Engineering High-Assurance Software for Distributed Adaptive Real-Time Systems
Sagar Chaki, Dionisio de Niz, Mark Klein

Software Solutions Conference 2015
November 16–18, 2015
Motivation

Distributed Adaptive Real-Time (DART) systems are key to many areas of DoD capability (e.g., autonomous multi-UAS missions) with civilian benefits.

However achieving high assurance DART software is very difficult

- Concurrency is inherently difficult to reason about.
- Uncertainty in the physical environment.
- Autonomous capability leads to unpredictable behavior.
- Assure both guaranteed and probabilistic properties.
- Verification results on models must be carried over to source code.

High assurance unachievable via testing or ad-hoc formal verification

**Goal**: Create a sound engineering approach for producing high-assurance software for Distributed Adaptive Real-Time (DART)
DART Approach

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

System + Properties (AADL + DMPL)

Verification

Code Generation

Brings Assurance to Code
1. Middleware for communication
2. Scheduler for ZSRM
3. Monitor for runtime assurance

1. ZSRM Schedulability (Timing)
2. Software Model Checking (Functional)
3. Statistical Model Checking (Probabilistic)

Demonstrate on DoD-relevant model problem (DART prototype)
- Engaged stakeholders
- Technical and operational validity
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)
Example: Self-Adaptive and Coordinated UAS Protection

Adaptation: Formation change (loose ⇔ tight)

Loose: fast but high leader exposure
Tight: slow but low leader exposure

Video
Key Elements of DART

- Constrain the system structure and behavior to facilitate tractable analysis and code generation.
- Program DART systems and specify properties in a precise manner.
- Use probabilistic model checker to repeatedly compute optimal adaptation strategies with bounded lookahead.
- Evaluate adaptation strategy quality over mission lifetime.
- Constrain the system structure and behavior to facilitate tractable analysis and code generation.

- Efficient middleware provides distributed shared variables with well-defined data consistency.
- Combines model checking & hybrid analysis to ensure end-to-end CPS correctness.
- Functional Verification
- Architecture
- DMPL AADL
- Proactive Self-Adaptation
- Statistical Model Checking
- MADARA
- ZSRM Scheduling

- Ensures high-critical tasks meet their deadlines despite CPU overload.
### Architecture

<table>
<thead>
<tr>
<th>DMPL AADL</th>
<th>Adaptation</th>
<th>Statistical MC</th>
<th>MADARA</th>
<th>ZSRM Scheduling</th>
<th>Functional Verification</th>
</tr>
</thead>
</table>

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.

**Environment** — network, sensors, atmosphere, ground etc.

**Design constraint enables analysis tractability**

**Baked into the programming languages used**

**Distributed Shared Memory**

**Node₁**

**Nodeₖ**

**MADARA Middleware**

**ZSRM Mixed-Criticality Scheduler**

**OS/Hardware**

**High-Critical Threads (HCTs)**

**Low-Critical Threads (LCTs)**

**Sensors & Actuators**

**Baked into the programming languages used**
System Architecture for Demo

Leader Threads
- Collision Avoidance
- Waypoint
- Adaptation Manager

Protector Threads
- Collision Avoidance
- Waypoint

MADARA Middleware
ZSRM Mixed-Criticality Scheduler
OS/Hardware

Leader Node

Protector Node
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)
  - C-style syntax. Invoke external libraries and components
- Functional properties (safety) : software model checking
- Probabilistic properties (expectation) : statistical model checking

**AADL and DMPL supports the right level of abstraction at architecture and code level to formally reason about DART systems**
DART Modeling and Programming Language (DMPL)

Domain-Specific Language for DART programming and verifying
- C-like syntax
- Balances expressivity with precise semantics
- Supports formal assertions usable for model checking and probabilistic model checking
- Physical and logical concurrency can be expressed in sufficient detail to perform timing analysis
- Can invoke external libraries and components
- Generates C++ targeted at a variety of platforms

Developed syntax, semantics, and compiler

AADL and DMPL supports the right level of abstraction at architecture and code level to formally reason about DART systems
Monitor

Analyze

Knowledge

Plan

Execute

Proactive Self-Adaptation via MAPE-K

Target system

MAPE-K Emerged from Autonomic Computing Community [Kephart 2003]
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
Adaptation using a Markov Decision Process

- 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

Environment

Tactic 1

Clock

Tactic N

Self-adaptive system

Module

Shared action

Starting tactic or not is a nondeterministic choice
<table>
<thead>
<tr>
<th>Architecture</th>
<th>DMPL</th>
<th>Adaptation</th>
<th>Statistical MC</th>
<th>MADARA</th>
<th>ZSRM Scheduling</th>
<th>Functional Verification</th>
</tr>
</thead>
</table>

**t=0**

- System
- Environment

**t=1**

- T1
- T2

**PRISM strategy synthesis**

Resolves nondeterministic choices to maximize expected value of objective function.

First choice independent of subsequent environment transitions.

ongoing work: replace probabilistic model checking with dynamic programming for speed.
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)
The diagram illustrates the DART Statistical MC Workflow. It starts with the SMC Client, which communicates with the SMC Aggregator. The workflow involves the following steps:

1. **DMPL Compiler** (DMPL Compiler) compiles the `M (DMPL)` and `ϕ (DMPL)` into an executable.
2. The executable generates logs using the `dmplc` tool.
3. These logs are then analyzed by the Log Generator and Log Analyzer.
4. The result of this analysis is the One Bernoulli Trial.

The workflow is designed to ensure high-assurance software development.

**Batch Log and Analyze**

- log-gen
- log-analyze
- Result

**SMC Client**

**SMC Aggregator**

- DART Distributed Statistical MC

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

**Update Result and RE**

- Yes: Result
- No

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.
Statistical MC Results

Total Coverage

Protected Area

Leader

Protectors

Single Protector Coverage

\[ \theta = 2\sin^{-1}\left(\frac{r}{d}\right) \]
MADARA: A Middleware for Distributed AI


Legend

- User
- OS/file
- KaRL
- Transport

MADARA Architecture
GAMS: Group Autonomy for Mobile Systems

1. Built directly on top of MADARA (https://github.com/jredmondson/gams)
2. Utilizes MAPE loop (IBM autonomy construct)
3. Provides extensible platform, sensor, and algorithm support
4. Uses new MADARA feature called Containers, which support object-oriented programming of the Knowledge Base
Zero-Slack Rate Monotonic (ZSRM) software stack
- ZSRM Schedulability Analysis as AADL/OSATE Plugin
- ZSRM Scheduler as Linux Kernel Module
- ZSRM Priority & Criticality Ceiling Mutexes

Pipelined ZSRM
- Based on pipelines that allows parallel execution of multiple tasks in different stages.
- Avoids assuming all tasks start together in all stages
- Reduces the end-to-end response time and improves utilization
- Paper submitted to RTAS’16
Verifying “cyber & physical” behavior

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
  • Previously limited to bounded verification only

Prove controller-platform contract via hybrid reachability analysis
  • Done by AFRL

Working on automation and asynchronous model of computation

Assume-Guarantee Contract

Proof of collision avoidance

Assume-Guarantee Contract
Interactive Verification of $I_{AC}$ at Source Code Level

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
}
```

Ongoing work: more automation, moving to asynchronous model of computation.
Challenges and Future Work

Transition and application to realistic systems

Logical Isolation between Verified and Unverified Code

Big Trusted Computing Base (Compilers, Operating Systems, Middleware)

Discovered more complexity and nuances about mixed-criticality scheduling (end-to-end)

Importance sampling for distributed systems

Longer term: Ultra-Large Scale, Fault-Tolerance, Runtime Assurance, Security
Conclusion

Summary
Distributed Adaptive Real-Time (DART) systems promise to revolutionize several areas of DoD capability (e.g., autonomous systems). We want to create a sound engineering approach for producing high-assurance software for DART Systems, and demonstrate on stakeholder guided examples.

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

https://github.com/cps-sei/dart

QUESTIONS?