Modeling, Verifying, and Generating Software for Distributed Cyber-Physical Systems using DMPL and AADL

Sagar Chaki, Dionisio de Niz,

Joseph Seibel

October 6, 2016

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

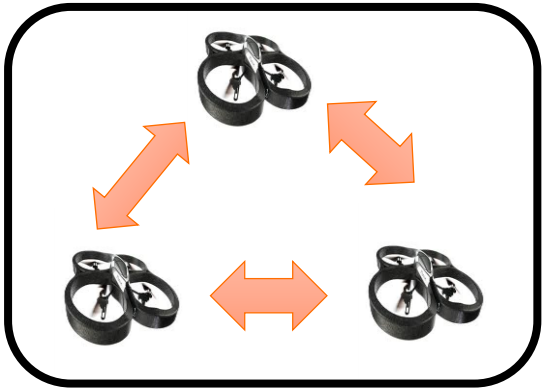
DM-0004058

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 is unachievable via testing or ad-hoc verification

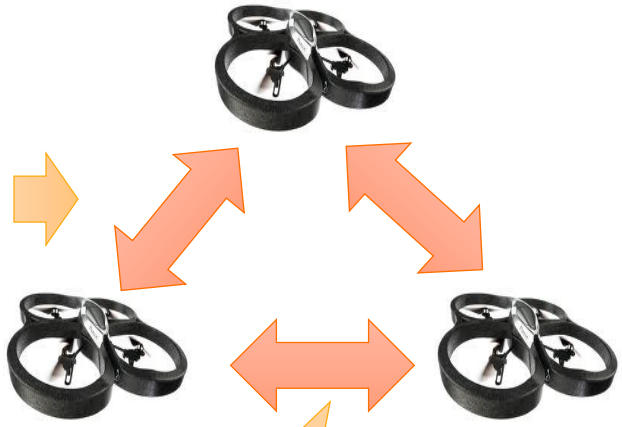
Goal: Create a <u>sound</u> engineering approach for producing high-assurance software for Distributed Adaptive Real-Time (DART)

DART Approach

1. Use DMPL (a DSL we developed) & AADL
2. Enables compositional and requirement specific verification
3. Use proactive self-adaptation and mixed criticality to cope with uncertainty and changing context

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

Formal Description of System & Properties → **Verification** → **Code Generation**



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

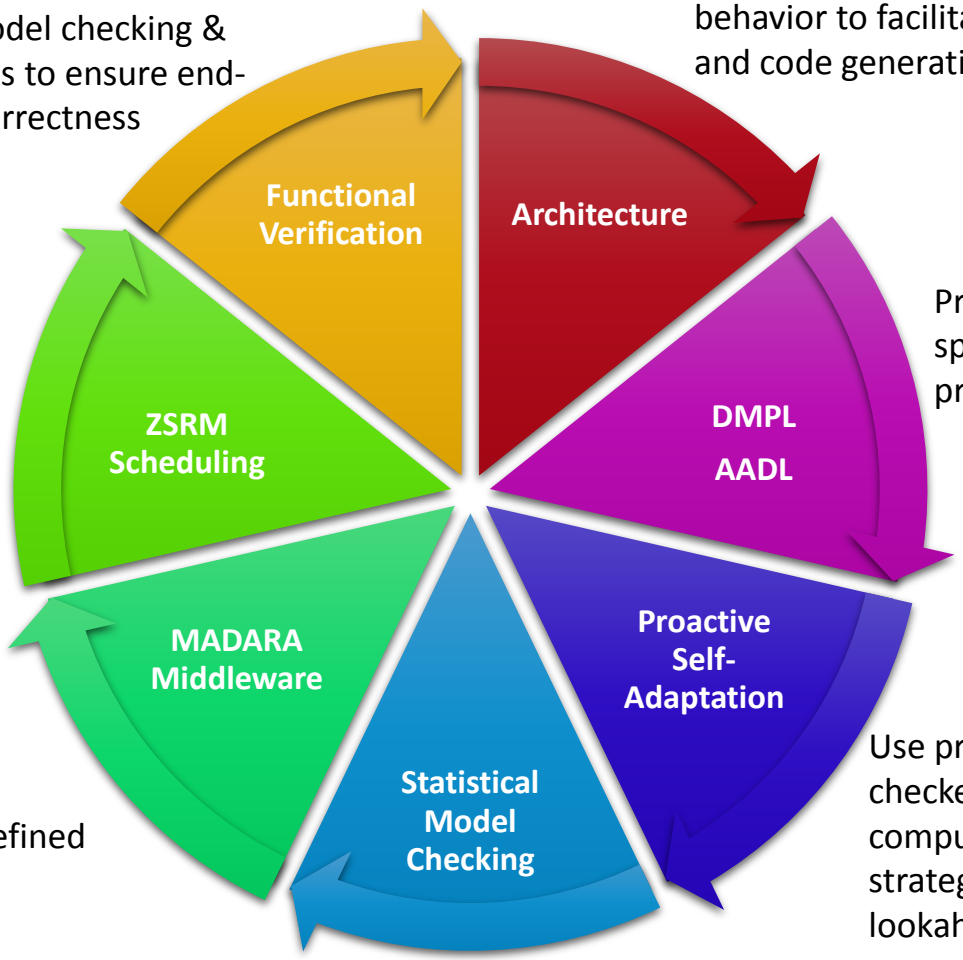
Demonstrate on DoD-relevant model problem (DART prototype)
- Engaged stakeholders
- Technical and operational validity

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

Key Elements of DART

- Parameterized Verification
- Combine model checking & hybrid analysis to ensure end-to-end CPS correctness

Constrain the system structure and behavior to facilitate tractable analysis and code generation



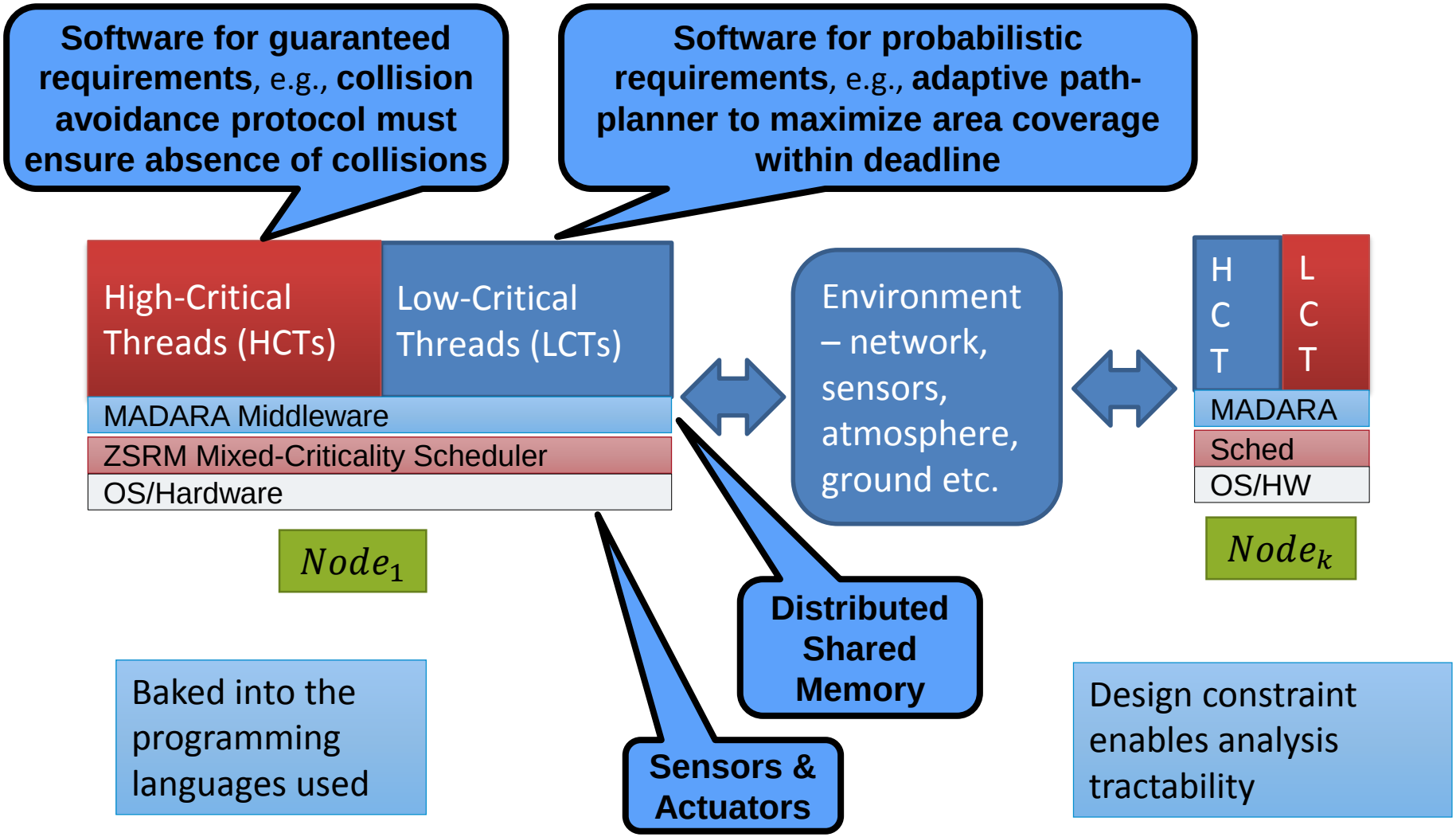
Ensures high-critical tasks meet their deadlines despite CPU overload

Program DART systems and specify properties in a precise manner

Provides efficient distributed shared variables with well-defined data consistency and quality of service

Use probabilistic model checker to repeatedly compute optimal adaptation strategies with bounded lookahead

Evaluate adaptation strategy quality over mission lifetime



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
- Contains DMPL code as sub-language (annex)

DMPL : Behavior (standalone or as AADL annex)

- 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

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

DART Modeling and Programming Language (DMPL)

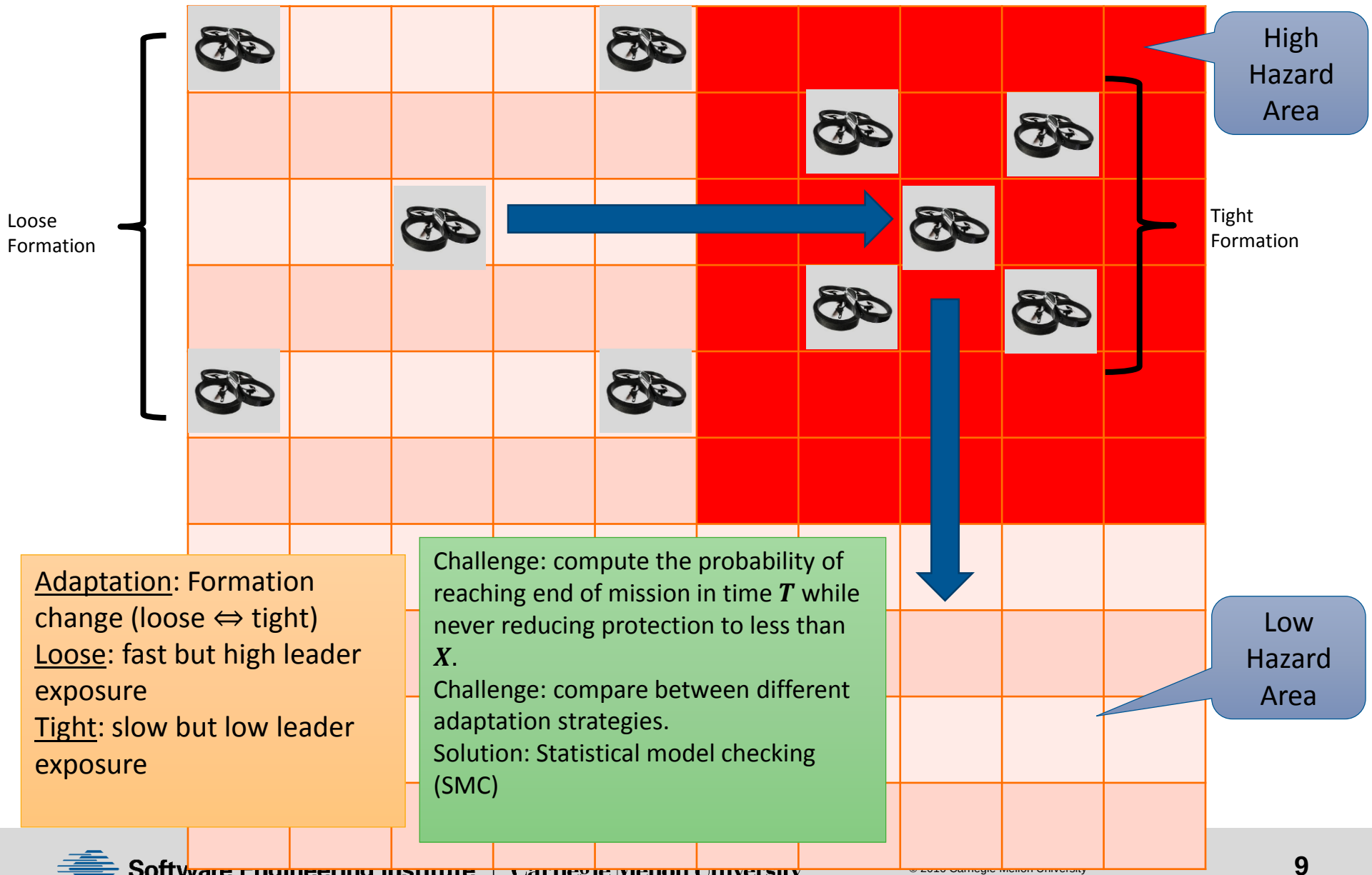
Domain-Specific Language for DART programming and verifying

- 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

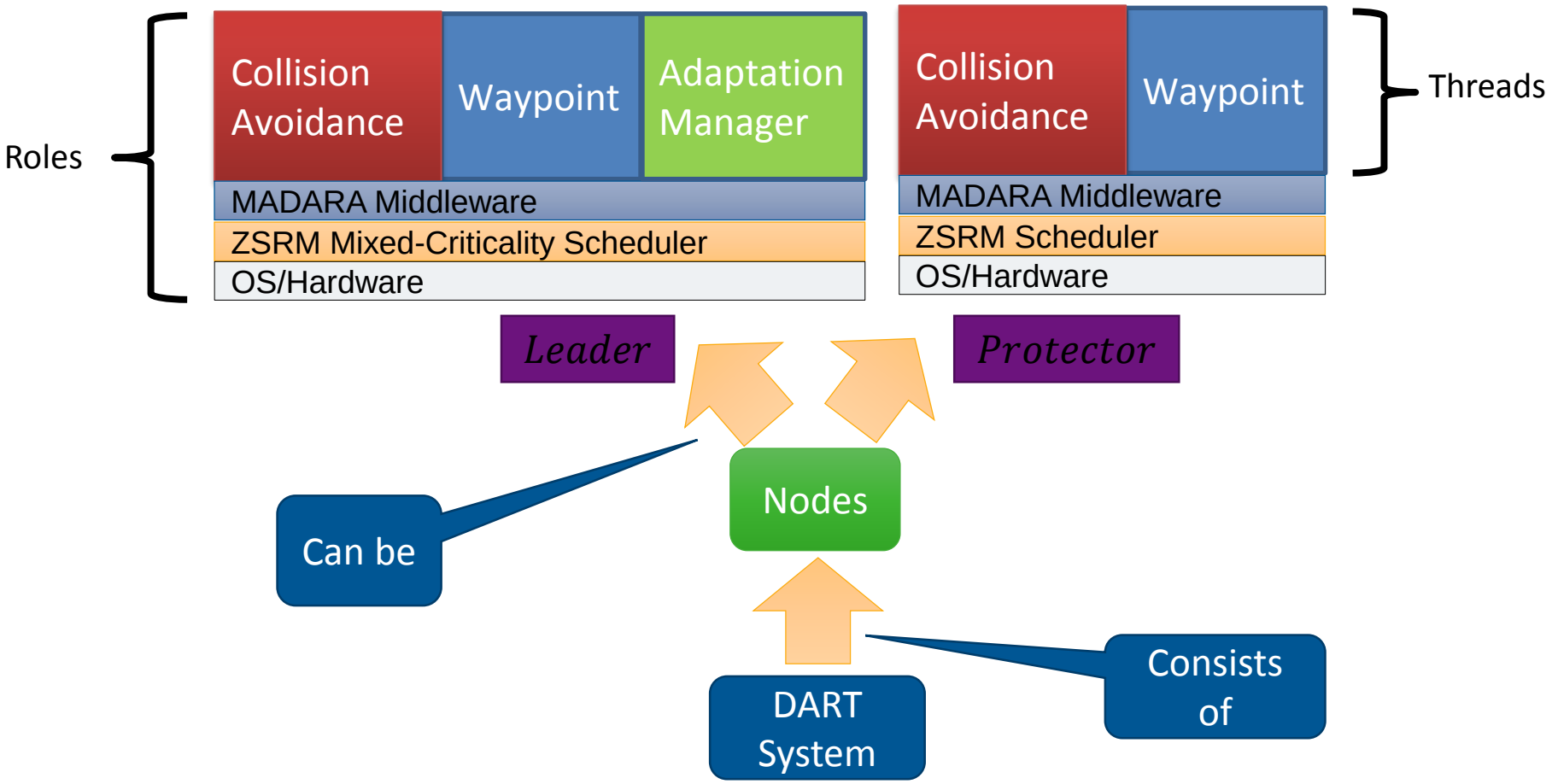
AADL and DMPL supports the right level of abstraction at architecture and code level to formally reason about DART systems

Example: Self-Adaptive and Coordinated UAS Protection



Adaptation: Formation change (loose ⇔ tight)
Loose: fast but high leader exposure
Tight: slow but low leader exposure

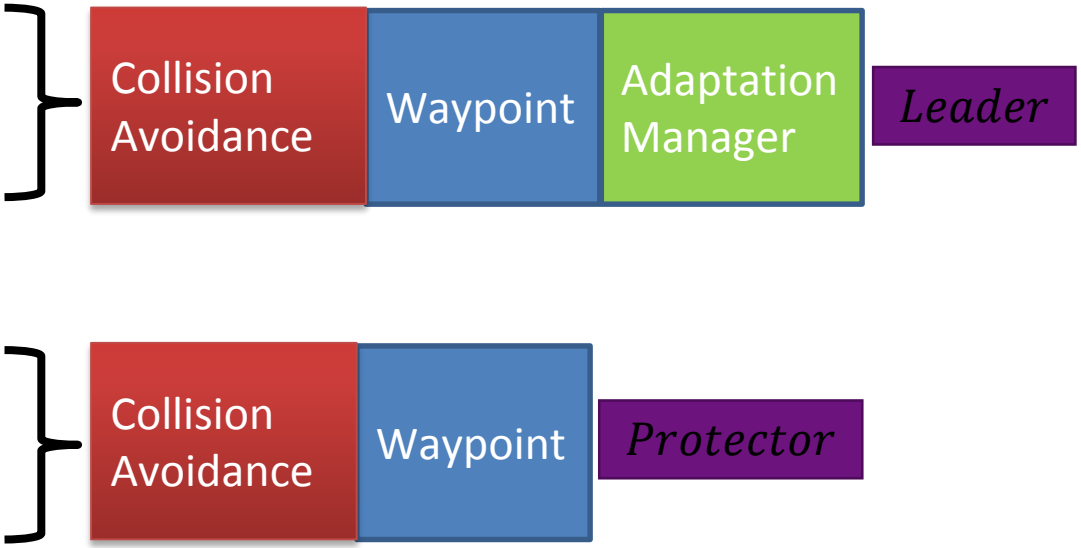
Challenge: compute the probability of reaching end of mission in time T while never reducing protection to less than X.
Challenge: compare between different adaptation strategies.
Solution: Statistical model checking (SMC)



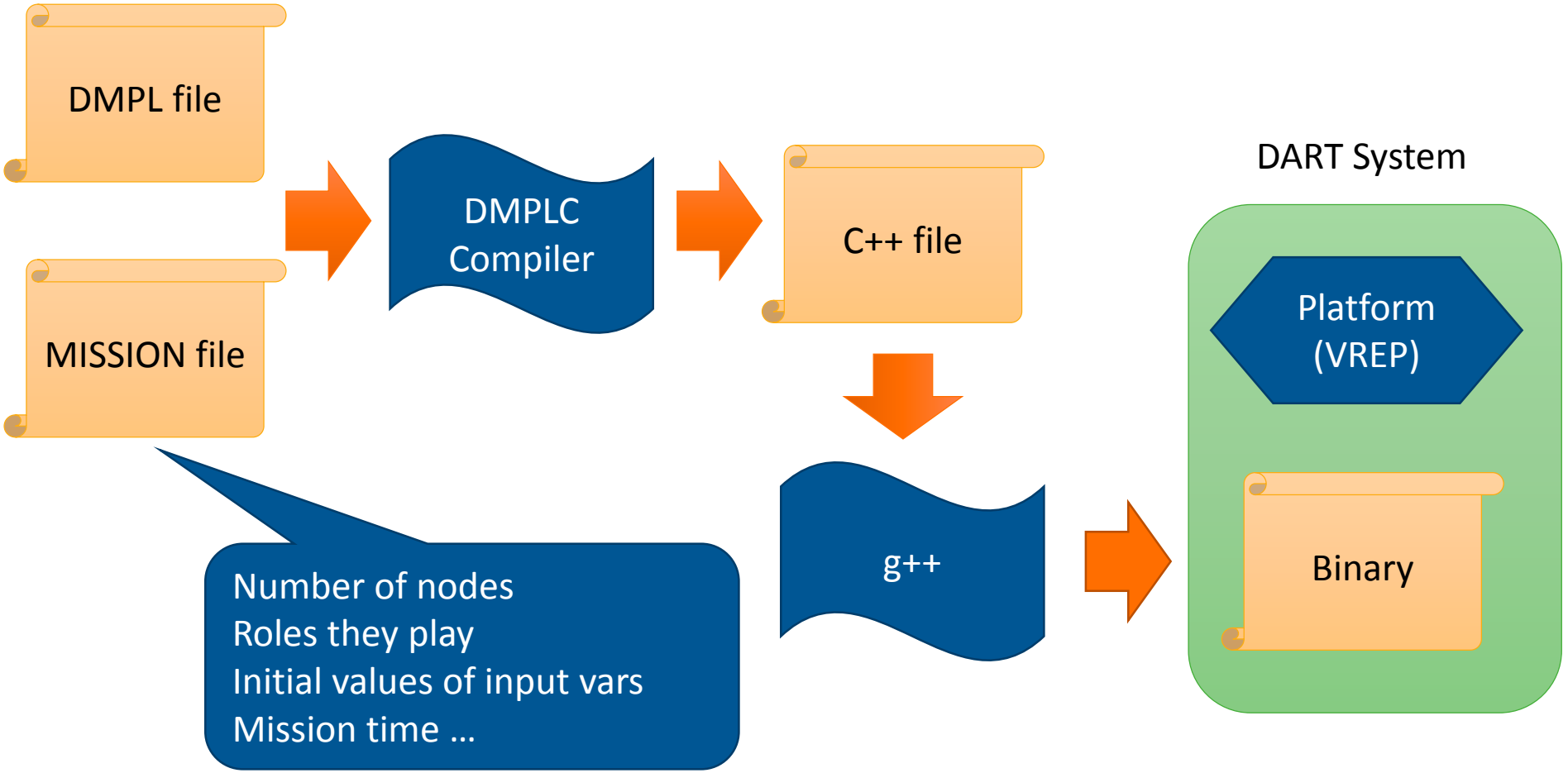
```
node uav {
  local input int x,y;
  local int xp=x, yp=y;
  global lock[X][Y] = {…}

  role Leader {
    thread COLLISION_AVOIDACE {…}
    thread WAYPOINT {…}
    thread ADAPTATION_MANAGER {…}
  }

  role Protector {
    thread COLLISION_AVOIDACE {…}
    thread WAYPOINT {…}
  }
}
```

Shared between threads on the same node. Used to communicate next waypoint.

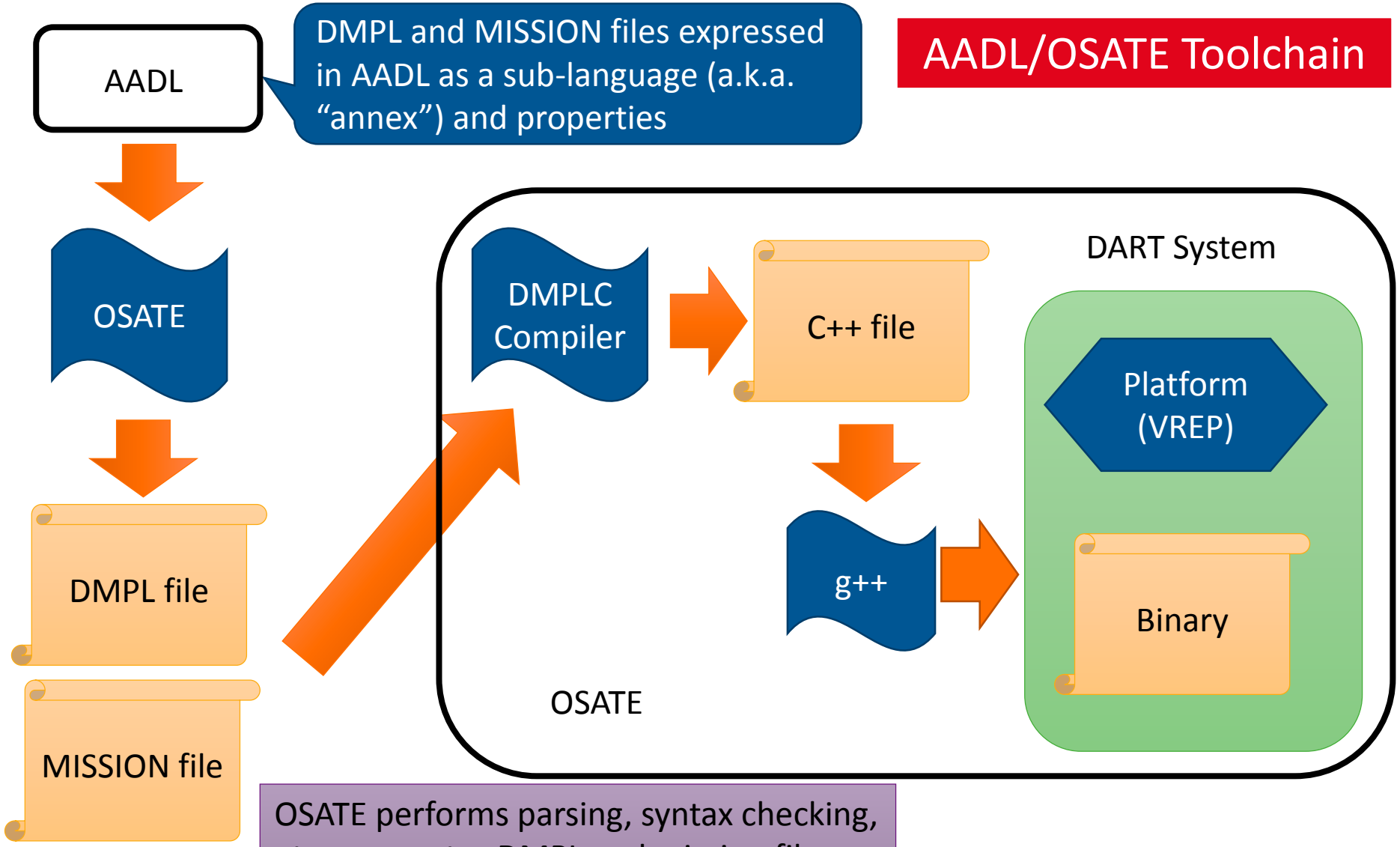
Shared between threads on different nodes. Used for collision avoidance,



Command Line Toolchain



AADL/OSATE Toolchain



OSATE performs parsing, syntax checking, etc., generates DMPL and mission files and invokes the rest of the toolchain

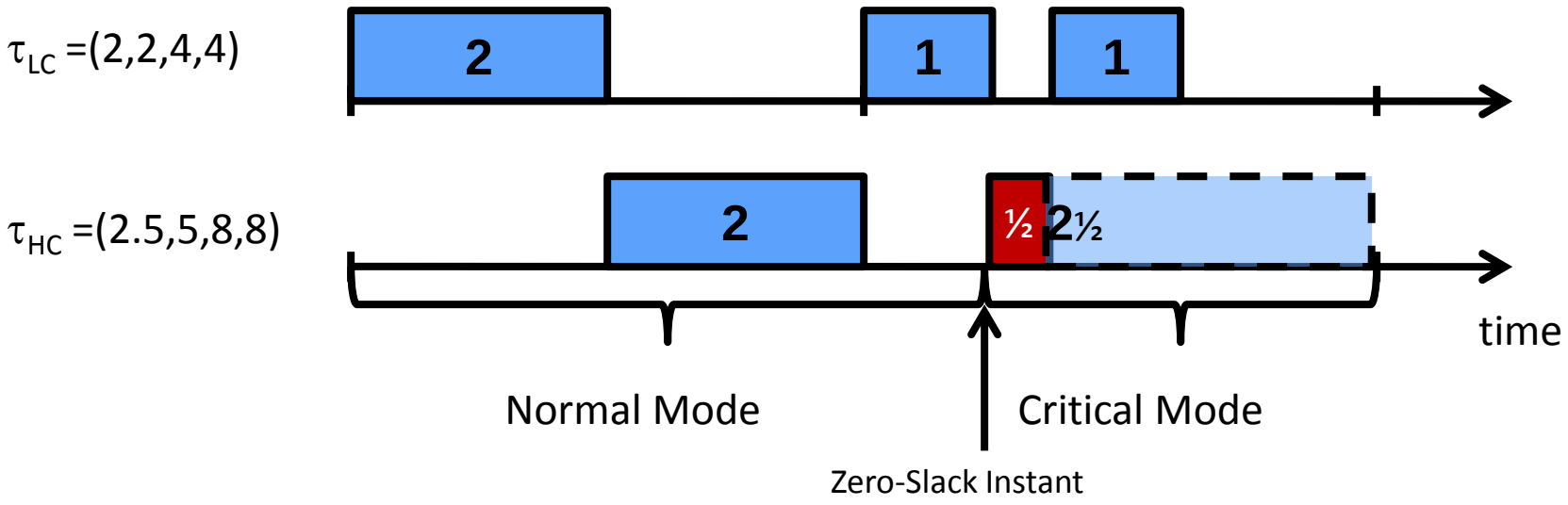
Guarantees deadlines of high-criticality tasks even in overloads

- e.g. too many obstacles to avoid

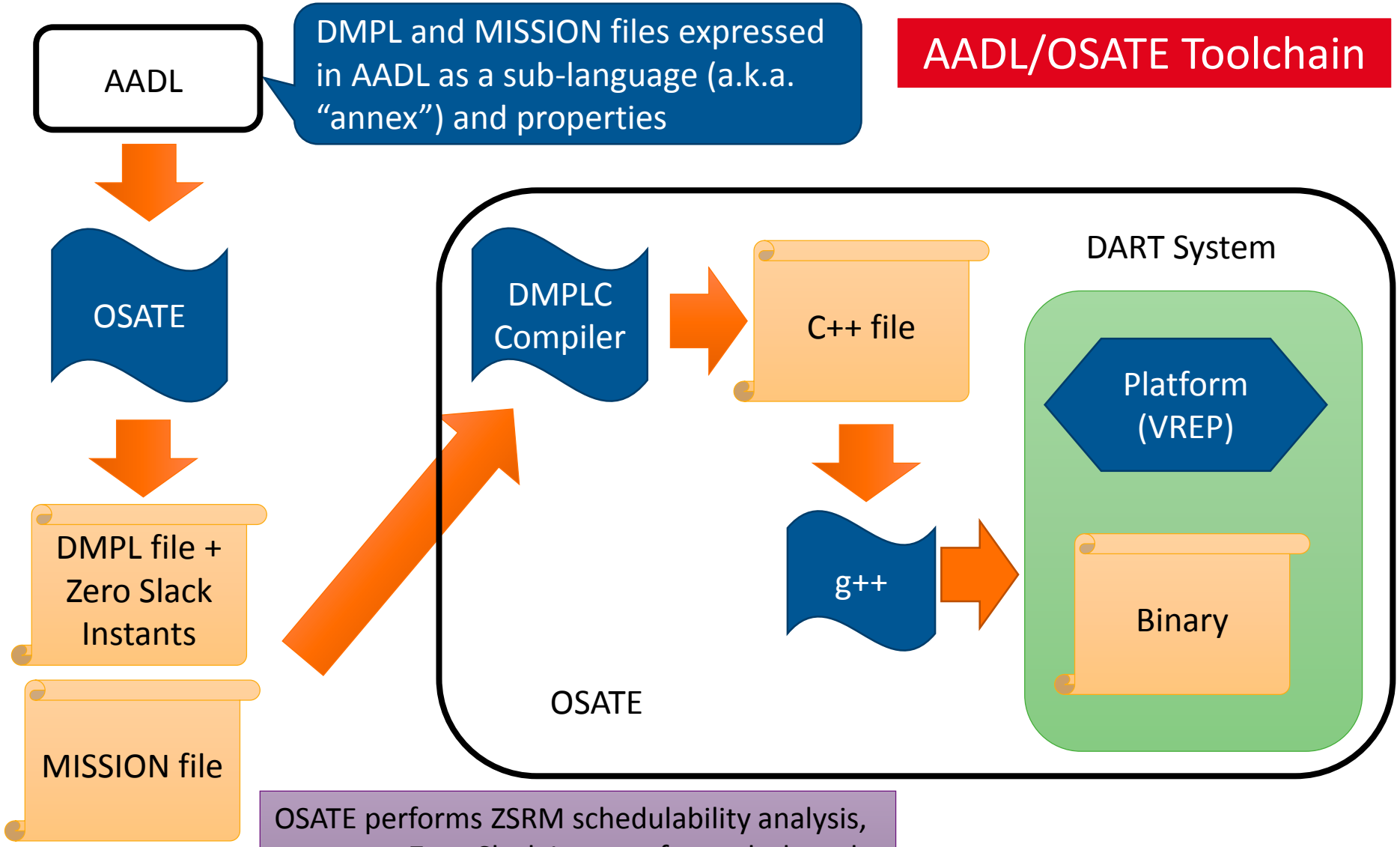
Lower criticality tasks meet their deadlines if no overloads in higher criticality

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

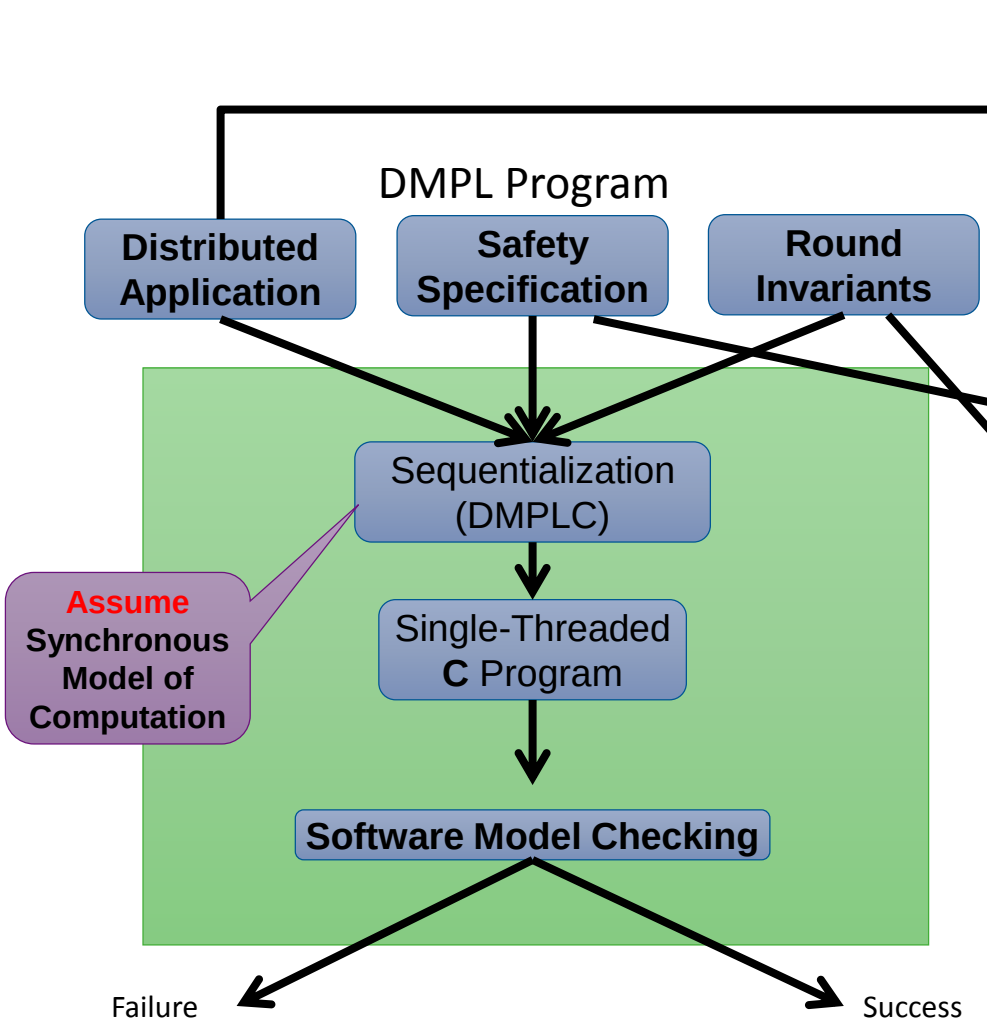
Schedule under RMS, stop lower-criticality tasks at last instant to ensure finishing overload by the deadline (zero-slack)



AADL/OSATE Toolchain



OSATE performs ZSRM schedulability analysis, computes Zero-Slack Instants for each thread and inserts them in the DMPL file




```
node uav {
  local input int x,y;
  local int xp=x, yp=y;
  role Leader {…}
  role Protector {…}

  forall_distinct_nodes(i1,i2)
     (x@i1 != x@i2 || y@i1 != y@i2);

  forall_nodes(i)
     (x@i == xp@i || y@i == yp@i);
}
```

OSATE performs sequentialization (via dmplc) and invokes CBMC

Bounded Model Checking can prove correct behavior up to **a finite number of execution steps** (e.g., rounds of synchronous computation.

Useful to find bugs.

But incomplete. Can miss bugs if we do not check up to sufficient depth.

Unbounded Model Checking can prove correct behavior up to a **arbitrary number of execution steps.**

Useful for complete verification. Will never miss bugs.

But can be expensive to synthesize inductive invariants. Cost can be managed by supplying invariants manually and checking that they are inductive. We have experimented with both approaches.

Parameterized Model Checking can prove correct behavior up to a arbitrary number of execution steps and an **arbitrary number of nodes.**

Useful for complete verification. Will never miss bugs even if you have very large number of nodes.

Very hard in general but we have developed a sound and complete procedure that works for programs written in a restricted style and for a restricted class of properties. This was sufficient to verify our collision avoidance protocol.

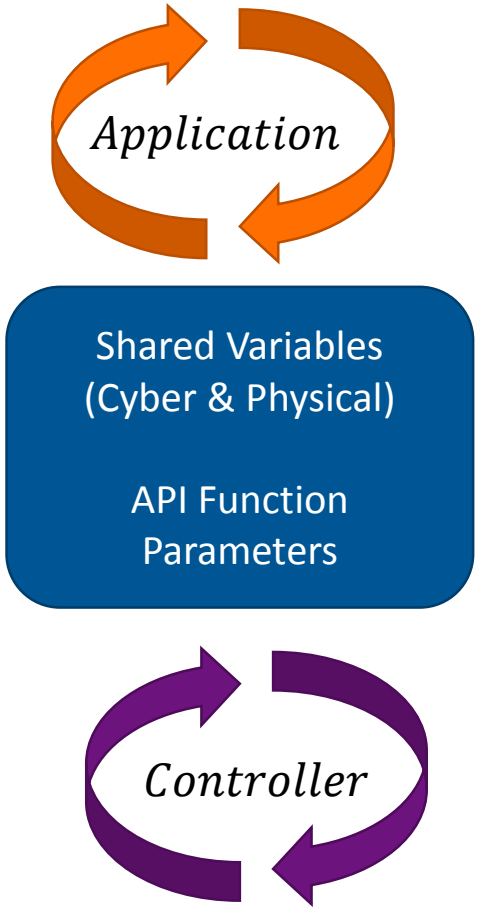
No existing tools to verify (source code + hybrid automata)

- But each domain has its own specialized tools: software model checkers and hybrid reachability checkers
- Developing such a tool that combines the statespace A and C in a brute-force way will not scale

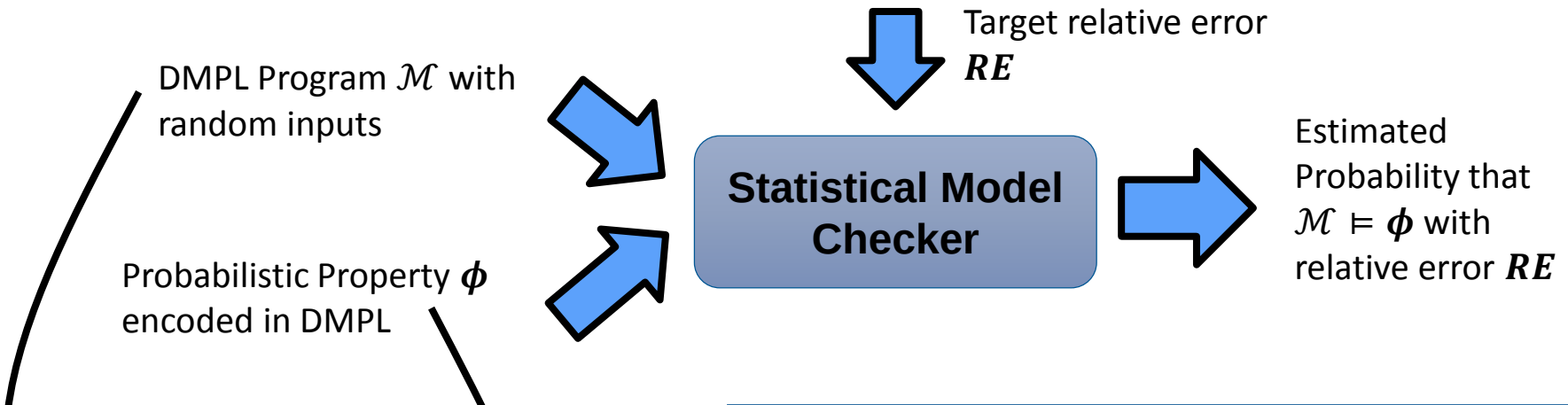
Insight: application and controller make assumptions about each other to achieve overall safe behavior

Approach:

- Use "contract automaton" to express inter-dependency between A and C
- Separately verify that A and C implement desired behavior under the assumption that the other party does so as well
- Use an "assume-guarantee" style proof rule to show the $A \parallel C \vDash \Phi$



Verifying Cyber-Physical Systems by Combining Software Model Checking with Hybrid Systems Reachability. Stanley Bak, Sagar Chaki. International Conference on Embedded Software (EMSOFT), 2016




```
node uav {
  local input int x,y;
  local int xp=x, yp=y;
  role Leader {…}
  role Protector {…}
  double coverage() {…}
  expect at_end (coverage() > 0.8);
}
```

Etimate probability for each property via "Bernoulli Trials"

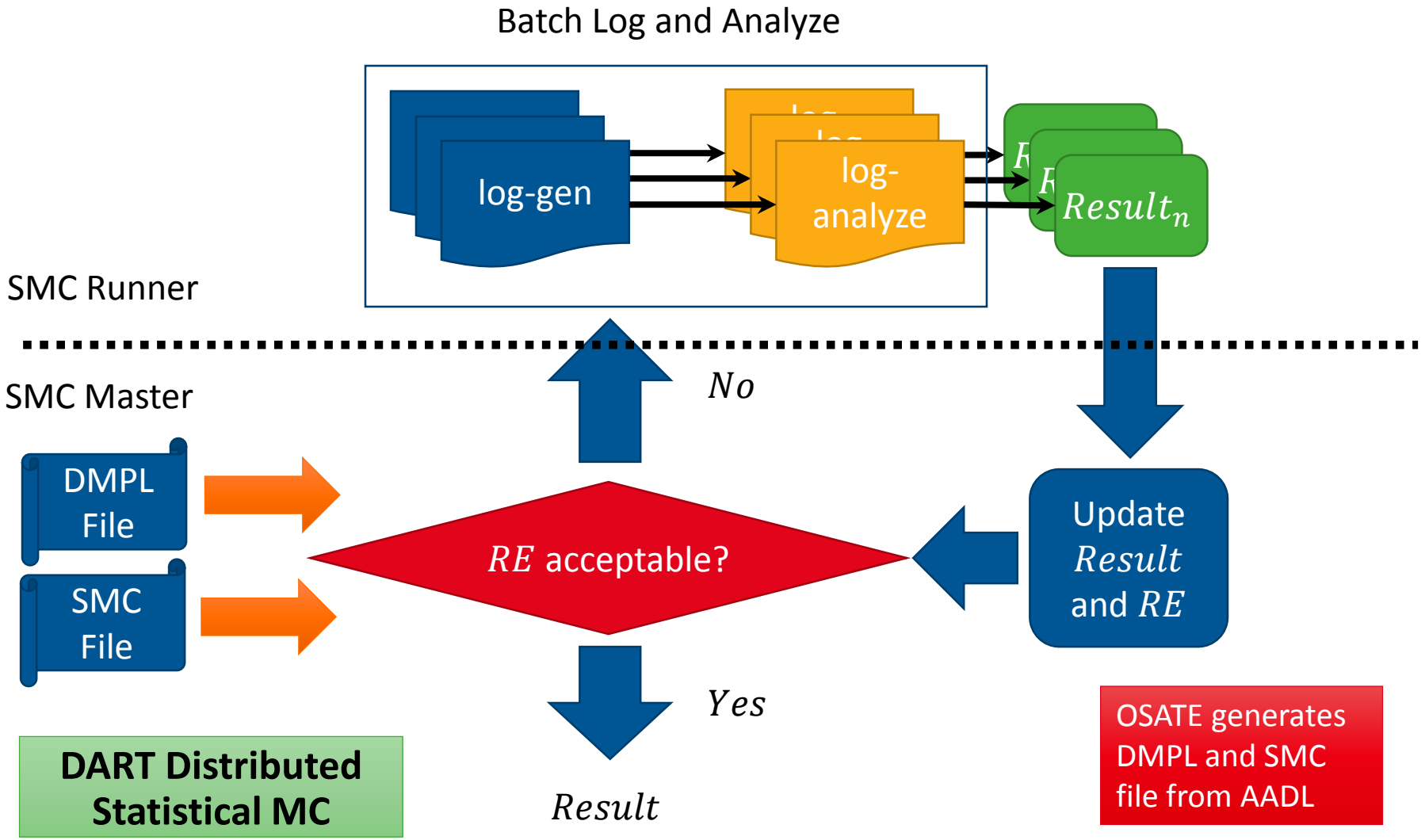
Number of trials depends on

- desired "relative error" (st.dev. / mean)
- true probability of the property

Running trials in parallel reduces required simulation time.

- *SMC Runner* invokes Vrep simulation on each node.
- *SMC Master* collects results and determines if precision is met.
- Simulations run in "batches" to prevent simulation time bias.

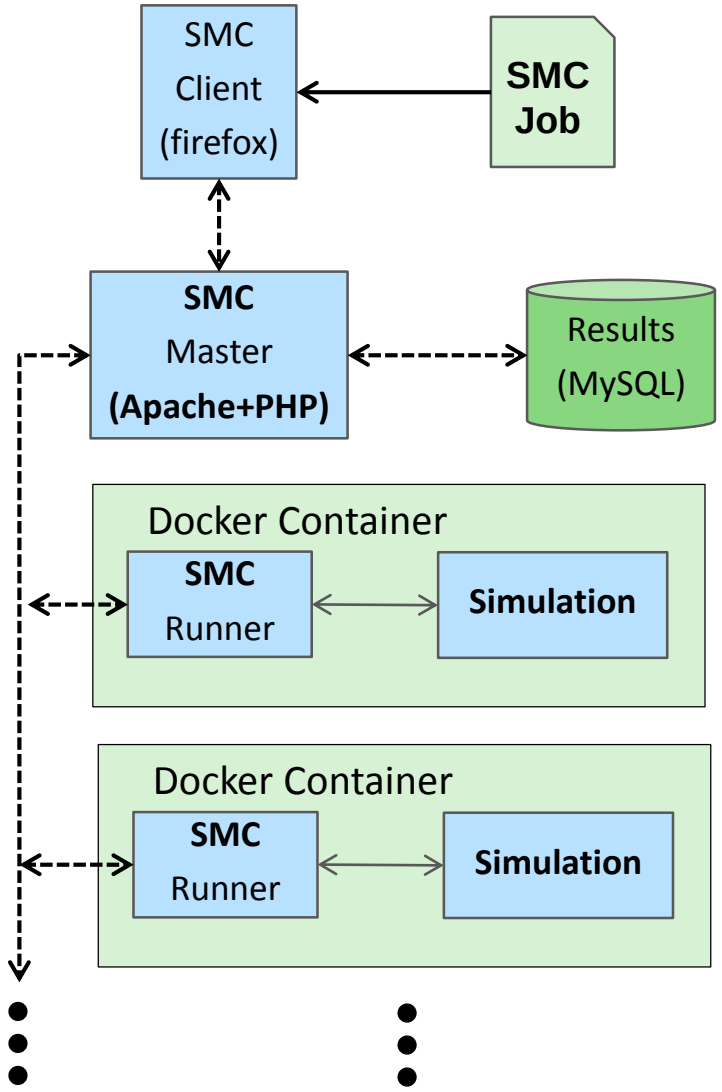
Importance sampling (focuses simulation effort on faults)



Goal: Develop parallel infrastructure for SMC of DART systems

Accomplishments:

- Initial implementation with hand-written scripts for managing multiple virtual machines
- Created master-client SMC architecture with web-based control
  - Each client runs a simulation managed by master
  - Results stored in mysql database.
- Update SMC code generation to new DART/DMPL syntax
- **DEMETER:** More robust infrastructure using "docker"



David Kyle, Jeffery P. Hansen, Sagar Chaki: Statistical Model Checking of Distributed Adaptive Real-Time Software. RV 2015: 269-274

Jeffery P. Hansen, Sagar Chaki, Scott A. Hissam, James R. Edmondson, Gabriel A. Moreno, David Kyle: Input Attribution for Statistical Model Checking Using Logistic Regression. RV 2016: 185-200

Scenarios

- ✓ Stage 0 – basic 3D collision avoidance
- ✓ Stage 1 – Navigation of "ensemble" from Point A to Point B
- ✓ Stage 2 – Navigation of "ensemble" from Point A to Point B through intermediate waypoints
- ✓ Stage 3: Add detection of solid objects, obstacles
  - Assume unobstructed path exists between Point A and Point B
  - Navigation of "ensemble" from Point A to Point B
- ✓ Stage 4: "Map" obstructions in a 3D region
- ✓ Stage 5
  - Add ability to detect location of potential "threats" (analogous to identifying IFF transponders)
  - "Map" threats and obstructions in 3D region
- Stage 6
  - Add mobility to "threats"
  - Maintain overwatch of region and keep track of location of "threats" that move in the environment