BRACE: An Assertion Framework for Debugging Cyber-Physical Systems

Kevin Boos, Chien-Liang Fok, Christine Julien, Miryung Kim
Center for Advanced Research in Software Engineering
University of Texas at Austin
kevinaboos@utexas.edu, {liangfok, c.julien}@mail.utexas.edu, miryung@ece.utexas.edu

***Abstract*—Developing cyber-physical systems (CPS) is challenging because correctness depends on both logical and physical states, which are collectively difficult to observe. The developer often need to repeatedly rerun the system while observing its behavior and tweak the hardware and software until it meets minimum requirements. This process is tedious, error-prone, and lacks rigor. To address this, we propose BRACE, a framework that simplifies the process by enabling developers to correlate cyber (i.e., logical) and physical properties of the system via assertions. This paper presents our initial investigation into the requirements and semantics of such assertions, which we call *CPS assertions*. We discusses our experience implementing and using the framework with a mobile robot, and highlight key future research challenges.**

I. INTRODUCTION AND MOTIVATION

Consider a cyber-physical system (CPS) consisting of mobile robots operating in a smart home in which both the robots and the home contain a plethora of sensors and actuators. The sensors enable software applications to perceive the physical environment, while the actuators enable them to change the environment's physical state, e.g., by moving objects, toggling switches, and turning knobs. Applications include patrolling the home for security or automatically adjusting the lights or thermostat based on occupants' locations.

Traditional debugging tools are insufficient since they only consider a program's logical state. To debug a CPS, programmers must jointly reason about logical and physical state. We propose BRACE, a *cyber-physical assertion* middleware and framework. First, BRACE introduces new forms of assertions catered to the unique demands of CPS. For example, CPS assertions can span logical and physical variables and nodes, and specify both spatial and temporal properties. They are checked by an external, omniscient, process that can independently observe the physical states. Second, BRACE supports asynchronous checking of CPS assertions to avoid critical timing failures caused by processing latencies. Third, BRACE supports explicit actuation of error-handling code, e.g., when an assertion is violated, the system can be configured to either halt or execute a user-provided callback function. Fourth, BRACE can be configured to tolerate spatial and temporal discrepancies.

Consider the example in Figure 1, which shows three instances of the same program that moves the robot 1m forward, turns off the light, and, if successful, moves the robot another meter forward. Figure 1(a) shows how this program would be written using existing techniques. The onus is on the developer to determine the program's correctness. Developers need to *sense* the physical state and correlate this state with the program's logic to determine correctness. This is error-prone and sometimes infeasible because many cyber-physical systems involve a plethora of networked devices that rapidly performing actions in parallel, rendering manual observation extremely difficult.

```
1. // Developer observes robot's initial location
2. robot.move(1);
3. if(turnOffLight()==SUCCESS) robot.move(1);
4. // Developer observes robot's final location and state
5. // of the lights; concludes if program executed correctly
```

(a) Without CPS assertions

```
1. let prevLoc = getCurrentLoc();
2. robot.move(1);
3. if(turnOffLight()==SUCCESS) robot.move(1);
4. assert(getRoomBrightness() == DARK
5.    && prevLoc + 2 == getCurrentLoc());
```

(b) Manually-created CPS assertions

```
1. ConfigCPMap(roomBrightness,lightSensor());
2. ConfigCPMap(robotLocation,cameraSensor());
3. initCPState(robotLocation, 0);
4. robot.move(1);
5. if(turnOffLight()==SUCCESS) robot.move(1);
6. CPSAssertAsync("roomBrightness == DARK
7.    && robotLocation == 2");
```

(c) CPS assertions with BRACE

Figure 1. Example program with and without cyber-physical assertions.

To address this problem, one may attempt to manually code CPS assertions, as shown in Figure 1(b). In this program, the developer must implement accessors to physical state (e.g., `getCurrentLoc()` and `getRoomBrightness()`). Creating these methods is non-trivial as it requires sensing the environment and transforming that physical state into a logical program state.

Figure 1(c) shows CPS assertions written using BRACE. The correlation between logical variables and physical state is defined by configuration statements on lines 1-2. The logical variable `roomBrightness` is mapped to a BRACE-provided service called `lightSensor`, which uses a photocel to measure the brightness of the room. The logical variable `robotLocation` is mapped to a service called

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland
New Ideas and Emerging Results

cameraSensor, which uses a camera to track the robot's location. Once these mappings are formed, BRACE automatically updates the mapped logical variables based on physical state as determined by the specified service. Line 3 initializes the current location of the robot to zero, defining a reference for robotLocation and allowing developers to use a relative location instead of absolute coordinates on line 7. Lines 4-5 contain the program's core functional logic. Lines 6-7 contain a CPS assertion specifying that if the room is dark, the robot must have moved two meters. This assertion only references logical variables automatically maintained by BRACE, which are significantly easier to write than the methods in Figure 1(b).

In this paper, we undertake three concrete tasks related to BRACE: (1) we identify requirements and types of CPS assertions; (2) we design assertion APIs that jointly operate over logical and physical states; and (3) we present a prototype implementation of location-centric CPS assertions. The remainder of this paper is organized as follows. Section II presents related work, followed by our design of CPS assertions and preliminary evaluation in Section III. We conclude with our planned next steps in Section IV.

II. RELATED WORK

Assertions are one of the most useful automated techniques available for detecting and locating faults, even when faulty code is executed but does not cause a failure [1]. BRACE aims to make it easier for programmers to write and check assertions across both logical and physical states of a cyber-physical system. Its main goal is to abstract and aggregate over various sources of sensor inputs to help developers check explicit system properties at runtime.

Attempts to rigorously validate complex cyber-physical systems exist. Cleveland et al.'s *instrument-based validation* attaches *monitors* to controller models during simulation [2]. BRACE differs by checking runtime assertions within an actual system as opposed to in simulation, as well as checking assertions that span time and multiple nodes. Passive distributed assertions [3] is a tool that allows developers to add assertions about a global network state. It works by having each node broadcast small amounts of additional information that is received by a secondary "sniffer network," which analyzes this data to determine the assertion's validity. This is similar to BRACE's continuously evaluated assertions. The main difference is that BRACE allows users to explicitly map logical variables to physical states using configuration statements, making it easier to write CPS assertions.

Mercadal et. al [4] use a domain-specific Architecture Description Language (ADL) to safely handle application- and system-level errors in pervasive computing systems. Their ADL requires a special compiler to generate Java code, whereas BRACE only relies on the default Java compiler. BRACE focuses on identifying and reacting to unexpected buggy behavior by using an external sensor network and monitoring process, instead of trying to recover from severe errors or influence the flow of the original CPS application.

```
1. ConfigCPMap(robot1Loc, locSensor(robot1));
2. ConfigCPMap(robot2Loc, locSensor(robot2));
3. ConfigCPMap(babyRoomTemp, tempSensor);
4. ...
5. MonitorCPProp("robot1Loc == (1,1)
6.    AND robot2Loc == (2,1) at time 1pm");
7. MonitorCPProp("robot1Loc == (1,2)
8.    AND robot2Loc == (2,2) at time 2pm");
9. MonitorCPPropAction("babyRoomTemp > 71
10.   AND babyRoomTemp < 73", ALARM());
```

Figure 2. Example continuous assertions.

Macrodebugging [5] is a debugging tool for wireless sensor networks that allows developers to sequentially step through application code despite the fact that the code executes asynchronously across distributed nodes. For example, a programmer can set a breakpoint on a *macro program*, which maps to respective program execution points in individual micro programs. BRACE's focus is to make it easier to write and check CPS assertions, as opposed to providing methods for setting breakpoints and investigating captured traces. Also related is automatic calibration [6], which differs from the goals of BRACE because its objective is to fine-tune program parameters to make an observed value match a desired value according to a user-defined model. BRACE's goal is to simplify the process of defining and checking system properties at runtime.

III. APPROACH AND PRELIMINARY EVALUATION

We have three primary objectives: (1) APIs for expressing cyber-physical assertions that jointly reason about logical program state and physical environment state, (2) a middleware that converts physical state into logical state, and (3) runtime processes that evaluate CPS assertions.

A. Key Requirements of BRACE

Mapping Cyber and Physical States. BRACE provides logical abstractions of the physical environment that CPS programs can directly reference. BRACE's cyber-physical assertions only reference logical variables, some of which are mapped to physical properties. This mapping is done internally within BRACE, thus simplifying the process of implementing these assertions. Some examples of mapping cyber and physical states include (1) position sensors that measure the location of a mobile robot, which can be mapped to a location variable in a program that coordinates a multi-robot security patrol; (2) temperature sensors embedded in a home, which can be mapped to a temperature variable in a program that controls an HVAC system; and (3) occupancy and identification sensors, which can be mapped to user-preference variables in a program that controls a smart-home's audio system. These are just a few examples of the ways in which cyber and physical states can be mapped.

Synchronous vs. Asynchronous Assertions. Cyber-physical systems are inherently time-sensitive, making them sensitive to latencies introduced by interruping normal program flow with assertion checks. To address this, BRACE introduces the notion of *asynchronous* cyber-physical assertions. The goal is to evaluate assertions only when the processing latency will not impact the application. Continuous assertions, like those shown in Figure 2,[1] are executed by third-party "observer" processes and thus have minimal impact on the temporal properties of the system. In-lined synchronous assertions, such as those shown in Figure 1(c), introduce latencies if the assertion must execute *sequentially* with the surrounding code. BRACE allows assertions to be executed asynchronously upon a user request, as exemplified by the `CPSAssertAsync` assertion on line 6 of Figure 1(c). When the program reaches an asynchronous assertion, the assertion is not immediately evaluated; instead, the states of the variables referenced by the assertion are saved, and a background task is spawned to evaluate the assertion when the program is idle. One consequence is that the program may execute beyond where the assertion occurred, making fault localization more difficult since faults would have more time to propagate. We evaluate assertions on a first-come first serve basis, meaning that asynchronous assertion tasks will always execute in the order they were spawned. These assertions are evaluated using the state of the system at the time when the assertion was requested, which is before the fault can propogate. In addition, a user still has the option to specify a synchronous assertion, which mandates that the assertion must be completely evaluated before executing the next program statement.

Continuous Assertions. CPS applications must interact with the real world, in which time and space are intrinsic. For this reason, BRACE supports continuous assertions that span a period of time by checking system invariants continuously. Suppose the CPS application involves two robots patrolling the perimeter of an infant's smart-bedroom to ensure the baby's safety. The application's correctness depends on whether the robots move in a coordinated and timely fashion and whether the room's temperature stays within certain limits. Using BRACE, the program's correctness can be automatically ascertained using the assertions in Figure 2. These assertions span multiple nodes, locations, and time frames. The assertion on lines 9-10 demonstrates how the program can be interrupted by a callback to error handling code (e.g., `ALARM()`) if the assertion fails. These assertions effectively define invariants because they are continuously evaluated over the program's execution, not just at particular points in the application's code, as with the assertions in Figure 1(c). If execution of the CPS could be discretized into a sequence of actions, these assertions would be evaluated after each action. In reality, our framework uses a separate "third-party" process to continuously evaluate these assertions without interfering with the application.

[1]For clarity, we present pseudo-code, not an actual Java-based API.

Table I
DEFAULT SENSOR SUPPORT

Dimension	Sensor Type	Provides
Location	2-D Camera	2-D Position♢
	3-D Camera	3-D Position
	Infrared	Distance/Range♢
	GPS	2-D Position♢
	Cricket Mote	2-D Position, Range♢
	RFID	Proximity
	Sonar	Distance/Range
Orientation	Compass	Bearing♢
	Gyroscope	Spin, Orientation
	Magnetometer	Magnetic Fields
Force	Force Sensor	Force, Contact
Light	Photodetector	Ambient Brightness
Temperature	Thermometer	Temperature
Pressure	Barometer	Pressure
Acceleration	Accelerometer	Acceleration, Tilt♢

Tolerant Assertions. CPS assertions reference both physical and program state. Due to inherent limitations in measuring physical state, BRACE assertions can specify tolerance values that can be either a numerical margin of error or a temporal window in which to watch for acceptable values. For example, a developer may modify the assertion in Figure 2, lines 5-6, as follows:

```
MonitorCPProp("robot1Loc == (1 ± 0.2,
1 ± 0.4) && robot2Loc == (2 ± 0.1,
1 ± 0.3) at time 1pm ± 5min")
```

These tolerance values help prevent real-world sensor inaccuracies from causing assertions to fail.

An Extensible Framework for Sensor Support. BRACE will incorporate a wide variety of sensor interfaces, allowing the programmer to focus on more abstract CPS functionality instead of low-level device drivers. Table I lists the default sensor interfaces included in BRACE and the usage of each interface. BRACE will provide software hooks enabling new sensing services to be added to account for the peculiarities of a specific deployment environment.

Table II
CYBER-PHYSICAL ASSERTION APIS

Assertion API	Description
`ConfigCPMap(l,p)`	Maps physical state p to a logical var. l
`InitCPState(l,v)`	Assigns initial value v to logical var. l
`CPSAssertAsync(a)`	Asynchronous execution of assertion a
`CPSAssertAsync-Action(a, h)`	Asynchronous execution of assertion a with callback to error-handling code h
`CPSAssertSync(a)`	Synchronous execution of assertion a
`CPSAssertSync-Action(a, h)`	Synchronous execution of assertion a with callback to error-handling code h
`MonitorCPProp(a)`	Continuous checking of assertion expr. a
`MonitorCPProp-Action(a, h)`	Continuous checking of assertion expr. a with callback to error-handling code h

B. Prototype Implementation and Initial Results

Our initial implementation of BRACE is written in Java and uses both Player/Stage and ROS robotics middleware;

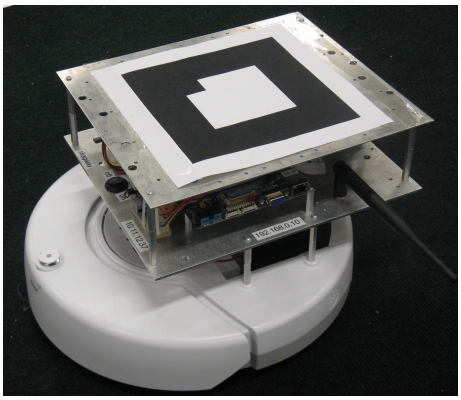

Figure 3. Our mobile robot: Roomba with top-mounted decal.

its API is shown in Table II. We evaluated it using an iRobot Create with a top-mounted decal for camera-based localization, as seen in Figure 3. A ceiling-mounted camera tracks the decal as the robot moves, and BRACE polls the camera for location updates. In our sample application, we instruct the uncalibrated robot to execute several consecutive movement commands and evaluate an assertion comparing its expected location against its actual location after each movement. Figure 4 displays a sample traversal;[2] in this case an assertion was performed at each vertex in the path, using syntax such as `CPSAssertAsync("robotLoc == (0 ± 0.05, 1 ± 0.05)")`. In this case, it is visually apparent where the robot begins to stray, and BRACE successfully detects and quickly reports the discrepancy between the expected (logical) and actual (physical) location. BRACE also saves a time-stamped trace of the assertion history for postmortem analysis. We used both synchronous and asynchronous assertions across several trials; synchronous assertions allowed us to pinpoint exactly when and where the robot strayed, while asynchronous assertions provided the same error localization at a later point in time.

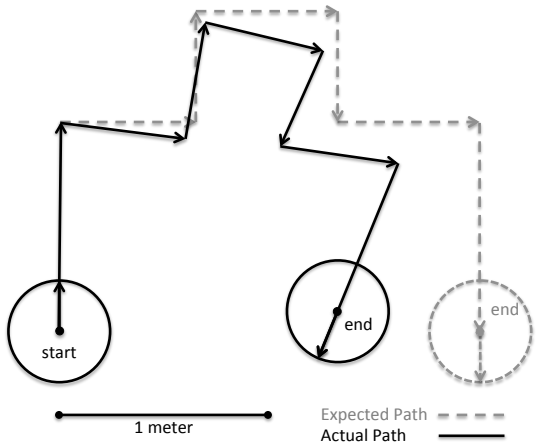

Figure 4. An example traversal path.

[2]For a video of the system running, see: http://tinyurl.com/78x4gb2

We also measured the memory overhead and execution delay of performing assertions with camera-based localization. BRACE uses a negligible amount of memory (a few kilobytes at most) to store the data required for making assertions, which is quickly reclaimed after saving the assertion trace to disk. On average, each *synchronous* assertion delays the program's execution by 22.89 ± 6.13ms, but each *asynchronous* assertion delays the program by only 0.16± 0.06ms. These delays represent the time required to read and store necessary state data and to perform a context switch between the CPS application and BRACE. Clearly, the execution delay depends on the complexity and location of the sensor. The benefit of asynchronous assertions is demonstrated here by the ∼140X improvement in execution delay over synchronous ones. These preliminary results demonstrate the *potential* for BRACE to operate within constraints necessary for time-sensitive CPS applications.

IV. NEXT STEPS

In the short term, we plan to support more sensor types and services to enhance BRACE's ability to collect physical data. Table I shows the list of sensors that are already supported by BRACE (denoted by ♢) and the sensors planned for future support. While our current implementation only supports in-lined assertions, we plan to implement support for continuous and temporally-tolerant assertions.

In the long run, we plan on providing an off-line debugging tool that enables developers to investigate runtime traces captured by BRACE to identify the root cause(s) of failed assertions. We also envision using CPS assertions as a basis for automatically correcting CPS system behavior in the future. By differencing expected physical states and actual physical states, a novel patch-generation algorithm can modify the program to make physical states match the corresponding logical states.

REFERENCES

[1] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 25–37, May 2006.

[2] R. Cleaveland, S. A. Smolka, and S. T. Sims, "An instrumentation-based approach to controller model validation," in *Proc. of ASWSD*, 2008, pp. 84–97.

[3] K. Romer and J. Ma, "PDA: Passive distributed assertions for sensor networks," in *Proc. of IPSN'09*, 2009, pp. 337–348.

[4] J. Mercadal, Q. Enard, C. Consel, and N. Loriant, "A domain-specific approach to architecturing error handling in pervasive computing," in *Proc. of OOPSLA '10*, 2010, pp. 47–61.

[5] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse, "Macrodebugging: global views of distributed program execution," in *Proc. of SenSys*, 2009, pp. 141–154.

[6] J. Weng, P. Cohen, and M. Herniou, "Camera calibration with distortion models and accuracy evaluation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, pp. 965–980, Oct. 1992.