable
    - Dead
    - Overloaded
  - Sensor
    - Intermittently unavailable
    - Dead/Stuck
    - Data glitches
    - Permanently biased
  - Actuator...
  - Software...
  - etc...
Test & Configuration
Explosion Problem

- For a simple software PID controller...

Where can things go wrong?
- Second, assume we can mitigate all possible faults that we know about...
- We need to demonstrate that no faults affect the performance of the deployed system.

Roughly, for a single requirement:

\[
#cases = \sum_{f \in F} \sum_{c \in C(f)} I(f, c)
\]

Where
- \( F \) is the set of failure cases,
- \( C(f) \) is the set of component instances in which failure case \( f \) can be applied,
- \( I(f, c) \) is the set of input cases to establish correct operation for the application of \( f \) to component instance set \( c \).
Test & Configuration
Explosion Problem

• For a simple software PID controller...

Roughly, for a single requirement:

\[
\#cases = \sum_{f \in F} \sum_{c \in C(f)} I(f, c)
\]

- \(F\) is the set of failure cases,
- \(C(f)\) is the set of component instances in which failure case \(f\) can be applied.
- \(I(f, c)\) is the set of input cases to establish correct operation for the application of \(f\) to component instance set \(c\).

Important: The cost of evaluation includes the length of the test trace. For complex conditions or intermittent behaviors, traces can be long.
Modeling and Analyzing Faulty Behavior

What about model checking?

- Needs a formal behavioral model and conditions -- conditions are roughly one-to-one with requirements (~one-to-few).
- Exhaustive abstract execution of the behavior space (get rid of the input set size & trace depth factors).
- Limited in hybrid cases – analyses for timing and continuous variables don’t scale unless we use some tricks (i.e. we need a model checking mechanic).
- Bigger problems: We still have to model the behavior of the possible fault cases!

Classic execute & evaluate-style testing:

- Required for behaviors that are too hard to formalize or analyze.
- Still need conditions and faulty behavior models.
- Also need meaningful input sets.
- Big problem: Required trace depth.
- Statistical model checking falls under test & evaluate:
  - Wrapper to collect statistics from repeated runs of an operational model which includes the faulty behaviors.
  - Hypothesis testing to establish confidence intervals for an LTL-specified correctness condition.
  - May be able to reduce behavior models statistically.
  - Still need to determine meaningful input sets.

We’re using implementation from Ed Clarke’s verification group at CMU.
Modeling and Analyzing Faulty Behavior - Related Work

Modeling


- Identified the need to separate the specification of the fault behaviors from the nominal model.
- Example extending Lustre to model and analyze faulty system behaviors.


- Uses the AADL Error Model Annex to specify possible fault states, propagation, and conditions.
- Translates from the AADL model into an analyzable fault tree.

Risk Evaluation in Railway Systems Supported By Modeling Languages and Tool, Fabien Belmonte, Frédéric Thomas, Luis-Fernando Mejía, Alain Blas. In Lambda-Mu, 17ème Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement (IMDR), La Rochelle, France, 2010.

- Some standard and semi-standard fault analyses supported by modeling tools.
- Multiple viewpoints – tabular, graphical, etc.

Analysis


- Bounded LTL conditions.
- Randomized fault behavior included in the model.
- Repeated simulations and automatic trace evaluation against the conditions to build confidence intervals for the correctness hypotheses.


- Modeling and online analysis of fault propagation through system interconnections.
- Online diagnosis engine is configured with fault propagation graphs, allowing online inference to isolate faults based on detected fault effects.
Correct-by-construction methods rely on component behavior conditions and interconnection rules to guarantee correct behavior, reducing verification burdens.

Structural conditions guaranteeing correctness must be maintained throughout the implementation generation process.
ESMoL Design Flow

Single iteration of the design and assessment flow:

1. Import a Simulink control design into an ESMoL models.
2. Specify software component functions and interfaces, and instantiate the components into a logical software dataflow model.
3. Specify the hardware topology for a time-triggered distributed processing network.
4. Define deployment of the logical dataflow to the hardware, and give timing parameters.
5. Transform the ESMoL model (via the Stage 1 transformation) into a model in the ESMoL Abstract language, resolving all implied relationships and structural model inferences.
6. Transform ESMoL_Abstract models into analysis models.
7. Import results from the analysis back into the ESMoL Abstract and ESMoL models.
8. Create platform-specific simulations and generate deployable code using the Stage 2 transformation.

Why ESMoL?

Our approach is to restrict semantics to useful behavior sets that provide leverage for analysis of distributed software control designs:

<table>
<thead>
<tr>
<th>Technique</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive control design</td>
<td>Compositional with respect to stability.</td>
</tr>
<tr>
<td>Synchronous controller models</td>
<td>Compositional with respect to deadlock, and timing-deterministic.</td>
</tr>
<tr>
<td>Time-triggered architecture</td>
<td>Preserves deadlock freedom and timing determinism if schedulable.</td>
</tr>
<tr>
<td>Fault modeling and analysis?</td>
<td>What works here? Some candidates:</td>
</tr>
<tr>
<td></td>
<td>• Statistical behavior models can abstract complex subsystems.</td>
</tr>
<tr>
<td></td>
<td>• Reuse of complex fault models.</td>
</tr>
</tbody>
</table>
Quadrotor Design Example

Quadrotor Control Architecture

- Trajectory Generator
- Outer Loop Controller (Inertial Position)
- Inner Loop Controller (Attitude)
- Motor Compensator
- Quadrotor

CONTROL DESIGN
Quadrotor: Simulink

**Inner Loop Controller (Attitude)**
- Zero-Order Hold
- Rate Transition
- gravity_effects
- desired_motor_thrust
- wind_velocity
- starmac_dynamics

**Outer Loop Controller (Inertial Position) and Trajectory Generator**
- robostix
- u_euler
- u_T
- Ref_data
- OUT1
- OUT2

**Quadrotor**
- Xe_s
- Ve_s
- 3DM_GX1
- SUPERSTAR_II
- CONTROL DESIGN
Quadrotor Software Design: GME & ESMoL

Processor

Platform Design Model

Block == Component Instance

Bus

Port == Message Instance

Node Ports == Comm Channels

Message Assignment

Component Assignment

Software Deployment

Logical Architecture (Dataflow)

SOFTWARE IMPLEMENTATION
Quadrotor Platform Simulation with TrueTime

SOFTWARE IMPLEMENTATION

SOFTWARE ANALYSIS

GENERATION & EXECUTION

Component Timing Parameters:
- TT Schedule – start times
- ExecPeriod
- WC Duration

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Preferences</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT Schedule</td>
<td>0.015</td>
<td></td>
</tr>
<tr>
<td>ExecPeriod</td>
<td>20ms</td>
<td></td>
</tr>
<tr>
<td>WC Duration</td>
<td>1.5ms</td>
<td></td>
</tr>
</tbody>
</table>

Resolution
- Proc RS 4MHz 0s 0s
- Comp InnerLoop = 50Hz 1ms
- Comp DataHandling = 50Hz 1ms
- Comp ADC = 50Hz 1us
- Comp SerialIn = 50Hz 1ms
- Comp SerialOut = 50Hz 1ms
- Msg DataHandling.sensor_data 8B RS/ADC RS/DataHandling
- Msg DataHandling.pos_ref 8B RS/SerialIn RS/DataHandling
- Msg InnerLoop.thrust_commands 8B RS/InnerLoop RS/SerialOut
- Msg LocalOrder 1B RS/DataHandling RS/InnerLoop

Proc GS 100MHz 0s 0s

Bus TT_I2C 100kb 1ms

Msg OuterLoop.ang_ref 8B GS/OuterLoop RS/InnerLoop

Msg DataHandling.pos_msg 8B RS/DataHandling RS/OuterLoop

Scheduler Spec

Calculated Schedule

Generator

Scheduler Spec (.scs)

Scheduler

Scheduler Result (.rslt)

ESMoL Model File (.mga)

Importer

TrueTime Simulink Model (mdl)

Task Execution Code (C)
Quadrotor: TrueTime
Execution and Schedule

Primary execution loop is an embedded Time-Triggered scheduler

Time-triggered execution schedule for one Node

Tracking performance is close to ideal...

...but platform effects are successfully simulated
Nominal Quadrotor Trajectories
Before & After

Synchronous Digital Controller

Distributed Controller

Software Implementation

Zero-Order Hold

Nominal quadrotor trajectories before and after implementation, comparing the synchronous digital controller and distributed controller with relevant graphs and diagrams.
Digital Controller Has Infinite $L^m_2$ - Gain

Let $H_{dc}: u_{dc} \rightarrow y_{dc}$ be a constant gain $k_{dc} = 1$ with zero phase ($e^{j \theta}, \theta = 0$).

$$u_c(t) = \begin{cases} 
1, & t = iT_s \\
-\delta (\delta > 0), & \text{otherwise.}
\end{cases} \quad \rightarrow \quad y_c(t) = 1$$

Therefore $H_c$ cannot be bounded in terms of phase or magnitude.

$$\int_{0}^{NT_s} y_c(t)u_c(t)dt = NT_s < 0 \rightarrow H_c \text{ is not passive}$$

$$\int_{0}^{NT_s} y_c^2(t)dt = NT_s \rightarrow 2 < \frac{1}{2} = \lim_{N \rightarrow 0}$$
Fault Behavior Injection

Sensor Glitch Model

Synchronous Digital Controller

Distributed Controller
% Start sampling
while (out == 2),

% The model writes its output in the workspace variable 'simout'
sim('quadrotor_demo_noscopes');

% Update counters (if simout = 0, no violation)
if (simout == 0), x = x + 1; end;
n = n + 1;

% Apply test
out = BFT(theta, T, n, x, 1, 1, podds); % Hypothesis test
end;

From here:
1. Run lots of times.
2. Check traces against condition.
3. Track statistics until confidence level is reached.
% Start sampling while (out == 2),

\[
\text{if (simout} = 0), x = x + 1; \text{end;}
\]

\[n = n + 1;\]

% Apply test
\[\text{out} = \text{BFT}(\theta, T, n, x, 1, 1, \text{podds});\]

\[\text{Hypothesis test}\]

90% confidence in \(~19\) minutes

99% confidence in \(~156\) minutes
ESMoL Language Integration Concepts

Proposed modeling and test generation flow:

ESMoL System Design → ESMoL Fault Scenarios → ESMoL Test Cases → Simulink/TrueTime Fault Models

- Fault effects capture.
- Creating test cases from requirements & fault scenarios.
- Generation & evaluation.
- Test scripts & input data sets.
- Statistical model checking conditions.
ESMoL Language Integration Prototypes

Concepts:

• Separate specification of nominal design, fault behavior, and test cases.
• Use model references and break out details where needed.

Note: This is semi-orthogonal to the state/transition/propagation descriptions in the AADL Error Model Annex. Our focus for now is on specifying causes and effects at a detailed level.
Part I: ESMoL Design Model

Types: Component definitions with Simulink function specifications.

Two views of the design model:

Dataflow between component instances.

Hardware deployment of component instances.
ESMoL Language
Integration Prototypes

Part II: Fault Behavior Model

Description of fault scenario

Call out the design model to give the context for the faulty behavior.

Fault Model

Fault model port elements specify fault behavior & composition.

Break out design model component (by reference) to attach fault behavior.
Part III: Test Case Models

Fault scenario is called out to provide context for the test case.

Can we use multiple fault scenarios in the same test case?

Fault conditions relate tests to requirements. Expression variables are broken out (by reference) from fault scenario models.

Inputs, outputs, and referred components (from context objects) specify participants in auto-generated test harness.
Things to Consider

Possibilities:

• Using statistical behavior models may allow us to reduce the complexity of simulations for faulty behaviors.
• Reuse of fault behavior models as templates to speed up modeling of cases to be evaluated.

Problems:

• Test input space size is still an open problem when considering faulty behavior.
• Required trace depth + TrueTime performance can lead to long runs.
• Still hard to find and evaluate rare events.
Questions?

Those interested in trying our tools can visit
https://wiki.isis.vanderbilt.edu/hcddes/index.php/The_ESMoL_Tool