Real-Time Simulation Support for Runtime Verification of Cyber-Physical Systems

XI ZHENG, Deakin University
CHRISTINE JULIEN, The University of Texas at Austin
HONGXU CHEN, Tsinghua University
RODION PODOROZHNY, Texas State University
FRANCK CASSEZ, Macquarie University

In Cyber-Physical Systems (CPS), cyber and physical components must work seamlessly in tandem. Runtime verification of CPS is essential yet very difficult, due to deployment environments that are expensive, dangerous, or simply impossible to use for verification tasks. A key enabling factor of runtime verification of CPS is the ability to integrate real-time simulations of portions of the CPS into live running systems. We propose a verification approach that allows CPS application developers to opportunistically leverage real-time simulation to support runtime verification. Our approach, termed BraceBind, allows selecting, at runtime, between actual physical processes or simulations of them to support a running CPS application. To build BraceBind, we create a real-time simulation architecture to generate and manage multiple real-time simulation environments based on existing simulation models in a manner that ensures sufficient accuracy for verifying a CPS application. Specifically, BraceBind aims to both improve simulation speed and minimize latency, thereby making it feasible to integrate simulations of physical processes into the running CPS application. BraceBind then integrates this real-time simulation architecture with an existing runtime verification approach that has low computational overhead and high accuracy. This integration uses an aspect-oriented adapter architecture that connects the variables in the cyber portion of the CPS application with either sensors and actuators in the physical world or the automatically generated real-time simulation. Our experimental results show that, with a negligible performance penalty, our approach is both efficient and effective in detecting program errors that are otherwise only detectable in a physical deployment.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems;** • **Software and its engineering → Formal software verification;** *Software testing and debugging;*

Additional Key Words and Phrases: Runtime verification

106

This work is supported by the National Science Foundation, under grant CNS-1239498.
Hongxu Chen is supported by the China Postdoctoral Science Foundation (Grant No. 2016M591162) and the National Natural Science Foundation of China (Grant No. 51605242).
Authors' addresses: X. Zheng, School of Information Technology, Faculty of Science Engineering & Built Environment, Deakin University, 221 Burwood Highway, Burwood, VIC, 3125, Australia; email: xi.zheng@deakin.edu.au; C. Julien, Department of Electrical and Computer Engineering, One University Station, C500, The University of Texas at Austin, Austin, TX 78712, USA; email: c.julien@mail.utexas.edu; H. Chen, Department of Automotive Engineering, Tsinghua University, A539-1 Lee Shau Kee Science and Technology Building, 1 Tsinghua Park, Haidian, Beijing 100084, China; email: herschel.chen@gmail.com; R. Podorozhny, College of Science and Engineering, CMAL 307E, San Marcos Campus, Texas State University, USA; email: rp31@txstate.edu; F. Cassez, Department of Computing, Faculty of Science and Engineering, Macquarie University, NSW 2109, Sydney, Australia; email: franck.cassez@mq.edu.au.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 1539-9087/2017/05-ART106 $15.00
DOI: http://dx.doi.org/10.1145/3063382

ACM Reference Format:
Xi Zheng, Christine Julien, Hongxu Chen, Rodion Podorozhny, and Franck Cassez. 2017. Real-time simulation support for runtime verification of cyber-physical systems. ACM Trans. Embed. Comput. Syst. 16, 4, Article 106 (May 2017), 24 pages.
DOI: http://dx.doi.org/10.1145/3063382

1. INTRODUCTION

Cyber-Physical Systems (CPS) entail complex software and exhibit sophisticated interactions among digital devices, analog components, and the surrounding world, including humans in that world. CPS are often safety critical and must be reliable. However, CPS contain both digital and analog components and must be modeled as hybrid systems, which are known to be hard to formally verify [14]. Simulation-based approaches to verification, on the other hand, are restrictive both in expressiveness (e.g., of quantitative properties) and coverage (i.e., the cyber part is *modeled* instead of testing the real implementation). Thus, common but subtle bugs that result from the interaction of cyber and physical components are often not detectable.

Runtime verification of CPS provides a middle-ground between formal verification and simulation. Our recent work on runtime verification [36] can check both qualitative (e.g., safety, liveness) and quantitative (e.g., bounded safety and liveness, responsiveness) properties. However, to detect the insidious CPS bugs that are manifest only in a specific deployment environment, this runtime verification requires repetitive deployments that are either too expensive (e.g., in labor, time, and/or money), dangerous (e.g., involving autonomous vehicles), or infeasible. As an example, an unmanned rover deployed to the moon was unable to move after the first lunar night. A *post hoc* analysis found that the temperature on the moon is considerably lower than the rover's components had accounted for; as a result, the rover effectively suffered from frostbite.[1] Runtime verification of CPS in general requires a repetitive and flexible test environment, where settings can be changed easily to determine whether the properties being checked will hold in all situations. In the case of the rover, there are relatively accurate models of involved physical processes (e.g., rover dynamics and moon environment). However, these models are separate from the runtime verification of the system's cyber components, ultimately leading to the failure. Our goal in this work is to create a middleware that enables the seamless integration of these expressive models with the cyber portion of a CPS application.

The state-of-the-practice in creating a repetitive and flexible test environment is to use real-time simulation, where computer models are used to accurately produce values of internal variables; these models are designed to operate on the same time-scale as the corresponding physical system [5]. However, to date, the application of real-time simulation in CPS verification is limited. In hardware-in-the-loop tests [7, 35], a physical controller is connected to an executing real-time simulation representing a virtual plant, and this is used to verify the controller. In software-in-the-loop tests [23], both controller and plant are simulated. Besides being expensive (e.g., requiring proprietary hardware) and bound to a specific simulation platform (e.g., Simulink), all these approaches use testing to check the controller algorithm that resides in the cyber components of a CPS. In comparison, an intuitive and generic way of leveraging real-time simulation with a more formal runtime verification would give more thorough bug detection not limited by the coverage of a particular suite of tests.

However, there are nontrivial research challenges in integrating real-time simulation with runtime verification. First, many CPS systems have heterogeneous

[1] Chen, Stephen. "Last-ditch efforts to salvage mission of China's stricken Jane Rabbit Lunar rover." *South China Morning Post* 18 April 2014 (http://tinyurl.com/oq5qnqx).

components (e.g., automobiles have mechanical, thermodynamic, and electrical sub-systems); data exchange and time synchronization among the simulation models for these heterogeneous components are very challenging in the real-time simulation environment. For instance, simulations of different physical components have different step sizes depending on the attributes of the underlying physical process (e.g., a car engine requires a millisecond level step size, while an actuator for an automatic window normally requires a step size on the order of seconds); different solvers may be required depending on the complexity of linear/differential equations underneath; and different simulation environments (e.g., LabVIEW or Simulink) may be required for a specific subsystem due to particularities of the verification task (e.g., the underlying language used in the application, built-in solvers and libraries, knowledge of a multi-disciplinary team). Another challenge in CPS simulation stems from the fact that the step size needed for each model may not always be easy to estimate *a priori*, especially for the physical portions of the system. When integrating real-time simulation with a verification environment, another issue is the impact of this integration on the monitored CPS application. For instance, several monitoring and verification tasks are quite computationally intensive. Effectively managing the computational overhead of the intended verification environment is a real and as yet unresolved challenge [36]. Finally, the verification environment integrated with real-time simulation inevitably has latency and clock drift. Minimizing the impact of these on the accuracy and scalability of our verification approach is another research challenge.

In this article, we motivate an integrated runtime verification, *BraceBind*, that allows a developer to specify "hooks" that can be connected, at verification time, to either simulated models of physical processes *or* to the physical processes themselves; in a given "run" some hooks can be connected to models while others are connected to physical components. We can change parameters of the models to mimic different kinds of deployment environments, making it possible to detect bugs that appear only in a specific environment. The approach can therefore be used to identify incorrect assumptions about environments that are expensive or infeasible to replicate in hardware-in-the-loop or software-in-the-loop tests alone. Concretely, we make the following contributions:

- We propose a real-time simulation architecture (which we abbreviate as *RTS*) to generate, manage, and optimize an integrated real-time simulation with input from heterogeneous models.
- We create a software architecture based on an aspect oriented adapter (abbreviated as *AOA*) to integrate existing representative real-time verification tools with the automatically generated *RTS* to effectively manage the computational overhead and latency of the integrated verification environment.
- We evaluate our solutions using a real world multi-agent vehicle system[2] to demonstrate that the real-time simulation is highly compliant with the original simulation models and that the integrated verification environment is both efficient and effective in detecting real bugs.

2. RELATED WORK & RESEARCH GAPS

We combine the strengths of real-time simulation (where interactions between cyber and physical components can be easily analyzed and tested for various settings) and runtime verification (where properties are formally specified and checked at runtime).

[2]This CPS application is used as the motivating application throughout the article. In the application, a number of autonomous rovers are assigned dynamically to visit a set of provided tasks, where each task is an obligation to visit a waypoint.

Either alone is insufficient for CPS verification. With only real-time simulation, many aspects are either not representative of an actual deployment or are intractable; runtime verification alone requires access to a complete deployment environment, which may be difficult or infeasible. Although high-quality simulations of physical processes exist and are widely used, significant gaps remain in making these models usable for verification alongside cyber components. We overview the related work, identify the gaps, and preview how we address them.

Challenges in Establishing Accurate Real-time Simulation. Real-time simulation is an "online" version of discrete-time simulation, where time moves forward in steps of pre-defined duration [33]. To solve the underlying mathematical equations (e.g., differential equations) at a specific time step, the model is solved using the input of the variables or states from the preceding time step. As compared to an "offline" version, the execution time T_E required to solve the equations for a time step must be shorter than the specified step size T_S, otherwise the real-time simulation is considered erroneous [5]. Since CPS often entail multiple physical processes, and each physical process may be modeled separately, different real-time simulations must be able to coordinate, even potentially exchanging state information during a single time step T_S. Different simulation models and platforms may have different time steps, depending on their physical laws (e.g., a dynamic electrical system has a fast time step while a dynamic thermal system may have a much slower one). Each real-time simulator (i.e., the executable implementing the simulation for a given model) has to execute a number of tasks within T_E, including reading inputs, solving model equations, generating outputs, and exchanging results with other simulation models. All these tasks are important, and failures or inaccuracies in any of them can render the real-time simulation useless [5]. Accurate synchronization among different simulation models is crucial to ensuring simulation stability [4].

The Functional Mockup Interface (FMI) [6] is an independent standard to create a co-simulation environment where C code for a specific dynamic system model is generated in the form of an input/output block, and two or more models (with different solvers) can be coupled. FMI requires each simulation platform provider (where each dynamic model is created) to explicitly support an FMI interface for model exchange so it is possible to automatically generate a Functional Markup Unit (FMU) from the dynamic model. A FMU is a combination of C code and a helper XML specification that has definitions for all the variables in the given dynamic model. However, the two fundamental challenges in establishing real-time simulation, namely time synchronization and data integration among simulation models, are left for developers to implement in the form of Master Algorithm. The MODELISAR [29] project supports FMI and includes a prototypical implementation of a Master Algorithm. However, the existing implementation does not guarantee the efficiency and simulation speed, which largely depend on the problem to be solved (e.g., the size of the underlying ordinary differential equation or differential algebraic equation) and the host computer's power [2]. This kind of implementation of the Master Algorithm is not acceptable for an integrated runtime verification environment, where efficiency and speed of the real-time simulation must be optimized to guarantee necessary precision of the outputs; further, writing a suitable master algorithm is very error-prone and poses significant challenges for developers [2]. Since numerical integrations deal with approximations, it is of vital importance to have an alternative automated solution that can guarantee efficiency and speed of the real-time simulation (instead of an interface or a requirement for data integration and time synchronization) to maintain a satisfactory balance between the simulation speed (i.e., latency) and precision (i.e., simulation errors) [20]. In Ref. [1], a co-simulation platform is proposed to integrate the *ns-2* network simulator with the *Modelica* physical systems simulator. The simulation platform is able to support

asynchronous events inside both physical and network systems. The main contribution of the work is to solve real-time synchronization to make sure both simulators will advance at the same wall-clock rate. Our work is similar in terms of time synchronization; however, we are more focused on building an integrated runtime verification suite for a real implementation of the cyber part instead of studying and analyzing only simulation.

In industry, real CPS practitioners guarantee the simulation speed and precision by using dedicated machines and software to build the real-time simulation environments (i.e., NI PXI server [30] and LabVIEW real-time module [24]). However, this approach is very expensive (e.g., a basic NI PXI server costs around 10,000 USD [31]) and is not scalable. Also, this approach does not provide a solution for complex CPS where sub-system models must be created in different simulation platforms for a variety of reasons (e.g., knowledge and preference of the interdisciplinary team, different costs, and different built-in solvers). In Ref. [20], a fine-grained co-simulation method is explored that enables numerical integration speed-ups. The method is to partition the existing models into loosely coupled sub-systems with sparse communication between partitioned modules. The parallel execution is mainly to exploit multi-core processors to deal with originally sequential ordinary differential equations in CPS sub-system models. Our work is similar to this approach but we extend the parallel simulation into a coordinated distributed simulation, which is more suitable for complex CPS with different sub-systems, where models of each sub-systems can be executed in separate physical computation units. In Ref. [22], a time-predictable computer architecture for digital emulation is proposed for CPS. The architecture can be implemented on top of a Field Programmable Gate Array (FPGA) to provide low latency emulation. This solution is a hardware based approach; in contrast, we take a software approach that we argue is more automated and accessible. In Ref. [34], an integrated platform is proposed to integrate Matlab/Simulink simulation tool with the DETERLab emulation testbed. The runtime environment provides time synchronization and data communication to coordinate two simulation platforms for security experiments. The work is very similar to ours; however, our contribution is more targeted to generating a distributed real-time simulation environment for fairly large size simulation models as compared with creating a runtime environment to integrate simulation with emulation. Also, the application of this approach is suitable for simulation-based testing by observing signals (e.g., recording the signals received for a unmanned aerial vehicle controller), while our approach is specifically designed for integrated formal verification that integrates the real-time simulation with runtime verification tools.

The above motivates us to create a Real-Time Simulation (*RTS*) architecture to generate, manage, and optimize a distributed simulation environment in which models of each subsystem are converted automatically into executables and deployed into a designated machine. Our previous work in Ref. [7] provides a highly efficient and inexpensive technique to build a real-time simulation environment for powertrain simulation of an electric vehicle. The existing work, however, is restricted to converting a relatively simple simulation model (e.g., a Vehicle Control Unit) into a C++ executable. Building on this work, the *RTS* described in this article is able to convert multiple simulation models across different underlying subsystems of a CPS into a distributed real-time simulation environment with a group of C++ executables running on variety of platforms. *RTS* also provides two modes of data integration services depending on the specific requirement of each model and both in-model and cross-model synchronization for the real-time simulations; this synchronization maximizes simulation speed and minimizes latency.

Limited Use of Real-time Simulation. For real-time simulation in CPS verification, hardware-in-the-loop testing [7, 11, 13, 35] is becoming increasingly common.

In Ref. [13], a real-time simulation environment with microsecond latency is established to test AC motors. However, the environment is specially designed for AC motors, and each simulation module is hard-wired to a specific processor. Our solution ties the (automatically generated) real-time simulation to a compatible runtime verification tool using a message based middleware that requires developer's annotations in the implementation. Our solution is not constrained to any specific hardware or tied to any specific processors, nor is it limited by the coverage of a particular suite of tests. Work in co-design [12, 27] validates device function for a wide range of operating conditions. A real device communicates to mathematical models that are implemented as FPGA circuits or processor instructions; the models are executed in real-time to simulate the interacting environment. In comparison, our solution can build the required real-time simulations based on existing physical models (which are often created anyway during a prototyping phase) from off-the-shelf simulation platforms as long as they provide a mechanism to convert models into executables (e.g., C and C++). The annotations supported by our approach enable switching the test environment from a physical setting to a real-time simulation and enable test settings that consist of combinations of physical devices and simulation models.

3. AN ASPECT ORIENTED ADAPTER TO INTEGRATE WITH RUNTIME VERIFICATION

One major intellectual contribution of our work is to create a middleware based on Aspect Oriented Programming [21]. Specifically, we develop an Aspect Oriented Adaptor (*AOA*) to connect cyber components to values that reflect the physical environment, whether through real-time simulation or physical transducers (e.g., sensors and actuators).

This *AOA* allows CPS developers to use existing runtime verification tools to repeatedly verify important properties of the system in a well-controlled environment. The use of the *AOA* greatly increases our ability to reproduce those "hidden" bugs that are very hard to reproduce and detect in a real deployment environment but are mission critical. We discuss the details of BraceBind's *AOA* architecture, including how we manage to reduce computational overheads while preserving accuracy in detecting errors.

3.1. Foundations

Before we detail the design of our AOA and its integration with runtime verification of CPS, we first review some foundations of our runtime verification architecture more generally. To support runtime verification of CPS, we integrate an existing runtime verification tool [36] with real-time simulation. We assume that the runtime verification tool models the execution of a CPS application as an infinite sequence of observations $\delta = \delta_0\delta_1 \cdots \delta_n \cdots$. Each $\delta_i \subseteq 2^E$, where E is a set of propositions that describes the observed state of the application. Since a CPS application is also a real-time system, timing information must be captured. A *timed trace* is a pair $\Theta = (\bar{\delta}, \bar{\tau})$, where $\bar{\delta}$ is a trace and $\bar{\tau}$ is an infinite sequence of non-negative real numbers representing the time of each event. The timing sequence respects *monotonicity* and *progress* ($\tau_i < \tau_{i+1}$ and $\exists i \in N, \forall j \in R, \tau_i > j$).

In our work, we name those variables that are associated with sensor data and actuator commands from the observed CPS application as *Physical Variables*. At runtime, the timestamped values of designated *Physical Variables* are recorded into a program trace file. This trace file can be merged with a trace file for existing runtime monitoring tools (e.g., a trace capturing the logical elements of the program state), assuming the runtime verification tool has the same (timed) execution model as above. The use of a dedicated trace file is just an implementation choice; the needed values could

also be made accessible through an established programming interface or stored in a concurrent lock free queue, for example.

We also assume for each physical deployment environment, a hardware abstraction layer like Ref. [37] is available; we assume that this layer allows a CPS developer to pass a value to a specific actuator by knowing the actuator's unique name and to retrieve a value from a sensor by knowing the sensor's unique name.

3.2. AOA and AOP

BraceBind's *AOA* is designed to intercept value changes in *Physical Variables* and to inject them into the running CPS application so the application can be run against either a real-time simulation or a physical deployment environment or a combination of the two. To reduce the computation overhead and its potential impact on the observed CPS application, BraceBind's *AOA* uses Aspect Oriented Programming (AOP) [21].[3] AOP enables behavior to be added to an application without changing the original code; using the approach, the same program can be compiled either with or without the added behavior.

When the CPS developer wants to execute the program without the supporting structure of BraceBind's *AOA*, he can do so without making any additional alterations (outside of recompiling the program without the *AOA* interceptors). AOP achieves this flexible behavior by monitoring a specific action of the application (e.g., a method call or an object initialization) and executing a piece of code associated with the action. The target action is called a *pointcut*, which generically identifies a set of points in the control flow of the application. The piece of code associated with the pointcut is defined by the AOP programmer as *advice*. Both the *pointcut* and *advice* are defined outside of the application's source code. However, it is not straightforward to apply AOP to a CPS application in this way as there are no existing specifications to differentiate *Physical Variables* from other program variables, and it is quite error prone to ask CPS developers to manually create the underlying *pointcuts* and pieces of *advice* that are essential in AOP.

Therefore, our first effort in creating the *AOA* is to create customized annotations that are able to provide a view that unifies a CPS application's physical variables and cyber variables, thereby explicitly capturing something the CPS developer knows but is difficult to automatically infer; these annotations also specify how the simulation or transducers can connect to that view.

To use BraceBind's *AOA* architecture, a CPS developer is only required to augment the CPS application with the necessary subset of these customized annotations. Then the CPS developer can execute the CPS application within BraceBind's *AOA* framework without requiring the developer to have detailed knowledge of *AOP*. Using the CPS developer's provided annotations, BraceBind's *AOA* automatically generates a *pointcut* for each identified physical Variable. To do this, *AOA* generates a "pseudo method call" for the assignment to each such variable; this method is called every time the value of the indicated variable is changed (i.e., actuator input) or is accessed (i.e., sensor output). These pseudo method calls are invisible to the CPS developer.

Our *AOA* also uses the CPS developer's annotations to generate the corresponding *advice*. In *AOA*'s use of AOP, the *advice* is invoked every time the pseudo method is called; the advice directs the interaction of the physical variable to either the real-time simulation or physical environment, depending on the annotation. Figure 1 shows this general process.

A developer uses a customized annotation class called `PhysicalVariable` to explicitly identify variables in the cyber part of the implementation that reference the physical

[3]In its implementation, our prototype specifically uses AspectJ: http://aspectj.2085585.n4.nabble.com.

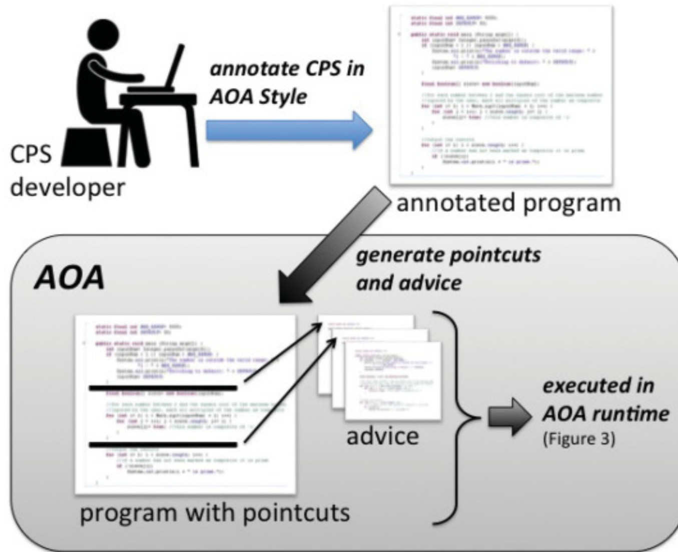


Fig. 1. Underlying Annotation and AOP.

world. This reference may be because a variable's value comes from one or more sensors or because the value is used (often as a result of a control algorithm) to command one or more actuators. These annotations indicate points in the CPS program where, when directed to do so, *AOA* can replace the interaction with the physical world by an interaction with an appropriate real-time simulation.

In the annotation, the CPS developer states the physical variable's `type`, which determines whether the value is used as an *input* to the physical world or the real-time simulation (i.e., it acts on actuators by sending information to the physical world) or as an *output* from the physical world or the real-time simulation (i.e., it receives information from the physical world that is used in the application). The developer's annotated program is passed to BraceBind's *AOA*, which automates the remainder of the process by automatically generating the necessary *pointcuts* and *advice*. Ultimately, these annotations make it possible to pass information about changes in *PhysicalVariable*s to and from the CPS application.

We give an example of an *AOA* annotation in Figure 2; this example is drawn from our rover application introduced previously and used in Section 5. In this example, when the rover senses a change in any of several monitored properties, it executes the `onSensorChanged` method. Within this method, the program determines a new value for the `angle` variable, which determines future movements of the rover. In this example, the CPS developer has declared a physical variable called `Angle_Change` and linked that physical variable to the local program variable `angle`. This variable is an *actuator*, meaning that its value, when changed, should propagate to the executing CPS system; in fact, in this case, the change should propagate to a *simulation* of the actuator.

For the example in Figure 2, without the *AOA*, when the program changes the value of `angle`, it is effectively directly setting a new value for `angle`. When the developer's annotations are compiled with BraceBind's *AOA*, BraceBind automatically generates a pseudo method for the `angle` variable[4] and inserts a *pointcut* that calls this pseudo

[4]If the `angle` variable is referenced in multiple annotations (e.g., in different methods, since `angle` appears to have scope wider than the method), then does BraceBind generate multiple pseudo methods? Does it matter?

```
@PhysicalVariable(name="Angle_Change",
                  variable="angle",
                  type="Actuator",
                  mode="Simulation")
public void onSensorChanged(SensorEvent event) {

  //... somewhere in the function
  float orientation[] = new float[3];
  SensorManager.getOrientation(R, orientation);
  mBearing = (float) (orientation[0] * (180 / Math.PI));

  //... somehwere else in the function
  velocity = (0.15f);
  angle = ((mBearing - desiredBearing) * turn_factor);
}
```

Fig. 2. Physical variable annotation example.

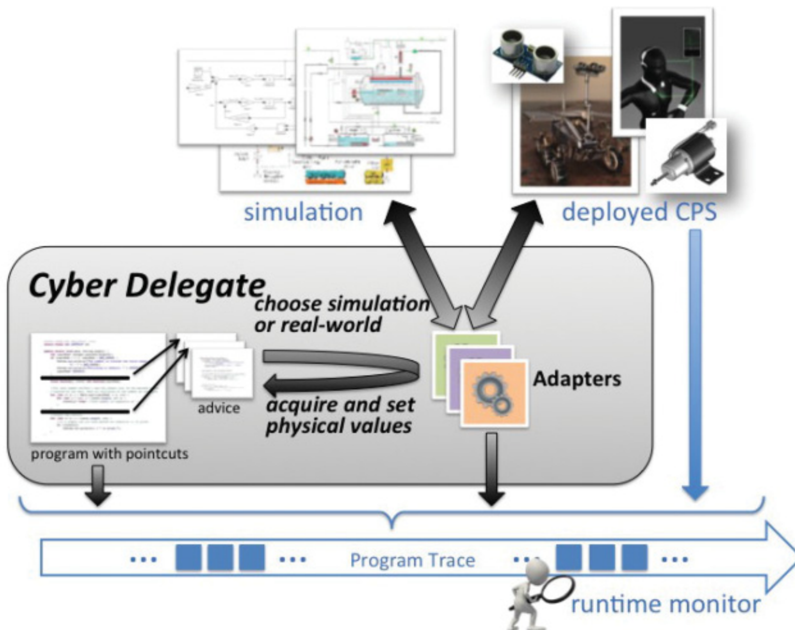


Fig. 3. AOA Adapter Model.

method after each assignment to the variable (e.g., the last line in Figure 2). Given the particular annotation here, the pseudo method connects the CPS program code with the *simulation* of the `Angle_Change` physical variable, using BraceBind's *Cyber Delegate* model.

The *Cyber Delegate* in Figure 3 is a core component of the *AOA*. The *Cyber Delegate* is responsible for intercepting and handling the value changes for *Physical Variables*. Based on directions from the CPS developer's annotations, the *Cyber Delegate* maintains the mapping from each *Physical Variable* to the corresponding sensor output or

actuator input in either a real-time simulation environment or a real-world deployment environment. Specifically, the *Cyber Delegate* synthesizes an *Adapter* for each *Physical Variable*.

An *Adapter* functions as a software interface that connects the value of a particular physical variable with either the physical world or a simulation of that world. Each time a piece of *AOA*'s *advice* is activated, if that advice is associated with data acquisition, the relevant *Adapter* pulls the latest sensor data from the real-time simulation, replaces an explicit access to a sensor with access to the simulation, or, in the case that the *AOA* intends for the variable to access the physical world, simply passes the access along to the sensor.

In contrast, if the activated *advice* is associated with sending a command or variable assignment to an actuator, the *Adapter* pushes the command or variable assignment to the real-time simulation or passes the original action through to the actuator. Every action from the *Adapter* generates a signal that is recorded in a *AOA* program trace file that can be merged with a global program trace file and used as input to a runtime verification tool (e.g., Ref. [36]).

Algorithm 1 shows the main logic of the synthesized *Adapter*. If the variable associated with the *Adapter* is targeted to real-time simulation, then the *Adapter* synthesizes a publisher and pushes the data into a *Real-Time Simulator* (*RTS*) to be delivered to any subscribed models internal to the *RTS* (if the variable is an actuator command), or generates a subscriber and retrieves the sensor output from *RTS* (if the variable is a sensor output). Either type of action is recorded, along with the corresponding timestamp into the *AOA* program trace to be merged with existing real-time simulation trace files (lines 2–11). If the variable associated with the *Adapater* targets a specific element of a real-world deployment environment, then the *Adapater* passes the action directly to the specified actuator or retrieves the requested data from the specified sensor. As in simulation mode, the behavior is recorded into *AOA* trace file (lines 12–21).

ALGORITHM 1: Adapter Algorithm

```
1  if mode is Simulation then
2      if type is Actuator then
3          registerIntoTrace (name,value) registerPublisherInRTS (name, value)
4      end
5      if type is Sensor then
6          registerSubscriberInRTS (name)  value = retrieveDataIntoQueue (name)
               registerIntoTrace (name,value)  infoDataChangeToAOA (name)
7      end
8  end
9  if mode is Real Deployment then
10     if type is Actuator then
11         registerIntoTrace (name,value)  passToActuator (name, value)
12     end
13     if type is Sensor then
14         value = retrieveDatafromSensor (name)  registerIntoTrace (name,value)
               infoDataChangeToAOA (name)
15     end
16 end
```

For the example in Figure 2, the pseudo method in the *AOA*-generated advice connects to the *Cyber Delegate* that intercepts the variable assignment and determines whether the actuation should occur in the real world (in which case the synthesized *Adapter* does nothing but perform the original assignment) or in simulation (in which

case the *Adapter* passes the change into the real-time simulation of the rover's movement model).

The semantics of the developer's annotation that connect the `angle` variable in the CPS program to the `Angle_Change` physical variable are passed from the pseudo method as input to the *Cyber Delegate*. The *Cyber Delegate* will then generate an *Adapter* for the `angle` variable. Since the semantics of this specific example require access to real-time simulation and the action is an actuator command (i.e., to change the setting for the rover's movement angle), the *Adapter* registers with the *RTS* as a publisher for `Angle_Change` data, and the relevant models inside the *RTS* will be updated every time the application invokes the `onSensorChanged` event (which triggers the pseudo method for setting the value of the physical variable `angle`).

While the example shows the annotation attached to a particular method, our prototype of *AOA* also allows annotation at the class level. Another thing to note from the example is the property "mode." The value "Simulation" means the variable is connected to the real-time simulation, while "Physical" means the variable is connected to a real device. There is also a project level "mode" that can connect the entire cyber part with a real-time simulation or with a physical environment. By employing a mix of granularities, a CPS developer can simulate all or just parts of the physical system, making it possible to perform hybrid verification that mixes real-time simulation and real world interaction.

4. REAL-TIME SIMULATION ARCHITECTURE (RTS)

The second significant intellectual contribution of our work is to provide a *Real-Time Simulation* architecture (*RTS*), tailored to CPS, as a software solution to automate the generation of an integrated real-time simulation environment. Using guidance from the CPS application developer, *RTS* combines existing simulation models and connects them to a runtime verification architecture in a way that supports high-fidelity runtime verification and maintains a satisfactory level of simulation accuracy by improving simulation speed and reducing latency. We start by describing our assumptions and other foundational material and then present the architecture of BraceBind's *RTS*, including the details of some of its core algorithms.

4.1. Foundations and Functional Overview

The *RTS* design is based on a few key assumptions. First, we assume the simulation platforms in which the models are created provide utilities to generate executables (e.g., as done by Simulink Coder and LabView C Code Generator) and we assume that the generated code faithfully implements the models. BraceBind's *RTS* requires that essential information about these models is provided in summary form by the CPS developer in an intuitive way. In our prototype, we use an XML schema, though we could alternatively rely on FMI as a *de facto* standard.

Within this description, *RTS* requires the model's inputs and outputs and some other basic information about the model and its realization to be able to connect it properly to the CPS implementation. This information includes the simulation platform (e.g., Simulink), the file name for the model, and the solver algorithm (e.g., Runge-Kutta, Dormand-Prince, or Euler), as required by those utilities (e.g., Simulink Coder) to generate executables.

We assume that each simulation model can be deployed to its own machine to execute and that these machines' local clocks are sufficiently synchronized (i.e., that the worst case drift is below a small and acceptable σ); this is achievable using established clock synchronization algorithms [10, 28]. The isolation of models to machines gives required flexibility for *RTS* to deploy simulation models with optimal performance. For

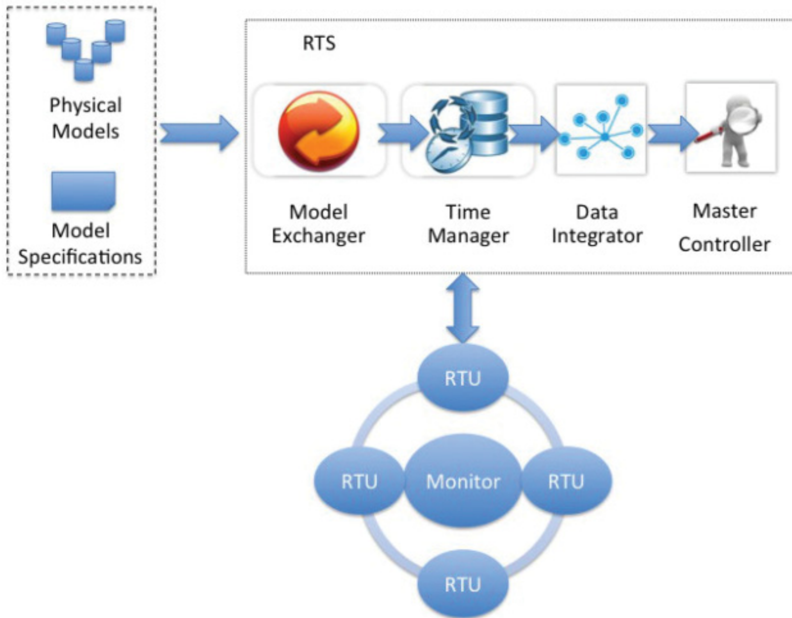


Fig. 4. RTS Architecture Overview.

instance, for a specific simulation model requiring a very small step size (e.g., at the microseconds level), it is highly ideal to deploy the model to its own machine to guarantee required simulation speed and to reduce simulation latency. In this article, we allow the separation of simulation models so these simulation models can be deployed to a separate deployment machine. We leave the study of more optimized and cost saving solutions as independent studies for future work.

In building BraceBind's *RTS*, we focus on improving simulation speed and reducing simulation latency as they are two important factors that determine the output accuracy of a real-time simulation environment. Failure to do so would result in inconsistencies between simulated and actual dynamics, which can render the entire real-time simulation untruthful and misleading [32]. There are a few other factors that are also important to the success of a real-time simulation architecture, including the complexity of models, the experience of CPS developers in creating the simulation models, and the amount of effort required to create the simulation models. We leave the study of these factors as independent studies for future work.

Figure 4 shows how BraceBind's *RTS* generates and manages real-time simulations using existing simulation models. The RTS has four main components: *Model Exchanger*, *Time Manager*, *Data Integrator*, and *Master Controller*. First, using the information provided by CPS developers regarding the simulation models to employ, the *Model Exchanger* invokes the associated vendor-provided utilities to generate executables (e.g., in C or C++) for each model. These executables are generated in such a way to make it possible for them to exchange data with other models in the real-time simulation (e.g., consider the data exchanged between the rover dynamic model and the moon model for the example in Section 1). The challenges in constructing the *Model Exchanger* are primarily in its engineering (e.g., creating a façade for each vendor provided utility); we skip the details for brevity.

The output executables from the *Model Exchanger* are then processed by the *Time Manager*, which adds code to implement time synchronization to the executables to guarantee that outputs from these executables in the real-time simulation environment are consistent with outputs from the actual dynamics simulated by the original physical models. Details of the *Time Manager* are elaborated in Section 4.3. The output executables from the *Time Manager* are then processed by the *Data Integrator*.

The *Data Integrator* adds code to provide data integration to allow optimized data synchronizations across the executables in the real-time simulation environment. For instance, for the rover application introduced before, there are essentially three physical models: the rover motor model, the rover dynamic model, and the rover environment model. Velocity output from the rover motor model is required as one of the inputs to the rover environment model. The frictional force on x and y axles output from the rover environment model are required as inputs to the rover dynamic body model.[5] The output executables from the *Data Integrator* are then processed by the *Master Controller*.

The *Master Controller* synthesizes a publisher (the *Controller Agent*) in each executable with a fixed length queue. The default length is 50, which implies that each executable can store a maximum of 50 historical instances for the outputs. This number is retrieved empirically based on the smallest step size of this rover simulation models. We leave the investigation of the optimal value (as a balance between accuracy and the memory cost) as our future work. As for now, a CPS developer can overwrite this setting empirically depending on how fast the underlying simulation model is expected to run and the memory restriction on the host machines in the real-time simulation environment. After each time step, the simulation's *Controller Agent* stores the simulation's output to the fixed length queue and listens to subscriptions from outside the real-time simulation (i.e., reserved for the integration with runtime verification tool). The *Master Controller* then deploys the executables, which we called Real-Time Units (*RTU*s) based on the deployment instruction for each model provided by the CPS developer as part of model specifications. That is, for each model, the CPS developer can specify the deployment machine's IP Address, File Path, and the Port Number for publishing output data in XML.

For each real-time simulation, the *Master Controller* also synthesizes and deploys a Real-Time Simulation Monitor (*RTSM*) to a dedicated *Monitor* node (i.e., a computer that is connected to every machine hosting an *RTU*). The *Monitor* node monitors the running status of each *RTU* by subscribing to the output data from each *RTU*.

In summary, the *Time Manager* is used to reduce simulation latency by maintaining synchronization among all models in the real-time simulation. The *Data Integrator* is used to improve simulation speed and scalability and to reduce latency by allowing multiple ways for models in the real-time simulation to effectively exchange input and output data. We will walk through these two components in more detail.

4.2. Data Integrator

The *Data Integrator* supports two modes of data integration: one based on message queuing (e.g., MQTT [16]) and another based on a shared memory architecture (e.g., OpenMP [8]). Before the *RTS* deploys an *RTU*, based on each model's input and output information provided by CPS developers, the *Data Integrator* constructs a dependency graph among those models. Based on each model's deployment information, the *Data Integrator* assigns the shared memory data integration mode for those deployment models in the same machine (i.e., deployed to the same IP address). This scenario is applicable to those models requiring faster simulation speed and allowing low latency

[5] Details of the *Data Integrator* are elaborated in Section 4.2.

to maintain accuracy required by the underlying models, which have very small step size (e.g., models for car engine electronic control units).[6]

In shared memory data integration mode, all *RTU*s deployed into a single machine are managed by a *Delegate RTU* (automatically synthesized by the *Data Integrator*) that executes each *RTU* in parallel and exchanges data among these *RTU*s using shared memory. For those models that must exchange data but are deployed as *RTU*s across different deployment machines, the *Data integrator* automatically synthesizes for each *RTU* a message subscriber for every single input data and a message publisher for all output data. To reduce latency in message queuing mode, the *Data Integrator* also automatically synthesizes a directory look-up agent called the *Message Hub*, which contains the IP address and port number for each *RTU*.

Each *RTU* participating in message queuing mode can look up from the *Message Hub* all the required information for designated publishers and later establish direct connection to each publisher required. The *Message Hub* essentially provides a directory service for the *RTU*s and allows direct connection between publishing *RTU*s and subscribing *RTU*s to reduce communication latency.

4.3. Time Manager

In a system that combines real-time simulation with runtime verification, it is essential to guarantee that the real-time simulation produces outputs and internal states at a rate corresponding to the physical process (e.g., if a water tank takes ten minutes to fill in the real world, then a real-time simulation should take exactly or very close to ten minutes to fill a virtual tank with simulated water). To make a real-time simulation valid, the time that elapses in executing a single step of the real-time simulation model, that is, T_E, must be shorter than the wall clock duration of each fixed step size of each model (T_S). For the duration of a model's idle-time (i.e., $T_S - T_E$), the simulation must wait until the clock ticks to the next step. This is needed to guarantee that all of the components in Figure 3 (i.e., the simulations, the real world, and the CPS program itself) are in sync. Without this, the simulation is considered erroneous (i.e., "overrun" occurs) [5].

We omit the synchronization algorithms for T_E and T_S in each real-time simulation model (i.e., in-model synchronization); our in-model synchronization algorithms are able to automatically detect the deployment environment (e.g., in our prototype, we support Windows, Ubuntu, and Mac OS) and activate required platform-specific library calls to get wall-clock time and build a nano-second accurate wait function. In this section, we instead elaborate on our solution of creating the *Time Manager* to guarantee accurate time synchronization and find optimal synchronization *across* real-time simulation models in the deployment environment.

In real-time simulation, physical models with a smaller step size should wait for those (often interactive) physical models with a larger step size to achieve true real-time simulation. To deal with this challenge efficiently, BraceBind's *Time Manager* enforces a concept of *TimeZone*. The *Time Manager* groups RTUs with similar timing requirements into a single *TimeZone* based on user provided model information (e.g., step size). For instance, in our running example, the *Time Manager* groups the rover dynamic RTU and the moon environment RTU into one *TimeZone* that supports microsecond level time synchronization. Other less stringent RTUs can be placed in a different *TimeZone* with a larger time step.

Ideally, all the RTUs in a given *TimeZone* are deployed into one machine whose capabilities are matched to the requirements of the *TimeZone*. Alternatively, the RTUs can execute in an intra-net with a clock synchronization algorithm with sufficient

[6]Message queuing is the default model for *across* machines.

ALGORITHM 2: Find Optimal StepSize

input : *TimeSyncZones* for all TimeZones registered
output: Each time zone is configured with optimized step size

```
1  for ∀zone ∈ TimeSyncZones do
2  |  for ∀model ∈ zone.models do
3  |  |  if !run(model.maxSize, model) then
4  |  |  |  reportTimeOutError(model)
5  |  |  end
6  |  end
7  |  existingMaxSize ← findMax(zone) maxSize ← findMax(zone, existingMaxSize)
   |  maxMinSize ← findMaxMin(zone) for maxSize ≥ maxMinSize do
8  |  |  for ∀model ∈ zone.models do
9  |  |  |  if model.maxSize > maxSize ∧ !run(maxSize, model) then
10 |  |  |  |  goto nextTimeZone
11 |  |  |  end
12 |  |  end
13 |  |  if maxSize == maxMinSize then
14 |  |  |  goto nextTimeZone
15 |  |  end
16 |  |  existingMaxSize ← maxSize maxSize ← findCandidateMax
   |  |  (zone, existingMaxSize) if maxSize == existingMaxSize then
17 |  |  |  maxSize = maxMinSize
18 |  |  end
19 |  end
20 |  nextTimeZone: apply( zone, maxSize)
21 end
```

accuracy [18] (to account for clock drifts among those deployment machines). For each *TimeZone*, the *Time Manager* automatically synthesizes a dedicated timezone delegate agent that subscribes to input data and publishes output data for the *RTU*s inside that *TimeZone* and performs each time step while fully synchronizing with other *TimeZone* delegate agents.

We are also motivated to optimize time steps within each *TimeZone*; reducing the step size of a model's ODE solver by a factor of λ can reduce local error (per time step) by approximately $\lambda * n + 1$ and global error by approximately $\lambda * n$, where n is the order of the ODE solver [25]. The *Time Manager* balances accuracy and computational efficiency by computing the best step size for each *TimeZone* at deployment time (Algorithm 2).

We iterate through each zone to test, for each model, whether the default step size works for the deployed machine (i.e., we check that T_S for the *TimeZone* is larger than T_E). If not, then the algorithm indicates that either the deployment machine is not suitable or the step size is not set correctly (lines 3–4). For each model in the zone, the CPS developer provides a minimum and maximum step size; based on these inputs, we iteratively reduce the step size for the *TimeZone* without causing overrun for any models in the *TimeZone* (lines 8–18).

In Summary, RTS provides an automatic real-time simulation platform that seamlessly works with BraceBind's AOA to provide an integrated solution for runtime verification of CPS. In comparison with the state of the art and state of the practice in real-time simulation, RTS is highly compliant with the original models, guarantees the accuracy of the integrated runtime verification, and is cost-effective (without the need to purchasing expensive and dedicated real-time simulation machines like NI PXI servers [30]).

5. CASE STUDY AND EVALUATION

In this section, we describe our empirical design and explain the results, ending with a discussion of validity.

5.1. The Empirical Design

As an evaluation application, we used an existing CPS application based on the Soft Real-Time Agent Control Architecture [15] and GPGP coordination architecture [26], which are representative of common CPS systems. We acquired this application from other engineers in the CPS domain in an attempt to mitigate biases introduced in using applications implemented just for the purpose of evaluating BraceBind's dual architecture, namely the *RTS* and *AOA*. The autonomous vehicles we used are based on the Rover 5 Platform[7] equipped with accelerometer and orientation sensors. The application is built in Android and deployed to Samsung Galaxy S3 phones that control the rovers. The application coordinates the rovers to visit a set of provided waypoints by assigning each rover a set of *tasks*, where each task is an obligation to visit a waypoint.

We implemented a prototype of *RTS* in C++ and conducted an experiment ***E1*** to evaluate the accuracy of *RTS* by checking whether *RTS*-generated simulations are compliant with original simulation models. To rule out the compliance's reliance on the complexity of the simulation models and the knowledge of CPS developers of using *RTS*, before the experiment, an independent electrical engineer specialized in autonomous vehicles and who was not aware of *RTS* created and tested physical models for the vehicle, including its motor, dynamics, and sensors. Some of these physical models are fairly complex, containing dozens of differential equations, while others are fairly straightforward containing only a few linear equations. The models are all in *Simulink*. We also asked the engineer to create specifications for each model as input to *RTS*. The generated real-time simulation environment executed on two laptops, one running Mac OS X 10.9.3 with 2.5GHz Intel Core i5 and 8G memory and a second running Ubuntu 12.04 with 2.5GHz Intel Core i5 and 4G memory. The models requiring microsecond level latency executed in the first machine; the other models executed in the second machine.

We created a single large simulation combining all the models and ran it in Simulink on the Mac OS X machine (as compared to executing each model as an independent RTU across separate machines in the real-time simulation environment). We created a program that provides inputs for the actuator models and records the outputs from the sensor models in the *RTS*-generated real-time simulation, and we use this program to feed randomly generated inputs to the real-time simulation; we execute this scenario 20 times. Each time, we used the same input to feed into the big Simulink model. The running time of the real-time simulation and the big Simulink model are the same in each run, but they vary together across executions from 5 to 30min. We then compared the outputs, which include the rover's position, velocity, and angle, between this big model and the real-time simulation that *RTS* automatically generates from the models and input specifications. ***E1*** is designed to answer the research question

- ***RQ1*****:** How compliant is real-time simulation established by *RTS* with original simulation models?

We implemented a prototype of *AOA* in C++ and AspectJ and then asked developers familiar with the evaluation application to annotate the code's *PhysicalVariable*s using *AOA* annotations. We conducted another experiment ***E2*** in a lab to test *AOA*'s computation overhead on the observed CPS application and its efficiency of detecting

[7]https://www.sparkfun.com/products/10336.

subtle bugs that are closely related to a specific deployment environment. We deployed the evaluation application to a lab with three surfaces to mimic different physical deployments (wood, grass, and linoleum[8]) and an overhead camera for both positioning and recording video of the tests. The wood environment has a few bumps that we did not include in the environment model. We use this environment to check how *AOA* works given stochastic factors that cannot be easily captured in the physical models. The grass is actually a synthesized grass, while the corresponding simulation model is for true grass. In addition, in the grass environment, we reduce the rover's actual weight by one third while keeping the original weight in the simulation model. This environment exposes how *AOA* works when CPS developers are unable to get accurate values for model parameters and must rely on estimates. The linoleum environment and corresponding simulation model are highly consistent with one another. This environment represents how *AOA* works when a CPS developer is confident in the accuracy of the simulation models. We asked the same electrical engineer in ***E1*** to create the corresponding simulation models. We deployed and tested the application in the corresponding simulation deployment environments.

We used BraceBind's *AOA* to connect the application with either the physical deployment environment (physical rovers and a specific lab environment) or the simulation environment (simulated rovers and a specific simulated environment established by *RTS*). We used a lock free linked list to merge the trace file from *AOA* with the trace file from an existing runtime verification tool [36]. The verification tool is used to detect violations of the following properties in the CPS application: the completion time for a given task is bounded (*P1*); the integral of cross track error is bounded (i.e., the vehicle is not weaving) (*P2*); and the duration of the main control loop is bounded (*P3*). Since the verification tool has no knowledge of whether the application is deployed to a physical environment or to a real-time simulation environment, we recorded the number of property violations found by the runtime verification both for the physical deployment and the real-time simulation by manually checking the violations report generated by the verification tool after each test and compared the two versions. We instructed the application to issue 180 tasks to the rovers for each test. We also recorded the CPU and Memory usage when the *AOA* runtime is executing along with the observed application and compare this overhead against the application running without the *AOA*. To understand more what impacts the monitoring accuracy (when the verification tool is running against the real deployment environment or against the real-time simulation), we also compare the timestamped positions of each rover across the physical deployment and the real-time simulation. ***E2*** is designed to answer the following research questions:

- ***RQ2*:** How effectively does *AOA* detect bugs when integrated with an existing verification tool?
- ***RQ3*:** What is the computational overhead of running *AOA* along with the observed application?
- ***RQ4*:** How truthful is the real-time simulation connected by *AOA* in comparison with the modelled physical deployment environment? In our case study application, this means, in particular, how faithful is the simulation connected by AOA in capturing the movement behavior of the rovers relative to the physical environment. What factor, in terms of errors and uncertainties for the values of parameters in the underlying simulation models, affect the truthfulness?

[8]Sample videos at: https://goo.gl/PkxDa4 (wood); https://goo.gl/Vv3fF2 (grass); and https://goo.gl/s097Ag (linoleum).

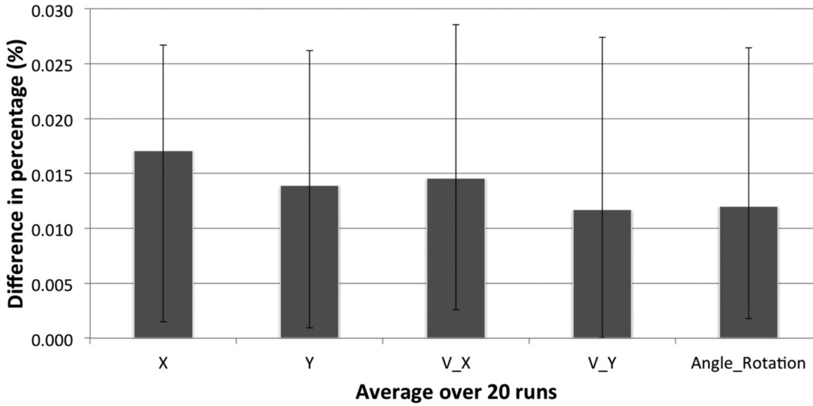
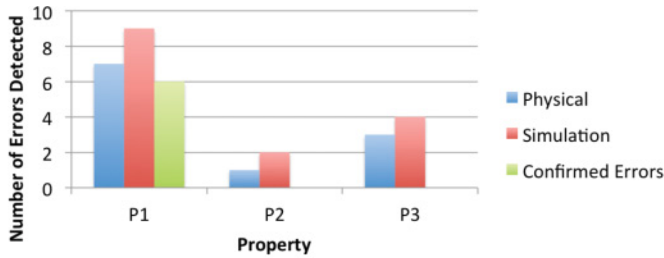


Fig. 5. RTS vs. Simulink.

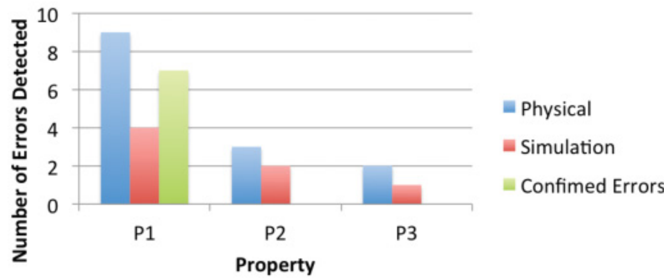
5.2. Empirical Results

RQ1 - Compliance with Simulation Models. Figure 5 plots the average difference between each output parameter using the *RTS* real-time simulation versus the big simulation in Simulink. The outputs from the big simulation are the X position, the Y position, the velocity of X, the velocity of Y, and the angle of a rover. These differences are *very* small (the maximum error, across all runs and all variables, was 0.029%), demonstrating that *RTS*'s real-time simulation environment is highly compliant to the original simulation models. The big simulation model is ideal in terms of accuracy but very expensive in computational overhead. When executing the big simulation model in the Mac OS X machine used for the experiments, we noticed the CPU quickly surged to more than 95% utilization, which shows that the real-time simulation version of the big simulation model most likely requires a dedicated real-time simulation server [30]. Our empirical results are similar to the findings in Ref. [3], which also found this type of simulation model requires a long time to complete the needed calculations. *RTS*'s accuracy is attributable to highly accurate time synchronization, low latency data integration, and step size optimization (i.e., improve simulation speed and reduce latency).

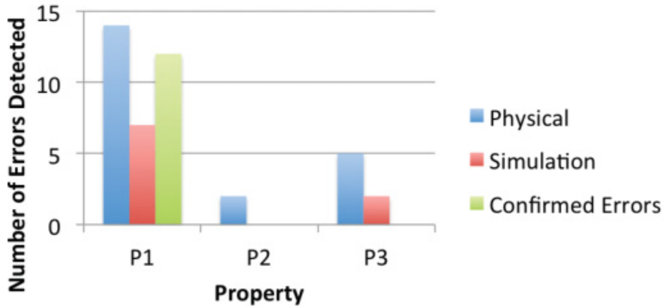
RQ2 - Effectiveness of *AOA* to detect "Wild" Bugs. Figure 6 shows that we were able to detect "Wild" bugs in both the physical deployment and real-time simulation as supported by *AOA*. In the Linoleum environment, six of the errors (for *P1*) were confirmed by the CPS developers to be actual bugs. The remaining errors are assumed to be true positives if they were found in the physical deployment and false positives otherwise. For the Linoleum environment, runtime verification supported by BraceBind's *AOA*-integrated real-time simulation had 0 false negatives (i.e., no (known) errors were not detected) and 1 to 3 false positives (i.e., errors "detected" based on the real-time simulation that were not actual errors). In the Wood environment, the *AOA*-integrated real-time simulation actually missed some confirmed errors (for *P1*) and some assumed errors for *P2* and *P3*. This implies a (likely small) false negative rate for the *AOA* case and a false positive rate that is on par with prior results [36]. These false negatives are mainly attributed to the bumps in the wood floor that are not accounted for in the physical model. Finally, in the Grass environment (where the simulation models deviated substantially from the physical deployment), there is, as expected, a larger number of false positives (i.e., 5 for *P1*, 2 for *P2*, 3 for *P3*), meaning that there is a limit to the applicability of *AOA* for runtime verification when the simulation models are not accurate.



(a) Linoleum



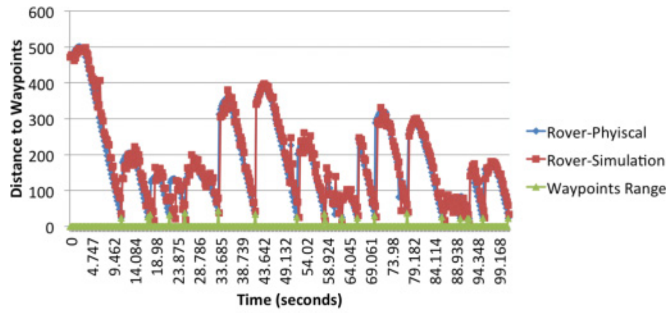
(b) Wood



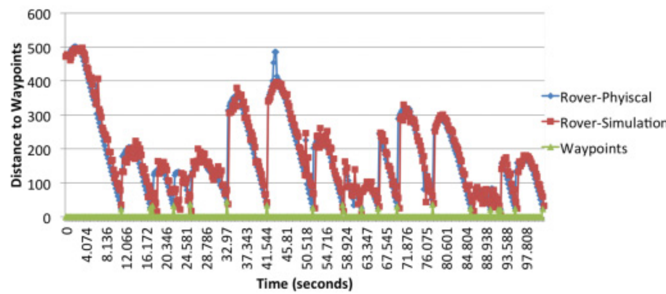
(c) Grass

Fig. 6. Detection of Bugs in the Wild.

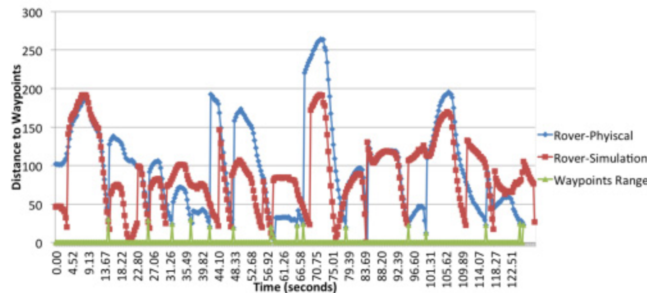
RQ3 - Computational Overhead & RQ4-Accuracy of Real-time Simulation. We measured the CPU and Memory overhead for running the evaluation application with and without the *AOA* runtime on a PC running Ubuntu 12.04 with 2.5GHz Intel Core i5 and 4G memory (average from 5 runs). We notice only a negligible increase in CPU usage and about 5–7% memory increase, which is reasonable given that the *AOA* runtime architecture is based on well established AOP and message queueing. The memory increase is related to the memory requirement for both AOP, generall, and, specific to BraceBind, the use of lock free queues to store values for *Physical Variables*. Figure 7 (top) shows the trajectories of one test vehicle for 19 assigned waypoints, both in the Linoleum environment and in *AOA*-integrated simulation of that environment. This trajectory is representative of trajectories for both vehicles across all waypoints. The Y axis shows the distance to the next waypoint. The Waypoints Range series shows



(a) Linoleum



(b) Wood



(c) Grass

Fig. 7. Faithfulness of *BraceBind* to Simulation.

the acceptance range of each waypoint (e.g., if the vehicle is within 25 centimeters of a waypoint, the vehicle is considered to have reached the waypoint). The simulation result from the *AOA* case is very consistent with the real deployment environment with average error (over both vehicles and across all trajectories) between 5.76% to 11.20%. The errors result from:

- though the physical models used are thoroughly tested, they still contain small errors with respect to their representation of the physical world; and
- we did not build a model of the battery into the simulation environment; we attempted to experiment with fully powered batteries in each experiment, but battery dissipation does have a small impact on the error.

Figure 7 (middle) shows the wood deployment. In this environment, there are a few bumps that cause the vehicle to stray slightly from the expected track (e.g., the spike around 43s). The error for the real-time simulation in this test is from 5.77% to 11.33%. Even with unexpected physical conditions, BraceBind's *AOA* can still guarantee acceptable error. Further, a more detailed model of the physical environment (e.g., one that models bumps) would give real-time simulation results that more faithfully represent the physical deployment (although the *trajectories* themselves may not line up if the arbitrary bumps in the real-time simulation are not in the same locations as in the physical world).

In the grass environment, the simulation deviated significantly from the actual environment, resulting in unacceptable error ranges (from 17.3215% to 81.7556%). This is a direct result of the settings for this scenario; we crafted it to intentionally demonstrate how BraceBind performs when the simulation model is not a faithful representation of the real physical deployment. However, the execution with *AOA* provides a decent representation of the trajectory, and *AOA* can still give a rough feel about how an application might behave in an "uncertain" environment.

In summary, these experiments show that BraceBind's *AOA* is highly effective and efficient in capturing "wild" bugs that are otherwise hard to detect (at least repeatedly) from a specific deployment environment. The reason behind BraceBind's demonstrated effectiveness and efficiency is its dual architecture, in which the real-time simulation (*RTS*) is deployed to separate machines to run in parallel and is ready to provide data whenever it is required. At the same time, the aspect-oriented architecture (*AOA*) runs on top of an existing well established AOP framework (e.g., in our case AspectJ) and high throughput concurrent data structure to store the latest values for relevant *Physical Variables* (in Figure 3).

5.3. Discussion

In addition to the results here, we have tested the impact of less accurate time synchronization. The result is that vehicles in the simulated environments often get stuck in an endless loop while trying to reach a waypoint, because the position data becomes completely out of sync. We also tried a message passing system with more latency (i.e., MQTT[9]) instead of ZeroMQ; the output data from the real-time simulations cannot reach the models fast enough for the view of the (simulated) physical world to remain consistent. The message passing system chosen must be suitable for the application; this concern may be largely mitigated by advances in networks and communication protocols.

Our prototype of *RTS* can support simulation platforms beyond Simulink. We have used LabView to create the vehicle motor model and used it in the experiments. The accuracy is good; however, the LabView generated C code is more computationally intense than the Simulink converted C++ code, causing the CPU utilization in our test machine to climb over 100%. This finding bolsters the need for *RTS*; without its optimizations, simulation-based run-time verification almost certainly requires special hardware. Compared with the state of the art using dedicated simulation servers running all simulation models together, *RTS* provides a more scalable and cheaper solution for a real-time simulation environment. Moreover, *AOA* integrates real-time simulation with runtime verification tools seamlessly, which is not provided by existing tools.

The empirical results in *E3* use our runtime verification framework as the baseline, which has demonstrated a very low number of false positives with negligible (and most of time with no) false negatives [36]. Moreover, for *P2* where we do have existing logs to confirm the real number of errors, we provide the confirmed errors to prove the

[9]http://mqtt.org/.

trustworthiness of the runtime verification framework and increase the accuracy of the test results.

When measuring computational overhead for *AOA*, we cannot simply measure the CPU and Memory overhead on Android phones (where the evaluation application is really deployed) as all the existing measurement tools incur too much computational overhead on the phones and stop the evaluation application from running properly. Instead we run the evaluation application with *AOA* runtime enabled or disabled to check the overhead for *AOA* on a standard PC. The real result of computation overhead incurred by *AOA* might be even better considering any real sensor/actuator access would incur additional computation overhead (while in our environment with a PC, there would be no such cost).

Though we use a relatively simple and low-speed physical rover, the physical models we use are quite representative of any electric vehicle models both in terms of number of sub-models and types (e.g., sensors, actuators, environment, and dynamic models). As future work, we intend to apply our approach to verify safety properties for a humanoid robotics application [9] and safety properties for modern vehicle systems.

In terms of the limitations of our approach, our *RTS* requires the domain expert to possess detailed knowledge of the physical systems underlying the application's implementation and to be familiar with using simulation platforms (e.g., LabView and MatLab) to create high-fidelity models for the underlying systems to achieve robust verification results. If the simulation models miss some important features of the deployment environment, then the real-time simulation environment produced by *RTS* would result in large errors, and our approach could miss some bugs. This is demonstrated in the grass deployment environment, where the grass model (for real grass) in the real-time simulation is different from the deployment environment (synthesized grass) and the rover's weight in simulation is one third heavier than the real rover in the deployment environment. We crafted this scenario specifically to mimic one in which important physical features are misrepresented in the models. In our future work, we can bring stochastic features into *RTS* to compensate for potentially flawed simulation models (e.g., stochastic differential equation model in Ref. [19]). Finally, our approach also requires explicit formal properties to be written for the CPS systems, thus is not applicable to legacy systems where the original requirements are largely missing. In our future work, we can use data mining techniques to generate formal properties for systems from existing logs or simulation traces as in Ref. [17].

6. CONCLUSION

In this article, we created a dual architecture to leverage real-time simulation to aid CPS runtime verification. The dual architecture requires a CPS developer to provide annotations and then relies on Aspect Oriented Programming to seamlessly integrate real-time simulation with runtime verification tools. In our implementation, we tackled challenges in establishing real-time simulation with affordable, scalable, and highly accurate solutions. In our case study, we proved that the real-time simulation established is highly compliant with existing simulation models, highly accurate compared with the corresponding deployment environments, and, most importantly, provides a repetitive and flexible environment to aid runtime verification in finding bugs detectable otherwise only in a physical deployment environment.

ACKNOWLEDGMENTS

The authors thank Stephanie McArthur from NICTA Australia and John Lam from University of New South Wales for help building the evaluation application and conducting tests and ZiWang Lu from Tsinghua University for the simulation models used in the evaluation.

REFERENCES

[1] Ahmad T. Al-Hammouri. 2012. A comprehensive co-simulation platform for cyber-physical systems. *Comput. Commun.* 36, 1 (2012), 8–19.

[2] J. Bastian, C. Clauß, S. Wolf, and P. Schneider. 2011. Master for co-simulation using FMI. In *Proceedings of the 8th International Modelica Conference*. Citeseer.

[3] Marcin Baszyński. 2016. Low cost, high accuracy real-time simulation used for rapid prototyping and testing control algorithms on example of BLDC motor. *Arch. Electr. Eng.* 65, 3 (2016), 463–479.

[4] R. Bednar and R. E. Crosbie. 2007. Stability of multi-rate simulation algorithms. In *Proceedings of Summer Computer Simulation Conference (SCSC'07)*. 189–194.

[5] J. Bélanger, P. Venne, and J. N. Paquin. 2010. The what, where, and why of real-time simulation. Planet RT 1.1: 25–29.

[6] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, and others. 2012. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International Modelica Conference*. 173–184.

[7] H. X. Chen. 2010. Simulink and VC-based hardware-in-the-loop real-time simulation for EV. In *Proceedings of Electric Vehicle Symposium (EVS-25'10)*.

[8] L. Dagum and R. Enon. 1998. OpenMP: An industry standard API for shared-memory programming. *Comput. Sci. Eng. IEEE* 5, 1 (1998), 46–55.

[9] Robocup Federation. Robocup Normal League. Retrieved at http://www.robocup.org/leagues/5.

[10] C. Fetzer and F. Cristian. 1995. An optimal internal clock synchronization algorithm. In *Proceedings of the Conference on Computer Assurance (COMPASS'95)*.

[11] A. Gholkar, A. Isaacs, and H. Arya. 2004. Hardware-in-loop simulator for mini aerial vehicle. In *Proceedings of the Real-Time Linux Workshop.*

[12] D. Goswami, R. Schneider, and S. Chakraborty. 2011. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *Proceedings of the Asia and South pacific design Automation Conference (ASP-DAC'11)*.

[13] M. Harakawa et al. 2005. Real-time simulation of a complete PMSM drive at 10 μs time step. In *Proceedings of the International Symposium on Parameterized and Exact Computation (IPEC'05)*.

[14] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. 1995. What's decidable about hybrid automata? In *Proceedings of the Symposium on Theory of Computing (STOC'95)*.

[15] B. Horling, V. Lesser, R. Vincent, and T. Wagner. 2006. The soft real-time agent control architecture. *Auton. Agents Multi-Agent Syst.* 12, 1 (2006), 35–91.

[16] U. Hunkeler, H. L. Truong, and A. Stanford-Clark. 2008. MQTT-SâĂŤA publish/subscribe protocol for wireless sensor networks. In *Proceedings of Comsware*. IEEE, 791–798.

[17] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. 2015. Mining requirements from closed-loop control models. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 34, 11 (2015), 1704–1717.

[18] T. Jones and G. A. Koenig. 2010. A clock synchronization strategy for minimizing clock variance at runtime in high-end computing environments. In *Proceedings of the Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'10)*.

[19] Michele L. Joyner, Chelsea R. Ross, Colton Watts, and Thomas C. Jones. 2014. A stochastic simulation model for anelosimus studiosus during prey capture: A case study for determination of optimal spacing. *Math. Biosci. Eng.* 11, 9 (2014).

[20] A. B. Khaled, M. B. Gaid, N. Pernet, and D. Simon. 2014. Fast multi-core co-simulation of cyber-physical systems: Application to internal combustion engines. *Simulat. Model. Pract. Theory* 47 (2014), 79–91.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'97)*. Springer, 220–242.

[22] M. Kinsy, O. Khan, I. Celanovic, D. Majstorovic, N. Celanovic, and S. Devadas. 2011. Time-predictable computer architecture for cyber-physical systems: Digital emulation of power electronics systems. In *Proceedings of the Real Time Systems Symposium (RTSS'11)*. IEEE, 305–316.

[23] W. H. Kwon and S.-G. Choi. 1999. Real-time distributed software-in-the-loop simulation for distributed control systems. In *Proceedings of International Symposium on Computer Aided Control System Design.* IEEE, 115–119.

[24] LabVIEW RealTime. 2016. LabVIEW RealTime. Retrieved from http://www.ni.com/labview/realtime/. (2016).

[25] LabViewManual. 2016. LabView User Manual. Retrieved from http://autnt.fme.vutbr.cz/lab/FAQ/labview/SimulationModule_UserManual_371013c.pdf (2016).

[26] V. Lesser et al. 2004. Evolution of the GPGP/TAEMS domain-independent coordination framework. *Auton. Agents Multi-Agent Syst.* 9, 1 (July 2004), 87–143.

[27] B. Miller, F. Vahid, and T. Givargis. 2011. Application-specific codesign platform generation for digital mockups in cyber-physical systems. In *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn'11)*.

[28] D. L. Mills. 1991. Internet time synchronization: The network time protocol. *IEEE Trans. Comm.* 39, 10 (October 1991), 1482–1493.

[29] Modelisar 2016. Modelisar. Retrived at http://www.modelisar.org. (2016).

[30] PXI 2016. What's PXI. Retrived at http://www.ni.com/pxi/whatis/. (2016).

[31] PXIPrice. 2016. PXI Sample Price. Retrived at http://sine.ni.com/nips/cds/view/p/lang/en/nid/210825. (2016).

[32] Camille Alain Rabbath, M. Abdoune, and Jay Belanger. 2000. Real-time simulations: Effective real-time simulations of event-based systems. In *Proceedings of the 32nd Conference on Winter Simulation*. Society for Computer Simulation International, 232–238.

[33] J. J. Sanchez-G., R. D'Aquila, W. W. Price, and J. J. Paserba. 1995. Variable time step, implicit integration for extended-term power system dynamic simulation. In *Proceedings of the Conference on Power Industry Computer Applications (PICA'95)*.

[34] Wei Yan, Yuan Xue, Xiaowei Li, Jiannian Weng, Timothy Busch, and Janos Sztipanovits. 2012. Integrated simulation and emulation platform for cyber-physical system security experimentation. In *Proceedings of the 1st International Conference on High Confidence Networked Systems*. ACM, 81–88.

[35] Z. Zhang et al. 2013. Co-simulation framework for design of time-triggered cyber physical systems. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS'13)*.

[36] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez. 2015. BraceAssertion: Runtime Verification of Cyber-Physical Systems. In *Proceedings of the Conference on Mobile Ad Hoc and Sensor Systems (MASS'15)*.

[37] X. Zheng, D. E. Perry, and C. Julien. 2014. Braceforce: A middleware to enable sensing integration in mobile applications for novice programmers. In *Proceedings of International Conference on Mobile Software Engineering and Systems*. ACM, 8–17.

Received July 2016; revised November 2016; accepted February 2017