DARTSim: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Smart Cyber-Physical Systems

Gabriel A. Moreno*, Cody Kinneer†, Ashutosh Pandey† and David Garlan†
*Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA
†School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
gmoreno@sei.cmu.edu, ckinneer@cs.cmu.edu, ashutosp@cs.cmu.edu, garlan@cs.cmu.edu

***Abstract*—Motivated by the need for cyber-physical systems (CPS) to perform in dynamic and uncertain environments, smart CPS (sCPS) utilize self-adaptive capabilities to autonomously manage uncertainties at the intersection of the cyber and physical worlds. In this context, self-adaptation approaches face particular challenges, including (i) environment monitoring that is subject to sensing errors; (ii) adaptation actions that take time, sometimes due to physical movement; (iii) dire consequences for not adapting in a timely manner; and (iv) incomparable objectives that cannot be conflated into a single utility metric (e.g., avoiding an accident vs. providing good service). To enable researchers to evaluate and compare self-adaptation approaches aiming to address these unique challenges of sCPS, we introduce the DARTSim exemplar. DARTSim implements a high-level simulation of a team of unmanned air vehicles (UAVs) performing a reconnaissance mission in a hostile and unknown environment. Designed to be easily used by researchers, DARTSim provides a TCP-based interface for easy integration with external adaptation managers, documentation, and a fast simulation capability.**

***Index Terms*—simulation, self-adaptation, cyber-physical system**

I. Introduction

Motivated by the need for cyber-physical systems (CPS) to perform in dynamic and uncertain environments, smart CPS (sCPS) utilize self-adaptive capabilities to autonomously manage uncertainties at the intersection of the cyber and physical worlds [1], [2]. In this context, self-adaptation approaches face particular challenges, which are not often present in information systems. First, monitoring the environment involves monitoring the physical world through sensors that are not perfect. Handling this typically requires dealing with probability distributions of the monitored states to capture the uncertainty of the information collected by the systems [3]. Second, the actions that the system can take to adapt—sometimes called adaptation tactics [4]—can take time to execute. For example, powering up a sensor after it was turned off to save power takes time. Also, there are cases in which adaptation actions require physical movement, which takes time given the dynamics of the system (e.g., the speed at which it moves). The time that an adaptation action takes to execute is known as *tactic latency*. Previous work has shown that explicitly considering tactic latency during the adaptation decision process improves the effectiveness of self-adaptation. The third challenge is that in some sCPS, there can be dire consequences if the system does not adapt in a timely manner. While in an information system a late adaptation may result in the violation of a service-level agreement (SLA), in an sCPS the consequence could involve physical damage to the system or elements of its environment. When this is a concern, it may be necessary to use proactive self-adaptation approaches [5]–[7], and even consider tactic latency explicitly [8]–[10], unless the system is conservative to compensate for the delayed reaction, resulting in less effectiveness. Finally, some classes of sCPS have incomparable objectives that have to be satisfied and cannot be conflated into a single utility metric. For example, in many cases it is not possible to put a value to the physical damage to the system or third parties in a way that can be traded off with the utility that the system provides, as we will show in Section II.

To enable researchers to evaluate and compare self-adaptation approaches aiming to address these unique challenges of sCPS, we introduce the DARTSim exemplar. This system, developed in the context of the DART (Distributed Adaptive Real-Time) Systems project at the Carnegie Mellon® Software Engineering Institute [11], represents a simulated team of unmanned aerial vehicles (UAVs) performing a reconnaissance mission in a hostile environment. In this mission, the team faces a trade-off between detecting targets on the ground—the main purpose of the mission—and avoiding being shot down by threats, which would result in the failure of the mission. Pre-planning the execution of the mission is not feasible because the environment (i.e., the location of targets and threats) can only be discovered as the team flies during the mission, and even then, with some uncertainty. In this cyber-physical system, self-adaptation is required for the team to deal with the uncertain environment, thus making it a smart cyber-physical system.

Designed to be easily used by researchers, DARTSim provides a TCP-based interface for easy integration with external adaptation managers regardless of the language they are written in. Furthermore, it includes a Java class that encapsulates the interaction with DARTSim through TCP, making it even easier to integrate adaptation managers written in Java. When interacting with the simulator using TCP, it runs as a process separate from the adaptation manager. However, DARTSim can also be linked as a library with adaptation managers written in C++. This is not only convenient, but also improves the overall performance by eliminating the use of TCP to interact with the simulator. Regardless of the method used to connect to the simulator, the adaptation manager controls when the simulation has to execute a simulation step, which happens in simulation time—not wall-clock time. Thus, simulations are fast because there is virtually no wait for events to happen. Example adaptation managers using both kinds of connection are included.

Although several aspects of the mission, such as sensing errors and the locations of threats and targets, are random, the random number generators are seeded, thus allowing the replication of the same conditions multiple times. This supports using DARTSim to compare different adaptation managers under the same conditions.

DARTSim has been used for evaluating and comparing self-adaptation approaches [9], [10], and for evaluating an input attribution approach for statistical model checking [12]. Following the proposal of Cito and Gall to use Docker containers for artifacts supporting software engineering research [13], we are now making this exemplar available as a Docker image for easy deployment on different operating systems.[1]

The rest of the paper is organized as follows. Section II gives an overview of DARTSim and the mission being simulated. Section III explains how adaptation managers can interact with DARTSim to do monitoring and execute adaptation actions on the system. Section IV shows how to run simulation and interpret the results.

II. Overview of DARTSim

DARTSim simulates an autonomous team of UAVs carrying out a reconnaissance mission in a hostile and unknown environment. As depicted in Figure 1, the team has to detect targets on the ground using a downward-looking sensor as it flies a planned route at constant speed. However, there are threats on the ground along the route that can destroy the team. The closer the UVAs fly to the ground, the more likely they are to detect the targets, but also the higher the probability of being destroyed by a threat. This poses a trade-off for the team, which is complicated by the uncertainty about the environment. Even though both targets and threats are static, neither their number nor their location is known a priori. The team can use tactics that include changing altitude, changing formation, and using electronic countermeasures (ECM) in order to maximize the number of targets detected, taking into account that if the team is lost to a threat, the mission fails. Mission success is defined as surviving the mission and detecting at least half of the targets.

DARTSim is a high-level simulation of this mission. The whole team of UVAs is considered as a single system with no distinction between its members. For example, what in DARTSim is simulated as a single downward-looking sensor, in reality would be composed of cameras in each UAV whose images are fused. The physics of the flying UAVs is not

[1] Available at https://hub.docker.com/r/gabrielmoreno/dartsim/.

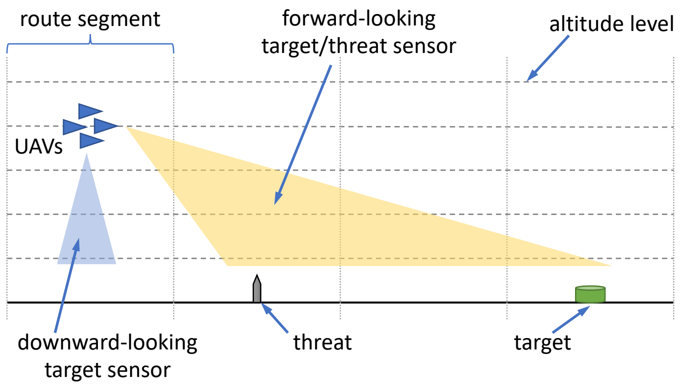
route segment
forward-looking target/threat sensor
altitude level
UAVs
downward-looking target sensor
threat
target

Fig. 1. Simulation overview.

simulated either. The focus of DARTSim is on the high-level self-adaptation decisions that the system must make to achieve mission success. The adaptation manager making these decisions is what gives this sCPS the “smartness” required to execute the mission autonomously.

Time and space are discretized in the simulation. Vertically, the simulator deals with altitude levels, and horizontally, the route is divided into segments[2] of equal length. With the team of UAVs flying at constant speed, time is discretized so that the team traverses one route segment in each simulation time step.[3] The adaptation manager can collect monitoring information, make an adaptation decision, and execute an adaptation if necessary at the boundaries between route segments.

The following subsections describe the monitoring information that the system provides, the adaptation tactics that can be used to change the team configuration, and the effect the team configuration has in its ability to detect targets and avoid being destroyed by threats.

A. Monitoring the System and the Environment

The state of the system is defined by variables divided into two groups: those that the can be controlled through adaptation tactics, and those that cannot be controlled. The former are known as the *team configuration*, and include the altitude level, the formation (i.e., tight or loose), whether ECM is being used, and variables that indicate the *time to complete (TTC)* in periods for each tactic with latency, which are described in Section II-B. When TTC is 0, the tactic is not currently being executed. The variables that cannot be controlled are the position of the team along the route and the direction of travel, since the team follows a predefined route. By default, the route of the UAVs is a straight line. However, the simulator can be configured to use a square map, in which the route covers every cell in the map following a lawnmower pattern. This presents the challenge of the forward-looking sensors not being able to sense the route ahead as the team is making a turn at the end of each horizontal run across the map.

[2] Since DARTSim also supports a square map, the term *cell* is also used.

[3] Since the *decision period* of the adaptation manager is equal to a *simulation step*, we use the two terms interchangeably.

TABLE I
ADAPTATION TACTICS.

Tactic Code	Description	Latency
`IncAlt`	Climb one altitude level	1 period
`DecAlt`	Descend one altitude level	1 period
`IncAlt2`	Climb two altitude levels	1 period
`DecAlt2`	Descend two altitude levels	1 period
`GoTight`	Change to tight formation	immediate
`GoLoose`	Change to loose formation	immediate
`EcmOn`	Turn ECM on	immediate
`EcmOff`	Turn ECM off	immediate

The team has two long-range forward-looking sensors to monitor the state of the environment ahead of the team. One is used to sense the presence of targets and the other is used for threats. For each cell or route segment in front of the sensor, the sensor reports whether it detects a target or threat, depending on the sensor. However, due to sensing errors, these reports can be false positive or false negative. An adaptation manager can get multiple observations to construct a probability distribution of threat or target presence in a cell.

B. Adaptation Tactics

DARTSim provides a number of tactics that an adaptation manager can use to change the team configuration. Table I lists the adaptation tactics including whether they have latency or are immediate. An immediate tactic produces its effect without delay. For example, if the adaptation manager uses the tactic `GoTight` at the beginning of a simulation step, the team will be in tight formation while flying over the route segment covered by that step. If a tactic with latency is used, the effect of the tactic will have a 1 period delay.[4] It is not necessary for an adaptation manager to use the complete tactic repertoire. For example, an adaptation manager may only use tactics to change altitude by one level and change formation. Adding tactics to the set of used tactics can be used to change the size of the adaptation space handled by the adaptation manager.

C. Effect of Team Configuration

The team configuration has an effect on the probability of being destroyed by a threat and the probability of detecting a target, which is important when deciding how to adapt.

A threat can destroy the team only if both are in the same segment. However, a threat has range r_T, and its effectiveness is inversely proportional to the altitude of the team, denoted by a. In addition, the formation of the team affects the probability of it being destroyed. The team can be in two different formations: loose ($\phi = 0$), and tight ($\phi = 1$). The latter reduces the probability of being destroyed [14] by a factor of ψ. When the team uses ECM ($E = 1$), the probability of being destroyed is reduced by a factor of 4. Taking altitude, formation, and the use of ECM into account, the probability of the team being destroyed, d is given by (1).

[4]The latency of the change altitude tactics can be configured.

$$d = \frac{\max(0, r_T - a)}{r_T}\left((1-\phi) + \frac{\phi}{\psi}\right)\left((1-E) + \frac{E}{4}\right) \quad (1)$$

The probability of detecting a target with the downward-looking sensor given that the target is in the segment being traversed by the UAVs is inversely proportional to the altitude of the team [15]. Furthermore, flying in tight formation reduces the detection probability due to sensor occlusion or overlap, and the use of ECM also affects target detection, reducing the probability of detection by a factor of 4. The probability g of detecting a target is given by (2).

$$g = \frac{\max(0, r_S - a)}{r_S}\left((1-\phi) + \frac{\phi}{\sigma}\right)\left((1-E) + \frac{E}{4}\right) \quad (2)$$

where r_S is the range of the sensor (i.e., at an altitude of r_S or higher, it is not possible to detect targets), and σ is the factor by which the detection probability is reduced due to flying in tight formation.

These performance characteristics can be leveraged by an adaptation manager to decide how to adapt. Whether they are used qualitatively or quantitatively depends on the decision-making approach used by the adaptation manager. The simple example described in Section III uses the knowledge of these characteristics qualitatively in a heuristic decision-making approach. More principled approaches attempting to use them quantitatively are faced with the problem of trading off two measures that are incomparable in principle, the survival of the team vs. the maximization of targets detected. For example, mission success requires that the UAVs survive the mission, whereas only half of the targets have to be found. Even though this implies that surviving the mission has higher priority, a decision approach that favors survival may be too conservative to detect enough targets. A more balanced trade-off could be guided by assigning value to surviving the mission and to the targets, so that they become comparable. However, given that the number of targets present in the route is unknown, doing this is not trivial. This is a challenge in many sCPS that this exemplar presents as well.[5] DARTSim includes a more complex example that uses these performance characteristics quantitatively with the PLA-SDP self-adaptation approach [10].

III. Implementing Adaptation Managers

Adaptation managers can interact with DARTSim in two ways. One is using it as a C++ library linked with the program implementing the adaptation manager. The other is running DARTSim as a separate process and interacting with it through a TCP socket. The first method has the advantage of having DARTSim and the adaptation manager in a single process; not incurring the overhead of the TCP socket connection; and interacting with DARTSim with simple method calls. The

[5]For example, survival in other sCPS could mean not running out of power when that depends on how the uncertain weather will affect the output of solar cells.

second method has the advantage of being language independent, providing flexibility to implement adaptation managers in most programming languages. Even when DARTSim is run as a separate process, the adaptation manager controls the simulation steps, allowing DARTSim to execute a simulation step as soon as the adaptation manager finishes making the adaptation decision. This results in fast simulations with both methods. The following sections explain how an adaptation manager can interact with DARTSim using each method.

A. *Interacting with DARTSim as a Library*

Listing 1 shows a fragment of the code of the adaptation manager in one the examples included with DARTSim. This example is implemented in C++ and demonstrates how to use DARTSim as a library. Lines 1–4 show how to create an instance of DARTSim and check if that was successful. The method `createInstance()` receives arguments in the same way as the C++ `main()` function. This allows configuring the simulation from the command line, for example (see Section IV for some of the allowed configuration arguments). Lines 6–8 demonstrate how the adaptation manager can obtain parameters of the configuration of the simulation that can be relevant. In this case, it is using the `getParameters()` method to determine what is the maximum altitude level allowed. There are several other parameters that can be obtained, such as parameters of the sensors (e.g., false positive and false negative rates), and the range of the threats.

The rest of the code of the adaptation manager is organized in a way similar to the MAPE-K loop [16]. The main adaptation loop is in lines 10–43. The method `finished()`, called in line 10, is used to check if the simulation has finished, either because the team reached the end of the route or because it was destroyed. Lines 12–15 implement the monitoring activities of the adaptation manager. Line 12 invokes the method `getState()` to get the state of the system—the team—and lines 14–15 read the forward sensors for targets and threats to monitor the environment. The methods `readForwardThreatSensor()` and `readForwardTargetSensor()` take the length of the look-ahead horizon in cells as argument, and return a vector of booleans with the sensor report for the presence of a threat or a target, respectively, in each cell over the look-ahead horizon, with the first element of the vector corresponding to the cell the UAVs are about to enter.

The analysis and planning aspects of the adaptation loop are combined in event-condition-action (ECA) style [17], and their implementation is in lines 17–36. The tactics that the planning decides to execute are collected in a list of tactics, which is then passed to the simulator for execution. At a high level, the logic of the adaptation decision is as follows. If a threat is detected in the horizon, increase altitude if possible (lines 18–21), otherwise decrease altitude if possible only if a target is detected in the horizon (lines 22–28). If there is a threat in the segment the team is entering, switch to tight formation if not already in that formation (lines 30–33). If there is no threat in that segment and the team is not in loose formation, switch to loose formation.

```
1  Simulator *dartsim = Simulator::createInstance(simArgc, argv);
2  if (!dartsim) {
3      usage();
4  }
5
6  auto simParams = dartsim->getParameters();
7  const int minAltitude = 1;
8  const int maxAltitude = simParams.altitudeLevels;
9
10 while (!dartsim->finished()) {
11     auto startTime = myclock::now();
12     auto state = dartsim->getState();
13     cout << "current position: " << state.position << endl;
14     auto threats = dartsim->readForwardThreatSensor(horizon);
15     auto targets = dartsim->readForwardTargetSensor(horizon);
16
17     Simulator::TacticList tactics;
18     bool threatAhead = any_of(threats.begin(), threats.end(),
19                                                   [](bool p){return p;});
20     if (threatAhead && state.config.altitudeLevel < maxAltitude) {
21         tactics.insert(Simulator::INC_ALTITUDE);
22     } else {
23         bool targetAhead = any_of(targets.begin(), targets.end(),
24                                                   [](bool p){return p;});
25         if (targetAhead && state.config.altitudeLevel > minAltitude) {
26             tactics.insert(Simulator::DEC_ALTITUDE);
27         }
28     }
29
30     if (threats[0]) { // is there an immediate threat?
31         if (state.config.formation != TeamConfiguration::Formation::TIGHT) {
32             tactics.insert(Simulator::GO_TIGHT);
33         }
34     } else if (state.config.formation != TeamConfiguration::Formation::LOOSE) {
35         tactics.insert(Simulator::GO_LOOSE);
36     }
37
38     auto delta = myclock::now() - startTime;
39     double deltaMsec = chrono::duration_cast<
40                     chrono::duration<double, std::milli>>(delta).count();
41
42     dartsim->step(tactics, deltaMsec);
43 }
44
45 auto results = dartsim->getResults();
46 if (!results.destroyed) {
47     cout << "Total targets detected: " << results.targetsDetected << endl;
48 }
49 cout << dartsim->getScreenOutput();
50 delete dartsim;
```

Listing 1. Adaptation manager using DARTSim as a library.

The final task of the adaptation loop is to pass the (possibly empty) tactic list to the simulator so that they are executed. This is done by calling the method `step()` (line 42), which in addition to having the simulator execute the tactics, instructs it to perform one simulation step. All the tactics in the list are started by the simulator at the beginning of the simulation step. The second parameter passed to this method is the adaptation decision time that was measured by the adaptation manager (lines 11, 38–40). Although measuring decision time is not required, when done, DARTSim collects statistics that can be obtained at the end of the simulation. This is useful for comparing different self-adaptation approaches.

Once the simulation finishes, the code in lines 45–49 gets the simulation results from DARTSim and prints the simulation output in the format shown in Figure 2 and described in Section IV.

B. Interacting with DARTSim over TCP

DARTSim's TCP-based interface can be used for adaptation managers that run as an external program. The interface allows DARTSim to be used as a target system for a variety of adaptation managers, regardless of how they are implemented since most languages support communication through TCP sockets. In addition, for passing data structures through the TCP-based interface, DARTSim uses the JSON data interchange standard [18], which is supported by libraries available for many languages.

Although an adaptation manager and DARTSim will execute as separate programs, they interact with each other at each simulation step. At each step, the manager will probe DARTSim for its state and, if it decides that adaptation is needed, it will pass a list of adaptation tactics to the simulator. DARTSim will execute these adaptation tactics, starting them concurrently, and the simulation will advance one step.

DARTSim listens on port 5418 by default. Once a client (i.e., an external adaptation manager) opens a connection to this port, DARTSim waits for a command sent by the client as a text line terminated with a new-line character. If the command is not recognized or there is an error executing it, DARTSim replies with a text line with the prefix `'error:'`, followed by an error message.

To send a command, the client first needs to create a string containing the command name and the corresponding input parameters separated by spaces; this string is then sent to DARTSim using the TCP connection. Upon receiving a well-formed command string, DARTSim executes it, and sends the result back to the client in the form of a string. The client has to convert the string to the corresponding data type.

For composite data types (e.g., list and records) used either as input parameters or return values, DARTSim's TCP-based interface uses the JSON data interchange standard. For example, the command `step` has list of tactic names to execute as the first input parameter, and the adaptation decision time, which is a floating point number, as the parameter. In this case, the client needs to create a string in JSON format to encode the list of strings for the first parameter. For instance, the complete string for the command `step` telling the simulator to start the tactics `DecAlt` and `GoLoose`, and reporting an adaptation decision time of 132.0 milliseconds is

```
step ["DecAlt","GoLoose"] 132.0
```

To invoke this command with an empty list of tactics when no adaptation is needed, the empty list is represented with empty square brackets `'[]'`.

If the result of a command is a composite data type such as a record, then the client needs to decode the string encoded in JSON format upon receiving the result. For example, the result of the `getState` command would be encoded as follows (all in a single line):

```
{"altitudeLevel": 1,
 "directionX": 1, "directionY": 0,
 "ecm": false, "formation": 0,
 "positionX": 17, "positionY": 0,
 "ttcDecAlt": 0, "ttcDecAlt2": 0,
 "ttcIncAlt": 0, "ttcIncAlt2": 0}
```

JSON libraries that handle the conversion back and forth between a JSON string and composite data types are available for many programming languages [18].

DARTSim includes an example called `simple-java`, which includes a class `DartSimClient` that implements in Java the adaptation manager side of the TCP-based communication with DARTSim. This class can be reused by researchers to implement adaptation managers for DARTSim in Java. When using it, the interaction with the simulator is as simple as when using it as described in Section III-A. For researchers using another language, this class can be used as a reference implementation, and the complete TCP interface is documented in the artifact.

IV. Running Simulations

Running simulations with DARTSim is straightforward. This section shows how to run DARTSim with the provided examples using the default configuration, and explains how to interpret the results of a completed simulation. In addition, some of the most common configuration options are explained.

A. Running Examples

DARTSim includes three examples demonstrating integrating DARTSim with adaptation managers. Two minimal examples illustrate using DARTSim with adaptation managers in two ways: as a library in C++, and over its TCP interface using other programming languages (the provided example is in Java). A third adaptation manager provides an implementation of PLA-SDP [10] to demonstrate a fully featured adaptation manager integrated with DARTSim.

To run DARTSim as a process separate from the adaptation manager (i.e., when they interact through the TCP interface), the simulator can be started with the `run.sh` script executed from a shell in the top-level directory of DARTSim. In that case, DARTSim waits for a connection from an adaptation manager, which has to be started separately. To simplify this, both processes can be launched with the script `run-with-am.sh`. When DARTSim is used as a library, only the adaptation manager process must be started, which in turn, instantiates the simulator. Detailed instructions for running the examples are included in the artifact's documentation.

B. Interpreting the Results

The results of the simulation can be obtained as a visual trace and as raw data using the methods `getScreenOutput()` and `getResults()`, respectively (as explained in Section III), and is up to the adaptation manager to print them or save them to a file (see lines 45–49 in Listing 1). Figure 2 shows an example visual trace, which depicts a 2D side view of the team's route. The two-bottom lines represent the ground, and the lines above them represent the altitude level of the drones (vertical axis) at the different positions in the route (horizontal axis). The meaning of the symbols used in this representation is shown in Table II. The

```
#
 # #    ##   ## #   # # # ## # # # ## *#
  * *  #  * #  # # # * # #  # # # #  *
   **    #      *
 ^      ^       ^   ^ ^    ^

       T   X     X          T
```

Fig. 2. Sample simulation output.

TABLE II
LEGEND FOR SYMBOLS USED IN SIMULATION OUTPUT.

Symbol	Meaning
#	loose formation
*	tight formation
@	loose formation, ECM off
0	tight formation, ECM on
ˆ	threat
T	target (not detected)
X	target (detected)

raw data reported includes: the number of targets detected by the team, whether the team was destroyed or not, the location where the team was destroyed (if destroyed), whether or not the mission was successful, the average and variance of the decision time. Of these values, the most important are the first two, the number of targets and whether the team survived or not. These are the objective values that an adaptation manager should optimize, detecting as many targets as possible without being destroyed. The last two values enable an easy comparison of decision time between adaptation managers.

C. Configuring the Simulation

DARTSim furnishes a number of configurable options for the simulation. This section discusses some of the more important configuration options; however, a complete listing of options is available in the artifact's documentation. When DARTSim is run as separate process, configuration options should be passed to the simulator as command arguments. When used as a library, the options are passed to DARTSim in C++ *(argc, argv)* style as explained in Section III-A.

DARTSim uses randomness in several parts of the simulation, including generating the environment (the locations of targets and threats), and simulating the results of sensor readings and encounters with threats. When comparing adaptation managers, it may be desirable for these random processes to be consistent between runs (i.e., so that for each trial, the conditions are replicated for each adaptation manager). This is facilitated through the `--seed` configuration option, which allows the user to specify the master seed for the various random number generators (RNG) used in the simulator.[6] Keeping this number constant between executions of the simulator will result in reproducible conditions.

DARTSim also supports varying the size of the environment and the number of threats and targets present. The `--map-size` options allows the user to configure the number of segments in the route, which determines the length of the team's mission. The `--altitude-levels` option allows the user to determine how many discrete altitude levels are available for the team. This option provides an easy way to increase the number of possible states that the team can be in, which could be useful in evaluating the scalability of an adaptation manager. Some other useful options include `--num-threats` and `--num-targets`, which allows the user to specify how many threats and targets are in the environment. An exhaustive list of the DARTSim's configuration options is included in the documentation of DARTSim.

[6]This seed is for a master RNG that generates seeds for the other RNGs.

V. CONCLUSION

In this paper we have presented DARTSim, an exemplar that simulates an autonomous team of UAVs carrying out a reconnaissance mission in an unknown hostile environment. DARTSim poses challenges that self-adaptation approaches face when used to provide "smartness" to sCPS. These include the sensing error of the long-range forward-looking sensors; the adaptation tactics to change altitude levels, which take time to execute due to the required physical movement; the unknown threats that can destroy the UAVs if they do not evade them in a timely manner; and the difficulty in conflating the value of surviving the mission and the value of the detected targets in a single utility measure when the number of targets is unknown.

Several aspects of the mission are randomly generated in DARTSim, making it suitable for running many simulations with different conditions for statistical analysis when evaluating self-adaptation approaches. Nevertheless, DARTSim can replicate these random conditions, allowing researchers to compare different approaches. The exemplar is available as a Docker image (https://hub.docker.com/r/gabrielmoreno/dartsim/) for easy deployment on different platforms, and its source code is available at https://github.com/cps-sei/dartsim.

ACKNOWLEDGMENTS

Copyright 2019 IEEE. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center, and by AFRL and DARPA under agreement number FA8750-16-2-0042, the Office of Naval Research (N000141612961), and the National Science Foundation (CCF-1618220). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA, ONR, NSF, or the U.S. Government. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. Carnegie Mellon$^{®}$ is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM19-0066

References

[1] T. Bures, F. Krikava, R. Mordinyi, N. Pronios, D. Weyns, C. Berger, S. Biffl, M. Daun, T. Gabor, D. Garlan, I. Gerostathopoulos, and C. Julien, "Software engineering for smart cyber-physical systems – towards a research agenda," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 6, pp. 28–32, Nov. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2830719.2830736

[2] T. Bures, A. Knauss, P. Patel, A. Rashid, I. Ruchkin, R. Sukkerd, C. Tsigkanos, D. Weyns, B. Schmer, E. Tovar, E. Boden, T. Gabor, I. Gerostathopoulos, P. Gupta, and E. Kang, "Software engineering for smart cyber-physical systems," *ACM SIGSOFT Software Engineering Notes*, vol. 42, no. 2, pp. 19–24, Jun. 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3089649.3089656

[3] L. Bertuccelli and J. How, "Robust UAV search for environments with imprecise probability maps," in *Proceedings of the 44th IEEE Conference on Decision and Control*. IEEE, 2005, pp. 5680–5685. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1583068

[4] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, Dec. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2381464.2381594

[5] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster Computing*, vol. 12, no. 1, pp. 1–15, Oct. 2008. [Online]. Available: http://link.springer.com/10.1007/s10586-008-0070-y

[6] B. Trushkowsky, P. Bodík, A. Fox, and M. J. Franklin, "The SCADS director: Scaling a distributed storage system under stringent performance requirements," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. San Jose, California: USENIX Association, 2011, pp. 163—-176. [Online]. Available: http://static.usenix.org/legacy/events/fast11/tech/full_papers/Trushkowsky.pdf

[7] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos, "Model predictive control for software systems with CobRA," in *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '16*. Austin, Texas: ACM Press, 2016, pp. 35–46. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2897053.2897054

[8] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: a probabilistic model checking approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. New York, New York, USA: ACM Press, Aug. 2015, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2786805.2786853

[9] G. A. Moreno, O. Strichman, S. Chaki, and R. Vaisman, "Decision-making with cross-entropy for self-adaptation," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, May 2017, pp. 90–101. [Online]. Available: http://ieeexplore.ieee.org/document/7968136/

[10] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Flexible and efficient decision-making for proactive latency-aware self-adaptation," *ACM Trans. Auton. Adapt. Syst.*, vol. 13, no. 1, pp. 3:1—-3:36, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3149180

[11] S. A. Hissam, S. Chaki, and G. A. Moreno, "High assurance for distributed cyber-physical systems," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*. New York, New York, USA: ACM Press, Sep. 2015, pp. 1–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2797433.2797439

[12] J. P. Hansen, S. Chaki, S. Hissam, J. Edmondson, G. A. Moreno, and D. Kyle, "Input attribution for statistical model checking using logistic regression," in *Runtime Verification*, Y. Falcone and C. Sánchez, Eds. Cham: Springer International Publishing, 2016, pp. 185–200.

[13] J. Cito and H. C. Gall, "Using Docker containers to improve reproducibility in software engineering research," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 906–907.

[14] M. J. Veth, "Advanced formation flight control," Air Force Institute of Technology, Tech. Rep., 1994.

[15] A. Symington, S. Waharte, S. Julier, and N. Trigoni, "Probabilistic target detection by camera-equipped UAVs," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, May 2010, pp. 4076–4081. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5509355

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055

[17] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing–degrees, models, and applications," *ACM Computing Surveys*, vol. 40, no. 3, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1380584.1380585

[18] "JSON: JavaScript Object Notation," http://json.org, n.d.