Engineering High-Assurance Software for Distributed Adaptive Real-Time Systems

Sagar Chaki, Dionisio de Niz, Mark Klein

June 10, 2015

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Motivation

Assure Behavior of Distributed Adaptive Real-Time (DART) systems

• Operate in uncertain environments but with critical requirements
• Coordination is needed for mission success
• Key to developing high-assurance autonomous coordinated resilient systems

Currently validated via testing

• Low coverage (low assurance), late in development (high cost)
• Judicious combination of FV techniques
  • Support compositional reasoning
  • Deterministic/probabilistic requirements
DART in a Nutshell

1. Enables compositional and requirement specific verification
2. Use proactive self-adaptation and mixed criticality to cope with uncertainty and changing context

System + Requirements (AADL + DMPL) → Verification

Verification → Code Generation

Code Generation → Demonstrate on DoD-relevant model problem (DART prototype)
- Engaged stakeholders
- Technical and operational validity

1. Middleware for communication
2. Scheduler for timing contracts
3. Monitor for functional contracts

1. ZSRM Schedulability (Timing)
2. Software Model Checking (Functional)
3. Statistical Model Checking (Probabilistic)
DART High-Level Architecture

Software for guaranteed requirements, e.g., collision avoidance protocol must ensure absence of collisions

Software for probabilistic requirements, e.g., adaptive path-planner to maximize area coverage within deadline

High-Critical Threads (HCTs)  Low-Critical Threads (LCTs)

MADARA Middleware
ZSRM Mixed-Criticality Scheduler
OS/Hardware

Environment – network, sensors, atmosphere, ground etc.

Node₁

Nodeₖ

Research Thrusts
• Proactive Self-Adaptation
• Statistical Model Checking
• Real-Time Schedulability
• Functional Verification

Validation Thrusts
• Model Problem
• Workbench

Sensors & Actuators

MADARA
Sched
OS/HW

OS/Software Engineering Institute
Carnegie Mellon University
DART Programming : AADL + DMPL

AADL : Architecture Analysis and Description Language
DMPL : DART Modeling and Programming Language

AADL : High level architecture + threads + real-time attributes
  • Perform ZSRM schedulability via OSATE Plugin
  • Generate appropriate DMPL annotations

DMPL : Behavior
  • Roles : leader, protector
  • Functions : mapped to real-time threads
    • Period, priority, criticality (generated from AADL)
    • Behavior : C-style syntax. Can call-out to arbitrary libraries.
  • Functional properties (safety) : software model checking
  • Probabilistic properties (expectation) : statistical model checking

Being implemented as a DART Workbench. Upcoming public release.
Example: Self-Adaptive and Coordinated UAS Protection

**Adaptation:** Formation change (loose ↔ tight)

**Loose:** fast but high leader exposure

**Tight:** slow but low leader exposure

**Challenge:** compute the probability of reaching end of mission in time $T$ while never reducing protection to less than $X$.

**Challenge:** compare between different adaptation strategies.

**Solution:** Statistical model checking (SMC)
System Architecture for Example

Leader Threads
- Collision Avoidance
- Waypoint
- Adaptation Manager

MADARA Middleware
ZSRM Mixed-Criticality Scheduler
OS/Hardware

Leader Node

Protector Threads
- Collision Avoidance
- Waypoint

MADARA Middleware
ZSRM Scheduler
OS/Hardware

Protector Node
ZSRM
Schedulability
Zero-Slack Rate Monotonic (ZSRM) Scheduling

Guarantees deadlines of high-criticality tasks even in overloads
  • e.g. too many obstacles to avoid

Lower criticality tasks meet their deadlines if no overloads in higher criticality

Asymmetric protection: protect higher-criticality from lower-criticality but higher-criticality can steal CPU cycles from lower-criticality one.

Schedule under RMS, stop lower-criticality tasks at last instant to ensure finishing overload by the deadline (zero-slack)

\[ \tau_{LC} = (2,2,4,4) \]
\[ \tau_{HC} = (2.5,5,8,8) \]
DART ZSRM Scheduling Toolchain

AADL Description of DART System → AADL/OSATE + ZSRM Plugin → DMPL Program with threads and timing attributes

```
subprogram subprogram_adaptation_manager
properties
    Source_Name => "ADAPTATION_MANAGER";
end subprogram_adaptation_manager;
thread implementation adaptation_manager.i
calls main:{ m: subprogram subprogram_adaptation_manager; };
properties
    Compute_Entrypoint_Source_Text => "example-05.dmpl";
    Compute_Entrypoint_Call_Sequence => reference (main);
end adaptation_manager.i;
DART::DMPL_TIMING_PARAMETERS_TEXT => "/tmp/dmpl_nodes.dmpl";
```

```cpp
//-- MADARA receiver thread
NODE uav
{
    @Period(4000000)
    @Criticality(2)
    @WCExecTimeNominal(10000)
    @WCExecTimeOverload(20000)
    ADAPTATION_MANAGER();

    //-- WAYPOINT();
    //-- COLLISION_AVOIDANCE()
}
```
New Research: ZSRM in Pipelines (1)

High & low criticality run concurrently
New Research: ZSRM in Pipelines (2)

Reduces pipeline to single-resource scheduling
Avoids assuming worst alignment in all stages
But need to deal with transitive interferences due to zero-slack
Proactive Self-Adaptation
Proactive Self-Adaptation

MAPE-K [Kephart 2003]
Self-Adaptation in DART

Some aspects of the environment are unknown before the mission execution

- for example, the threat level of different areas
- the environment conditions are discovered as the mission progresses
- it’s not possible to plan everything in advance

Need for proactive adaptation

- Adaptations may take time (e.g., formation change), so they have to be started proactively
- Decisions taken at any point impact future outcomes (e.g., higher fuel consumption reduces range)

Current solution based on constructing a MDP and using probabilistic model checking to find the best strategy at each adaptation point

- Exploring integration with Machine Learning techniques
MDP for Adaptation Decision

- Stochastic model of the environment updated at run time
- Time over the decision horizon
- Starting tactic or not is a nondeterministic choice
- System model reflects effect of tactic when tactic completes
- High-level system properties relevant to computing objective function
- System model reflects effect of tactic when tactic completes

Clock

Tactic 1 _end

Tactic N _end

Self-adaptive system

Module

Shared action
Adaptation Decision with PRISM

First choice independent of subsequent environment transitions

PRISM strategy synthesis

Resolves nondeterministic choices to maximize expected value of objective function

Statistical Model Checking
Statistical Model Checking

- Probability estimate for each property evaluated via “Bernoulli Trials”
- Number of trials required to estimate probability of a property depends on
  - desired “relative error” (ratio of standard deviation to mean)
  - true probability of the property
- Running trials in parallel reduces required simulation time.
  - SMC Client invokes Vrep simulation on each node.
  - SMC Aggregator collects results and determines if precision is met.
  - Simulations run in “batches” to prevent simulation time bias.
- Importance sampling (focuses simulation effort on faults)
Statistical Model Checker

$M$ (DMPL)

$\phi$ (DMPL)

DMPL Compiler

dmplc

Executable

Log Generator

Log (1 per node)

Log Analyzer

Result

SMC Aggregator

One Bernoulli Trial

SMC Client
Distributed Statistical Model Checker

Batch Log and Analyze

log-gen → log-analyze → Result

SMC Client

SMC Aggregator

No

RE acceptable?

Yes

Update Result and RE

Each run of log-generator and log-analyzer occurs on a Virtual Machine. Multiple such VMs run in parallel on HPC platform. Clients can be added and removed on-the-fly.

Future Work: Importance Sampling to reduce number of simulations needed for “rare” events.
Example Demonstration

Adaptation: Formation change (loose ↔ tight)

Loose: fast but high leader exposure

Tight: slow but low leader exposure
Software Model Checking
End-to-End Verification of Collision Avoidance

Combining model checking of collision-avoidance protocol with reachability analysis of control algorithms via assume-guarantee reasoning

Prove application-controller controller contract for unbounded time
- Manually supplying invariants and checking them via CBMC

Prove controller-platform contract via hybrid reachability analysis
- Done by AFRL
Interactive Verification of $I_{AC}$ at Source Code Level

DMPL Program

- Distributed Application
- Safety Specification
- Round Invariants

Sequentialization

Single-Threaded C Program

Software Model Checking (CBMC, BLAST etc.)

Assume Synchronous Model of Computation

Generated C Program

```c
//-- INVAR : inductive invariant
void main()
{
    INIT();  //-- initialization
    assert(INVAR);  //-- base case
    HAVOC();  //-- assign all variable ND
    __CPROVER_assume(INVAR);  //-- IH
    ROUND_NODE_1();
    ...
    ROUND_NODE_k();
    assert(INVAR);  //-- inductive check
}
```
DART Team

Bjorn Andersson  Mark Klein
Bud Hammons  Arie Gurfinkel
Gabriel Moreno  David Kyle
Jeffery Hansen  James Edmondson
Scott Hissam  Dionisio de Niz
Sagar Chaki

QUESTIONS?

https://github.com/cps-sei/dart
Contact Information Slide Format

Sagar Chaki
Senior MTS
SSD/CSC
Telephone: +1 412-268-1436
Email: chaki@sei.cmu.edu

Web
www.sei.cmu.edu
www.sei.cmu.edu/contact.cfm

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257