Integrating Formal Model Checking with the RTEdge™ AADL Microkernel

Serban Gheorghe
Edgewater Computer Systems Inc
serban.gheorghe@edgewater.ca
What is RTEdge™

- **RTEdge™** is a software development environment
  - For Proof Based, Model Driven Development (MDD) of Modal Software Systems
  - Built on a language subset of the AADL and UML2
    - conceptually similar to the ADA Spark / Ravenscar subsets

- **Main goal**: assist in the construction of theoretically correct real-time applications
  - Hard real-time software systems
  - Mission critical software systems
  - Real-time systems distributed over communication networks

- **RTEdge™ approach**: convergence of three software engineering technologies

1. **Model Driven Development (MDD)**
2. **Component based specification and design**
3. **Proof based engineering**:
   1. **Timing** constraints checked by mathematical proof
   2. **Functional** correctness checked through static analysis and formal model checking

UML2 concepts:
- Component Interfaces:
  - Ports typed by Protocol contracts
- Component Behavior:
  - State Machine Subset
RTEdge™ as a subset of AADL sw component model

AADL component_category ::= 
  abstract_component_category 
  | composite_category 
  | execution_platform_category 
  | software_category

RTEdge Deployment Objects

RTEdge Components: 
Composite Capsule (CC) 
Atomic Capsule (AC)

RTEdge Atomic Capsule behaviour:  
Finite State Machines with embedded Programming Language Functions calls (Misra C, other)
RTEdge™ as a subset of AADL execution semantics

AADL Model

RTEdge Language Subset
- RTEdge Code generated from AADL
- Support AADL Thread Dispatch Modes
  - Periodic, Sporadic, Aperiodic, Timed and Hybrid
- Support AADL inter-thread communication/synchronization semantics

RTEdge Runtime Exec Semantics
- Asynchronous Message Passing Executive
- Time constrained Discrete Events only
- **AADL Threads ↔ Atomic Capsules**
  - Threads with FSM behavior, purely reactive
  - Features subset
    - Event/Event Data Ports
    - Required/Provides Data Ports
    - FSM behavior, a subset of UML SM
- **AADL Thread Groups ↔ Composite Capsules**
  - Resolved to communicating Atomic Capsules
- **Periodic Timer Service**

Bounded Overheads Run-time Executive
Use of RTEdge Formal Checking

- A mixture of threads with Periodic and Aperiodic behavior
- Flow Path Feasibility Analysis
- Deadline Monotonic Priority Assignment based on Flow deadlines
- WCRT Analysis
- Port Buffer Queue size analysis

Debug at model level
Consistent Execution Semantics
Bounded Overheads Executive

Analyzing, Deploying and Debugging AADL Software
Trust Layers for execution semantics

AADL Model

P1   P2

Abstract P3_RTEdge

Synchronous Annex Formal Semantics (??)

Asynchronous Annex Formal Semantics (??)

RTEdge Language Subset

RTEdge™ Application Formal Properties
- User defined Safety and Liveness properties
- Formal Model Checking Spin/Promela
- Evolution to Formal Compositional Verification

RTEdge Runtime Exec Semantics

• Formally defined operational semantics
• Axiomatic semantics – Theorem proving
  • Explore use of BLESS
Crossing the “V”

The AADL Spec covers many possible types of execution semantics. (95) A method of implementing a system is permitted to choose how executing threads will be scheduled. A method of implementation is required to verify to the required level of assurance that the resulting schedule satisfies the period and deadline properties. (AADL 2 Spec, page 94)

Static Model Analysis Tools
• huge cost savings benefits through early problem resolution
  HOWEVER
• must have intimate knowledge of
  1. Execution State Space
  2. Target Execution Semantics

Model Transformation to executable, target deployable code creates a semantic discontinuity

The effectiveness of Real-time and Functional Static Analysis Tools depends on the enforcement of a well defined Execution State Space and Target Execution Semantics

Single Source of Truth: Deployed Software Components Integration and Debugging must be performed in the context of the Models
The RTEdge™/AADL Solution

- The AADL subset execution semantics assumed by the Code Generator and Proof Tools is implemented and enforced in a HW independent Run-Time Exec library

- HAL adaptation of Run-Time Exec to multiple platforms
- A host RTOS is assumed on the HW Platform

- The Code Generator and Proof Tools for Real-Time and Functional Properties assume a well defined RTEdge/AADL subset execution semantics

Virtual Time Simulation Platform
- Enable execution of software components on host desktop computers by emulating target hardware real-time behaviour

RTEdge Tools
- Real Time Analysis Tool
- Functional Properties Analysis Tool
- Code Generator Tool
RTEdge™ SDE Components

- Model Editors
- Model Validator
- Flow Detector
- Schedulability Analyzer
- Promela Code Generator
- Code Generator + Build
- Debugger

PTC

- DOORS Link
- DOORS Database

- RTEdge-Spin Model Checking

ECLIPSE FRAMEWORK

TARGET

- GDB Server
- RTEdge™ Executive
- Real-Time OS
Protocol Contracts for Software Components

The **UML2** concept of Protocol is used in RTEdge as a specification mechanism:
- capturing and refining interaction Assumptions-Guarantee contracts
- conformance / consistence when implementing or composing Software Components
More on RTEdge™ – Application Model

- An Application maps into one RTOS Task
  - resolves to set of concurrent Atomic Capsule (AC) Finite State Machines (FSM)
  - built-in FSM event priority scheduling
- Closed System Model Definition:
  - “External Tasks” model environment RTOS Tasks I/O behavior and processor demands
  - Independent Input time arrival specification (Period, Jitter, Jitter bounds)

RTEdge Dispatch Event types:
- Independent Input arrival
- Signal Events received at Port FIFOs
- AC Transient State Activity Completion
- Static Event priority assignment- Deadline Monotonic

Application Flows Specification:
Event Flows with Real-Time Contracts
- Arrival spec (Period,Jitter)
- Deadline

Event Flow: a specification of a causal relationship between an ordered pair of RTEdge™ Dispatch Events

Key concepts:
- PROCESSOR and COMMS resource scheduling are based on Event Flows
- RTEdge automatically calculates:
  - Application implementation Flow paths and
  - Worst-Case Response Times for each Event Flow based on Execution costs
More on RTEdge™ - Transactions

A Transaction is the set of Event Flows originated from an Independent Input

- Conditional Flow Paths can happen in AC FSMs
- Flows can end on AC FSM States

Transactions can share AC FSM Instances

Sending a Signal Event through a Port creates a concurrent Flow Path

- A Transaction is complete when all the downstream events caused by the Independent Input have been consumed
- Indicates no State Machines Deadlock (events not consumed) or Livelock (infinite event loops)
Formal Model Checking Workflows

Define, Translate and Verify Application Annotations expressing User Defined Formal Properties

Application Model in RTEdge

User Defined Behavioral Assumptions

Translation Options

RTEdge to Promela Translation

User Defined Assertions (Safety)

User Defined Safety or Liveness Properties expressed as LTL

RTEdge Application Annotations

Application Model in Promela

Spin

Promela-Spin Tool Framework

Counter-examples

Promela Assertions and Invalid End States

LTL Formula

Translation

Generation

Refinement

Step-wise Model Refinement

Code Generation

Static Analysis Tools

RTEdge™ Toolset

Application Executable
Solution Issues - 
Mapping RTEdge™ Applications into Promela

RTEdge™ Application as a Closed System

Sensor_In
ExtTask1
Recalibrate

CC2

D

E

Adjust_ctrl
Calibrate_ctrl
ExtTask2
Calibrate_other

Each AC Instance is translated into a Promela Process
Each two-way Port Connection is mapped into two receive FIFO channels

Promela
Input Generator
Process

Dispatch Input Generator Process

Promela
Never Claim Process

RTExec Dispatch Promela Process

Independent Inputs Generator Promela Process translated from External Tasks

Dispatch Process enforces RTEdge™ Events dispatch using rendez-vous channels
Solution Issues -

Annotating the RTEdge™ Application Model

- Checking properties are specified as Annotations to a deployable RTEdge™ Application:
  ✓ Seven types of Annotations have been defined:

<table>
<thead>
<tr>
<th>ANNOTATION TYPE</th>
<th>INTENDED USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ - GCV (GCC) Global Checking Vars/Const</td>
<td>Define Verification Meta-Variables and their behavior through assignments</td>
</tr>
<tr>
<td>✓ - AC Instance Behaviour Annotation</td>
<td></td>
</tr>
<tr>
<td>✓ - Checking Assertions:</td>
<td></td>
</tr>
<tr>
<td>✓ Inline Checking Assertion</td>
<td>Define Assertions on Meta-Variables and App State Variables Expressions</td>
</tr>
<tr>
<td>✓ AC Instance Checking Assertion</td>
<td>with Different Evaluation Scope</td>
</tr>
<tr>
<td>✓ Application Scope Checking Assertion</td>
<td></td>
</tr>
<tr>
<td>✓ Input Data Range Constraints</td>
<td>Control Size of the Input State Space</td>
</tr>
<tr>
<td>✓ User Defined LTL Formula</td>
<td>Define Temporal Logic App Properties</td>
</tr>
<tr>
<td>✓ Transient State Behavioural Annotation</td>
<td>Abstract out State Machine Activity C Code</td>
</tr>
<tr>
<td>✓ Execution Condition Annotation</td>
<td></td>
</tr>
</tbody>
</table>
Defining meta-behavior Annotations (Application Checking View)

Application Inputs

Atomic Capsule (AC) Instances

Counter-examples

Global Checking Variables/Constants (GCV or GCC)

- User-defined or Tool-generated meta-variables declared outside the RTEdge™ Application state space and accessible in any Annotation
  - Any basic data type supported by Promela/Spin
- Tool-generated GCVs are created for use by the tool-generated LTL formula;
  - documented behavior
  - they can also be used in other user-defined Annotations

AC Instance Behavior Annotations

- Used to define GCV behavior through conditional assignments
  \[ \text{Check\_GCV} = (\text{attr.y} > 1) \rightarrow (\text{attr.x} + \text{Increment\_GCC}) : (\text{Fail\_GCV}) \]
- Attached to an AC Instance state machine Transition
- Executed inline wherever the Annotation is placed
Defining **Assertions**

Application Checking View

**Checking Assertion**

- Relational Expressions of GCVs, GCCs and in scope Atomic Capsule Attributes
- Used to verify safety conditions to which an Application must comply
- A finding is raised if condition returns false

**Application Scope Checking Assertions**

- attached to an Application and evaluated on every change of the global RTEdge™ Application state space (every AC instance state machine Transition)

**AC Instance Checking Assertions**

- attached to capsule instances and evaluated on every Transition within its state machine

**In-line Checking Assertions**

- attached to state machine Transitions and evaluated when the Transition is taken

**Examples:**

Check_GCV >= 10
(myDap.x < Status_GCC) && (Check_GCV < attr.y)
Defining **Input Data Range Constraints** (Application Checking View)

- **Application Inputs**
- **Capsule Instances**
- **Counter-examples**

**Input Data Range Constraints**

- Affect the Application state space
- Restrict the range of values for System Inputs
  - Mechanism for reducing the size of the Application input space
  - Attached to Application Inputs in the Application Checking View
  - Can reference Global Checking Constants

**Example:**
(sod_struct.b[0]: 5,8,12..15);
Defining **Transient State Behavior Annotation** (Application Checking View)

**Transient State Behavior Annotations and Execution Conditions**

- Used as an abstraction mechanism for AC Activities attached to Transient States
  - they do modify the Application state space (AC Attributes)
- **Transient State Behavior Annotation**: used to define how the state’s Activity code would modify local data
- **Execution Condition**: attached to Transitions that exit Transient States, used to define conditions under which that Transition is taken

**Default Promela Translation Option**

- **Use Transient State C Activity Code**
  - c_track, c-code, c_expr features of Spin for embedding implementation code in the Verifier executable
The Default LTL Formula

- Out-of-the-box tool generated GCVs, Predicates and a default LTL for a deployable Application, checking the property:
  “All Transactions of this Application will complete infinitely often”
  - Check for Transaction livelock: any condition whereby states are constantly changing but no progress is made (e.g., infinite loop)
  - Check for Transaction deadlock: any condition whereby Transaction events are enqueued but no progress is made
  - Assertions are also generated to check for undefined trigger exceptions: any condition whereby a Signal arrives at a Stable State that cannot handle it

- Tool-generated LTL uses Tool-generated Predicates that track Transaction event count GCVs for every Independent Input

Predicate \( \text{tgpredtimer1rttimeout1} \): \((\text{TG\_GCV\_TIMER1\_RT\_TIMEOUT\_TRANSACTION\_COUNTER\_1} = 0)\)

Generated LTL: \(! [\] (\! \text{tgpredtimer1rttimeout1} \rightarrow (<> \text{tgpredtimer1rttimeout1})))\)

Negated for the Never Claim
Always
Transaction Count is non-zero
implies
Transaction Count eventually reaches zero
eventually
## Solution Issues - RTEdge™-Promela Verification Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>LTL Run</th>
<th>Safety Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include Priorities</td>
<td>On</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verification Type</td>
<td>Safety</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acceptance</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-Progress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Independent Input Signal Generation</td>
<td>Guaranteed Arrival in Hyper-Period (Weak Fairness &amp; Bounded Jitter)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All orders in Hyper-Period (Weak Fairness and Unbounded Jitter)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All orders (no Weak Fairness &amp; Unbounded Jitter)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTL Option</td>
<td>Tool Generated</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>User Defined</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No LTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction Level relative to the concrete RTEdge™ Application</td>
<td>Other conditions: all Activity C-Code Included</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### False Positives

### False Negatives (Counter-Examples)
Solution Issues - Interpreting the counter-example

- Support is provided to relate a line from the counter-example file to the RTEdge™ Application model
- Atomic Capsules in the concrete Application are mapped into Promela processes with shortened names
- Meta-data is generated in the Promela files to help map short form Promela Process names to fully qualified names for AC instances
Verifying through Model Checking the RTEdge™ Schedulability Analysis Algorithm

- Modified Lehoczky, Harbour Klein Algorithm for Fixed Priority Task Set with Precedence

- Linear precedence sub-tasks of original algorithm supplemented with
  - Conditional execution nodes
  - Parallel (fork) nodes
  - Bounded Cycles

  to reflect RTEdge Transaction graphs

1. Static Priorities are assigned *per event and state* based on deadlines (DMS)
2. Priorities are statically adjusted based on resource usage (*priority ceiling protocol*)
3. Create task system from model (all RTEdge Transactions originated in Independent Inputs)
   - each task is modeled as a graph of subtasks (AC Activities) with cycles and conditional paths
   - ensure cycles are bounded by user-supplied bounds
4. calculate Worst Case Response Time (WCRT)
   - calculate utilization, give up if greater than one
   - for each task
     - determine worst case phasing, interference, blocking
     - calculate WCRT for each subtask, compare with deadlines

Using Formal Link Annotations, we can verify the correctness of the WCRT analysis for a model

1) Assume tool calculated WCRT is correct
2) Define an execution time accumulator GCV per Transaction and behavior to keep updated
3) Prove as a safety property on every execution path that WCRT can’t be exceeded
Additional Annotations

- Adaptation of IEEE 1850 Property Specification Language (PSL) to RTEdge Protocols, Capsule Interfaces and Capsule compositions
- Signal sequences specified as IEEE1850 PSL Sequential Extended Regular Expressions (SERE) of Signals
  - Regular Expressions of Signals on Protocols
  - Regular Expressions of Signals on Capsule Interfaces
  - Regular Expressions of Signals on Ports of Capsule Roles contained in a Composite Capsule type

Provide a formal framework for hierarchical Compositional Verification for component based applications

- Individual Component implementations investigated for conformance to formal Capsule Interface specification
- Specification of aggregated components can be checked for
  - consistency and conformance to the container Capsule Interface formal spec