Brace: Assertion-Driven Development of Cyber-Physical Systems Applications

Xi Zheng, Chien-Liang Fok, Christine Julien,
Sarfraz Khurshid, and Miryung Kim

The Center for Advanced Research in Software Engineering
Department of Electrical and Computer Engineering The University of Texas at Austin
jameszhengxi@utexas.edu, liangfok@mail.utexas.edu, c.julien@mail.utexas.edu,
khurshid@ece.utexas.edu, miryung@ece.utexas.edu

TR-ARiSE-2013-001

ARiSE
Advanced Research in Software Engineering

Brace: Assertion-Driven Development of Cyber-Physical Systems Applications

Xi Zheng, Chien-Liang Fok, Christine Julien, Sarfraz Khurshid, and Miryung Kim
The Center for Advanced Research in Software Engineering
Department of Electrical and Computer Engineering The University of Texas at Austin
jameszhengxi@utexas.edu, liangfok@mail.utexas.edu, c.julien@mail.utexas.edu,
khurshid@ece.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 difficult to observe collectively. Developers must repeatedly rerun the system, often in different physical environments, while observing its behavior. The developers then tweak the hardware and software until the entire system appears to meet some minimum requirements. This process is tedious, error-prone, and lacks rigor. In addition, there are always underlying and often unstated assumptions about the physical environment that are subject to variance; these assumptions should be captured early and explicitly in the development process. To address these issues, we present Brace, a framework that allows developers to explicitly specify both physical and logical assumptions and expected behaviors. Brace then enables run-time checking of these combined physical and logical specifications, provided in the form of assertions, using the physical environment in which a CPS application is running. Brace uses physics models and temporal semantics to guide CPS developers in creating appropriate assertions and to check specified assertions for *inconsistencies* with the physical world. This paper presents our initial investigation into the requirements and semantics of such assertions, which we call *cyber-physical assertions*, and the realization of cyber-physical assertions within the Brace framework. We discuss our experience implementing and using Brace with a variety of sensors.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Real-time and embedded systems; D.2.4 [**Software/Program Verification**]: Assertion checkers, Validation

General Terms

Design, Verification

FSE 2013 Saint Petersburg, Russia

Keywords

cyber-physical systems, assertions, debugging

1. INTRODUCTION

A ubiquitous and fundamental challenge in developing cyber-physical systems (CPS) is that such systems inherently integrate both physical and logical components in a way that necessitates jointly verifying, validating, and evolving the complete CPS. Existing state-of-the-art techniques rely on testing in the software and (computer) hardware domains, with methods tailored to validating the *cyber* portions of the system. Such approaches neglect key physical aspects and the *interplay* between the physical and cyber aspects. Traditional approaches that do combine the cyber and physical focus on calibration activities that tailor the behavior of the CPS to a particular operating environment. This is contrary to a more direct debugging approach that focuses on identifying mismatches between an actual physical environment and the developer's assumptions about that environment (whether explicit or implicit). Such a mismatch can even lead to logical errors in the cyber components of the system that cannot be reliably solved through calibration. This disconnect between the logical and physical is especially relevant in many safety-critical CPS, where computational elements must correctly assess and manipulate the state of physical elements in a verifiable manner.

In cyber-physical systems, design, implementation, and testing often partially or completely neglect concrete models of time and physics that have real impact on CPS application performance. Consider a simple CPS application deployed in a smart home. When an occupant wants to watch a movie, the smart home must make the environment dark, so it turns off all of the lights in the room. The smart home assumes that the room is dark and commences playback of the movie. An obvious logical programming error is highlighted by this oversimplified example: *turning off the lights in a room does not always make the room dark.* During the daytime, it may be necessary to also close the blinds. In creating CPS applications, system developers rely on ad hoc implementation and testing methods: we take a quick crack at implementing a system, throw it in a target environment, observe its behavior, and refine it by fine-tuning its behavior. Such an approach is neither reliable nor robust, and it is also not easily generalizable or extensible.

As a second example, consider the safe control of an autonomous vehicle tasked with transporting payloads from one location to another in a specific amount of time. Based

on observations in the development of the vehicle, a CPS designer may assume that a given amount of power to the motor for a specified time interval causes the vehicle to move a specific distance. However, if anything changes about the environment (e.g., the coefficient of friction of the road surface, the incline of the surface, or even the charging level of the battery), this assumption can fail. If the entire application is written around a specific set of environmental assumptions, the entire application can fail. In this case, the application was fine-tuned to a specific operating environment instead of debugged in the context of specifications of correct behavior of both the logical and physical components. In this paper, in contrast, we motivate a debugging process in which the correct behavior of physical components can be provided by well-understood models of physics.

Temporal aspects of cyber-physical systems present yet another challenge. Cyber-physical systems inherently include actuation, or actions that have real impact on a physical world, and actuation takes time. Consider a CPS in which a robotically controlled arm flips a light switch off. Debugging the CPS that flips the switch could very reasonably include checking to ensure that, after the instruction was issued to the arm, the room became darker. Automated debugging techniques generally assume that, when a line of code is executed, the impact of that execution is effectively instantaneous, and therefore a debugging technique can immediately check whether the expected result occurred. However, verifying that the robotic arm successfully turned off the light requires giving the arm time to do its job. It is not immediately obvious how to automate checking that such an actuation was successful without a clear specification of how that success relates to the passage of real time.

These examples highlight several aspects novel to cyber-physical systems that makes developing them, and, more to the point, debugging them, challenging. It is impossible to exhaustively test the complete system for all possible physical states of a target environment. Further, the physical environment may change over time, causing an application that worked at one time to later fail, simply because the new environment was not covered in the original testing. Simply stated, existing CPS development and debugging approaches are insufficient because they cannot ensure the correctness of a program that actuates on a system or environment in a way that considers physical ramifications.

To address these challenges, we introduce *Brace*, a runtime verification framework targeted to cyber-physical systems. Brace uses assertions over both the cyber and physical properties of an implemented CPS to check its correctness and to provide debugging information to the developer. Brace assertions can incorporate concrete models of the physical world (e.g., from a simple understanding of the impact of actuators on ambient properties of a smart home to a detailed kinematic model expressing the expected operation of an autonomous vehicle). Brace enables integration of physical models with a run-time debugging process. This paper dramatically extends our previous report on Brace [3], which introduced the motivation behind Brace, gave a set of toy examples, and experimented with a rudimentary prototype consisting of hard-coded assertion-checking

This paper includes the following components:

- **Assertions for CPS.** We introduce the idea of using assertions that relate physical properties, sensed by physical sensing devices to values of logical variables as a basis of developing robust cyber-physical systems.
- **The Brace Framework.** We present a framework for assertion-driven development of cyber-physical systems; this framework provides a novel integration of the physical and the logical worlds.
- **Implementation.** We present a generic and flexible implementation of Brace, which is based on AspectJ and the Java Concurrent library.
- **Evaluation.** We describe experimental evaluations using small representative CPS applications to demonstrate the usefulness and applicability of our framework as well to show the importance of supporting temporal semantics for CPS development.

As opposed to traditional uses of assertions, our assertion-driven approach for CPS development uses physics-based models and temporal semantics to guide creation of assertions, to identify inconsistent or infeasible assertions, and to localize potential causes for CPS failures and monitor cyber-physical assertions by their temporal attributes. In this paper, we describe, demonstrate, and evaluate the Brace tools for effectively debugging cyber-physical systems.

2. MOTIVATION AND 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 [7]. Assertion schemes have been extended beyond the domain of single-threaded, relatively deterministic programs to handle the inherent non-deterministic interleavings of increasingly common multi-threaded programs [4]. STROBE [1] supports asynchronous assertion checking for single- and multi-threaded Java programs using copy-on-write semantics to incrementally copy state. JUnit [2], a popular framework for testing Java programs, provides a number of assert methods for easier specification of equality comparisons. A similar approach for approximate equality comparisons was introduced in the Java modeling language [14]. These assertions are evaluated at runtime to assess the correctness of a program's parallelism by checking whether different interleavings lead to different results. We have similar challenges with respect to parallelism, but assertions for cyber-physical systems must also allow programmers to refer to the physical environment and assess the relationships between the physical and logical at runtime.

Attempts to rigorously validate complex cyber-physical systems also exist. Instrument-based validation attaches monitors to controller models during simulation [8]. Our motivation requires checking runtime assertions within an actual system as opposed to in simulation and checking assertions that span time and multiple nodes. Passive distributed assertions [23] allow developers to add assertions about a global network state; each node broadcasts small amounts of information to a secondary "sniffer network," which analyzes this data to determine an assertion's validity. We aim to allow users to explicitly map logical variables to physical states using a configuration utility, making it easier to write CPS assertions that are also more expressive. Mercadal et al. [17] use a domain-specific Architecture Description Language (ADL) to safely handle application- and system-level errors in pervasive computing; our initial focus is instead on identifying and reacting to unexpected buggy behavior by using an external sensor network and monitor-

ing process instead of trying to recover from severe errors or influence the flow of the original application.

Debugging approaches that connect the logical and physical have been explored in wireless sensor networks. Marionette [27] provides rudimentary debugging for sensor networks using remote procedure call. Sympathy [21] collects metrics about the sensor network performance and uses these metrics to detect and debug failures. DustMiner [12] identifies potentially faulty sequences of events based on execution traces. Such traces can direct programmers where to look for logical errors that lead to interactive bugs. Envirolog [16] also provides logging capabilities for wireless sensor networks; in Envirolog traces can be replayed, aiding in repeating buggy executions and identifying faults. Declarative Tracepoints [5] allows the developer to insert "action-associated checkpoints" using a declarative language. These tracepoints enable automating the debugging process. Clairvoyant is a source-level debugger for wireless sensor networks [28] that connects the developer to the WSN remotely and then employs fairly standard debugging commands at the source-level. KleeNet [24], on the other hand, is an offline symbolic execution tool that for wireless sensor networks. When KleeNet detects a bug, it automatically generates a test case that demonstrates the bug. Macrodebugging [25] addresses the debugging of macroprograms, which specify the behavior of an entire wireless sensor network as a unit; individual nodes' programs are then automatically generated. In macrodebugging, developers can set breakpoints in the macroprogram and then investigate the traces in a post-mortem fashion. In the domain of cyber-physical systems, we favor live interaction with the program as part of the debugging process; this is because the physical environment is an inherent part of the system, and the physical environment simply cannot be completely replicated.

Part of our motivation is to identify inconsistencies that cross the boundaries of a system's logical and physical components. Similarly, FIND [9] identifies nodes reporting faulty data so that a developer can localize a node contributing to inconsistencies in reported aggregates. NodeMD [13] was developed to identify nodes before failures completely disable them; by transitioning the node into a diagnostic state, NodeMD identifies and repairs node failures. Also related are techniques for automatic calibration [26], whose objective is to fine-tune program parameters to make an observed value match a desired value according to a user-defined model. In contrast, our aim is to simplify the process of defining and checking system properties at runtime.

The addition of the CPS perspective provides a fundamental shift from this related work. The fundamental difference between our motivation and the above (which is largely targeted to wireless sensor networks) is that the CPS developer is interested in asserting properties over both physical and logical states, and this intricate association with the physical world requires dynamic analysis approaches, motivating the design of our cyber-physical assertion. At the same time, significant distribution and resource constraints demand new semantics for cyber-physical assertion evaluation.

From a practical perspective, we rely on Aspect Oriented Programming to support our implementation. The Monitoring Oriented Programming framework (MOP) [6] uses a similar strategy to provide runtime monitoring in which monitors are automatically synthesized from specified properties and are used to check a system's dynamic behaviors. When a specification is violated or validated at runtime, user-defined actions are triggered. To use MOP in Java, a developer must use a set of logical plugins (e.g., plugins for various temporal logics) to specify properties; these logics are not intuitive and their connection to the program is not often obvious to a developer. JavaMOP employs AspectJ in its implementation; in Brace we build directly on top of AspectJ with an optimized concurrency thread manager that hides all the specification, temporal logics, and concurrent programming complexity from CPS developers. In Brace, we are motivated to completely hide the runtime monitors and present only interfaces for developers to specify cyber-physical assertions and connect them to salient physical aspects of the environment.

Instead of validating CPS application states and critical conditions on an ad-hoc basis, developers should explicitly specify the underlying assumptions connecting the physical to the logical. Given this motivation and the state-of-the-art, we have identified the following key gaps in supporting expressive and effective debugging of CPS:

- **Physical aspects should be considered during development and debugging.** Current techniques to verifying cyber-physical systems implementation incorporate physical aspects in only an ad hoc and not very repeatable way. A key challenge in enabling CPS development is to make consideration of the interplay between the physical and logical explicit. *Brace achieves this by defining an assertion language and evaluation framework that allows developers to explicitly reference expectations of the physical ramifications of the system.*
- **Existing debugging techniques targeting traditional systems do not consider the temporal requirements of real actuation.** Directly adapting existing assertion evaluation approaches fails to consider the fact that actuation takes real time, while executing code takes (almost) no time. *Brace uses inspiration from distributed debugging to consider joint evaluation of physical and logical properties with temporal dependencies and over long periods of time.*

3. THE BRACE FRAMEWORK FOR ASSERTION-DRIVEN DEVELOPMENT

A cyber-physical system inherently combines a logical program that runs within a physical space that can include a variety of sensing and actuation capabilities. The Brace framework directly targets the challenges identified in the previous section by annotating CPS programs with *cyber-physical assertions*. These assertions allow a developer to constrain the appropriate behavior of a CPS application to be dependent on the logical and physical states jointly.

Using Figure 1 as a guide, we first overview the Brace framework. CPS developers create programs that specify not only logical function but also physical behaviors in the form of sensing and actuating on an environment. This target environment is shown at the far right of Figure 1. When a CPS application is running in the context of Brace, it is augmented with a Brace *debugging environment*, which may include additional sensing and actuating capabilities. These latter capabilities are specific to debugging and are not expected to be available in a real deployment.

Using Brace, a CPS developer can annotate his program by defining cyber-physical assertions. These assertions are

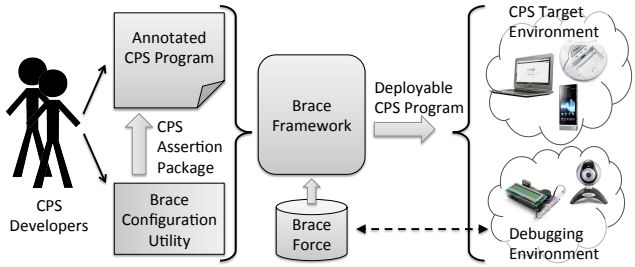

Figure 1: The Brace Framework Architecture

specified through the Brace configuration interface, and the end result is an annotated CPS program. Given an annotated CPS program, Brace connects the annotations to a real physical deployment environment through a provided mapping library we call Brace Force. Brace Force encapsulates laws of physics and connects these rules to the physical capabilities of the combined target CPS environment and debugging environment. During the debugging process, it is the combined environment that Brace assertions are evaluated over. In a real deployment, when the debugging environment is not present, the cyber-physical assertions are effectively turned off to generate a deployed CPS program.

Consider a set of motivating examples drawn from our experiences in the Pharos mobile computing testbed [18]. The testbed is built around a set of autonomous mobile vehicles with three different mobility platforms; the three platforms are shown in Figure 2 and include an iRobot Create with differential steering, a Segway RMP50 based on Segway's self-balancing products, and a Traxxas Stampede remote control car chassis with Ackerman steering. Pharos is designed to abstract away mobility; for this reason, all three mobility platforms are controlled via a single abstraction provided by the Robot Operating System (ROS) [20]. In Pharos, we build cyber-physical applications that operate on top of these mobility platforms and interact with the physical environment. The simplest program issues statically defined movement commands to the mobility platform. This can be extended to a program that, for examples, senses the distance traveled by the platform and issues movement commands based on measured progress. This can obviously be extended with additional sensors and actuators to a program that entails complex interaction with the environment.

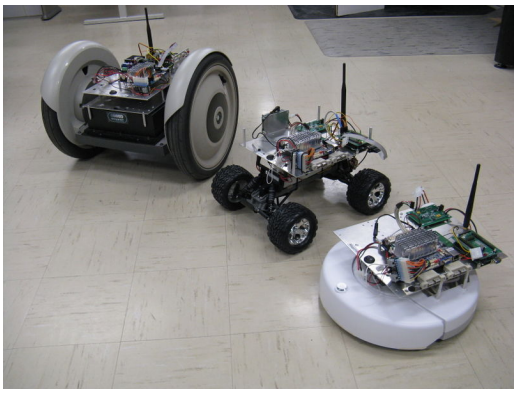

Figure 2: The Pharos Autonomous vehicles

In the context of Brace, the autonomous vehicle is a physical entity whose properties (e.g., location and speed) are captured within the Brace framework and represented as constructs that can be accessed in cyber-physical assertions. In the debugging environment, capturing these physical properties is accomplished via a set of sensors that can monitor the designated properties; for example in our testing environment we can employ an overhead camera that can track the movement of the autonomous vehicle.

The cyber-physical program in this example is a program written in Java that executes on top of ROS. The developer uses the provided movement libraries to specify the expected behavior. The developer's movement interface implies that the results of movement instructions are deterministic: one can instruct the robot to move a given distance at a specified speed or turn a given angle. The correctness with which the robot performs a movement, however, may depend on a variety of factors (e.g., even for a single mobility platform, differences in surfaces' coefficients of friction can cause drastic differences in movements and turns). Brace assertions allow the developer to explicitly capture the expectations of the program's behavior, even when that behavior is dependent on or requires interaction with a physical environment. For example, a developer may construct a cyber-physical assertion, placed after a movement instruction, that asserts that the robot has turned a specified degree or was able to move a specified distance in a specified amount of time. By connecting physical aspects of the environment to the logical world, Brace can check these assertions, allowing debugging that jointly considers the logical and the physical.

For purposes of exposition, the above assertions over a robot's movement are simple. More complex assertions may capture constraints that cross multiple entities. For instance, imagine an application that uses multiple robots to patrol a specified area. A cyber-physical assertion in this context may specify a required measure of *spatial coverage* that the robots must maintain for the target space.

In the remainder of this section, we provide details of each of the components depicted in Figure 1. We first show how cyber assets are linked to physical attributes and how cyber-physical assertions are tightly bound to laws of physics. We then concretely define the classes of cyber-physical assertions provided in Brace and give their temporal semantics. We then detail the architecture of the Brace framework and overview its implementation, including the Brace configuration interface and Brace Force components.

3.1 Mapping the Physical to the Logical

A cyber-physical system will be evaluated in a debugging environment that contains a variety of types of sensors that can be used to assess the system's physical aspects while providing different levels of precision. Even just assessing the robot's location, for example, can be done in myriad different ways given different available debugging environments, for example using ultrasound [19] or RFID [11, 15] indoor positioning systems or GPS outdoors.

As a developer annotates a CPS program with assertions that reference both physical and logical aspects of the system, Brace relies on the sensing capabilities of the target debugging environment to present that environment's validation capabilities. By connecting the logical CPS program with the physical state of the environment, these assertions capture the developer's expectations with respect to the expected impact of the logical instructions (e.g., a move instruction) on physical properties (e.g., the location of the

robot). The language of assertions also allows the assertions to capture the influence of external physics (e.g., friction) on the possible physical states.

To accomplish this, Brace introduces two data types: a *Physical Value* (PV) and a *Mapped Logical Variable* (MLV). In Brace, a PV encapsulates the value of some physical variable (e.g., the robot's location or the brightness of a room). It is coupled with a range that quantifies the margin of error with which the value is knowable. This accounts for the fact that it may not be possible to measure the precise value of a physical phenomenon with complete accuracy. Further, physical values of the same type generated by different sensors may have different associated ranges.

A Brace MLV, on the other hand, is a logical variable that is mapped to some selected physical value. It is an internal (logical) representation of the program's belief of a PV. At some level, a MLV is a proxy to one or more sensors that provide the actual value of the physical variable. It provides an accessor method (`public PhysicalValue getValue()`) that returns the value of the mapped PV. A key element of Brace is that it hides the implementation details of this accessor from the developer, separating the essence of assertions and their use to debug a CPS program from the specific environment (and composite sensors) that support that debugging.

3.2 Capturing Models of Physics

A fundamental goal of Brace is to explicitly relate assertions, which capture a developer's expectations of the behavior of a CPS program with the models of physics that govern the world in which that program executes. Such models play many roles in Brace. They can help define *tolerances* on expected behaviors, and they are essential in identifying physical properties' inter-relationships, especially in safety-critical applications. For example, in an autonomous vehicle-based CPS, physical models can guide specifying assertions that the vehicle does not side-slip or roll over. This implies that sensor information must be combined with these models of physics to inform assertion evaluation.

In addition, it is easy to write impossible assertions. That is, a developer can assert a behavior that is unachievable given a particular operating environment or set of devices. As a very simple example, it may be impossible to increase the brightness of a particular room beyond a certain level. Likewise certain movement or turning patterns may be unreasonable for a given autonomous vehicle because of nonholonomic properties. Using models of physics, combined with specifications of the available sensors and actuators as part of the CPS deployment, Brace ultimately aims to guide developers in writing and evaluating assertions using hints and directives about assertions' potential validities.

In Brace, a developer can associate a physical model with each CPS assertion; it is the developer's responsibility to at least identify what rules of the physical world govern reasonable behavior. A physics model consists of a set of *definitions*. A definition contains references to multiple mapped logical variables (MLVs) that in turn ultimately map to designated sensors available in a particular deployment or debugging environment (as described above).

When specifying a definition, a developer provides an abstraction of physical properties. For example, to define a speed based on underlying MLVs for distance and time, a developer can create the following definition:

$$\mathsf{DEFINE:\ Speed} = \mathsf{Distance}_{\mathrm{MLV}} \ / \ \mathsf{Time}_{\mathrm{MLV}}.$$

This definition assumes MLVs for both Distance and Time and that they are relatable to each other. This can be filled in given a sophisticated runtime monitor capable of monitoring and computing distances over time. Alternatively, when distance itself cannot be monitored and only locations can, a developer can create the following defintion:

$$\mathsf{DEFINE:\ Speed\ (t1,\ t2)} = (\mathsf{Location}_{\mathrm{MLV}}(\mathsf{t2}) \ominus \mathsf{Location}_{\mathrm{MLV}}(\mathsf{t1})) \ / \ (\mathsf{t2 - t1}),$$

where $\ominus$ applies the distance formula, and t1 and t2 are placeholders for two ends of a time window of measurement.

A physics model can also contain *relations*, which define constraints on absolute values that physical variables can have or constraints on relationships between one or more physical variables. Relations can include one or more MLVs and can use relational operators (e.g., $=$, $\leq$, etc.).

Referring back to the autonomous vehicle example, it is possible that physics simply does not allow vehicles to move faster than some specified speed. If that maximum speed is 200 units, this constraint can be provided as part of the physics model by specifying the following relation:

$$\mathsf{RELATE:\ Speed} \leq 200$$

Such a relation may also be dependent on the particular characteristics of the deployment environment, including the availability of particular devices in that environment. For example, the following relates the maximum speed of a vehicle to the capabilities of that vehicle:

$$\mathsf{RELATE:\ Speed} \leq \mathsf{Robot}_{\mathrm{SPEC}}.\mathsf{Speed}$$

The following relation limits the maximum brightness of a room based on both lighting devices and windows:

$$\mathsf{RELATE:\ Brightness} \leq \lceil \mathsf{LightBulb}_{\mathrm{SPEC}}.\mathsf{Brightness},\ \mathsf{Window}_{\mathrm{SPEC}}.\mathsf{Brightness} \rceil$$

These definitions and relations are not themselves cyber-physical assertions. Instead they capture truth; they are facts about the physical world that are independent of the action of the CPS and are not related to a developer's *expectations* of the behavior of the combination of the system's physical and logical components. These expectations are instead captured by *cyber-physical assertions*, which we define next. Cyber-physical assertions can refer to the physics definitions, and, more importantly, the physics definitions can guide the creation of cyber-physical assertions, highlighting impossible expectations and directing reasonable ones.

3.3 Semantics of Cyber-Physical Assertions

Like other assertions, a programmer inserts cyber-physical assertions at points in the logical program (i.e., the code) to declare that a certain presumption is true at that point. Unlike other assertions, a cyber-physical assertion's embodied presumption can consider both logical and physical attributes, that is, cyber-physical assertions enable developers to explicitly cross the boundary between the physical and logical. The Brace framework for cyber-physical assertions also shields the developer from detailed low-level concerns that arise in evaluating assertions over physical attributes, necessitated by the interaction with sensors that measure the physical world. Including physical attributes in assertions brings with it several new concerns that arise from physical constraints. For example, code-level instructions

(e.g., move) that entail an interaction with or actuation on the physical environment almost inevitably experience a latency between the issuance of the instruction and the completion (or even initiation) of the physical action. Further, measuring aspects of the physical environment is almost always imprecise, and the degree of precision is also dependent on the quality of the sensors used. For these reasons, the semantics of cyber-physical assertions are more complicated than traditional assertions. In this section, we present the semantics of the three classes of cyber-physical assertions in Brace; because of the importance of time in CPS, each class differs primarily along the temporal dimension.

3.3.1 The Immediate Assertion

The immediate assertion type provides Brace's closest analog to the semantics of a traditional assertion. An immediate assertion is evaluated synchronously with the program, i.e., the application thread that executes the assertion pauses the cyber-physical program until the assertion is completely evaluated and determined to be either true of false. An assertion specifies logical constraints on physical (and logical) values; Brace allows a variety of comparison operators to be used in specifying assertions. As a nod to the physical environment, there is one additional component to an immediate assertion in Brace: a *tolerance* value that allows some wiggle room in assessing the physical value.

The ability to mimic the semantics of a traditional assertion is essential in Brace. However, the immediate assertion also has several drawbacks and limitations. First, depending on the types of sensors included in the underlying debugging environment and connected in through the physics model, evaluating the assertion may incur significant latency. For example, an assertion may require interacting with a distributed set of sensors that may be duty cycled (i.e., allowed to periodically sleep to conserve energy). This latency has an impact on the calling thread from which the assertion originates, delaying its future actions. Thus, the behavior of the application in the presence of the assertions may differ from the behavior of the application in the absence of any assertions. In addition, physical values read from sensors used to evaluate the assertion may be out of date with respect to when the developer intended the assertion to be triggered, especially if significant latency is incurred in communicating with the sensors. Finally, as expected, an immediate assertion is evaluated immediately, but this immediacy may not allow time for actuation to happen, causing assertions that really should evaluate to true (i.e., the robot is *about to be* moving) instead evaluate to false.

The limitations associated with Brace's immediate assertions are fundamental to systems that consider physical properties, but they are also fundamentally new in comparison to existing assertion capabilities. To account differently for these limitations in CPS programs, we introduce two additional types of cyber-physical assertions.

Example. We use the following immediate assertion in our evaluation (in Section 4) to assert that the ambient sound level is greater than a specified decibel level (within a specified tolerance):

$$\text{ASSERT IMMEDIATE: Sound} > 10 \pm 0.1$$

Applicability. Brace evaluates these assertions immediately and returns the result directly to the application, in line with the running program.

Limitations. When used with sensors that incur delays, the calling thread is delayed, potentially changing its semantics. When used to validate actuation behavior, immediate assertions may not be able to capture real physical delays.

3.3.2 The Asynchronous Assertion

To mask the latency incurred by immediate assertions, Brace introduces asynchronous assertions, in which the program can continue to execute while the assertion is evaluated in the background. By allowing the program to continue running, the application's responsiveness (as perceived by a user) is less impacted. The semantics of asynchronous assertions are, however, complicated by the fact that they are not evaluated instantaneously. Asynchronous assertions are particularly useful when the data used to evaluate the assertion must be collected from multiple devices distributed in the cyber-physical network since the network communication required to collect the needed values only increases the delay of assertion checking. Executing an asynchronous assertion allows the cyber-physical program to continue, but if the assertion is deemed to be falsified, the host program can be notified by a user-defined response action (i.e., a callback) attached to the assertion. One consequence is that the program is likely to execute beyond where the assertion occurred, making fault localization more difficult since the fault has potentially had time to propagate.

A second motivation for the asynchronous assertion in Brace arose from an observation about cyber-physical systems behavior, namely that there is physical latency associated with actuation commands. That is, physical actuation is not immediately connected to the logical instructions that trigger it. Further there are inherent latencies with respect to an action being performed and that action having the desired effect. Take a simple example: a smart home application may have an instruction that sets the home's thermostat to 68 degrees Fahrenheit. Immediately following this instruction with an assertion that asserts that the temperature in the home is 68 degrees is obviously unreasonable since it takes the home's heating and cooling system some time to effect a temperature adjustment. The nature of this delay is common across many actuations, but the length of the delay is highly variable and depends on the physical variable, the particular actuators used, and the sensors used.

To address these issues, Brace's asynchronous assertions can specify a *delay* parameter that indicates how long the system should wait before attempting to evaluate the assertion. In addition, asynchronous assertions can also associate a *time-to-live window* with each assertion. The semantics of the assertion evaluation are such that, if the assertion is triggered at time t, it must evaluate to true in the interval $[t_0+delay, t_0+delay+window]$. Finally, Brace also allows the assertion to specify the number of attempts that its expression should be evaluated and the interval of time between these evaluations. An assertion fails if the majority of these attempts fail. This latter parameter may appear to be at a level of detail that is too fine for the assertion programmer, but it allows control over the (potentially high) overhead associated with evaluating an assertion repeatedly.

A subtle complication related to evaluating asynchronous assertions rests in determining what data is used to evaluate the assertion. Consider a simple example assertion that has to collect and then average location estimates for an autonomous vehicle from multiple distributed location sensors.

In an immediate assertion, the entire system is assumed to halt, the assertion is checked (in a distributed way), and then the system resumes (or halts, if the assertion fails). In an asynchronous assertion, the assertion is triggered and evaluated in the background by the runtime monitor (described in the next section), while the system continues. The triggered assertion executing in the runtime monitor collects the location estimates from the multiple distributed location sensors. Assume that the assertion was triggered at time t_0 and that, by the time the assertion thread gets results for two of the location sensors, the location readings are associated with times $t_0 + \Delta_1$ and $t_0 + \Delta_2$. These two values are associated not only with a different time than that of the assertion but also with different times relative to each other. Either of these differences may impact the correctness of the assertion evaluation, depending on the intent of the assertion. As a result, the program may experience a false negative (or a false positive) in assertion reporting.

Example. The following example assertion, used in our evaluation in Section 4, extends the immediate assertion above. It asynchronously checks whether the sound is above a threshold (within a tolerance) after a delay of 500ms. It is checked three times in the following second, ending in the invalid callback if the assertion is falsified a majority of the times and in the valid callback otherwise.

ASSERT ASYNC: Sound $> 10 \pm 0.1$
delay = 500, window = 1000, attempts = 3
validCallback = SoundValid()
invalidCallback = SoundInvalid()

Applicability. Brace evaluates these assertions asynchronously and returns the result to the calling application through a callback function. Asynchronous assertions can capture physical values with high accuracy while allowing for tolerance in the temporal evaluation of the assertion.

Limitations. Tuning appropriate values of *delay*, *window*, and *attempts* may be specific to particular physical values, and setting parameters innapropriately may result in lost accuracy or large unnecessary latencies. Asynchronous assertions must also deal with discrepancies in the time stamps associated with the data used to evaluate them.

3.3.3 The Continuous Assertion

It is not uncommon in CPS applications to have conditions that ought to remain true throughout an extended portion of lifetime of the system. In Brace, a continuous assertion is one that spans a period of time over which the assertion is periodically checked. Such assertions may span multiple nodes, locations, and time frames. These assertions effectively define program invariants because they are required to be continuously evaluated over the program's execution, instead of executed singularly at a particular point in the application's code. If the execution of the system could be discretized into a sequence of actions, these assertions would be evaluated after every action. In reality, Brace uses a real time multi-threaded monitor to continuously evaluate these assertions without interfering with the application's execution. In an autonomous robot application, a continuous assertion may be used to monitor wither the robot remains within a certain region of space. In a smart home scenario, a continuous assertion may declare that the temperature of a room must never be above or below specified thresholds. The semantics of continuous assertions are even more complex than asynchronous assertions. We must consider the overhead of the frequency of evaluating continuous assertions. Clearly the overhead of evaluating them constantly (e.g., after every instruction) is often unacceptable. However, evaluating them too infrequently can lead to inadequate knowledge in the conditions that cause assertions to fail (or it may cause Brace to miss many cases where the assertion is in fact falsified). Therefore continuous assertions also specify the frequency with which they are evaluated, putting the control over these potential inconsistencies in the hands of the application domain expert.

Example. The following example continuous assertion is slightly different from the asynchronous assertion above in that, as a continuous assertion, it simply checks the sound value once every second:

ASSERT CONT: Sound $> 10 \pm 0.1$
interval = 1000
validCallback = SoundValid()
invalidCallback = SoundInvalid()

Applicability. Brace evaluates these assertions continuously, and because they effectively inherit from asynchronous assertions, they can capture physical values with high accuracy. Continuous assertions are essential for monitoring system invariants over the lifetime of the CPS.

Limitations. Like asynchronous assertions, the values used in evaluating a continuous assertion may be distanced in time from the determination of a false instance, so tracing the actual failure condition may be non-trivial. Also, tuning the parameter (i.e., *interval*) may be dependent on the particular physical variable and/or sensing capabilities.

3.4 Implementing the Brace Framework

Brace provides a set of Java APIs that enable CPS developers to define cyber-physical assertions. As one of our goals is to minimize the learning curve for CPS developers to specify these assertions, we encapsulate the entire Brace framework in a library and provide this intuitive utility to enable developers to specify assertions abstractly. Starting with the developer on the far left of Figure 1, before the developer creates an annotated CPS application, he first interacts with the *Brace configuration utility* to specify his cyber-physical assertions. In Brace's current implementation, developers create cyber-physical assertions using an XML schema that encapsulates all of the properties and parameters described in the previous section; in the future, directly providing the XML can be replaced with an even more intuitive (and concise) representation that can be automatically mapped into a more machine readable format. Each assertion has a name, which is used by the developers use to insert assertions into the CPS program. The Brace configuration utility automatically generates a cyber-physical assertion package containing Java class files that associate each assertion with a public static Java method that can be invoked in the CPS program using a one-line Java statement.

In the case of immediate assertions, these one-line calls in the annotated program are simply (blocking) method calls; in the case of asynchronous and continuous assertions, a Brace *runtime monitor*, which is built on top of *AspectJ*, is used to store these assertions in a lock-free concurrent linked list [10]. During runtime execution, the runtime monitor assigns threads to evaluate the assertions in this list. The assertion's implementation includes the callback function, which is invoked upon completion of the assertion evaluation. If the assertion is continuous, it is placed back in the

list for subsequent evaluation.

The above process requires the definition of each of the components depicted in Figure 1[1]. The developer defines the assertion using the Brace configuration utility. Then the physical capabilities of the target environment must be considered by mapping the physical variables (PVs) that can be sensed to mapped logical variables (MLVs). This needs to be done once for any target environment and is also currently accomplished through an XML schema. Third, rules governing physics must be specified as rules of the form shown in Section 3.2. Conceptually, Brace provides a universe of rules that connect high-level variables to mapped logical variables that may or may not be available in a particular environment. Connecting the available PVs to the MLVs defines the capabilities of the environment and constrains the allowable cyber-physical assertions. That is, only MLVs that can be supported by available PVs can be referenced in cyber-physical assertions used in the CPS program.

4. EVALUATION

In this section, we evaluate the Brace framework in two ways. First, the *fidelity* of the framework is evaluated by demonstrating the consistency with which the cyber-physical assertions are able to track the actual physical state of the world. We also empirically ascertain the impact of cyber-physical assertions on the application behavior itself. In all cases, we use a simple ambient intelligence application that mimics a simple home automation application in which sound, light, and temperature controls are available to a CPS application developer.

Test Environment. Our testing environment consists of a series of sensors and actuators that enable interaction with and sensing of the ambient environment. Figure 3 shows the hardware configuration we used for our mock home automation environment. Not depicted are the sound control (which is built into the laptop) and the temperature control, which was done with the home's thermostat.

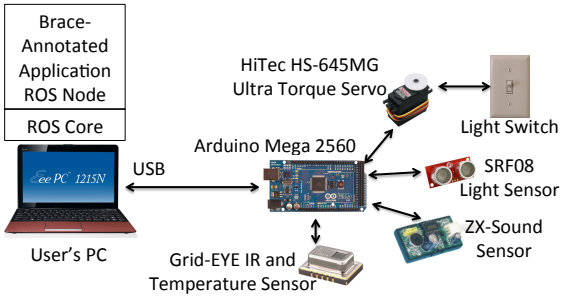

Figure 3: Test Environment Configuration

The laptop used was an Intel i5-2410 2.3GHz quad core with 3.9G of memory and running Ubuntu 12.04 and Java version 1.6. The sound is generated on the laptop at varying decibel levels using the Java Sound Library. This is connected to an Arduino Mega 2560 board, which provides a bridge to the sensing and actuating devices. For the sound sensor, we use a ZX-Sound sensor, for the light sensor, we use an SRF08 ranger/light sensor, and for temperature sensing, we use a Panasonic Grid-EYE infrared array sensor. The light sensor measures the ambient light level at 10Hz and sends the measurements through the Arduino to the laptop. To control the light levels, we use a servo attached to a light switch that controls a set of fluorescent ceiling lights.

Depending on the scenario, our smart home application may generate sound, control the light switch, or both. In addition, in different evaluations it is annotated with different immediate, asynchronous, and continuous assertions with a variety of parameters detailed within each experiment description. Alongside the CPS application, the test laptop runs Brace Force to connect the cyber-physical assertions to the physical environment and the Brace framework to monitor and evaluate the active cyber-physical assertions.

Brace Framework Fidelity. Our first tests evaluate the *fidelity* of the Brace framework. That is, the goal is to evaluate the consistency with which Brace assertions are able to reflect the ground truth of the physical environment. In many ways, this is more an evaluation of our selected sensors than it is of the Brace framework itself, but it demonstrates the capabilities of the framework within a reasonable environment with off-the-shelf monitoring capabilities that one could reasonably expect to have in debugging a CPS application for home automation. We first examine the ability of immediate and asynchronous assertions to validate a single actuation action. Specifically, we created a program that issues a command to increase the ambient decibel level to 4.5 (the decibel level is initially near 0). Immediately after this call, we assert that the ambient decibel level is greater than a certain threshold. We show results for fifteen different forms of this assertion: for five different threshold volumes, we specify one each of an immediate assertion, an asynchronous assertion with a one second delay and a single attempt, and an asynchronous assertion with a one second delay and 3 attempts, each separated by a half second.

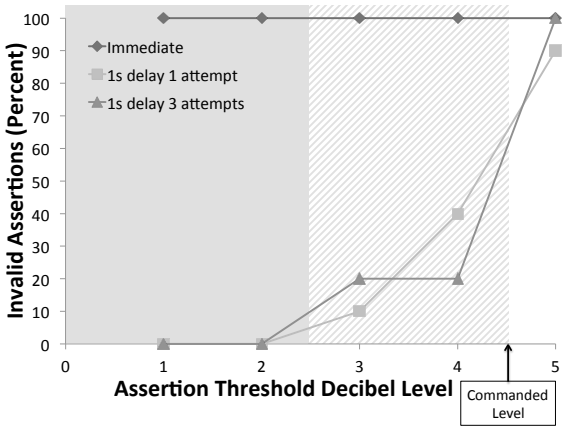

Figure 4: Sound Fidelity Test

Figure 4 shows the results of executing these three different assertions with the five threshold decibel levels. Each point plots the percentage of times out of 20 invocations that the evaluated assertion was false. The gray areas depict the region where the decibel level in the assertion's expression was below the commanded level (the assertion should correctly evaluate to true), and the white area is where the expression's decibel level was above the commanded level (the assertion should correctly evaluate to false). The immediate assertion *never* evaluated to true. This is because the immediate assertion directly follows the sound actuation in the logical code, which does not allow time for the

[1]The detailed XML specifications are omitted for brevity. They can be found at `http://mpc.ece.utexas.edu/brace`

physical actuation to happen. This demonstrates one of the fundamental motivations of Brace: the need for an assertion language that explicitly considers the temporal nature of real CPS applications. For the asynchronous assertions, in the hashed area, we *expect* that each assertion should evaluate to true, but there was inconsistency in Brace's performance. More reliable sensors (or multi-modal sensing that combines readings for the same physical value from multiple types of sensors) would decrease the width of this hashed area, providing an even higher quality of assertion correctness. In addition, different settings for the asynchronous assertion can help mitigate these concerns, for example providing longer delays, longer windows, or more attempts.

To drill down with respect to the impact of actuation delay on Brace's ability to successfully evaluate assertions, we take an example using the light switch and light sensor. In this experiment, our CPS application starts with the light on, turns it off, waits four seconds, then turns it back on. To understand the latency in the actuator and the sensor, we executed this snippet of our applications forty times. Figure 5 shows an actual trace of how the room's ambient light level changes throughout a single execution of the application. The program issues a command to turn off the lights at time $t = 1$, but the room's brightness does not actually dim until time $t = 1 + T_{off}$. Likewise, the program issues a command to turn on the lights at time $t = 5$, but the room does not actually brighten until time t = $5 + T_{on}$. On average across our forty trials, $T_{off} = 386.60 \pm 11.21$ms and $T_{on} = 492.65 \pm 11.00$ms, where the ranges denote 95% confidence intervals. The difference between T_{off} and T_{on} reflect a possible bias in the way the servo is attached to the light switch and the fact that fluorescent lights take time to turn on due to the use of a ballast. These concerns reflect the practical challenges that exist in debugging cyber-physical systems in comparison to debugging traditional programs.

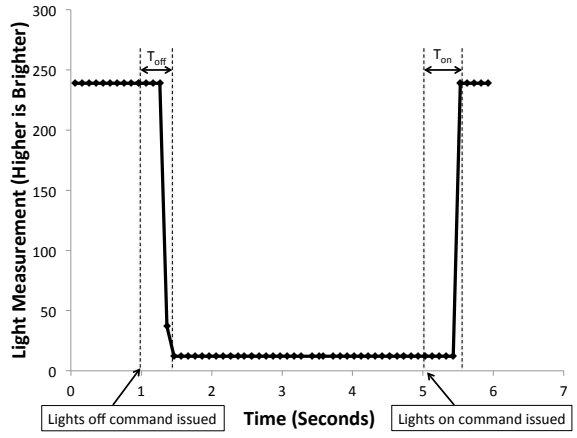

Figure 5: Brightness during application execution

From the application's prospective, issuing the actuating commands on the light switch is relatively quick. Over 40 rounds (80 actuation commands), the average software (i.e., cyber) latency is only 0.13±0.09ms, which is negligible relative to T_{off} and T_{on}. This difference between cyber and physical latencies demonstrates the necessity of using asynchronous cyber-physical assertions with delay windows that account for actuation and sensing latencies.

A Continuous Assertion. To demonstrate the use of a continuous assertion, we describe two examples: one provides assertions over the light level in the room, and the other provides assertions over the temperature in the room. In the first case, we used three different continuous assertions that assert that, continuously, the light level in the room should not fall below a specified level of lumens. On the actuation side, our program then alternates the light on and off every 120 seconds. While we evaluated a variety of settings for such a continuous setting, we include the trace of a single execution, shown in Figure 6, which is for a continuous assertion with a 500 millisecond delay window.

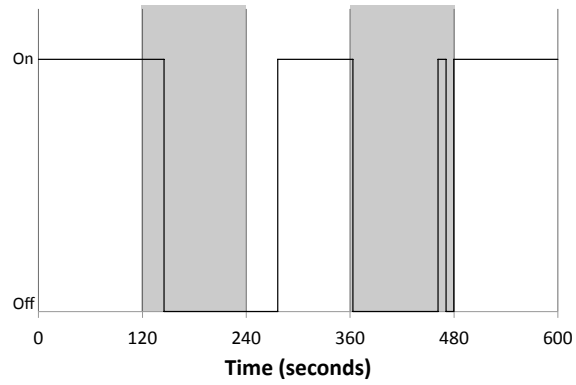

Figure 6: A Continuous Assertion

The white areas represent times when the light was on, and the gray areas represent times when the light was off. For clarity, the line tracks a representation of the assertion's belief about the state of the light (either "on" (high) or "off" (low)); in this case, "off" corresponds to cases when the assertion was triggered. The continuous assertion tracks the value of the physical variable decently well, modulo a delay that comes from a delay in sensing and actuating. In addition, an erroneous evaluation at approximately 470 seconds into the experiment indicates an instance when the assertion was *not* triggered, even though the light was off.

Our second continuous assertion experiment used an assertion to monitor the ambient temperature of the environment over a longer period of time. We changed the thermostat's setting every five minutes and used a pair of assertions to validate that the room's temperature remained within a specified range for the duration of the test. We tested under a variety of temperature settings and assertion ranges. A plot of the results is removed for brevity, but these continuous assertions were also able to track the physical variable well. We did find in this case that it was more effective to check the physical value over a longer time window and with multiple attempts; this is because the temperature value is slow to change, and the rate of change is unpredictable across instances. This demonstrates that CPS applications require the debugging flexibility that Brace provides in the delay, window, attempts, and interval parameters included in the asynchronous and continuous assertions.

The Impact of Instrumentation. An important aspect of Brace is its ability to separate its behavior from that of the CPS application. There is a danger of the overhead of evaluating Brace assertions changing the behavior of the application itself. This is dangerous because, when the developer is ready to deploy the application, he will remove the Brace annotations from his program and expect the behavior to remain unchanged. To begin to ascertain the potential for Brace's instrumentation of the CPS application to interfere

with the application, we evaluate the quality of assertion evaluation (the only way we have to assess the program's correctness) under increasing assertion loads.

Specifically, consider the simple light control snippet of our home automation application. Using the Brace configuration utility, we generated cyber-physical assertions with random selections of assertion types (immediate, asynchronous, or continuous), random selections of assertion thresholds for the light value, and random selection of the assertion parameters (delay, window, attempts, and interval). We remove the impact of actuation latency on assertion evaluation by inserting a software delay between the actuation action and the assertion trigger. Figure 7 shows the results.

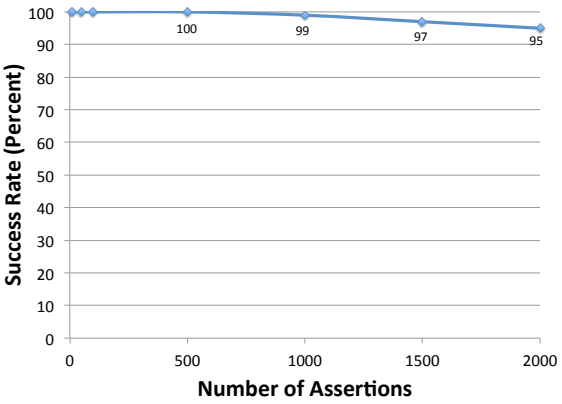

Figure 7: The Impact of Instrumentation

The figure shows the percentage of the assertions that evaluated correctly. Incorrect evaluations consist of both incorrect results (the assertion should have been triggered but was not and vice versa) and assertions that time out (i.e., those that fail to be scheduled before the expiration of their specified delay). As the number of assertions grows, the number of failures increases slightly. This is of course dependent on the hardware configuration, but even for large numbers of assertions on a common-place laptop configuration, Brace performs with high dependency, lending credence to the claim that evaluating Brace assertions has little impact on the execution of the CPS application under test.

5. DISCUSSION AND FUTURE WORK

To further explore the connection of cyber-physical assertions with physics models, we developed a physics model for the motion of an iRobot Create robot, given its specifications. We outfitted the robot with a motion sensor and implemented a small controller instrumented with a cyber-physical assertion that checks if the wheel speed is 0.5m/s. As the robot moves, the assertion holds true when the robot moves on a smooth surface but fails when the robot moves on carpet. This experiment demonstrates how cyber-physical assertions are tightly bound to the physics models; ignoring the models (i.e., in this case the surface coefficient of friction) can lead to faults in CPS applications.

This motivates inclusion of physics rules in Brace as described in Section 3.2. We envision that cyber-physical assertions that incorporate even more complex physics models will be applied to real-world CPS applications, including and especially mission-critical ones. Consider the domain of autonomous vehicles. Some existing high-end passenger cars already use cameras, radars, lasers, and GPS for automating various operations, such as blind-spot monitoring, lane detection, adaptive cruise control, and parallel parking. To facilitate the development and validation of such CPS applications, we envision support for representing and monitoring more complex physics models, say even to track how the behavior of adjacent vehicles may influence the safety of the host vehicle. A simple example assertion can constrain the speed of the host vehicle to be no more than the one ahead to maintain a safe distance. We can create the following physics model coupled with the subsequent assertion:

DEFINE: MySpeed = $\text{MyDistance}_{\text{MLV}}$ / Time_{MLV}
DEFINE: FrontCarSpeed = $\text{FrontCarDistance}_{\text{MLV}}$ / Time_{MLV}
ASSERT CONT: MySpeed $\leq$ FrontCarSpeed

Future work will also explore the potential use of physics models as part of the development process. We should be able to allow the developer to write some code, write an assertion, and have the physics model communicate, at compile time, that the assertion is an unreasonable expectation given the physical laws of the target environment.

Another topic we will explore is the use of physics models to improve the quality of sensor data acquisition and reduce its cost, which can be significant in many CPS domains. Achieving extended lifetime for these energy-constrained devices is non-trivial, where communication with the sensors causes the biggest energy drain but is required to acquire sensor readings. Recent work [22] has motivated using *model-driven* data acquisition for predicting trends of sensed data and incurring sensing cost only when readings deviate from the expected trends. We can adopt a similar approach in Brace. This will be especially useful in evaluating long-lived continuous assertions where the model underlying physical phenomenon is predictable or does not change rapidly.

6. CONCLUSION

As CPS applications become increasingly prevalent and sophisticated, new tools are needed to simplify and make more robust their development. In developing CPS applications, there are a few key challenges related to temporal conditions, physics models, and unchecked presumptions about the physical entites and the environment that must be considered during the debugging process. Furthermore, existing techniques are ad hoc and treat logical and physical states disjointly. We introduced Brace, a framework that provides an interface for defining cyber-physical assertions and runtime support for evaluating them. Using cyber-physical assertions, developers can assert the expected relationships between logical variables and physical properties, capture critical assumptions in the form of cyber-physical assertions, and closely bind those assertions with temporal conditions. We presented a prototype implementation of Brace and a small evaluation that demonstrates the fidelity of Brace's assertion evaluation and the impact of assertions on the behavior of an instrumented CPS application.

7. ACKNOWLEDGMENTS

This work was supported in part by the NSF under grants CCF-1149391, CCF-1117902, CCF-1043810, SHF-0910818, and CCF-0811524 and Microsoft SEIF award. Any opinions, findings and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring parties.

8. REFERENCES

[1] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *Proc. of OOPSLA)*, pages 275–288, 2011.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[3] K. Boos, C.-L. Fok, C. Julien, and M. Kim. Brace: An assertion framework for debugging cyber-physical systems. In *Proc. of ICSE (NIER)*, pages 1341–1344, 2012.

[4] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Proc. of ESEC/FSE*, pages 3–12, 2009.

[5] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In *Proc. of SenSys*, pages 85–98, 2008.

[6] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *Proc. of ICFEM*, volume 3308 of *LNCS*, pages 357 – 373. Springer-Verlag, 2004.

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

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

[9] S. Guo, Z. Zhong, and T. He. FIND: Faulty node detection for wireless sensor networks. In *Proc. of SenSys*, pages 253–266, 2009.

[10] J.D.Valois. Lock-free linked lists using compare-and-swap. In *Proc. of PODC*, pages 214–222. ACM Press, 1995.

[11] G.-Y. Jin, X.-Y. Lu, and M.-S. Park. An indoor localization mechanism using active RFID tag. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, nand Trustworthy Computing*, 2006.

[12] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proc. of SenSys*, pages 99–112, 2008.

[13] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proc. of MobiSys*, pages 43–56, 2007.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, June 1998.

[15] X. Liu, M. Corner, and P. Shenoy. Ferret: RFID localization for pervasive multimedia. In *Proc. of Ubicomp*, pages 422–440, 2006.

[16] L. Luo, T. He, G. Zhou, T. Abdelzaher, and J. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog. In *Proc. of INFOCOM*, 2006.

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

[18] Pharos: The pervasive computing test bed. `http://mpc.ece.utexas.edu/pharos`, 2011.

[19] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket location-support system. In *Proc. of MobiCom*, 2000.

[20] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: An open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[21] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. of SenSys*, pages 255–267, 2005.

[22] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. What does model-driven data acquisition really achieve in wireless sensor networks. In *Proc. of Percom)*, pages 85–94, 2012.

[23] K. Romer and J. Ma. PDA: Passive distributed assertions for sensor networks. In *Proc. of IPSN*, pages 337–348, 2009.

[24] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the* 9^{th} *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 186–196, 2010.

[25] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global views of distributed program execution. In *Proc. of SenSys*, 2009.

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

[27] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using RPC for interactive development and debugging of wireless embedded networks. In *Proc. of IPSN*, 2006.

[28] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proc. of SenSys*, 2007.