High Assurance for Distributed Cyber Physical Systems

Scott A. Hissam, Sagar Chaki, Gabriel A. Moreno
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA
{shissam, chaki, gmoreno}@sei.cmu.edu

Abstract
Distributed Adaptive Real-Time (DART) systems are interconnected and collaborating systems that continuously must satisfy guaranteed and highly critical requirements (e.g., collision avoidance), while at the same time adapt, smartly, to achieve best-effort and low-critical application requirements (e.g., protection coverage) when operating in dynamic and uncertain environments. This short paper introduces our architecture and approach to engineering a DART system so that we achieve high assurance in its runtime behavior against a set of formally specified requirements. It describes our technique to: (i) ensure asymmetric timing protection between high- and low-critical threads on each node in the DART system, and (ii) verify that the self-adaptation within, and coordination between, the nodes and their interaction with the physical environment do not violate high and low criticality requirements. We present our ongoing research to integrate advances in model-based engineering with compositional analysis techniques to formally verify safety-critical properties demanded in safety-conscious domains such as aviation and automotive; and introduce our DART model problem to demonstrate of our engineering approach.

Categories and Subject Descriptors
C.3 [**Special-purpose and Application-based Systems**]: *Real-time and embedded systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.11 [**Software Engineering**]: Software Architectures – *Domain-specific architectures.*

General Terms
Algorithms, Design, Reliability, Languages, Verification.

Keywords
Real-time, architecture, model-checking, self-adaptation.

1. Introduction
Testing and verification, be it statistical or formal, is a founding tenet of all engineering disciplines, including software. Regardless, NIST reported in 2002 that software errors cost the US economy nearly US$60 billion [1]. That isn't because of failures in testing or verification alone, but is because of a systemic error or disconnect (then as it is now) in how software-intensive systems are engineered ("from beginning to 'failure'"). The failure is in the integration of all the sub-disciplines that need to be integrated at the time a system is conceived from inception to transition, and into the system's execution in the physical world. It is here, in the physical world, where resiliency in the face of uncertainty needs to be handled adequately and safely—something that cannot be exhaustively tested *a priori*.

Disastrous failures in embedded systems which interact with the physical world (such as Therac-25, Swedish JAS 39 Gripen, Boeing V-22 Osprey, and Airbus A320-200) have demonstrated the consequences of not adequately verifying the correctness of the software. Embedded systems with critical runtime properties are becoming increasingly distributed, consisting of interconnected nodes (i.e., multi-agent) that collaboratively provide more capability. Furthermore, to achieve their goals even when operating autonomously in uncertain environments, they will have to be self-adaptive. However, coordination, adaptation, and uncertainty pose key challenges for assuring the safety and application-critical behavior of such Distributed Adaptive Real-Time (DART) systems. These challenges are exemplified by:

- Timeliness: performing the right function, and doing so at the right time;
- Resource constraints: limits with respect to power, weight, bandwidth, connectivity, processing capacity;
- Sensor rich: sensing the physical world with sensors that have varying fidelity and can fail;
- Cyber-physical interactions: the physical world is continuous and the digital interface cannot account for everything;
- Autonomous behavior: adapting smartly to events within an agent (e.g., failure), between agents (e.g., loss of a peer) and external (e.g., unexpected obstacle);
- Computationally complex decisions: number of interacting agents and co-dependent decisions made in real-time without causing interference.

We present our ongoing research to integrate compositional analysis techniques with model-based engineering to address these challenges, describe the DART architecture, and introduce our DART model problem, which serves as an end-to-end demonstration of our integrated engineering approach. The end goal is to enable engineering of high-assurance DART systems for safety-conscious systems.

2. Related Work
Since the 1980's, research and development in the fields of Computer-aided Software Engineering (CASE), Model-based engineering (MBE), Model-driven engineering (MDE), Model-centric software engineering (MCSE), and others have attempted to leverage and integrate techniques for requirements, environment specification, architecture definition, domain-specific languages, design patterns, code-generation, analysis, test-generation, simulation, and emulation to support system development [2]. Further, Schmidt recognized the challenges to MBE (generalized to all such approaches) to include

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ECSAW '15, September 07 - 11, 2015, Dubrovnik/Cavtat, Croatia
© 2015 ACM. ISBN 978-1-4503-3393-1/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2797433.2797439

synchronization between the models and source code, debugging at the model level, expression of the design intent, and quality of service properties and the certification of safety properties [2].

Wallnau's work on predictability by construction [3] made two notable contributions with respect to these challenges. The first was treating quality attributes (non-functional requirements such as real-time performance and safety) as a new type of program property that is checkable at compile time. The second contribution was a component language demonstrating the capacity to (a) generate and evaluate an analytic model of a system suitable for quality attribute analysis; and (b) generate executable code for the system if its analytic model did not violate its quality attribute requirements as determined by quality-attribute-specific analyses [4][5]. The intent for this work was to narrow the gap between design-time artifacts (architecture and design specifications, quality-attribute-specific analytic models) and the implementation, through automation and the enforcement of those specifications in the generated code.

Feiler and Gluch's work on the SAE Architecture Analysis & Design Language (AADL) is used to model both software and hardware for embedded, software-reliant systems [6]. It supports MBE analysis practices, encompassing software system design, integration, and assurance. Furthermore, AADL is extensible to additional analysis and specification techniques necessary for domain-specific application.

3. DART System

DART systems are interconnected and collaborating systems that continuously must satisfy guaranteed and highly critical requirements, while at the same time adapt, smartly, to achieve best-effort and low-critical application requirements when operating in dynamic and uncertain environments.

In this section we describe the runtime architecture and model of computation (MOC) for a collection of agents (i.e., nodes) in a DART system, the role of self-adaption, and the model problem which serves as the context for verifying the critical properties of the demonstration application.

3.1 Architecture and MOC

The runtime architecture of a DART system used in our approach is shown in Figure 1. A DART system consists of a finite set of physically separate nodes communicating wirelessly. The software on each node consists of three layers – application, middleware, and real-time scheduler – running on an OS.

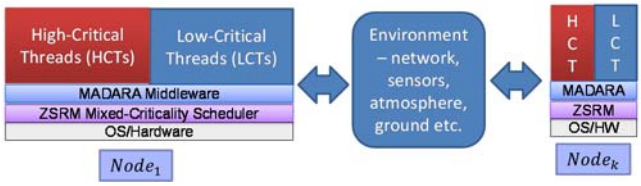

Figure 1: DART Runtime Architecture

At the top, the application consists of a set of high-critical threads (HCTs) and low-critical threads (LCTs). All threads are real-time and periodic, scheduled with fixed-priority Zero Slack Rate Monotonic (ZSRM) [9] scheduling. ZSRM scheduling ensures that HCTs (which implement guaranteed requirements) do not miss deadlines (by starving LCTs if needed) even under overload conditions. This asymmetric temporal protection between HCTs and LTCs is crucial for the valid use of software model checking (SMC) to ensure functional correctness of HCTs, since SMC assumes that HCTs never miss deadlines.

The application threads (both within and across nodes) communicate via shared variables. The MADARA middleware, at the next software layer, provides the necessary distributed shared variable abstraction [10]. Values written to a shared variable by a node are propagated via network messages to the remaining copies of the variable at the other nodes. Lamport clocks are used by MADARA to ensure that reads and writes to shared variables occur in a sequentially consistent manner. Again, precise memory consistency is needed for the sound application of functional verification techniques.

At the lowest level is the ZSRM scheduler which provides correct runtime enforcement of asymmetric temporal protection between HCTs and LCTs. It also schedules the special thread used by MADARA on each node to update values of shared variables received from other nodes. Finally, the ZSRM scheduler provides criticality-and-priority-inversion-preventing mutex mechanisms to "lock" the shared variables.

Such locking is needed to implement our DART MOC which works as follows. At the beginning of each period, a thread locks all shared variables, reads their values, and releases the lock. It then computes new values of shared variables by executing the corresponding thread function. Finally, the thread locks the shared variables again, updates their values, and releases the lock. Thus, our MOC is designed such that each periodic execution of a thread is semantically equivalent to a transaction. This reduces the system statespace and improves tractability of verification.

3.2 Self-Adaptation

Adaptation is a key aspect of DART systems that is required to deal with different mission stages and environment conditions. For the kinds of missions that these systems will carry out, there are elements of the environment where they will operate that cannot be known before mission execution. For example, in the DART model problem (described in the next section), it is not possible to know beforehand what the threat level of the different areas the swarm will go through is going to be. Instead, these environment conditions are discovered as the mission progresses; and even then, knowledge about them is subject to uncertainty. Therefore, the system must self-adapt at run time as it is not possible to plan adaptations in advance.

We argue that self-adaptation in DART systems has to be proactive for two reasons. First, adaptations requiring coordination and/or physical actuation may take time (e.g., a formation change), so they have to be started proactively. Second, adaptation decisions taken at any point impact future outcomes (e.g., adaptation resulting in higher fuel consumption reduces range). Thus, adaptation decisions must take into account not only the immediate conditions, but also expected near future needs.

Our approach to self-adaptation in DART is based on the explicit monitor-analyze-plan-execute (MAPE) control loop [7]. Although our end goal is to further decentralize the different steps of the loop, initially we are using the master/slave pattern for decentralized control, in which the analysis and planning steps are centralized, whereas the monitoring and execution steps are decentralized [13]. In particular, the knowledge that adaptation decisions are based on, and the communication of adaptations to be executed are captured by the MADARA middleware in the DART architecture. The analysis and planning steps implement the core of the proactive self-adaptation using an approach that deals with the latency of adaptations and with environment uncertainty [12].

3.3 Model Problem

The model problem involves collaborating swarms of autonomous agents (i.e., UAVs) that require both guaranteed, and best-effort requirements. Figure 2 depicts two fleets of agent-based swarms, each having independent objectives but sharing the same goal (e.g., search and rescue). The swarm is made up of a number of agents. Agents within the swarm must collaborate to maintain separation so as not to collide with one another (a critical safety-property) while maintaining a formation so as to best protect a leader (a best-effort property) during the time it takes to reach objectives along the swarm's route. We consider the distributed algorithm to maintain physical separation the highly critical property to be guaranteed, and the protection property to be the low(er) criticality property we also want to satisfy. In all cases, the real-time high-critical deadlines cannot be missed.

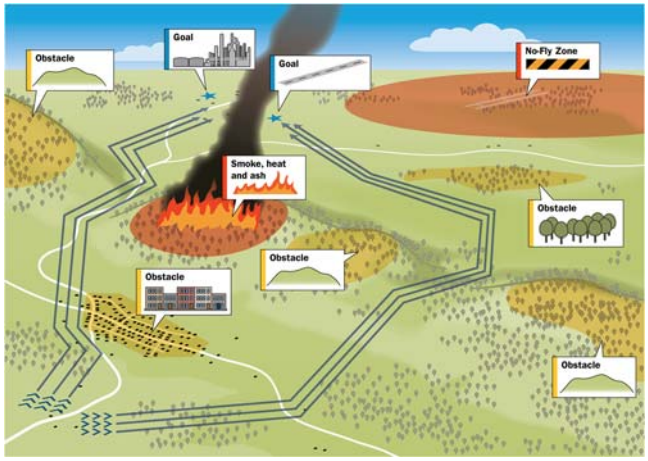

Figure 2: Context for DART Model Problem

4. DART Software Engineering Approach

The approach to engineering a DART system verifies that the safety-critical properties of the system are not violated, and are assured by objective evidence. Thus, the DART approach focuses on the artifacts that are produced during the construction process and the type of analyses conducted on those artifacts.

From a tooling perspective, DART seeks an Integrated Development Environment (IDE) that supports the specification of safety-critical and quality-attribute requirements, architecture and code design elements, tests (both unit and integration), along with code generation, compilation, deployment and debugging. Further, it is necessary to be able to trace such specifications backwards and forward through all the transformations supported by the IDE.

The DART software engineering approach is integrating:

- Mixed-criticality analysis to verify the asymmetric timing protection and schedulability of threads with different criticality that share resources (e.g., CPU(s)) in a single node [9];
- Domain-specific language (DSL) and safety specification notation for distributed applications comprising multiple nodes [10] with higher-level architecture descriptions in AADL;
- Model-driven verifying compilation system which generates C++ code if the safety properties specified for the application are verified successfully by a software model checker [10];
- Proactive, latency-aware self-adaptation mechanism as a means for assuring resiliency when dealing with planned mode changes and unexpected events from the physical environment during runtime [12];
- Statistical model checking for computing the bounded probability that best-effort properties of the system are within the application's requirements despite the stochastic behavior of the environment [11].

A complete demonstration of an integrated approach to engineering a DART system is staged in two phases. The first phase is to use the analysis techniques listed above and incrementally improve upon them to address their respective limitations when applied to a DART system. The second phase will use the lessons from those improvements to drive requirements and improvements (or extension) to AADL (i.e., ALISA[1], a plug-in currently under development for AADL's IDE) so that safety-critical properties such as the ones verified during the first phase can be properly encoded and traced through AADL's IDE during the end-to-end engineering of a complete DART system.

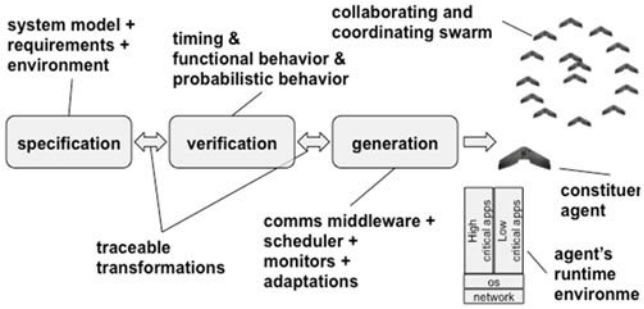

Figure 3: Abstract Tool Chain for DART (Phase 1)

Figure 3 shows the initial tool chain for Phase 1. The tools and techniques are founded on those discussed in [9] through [12]. System level specifications are encoded in our domain-specific language from [10]. Specification of application level requirements, and the environment come from subject matter experts. Initially, that is manually crafted into our DSL so that the necessary verification steps can be performed. In Phase 2, the plan is to encode and formalize that knowledge in AADL.

Continuing in Figure 3, each verification tool takes its respective inputs from the specifications to perform the analysis (i.e., timing, functional, and probabilistic). If verification fails, a trace back to the specification(s) that formed the basis for a failed check is identified. Here, verification takes these three forms:

- Mixed-criticality temporal protection mechanisms between runtime threads hold; passing these checks means that real-time deadlines for high-critical tasks are guaranteed to be met;
- Guaranteed property: physical separation among multiple agents; passing these checks mean that the invariants for the collision avoidance algorithm used by the swarm hold;
- Best-effort property: agents provide adequate physical protection to the leader in a probabilistic environment; passing these checks mean that over a given mission time, the probability of mission failure due to insufficient protection is below a specified threshold.

[1] ALISA supports the specification of goals, requirements, and claims; concepts of obstacles, hazards, vulnerabilities, challenges, and defeaters; and concepts of static analysis, verification activity, evidence, and counter evidence.

Code is generated for the target hardware platform after passing the verification checks (see Figure 4). It includes all the functional code (allocated to one or more threads as identified from the specification), and interfaces to the underlying operating system scheduler and networking services. Additionally, monitoring code supporting each of the requirements is generated:

- Guaranteed requirement: when the `ASSERT()` generated from the `require()` property could not be verified (due to scalability limits of the model checker). However, no monitoring code is generated if the property is verified;
- Best-effort requirement: when the variables used to evaluate the `expect()` property specification that are also used by the self-adaptation mechanism are passed to the adaptation manager's monitoring interface.

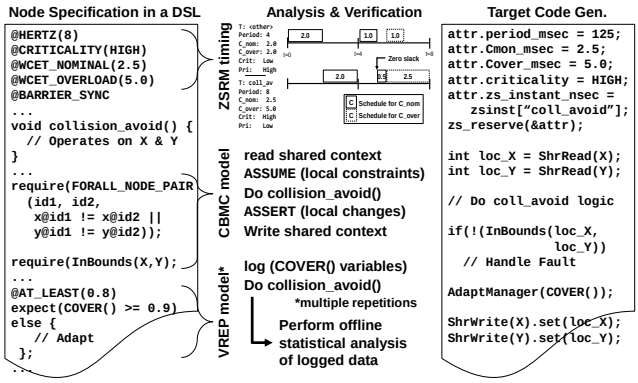

Figure 4: Code Generation based on Specifications and Analysis

For properties that require analysis across variables spanning more than one node (e.g., the require (`FORALL_NODE_PAIRS`) node specification in Figure 4), it is impractical to share those variables across those nodes at runtime—for now, no monitoring code for the target platform is generated in those cases. In the case that a property is deemed intractable and node-spanning, this generates a warning that would need to be addressed by a human, requiring changes in requirements or design.

The initial self-adaptation mechanism for the model problem deals mainly with the decision to change the formation of the swarm. Different formations provide tradeoffs between different qualities (e.g., protection vs. speed), which are desired for different stages of the mission. The adaptation mechanism must deal not only with planned mission events, but also with uncertain environment conditions (e.g., an unplanned forest fire that is on the current route must be avoided).

5. DART Future Work

Both phases are intended to build upon the properties that can be verified for the individual agent as well as the composed swarm of agents. In Phase 2 our work will be extended to properties that can be verified among the fleet of swarms. Further, we expect to:

- Encode our DSL as AADL requirement specifications which are then mapped to generated specifications for both analytic models and code, and to support debugging and back tracing;
- Use the architectural models that are already in our approach to do architecture-based self-adaption as in Rainbow [8];
- Introduce additional adaptation tactics and machine learning for the adaption manager;
- Support asynchronous multi-agent coordination in guaranteed behavior as it applies to unbounded checks;
- Reduce the total number of samples needed for a given level of precision for statistical model checking.

Finally, more research is needed to scale our approach to more numerous, interconnected critical systems.

6. Acknowledgments

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. This material has been approved for public release and unlimited distribution. DM-0002510

7. References

[1] Tassey, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing ", Technical Report NIST 2002-10, National Institute of Standards and Technology, May 2002.

[2] Schmidt, D.C., "Guest Editor's Introduction: Model-Driven Engineering," IEEE Computer 39 (2), Feb. 2006.

[3] Wallnau, K.C., "Predictability by Construction: Working the Architecture/Program Seam," Mälardalen University Press Dissertations, No. 85, Sept, 2010.

[4] Moreno, G.A., Hansen, J., "Overview of the Lambda-star Performance Reasoning Frameworks." CMU/SEI-2008-TR-020, Software Engineering Institute, Carnegie Mellon University, Feb. 2008.

[5] Chaki, S., Ivers, J., Sharygina, N., Wallnau, K. "The Comfort Reasoning Framework". 17th International Conference on Computer Aided Verification, Springer, LNCS, vol. 3576, July 2005.

[6] Feiler, P.H., Gluch, D.P., "Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language," Addison-Wesley Professional, 2012.

[7] Kephart, J.O., Chess, D.M., "The Vision of Autonomic Computing." Computer 36.1 (2003): Jan. 2003.

[8] Garlan, D.; Schmerl, B.; Cheng, S-W. "Software Architecture-based Self-adaptation." In Autonomic computing and networking, Springer, 2009.

[9] de Niz, D.; Lakshmanan, K.; Rajkumar, R., "On the Scheduling of Mixed-Criticality Real-Time Task Sets," Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, Dec. 2009.

[10] Chaki, S., Edmondson, J., "Model-Driven Verifying Compilation of Synchronous Distributed Applications," Model-Driven Engineering Languages and Systems (MODELS), Springer, LNCS, vol. 8767, Oct. 2014.

[11] Hansen, J.P., Wrage, L., Chaki, S., de Niz, D., Klein, M., "Semantic Importance Sampling for Statistical Model Checking," Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, LNCS, Apr. 2015.

[12] Moreno, G.A., Camara, J., Garlan, D. & Schmerl, B. "Proactive Self-Adaptation under Uncertainty: a Probabilistic Model Checking Approach." European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE) Sept. 2015 (to appear).

[13] Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., & Göschka, K. M. "On patterns for decentralized control in self-adaptive systems." In Software Engineering for Self-Adaptive Systems II. Springer Berlin Heidelberg, Jan 2013.