Statistical Model Checking of Distributed Adaptive Real-Time Software

David Kyle Jeffery Hansen Sagar Chaki
{dskyle,jhansen,chaki}@sei.cmu.edu

Abstract. The problem of estimating quantitative properties of distributed cyber-physical software that coordinate and adapt to uncertain environments is addressed. A domain-specific language, called DMPL, is developed to both describe such a system and a target property. Statistical model checking (SMC) is used to estimate the probability with which the property holds on the system. A distributed SMC tool is developed and described. Virtual machines are used to implement a realistic execution environment, and to isolate simulations from one another. Experimental results on a coordinated multi-robot example are presented.

1 Introduction

A Distributed Adaptive Real-Time (DART) system consists of a set of physically disjoint nodes that communicate and coordinate to achieve a set of objectives, and increase the likelihood of achieving these objectives (i.e., success) under an uncertain environment via self-adaptation. Given a stochastic system $\mathcal{M}$, an event Φ in its execution, and an error bound RE, statistical model checking (SMC) [10] is a systematic use of Monte-Carlo simulations to estimate the probability of Φ with an error of no more than RE. In this paper, we present and evaluate an approach for statistical model checking of DART systems (DARTs).

We make three contributions. First, we develop a language called the DART Modeling and Programming Language (DMPL). A DMPL program $\mathcal{P}$ is a triple $(\mathcal{M}, \Phi, T)$, where $\mathcal{M}$ is the DART system, and T (a time limit) and Φ (a predicate over executions of $\mathcal{M}$) express the target property. Our goal is to estimate the probability $\mathbf{p}$ that Φ holds on a random execution of $\mathcal{M}$ of duration T. Second, we develop a compiler, DMPLC, that given a DMPL program $\mathcal{P}$, generates: (i) a log generator *LogG*; a run of *LogG* produces one log for each node of $\mathcal{M}$; and (ii) a log analyzer *LogA* that combines all the logs from one execution of *LogG*, and produces the value of Φ at time T. Finally, we implement a distributed SMC tool, SMCD, that uses *LogG* and *LogA* to estimate $\mathbf{p}$ with a target precision.

We evaluated our approach on a DART example with mobile robots, where success involves avoiding collisions, while maximizing speed, and minimizing exposure to environmental hazards. For our experiments, we use the ZSRM [7] scheduler, the V-REP [9] physics engine, and the MADARA [5] middleware. Our approach easily handles this example system with 5 nodes, each running 3 threads, and should scale to much larger systems. We also demonstrated running on clusters with 5 VMs. Further details are provided in Sec. 4.

```
 1 pure double coverage()
 2 {
 3   double cover = 0.0, dist, lat, lng, xd, yd;
 4   lat = GET_LAT();
 5   lng = GET_LNG();
 6   forall_other(nid) {
 7     xd = GET_LAT()@nid-lat;
 8     yd = GET_LNG()@nid-lng;
 9     dist = LL2M * sqrt(xd*xd + yd*yd);
10     if(dist == 0.0) continue;
11     cover = cover+asin(RAD/dist)/M_PI;
12   }
13   return cover;
14 }
15
16 @AtLeast(0.5) expect(coverage() > 0.9);
```

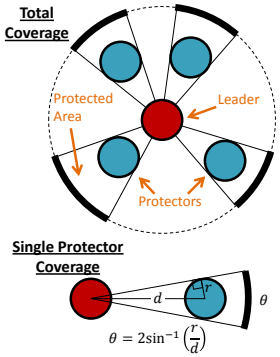

Fig. 1. A DMPL property for coverage, with geometric justification.

Related Work. SMC has been applied to various types of "models": stochastic hybrid automata [3], real-time systems [4], and Simulink models of cyber-physical systems [2]. Our work bridges the gap between what is analyzed and what will be executed. In addition, unlike current distributed SMC tools, SMCD handles dynamic addition and removal of simulators. Younes [11] showed that naive parallelization of SMC is incorrect due to a bias caused by differences in execution time, and that [11] this bias is eliminated by performing simulations in "rounds". Bulychev, et. al. [1] proposed two optimizations to round-based parallelization – batching and buffering. Since we simulate the actual system for time T, the time to perform a simulation is relatively large, and the simulation times for "true" and "false" results are close. Hence, we do not believe that batching or buffering will be helpful for us, but we do apply the basic round-based approach to avoid bias. SMC performance can also be improved via techniques orthogonal to ours, e.g., importance splitting [6], and importance sampling [8].

2 The DMPL Language

A DMPL program $\mathcal{P}$ is a triple $(\mathcal{M}, \Phi, T)$. The system $\mathcal{M}$ is a triple (V, F, T), where V is a set of shared variables; F is a set of procedures (functions); T is a set of threads. DMPL defines $\mathcal{M}$ and Φ, through a mostly C-like syntax; however:

- DMPL does not support pointers, to avoid variable aliasing complications.
- DMPL functions in F can be declared `pure`. These functions cannot modify V; DMPLC will reject code that violates this.
- DMPL defines threads statically, like functions, but with the `thread` keyword. DMPL automatically spawns these threads. DMPL threads are inherently periodic; each thread's code runs within an implied infinite loop.
- DMPL variables, comprising V, may be defined as `local` or `global`. Threads on a given node share `local` variables, while threads across all nodes share `global` variables. DMPL uses a "read-execute-write" computation model. Threads operate on cached copies of these variables, read atomically at the start of each period, and write atomically back at the end. Additionally,

each node publishes its own version of `global` variables, which other nodes cannot "overwrite". Nodes, however, can "read" others' versions.

- DMPL supports defining a Φ as part of its source, using `expect` clauses. These clauses specify a Boolean expression, over values in V and returned from `pure` functions, whose truth SMC will evaluate.
- DMPL can call arbitrary `extern` C++ functions; however, these functions cannot directly access V. They may be labeled `pure`, indicating that they are safe to call from `expect` clauses, i.e., they do not affect runtime behavior, only gather information about it. DMPLC does not enforce this contract.

DMPLC creates *LogG* and *LogA* from a DMPL program. *LogG* includes an observer thread which periodically logs (with a timestamp) all variables in V appearing in `expect` clauses. The read-execute-write model ensures consistent state observation. Functions declared `pure` are executed in either *LogG* or *LogA* as needed. *LogA* uses the timestamps to cross-reference the logs and evaluate each `expect` clause. Figure 1 shows an example DMPL program for a scenario used later in Experiment 1. The `@AtLeast(0.5)` annotation in the `expect` clause means that the specified coverage property should hold true at least 50% of the the time for a mission run to be successful. DMPL also supports an `@AtEnd` annotation; such `expect` clauses must hold true at the end of a mission run.

3 Statistical Model Checking (SMC)

The goal of SMC is to estimate the probability that the property Φ holds in the system $\mathcal{M}$. We model this as an indicator function $I_{\mathcal{M}\models\Phi} : x \to \{0, 1\}$ where $x \sim f$ (i.e., x is a random input vector distributed by f). We can then state the SMC problem as determining the probability $\mathbf{p} = E[I_{\mathcal{M}\models\Phi}(x)] = \int I_{\mathcal{M}\models\Phi}(x)f(x)dx$ which can be estimated as: $\hat{\mathbf{p}} = \sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i)$, where x_i is the i-th of N trials. The precision of $\hat{\mathbf{p}}$ is quantified by its "relative error" $RE(\hat{\mathbf{p}}) = \frac{\sqrt{Var(\hat{\mathbf{p}})}}{E[\hat{\mathbf{p}}]}$ where $Var(\hat{\mathbf{p}})$ is the variance of the estimator. It is known [2] that: $RE(\hat{\mathbf{p}}) = \sqrt{\frac{1-\mathbf{p}}{\mathbf{p}N}} \approx \frac{1}{\sqrt{\mathbf{p}N}}$ and $N = \frac{1-\mathbf{p}}{\mathbf{p}RE^2(\hat{\mathbf{p}})} \approx \frac{1}{\mathbf{p}RE^2(\hat{\mathbf{p}})}$.

Our SMC tool SMCD consists of one or more collectors and an aggregator. Each collector is deployed on a VM, where it: (i) awaits a signal from the aggregator; and (ii) runs a simulation η, computes the result $\eta \models \Phi$, and transmits it back to the aggregator. The aggregator manages the SMC in rounds to avoid execution time bias [11]. At the beginning of each round, the aggregator sends a message to each collector to begin a simulation. After all collectors have reported their result, the current probability and relative error is calculated. If the calculated relative error is less than the target relative error RE, the algorithm terminates. If not, a new round of simulations is started.

Since our simulations execute the actual system code, each may take significant time. Moreover, we know that many simulations (N) are needed if $\mathbf{p}$ and RE are small. Thus, to analyze systems of realistic complexity, SMCD collectors might be deployed on large clusters of machines with varying availability. Hence, the aggregator is designed so that collectors may join and drop at any time. If

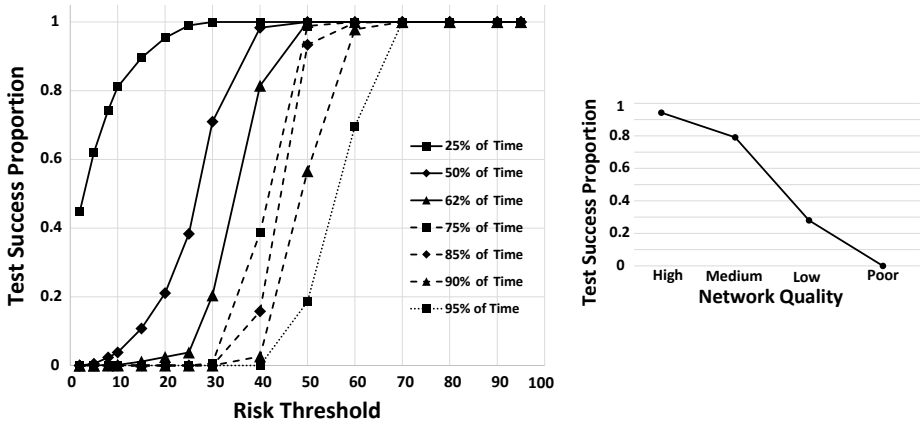

Fig. 2. Graphs of results from experiments 1 (left) and 2 (right).

a collector joins during a round, it is held in reserve until the next round but not used during the current round. If a collector disconnects during a round, an "abort" message is send to the other collectors, and the results of that round are discarded. Thus, we avoid any potential bias in the probability estimation.

4 Experiments

Our scenario (c.f. Figure 1 for DMPL code fragment) involves a reconnaissance mission with five flying robots (i.e., nodes) on a 2-dimensional grid. One node (leader) has mission-critical sensors, while the others (protectors) provide physical defense from attackers (so we want to maximize coverage). The leader must follow a specific flight path, and reach a particular location by time T while maintaining a minimum level of protection for mission success. Nodes execute a (presumed correct) collision-avoidance protocol, which slows down the the fleet, due to additional coordination. Each grid cell has a random hazard level with known probability distribution. The system has two formations: (i) tight – the protectors are closer to the leader, and (ii) loose – protectors are further apart. The tight formation provides better coverage to the leader but is about twice as slow. The leader executes a self-adaption algorithm for formation selection, based on upcoming hazards and remaining mission time, to increase likelihood of mission completion. Further details are not germane to this paper. However, our tools and the complete example is available at `http://www.andrew.cmu.edu/~schaki/misc/smc-dart.tgz`.

Experiment 1: Quality of Formation. First, we analyzed the quality of formation-keeping by the protectors. At any time instant, let us define the leader's risk ($\mathcal{R}$) as the product of the hazard at its location and its exposure (one minus its coverage, computed via the method shown in Figure 1). We selected 14 $\mathcal{R}$ values: $\{2, 5, 8, 10, 15, 20, 25, 30, 40, 50, 60, 70, 80, 90, 95\}$, and 7 `@AtLeast` values: $\{.25, .50, .62, .75, .85, .90, .95\}$. For each $\mathcal{R}$ value ρ and `@AtLeast` value al, we defined an `expect` clause to express mission success only if the leader's risk

remains below ρ for at least al fraction of the time. This yielded 98 properties, whose probabilities were computed using SMCD. We ran 603 simulations, on 5 VMs in parallel, using $T = 115s$. This achieved $RE = 0.1$ for most of the properties. A few properties had $RE > 0.1$, and would require techniques to handle rare events [6,8]. Our results are summarized in Figure 2. Each curve corresponds to a different `@AtLeast` value. As expected, the probability increases with the risk threshold, but falls with increasing `@AtLeast` value.

Experiment 2: Resilience to Network Disruption. Next, we instrumented our simulation to randomly drop messages between the nodes. This slows down the fleet due to increased coordination time for collision avoidance. We defined four network categories based on drop rate ranges: high (0% to 20%), medium (20% to 40%), low (40% to 60%), and poor (60% to 80%). Using an `@AtEnd expect` clause we defined the following property: at the end of the mission, the leader must be at the target location. We then computed the probability of this property for each network category. For each experiment for a category, we randomly selected a drop rate from that category's range, uniformly distributed. To achieve $RE = 0.1$, we needed 35 experiments for "high", 43 experiments for "medium", and 264 for "low". Fig. 2 shows the results. As expected, the probability drops with decreasing network quality. For the "poor" network, we stopped after 43 experiments, since we saw no successes. Techniques to handle rare events in SMC [6,8] are needed for this case as well[1].

References

1. Bulychev, P.E., David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Checking and Distributing Statistical Model Checking. In: Proc. of NFM (2012)
2. Clarke, E.M., Zuliani, P.: Statistical Model Checking for Cyber-Physical Systems. In: Proc. of ATVA (2011)
3. David, A., Du, D., Larsen, K.G., Legay, A., Mikucionis, M.: Optimizing Control Strategy Using Statistical Model Checking. In: Proc. of NFM (2013)
4. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Proc. of CAV (2011)
5. Edmondson, J.R., Gokhale, A.S.: Design of a Scalable Reasoning Engine for Distributed, Real-Time and Embedded Systems. In: Proc. of KSEM (2011)
6. Jégourel, C., Legay, A., Sedwards, S.: Importance Splitting for Statistical Model Checking Rare Properties. In: Proc. of CAV (2013)
7. de Niz, D., Lakshmanan, K., Rajkumar, R.: On the Scheduling of Mixed-Criticality Real-Time Task Sets. In: Proc. of RTSS (2009)
8. Srinivasan, R.: Importance Sampling: Applications in Communications and Detection (2002)
9. V-REP website, `http://www.coppeliarobotics.com`
10. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. Ph.D. thesis, Carnegie Mellon University (2005)
11. Younes, H.L.S.: Ymer: A Statistical Model Checker. In: Proc. of CAV (2005)

[1] 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-0002365