Copyright
by
Xi Zheng
2015

The Dissertation Committee for Xi Zheng
certifies that this is the approved version of the following dissertation:

Physically Informed Runtime Verification for Cyber Physical Systems

Committee:

Christine Julien, Supervisor

Dewayne Perry

Miryung Kim

Raul Longoria

Sarfraz Khurshid

Physically Informed Runtime Verification for Cyber Physical Systems

by

Xi Zheng, B.COMP.INFO.MGMT.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2015

To Dr. Julien for her consistent support and guiding me through the most difficult journey of my life.

To Dr. Kim, Dr. Perry, Dr. Khurshid, and Dr. Longoria for putting me through the best education possible.

To Mable, Charles, and my extended family for their understanding and sacrifice.

To all my collaborators for helping me complete those challenging projects.

Physically Informed Runtime Verification for Cyber Physical Systems

Publication No. ____________________

Xi Zheng, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Christine Julien

Cyber-physical systems (CPS) are an integration of computation with physical processes. CPS have gained popularity both in industry and the research community and are represented by many varied mission critical applications. Debugging CPS is important, but the intertwining of the cyber and physical worlds makes it very difficult. Formal methods, simulation, and testing are not sufficient in guarantee required correctness. Runtime Verification (RV) provides a perfect complement. However the state of the art in RV lacks either efficiency or expressiveness, and very few RV technologies are specifically designed for CPS. The CPS community requires an intuitive, expressive, and practical RV middleware toolset to improve the state of the art.

In this proposal, I take an incremental and realistic approach to identify and address the research challenges in CPS verification and validation. Firstly, I carry out a systematic analysis of the state of the art and state of the practice in verifying and validating CPS using a structured on-line survey, semi-structured

interviews, and an exhaustive literature review. From the findings obtained, I identify the key research gaps and propose research directions to address these research gaps. My second work is to work on the most pertinent research direction proposed, which is to provide a practical and physically informed runtime verification tool-sets specifically designed for CPS as a sound foundation to the trial and error practice identified as the state of the art in verifying and validating CPS. I create an expressive yet intuitive language (*BraceAssertion*) to specify CPS properties. I develop a framework (*BraceBind*) to supplement CPS runtime verification with a real time simulation environment which is able to integrate physical models from various simulation platform. Based on *BraceAssertion* and *BraceBind*, which collectively captures the combination of logical content and physical environment, I develop a practical runtime verification framework (*Brace*), which is efficient, effective, expressive in capturing both local and global properties, and guarantee predictable runtime monitors behavior even with unpredictable surge of events. I evaluate the tool-set with increasingly complex real CPS applications of smart agent systems.

Table of Contents

Abstract . . . **v**

Chapter 1. Introduction . . . **1**

1.1 Intended Contributions . . . 5

1.2 Overview . . . 6

Chapter 2. On the State of the Art in Verification and Validation in Cyber Physical Systems . . . **8**

2.1 Introduction . . . 9

2.2 Methodology . . . 12

2.3 Literature Review . . . 16

2.4 The on-line survey . . . 22

2.4.1 Background and Definitions . . . 22

2.4.2 Perceptions . . . 26

2.4.3 Experiences . . . 28

2.5 Interview . . . 34

2.6 Future Research Directions . . . 41

2.7 Threats to Validity . . . 43

2.8 Research Contributions . . . 45

2.9 Chapter Summary . . . 45

Chapter 3. BraceForce: A Middleware to Integrate Sensing in CPS Applications . . . **47**

3.1 Introduction . . . 48

3.2 Related work . . . 52

3.3 BraceForce . . . 56

3.3.1 BraceForce Abstract Architecture . . . 56

3.3.1.1 BraceForce Layered Architecture. . . . 57

3.3.1.2 BraceForce Deployment Scenarios. . . . 61

3.3.2 BraceForce Implementation 64
3.3.2.1 Android Programming Idiosyncrasies 64
3.3.2.2 Unified Data and Subscription Interfaces 65
3.3.2.3 Thread Management and Android services 66
3.3.2.4 Networking 67
3.3.2.5 Sensor Driver Definition and Discovery 68
3.3.2.6 Model Driven Data Acquisition 70
3.4 Empiricial Design 71
3.5 Results and Discussion 75
3.6 Research Contributions 82
3.7 Chapter Summary 82

Chapter 4. BraceAssertion: Behavior Driven Development for CPS Application 84
4.1 Introduction 85
4.2 Motivation and Overview 87
4.2.1 Motivating Application 87
4.2.2 Behavior Driven Development 89
4.2.3 My Basic Formal Framework 90
4.2.4 ECA and SCL 91
4.3 BraceAssertion 93
4.3.1 BraceAssertion Language 94
4.3.2 BraceAssertion Formal Semantics 98
4.4 Dual Monitor Architecture 98
4.4.1 Event Monitor 101
4.4.2 ECA Monitor 103
4.4.3 Combinatorial Analysis 104
4.5 Case Study and Evaluation 107
4.5.1 The Case Study Briefly 107
4.5.2 Research Questions (RQs) 108
4.5.3 Experiment Design 108
4.5.4 RQ1: The Efficiency of the Event Monitor 111
4.5.5 RQ2: The Effectiveness of BraceAssertion 113

4.5.6 RQ3: ECA Monitor Efficiency and Unique Features 114
4.5.7 Threats to Validity 116
4.6 Related Work 117
4.7 Research Contributions 118
4.8 Chapter Summary 119

Chapter 5. BraceBind: Combining Real-Time Simulation with Runtime Verification for Cyber Physical Systems 120

5.1 Introduction 121
5.2 Motivation and Related Work 124
5.3 Interface Specification for Physical Models and the Cyber 128
5.3.1 Model Interface Specification 129
5.3.2 Annotations for physical aspects 131
5.4 BraceBind 132
5.4.1 BraceBind Architecture Overview 133
5.4.2 Model Transformation 135
5.4.3 Data Integration 136
5.4.4 Time Synchronization 138
5.4.5 Master Synchronizer 140
5.4.6 The Case Study 145
5.4.7 The Case Study 145
5.4.8 Research Questions (RQs) 146
5.4.9 Experiment Design 147
5.4.10 BraceBind VS Simulink (E1) 149
5.4.11 BraceBind VS Physical Deployment (E2) 149
5.4.12 BraceBind VS Physical Deployment in Runtime Verification (E3) 153
5.4.13 Discussion 154
5.5 Research Contributions 157
5.6 Chapter Summary 157

Chapter 6. Brace: A Middleware for Practical On-Line Monitoring of Cyber-Phyiscal System Correctness **159**
6.1 Introduction 160
6.2 The Formal Specification: BraceAssertion 163
6.3 The Brace Middleware 169
6.3.1 Local Optimization Controller 171
6.3.2 Communication Agent 174
6.3.3 Event Synchronizer 178
6.3.3.1 Synchronizing the Events of a Single Node 179
6.3.3.2 Synchronizing Events across Multiple Nodes 180
6.3.3.3 Synchronizing and Aggregating Events and Attributes 182
6.3.4 Global Property Monitors and their Automata 184
6.4 Case Study and Evaluation 188
6.4.1 The Case Study 188
6.4.2 Android Test Bed 191
6.4.3 Orbit Test Bed 192
6.4.4 Validity discussion 197
6.5 Related Work 198
6.6 Research Contributions 201
6.7 Chapter Summary 201

Chapter 7. Conclusion **203**

Appendix **205**

Appendix A. BraceAssertion **206**
A.1 SCL Syntax and Semantics 206
A.2 BraceAssertion Syntax (in BNF) 208
A.3 BraceAssertion Semantics 209
A.4 Algorithms in BraceAssertion 211
A.4.1 AspectJ Instrumentation 211
A.4.2 Monitor Synthesis (Main) 212
A.4.3 Event Monitor Synthesis 214
A.4.4 ECA Monitor Synthesis 214
A.5 The Case Study - Full Version 215

Bibliography **218**

Chapter 1

Introduction

Cyber-Physical Systems (CPS) are gaining popularity and are widely used in biomedical and healthcare systems, autonomous vehicles, large scale real time rescue, smart grid and renewable energy, and many other industrial applications [50, 113, 156, 183].

A ubiquitous and fundamental challenge in developing 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. Traditional approaches (e.g., Formal Methods and Testing) focus mainly 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. The state of the art approaches (e.g., model-based approaches [32, 68, 122], and debugging tools in sensor networks [101, 163, 166, 176, 188]) 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 proposal, in contrast, I 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.

The tenuous connections between the debugging tasks and sensor platforms also pose significant challenge for verifying and validating CPS. It is difficult (and at times impossible) to incorporate physical sensor readings, which are often not

reliable, into debugging rules and tools. This issue is further complicated by the heterogeneous nature of sensor platforms; different platforms require their own languages and operating systems making it challenging to add sensor information to the debugging task in a general purpose way.

As for now, CPS developers rely heavily on simulation to safeguard the correctness of CPS. As an example of the limitations of simulation, a vehicle in the 2007 DARPA Urban Challenge dangerously deviated from its computer-generated path and stuttered in the middle of a busy intersection. The bug was ultimately determined to be related to limitations of the steering rate at low speeds, which was undetected by the (sophisticated) simulation model used [133]. While simulation models may provide very good representations of the real world, they often fail to accurately represent the environment; playback of recorded traces in simulation environments suffers from a similar limitation in that it reduces the inherently continuous environment to a discrete one.

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.

1.1 Intended Contributions

This proposal addresses the challenges of verifying and validating CPS by providing a tool-set for CPS runtime verification. Fig. 1.1 shows the components of the tool-set and how these components will support the specification, development, and debugging of CPS applications from a high-level abstraction [192].

More specifically, I will make the following contributions:

1. Conduct a comprehensive investigation on the state of the art in CPS verification and validation through an exhaustive literature review, a structured on-line survey, and interviews [RC1].

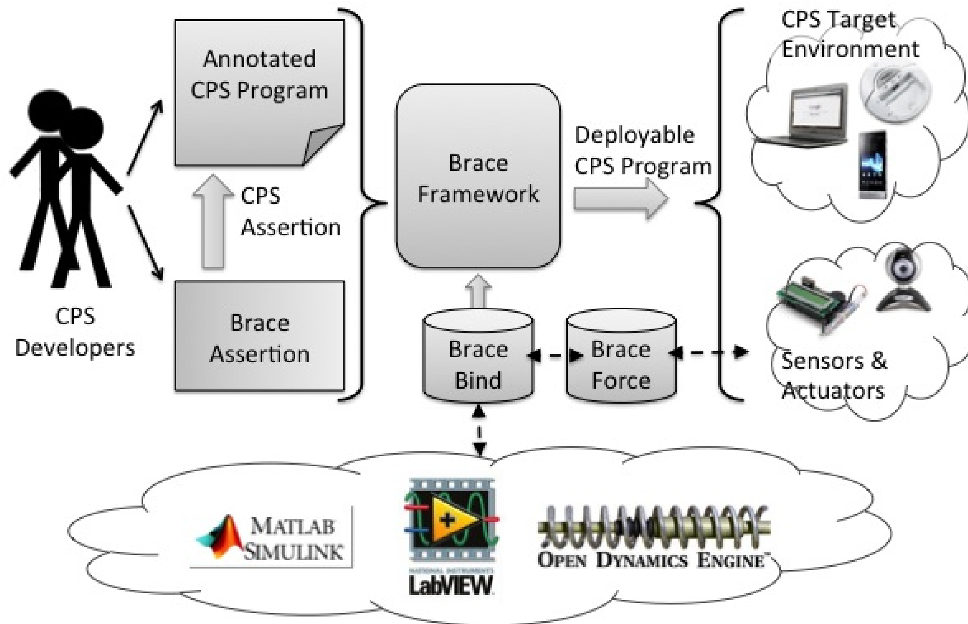

Figure 1.1: The Brace Framework Architecture

2. Create BraceForce as a supporting infrastructure to access sensors from heterogeneous platforms [RC2].

3. Create BraceAssertion as a new intuitive and expressive language to specify CPS properties with timing constraints (qualitative and quantative) and predicate logics [RC3].

4. Create BraceBind as a middleware to enable real time simulation of physical models from heterogeneous simulation platforms to inform CPS runtime verification [RC4].

5. Create Brace as a practical, efficient, and effective online runtime verification middleware specifically designed for CPS applications [RC5].

6. Evaluate the multi-component toolkit using real increasingly complex CPS applications of multi agent systems [RC6].

1.2 Overview

The rest of this proposal is organized as follows. Chapter 2 presents the results of our investigation on the state of the art in CPS verification and validation. Chapter 3 presents *BraceForce* which provides a simplistic and unified programming interface to access sensor data across heterogeneous sensor platforms. *BraceForce* facilitates debugging of CPS applications in various physical test environment with different set of sensors. Chapter 4 presents *BraceAssertion* language which is expressive of metric temporal logics and predicate logics, and intuitive of using natural language to specify CPS properties. *BraceAssertion* is supported with a dual moni-

tor architecture which uses *Event Monitor* to generate filtered and aggregated timed trace, and *Timed Automata Monitor* to detect violations of CPS properties based on the timed trace (offline checking). Chapter 5 presents *BraceBind* which is capable of generating a real time simulation environment that allows real time execution of physical models from different simulation platforms (e.g. Simulink and LabView) and using the results to inform CPS runtime verification. *BraceBind* avoids expensive and often infeasible repeated physical deployment of CPS applications as currently required for CPS debugging and testing (including CPS runtime verification). Chapter 6 presents *Brace* as an online runtime verification middleware that is efficient (in terms of CPU and Memory), effective (in terms of number of false positives and false negatives), expressive in capturing both local and global properties, and as first of the kind to guarantee predictable behavior of runtime monitors even with unpredictable surge of (random) events. I also discuss a real and increasingly complex CPS application used to evaluate *BraceAssertion*, *BraceBind*, and *Brace* in each corresponding chapter. Finally, Chapter 7 concludes.

Chapter 2

On the State of the Art in Verification and Validation in Cyber Physical Systems

It is widely held that debugging cyber-physical systems (CPS) is challenging; many strongly held beliefs exist regarding how CPS are currently debugged and tested and the suitability of various techniques. For instance, dissenting opinions exist as to whether formal methods (including static analysis, theorem proving, and model checking) are appropriate in CPS verification and validation. Simulation tools and simulation-based testing are also often considered insufficient for cyber-physical systems. Many "experts" posit that high-level programming languages (e.g., Java or C#) are not applicable to CPS due to their inability to address (significant) resource constraints at a high level of abstraction. To date, empirical studies investigating these questions have not been done. In this chapter, I qualitatively and quantitatively analyze why debugging cyber-physical systems remains challenging and either dispel or confirm these strongly held beliefs along the way. Specifically, I report on a structured on- line survey of 25 CPS researchers (10 participants classified themselves as CPS developers), semi-structured interviews with 9 practitioners across four continents, and a qualitative literature review. I report these results and discuss several implications for research and practice related to cyber-physical systems.

2.1 Introduction

Cyber-Physical Systems (CPS) feature a tight coupling between physical processes and software components [113] and execute in varying spatial and temporal contexts exhibiting diverse behaviors across runs [183]. CPS are widely used in biomedical and healthcare systems, autonomous vehicles, smart grids, and many industrial applications [113, 156, 183]. Over the years, systems and control engineers have made significant progress in developing system science and engineering methods and tools (e.g., time and frequency domain methods, state space analysis, filtering, prediction, optimization, robust control, and stochastic control) [14]. At the same time, computer science and software engineering researchers have made breakthroughs in software verification and validation (e.g., systematic testing, formal methods). While there are existing well-grounded testing methodologies for other domains of software, and while formal methods have been used for verification of mission-critical systems in practice, verifying and validating CPS are complicated because of the physical aspects and external environment. For instance, there are insufficient methods for investigating the impact of the environment, or *context*, on a CPS [171]. External conditions, which are often hard to predict, can invalidate estimates (even worst-case ones) of the safety and reliability of a system. Modeling any CPS is further hampered by the complexity of modeling *both* the cyber (e.g., software, network, and computing hardware) and the physical (physical processes and their interactions) [121]. Simplified models that do not anticipate that the physical and logical components fail dependently are easily invalidated.

In a 2007 DARPA Urban Challenge Vehicle, a bug undetected by more than

300 miles of test-driving resulted in a near collision. An analysis of the incident found that, to protect the steering system, the interface to the physical hardware limited the steering rate to low speeds [133]. When the path planner produced a sharp turn at higher speeds, the vehicle physically could not follow. The analysis also concluded that, although simulation-centric tools are indispensable for rapid prototyping, design, and debugging, they are limited in providing correctness guarantees. In some mission-critical industries (e.g., medical devices), correctness is currently satisfied by the documentation for code inspections, static analysis, module-level testing, and integration testing [95]. These tests do not consider the context of the patient [95]. Such a lack of true correctness guarantees could easily cause something like the Therac-25 disaster [117] to reoccur.

I seek to address the dearth of empirical information available about CPS development, specifically in debugging and testing. While limited studies of CPS verification and validation exist [58, 112, 165], there is no study that systematically addresses the entire range of existing approaches. In the past decade, as research on CPS has exploded, many strongly held beliefs have emerged related to developing, debugging, and testing these systems. I conduct a broad literature review, a quantitative survey, and qualitative interviews with CPS experts to uncover the state of the art and practice in CPS verification and validation. My surveys and interviews start with basic questions, identifying the technical backgrounds of actual CPS experts. I then move into specifics related to tools and techniques used on a daily basis. I take a broad view, encompassing simulation, formal methods, model-driven development, and more *ad hoc* approaches. I also attempt to ascertain

what aspects of CPS development remain unaddressed *in practice*, with the aim of eliciting a targeted research agenda for software engineers desiring to support the ever-growing domain of CPS development.

Table 2.1: Summary of strongly held beliefs about CPS development

Belief	**Sections**
I. CPS developers are largely untrained in traditional software engineering methodologies. [62, 156]	§2.4.1, §2.5
II. CPS developers are generally unfamiliar with traditional software verification and validation tools and methodologies. [62, 156]	§2.4.1, §2.5
III. High-level programming languages (e.g., Java) are not applicable to CPS. [72, 182]	§2.4.1, §2.5
IV. Resource constraints (e.g., CPU, memory, and storage) are a major issue in developing and debugging CPS. [107, 182, 190]	§2.4.1, §2.5
V. Existing model checking and other formal techniques are insufficient to meet CPS applications' needs. [40, 47, 111, 178]	§2.3, §2.5
VI. Simulation alone is insufficient in supporting verification and validation of CPS. [133]	§2.4.2, §2.4.3, §2.5
VII. An ad hoc, trial-and-error approach to development is the state of the art for CPS systems. [137, 156]	§2.4.2, §2.5
VIII. There is a significant gap in language between formal models of computing and communications and models of physics that makes applying them jointly in CPS challenging. [2, 113, 170]	§2.4.2 §2.4.3, §2.5

To the best of my knowledge, my study is the first to quantitatively assess the state of the art in this area. I start by identifying strongly held beliefs about CPS debugging (Section 2.2) and then review the relevant available literature (Section 2.3). I follow this with detailed results from an on-line survey (Section 2.4) and one-on-one interviews with CPS experts (Section 2.5). I conclude with some future research directions for CPS verification and validation elicited from my investigation (Section 2.6).

2.2 Methodology

Cyber-physical systems are increasingly prevalent, and they pervade many other emerging domains, including pervasive computing in general and the Internet of Things. The rise has been so rapid over the past decade that software engineering support for these new domains has not kept pace. I seek to confirm or dispel several widely held beliefs related to developing CPS, with a specific focus on the verification and validation stages. Table 2.1 documents several of these beliefs, along with relevant references to the literature. The section of the chapter in which I address each belief is listed in the right-most column of Table 2.1. my investigation takes three parts: a broad literature review, a quantitative on-line survey, and qualitative interviews. This combined study benefits from the strengths of each of its parts; while I feel the study methods are rigorous, some threats to validity still exist. I discuss these in Section 2.7. For each of my three methods, before discussing the results, I here briefly describe the goals of the approach, my protocol, and how I analyzed the data.

Literature Review. It is obviously not possible (or desirable) to perform a *complete* literature review of CPS verification and validation in this chapter. Instead I aim to provide a broad look at the variety of techniques and approaches that could be applied to CPS verification and validation. The approaches analyzed in the literature review helped us shape the questions for the on-line survey and interviews.

Protocol. I conducted this review by exploring related publications in the recent past in the areas/categories of static analysis, theorem proving, model checking, run-time verification, simulation based testing, synchronous approaches of real-time

systems testing, model driven development (MDD) based tools, and finally social and cultural impact of verification and validation. All of these reviews were focused through a lens capturing cyber-physical systems and other closely related domains (e.g., hybrid systems and real-time systems). The review reported in this chapter is a refinement of a much broader look that included additional domains (e.g., distributed systems in general, reactive systems, sensor networks, etc.) and a deeper look at specific categories of approaches. The artifacts covered in this chapter serve as (highly referenced) exemplars of the state of the art in verification and validation for CPS.

Data Analysis. For each category in my review, I chose a few representative approaches (selected based on measures of popularity including citations and discussions of practical applications) and provide a short summary (due to the size limitation) of their pros and cons.

On-line Surveys. My survey has two aims. First, I aim to corroborate findings from the literature review by cross-checking them with those CPS researchers with hands-on experiences. Second, I seek to confirm or dispel the strongly held beliefs listed in Table 2.1.

Protocol. Based on the findings from my literature review, I created a set of multiple choice questions that attempt to resolve the veracity of the strongly held beliefs surrounding CPS development and debugging. I also designed a set of open-ended questions motivated to complement the variety of information collected

in the literature review[1].

I sent the invitation of the on-line survey to 82 CPS researchers, who publish work related to real-world CPS development and deployment in relevant academic conferences and received 25 responses. I reached experts from a wide range of subfields, including electrical, mechanical, chemical, and biological engineering and from computer science; 37.5% of them have expertise in control systems and AI, 37.5% of them in networking, 16.7% of them in cyber-security, 16.7% of them in civil engineering, mechanical engineering, or other "traditional" engineering fields, 37.5% in real-time systems, distributed systems, algorithm, verification, testing, and software engineering, in general. When asked about their primary role(s), 70.8% have roles as CPS modeling experts, designers, and architects; 54.2% have roles in validation and verification; and *41.7% classified themselves as CPS developers.* The participants had, on average, 8.35 years of software development experience and 6.69 years of experience in CPS applications.

Data Analysis. I performed statistical analysis on the multiple choice questions. I collated the free-text responses by combining responses that aligned contextually. I use a phenomenological approach [53], which attempts to aggregate meaning from multiple individuals based on their "lived experiences" related to the concept (i.e., phenomenon) under study. my on-line survey (and, in fact, my interviews), are exactly targeting the conclusions I can draw based on studies of the experiences of a group of individuals, in this case, experts in CPS development.

[1]The surveys were delivered via SurveyMonkey; the full text is available at (`https://www.surveymonkey.com/s/MP7HP7W`)

Interviews. To more deeply examine the implications of several of the responses in the survey and corroborate the findings in the survey relative to the strongly held beliefs listed in Table 2.1, I created open-ended questions around the trends I saw in the survey results, to explore further CPS practitioners opinions related to CPS verification and validation. The full questions list is available online[2].

Protocol. I conducted these interviews through personal interactions[3]. The audio of the interviews was recorded with the participants' consent. In the case of in-person interviews, the participants often showed the interviewer documents, papers, devices, and other artifacts that were relevant to the interview questions; this often highlighted the real constraints and limitations or showcased the development and deployment environments. The interviewees were CPS experts in charge of real-world CPS systems from around the world (from North America, Europe, Asia, and Australia). I found the participants through my review of the CPS literature and development tools; the selected interview participants are in charge of the development and deployment of real-world CPS applications development and deployment. I interviewed an expert in *autonomous vehicles*, another in closely related *autonomous robots*, one in *medical CPS*, two in *formal methods*, one in *unmanned aerial vehicles and wireless sensor networks*, one in *assisted living*, one in *wearable devices*, and the last one in *structural health monitoring.*

Data Analysis. I transcribed the interviews and then used the same methodologies as I did for the on-line survey.

[2] `http://goo.gl/5vwvPf`
[3] Two interviews were done over Skype; the remainder were in person

2.3 Literature Review

In my literature review, I focus on breadth of coverage, providing exemplars in the wide variety of applicable areas, including formal methods, model- and simulation-based testing, runtime verification, and multiple practical tools. I also look briefly at social and cultural factors that have a non-trivial impact on the adoption of these techniques.

Formal Methods. Static analysis is used to efficiently compute approximate but *sound* guarantees about the behavior of a program without executing it [60]. Abstract interpretation relates abstract analysis to program execution [51] and can be used to compute invariants [6,16,28,46,59,77,93,132]; these approaches have been applied in CPS, including in avionic and embedded systems [30] and for remote space rovers [60]. In general, the efficiency and quality of static analysis tools have reached a level where they can be practically useful in locating bugs that are otherwise hard to detect via testing. However, for mission-critical CPS applications (which may contain millions of lines of code that interact in complex ways with a physical world), existing tools do not scale well and tend to introduce many false positives [64].

Theorem proving has been applied in deductive verification [114,115], where validity of the verification conditions are determined, and in generating invariants for runtime assertions [92]. Theorem provers have also been used for verifying hybrid systems [1, 29]. Isabelle/HOL [141] has been used to formally verify the kernel piece of seL4 [103], which is the foundation OS for a highly secure military CPS application. This work shows that, with careful design, a (critical component of a)

complex CPS can be formally verified by the state of art theorem prover. However, the requirement for human intervention and high costs (the total effort for proof was about 20 person-years, and kernel changes require 1.5-6 person-years to re-verify [102]) makes applying theorem proving impossible for general-purpose CPS applications, which may contain millions of lines of code [150] and require much quicker (and less expensive) changes.

The verification world is also rife with highly capable model checkers [57,109], including those that handle real-time constraints [181], parametric constraints [83], stochastic effects [89], and asynchronous concurrency [70], all of which are common in CPS. Model abstraction and reduction can make analysis more tractable (and thus more applicable to CPS) [49,78,179], however, error bounds are usually unquantified, which makes the verification unsafe. While model checking allows verification to be fully automated, in addition to issues such as state-explosion, complexity in property specification, and inevitable loss of representativeness [15], CPS exhibit bugs that crop up only at run-time based on the physical state of the deployment world; such bugs cannot be captured by model checking alone. In hybrid systems, online model- checking has received some attention, investigating, for example the potential behavior of a system over some short-term (time-bounded) future. Such approaches have been applied to checking medical device applications [40], where the findings have motivated further investigation into adaptations of model checking targeted for CPS-like domains.

It is exceedingly difficult to prove properties of CPS automatically because of the disconnect between formal techniques for the cyber and well-established engi-

neering techniques for the physical [48,149]. This disconnect is the root of Belief VIII in Table 2.1. Further, the large scale of CPS applications pushes scalability requirements well beyond the capabilities of existing tools. Though significant progress has been made in formal verification that has the potential to change this landscape for CPS, without support from other approaches, including run-time verification (which is much less constrained by scalability issues) [25], formal methods alone are not enough to tackle the challenges in CPS verification and validation [47, 111]. These positions from the literature are the root of Belief V in Table 2.1; my survey and interviews will try to further identify uses and challenges associated with real-world CPS developers applying formal techniques.

Run-Time Verification. In run-time verification, correctness properties specify all admissible executions using extended regular expressions, trace matches, and others formalisms [25]. Temporal logics, especially variants of LTL [152] are popular in runtime verification. However, basic temporal logics do not capture non-functional specifications that are essential in CPS (e.g., timeouts and latency) and lack capabilities to deal with the stochastic nature of many CPS applications [75,153, 164]. In [174], a monitor is created for hybrid systems and a monitorability theorem is provided. However, there is little discussion of whether the monitor will impact the system's functional and non-functional behaviors. In [104], an efficient runtime assertion checking monitor is proposed for memory monitoring of C programs. This non-invasive monitoring is well suited to mission- critical and time-critical CPS applications. In summary, the state of art in run-time verification can potentially provide a great supplement for formal methods and traditional testing in CPS.

However many opportunities remain to make run-time verification more suitable to the idiosyncrasies of CPS and approachable to CPS developers. For instance, aspect-oriented monitoring tools [42] are less intrusive, and their adaptation to CPS run-time verification may prove more approachable for developers.

Model-Based Approaches. In this category, I include model-based testing [180] and model-driven development (MDD), which both use models to automatically generate code. I include simulation tools (e.g., [32, 63, 122]) since they rely on models to evaluate executions of programs. Modeling real-time components has been decomposed into behaviors, their interactions, and priorities on them; reasoning can then occur layer by layer [23, 172]. In general, such approaches allow the verification of all system layers from the correctness proof of the lower layers (i.e., gate-level) to the verification procedure for distributed applications; such an approach has been used to verify automotive systems, a key exemplar of CPS [35]. The practicality and costs of development associated with these approaches are still unknown.

While there are many computational and network simulators that many software engineers may be familiar with, in the CPS domain, the most relevant system is Simulink, which is widely deployed in the automotive industry and other mission critical domains (e.g., avionic applications [81]). As explored further in my on-line survey and interviews, simulation based approaches lack sufficient expressiveness to serve as an end-to-end solution and the truthfulness of the behavior is often in question [112]. When system verification has relied exclusively on simulation, the verification has failed to identify key failure points [133]; in addressing Belief VI from

Table 2.1 in my survey and interviews, I seek to identify situations when real-world CPS developers rely on simulation and when it falls short.

Model-based approaches are gaining momentum, and it seems inevitable that approaches will emerge that can be applied to general purpose CPS. For now, the high learning curve associated with creating the models, the costs of developing them, and scalability remain major hurdles to wide adoption.

Testing and Debugging Tools. Though sensor networks and CPS are not exactly the same, several tools exist to support testing and debugging deployed wireless sensor networks, which provides insight into directions and challenges for CPS. *Passive distributed assertions* [163] allow programmers to specify assertions that are preprocessed to generate instrumented code that passively transmits relevant messages as the assertions are checked. Dustminer [101] collects system logs to look for sequences of events responsible for faulty interactions among sensors. Clairvoyant [188] uses a debugger on each sensor node to instrument the binary code to enable GDB-like debugging behavior. MDB [177] provides the same style of behavior for *macroprograms* specified at the network level (instead of the node level). Envirolog [125] records all events labeled with programmer-provided annotations, allowing an entire execution trace to be replayed. *Declarative tracepoints* allow the programmer to insert checkpoints for specified conditions that occur at runtime. Sympathy [158] collects and analyzes a set of minimal metrics at a centralized sink node to enable fault localization across the distributed nodes. Finally, KleeNet [166] uses symbolic execution to generate distributed execution paths and cover low- probability corner-case situations. In summary, these tools and algo-

rithms can tackle various similar issues in CPS; an immediate effort to adapt them more specifically to CPS would be one that directly accounts for the physical world with continuous dynamics.

Cultural and Social Concerns. To improve the state of art and practice of developing and debugging cyber-physical systems, it is essential to have a robust, scalable and integrated toolset that not only provides accessible approaches for verification and validation but approaches that are also more willingly adopted by practitioners. The major cultural and social impediments include *lack of funding* and *lack of priority* [5], which are not related to technical aspects at all. Apart from commonly known false positives, the reasons developers do not use static analysis tools have been documented as *developers' overload* [97], while *mandate from supervisors*, the *(in)ability to find knowledgeable people*, and *code ownership* all play a role in how bugs are fixed [136].

In many of the approaches I reviewed, even when the CPS developers attempted to provide significant rigor to verification and validation tasks, they almost always had to fall back on a "trial and error" approach, which results in a more *ad hoc* approach to verifying the system [137]. In addressing Belief VII from Table 2.1, my survey and interviews seek to uncover how pervasive trial and error is among real-world CPS developers. In correlation with Belief I, I are also interested in whether real-world CPS developers are classically trained software engineers who are aware of more formal methods or whether they are "outsiders" who simply default to *ad hoc* methods.

2.4 The on-line survey

My on-line survey consisted of 32 multiple choice and free answer questions designed to (1) understand participants' definitions of CPS; (2) determine participants' familiarity with existing techniques for verification and validation, and how they are applied to CPS; and (3) to collect information about the main challenges in verification and validation of CPS, from an "in the trenches" perspective. Table 2.2 shows an abbreviated version of the survey. The specific questions were driven by my literature review, which also helped elicit the strongly held beliefs in Table 2.1. The questions were crafted to help confirm or dispel each of these beliefs. my approach in the survey was *intentional*. I began by generating definitions of both *cyber- physical systems* and of *verification and validation*. I then built on this foundation to determine, in detail, the experts' various approaches to and perceptions of the wide array of CPS verification and validation techniques.

2.4.1 Background and Definitions

Among CPS developers, there are strong opinions about appropriate programming languages. The programming language greatly influences the verification tools and techniques that can be applied; while some techniques apply at the design level and are thus more general purpose, others apply at the language level. The responses to the question "**What programming languages are you familiar with**?" mirror surveys of programming language adoption in general (with C/C++ and Java taking the top spots, and Python a close third). Only one of my respondents was familiar with nesC, the programming language for TinyOS sensor

Table 2.2: Summary of Survey Questions (abbreviated; see https://www.surveymonkey.com/s/MP7HP7W for complete survey)

Background
What are your primary application domains of expertise (multiple choice)?
What are your primary roles (multiple choice)?
How many years of cyber-physical systems development experience do you have?
What programming languages have you used in developing CPS applications? (multiple choice)
Definition
What are the differences between CPS and embedded systems? (free text)
How do you define verification and validation (in general) (free text)?
Perceptions
Please rate agreement or disagreement (Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree) - Simulation alone is sufficient for verification and validation of CPS. - Formal methods for verification and validation of CPS is not tractable with respect to resources and time. - The state of the art of verification and validation of CPS involves repeatedly rerunning the system in a "live" deployment, observing its behavior, and tweaking the implementation (both hardware and software) to achieve the stated requirements.
Experience
What percentage of your work is devoted to verification and validation?
How do you think code inspection can help with verification and validation? (free text)
What testing methodologies do you employ during verification and validation of CPS? (multiple choice)
What model checker(s) do you employ during verification and validation of CPS? (multiple choice)
What simulation tools have you used? (multiple choice)
Have you written assertions to aid in verification and validation of CPS?

network platforms. This is interesting given that many CPS experts purportedly believe that high-level programming languages are not appropriate for cyber-physical

style systems [72, 182].

A subsequent question broached this question directly, when I asked participants to rate their agreement with, "**A programming language like Java is not applicable to systems with hard real-time constraints**." Figure 2.1 shows the results; I were surprised by the implication that many of the surveyed CPS experts found Java to be reasonably appropriate for CPS development (50% of self-classified developers, referred as developers, selected disagree/strong disagree, 30% selected neutral). Consider other Java dialects such as RT-Java or Java Embedded, or Java ME, that are designed specifically for developing real-time or embedded applications, this further counters the colloquial claim expressed as Belief III in Table 2.1 that high-level languages are not appropriate to CPS development, which has a potential rippling effect on future research directions. In my interviews, I found even stronger evidence for these findings (Section 2.5).

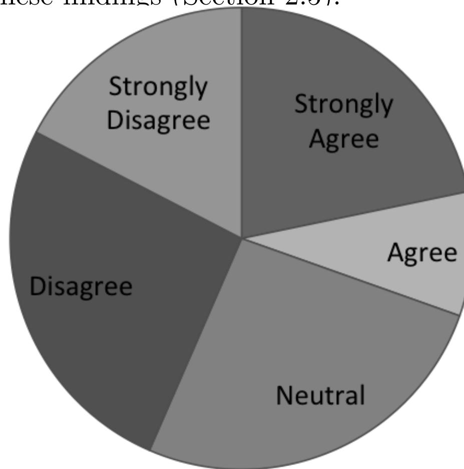

Figure 2.1: "A programming language like Java is not applicable to systems with hard real-time constraints."

I also asked the participants to express definitions of both cyber-physical

systems and verification and validation in their own words. This is important in setting a foundation for the remainder of the survey responses. When I asked, "**In your opinion, what are the main differences between cyber-physical systems and conventional embedded systems**," most respondents' answers identified commonly cited key distinctions; the following responses were typical:

> "*embedded systems were mostly focused on software/hardware interacting with low level sensing and real-time control. CPS includes embedded systems but also networks, security, privacy, cloud computing, and even big data.*"
>
> "*CPSs tend to focus more on the interplay between physical and virtual worlds, and the kind of applications possible with the observation (and modification) of the physical world done through devices embedded in the environment.*"

While the above gets at participants' individual definitions of CPS (which largely converge), I also wanted to understand CPS developers' perspectives on verification and validation. In response to "**How do you define verification and validation**," over half of the participants gave something quite similar to commonly accepted definitions (i.e., that verification establishes how well a software product matches its specification, while validation establishes how well that software product achieves the actual goal [33]). Many other respondents (30% developers with incorrect answers) failed to correctly express the concepts. Intuitively, these results motivate the creation of easy-to-use tools and better education that enable even CPS developers without a rigorous software engineering background to develop robust systems.

2.4.2 Perceptions

One of the primary goals of this survey is to uncover the veracity of the beliefs in Table 2.1. I phrased several of these sometimes controversial points as questions about "perceptions" associated with CPS development. I asked the participants to rate their level of agreement (or disagreement) with the statements using a five-point Likert scale.

It is often stated (and even empirically demonstrated [133]) that simulation does not sufficiently match a system's behavior in the real world. I asked my participants to rate their agreement with "**The use of simulation alone is sufficient for supporting verification and validation of cyber-physical systems.**" Given the variety of backgrounds among my participants, this question has the potential to tease out a potential dichotomy among CPS developers with different backgrounds. In fact, all but one of the survey respondents selected either "Disagree" or "Strongly disagree." The one respondent who selected "Strongly Agree" was also one of the four survey respondents who gave their primary area of expertise as "Civil Engineering/Mechanical Engineering/Other Engineering," where models are more traditionally accepted as complete representations of the system.

Another commonly held belief (Belief V in Table 2.1) is that formal approaches have too high of an overhead to be practically applied in CPS [178]. When I asked the participants to rate "**The use of formal methods for verification and validation of cyber-physical systems is not tractable with respect to resources and time**," the diversity of answers was surprising, as was the apparent support for at least limited use of formal methods for CPS. Figure 2.2 shows the

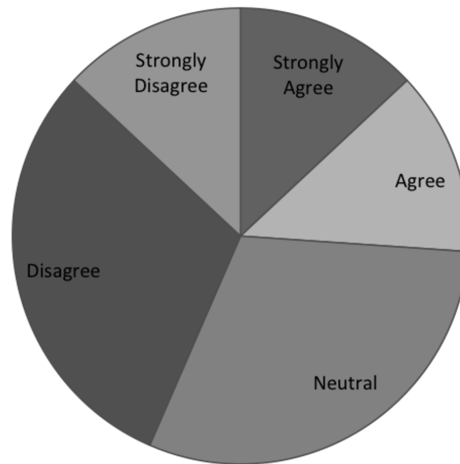

Figure 2.2: "The use of formal methods for verification and validation of cyber-physical systems is not tractable with respect to resources and time."

distribution of responses (40% developers were in favor and 30% selected neutral).

CPS developers will widely claim that the most common approach to debugging CPS requires a significant amount of "trial and error" [137, 156] (Belief VII in Table 2.1). To evaluate this claim, I asked the participants' opinions regarding "**The current state of the art of verification and validation of cyber-physical systems involves repeatedly rerunning the system in a 'live' deployment, observing its behavior, and subsequently tweaking the implementation (both hardware and software) to adjust the system's behavior to achieve the stated requirement.**" 91.3% of the participants expressed either "Strongly Agree or Agree." The current "trial and error" processes are neither rigorous nor repeatable, but the extensive amount of *in situ* debugging that these responses demonstrate motivates better support for approaches to verification and validation that function "in the wild."

My literature review found that approaches to CPS verification and valida-

tion tend to focus either on computational models or on models of physics. Rarely do the two converge. In attempting to address Belief VIII from Table 2.1, the next question in my survey attempted to ascertain whether this is intentional or accidental. I asked the participants to rate their agreement with "**A lack of formal connection to models of physics is a key gap in the verification and validation of cyber-physical systems.**" I found that 69.6% of the respondents (and 60% of the respondents who also self-identified as CPS developers) selected either "Strongly Agree" or "Agree," while 26.1% (30% of the CPS developers) were "Neutral." This is corroborated by a second question, in which I asked the respondents whether they agreed with the statement, "**Since CPS has both cyber and physical parts, any approach for verification and validation would need to allow an engineer to, in some way, examine both parts at the same time**," to which only 27.3% of the respondents selected "Disagree" or "Strongly Disagree." These two results in conjunction indicate a need for more expressive and integrated models that cross the cyber and physical worlds.

2.4.3 Experiences

The third section of my survey queried the participants about their use of verification and validation techniques, most specifically applied to CPS. As Figure 2.3 shows, more than 60% of the participants spent between 30-60% of the system's development time on debugging; more than 20% of the respondents spent *more* time than that. Clearly, debugging CPS is expensive and time consuming.

Only about half of the participants indicated that they employed *code in-*

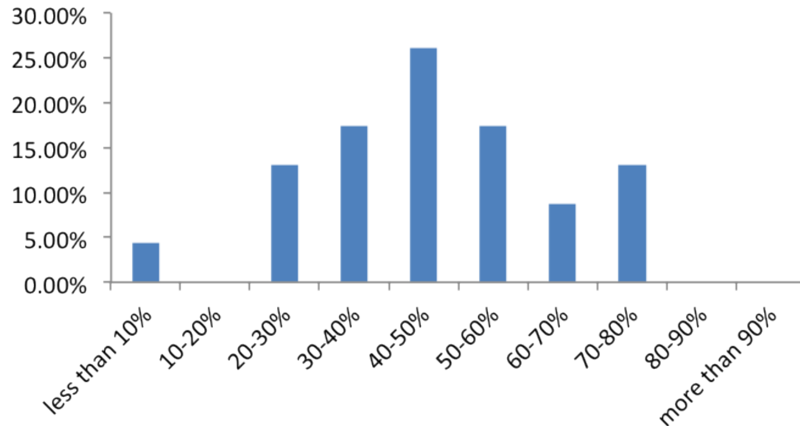

Figure 2.3: Project time spent in debugging

spection. Of the respondents who did not use code inspection, the majority found it to be "not relevant." While code inspection is not universally used, it is believed by some developers to provide an important and useful tool to improving code quality and code understanding, which is known to lead to less error-prone implementations [175]. While this motivates better tool support for code inspection of CPS, it is not a clear significant concern of active CPS developers.

I received an evenly balanced response to "**Have you used systematic testing to aid in verification and validation?**" Participants who responded affirmatively reported improvement of code coverage, systematic review, and identification of corner cases as benefits. Respondents who have not used systematic testing gave standard reasons, including a "*lack of time and deep familiarity*" and "*no easily available tools.*" Generally, CPS developers are not universally familiar with traditional systematic testing tools. Though systematic testing is well established in more general purpose software engineering domains, there are research

challenges in bridging the gap between existing techniques and CPS development.

When I asked "**Have you used formal methods (e.g., model checking) to aid in verification and validation?**" the majority replied affirmatively. When I followed up with the participants who had employed formal methods about the advantages, they cited inferring useful patterns, complete testing, finding corner cases, and verifying key components. Some participants even reported a sense that model checking was becoming increasingly practical for real systems. Those who did not use model checking said that it is (for example):

"*overly complicated for most purposes; most bugs arise from time dependent interactions with physical systems.*"

"*not applicable to my domain; demanding and unreliable.*"

When I asked "**What specific model checker(s) do you employ?**" participants reported high usage of Spin [90] (53.85%), NuSMV [45] (46.15%), and UPPAAL [109] (46.15%)), as shown in Figure 2.4. There was also substantially high use of other (mostly domain-specific) model checkers.

From the free form responses, I noticed that participants gravitate towards general purpose model checkers for very small, very specific pieces of their systems. These model checkers do not enable combined reasoning about the cyber and physical portions of the systems, which is critical to complete and correct verification of CPS.

The responses to "**Have you used simulation to aid in verification and validation?**" were overwhelmingly positive; only one participant said "no." Par-

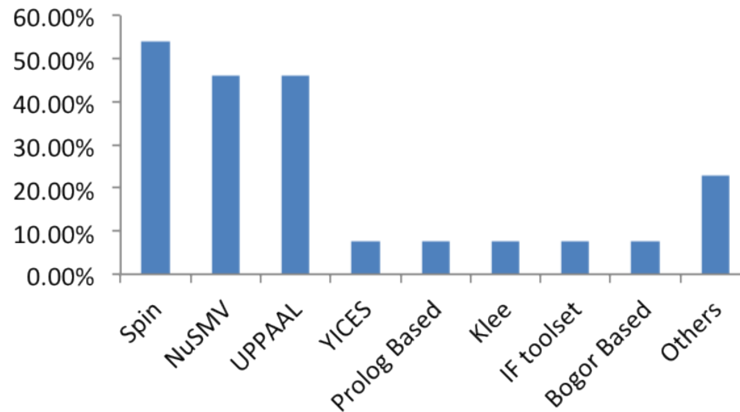

Figure 2.4: Model checkers used

ticipants reported using simulation to understand the system, prototype behavior, refine specifications, explore configurations, and minimize test effort:

"*can provide some preliminary confidence of the system*"

"*can help refine the specification and validate the system*"

"*allows assumptions made in modeling to be cross-validated against another source of ground truth*"

"*helps save and focus testing effort.*"

One participant noted, "*modeling and simulation only goes so far. No one ever found oil by drilling through a map on a table.*" The one participant who did not rely on simulation stated that, "*Good enough simulation does not exist.*" Participants reported high usage of Simulink (61.1%) and proprietary tools (77.8%), as shown in Figure 2.5. The remarkably high use of in-house simulation is concerning because it naturally limits reproducibility and generalizability and implies that

developers find that simulation tools in general are not sufficient.

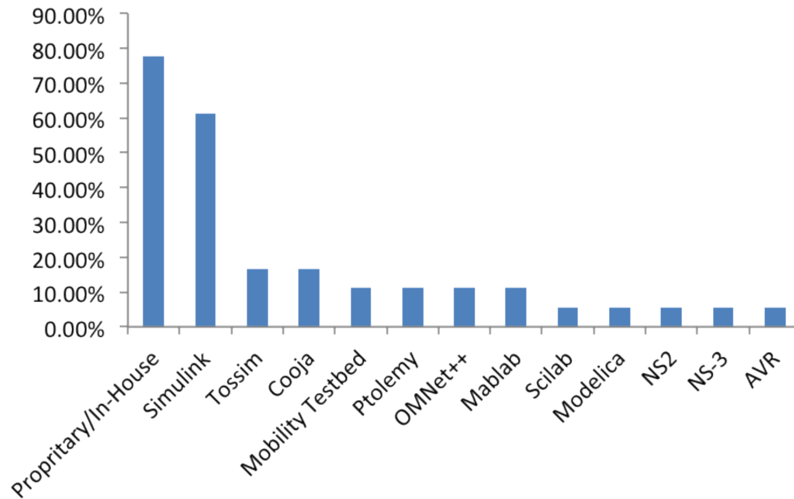

Figure 2.5: Simulation tools used

A common approach to debugging at the source level is to augment a program with assertions that provide checkpoints on the program's state throughout its execution. When I asked the participants about their use of assertions in CPS, the vast majority (more than 70%) had used them. The respondents used assertions primarily for bug detection and writing formal specifications, and, to a slightly lesser extent, to document assumptions. The participants stated that the use of assertions:

> "*[provides] formal documentation [and] explicitly states otherwise implicit assumptions, enabling to detect errors sooner and have a better indication of where a problem stems from*"

> "*forces developer to write down (basically as part of the code) the expectation for correct behavior [with] the bonus of being able to 'execute' the assertion.*"

The two main reasons cited for not having used assertions in CPS development were concerns about performance and the difficulty in tracing assertions in deployed systems. In general, these results bolster my hypothesis that assertions are a useful means to verify and validate CPS; future research that tailors assertions to particular challenges of CPS (e.g., distribution and physical aspects) may ameliorate some of the concerns.

Finally, I asked my participants' perceptions of open challenges in verification and validation of CPS. Almost 50% reported issues with physics models, more than a quarter cited scalability, and nearly a quarter reported issues with a lack of systematic verification and validation methods. Figure 2.6 reports the complete results. Example statements include:

"*CPS models are fragile. I need verification and validation methods that scale with the complexity of the model and are robust to slight variations of the model.*"

"*Time plays a critical role and is misunderstood.*"

"*Impossible to fully understand environment dynamics.*"

The survey results indicate models of software systems, models of physics, and the integration of the two are major bottlenecks in verification and validation of CPS. Scalability of existing techniques and a lack of a capability of directly verifying CPS code from simulation motivate new, tailored approaches that build on and complement the current state of the practice. Before exploring these research challenges, I look at individual interviews with CPS developers.

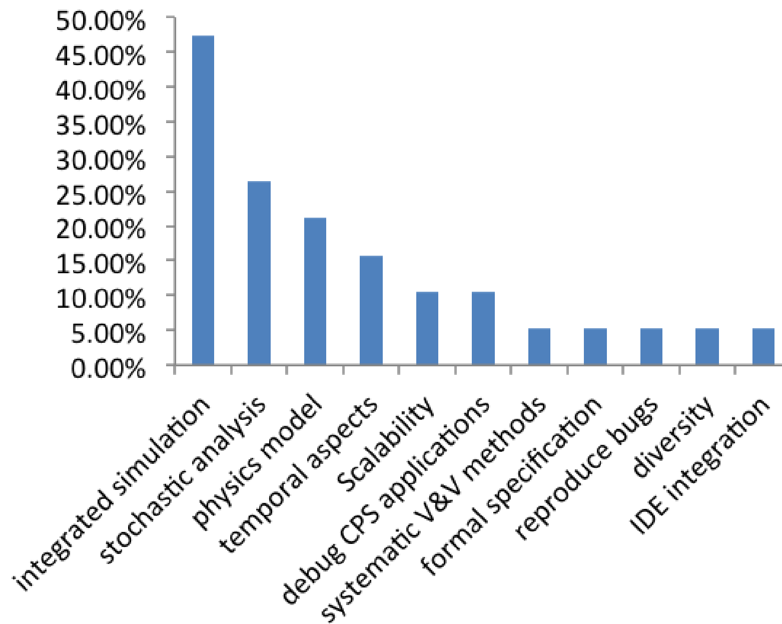

Figure 2.6: Main research challenges

2.5 Interview

The final piece of study is a set of in depth interviews with CPS developers in charge of real-world CPS systems, most of which are mission critical ones (e.g., medical device and bridge structural health monitoring). I was somewhat surprised to find that my survey respondents were not completely against using high-level programming languages like Java to build CPS. The interviews corroborated the survey results and, in fact, highlighted high-level languages that are popular among the CPS developers I interviewed. Specifically, typical responses from my interviewees in response to the discussion question "**What are the main programming languages you used in developing CPS applications?**" were:

"*For the high level, mostly it is Python. Low level is C and C++. In the middle*

is Java."

"For wireless sensor networks, mainly C, nesC; for aerial drones, mainly C++, Java. I also extended C++, C, and Java with high level abstractions for specific needs."

"Algorithms developed in MatLab for modeling and off-line validation; C++ and Java are written on Android phones to reproduce codes in Matlab"

"The sensor [...] software was mainly written in C and C++. The [server software] shown to end users with a web based system was written in C++ and Ruby on Rails."

Quite simply, while "low-level" languages are still popular among CPS developers, high-level languages are also commonly used for CPS development.

I entered my studies with the perception that CPS developers are hampered by severe resource constraints in their deployment environments. My analysis of the survey results hinted that this might not be the case. In my one-on-one interviews, I discussed the actuality of the resource constraints of the interviewees' target platforms *and* their perceived impact of those constraints on the debugging task. The interview results indicate that CPS developers do not always perceive their target platforms to be resource constrained. Further, the CPS developers I interviewed did not perceive any resource constraints the platforms may have to be a significant impediment to development and debugging:

"*The computation platform is not resource constrained.*"

"*We don't have concerns of resources in general. However, to me, wireless sensor networks are a specific type of CPS, the device nodes are for sure resource constrained. But for other types of CPS applications, it might not be the case.*"

This is an important finding in the sense that techniques for supporting development tasks for CPS (including those for verification and validation) often have a quite significant focus on resource constraints; these efforts may, in fact, be misplaced or at least over-emphasized.

I know from my experience and from the literature that simulation is commonly used in verification and validation in CPS. However, many researchers and practitioners discount the value of simulation. During my interviews, I asked the interviewees about the simulation tools they use and their perception of the pros and cons of using simulation. The following are some samples of the resulting discussions:

"*These simulations are not very truthful. We used an in-house hybrid simulation tool, [but] even with this in-house simulation, we need real testing as the risk is too big for any undetected errors in autonomous vehicles.*"

"*I am not happy with [...] simulation tools as they are not accurate enough; they give you a basic sense of how the system would work, but actually making the simulation work requires [too much effort] to tune parameters.*"

"*We used simulation but what you can test through simulation is only a very small fraction of the problems which can potentially come out when you deploy the system.*"

A common theme was the revelation that the primary simulation tools used were in-house simulators. Further, though from my interviews, I noticed that simulation is increasingly likely to be used primarily only in the earlier stages of design to give a rough view of the system and its behavior.

Concerns about the applicability of model checking to CPS appeared to crop up in my survey; my interviews delved deeper into the use of model checkers by my interview subjects. The responses I received to the question "**How do you use model checkers in verifying and validating CPS applications?**" indicate that model checkers enjoy only a limited use by in-the-field CPS practitioners, usually employed to check only small pieces of the larger system:

"*We use a very simplistic model for partial ordered sets and use Spin to check it.*"

"*We would like to transform our questions into timed automata and feed the input into UPPAAL. But the model checking suffers from space explosion, and we have to restrict our input to very small set. It is not that useful [...] model checking [does not] fit our needs.*"

The interviewees' comments related to model checking further indicated a desire supplant model checking with more robust run-time verification that is both "online" and "incremental."

I asked, generally, "**How do you test CPS applications?**" Across the board, the responses validated our view that trial and error is currently the most prevalent approach:

"*We use simulation and trial and error to observe errors.*"

"*Mainly visual observation, look at what robots are doing, take videos and sensor data. Basically it is trial and error.*"

"*Test software isolated from sensors and controller, then use trial and error to visually observe what is going on.*"

"*Visual observation. We collect traces and print out sensor values. We manually read the traces. It is trial and error.*"

"*We use ground truth and testing to compare results.*"

"*We mainly use automated formal verification and some code review for the kernel part.*"

"*We used volunteers to collect real data and applied them to MatLab models. The real test is done on real patients. Trial and error. Ground truth is provided by nurses and cameras.*"

The majority of our survey respondents identified a lack of formal connection to models of physics as a key concern. I explored this gap more in the interviews by asking the subjects about the software and physics models they employ. I was surprised to find that CPS developers tend not to deeply consider (formal) models of physical systems during development. They also found available software models inadequate. Some examples of their responses include:

"*The environment is not ideal, we created static physics models to handle noise. The models are still immature and fixed. We need on-line learning models.*"

"*We used physics models of motors [and a] flow dynamics model. We use these models to determine what forces to counteract using actuation.*"

"*We mainly used distribution models (e.g., partial order models, lattice models) to detect global [correctness] predicates. We abstract away the physics model.*"

Recent emerging work has demonstrated the use of model-driven development to automatically generate CPS software from heavily validated models [95,148]. My interviews attempted to ascertain a practitioner's view on the use of these approaches. Model-driven development, though having a quite lengthy history, is far from mature, especially with respect to CPS. Some examples of my interviewees' responses include:

"*For simple problems, model-driven development might be possible. But for complex problems, [...] model-driven development is not very realistic.*"

"*I am not optimistic about this approach. To create models that are very accurate takes too long, which is not useful.*"

"*There is a significant gap between the perceived environment and the modeled environment. [You risk] building a model more complex than the traditional programming task.*"

I also asked my subjects, "**How have you used assertions? What improvement you like to see for the use of assertions in CPS?**" My results confirm my intuition that assertions are a reasonable approach to debugging CPS, but that to make them even more appealing, especially to domain experts, an as-

sertion framework should be complemented by CPS-specific features (e.g., temporal and physics aspects).

"*We used assertions to assert the effects of actuation, mainly used for debugging. We actually need to debug assertions, as assertion happens too quickly and it fails to observe the effects of actuation. In CPS, actuation latency is not taken care of by traditional assertions.*"

"*I used assertions to figure out errors. Since I could not step through code since the interaction with the physics, I find assertions is very useful in this regard.*"

Finally, to explore my subjects' opinions on future research directions for CPS development and debugging, I ask open-ended questions, "**What are your ideal testing tools for CPS that are not currently available?**" The interview subjects described a need for integrated simulation tools, more accessible yet expressive modeling languages, and debugging tools that give programmers greater visibility into the entire system's behavior (both cyber and physical) and better fault localization. The following are direct quotes:

"*high-fidelity simulation with on-line learning of models.*"

"*accurate run-time models of physics and software models to use for off-line development.*"

"*tools that can reproduce bugs.We could throw random errors into the model to check how the system reacts. It is also ideal to have multi-domain models for motors, mechanical systems (for integrated simulation).*"

"[techniques for] formally specifying behaviors, automating fault localization by specifying a syndrome (e.g., a pattern)."

"Theorem proving requires too many human intervention, any more automation can help."

"There are automated code generation from Matlab models to C++ and Java ready for smart phones. Integration tests have to be done manually; it would be good to have integration test in simulation for heterogeneous models."

"Need a formal specification language for better clarification. Integration test can not be automated between client and server side. [Sensor] node developers and server developers have to manually collaborate for testing."

2.6 Future Research Directions

From my literature review, survey, and interviews, I have collected a set of potential research directions that have the potential to move CPS debugging into a world where the techniques are more rigorous and repeatable than the *ad hoc* testing that is the current state of the practice.

Formal Methods. To analyze continuous aspects of CPS, higher-order-logic automatic theorem provers [80] have been employed. Reducing the enormous amount of user intervention required is a key research challenge. Further, these approaches need to be made more expressive to capture the heterogeneity of CPS. As a future direction, a generic prover supporting other forms of differential equations (i.e., non-homogeneous) is highly desired. Theorem proving can also be supplemented by static timing analysis to perform program flow analysis, making tradi-

tional theorem proving more tenable. Future research in static analysis must deal with the challenges imposed by complex hardware (e.g., multicore platforms with caches) [38]. As for model checking, it would be ideal to have an integrated platform to combine model checkers; for instance, CPS systems could benefit from a combination of a stochastic model checker [48] with KRONOS [57] to explore both stochastic and real-time features of CPS.

Simulation. From my on-line surveys and interviews, a simulation approach that explicitly integrates the cyber and the physical is required. Such *co-simulation* has begun to be explored, for instance, to combine the network simulator ns-2 [173] with Modelica [63] to simulate industrial automation and a power grid [3]. As a step further, CPS developers would benefit from a flexible framework for moving between full simulation (co-simulation) and a full testing environment, allowing aspects of the simulation to be incrementally replaced by physical devices and other characteristics of the real deployment environment. This framework requires an environment in which models and physical devices can "talk" the same language, making the transition from one to the other transparent to the CPS developer and his debugging task.

Run-time verification. Temporal logics are often used to specify correct system behaviors and used to generate run-time monitors for CPS. However, there are no existing algorithms to generate monitors from Metric Temporal Logic [105, 128, 145]. Combined with the promise that run-time assertions demonstrated in my studies, I expect that a run-time assertion checking framework [193] that captures the essence in MTL (e.g., specifying latency) combined with a high-level model-

ing language similar to Java Modeling Language [110] would be more accessible to developers in annotating CPS programs. Such automatically generated monitors would enable CPS validation at run-time in a non-intrusive manner with respect to functional and non-functional behaviors of the CPS applications under study. In general, I should promote solutions that do not interfere with the developer's process (see existing testing methods, for instance). Because the physical world is an essential component of CPS, that means successful approaches will likely function "in the wild."

2.7 Threats to Validity

Internal validity. I made some assumptions in some of the findings in Sections 2.4 and 2.5. For instance, from the reported high ratio of in-house simulation, I draw a conclusion that general purpose simulation tools are not sufficient. There might be other confounding variables that result in the high ratio of in-house simulation, for instance participants might have no access to the general purpose simulation tools due to license issues. The on-line survey's lack of interaction restricted us from ruling out those confounding variables. To mitigate these issues, I used the literature survey and interviews to corroborate my findings. When analyzing interviews and free text answers, I chose to use a phenomenological approach instead of grounded theory [53] because I wanted to attempt to study the *process* of CPS verification and validation and not the *agents* of the process (i.e., the developers themselves) [127]. Grounded theory is also particularly useful if existing theories about the process do not exist [54], which, given my deep literature sur-

vey, is clearly not the case here. In analyzing my results, I draw usage conclusions from perceptions about the use (e.g., familiarity with a programming language is indicative of the use of the programming language); again conclusions from the survey were substantively corroborated by the interviews. In my survey, 41% of the CPS researchers also classified themselves as developers. I did not always distinguish results between researchers and these self-classified developers. However, for any questions with significant differences (i.e., the question about the use of formal methods), I did explore these potential two communities for their comparability. My interviews focused more on practicing CPS developers.

Construct validity. The categories in the literature review and questions in my survey and interviews may neglect important aspects, which may consequently cause us to overlook key issues in verification and validation of CPS. To mitigate this, I carried out an even more in-depth literature study than is reported here; this study covers hundreds of research papers across relevant domains and publication venues. This coverage mitigates the concern that I missed a significant question for my survey or interview. Another construct validity issue lies in the number of participants in the on-line survey (25) and interviews (9). I did successfully reach a wide cross-section of disciplines and cultures, including both researchers (survey) and practitioners (interviews) across the world.

External validity. The participants in the interview are (necessarily) from a limited set of domains. These interviews do not include CPS practitioners from many interesting CPS fields like smart energy grids, and smart cities. The conclusions drawn from the interviews may not be applicable to these domains. To mitigate

these issue, I did hand pick practitioners across four continents who are directly involved with developing and deploying real (a few large scale) CPS applications from a wide range of subfields.

2.8 Research Contributions

In this chapter, I made the following contribution:

Research Contribution 1: Conduct a comprehensive investigation on the state of the art in CPS verification and validation through exhaustive literature review, structured on-line surveys, and interviews. The results lay a solid theoretical foundation and create strong motivation for the rest of the works.

2.9 Chapter Summary

I generated an overall picture of the state of the art and state of the practice of verification and validation in cyber-physical systems through a broad literature survey, an on-line survey of CPS researchers, and qualitative interviews of CPS practitioners. I focused my investigation around a set of strongly held beliefs associated with the development of CPS. The results for the first two beliefs were mixed: while some CPS developers are deeply familiar with classical software engineering approaches, many are not and even those that are familiar do not apply these techniques generally to CPS. I dispelled the second two beliefs: in fact, high-level programming languages are used by CPS developer experts, and these same experts are not overly hindered by resource constraints. I confirmed that existing formal method techniques and simulation are, as yet, insufficient for supporting the

development of entire general-purpose CPS. I also confirmed strongly that the current state of the practice in CPS verification and validation remains an *ad hoc* trial and error process. Finally, I confirmed that there are still significant gaps between the formal models of computing and the formal models of physics that underpin today's CPS systems. This investigation has elicited a set of research directions that have the potential to directly address challenges that real CPS developers cited in the experiences in developing and debugging real-world CPS.

Chapter 3

BraceForce: A Middleware to Integrate Sensing in CPS Applications

From the survey, I chose to work on the most pertinent research direction proposed, which is to provide a practical and physically informed runtime verification tool-sets specifically designed for CPS as a sound foundation to the trial and error practice identified as the state of the art in verifying and validating CPS. However, runtime verification of CPS requires use of sensor networks in different debugging and deployment environment, and our ability to seamlessly and efficiently incorporate sensor network capabilities remains astoundingly difficult. Today, accessing remote sensing data and integrating this data into the adaptive behavior of a dynamic user-facing CPS application requires interacting with multiple sensor platform languages, data formats, and communication channels and paradigms. In this chapter, I present BraceForce, an open and extensible middleware that allows developers to access the myriad remote sensing capabilities inherent to cyber-physical systems using very minimal code. Further, BraceForce incorporates event- and model-driven data acquisition as first-class concepts to provide efficient access to sensing while retaining expressiveness and flexibility for applications. This chapter presents the BraceForce middleware architecture and key abstractions, describes their implementations, and provides an empirical study using BraceForce to support

CPS applications. More relevantly, BraceForce serves as a supporting infrastructure for the entire tool-set by providing access to various sensors available in the debugging environment [199, 200].

3.1 Introduction

Developing and deploying CPS applications involves a large amount of low-level programming that requires interacting with different (often proprietary) data formats, languages, and operating systems. In practice, applications are built for specific sensor network platforms with little potential for portability to other platforms or integration with other sensors. Debugging CPS applications that inherently integrate with a physical environment requires not only the aforementioned integration of the application with sensing but also the use of a testing harness that, at debugging time, accesses sensed data about the physical environment for the purposes of validating the actions of the application. Integrating sensing for application and debugging support in a way that is easy, flexible, and portable is essential for supporting CPS application development.

In general, research in this space has been focused on demonstrating the feasibility of applications, the development of support services such as routing protocols or energy-saving algorithms, or on advancement of hardware platforms and operating systems. Little focus has been applied to effective development support for applications that integrate the capabilities of networked sensing platforms in easy to use and extensible ways. In addition to the variety of data formats, communication technologies, and programming platforms a developer must tackle, CPS

applications also require handling network dynamics and energy constraints.

This chapter introduces BraceForce, a middleware for CPS application development that simplifies the development, deployment, and debugging of CPS applications. In supporting application deployment, BraceForce allows the developer to connect the application to sensing assets in the deployment environment. In supporting application debugging, BraceForce can be used to monitor a test environment using capabilities that may not be available in the deployment environment. Architecturally, BraceForce defines functional *tiers* that encapsulate related aspects but coordinate in such a way that the tiers' deployment to particular physical assets is flexible. Different tiers can be deployed on user-facing devices or on sensing devices with limited capabilities, directly addressing and leveraging the specific capabilities and intentions of each device. BraceForce provides auto-discovery of new sensing and computing assets, allowing easy integration of new capabilities into existing applications or automatic and seamless extension of a debugging environment. As BraceForce discovers new components, the tiered architecture organically extends to these new components, flexibly deploying the necessary tiers to the new components.

Instead of using a proprietary programming language (e.g., nesC [71]), supporting a single hardware platform (e.g., ODK [39]), or creating a new standard (e.g., Dandelion [120] or DASH [56]), BraceForce presents a simple Java programming interface for integrating new sensing capabilities. To add a new sensor to BraceForce, a developer just needs to implement a simple sensor driver interface that provides essential commands (e.g., `open`, `close`, `query`) and configuration. The

driver implementation can be specific to the particular sensor but basically translates raw data into meaningful BraceForce sensor data constructs, guided by the provided BraceForce interfaces. BraceForce handles issues such as thread management, networking, remote procedure call, etc.

BraceForce supports both the traditional *pull* and more flexible *push* based interaction with sensing devices. Out of concern of energy-saving, BraceForce's default form of interaction is *event-driven data acquisition*, in which sensed data is only pushed to an interested application on the occurrence of an *event*. Applications specify thresholds on changes in sensed values that determine when new samples are returned. Such an approach has been shown to dramatically reduce energy costs of interacting with networked sensors. Existing approaches that support some integration of sensing capabilities with mobile application development [39] provide only continuous sampling or instantaneous polling, which require the application to tune the interaction with the sensing platform and therefore either expend extraneous energy or miss important sensed value changes.

BraceForce embraces *model-driven data acquisition* (MDDA) [94, 138, 139, 162, 187] to reduce energy costs of integrating sensing in CPS applications. MDDA suppresses sensor readings that are *predictable* according to some *a priori* or learned model. BraceForce supports (*i*) temporal models of data evolution based on previous sensor readings [162]; (*ii*) models based on underlying physics principles of the sensed system [138]; (*iii*) models derived from applying data mining to prior sensed data [94]; and (*iv*) models expressing correlations of data in space and time [187]. While the use of model-driven data acquisition is itself not novel to BraceForce,

what is new is how BraceForce integrates model-driven data acquisition in a general purpose programming framework for acquiring and interacting with sensed data.

This chapter describes BraceForce, from its flexible architecture to a prototype implementation. I directly assess BraceForce's ability to ease CPS application development through an exploratory study of BraceForce applied to real CPS applications that rely on networked sensors. BraceForce provides a clear design contract for integrating new sensors into existing applications. This chapter makes the concrete contributions in three areas:

1. *Supporting CPS application developers and users*
   - BraceForce provides a simple and unified programming interface to reduce programming barriers of CPS applications that integrate sensing.
   - BraceForce's tiered architecture enables dynamic updates to CPS applications and their physical deployments without intervention by the users.
2. *Supporting flexible and expressive CPS applications*
   - BraceForce can be deployed to heterogeneous devices in different configurations to meet constraints of CPS applications and devices.
   - BraceForce supports both pull- and push-based interaction with sensing devices.
   - BraceForce embraces model-driven data acquisition to address energy concerns of CPS applications.
3. *Studying CPS application development*
   - I demonstrate and evaluate how BraceForce makes both the development process more approachable and the resulting CPS application more efficient.

- I demonstrate that MDDA within BraceForce can significantly reduce the network overhead, which in turn reduces energy consumption.

The next section places this chapter's contributions in the context of related work. In Section 3.3, I provide complete details of the BraceForce middleware. Section 3.4 presents the empirical design for my evaluation, while Section 3.5 describes its results.

3.2 Related work

BraceForce aims to make sensor programming transparent to CPS application developers by explicitly and intentionally hiding low-level sensing concerns. BraceForce provides CPS application developers dynamic deployment and automatic discovery of networked sensors, distributed data caching and aggregation capabilities, and presents the networked sensors through a combination of event-driven and model-driven accessors, providing both flexibility for developers and efficiency for the deployment. Existing approaches tackle similar and in some cases overlapping concerns, but, to my knowledge, BraceForce is the first to provide transparent support of CPS application development and debugging.

Sensor Platforms. Much work on sensor networks has focused on hardware and software platforms to support increasingly sophisticated capabilities. TinyOS [85] is an operating system for sensor networks that enables developers to use networked sensors to solve distributed sensing, computational, and coordinating tasks. Its focus has largely been on supporting increasingly complex and sophisticated applications at the expense of usability and flexibility [118]. The Robot Operating

System (ROS) [155] provides a component-oriented style of programming, in which components communicate via publish/subscribe mechanisms and two-way service calls, both using user-defined *topics*. However, ROS only supports a communication scale of one machine, and connection mechanisms are among ROS components running on the same machine; CPS systems require coordination among distributed components. ROS also comes with a non-trivial learning curve.

Arduino has gained wide popularity due to simplicity, high degrees of usability, and the resultant enabling of rapid prototyping. While programming for Arduino is more straightforward, programmers are still required to interact with the Arduino at a very low-level of abstraction. This does not allow for flexible updating and discovery of local sensing devices. The Android Sensor Framework [11] provides a programming interface to access hardware and software sensing components on Android. This framework is also limited in its abstract capabilities; the developer is still entirely responsible for thread control and other essential concerns for accessing sensors. Further, the Android Sensor Framework does not enable connecting to external sensors connected to the device via BlueTooth, USB or other connectivity modes. Instead, the developer must use other libraries.

Integrating sensing in CPS applications requires high level abstractions to provide automatic sensor discovery and flexible and efficient access to the sensor data via event- and model-driven data acquisition. BraceForce unifies access to a variety of sensor platforms and provides a flexible and expressive application programming interface with high-level programming constructs tailored to networked sensor integration.

Sensor Programming Frameworks. Easing programming of sensor networks has received significant attention. Maté [119], for example, is a tiny virtual machine that allows developers to concisely express sensor programs and cause these programs to be dynamically deployed. A virtual machine approach is very flexible, but Maté comes with the cost of increased complexity due to the severe and unmitigatable resource constraints of the target environment [84]. The mixed (and richer) capabilities of the target CPS environment allow us to circumvent these resource constraints using a distributed architecture. In addition, I can embed data-driven techniques in my programming abstractions to make acquiring and integrating sensor data a natural part of the programming task.

SensorWare [36] shares the resources of a single node among many applications. SensorWare's primary abstractions have a network focus and are intuitive for sensor and networking experts but do not promote data and data integration for application developers. MiLAN [82] builds on networking and discovery protocols, using a plug-in mechanism to incorporate arbitrary protocols. Application developers use QoS graphs to specify their sensing requirements, and MiLAN uses this information along with the sensor network state to determine how to configure the network and sensors to meet the provided requirements. This high-level data-centric abstraction is the style I target in my middleware. I couple these abstractions with automatic sensor discovery and a distributed architecture that, by its nature, addresses the mixed resource constraints of my target environment.

Dandelion [120] supports developing wireless body sensor applications on smartphones using a programming abstraction called a *senselet*, which abstracts

a device driver and allows applications to integrate data from that device through remote method invocation. This abstraction is data-centric and provides a jumping-off point for BraceForce's abstractions, but Dandelion does not incorporate higher-level programming constructs for aggregation, model-driven data acquisition, and automatic sensor discovery and integration.

Perhaps most similar to my goals, Open Data Kit (ODK) Sensors [39] simplifies deployment of smartphone applications that rely on data from a smartphone's external sensors (e.g., connected via USB and BlueTooth). The application logic may rely on data from both on board and external sensors, and the sensor driver developer implements specific driver interfaces for configuring sensors, packaging the data they generate, and connecting that data into the Android framework. Interactions with sensors are confined to pull-based acquisition with only locally connected sensors. ODK Sensors is tightly integrated with Android, limiting its applicability.

I focus on easing the integration of distributed sensor data into CPS applications without limiting the expressiveness of sensing capabilities or over-utilizing precious constrained resources. This demands not only automatic sensor discovery and integration but also abstractions for acquiring sensor data in ways other than through direct sensor polling.

Model Driven Data Acquisition. Model-driven data acquisition (MDDA) can limit costly communication with networked sensors by suppressing polling and notifications from the sensors except when the sensor readings deviate from some pre-defined or learned model. Gupta et al. studied the problem of collecting spatially correlated data in a wireless sensor network, based on the theory of dominating

sets [76]. Other work has focused on achieving data suppression using temporal and spatial data correlation either by dividing the network into clusters [123], exploiting domain knowledge regarding reasonable ranges of sensed values [186], or combining temporal and spatial correlations [187]. Other approaches use simple models of sensed value trends to generate readings only when the sensed value deviates from the model's predicted value. Simple linear models are very effective [162], and the approach can also be applied to in-network aggregation of raw values [61].

I am motivated to use similar techniques in BraceForce to suppress data without sacrificing the quality of knowledge about the sensed entity. BraceForce allows CPS developers to incorporate different models of MDDA both at the level of individual sensors and at the level of an entire distributed application to handle different deployment scenarios and meet specific energy requirements of CPS applications. I avoid approaches that demand a particular topological structure (e.g., designated clusters of nodes) to avoid unnecessary rigidity in a dynamic network of supporting sensors and rely on simple models that have shown significant gains.

3.3 BraceForce

I first describe BraceForce from an abstract architectural perspective; I then detail my prototype implementation.

3.3.1 BraceForce Abstract Architecture

BraceForce comprises five abstraction layers: the *sensor driver* layer, the *data* layer, the *distribution* layer, the *distributed data cache and aggregator* layer,

and the *CPS application* layer. Fig. 3.1 shows one example BraceForce deployment with these five layers.

3.3.1.1 BraceForce Layered Architecture.

Sensor Driver Layer. In the current state of the art, CPS application developers must understand low-level protocols and hardware-specific aspects of sensors to be able to write sensor drivers. Moreover, the raw data from the sensor has no open standard and often loses the original temporal information associated with the sensor data retrieval. These issues demonstrate a gap between the low-level sensing capabilities of both on-board and external sensors and the application space. At BraceForce's base, the sensor driver layer bridges this gap. The sensor driver layer encapsulates functions related to interacting with a sensor, from configuration, through starting, querying, reconfiguring, stopping, and cleaning up allocated resources. The sensor driver layer requires driver developers to adhere to a design contract enabling these common functionalities and unifying on-board and external sensors into a shared data packet format that can be used throughout the remainder of the BraceForce architecture. A driver developer creates the connection between the sensor and the driver layer by defining how raw sensor data (e.g., in the form of a binary array) is converted into a BraceForce *sensor data response*, which defines an abstract data type unified across all sensor data. Each sensor data response maps a data type (or types) to a value (or values) and associates a timestamp with the data. This timestamp ensures that data used by higher layers is *fresh* within the requirements of the application.

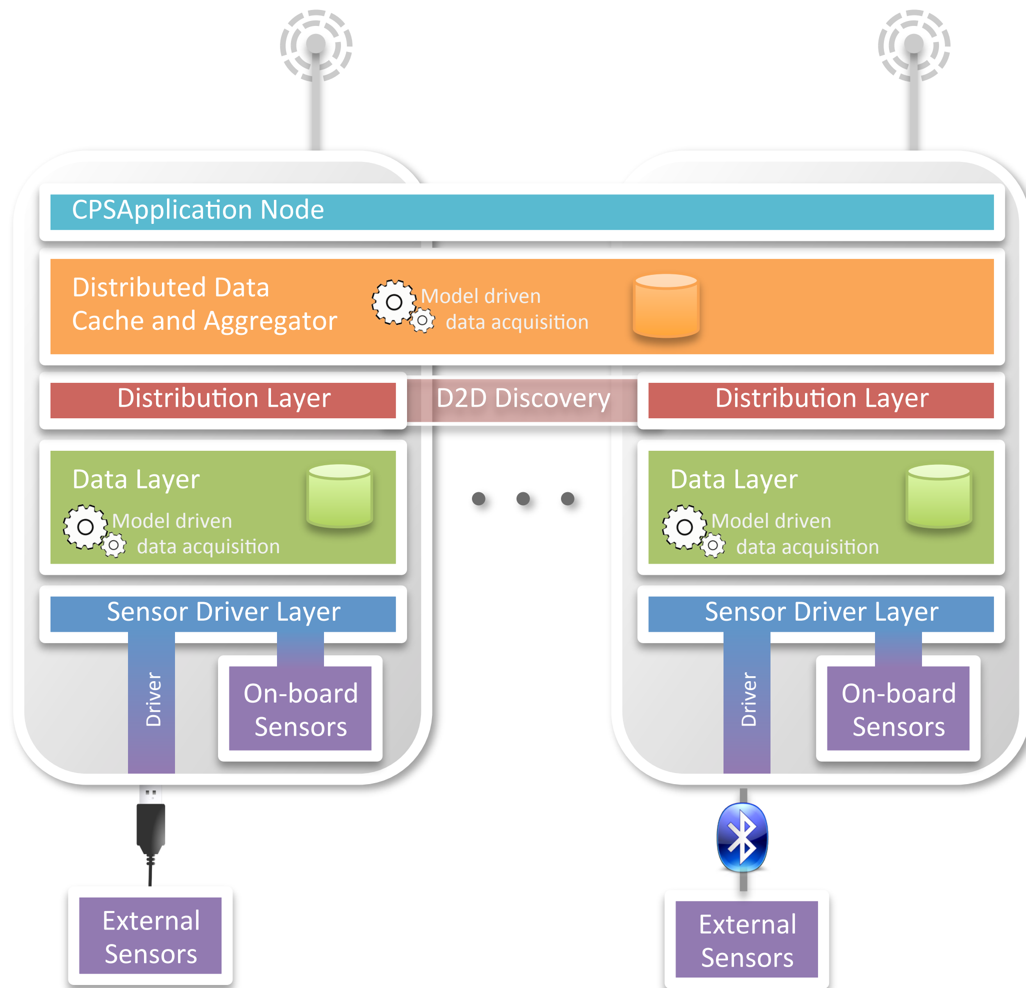

Figure 3.1: BraceForce: Uniform Environment

Data Layer. Mobile operating systems allow sensors to be accessed directly using BlueTooth, Near Field Communication (NFC), USB, and proprietary interfaces to on-board sensors. This manner of access is device-specific and requires a non-trivial amount of low-level development and testing using interfaces that are not portable across platforms. These programming methods also usually entail a steep learning curve. BraceForce encapsulates these communication and sensor framework

interactions in the data layer, thus explicitly separating user-application code above from sensor- and platform-specific code below. The data layer also controls data retrieval for locally connected sensors (both on-board and external), where the options include *push*, in which a sensor pushes every data reading to the data layer; *pull*, in which the data layer periodically polls the connected sensor; and *event-driven*, in which a sensor notifies the data layer only upon the occurrence of some predefined "event. " At the data layer, BraceForce introduces single-device MDDA, whereby simple models of temporal correlation can be used to suppress sensor readings as long as they follow an expected model [162].

Distribution Layer. CPS applications require a distributed view of active and available sensors, both on the local device and available remotely on other devices in the network. Programmers must use message passing mechanisms or RPC-like connectors to make devices in the network communicate and coordinate their actions. Developers must also handle location transparency and other intrinsic issues related to distributed programming (e.g., concurrency, failures, and time synchronization). In the distribution layer, BraceForce provides automatic discovery of other BraceForce-enabled devices, and thereby other BraceForce connectable sensors. BraceForce uses a combination of best effort protocols for the on-the-fly discovery and reliable protocols for initiating and accepting remote method calls to transfer sensor data from discovered sensors. The distribution layer also extends the data acquisition modalities of the data layer by defining data listeners that bridge between the data layer below and the aggregation layer above and, more importantly, across distributed devices. One specific function of this bridge is to use guidance

from the cross-device MDDA performed at the aggregation layer (described next) to correctly configure and query underlying (distributed) sensors. Within the distribution layer, BraceForce's automatic detection of distributed sensors can be deployed alongside the Sensor Driver Layer, circumventing the data layer, simplifying the BraceForce deployment on devices that have significant computation and storage constraints.

Distributed Data Cache and Aggregator. CPS applications that require large scale deployments often include sensing devices that have severe computation and storage constraints. In this case, it is highly ideal that data storage and processing are done in backend servers or other more powerful devices in the network. Programming these features in a scalable and reliable way is a challenge for CPS application developers. BraceForce's aggregation layer is specifically designed to deal with this challenge through a unified view of all of the dynamic sensing capabilities of the networked sensing support infrastructure. As new sensors are discovered on-the-fly or known sensors disappear, these dynamics are handled seamlessly by BraceForce. Given a distributed view of aggregate sensing capabilities, the aggregation layer can perform high-level MDDA, for example, by using more sophisticated models based on spatial correlations or learned relationships among sensed data [139]. In my prototype, I demonstrate the potential for this high-level MDDA using a spatial correlation model that suppresses sensor readings that are similar to neighboring readings. More generally, from the aggregation layer, BraceForce can expressively direct data acquisition (e.g., by choosing between event-driven data acquisition, the push modality, and the pull modality) for all of the sensors under its purview. This

layer also encapsulates data mining and compression.

CPS Application Node. To support the CPS application node, BraceForce maintains a registry of aggregation and sensing capabilities, and the programming interface allows the application to subscribe to them. Based on the particular application goals, these sensors can be integrated just as part of a testing harness at debugging time or can provide essential functional information for the application at deployment time.

3.3.1.2 BraceForce Deployment Scenarios.

Fig. 3.1 showed just one possible deployment scenario in which all of the devices are homogeneous (e.g., smart phones) and have the same capabilities. In BraceForce, different physical assets can support different pieces of the architecture, depending on a particular device's capabilities and functional requirements. In any deployment, the sensor driver layer and the discovery layer must be present on any sensing-capable device, as they are essential to connecting to and getting data from sensors. On moderately capable devices, the addition of the data layer adds multiple modalities for driving data acquisition and the potential for expressive temporal-based MDDA. Fig. 3.2 shows an alternative deployment in which the system consists of heterogeneous devices; in this example, the application runs on a dedicated high-powered device (e.g., a laptop) that connects to a variety of sensing devices of different capabilities, held together by the distribution layer.

I envision three primary use cases for BraceForce. In the obvious case, BraceForce is an integral part of the CPS application. Take a smart home application

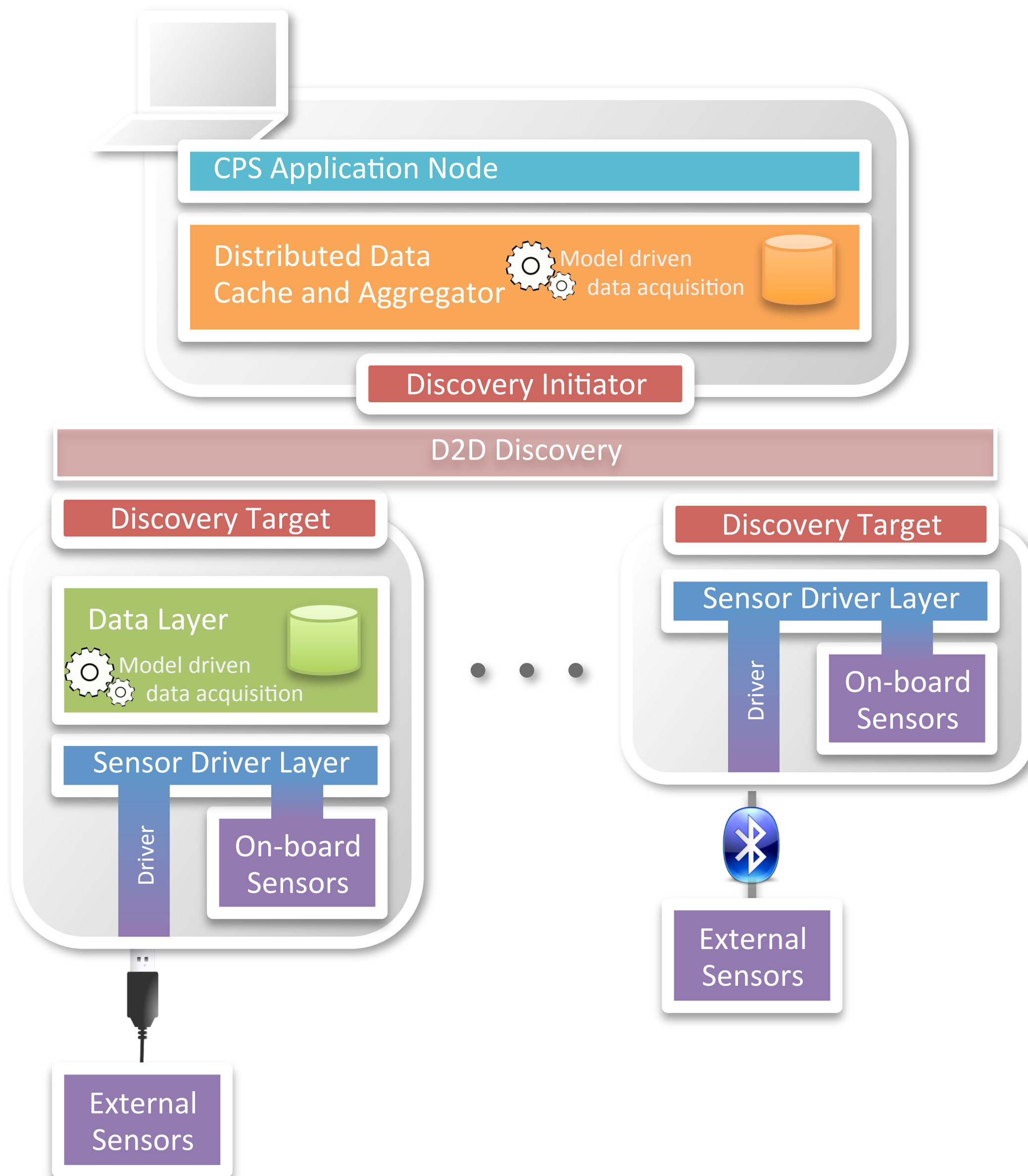

Figure 3.2: BraceForce: Mixed Environment

in which a home controller connects to various sensing devices integrated with the home to control the ambient environment. In this case, the homeowner may introduce and remove components from the home over its lifetime, and, using BraceForce,

the changing capabilities are seamlessly reflected to the application.

As a second case, I envision the middleware to be deployed for large scale ad-hoc sensor networks (e.g., to monitor the fire situation in a forest). The Sensor Driver Layer and Discovery Layer can be combined into a service to be deployed on cheap sensor devices that have basic sensing and networking capacity without additional computation and storage resources. These devices can be deployed at a mass scale. The Distributed Data Cache and Aggregator can be deployed as service to more powerful Android devices or a cluster of backend servers for data processing. The sensor data then will be passed to the CPS application node, which runs on user-facing Android devices to notify users (e.g., fire workers and rescuers) of the real time situation in the environment and allow them to make informed actions.

As a third use case, consider a mobile autonomous robot application wrapped in a test harness that connects to expressive sensing capabilities available in the debugging environment but not expected to be available in the deployment environment. At deployment time, the robot may be placed in an unknown territory and expected to perform some tasks. At debugging time, however, the developer may control the environment and may be able to monitor various aspects of the robot using sensors embedded in the environment (e.g., overhead cameras). The debugging program that surrounds the actual control application can use BraceForce to access these sensors for debugging; at deployment time, these connections to BraceForce and the debugging environment are removed.

3.3.2 BraceForce Implementation

To demonstrate and evaluate BraceForce, I have built the entire architecture in the Android operating system. I choose Android as my initial platform as it is open source and has extensive support for background processes, including several built-in constructs for inter-application communication [167]. BraceForce is not particular to Android, and I avoid using low-level constructs and interfaces specific to Android and not replicated on other platforms.

3.3.2.1 Android Programming Idiosyncrasies

A handful of accidental complexities arise from my choice of Android; below, I describe these challenges and my solutions as they arise. In addition, to understand the discussions of the architecture, I briefly review a vocabulary related to Android:

Intent: a passive data structure holding an abstract description of an operation to be performed; something that has happened and is being announced.

Android Interface Definition Language (AIDL): allows definition of the programming interface that the client and service use to communicate with each other using interprocess communication (IPC).

Bundle: a data structure of key-value mappings but not limited to a single String/Object mapping.

Android does not provide significant high-level abstractions for programming coordination among distributed devices. From a model perspective, BraceForce assumes such capabilities, e.g., in the form of Java RMI. To support my BraceForce

prototype on Android, I therefore implement my own version of RMI by building on several open source projects. I use JsonBeans [100] to serialize and deserialize Java object graphs to and from JSON [55]. JsonBeans is an obvious choice for this task because it is very lightweight (45KB) with no external dependencies. I use ASM [12,106] to dynamically generate classes involved in the RMI process in binary form.

3.3.2.2 Unified Data and Subscription Interfaces

Programming for Android requires interacting with the Dalvik virtual machine, and it is inevitable that my implementation accesses some proprietary Android constructs (e.g., Bundle). To remain as general as possible, I implement a wrapper that encapsulates these proprietary components and presents a generic interface to the BraceForce implementation. Within this wrapper, BraceForce translates Android data structures to reusable and portable Java data structures.

Android provides sensor data subscription only for internal sensors. To subscribe for sensor data from external sensors (e.g., sensors connected via USB), developers have to create a subscription model themselves (e.g., using the observer design pattern) and write a large amount of low-level code to access sensor data. Retrieving data from other devices is even more difficult. In BraceForce, I provide a unified subscription interface for developers to access internal sensors, external sensors, and networked sensors. Fig. 3.3 shows the BraceForce API used to subscribe sensed data from an internal accelerometer (line 1), a temperature sensor (Dallas DS18B20) connected via USB (line 2), and all accelerometers on networked devices

(line 3). Line 4 shows how to retrieve sensor data; it is the same regardless of the type of sensor connection. The retrieved data contains a timestamp of the data and where the data is from; data values are accessed through meaningful keys instead of array indices provided by Android.

```
1 BraceSensorManager.subscribeSensorEvent(BuiltInSensorType.ACCELEROMETER.name(), this, this);
2 BraceSensorManager.subscribeSensorEvent("DS18B20", this, this, SensorAutoDetection.USB );
3 BraceSensorManager.subscribeSensorEvent( BuiltInSensorType.ACCELEROMETER.name(), this, this,
      true);
4 BraceSensorManager.retrieveSensorData(event);
```

Figure 3.3: Interface for data subscription

3.3.2.3 Thread Management and Android services

Android provides much support for multi-threaded applications. As with any multithreaded environment, the added flexibility comes with a significant increase in complexity. In Android applications that interface with sensing, developers must implement (and debug) threads to listen for and connect to multiple connections. These tasks are far from trivial, as they require the developer to have a deep understanding of the relationships between the operating system and the life cycles of application components. As my user study (described in Section 3.5) demonstrates, developers have a difficult time navigating the complexities of the Android APIs related to thread management and concurrency.

BraceForce exposes thread managers that explicitly enable thread-safe access to the shared sensor data available to the CPS applications. To enable communication beyond shared memory, I expose the threads and their embodied data using the Android Binder service's remote procedure call capabilities. To conserve energy, BraceForce activates threads only when necessary. This is guided by developers

through the interface in Fig. 3.3, e.g., by specifying the types of communication interfaces to attach to event listeners. For example, in Fig. 3.3, line 2 specifically indicates that BraceForce should activate the USB thread to communicate with the connected temperature sensor. Alternatively, a developer with less experience in low-level details can indicate, via a boolean parameter, activation of all the networking threads, albeit at increased cost.

3.3.2.4 Networking

my prototype supports simple device-to-device discovery based on UDP. The relevant threads for discovery and exposed Android Binder services are built in the distribution layer. When each discoverable service is started within BraceForce, the distribution layer implementation creates a new Android intent and attaches the discovery capability to that intent. When the discovery layer starts a service, it listens on the UDP port specified in its intent and reacts to discovery requests received on this port. The discovery layer also actively engages in discovery by sending UDP packets to neighboring devices. Each node maintains an active list of other nodes; this list is maintained by removing any nodes that have failed to respond to three consecutive periodic requests. Using UDP for discovery is a simple approach to enable automatic device discovery. BraceForce could potentially integrate other on-the-fly discovery mechanisms in the distribution layer, e.g., [13], so that devices can talk to each other even when they are in different networks (e.g., behind NAT).

3.3.2.5 Sensor Driver Definition and Discovery

BraceForce's driver layer separates CPS application developers from sensor driver developers by providing a contract for implementing new sensor drivers. Fig. 3.4 shows this contract and the simple `SensorDataResponse` structure, which is mapped to a Java `Hashtable`. The interface shown in Fig. 3.4 provides a high-level abstraction that allows access to sensor data in key-value pair fashion instead of requiring the application developer to directly interact with myriad forms of raw sensor data.

```
1  public interface Driver {
2    byte[] getCmdForSensorConfig(String configKey, Hashtable configParams);
3    byte[] getCmdForSensorData();
4    byte[] getCmdForStartSensor();
5    byte[] getCmdForStopSensor();
6    SensorDataResponse getSensorData(List<SensorDataPacket> rawData);
7    byte[] sendDataToSensor(Hashtable actuatorData);
8    void shutdown();
9  }
10 public interface SensorDataResponse {
11   List<Hashtable> getSensorDataInCollection();
12 }
```

Figure 3.4: Interface for defining sensor drivers

The driver interface is exposed through an AIDL file. I have built the drivers for common built-in sensors on Android devices. For any external or uncommon sensors, the sensor driver developer must implement the interface in Fig. 3.4 to provide access to those sensors[1] . This is accomplished by creating an Android library project that includes the interface implementation and `AndroidDeviceDrivers.aidl`. The project can be uploaded to the mobile devices connected to the sensor(s) for which

[1]While the byte[] data type clearly requires programming at a very low-level, the sensor driver developer is an expert in the sensor and, as such, is required to think about sensors at a low-level. The CPS application developer, however, is shielded from these low-level concerns by the high-level data types that BraceForce provides.

the driver(s) are written. Given the project's manifest file, the BraceForce driver layer can automatically detect the driver definition and create AIDL stub clients that connect to the new driver service.

The BraceForce data layer automatically discovers connected sensors (e.g., those attached via USB and BlueTooth in my prototype). Once the data layer in BraceForce detects external sensors, a manager agent interacts with the underlying communication channel manager and the created driver stub to capture raw sensor data, convert it to the BraceForce `SensorDataResponse`, and make it available for the higher layers of the architecture. The manager's interface allows the higher level (and ultimately the application developer) to specify how BraceForce should interact with the sensor. In the case of push-based interactions or event-driven data acquisition, the application developer defines the behavior of a listener, as shown in Fig. 3.5.

```
1 public interface SensorDataChangeListener {
2   void bindDataProvider(SensorDataProvider provider);
3   void onDataChanged(SensorDataChangeEvent event);
4 }
5 public interface SensorDataProvider {
6   List<Hashtable> getSensorData();
7   void addSensorData(Hashtable sensordata);
8   void addSensorData(List<Hashtable> sensordataList);
9 }
```

Figure 3.5: Data Change Listener Interface

The first method in the interface binds the listener to a data provider; the latter provides a wrapper for a specific sensor/driver pair on a specific device. The second method is invoked automatically by the data layer when a reading is pushed or an event notification occurs.

3.3.2.6 Model Driven Data Acquisition

To help CPS application developers to design energy-efficient solutions, BraceForce enables MDDA within the data layer for single device models and within the aggregation layer for distributed models.

In my prototyped data layer, I use the temporal model from [61]; specifically, if the value change for a single sensor between two consecutive readings lies below a specified threshold, the reading is suppressed (Fig. 3.6). This is a simple form of data suppression based on temporal data correlation; even only slightly more complex models [162] may substantially further reduce the overhead of high quality sensing since the energy of communication dominates computation [187]. my goal with this simple implementation is to demonstrate how MDDA dovetails with the BraceForce architecture.

```
1  float lastData =
2   (Float)historicalDataPack.get(mainParameterName);
3  float thisData =
4   (Float)currentDataPack.get(mainParameterName);
5  long accessTime =
6   (Long)historicalDataPack.get("timestamp");
7  long currentTime =
8   (Long)currentDataPack.get("timestamp");
9  if(currentTim−accessTime<timeDelta){
10   if(Math.abs(thisData−lastData)<dataDelta){
11     Log.d("SensorEventDrivenListener",
12           "Data suppressed");
13     return;
14   }
15 }
```

Figure 3.6: Simple temporal data correlation

In the aggregation layer, I apply MDDA using simple spatial data correlation. The data aggregation service maintains a list of nearby devices and their sensing capabilities. Periodically (as specified by the application), BraceForce ag-

gregates sensed values by type from all connected devices. my model assumes that each sensed value has within its `SensorDataResponse` a location and timestamp. my implementation of spatially correlated MDDA checks the sensed values from devices located close to one another and suppresses readings from sensors that would be redundant with respect to the spatial coverage of the sensing task. Because the aggregation layer is above the distribution layer in the BraceForce architecture, this data suppression decision must be transmitted to the distributed sensing devices; this is accomplished through the RMI implementation encapsulated within the distribution layer. I use a simple model that suppresses readings for a device physically located between two similarly capable devices if the two devices on either side sense values within a specified threshold of each other.

3.4 Empiricial Design

Using my prototype implementation, I have carried out an empirical evaluation to answer the following questions about the performance and use of BraceForce.

Question 1: Does BraceForce simplify development of applications that require interaction with distributed and aggregated CPS sensor data? If so, how and why?

Question 2: What are the potential ramifications of using BraceForce to interface with the sensors?

Question 3: Can BraceForce save energy by employing even primitive predictive models of temporal and spatial data correlation? If so, under what conditions does it work and at what cost to quality of knowledge supplied to the application?

Question 1: Simplifying the Programming Task. I conduct a user study involving twelve participants from a diverse student population and with varying levels of programming experience. All have basic knowledge of Java. I ask each participant to fill in the sensor data retrieval sections for three different applications.[2] Each participant performed each of the three tasks once using BraceForce and once using the Android SDK alone. Each participant was awarded a $20 Amazon gift card. The order of the two subtasks was randomized for each user and each application.

- Application 1 relies only on a single Android internal sensor (the accelerometer). The application mimics part of a smarthome CPS application that uses an Android device to detect the user's shake orientation. The hardware used is an Android phone.
- Application 2 relies on an external temperature sensor. The application represents part of a CPS application that requires getting information from external sensors to an Android device. The hardware includes a temperature sensor (Dallas DS18B20) connected via an Arduino MEGA board, which in turn is connected to an Android tablet.
- Application 3 mimics a large scale wireless sensor network. Sensor readings come from accelerometers on low-budget Android phones. The sensor data must be aggregated to a more powerful data processing device (an Android tablet). The sensor data is then displayed in an application running on a user's Android phone.

[2] Videos of how BraceForce works for these three application are available at `http://goo.gl/62WMyc`, `http://goo.gl/KVDQZT`, and `http://goo.gl/5mE64m`

The participants create the sensor retrieval and aggregation components (including a primitive MDDA model of temporal data correlation for which pseudo-code is provided). These components are deployed to the Android mobile phone and an Android tablet, respectively. Connectivity among the devices relies on a local wireless network.

I provide training sessions for the participants, which includes a five minute session on BraceForce, a five minute session on the scope and functions of the applications,[3] and a thirty minute session for the Android architecture and APIs related to sensor data retrieval.[4] The participants were given additional material on Android (e.g., URLs for highly relevant Android programming) to review at home prior to the study. I record for each participant how many hours they spent on Android reading after the training but before the study. To rule out other confounding variables (e.g., human fatigue), I set the time limit for each user study to four hours.

Before the user study, each participant fills out a pre-questionnaire to provide basic information of their programming experience in general, and specifically with Android and with sensors. After each user study, each participant fills out a post-questionnaire about how they feel about using BraceForce compared with Android SDK for each application and their overall feeling about the middleware. If a user fails to complete an evaluation application, in the post-questionnaire I ask what led to the failure and the user's estimate of how many more hours would be needed to

[3]The tutorials for both BraceForce and evaluation applications are available at `http://goo.gl/mMXQj5`

[4]Tutorial for Android are available at `http://goo.gl/8j9tBA`

complete it.

I measure the development time and accuracy for each user for each of the evaluation applications. The results are averaged across participants. I also provide in-depth qualitative analysis of participants' feedback.

Question 2: Potential ramifications of using BraceForce. To answer the second question, I use the same evaluation applications in the first question. In this case, I implement the Android versions ourselves as the baseline to compare with the BraceForce versions. I measure the accuracy of the applications (using out-of-band validation, for example, by measuring the temperature using an ordinary thermometer) and the running time required to acquire the sensed data; these results are averaged across five runs of each application. In Application 1 and 3, a "run" is defined as 10 distinct rotation tests; in Application 2, a "run" is defined as 10 measurements of body temperature.

Question 3: Energy Savings with Model-Driven Data Acquisition. To answer the third question, I created a fourth evaluation application, which is a piece of a smart home application in which a user retrieves light readings from multiple sensors deployed around a home. The deployment is at the home of one of the authors; the lighting levels are subject to controlled lighting as well as uncontrolled conditions that include the glow of street lights, passing cars, daylight, etc. The deployment consists of six sensing devices, a pair of devices for data aggregation, and a completely separate laptop that runs the CPS application node.

I employ MDDA at both the data and aggregation layers. I deploy the six sensing devices in pairs; one of each pair executes MDDA, while the second of

the pair does not. Across all runs, I measure both the number of data packets transmitted with and without MDDA, along with the accuracy of the sensed data. In this experiment, I am comparing the use of MDDA to not using it; therefore the device not using data suppression is treated as the "ground truth" for determining the accuracy of MDDA. The frequency of data acquisition is fixed for both groups at 10 seconds. For the MDDA group, the suppressed data is predicted to compare with the sensed data from the comparison group. For a specific time, the predicted value for a suppressed sensor is either the average of the neighboring readings (in the case of the spatial model) or a running average of the previous readings (in the case of the temporal model).

3.5 Results and Discussion

Question 1. I evaluated how BraceForce eases the development of CPS applications for each of my three applications, I evaluated the benefits of using BraceForce specifically in terms of reduction in development time. The results show the development times for each of three evaluation applications when using BraceForce are not substantially different, even though the applications are increasingly complex. However, the development times when using the Android SDK increase dramatically among evaluation applications (a 349.8% increase from Application 1 to Application 2, and then another 147.9% increase from Application 2 to Application 3). Fig. 3.7 shows a box-and-whisker plot of the percentage decrease in development time for BraceForce versus Android. BraceForce reduces the median development time in the range of 66% (for Application 1) to 98.8% (for Application 3) compared

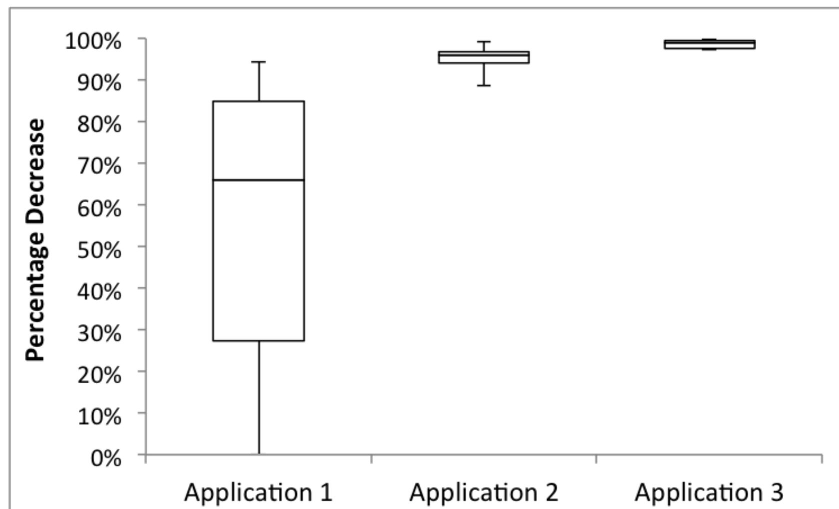

Figure 3.7: Decrease in development effort using BraceForce vs. Android sensor framework

to the Android SDK.[5] The variance for the reduction for Application 1 is wide because several of the test subjects were able to complete the (simple) Android task very quickly. As the application complexity increases, however, BraceForce's abstractions provide significant support for the development task. All participants were able to deliver working applications in all three cases when using BraceForce, but when using the Android SDK, only half of the participants delivered a working Application 1, one participant was able to successfully implement Application 2, and no participants completed Application 3.

One of the most useful sets of feedback from the post-questionnaire explained the users' stumbling blocks with respect to completing the tasks using the Android SDK:

[5]If a participant was not able to complete a specific task within the time limit (i.e., 4 hours in total), I asked the user to estimate the time he or she thought would be required to complete the task; this value was used in computing the relative development efforts.

"*Android interface is harder to understand*"

"*Complex Android APIs*"

"*Android permission is complicated; hard to locate the right documents*"

"*Android SDK is complex and not easier to understand, not user friendly*"

"*Complex logic regarding permission, broadcast receivers and so on; distributed programming in Android is very difficult*"

I also asked participants whether they agree that using BraceForce was much easier and quicker than the Android SDK and if so, why. Eleven participants responded that they "Strongly Agree" with the statement, while the remaining participant responded "Agree". As for why, the users' justifications included the following:

"*Easy to manage and read, simplify the implementation*"

"*Much more efficient, easier to use, easier to understand*"

item "*More convenient, more integrated in communication*"

"*Convenient, cleaner, user friendly*"

"*Very simple, easy to understand, elegant*"

"*Very easy to use, hides all the low level details*"

"*It hides the complex logic, easier to use, simple to debug*"

Question 2. I evaluated the ramifications of using BraceForce to interface with the sensors in terms of the accuracy of the sensing task and the running time. Fig. 3.8 shows the results. The accuracy of the versions of all three applications was not noticeably different (it was marginally better using BraceForce), but Fig. 3.8

shows that executing through the BraceForce tiered architecture has some impact on the application's execution efficiency (in terms of how long it took the application to run, including interacting with the sensors). While this burden was relatively higher (an 8% increase) for Application 1, it was substantially lower for the more sophisticated Applications 2 and 3, since the interaction with the BraceForce architecture is amortized over more extensive and potentially distributed interactions.

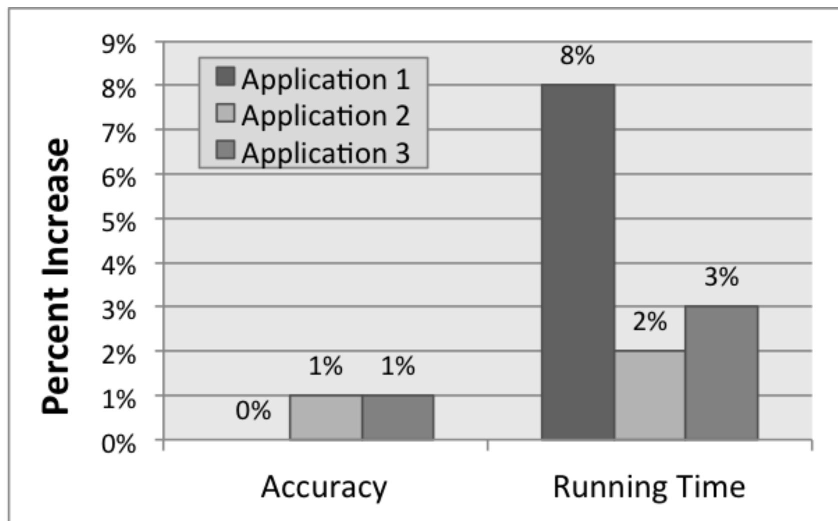

Figure 3.8: BraceForce's percent increase in accuracy and running time vs. Android alone

Question 3. In this experiment, I report results from using MDDA for a run of the smarthome application over three hours. I report results that examine the impact of MDDA at the aggregation level. That is, in the experiments I report in this section, I fix the frequency of data acquisition in the data layer and instead explicitly suppress updates from neighbors where the sensor readings differ by less than x% from the average of the neighboring readings. I vary x from 0.5% (the most sensitive) to 5% (the least sensitive). I report results for three sensitivity settings in

Fig. 3.9, which plots the tradeoff of MDDA with respect to accuracy and communication costs. While MDDA at the data layer is also useful to sensing driven CPS applications, for space considerations, I focus on the higher level MDDA because it has a more significant potential to impact communication costs since the reasoning at the aggregation layer requires network communication among distributed sensing devices.

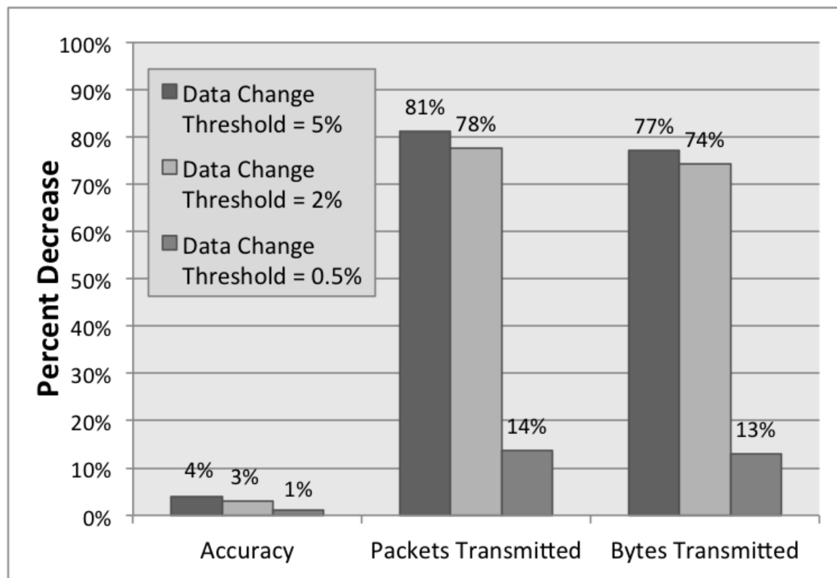

Figure 3.9: Tradeoffs of MDDA

As shown, there is a modest loss of accuracy when sensor readings are suppressed, but this comes at a substantial reduction in the communication overhead. I measure both the number of packets transmitted and the number of bytes transmitted because some communication is amortized over the entire run of the application. In my particular scenario, suppressing sensor readings that are within 2% of neighboring readings entails a only 3% loss of accuracy (in comparison to the out-of-band measured ground truth) but incurs 74-78% less communication overhead. These re-

sults are dependent on the particular model used and the ability of the phenomenon sensed to be predictably modeled. The results do show that, in applications where this is the case, MDDA whose sensitivity is tuned by the application domain expert can provide a significant benefit in terms of the cost to deploy or debug a CPS application that must acquire data from a networked set of sensors.

Discussion. I briefly discuss threats to validity.

Construct and Internal Validity. I have used two simple models for MDDA. I posit that using more sophisticated models will only bring more benefits to the suppression of superfluous transmissions of sensor data, but I have not validated this hypothesis. The application used to answer Question 3 was developed by the authors; since the focus of the evaluation for Question 3 is not on the usability of BraceForce but instead on the importance of MDDA (from a performance aspect), the particular developer should be irrelevant.

I use automated measures of CPU usage, WiFi usage, and communication latency to assess BraceForce. These statistics can be influenced by network noise, background threads, and other processes running on the test devices. To mitigate these concerns, I perform multiple trials and average values across the trials.

I use qualitative analysis of BraceForce simplicity based on Software Engineering and Computer Science students in a public university. The results may be different with professional CPS developers, however, my target is to enable *novice* programmers to build CPS applications, and my users represent these novice programmers. In my study's pre-questionnaire, the majority of my participants labeled their programming capabilities as "Average" (while five labeled their experience as

"Above Average"), and only one study participant had any prior experience in interfacing with sensing devices. Future studies will include both more experienced CPS developers and even more novice programmers.

I minimize learning effects by randomizing the order of the use of BraceForce and the Android Platform. For those participants who cannot complete the evaluation applications, I use the users' estimate on how long it would take them to finish the application. It is a guess at best and it might be different (e.g., more accurate) for professional CPS developers. But I find out in the study, for those participants whose programming capabilities as "Above Average", they tend to give much higher time estimation(e.g., five times more). From this, I have a hypothesis that more experience a developer has, more accurate the time estimation is; and those with average programming skills tend to give optimistic estimation. The hypothesis is not validated though.

External Validity. I have implemented the BraceForce prototype only for the Android operating system. I therefore cannot draw concrete conclusions about the external validity in terms of BraceForce's applicability to other operating systems and platforms. I have attempted to mitigate these concerns by avoiding proprietary interfaces whenever possible and, when not possible, wrapping those interfaces in a generic way that should be repeatable for other platforms and systems. The same is true for my use of external sensors connected via the Arduino board. While I have focused my prototype implementation on the Android and Arduino platforms, I have looked at a wide variety of devices, and my design and implementation has focused on abstractions that are, in theory, easily transferable to other domains.

Future work will include this translation to additional platforms as well as a novel MDDA tailored for the needs of the CPS applications.

3.6 Research Contributions

In this chapter, I made the following contribution:

Research Contribution 2: Create a supporting infrastructure to access sensors from heterogeneous platforms with energy efficiency and accessible programming interface. This work allows flexible debugging and deployment environment to access heterogeneous sensing capabilities required for CPS runtime verification.

3.7 Chapter Summary

I motivated the need for a generic and principled framework that allows for cross-platform integration of networked sensing capabilities with CPS applications. This is important in a variety of use cases, most notably to support developers of CPS applications that interact with on-board, external, and networked sensors and to support debugging of CPS applications that require direct interaction with the physical world. BraceForce provides a layered architecture that explicitly separates the application developer from the low-level complexity of interacting with sensing devices and enables the programming task to instead focus on the application logic and the integration of sensing into that logic. my evaluation demonstrated that BraceForce does indeed lower the barrier to creating these applications. Further, BraceForce also provides two essential points in the layered architecture to perform model-driven suppression of the sensing tasks, resulting in a significant reduction

in the cost of sensing (as measured in energy cost and communication cost) while mitigating the concomitant loss of sensing accuracy.

Chapter 4

BraceAssertion: Behavior Driven Development for CPS Application

Even with accessible sensing capabilities provided by *BraceForce*, however, developing and debugging CPS remain significant challenges. Many bugs are detectable only at runtime under deployment conditions that may be unpredictable or at least unexpected at development time. The current state of the practice of debugging CPS is generally *ad hoc*, involving trial and error in a real deployment. For increased rigor, it is appealing to bring formal methods to CPS verification. However developers often eschew formal approaches due to complexity and lack of efficiency. This paper presents *BraceAssertion*, a specification framework based on natural language queries that are automatically converted to a deterministic class of timed automata used for runtime monitoring. To reduce runtime overhead and support properties that reference predicate logic, I use a second monitor automaton to create filtered traces on which to run the analysis using the specification monitor. I evaluate the *BraceAssertion* framework using a real CPS case study and show that the framework is able to minimize runtime overhead with an increasing number of monitors.

4.1 Introduction

Cyber-Physical Systems (CPS) are found in applications in structural monitoring, autonomous vehicles, and many other fields. Compared with the growth of the domain, verification and validation of CPS lags far behind [194]. The state of the practice in debugging CPS generally entails a combination of simulation and *in situ* debugging. In a 2007 DARPA Urban Challenge Vehicle, a bug undetected by more than 300 miles of test-driving resulted in a near collision. An analysis of the incident found that, to protect the steering system, the interface to the physical hardware limited the steering rate to low speeds [133]. When the path planner produced a sharp turn at higher speeds, the vehicle physically could not follow, and this unanticipated situation caused the bug. The analysis concluded that, although simulation-centric tools are indispensable for rapid prototyping, design, and debugging, they are limited in providing correctness guarantees. State of the art formal methods tools, including static analysis, theorem proving, and model checking, are insufficient in tackling the challenges in CPS verification and validation [194]. Other verification techniques, including model-based testing [41] and simulation [81] have high learning curves, impractical development costs, and scalability issues. Domain specific tools (e.g., passive distributed assertions [163] and symbolic execution [166]), though more scalable, fail to formally verify either qualitative constraints (e.g., the ordering of events), quantitative ones (e.g., timing), or both. *Ad hoc* debugging has become the *de facto* standard for debugging CPS, though it suffers tremendously from a lack of robustness [194]. More formal runtime verification is a perfect candidate to identify subtle errors that are otherwise hard to capture due to scalability

(state explosion) or unexpected interactions with physical environments. Moreover, on-line monitors (e.g., JavaMOP [42]) can react to errors in actual executions, which is essential for CPS applications.

Runtime verification requires correctness properties to be written in a formal specification language. In addition to many other characteristics, every CPS is a real-time system; therefore formal specifications of CPS must capture real time properties such as event ordering, timeout, and delay. Temporal logics and first/higher order logic have been used to specify concurrent and real-time programs. However, CPS practitioners tend not to use formal specification because of the steep learning curve and a lack of reliability [37]. The use of informal models as a transition has been recommended [67], and Behavior-Driven Development (BDD) tools are gaining popularity among CPS developers [194]. This paper introduces *BraceAssertion*, a BDD-style specification language that is accessible to CPS developers yet expressive enough to represent a determinizable class of Timed Automata [9] and to support predicate logic. I also design a monitor framework to automate verification based on BDD specifications. My concrete contributions are:

- I bring Behavior-Driven Development to Cyber-Physical Systems (CPS) to enable formal specification of correct system behaviors using natural language.
- I create the *BraceAssertion* language, which extends BDD to support the expressiveness of deterministic timed automata and adds support for predicate logic to cater for complex requirements of CPS applications.
- To support BDD-style specification, I implement an efficient dual monitor archi-

tecture, consisting of a synthesized *event monitor* that generates filtered traces and a synthesized *timed automata monitor* to verify quantitative properties on traces.

◦ I provide real-world case studies to evaluate the effectiveness and efficiency associated with the synthesized monitors derived from *BraceAssertions*.

4.2 Motivation and Overview

In this section, I introduce a motivating CPS application and the research challenges therein. I then provide background information on Behavior Driven Development (BDD), my formal framework, and the foundational logics.

4.2.1 Motivating Application

My work can be easily motivated by agent-based CPS [134], where the system's concurrency and distribution are handled by each agent, so verification can be localized to each agent. These systems require a runtime verification framework with high expressiveness, intuitiveness, and with low runtime overhead. I use this example throughout the paper, though the *BraceAssertion* framework is applicable to CPS in general.

Consider a multi-agent vehicular patrol application with a set of unmanned vehicles that coordinate to achieve a global monitoring task. As part of the cooperative exercise, each vehicle's agent develops a schedule of tasks to execute. Such an application may specify the following three constraints: (1) if a vehicle is selected as responsible for a waypoint, its schedule will eventually contain a task to reach

that waypoint (*Spec 1*); (2) a vehicle will reach the locale of each selected waypoint before a specified deadline (*Spec 2*); and (3) the choice of which vehicle to perform each waypoint task is optimal relative to a chosen system utility function (*Spec 3*).

Spec 1 & 2 require qualitative constraints (e.g., those that reference the ordering of events) and quantitative constraints (e.g., those that reference timeouts and bounds on response times). JavaMOP [42], a state of the art runtime verification framework, is not able to capture these constraints. Instead, capabilities like those of Metric Temporal Logic (MTL) [105] and Metric Interval Temporal Logic (MITL) [8] are required. *Spec 3* is more subtle and actually requires a predicate logic whereby each waypoint task can be quantified and a predicate function can evaluate whether a chosen utility function is optimal. Existing runtime verification techniques based on metric temporal logic cannot capture this expressiveness. Recent work on Metric First-Order Temporal Logic (MFOTL) [21] can, but the complexity and lack of binding between the specification and the implementation make it unwieldy and impractical for CPS practitioners. An even more challenging requirement is to minimize the runtime overhead for the application, since CPS are usually deployed to resource constrained platforms; this challenge remains open.

In summary, the motivating application requires a specification that is intuitive to specify, provides a tight binding to the implementation, can express quantitative requirements and a predicate logic, and incurs low runtime overhead. These combined research challenges push us first to look at a widely accepted and intuitive industrial specification, Behavior Driven Development (BDD), on which my work is based.

4.2.2 Behavior Driven Development

Behavior Driven Development (BDD) was created to clarify common misunderstandings in Test-Driven Development related to what to test, what not to test, how much to test, and how to understand why a test fails [26]. However, formal semantics are still lacking for BDD. A BDD specification typically starts with a story template: *As a* [X], *I want* [Y], *So that* [Z] and is further refined into multiple test scenarios of the form: *Given* [initial condition], *When* [events occur], *Then* [ensure some outcomes]. As an example, a correctness specification for the push operation on a *Stack* data structure would be *Given* [A stack is not full] *When* [an element is added to the stack] *Then* [that element is at the top of the stack]. Each fragment of a test scenario has to be associated with segments of program code using inheritance or as a *plug-in* to the programming language.

Intuitively, to give formal semantics to the *Given-When-Then* template, I can treat the template as a state transition in a finite automaton. *Given* specifies in which states the transition is enabled, *When* specifies which input signals or events trigger the transition, and *Then* specifies what actions to take and which state(s) to move to. The popularity of BDD reflects the willingness of developers "in the wild" to accept a less formal specification language. However, state of art frameworks in BDD (e.g., Cucumber, JBehave[1]) ignore quantitative constraints such as timing, qualitative ones such as ordering of events (*happens-before*), quantification ($\exists$ and $\forall$), and predicates (as in first-order logic), which are all crucial in specifying CPS

[1] `cukes.info`, `jbehave.org`

applications. my goal is to support CPS applications by filling the gap between BDD (which is reasonably accessible to CPS developers) and formal methods (which are not). Instead of asking developers to write the underlying temporal logic formulae directly, I create *BraceAssertion*, a natural description language in BDD style that allows developers to capture a correctness specification's essential semantics and to annotate the CPS application to connect the implementation to the specifications. My monitor synthesis algorithms can automatically generate runtime monitors that are timed automata obtained from the textual description.

4.2.3 My Basic Formal Framework

Because CPS applications require reactions to timeouts or incoming events, my models must consider a dense time domain, for which I use non-negative real numbers. The *implementation* of such a time domain requires digital clocks; I assume that local clocks are sufficiently synchronized (i.e., that the worst case drift is below a very small and acceptable σ); this assumption is achievable using established clock synchronization algorithms [65, 131].

I model 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, events' 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 at which each event is observed. The timing sequence respects *monotonicity* and *progress* ($\tau_i < \tau_{i+1}$ and $\exists i \in N, \forall j \in R, \tau_i > j$).

This basic framework underpins the *BraceAssertion* language, which captures both qualitative and quantitative constraints and identifies constraint violations by considering captured timed traces of the system's execution. Event Clock Automata (ECA) [9] has many of the semantic capabilities required for modeling quantitative properties of CPS. I next provide a brief introduction to ECA and the associated State Clock Logic (SCL) [160], which is used to express ECA, and relate their capabilities to the goals of *BraceAssertion*.

4.2.4 ECA and SCL

Timed automata [7] are finite automata extended with real-time clocks. In timed automata [7], one can annotate state transitions with timing constraints using real-valued clock variables. The constraints on clocks can specify time intervals e.g., a given event eventually happens or happens before a deadline. Contrary to standard automata, timed automata are not always determinizable and thus difficult to use directly for runtime verification. Event Clock Automata (ECA) [9] restrict the use of the clock and thus embody a determinizable class of timed automata. In ECA, for each event, a *recording clock* records the time of the last occurrence and a *predicting clock* predicts the time of the next occurrence. An ECA is in the form $A = (\Sigma, L, L_0, L_f, E)$, where Σ is a set of symbols, L is a set of states, L_0 represents a single start state, L_f represents a set of accepting states, and E is a set of edges representing transitions. Two edges with the same source and the same input must have mutually exclusive clock constraints to preserve determinism. Notice that finite automata are ECA with no clocks.

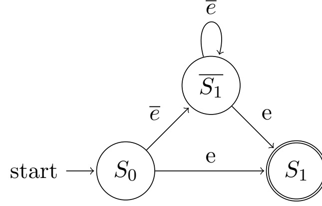

Figure 4.1: Specification in ECA: *If a vehicle is selected for a waypoint, its schedule eventually contains that waypoint.*

Fig. 4.1 shows how *Spec 1* can be expressed using a finite automaton. S_0 refers to the *initial* state after the agent was assigned to reach the given waypoint. S_1 is the *accepting* state, where the agent's schedule contains the task to reach the waypoint. The event e adds the waypoint to the agent's schedule. $\overline{S_1}$ represents the set of states reachable from S_0 in which the waypoint is not in the agent's schedule; $\overline{e}$ represents any event that does not insert the waypoint into the agent's schedule. This very simple automaton accepts any trace in which the desired event *eventually* happens, which is no more or less than required by the specification. Fig. 4.2 shows how the quantitative properties *Spec 2* can be represented with ECA. Here, in the *accepting* state S_1, the agent has reached the waypoint before a deadline. The timing constraint x_a associated with the edge from S_0 to S_1 ensures that each r (which refers to the event that the agent has reached the waypoint) occurs within τ time units of the *preceding* a (which refers to the event that the agent was assigned to reach a waypoint).

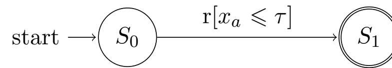

Figure 4.2: Specification in ECA: *A vehicle will reach the locale of the waypoint within a bounded time τ.*

As a *de-facto* standard, State Clock Logic (SCL) is used to express ECA [160]. However, there remain characteristics of CPS that limit the usefulness of SCL to

runtime verification. Consider *Spec 2*; the event that causes the transition to the final accepting state is associated with an action "enter the waypoint." Such methods in CPS interact with the environment in ways that are not possible to capture in SCL. In my example, the nature of the physical space (e.g., indoors versus outdoors), or environmental conditions (e.g., wind) and their impacts on the system's correctness must be considered. As a simple example, the vehicle must have an auxiliary procedure that *detects* that it has reached the waypoint with some guarantee; this procedure generates the event r that causes the state transition in Fig. 4.2. Clearly this requires the specification to be expressive enough to *locate* this procedure in the implementation, *quantify* the waypoint, and *evaluate* the procedure at the *right* time, none of which is available in SCL (or in temporal logics in general). This limitation of SCL applied to CPS is even more striking for *Spec 3*. This constraint cannot even be specified in SCL because SCL does not have the ability to associate utility with events. The demand of CPS applications for a more powerful logic to express such constraints, along with the low acceptance of formal specification in general [184] and by CPS developers especially [194] motivates us to create the *BraceAssertion* specification language on top of ECA.

4.3 BraceAssertion

My *BraceAssertion* language is expressive enough to specify quantitative and qualitative constraints that characterize behaviors of CPS applications. At the same time, it builds on the simplicity and abstraction level of Behavior Driven Development (BDD) frameworks to make it more accessible to CPS developers. BDD

is based on natural language and provides tighter integration between specification and implementation through customized annotations in the code. In this section, I introduce the essential syntax of *BraceAssertion* and show how to express standard real-time constraints. I then define the formal semantics in terms of SCL formulas.

4.3.1 BraceAssertion Language

Qualitative Constraints. In existing BDD approaches, support for specifications that reference the order of events is minimal. Further, providing specifications in the usual BDD form e.g., *Given* [initial condition], *When* [events occur], *Then* [ensure some outcome], only considers instantaneous (logical) state transitions, which are unlikely in CPS applications considering delays in responses from physical devices and the environment. *BraceAssertion* augments the BDD language with the capability to define the system as an aggregate of individual components. *BraceAssertion* then allows associating automata (monitors) with each component to enable distributed runtime verification at the level of each component. To connect this with BDD, I change the BDD story template "*As a* [X]-*I want* [Y]-*So that* [Z]" into "*In [S]-As a* [X]-*I want* [Y]-*So that* [Z]" where S is a component for the story. A story is associated with one or more BDD specifications in the *Given*-*When*-*Then* form[2]. *BraceAssertion* also adds a *When-Then-Else* construct to define alternative state transitions from a *Given* state and *When* events. As a concrete example, using these new qualitative constraint constructs, *Spec 1* can be written as "*In* [Agent], *Given* [All] *When* [agent was chosen to reach a waypoint] *Then* [Eventually its

[2]Following BDD convention, nesting of the BDD constructs is not allowed. Users can always use separate BDD specifications instead.

scheduler contains the task to reach that way point]." I also extend the BDD *Then* fragment to support four qualitative operators: *Always*, *Eventually*, *Eventually Permanent*, and *Never*, which correspond, respectively, to the linear temporal logics operators $\Box$, $\Diamond$, $\Diamond\Box$, and $\neg\Box$ (see [160] for the timed version of the operators).

Quantitative Constraints. While timing constraints are essential in CPS applications and in theory expressible in SCL, BDD lacks the semantics to annotate state transitions with timing constraints such as specifying deadlines. To bridge this gap, I add new constructs and keywords *Within*, *Exactly*, and *More Than* in a BDD *When* fragment. Each construct and its associated time value provides a hard deadline constraint on the timing of the event clause. This enables us to easily specify standard timing constraints in CPS. As an example, the *bounded time response* constraint "a p event is always followed by a q event within k (time units)," is expressed in *BraceAssertion* as: "*Given* [p] *When* [Within k (time units) After [p]] *Then* [q]." The BraceAssertion to specify an *exact response time* is similar but uses the "Exactly" keyword in place of the "Within" keyword. A *timeout* specification can be given by a *BraceAssertion* of the form: "*Given* [All] *When* [Exactly k (time units) After q] *Then* [p]." Triggering an *alarm* can be specified by using a negated guard and the "*Within*" keyword. Finally, constraining the *minimal time interval between two events* can be specified by a *BraceAssertion* in the form: "*Given* [All] *When* [More Than k (time units) Before p] *Then* [q]." As a concrete example, *Spec 2* can be written as: "*In* [Agent], *Given* [chosen to reach a waypoint] *When* [Within xx time units After] *Then* [reach the locale of that waypoint]."

Predicate Logic. To tie specifications of correctness properties to the implementation, developers using BDD associate every event to a method signature. The BDD specification treats the method signature as a propositional expression i.e., the evaluation of "has the method A been executed?". In CPS, however, it is insufficient to bind events only to method invocations. CPS applications require events to be associated with a variety of additional system aspects (e.g., thread safety checks against specific data structures), but most importantly and uniquely, elements of the physical environment (e.g., validation against sensor values). For instance, consider *Spec 1* once again. The text description in the *Then* clause actually requires predicate logic because the specification existentially or universally quantifies over tasks. To handle such complexities, CPS correctness specifications require predicate logic (e.g., first order logic). To continue to support the tight integration between the specifications and the implementation in BDD, I (1) add two new constructs and keywords to the *Given-When-Then* structure in the BDD specification, (2) I define additional semantics in the BDD *Then* fragment, and (3) I create additional BDD annotations for the implementation.

The two new constructs based on the keywords *With* and *And*, allow the creator of the specification to indicate parameters to the logical predicate contained within the *Then* clause. The three new BDD annotations connect the predicate provided in the extended *Then* clause to the implementation. For instance, the quantification of the parameters in a predicate (i.e., the task and schedule in my first specification) relies on a BDD annotation created specifically for this purpose. More generally, Table 4.1 lists my additional BDD annotation classes: *Event*, *Predicate*,

Param, and *Execution*.

Table 4.1: New BDD Annotations

Annotation	**Description**
Event	bind an event to a method invocation or a single statement
Predicate	reference a boolean function
Param	identify quantified variables
Execution	identify execution place for a predicate

The following code snippet shows how the implementation reflects the newly introduced annotations. The developer uses the *Predicate* annotation to associate a boolean function *checkTaskOptimal* with the predicate ("check schedule is optimal"). The developer provides the implementation of the function, which is standard practice in BDD. The developer uses the *Param* annotation to locate the variable associated with the quantified parameter ("the task"), while the *mode*'s value of *List* indicates that the variable's quantification is ∀, i.e., each instance is verified against the predicate. Alternatively, the *mode* can be assigned *Single* to specify ∃, where the latest instance is verified against the predicate. Finally the timing for predicate evaluation is determined by the *Execution* annotation (in this case, after execution of *CalculateSchedule*).

```
@Predicate(name="check schedule is optimal")
public boolean checkTaskOptimal(Task task){
  //developer's implementation of predicate check
}
//somewhere else...
@Param(name="task", variable="t", mode=List)
@Execution(name="check schedule is optimal", mode=ExecutionMode.After)
public Schedule CalculateSchedule(Task t){
  //developer's implementation of a piece of logic
  //from the actual CPS application
}
```

Enabling support for predicate logic in *BraceAssertions* includes adding support for these new keywords and annotations. I omit the details for brevity, but I accomplished the integration of this support via non-trivial engineering efforts based on aspect-oriented programming (e.g., through AspectJ) and some data structure manipulation. The complete *BraceAssertions* syntax is given in Chapter. A.

4.3.2 BraceAssertion Formal Semantics

The formal semantics of *BraceAssertion* is given in terms of SCL formulas. Each *BraceAssertion* is translated into an SCL formula according to the rules of Table 4.2, where ϕ, ϕ_1, ϕ_2 are predicate logics formulas (with no timing constraints), $\backsim \in \{<, \leq, =, >, \geq\}$, and c is an integer. The syntax and formal semantics of SCL over timed traces are given in Chapter. A. By providing a translation (Table 4.2) I provide a formal semantics to *BraceAssertion*. My translation into SCL provides a correct-by-construction algorithm to build monitors to check *BraceAssertion* specifications. Indeed, one the result of [160] is that, for any ϕ in SCL, an ECA A_ϕ can be constructed that accepts exactly the timed traces that satisfy ϕ. As any *BraceAssertion* specification f is translated in an SCL formula ϕ_f, the ECA A_{ϕ_f} thus accepts exactly the timed traces defined by f.

4.4 Dual Monitor Architecture

One of my main concerns is to establish a tight integration between specification and implementation. At the conceptual level, I accomplish this through the annotations in the BDD-based *BraceAssertion*. From a practical perspective, I lever-

Table 4.2: Formal Semantics of BraceAssertion

BraceAssertion	**SCL**
Given ϕ_1 *When* After *Then* ϕ_2	$\phi_1 U \phi_2$
Given ϕ_1 *When* Before *Then* ϕ_2	$\phi_1 S \phi_2$
When $\backsim c$ Before ϕ	$\rhd_{\backsim c}\ \phi$
When $\backsim c$ After ϕ	$\lhd_{\backsim c}\ \phi$
When $\backsim c$ After ϕ_1 *Then* ϕ_2	$\phi_1 \rightarrow \rhd_{\backsim c}\ \phi_2$
When $\backsim c$ Before ϕ_1 *Then* ϕ_2	$\phi_1 \rightarrow \lhd_{\backsim c}\ \phi_2$
Then Always ϕ_1	$\Diamond \phi_1$
Then Eventually ϕ_1	$\Box \phi_1$
Then Eventually Permanent ϕ_1	$\Diamond \Box \phi_1$

age *Aspect Oriented Programming* (AOP) to effectively insert behavior-augmenting pieces of code based on the annotations; this code is used to check the programmer specified properties. Using AOP, I devise a dual monitor architecture shown in Fig. 6.1. My AOP-based approach allows *BraceAssertion* correctness properties to include complex logics (e.g., predicates and quantification) that can be resolved using *pointcuts* in AOP. Further, the architecture allows for a separation of concerns related to event maintenance and monitor execution: an *Event Monitor* uses the BDD annotations to generate a filtered event trace that can be analyzed by runtime monitors, which explicitly *monitor* the run-time state to check a specified property during program execution. Because the *BraceAssertion* annotations give us the *pointcuts* for parameters, predicates, and points of execution, the Event Monitor can weave them together with the source code of the CPS application and generate only the needed (aggregated) events that result from evaluating each predicate at runtime. *BraceAssertion*'s support for customized predicates makes my framework more flexible than the state of the art in runtime monitoring [21,22], which constrain

monitoring to a set of predefined predicates.

As an overview of the entire process of employing *BraceAssertion*, a CPS developer creates a system specification by defining *BraceAssertion* specifications and annotating the program in BDD style. My framework synthesizes two monitors from each specification[3]. I first synthesize an Event Monitor that generates *point-cuts* and *advice* [86], monitors underlying events at runtime, and generates *filtered* execution trace. I then synthesize an ECA monitor to verify system correctness based on collected execution traces.

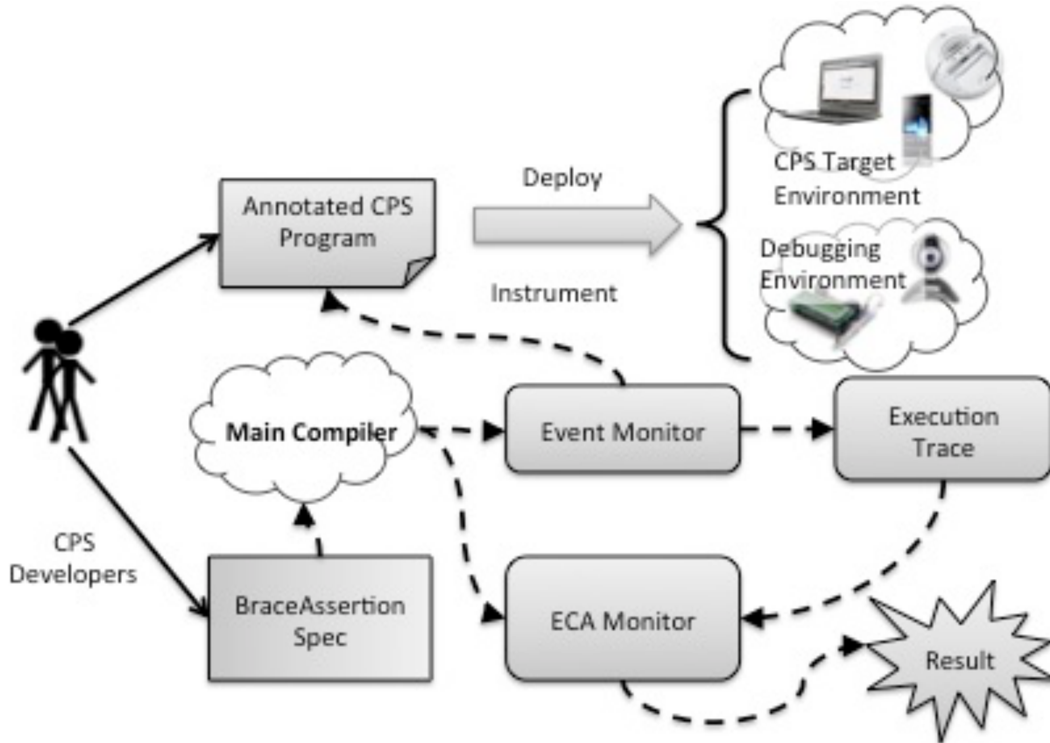

Figure 4.3: The Dual Monitor Architecture

[3]I omit the monitor synthesis algorithms, see Chapter. A for details.

4.4.1 Event Monitor

The event monitors synthesized from the BDD specifications instrument the annotated program by injecting AspectJ pointcuts which at runtime feed "raw" signals to event monitors to generate filtered timed traces. My creation of the event monitor is driven by performance limitations of existing runtime verification approaches and by scalability challenges that emerge in attempting to directly implement *BraceAssertion*s. First, as shown in Fig. 4.1, a *BraceAssertion* can refer to the negation of an event. Every time an event that is *not e* occurs, an event $\overline{e}$ must be output to the trace. Second, the *Given-When-Then* notation specifies events (and not states) [26]. A naïve implementation of *BraceAssertion* requires generating an internal state for each unique event and an aggregate state for every set of events within a given structure (e.g., I require an aggregate state for the multiple events specified in a *BraceAssertion*'s *When* clause). A complete transition table must consider all possible permutations of internal states (including aggregate states). In addition, my BDD-based approach requires instrumenting the implementation to support the semantics of BDD annotations, which can also include predicate logic. Such predicates must be dynamically evaluated at runtime. For all of these reasons, I synthesize an *Event Monitor* from a given *BraceAssertion* specification. This synthesized monitor allows the *BraceAssertion* framework to (1) create *pointcuts* in the implementation through instrumentation and (2) monitor events at runtime and generate *filtered* traces over which correctness specifications can be checked. For brevity, I omit the main algorithm the *Event Monitor* uses to instrument the program; refer to Chapter. A for details.

At runtime, the injected pointcuts pass events to the *Event Monitor* to create expressive runtime traces. The *Event Monitor* can filter/generate four types of events from input atomic events: (1) clock events; (2) single events; (3) complemented events; and (4) aggregate events.

Algorithm 1 Event Filter (FILTER)

1: **if** $e.isClockEvent \vee e.isEvent$ **then**
2: output(e)
3: **for** $\forall ce \in e.ces$ **do**
4: FILTER(ce)
5: **for** $\forall em \in eventMonitorStore(e.id)$ **do**
6: $em.setTransition(e.id)$

Algorithm 1 shows the basic filter algorithm, which filters atomic events (e) passed from the instrumented execution. If the atomic event is a clock event or single event (e.g., with only one event with one *When* clause), the algorithm outputs the event to the trace (lines 1-2). If the input atomic event has any complemented events, I recursively apply the algorithm to the complemented events instead of outputting them directly (lines 3-4). Each *Event Monitor* is essentially a state machine that keeps track of a list of atomic events and makes a state transition upon receiving an event. If the input atomic event is associated with any *Event Monitor*, the algorithm invokes *setTransition* of the *Event Monitor* to check whether all the internal events for the *Event Monitor* have been detected and, if so, outputs the aggregate event for the *Event Monitor* (lines 5-6). The *Event Monitor*s are therefore essential in limiting the scale of the generated event traces.

4.4.2 ECA Monitor

Given a *BraceAssertion*, I automatically construct an ECA that checks whether the specification is violated. While similar approaches are non-trivial [160], my synthesis algorithm is straightforward and efficient since the *BraceAssertion* is already a textual description of an ECA. For instance, from the *Given-When-Then* specification, I can extract a set of transitions $\{E\}$ together with initial and accepting states. Checking whether a trace violates a specification amounts to checking whether the ECA accepts the trace. My verification algorithm is based on a standard decision algorithm for testing membership of a trace in a regular language [91]. Testing membership of timed traces for general timed automata is NP-complete [10], but, as I prove below, because I restrict ourselves to a subclass of timed automata that are deterministic, testing membership can be done in polynomial time.

I also introduce two major enhancements. The first one is essential in dealing with the recording and predicting clocks for each *BraceAssertion*. Because I evaluate specifications over trace files, I handle the predicting clock using a *look ahead* algorithm that relies on a concurrent queue to access the future in the trace file. A *producer* reads from the trace file and fills in the concurrent queue, while a *consumer* reads one timed word after another as input to the ECA monitors. The length of the queue is (lower) bounded by the maximum clock constraint across all specifications. The length of the queue also reflects another parameter specifying a minimum number of timed words in the queue (to minimize the overhead of context switching when the buffer is too small). When a predicting clock is referenced, the ECA monitor can look ahead in the queue to check a constraint.

My second enhancement is essential in handling the fact that there may be a large number of *BraceAssertion* specifications for even a modest system, and I need a scalable solution for checking the correctness of these specifications. I use *lazy initialization* to activate an instance of a monitor only when the monitor's initial event is detected, and I terminate the monitor instance as soon as I reach an *accepting* (specification passed) or *rejecting* (specification violated) state. This minimizes the number of active monitors. Each timed word can be consumed by only one instance of a particular monitor but can be shared among instances of other monitors (to allow parallel processing). At any point in the process, there are three possible values for each ECA monitor: *accepting*, *rejecting*, or *undetermined* (if the monitor is still active).

In Chapter. A, I provide the complete ECA monitor synthesis algorithm. In essence, since SCL is used to represent ECA [160] and *BraceAssertion* formal semantics are expressed using SCL, the synthesis algorithm is quite straightforward. The validity of the monitor synthesis algorithm has been demonstrated exhaustively via hundreds of synthesized timed traces.

4.4.3 Combinatorial Analysis

Because one of my goals is to reduce the overhead of runtime monitoring, I perform a combinatorial analysis of the Event Monitor and the ECA Monitor.

Lemma 4.4.1. *The time cost for an Event Monitor to output one event in the event trace is $O(|e| + |p|)$, and the storage cost (for all events) is $O(|e|)$, where $|e|$ is the number of events and $|p|$ is the number of parameters in the BraceAssertion.*

Proof. The atomic events are generated from the pointcuts. The *BraceAssertion* framework simply passes the atomic events or parameters directly to the Event Monitor. The Event Monitor filters and generates the required events, without any other logic; therefore the runtime cost is $O(|e| + |p|)$. My analysis of the Event Monitor is based on Algorithm 1. From lines 1-2, checking whether an input event is a clock event or a single event is $O(1)$ because the Event Monitor uses hashtables[4] to register the types of events. Similarly, from lines 5-6, the Event Monitor uses a hashtable to register all Event Managers, which in turn keep track of corresponding atomic events. Since each Event Manager has a maximum $O(|e|)$ events stored, the time cost is $O(|e|)$. Considering the above, the combined time cost for all events is $O(|e|)$; considering also lines 3-4, as each event has a maximum of one complemented event, the recursive function is called at most once for each event. So the overall time cost for each event (combined with the event generation) is $O(2 \cdot |e| + |e| + |p|) = O(|e| + |p|)$. During the synthesis process, an Event Monitor stores a number of auxiliary data structures (all based on hashtables), each of which has storage cost of $O(|e|)$. The number of these auxiliary data structures is constant, so the overall storage cost for all events is $O(|e|)$. □

Lemma 4.4.2. *The space complexity of the synthesis algorithm (i.e., the size of the ECA monitor) is $O(n \cdot |e|)$, where $|e|$ is the number of events, and n is the number of specifications.*

Proof. In the ECA monitor synthesis, I reuse a shared state transition table, which

[4]I get constant time performance using efficient hashing [146].

has maximum number of transitions of $O(|e|)$. For each monitor, I also record accepted and rejected states. The total storage cost is therefore $O(n \cdot |e|)$. □

Lemma 4.4.3. *The time complexity of the offline membership test is* $O(|t| \cdot |e| \cdot |c|)$*, where* $|t|$ *is the number of timed words in the trace file,* $|e|$ *is the number of events, and* $|c|$ *is the number of clock variables (constraints).*

Proof. My work is based on the membership test algorithm in [91]. The only difference is instead of checking words, I check timed words. So in the worst case scenario, for each timed word, I must traverse all state transitions ($|e|$) in the global transition table and check all clock constraints ($|c|$). In practice, the behavior of my monitor will be close to $O(|t|)$. Traversing each state transition is replaced by querying a hashtable for the transition records with the beginning state of the current state recorded in each ECA monitor (constant time on average), and there are a constant number of clock constraints per specification, thus on average the time cost can be reduced to $O(|t|)$. □

I implemented a Java prototype of my dual monitors and tested the semantic correctness on synthetic traces. Since my main contribution is to use BDD to *efficiently* and *effectively* bring ECA and first order logic into CPS development, I conducted an empirical study on a real multi-agent patrol system to analyze effectiveness and efficiency.

4.5 Case Study and Evaluation

I evaluate the *BraceAssertion* framework using a case study application[5] that existed before the creation of *BraceAssertion*; this is the application I have used for examples.

4.5.1 The Case Study Briefly

I used an existing robot planning system, which is a distributed version of Generalized Partial Global Planning (GPGP) [116]. This system's planning algorithm is a version of Anytime A* that is customized to distributed planning for a group of mobile vehicles. A group of vehicles is assigned patrols that must visit a set of specified waypoints. The vehicles negotiate, and each derives a schedule that contains a subset of the waypoints. The schedules are chosen to optimize the combined utility of the vehicles. Each vehicle hosts an intelligent agent that can optimize its actions, autonomously and interactively. Each agent includes a local scheduler, which derives a schedule based on a set of tasks assigned for execution; a negotiator, which coordinates with other agents to derive the schedule; and an execution system. To accomplish a global task, agents negotiate over multiple attributes. In this paper, I used a deployment instance in which two vehicles cyclically move along their generated waypoint sets. The utility of visiting a waypoint can change dynamically, which may change the agents' schedules.

[5] I refer readers to the complete case study in Chapter. A

4.5.2 Research Questions (RQs)

My evaluation answers the following research questions:

RQ1: How efficient is the Event Monitor in generating a filtered event trace (in terms of CPU, memory overhead, and the size of traces generated)?

RQ2: How effective is *BraceAssertion* in detecting runtime violations (e.g., to capture injected errors and, even better, to detect real bugs)?

RQ3: How efficient is the ECA Monitor in detecting runtime violations using the filtered event trace, and how do the features of the ECA Monitor help to improve efficiency?

4.5.3 Experiment Design

To answer these questions, I use my Java prototype and my case study application. I use the patrol application to generate traces as described in Section 6.3, and I check those traces for the three properties from Section 6.2:

- "*In* [Agent], *Given* [All] *When* [agent chosen to reach a waypoint] *Then* [Eventually its scheduler contains the task to reach that way point]." (*Spec 1*)
- "*In* [Agent], *Given* [agent was chosen to reach a waypoint] *When* [Within 40 seconds After] *Then* [reach the locale of that waypoint]." (*Spec 2*)
- "*In* [Scheduler], *Given* [All] *When* [a task is added] *Then* [check schedule is optimal With the task]." (*Spec 3*)

The application developer must annotate application code with hooks for *BraceAssertion* specifications. The code below shows how this happens for the *When* event in *Spec 1*. Using the *BraceAssertion* library, which contains customized Java annotation classes, the developer uses the *Event* annotation to assign the *name* field to the event name ("agent was chosen to reach a waypoint"). This is the only step required to connect an event in *BraceAssertion* to the implementation.

```
//somewhere in the Agent class...
@Event(name="agent chosen to reach a waypoint")
public void assignTask(Task task){
    // Developer's implementation when a static
    // task to reach a waypoint is assigned
}
```

I evaluate *BraceAssertion*'s ability to detect injected violations in traces, and I benchmark its performance. In this process, I also found violations in the patrol implementation other than those I injected. All of the experiments were performed on a PC with an Intel i5 CPU (2.30GHz, 2 cores 4 threads) and 4GB RAM. I control the size of an instance of the problem by adjusting the number of waypoints visited by the agents. For each of the three specifications, the Event Monitor monitors the atomic events and generates the required aggregate events in the trace.

I report results for the following experiments:

Baseline: I run the original application with an increasing number of statically determined waypoints: 6 (default), 48, 384, and 3072[6]. The last situation is a extreme one that requires monitoring recurring tasks at a very high frequency; in

[6] A monitor is activated when a waypoint is assigned. I increase the waypoints to increase the number of concurrent monitors.

normal situations, events generated for checking specifications have a much lower frequency.

Experiment 1 (*E1*): I re-run *Baseline* with the application annotated and instrumented with *Spec 1*. I randomly inject errors to exercise *Spec 1*.

Experiment 2 (*E2*): I re-run *E1* with the application also annotated and instrumented with *Spec 2*. I randomly inject errors to exercise both specifications.

Experiment 2-Wild (*E2-Wild*): I run the original application annotated and instrumented with a version of *Spec 2* that incrementally tightens the timing constraint.

Experiment 3 (*E3*): I run the original application annotated and instrumented with the third specification using an increasing number of dynamically determined tasks (i.e., waypoints): 6 (default), 32, and 64. I randomly inject errors to exercise the specification.

Experiment 4 (*E4*): I use the trace file from *E2*'s largest test and multiply it by 10 (i.e., pasting ten copies of the trace back to back). I then synthesize increasing numbers of arbitrary specifications (i.e., with arbitrary events for the *Given*, *When*, and *Then* clauses of a *BraceAssertion*): 3 specifications (default), 24, 192, and 1536.

Experiment 5 (*E5*): I use the same scaled trace file, using ECA monitors to verify the trace file while incrementally increasing the size of the trace buffer.

I report average values for CPU usage and memory consumption using VisualVM[7]. Results are averages of 5 runs.

[7] `http://visualvm.java.net/`

4.5.4 RQ1: The Efficiency of the Event Monitor

To report CPU usage and memory consumption, I compute the percent increase in comparison to *Baseline*, e.g., for *E1*, I compute $\frac{E1 - Baseline}{Baseline}$.

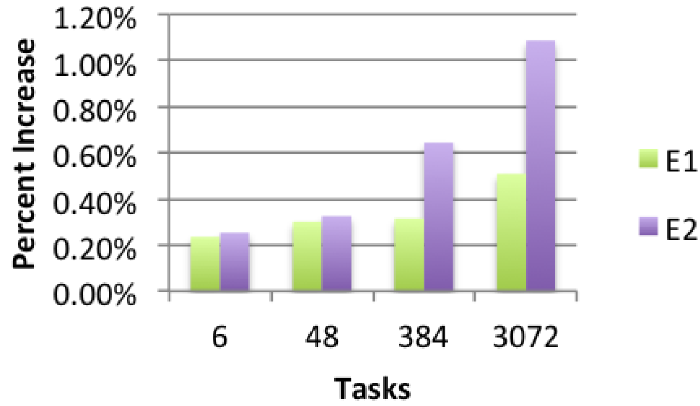

Figure 4.4: CPU Overhead

Fig. 4.4 shows that the CPU overhead of running the Event Monitor alongside the application is minimal, with a 0.23% increase in CPU usage for *E1* with 6 tasks and 0.25% for *E2* with 6 tasks, growing to only 1.09% for *E2* with the largest test set. The results show that using the Event Monitor to generate runtime traces incurs a trivial performance overhead for the monitored CPS application.

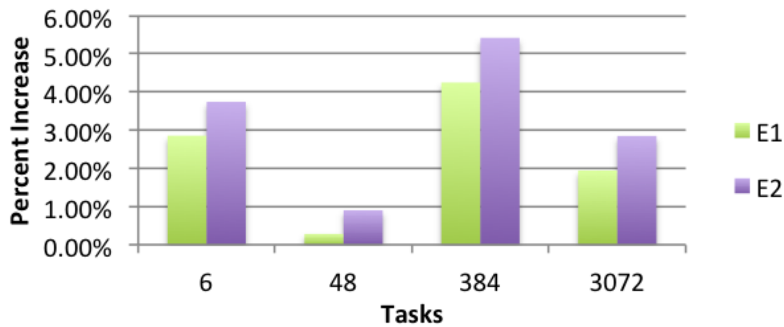

Figure 4.5: Memory Overhead of the Event Monitor

Fig. 4.5 shows the Event Monitor's memory overhead. The memory usage in E1 shows that adding one specification to monitor high-frequency events adds only less than 1% overhead. In both *E1* and *E2*, the memory overhead does not always increase with the size of the test sets. This results from my use of a lock-free concurrent queue for predicate parameters with ∀ quantification (e.g., *Spec 1* checks "for all" way points); in this implementation, the size of the queue depends on the timing of evaluating the predicate (e.g., "its scheduler contains the task") and some optimization parameters defined for the concurrent queue, which causes the observed discontinuities.

I also measured the size of the trace files generated as a measure of the *BraceAssertion* overhead (Fig. 4.6). Each task requires at least two timed words in the trace file. Even in the largest test set, the sizes of the trace files are 145.2 and 218.6 KB for *E1* and *E2*, respectively, which is a very reasonable size for the ECA monitor to process.

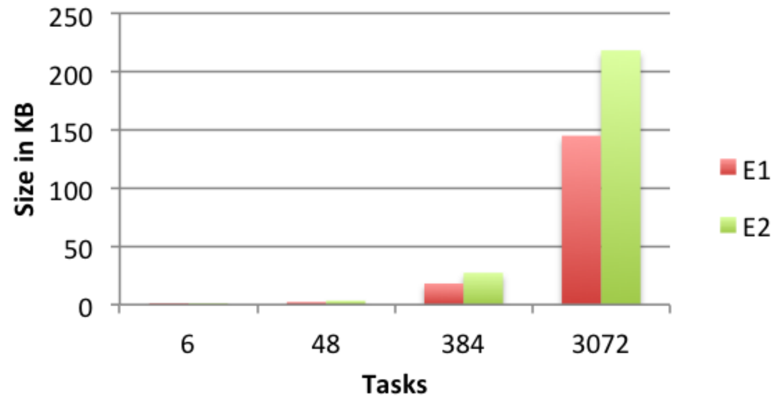

Figure 4.6: Size of Traces Generated (in KB)

4.5.5 RQ2: The Effectiveness of BraceAssertion

For each trace from *E1*, *E2*, and *E3*, I used the synthesized ECA monitor to verify the trace and compared the number of reported violations with my injected errors. In all cases, my synthesized monitor was able to find the injected violations. In my first few runs of *E2*, my ECA monitor located many more violations than the number injected. The application developers quickly determined that these were actual errors resulting from synchronization faults in the coordination across the distributed agents. Ultimately, I used an implementation with these bugs resolved for the results.

When I executed *E2* on the corrected version of the application, I successfully detected all of the injected violations, but I also detected one additional violation in the case of 384 tasks and two additional violations in the case of 3072 tasks. I devised and executed *E2-Wild* to adjust the timing constraint to attempt to find more subtle errors in the application. Fig. 4.7 shows that when I restrict the timing constraint from 40 seconds down to 10 and execute the dual monitors without injecting errors, I find violations "in the wild" in all cases. The application developers confirmed that these violations are the result of inefficient thread management and synchronization in the application.

Because dynamic tasks are more difficult for the patrol application to generate, checking the third specification for very long traces was not possible. For *E3*, I checked violations for the third specification with injected errors for 6 and 64 tasks; my ECA monitor found all injected violations.

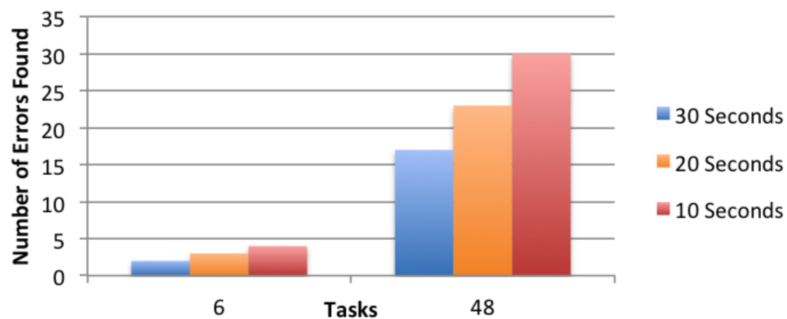

Figure 4.7: *E2-Wild*: Finding Real Bugs

4.5.6 RQ3: ECA Monitor Efficiency and Unique Features

For all the traces generated from *E1* and *E2*, it took the synthesized ECA monitor seconds or less (around 4 seconds for the largest trace in *E2*) to give verification results; these speeds are too quick for VisualVM to profile usage. For this reason, I used the largest trace from *E2* and increased its size by 10 times as a basis for the experiments here.

I first measured the performance impact of *lazy initialization*, which activates the ECA monitor only when the initial event is detected in the trace and deactivates the monitor as soon as the monitor reaches an *accepting* (or *rejecting*) state. I measured the CPU usage and memory consumption with and without lazy initialization (Fig. 4.8). Though there is a small amount of overhead required to maintain a registry of every ECA for activation, the overall performance reduction is effective. I believe the saving is mainly due to number of (specification related) concurrent data structures saved for monitors not activated or deactivated.

Finally, I used *E5* to measure the performance impact of setting the parameters on the lock-free concurrent queue. When the size of the buffer is low (e.g.,

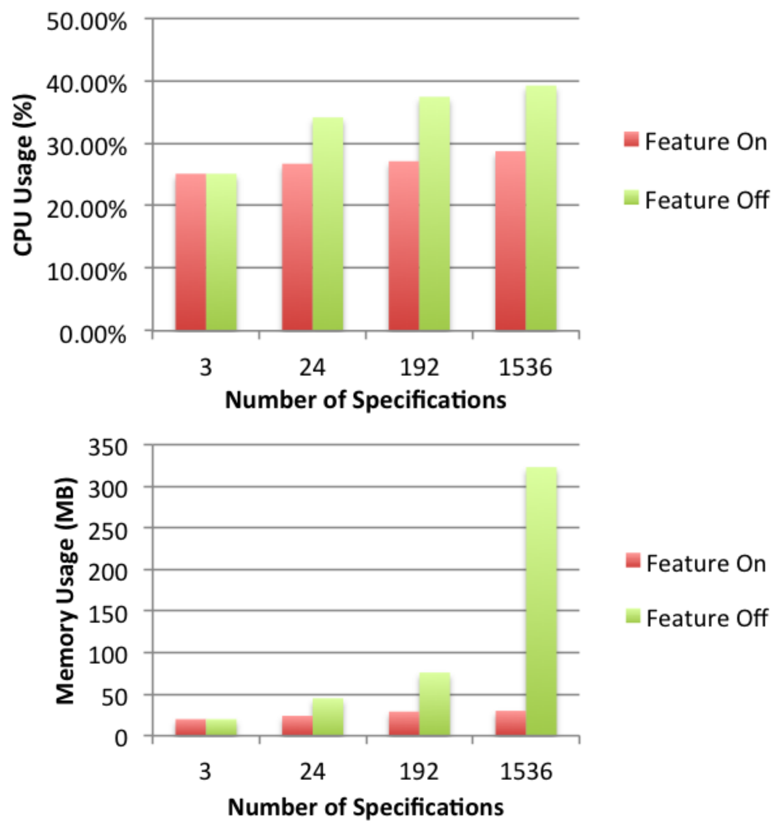

Figure 4.8: Lazy Initialization

10 to 30 timed words), the CPU and memory usage are quite high (Fig. 4.9) due to frequent context switching and conditional synchronization. When the size is very large, CPU utilization is marginally impacted, while memory usage is more significantly impacted due to wasted space in the buffer. The running time, for all settings, remains flat around 60 seconds. These results demonstrate that a balance in defining the size of the buffer is important.

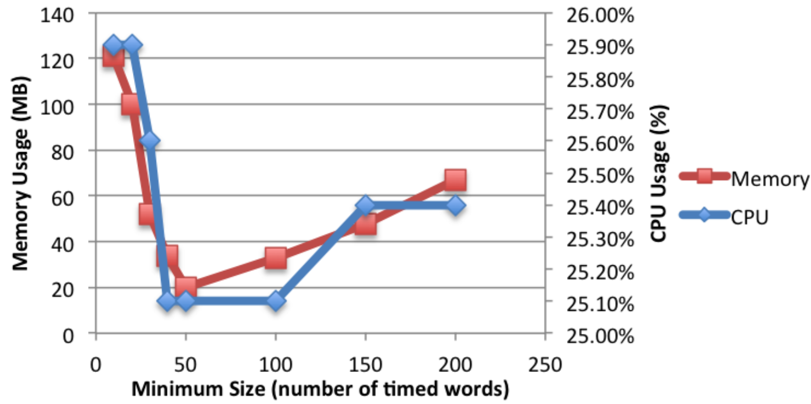

Figure 4.9: Trace Buffer- ECA Monitor

4.5.7 Threats to Validity

A potential threat to the external validity of my work arises because I evaluated *BraceAssertion* using a single application. I have attempted to mitigate this issue by choosing a real world application that is representative of a broad class of cooperative CPS applications. I also chose correctness properties that exercise diverse aspects (e.g., qualitative, quantative, and first-order expressiveness) of the *BraceAssertion* framework. Though I used a limited number of specifications, my experiments increase the number of waypoints dramatically to activate more instances of monitors, which is more relevant to real requirements of CPS applications.

With respect to construct validity, I measured the CPU and memory overhead of my approach using the free VirtualVM instead of a commercial-level tool like JProfiler[8]. In addition, the impact on the running time of the observed application is hard to assess. I measured a decrease in runtime when running an increasing number of monitors, which is counter-intuitive. However, I believe that this is reasonable

[8] `www.ej-technologies.com/products/jprofiler/overview.html`

for CPS because the interdependencies among multiple threads and events are not always deterministic and cannot be measured as in traditional software systems.

4.6 Related Work

In addition to the previously described work on runtime-verification and behavior-driven development, this paper is informed by work on runtime monitors based on temporal logics and other runtime monitors mainly designed for efficiency.

Monitors Based on Temporal Logics. Efforts in real-time temporal logics have resulted in Metric Temporal Logic (MTL) [105] and Metric Interval Temporal Logic (MITL) [8], which check execution traces for real-time properties. State Clock Logic (SCL) [160] includes *prophecy* and *history* clocks and is decidable by a simple decision procedure that relies on event clock automata. Eagle [19] is a fixed-point based logic capable of supporting MTL with bounded space/time complexity. This line of work introduces metrics (often with relaxed punctuality) into temporal logic and enables synthesizing decidable monitors. These approaches only support propositional logic, which cannot express quantification and predicates. Metric First-Order Temporal Logic (MFOTL) [44] adds first-order logic expressiveness and metrics to quantify timing constraints [22]. This work might be most similar to my approach, however, it does not measure runtime performance and cost, and there is no implementation to show this approach can work in a manner that is not intrusive to functional and non-functional behaviors of monitored applications.

Efficient Monitors. Based on JavaMOP, an optimization for parametric runtime monitoring relies on efficient data structures [96]. This is similar to my

approach; I aggregate repetitive atomic events to significantly reduce the number of events to be processed, and I use a global transition table backed by an efficient data structure. my approach is also similar to [154], where inter-property and intra-property monitor compaction deal with large numbers of monitors and high-frequency events. Another way to improve the efficiency of runtime monitoring is to apply static analyses to eliminate unnecessary instrumentation [31]. Since the underlying BDD for *BraceAssertion* requires manual instrumentation, this approach is orthogonal and complementary to my framework. There are a few existing efficient specification languages that I could use as the basis of my framework [150], however I believe the wide popularity and intuitiveness of BDD makes my work more accessible to real CPS practitioners.

4.7 Research Contributions

In this chapter, I made the following contribution:

Research Contribution 3: Create *BraceAssertion* as a new intuitive and expressive language to specify CPS properties with timing constraints (qualitative and quantative) and predicate logics. I support *BraceAssertion* with dual monitors for efficient online timed trace generation and effective offline monitoring of properties violation. This work lays solid foundation for efficient, effective, and more expressive (e.g, to be able to capture global properties violation) online runtime verification.

4.8 Chapter Summary

This chapter brings Behavior-Driven Development (BDD) into runtime verification of CPS applications. To provide a balance between expressiveness and accessibility, I bridge the gap between BDD and Metric Temporal Logics (e.g., SCL) and first order logic. I support the approach using a dual monitor architecture and algorithms, build a prototype on top of aspect-oriented programming, and show the framework to have minimal runtime overhead for the host applications.

Chapter 5

BraceBind: Combining Real-Time Simulation with Runtime Verification for Cyber Physical Systems

With *BraceAssertion* and its supporting monitors, CPS developers are able to capture those subtle bugs in CPS applications (e.g., bugs in the underlying distributed algorithms, and concurrency issues). However, debugging CPS is complicated by the fact that the correctness is dependent on the real-time interaction of the cyber portion of the system with some physical world. In general, existing approaches to debugging CPS *either* evaluate the cyber portion of the system against a (usually not very representative) simulation of the physical environment or deploy and test the cyber portion in a limited number of real environments. In this chapter, I introduce *BraceBind*, a middleware that brings expressive real-time simulation and real-world evaluations under one umbrella to support robust runtime verification of CPS. BraceBind uses an intuitive approach to specifying the physical model interfaces that allows integrating the cyber portion with different types of physical models (e.g., transducer models, rigid dynamic models, and environment models) created in various simulation platforms (e.g., Simulink and LabView). BraceBind also provides annotations that allow connections to these real-time simulations to be easily replaced with connections to an actual physical environment, e.g., to measurements from sensors. This chapter describes my novel time synchronization algorithms and

efficient data integration with microsecond latency, which are needed to bring real time physical process simulation into CPS runtime verification to detect subtle bugs that occur when cyber components interact with physical processes in unexpected ways. The use of BraceBind removes the need for a physical deployment of a CPS for every debugging activity, which can often prove too expensive or even impossible. I evaluate the accuracy, efficiency, and effectiveness of the BraceBind middleware on an empirical study of a real CPS smart agent system.

5.1 Introduction

Cyber-Physical Systems (CPS) are gaining momentum both in academia and industry and have been applied to myriad applications in health, structural monitoring, energy management, autonomous vehicles, and many other fields. Compared with the growth of the domain in general, verification and validation of complete CPS lags far behind [194]. Anecdotal failures are common and, after *post hoc* analysis, the failures are often traced back to a mismatch between the logical and physical components of the complex system. As just one example, in 2003, failures originating in a very localized piece of power equipment connected to a much larger electrical grid triggered the failure of the grid's software management system, which in turn caused a large-scale power blackout lasting more than 20 hours [4]. Considering that CPS have been widely applied to mission critical application areas and are expected to continue to grow in prevalence in these domains, robust approaches to verifying the intertwined physical and logical components are an essential research objective.

However, unlike finding bugs in traditional software systems, which have

only cyber components, bugs in CPS are often tied to the introduction of physical components. Such bugs are often induced by limitations and misrepresentations in the software interface to physical hardware. This is exactly the challenge that manifested in the electric grid example above; the control software embodied implicit assumptions (e.g., the maximum possible voltage loaded in the electric grid) about the underlying physical components that simply failed to hold at runtime. Such implicit assumptions are rarely externalized because today's technologies make it difficult for CPS designers and developers to express these assumptions, and they are often impossible to infer automatically with any reasonably sufficient precision.

An in-depth study of CPS designers and developers showed that the state of art in formal methods (e.g., theorem proving and model checking), simulation, and testing are not sufficient in capturing these bugs that arise from interactions between cyber and physical components [194]. As a complement to formal methods and testing, runtime verification provides a "last line of defense" in capturing these subtle bugs. However, in CPS, the physical properties of the deployment environment are often difficult or impossible to replicate at development time, resulting in the deployed application encountering unexpected conditions. For example, the designer of an autonomous vehicular system may assume (and benchmark) that applying a given amount of power to the motor for a specified amount of time causes the vehicle to move a specific distance (or at least within a given range of distances). However if the environment changes (e.g., the coefficient of friction of the road surface or the incline of the surface) changes, the assumption may fail to hold. This is a simplistic example chosen to make the point; it is easily (and commonly) avoided by

autonomous vehicle developers, but more insidious and harder to detect versions of the same fundamental problem routinely creep into deployed CPS systems [4, 133].

The de-facto standard in CPS runtime verification is to perform the end-to-end system tests after deploying the complete CPS system to the target deployment environment. However the deployment process itself is often very expensive and time consuming (e.g., road tests for autonomous vehicles) or not possible (e.g., for rover deployed to the Mars). Such a debugging approach that relies on the deployment process poses non-trivial challenges for creating an accessible and repetitive testing environment, which is essential for CPS (especially complex ones). In this chapter, I motivate an integrated testing environment that can combine real time simulation of physical processes with runtime verification; the debugging environment can be used to identify any incorrect assumptions about the underlying physical processes in those challenging environments. For instance, my motivating electric grid example would be greatly enhanced by a debugging environment in which the control software can be tested against simulated physical models of the grid itself before deployment. With the help of an expressive runtime verification framework [195, 196], essential properties of the control software (e.g., the fact that no power line at any time has overloaded voltage that is more than 765 000 V) can be specified and tested.

Concretely, this chapter makes the following contributions:

- I introduce real-time simulation of physical components into runtime verification to help expose CPS developers' implicit assumptions about the underlying physical components in the deployment environment.

- I create the *BraceBind* middleware that handles challenges associated with time synchronization and data integration essential for real-time simulation across the heterogeneous simulation platforms needed for runtime verification of CPS.
- I provide a simple model interface specification and tailored annotations for *BraceBind* to “glue” cyber components and various real-time simulation of physical components into an integrated runtime verification environment.
- I introduce annotations that allow connections to real-time simulation to be easily replaced with connections to actual measurements of the physical environment (e.g., from sensors).
- I evaluate the approach using a real CPS application of a smart agent system.

5.2 Motivation and Related Work

Runtime verification of CPS will be greatly enhanced by incorporating real-time simulation of physical models. However, a few significant challenges in doing so remain open in the context of the current literature and state of the art.

The first of these challenges lies in the mismatch between the language of such physical models and the language(s) used to implement the cyber portions of CPS. To incorporate real-time physical models into runtime verification, I must first transform the physical models or simulations into executable modules (e.g., C or C++). Several existing simulation platforms provide built-in or plug-in utilities for such physical model transformations. For example, Simulink Coder[1] (formerly

[1] `http://au.mathworks.com/products/simulink-coder/`

Real-Time Workshop) transforms Simulink models into C++ code, and the LabView C Code Generator[2] can transform LabView models into C code or Windows DLLs. Other work in the literature has also enabled the generation of C code from Modelica models [69, 142].

However, all of these existing tools require basic configurations to be set up before code transformation can happen. For instance, in Simulink Coder and LabView C Code Generator, one must choose a fixed step size and appropriate solver algorithms. As its name implies, the step size (τ_s) remains constant throughout the simulation. However, the real time required to execute each step (τ_p) varies depending on the hosting machine and rarely matches τ_s. Since τ_s is generally much larger than τ_p, some time synchronization behavior is required at runtime [43]. However, τ_s is set statically based on experience, which can result in values that are either too large (causing a loss of data accuracy) or too small (making it impossible for the host machine to finish one step within the step size). Also when time synchronization is required across different models hosted in different computation units, clock drift inevitably causes errors in local time synchronization algorithms (especially for smaller step sizes).

My solution within *BraceBind* determines the step size automatically based on the deployment machines and uses a network based time synchronization algorithm to synchronize τ_s with τ_p to deal with clock drift. Moreover, in a real-time simulation environment, models that require a smaller step size (e.g., a rigid dynamic body model requires a microsecond step size to match the simulated physical

[2] `http://sine.ni.com/nips/cds/view/p/lang/en/nid/209015`

device) have to synchronize with models with a (much) larger step size (e.g., sensor models generally require a millisecond step size) with (significant) delay imposed on the former. To avoid losing the required accuracy (by unnecessarily waiting for slower models), the *de-facto* standard in simulation (including in Simulink Real Time[3] and dSpace Real Time[4]) is to deploy a (often very large) physical model with all the components as subsystems; this model is then centralized and executed on a dedicated server. The step size for the model is determined by the fastest running subsystem. This approach wastes resources, especially for those models that do not require this finer level of step size (which could be a majority of the models in a CPS application). Further, this approach is not scalable with increasing complexity of models, and it is not applicable for establishing a real-time simulation environment that incorporates models across different simulation platforms (e.g., Simulink and LabView). I introduce a novel concept *TimeZone*, which groups models based on their timing requirements. This allows different levels of time synchronization inside and outside of a specific *TimeZone*.

The complexity of CPS systems in general demands heterogeneous simulation environments spanning various physical domains [87, 140]. Even with the presumption that each domain provides tools to transform models into executables, integrating them into a complete test environment remains a non-trivial research challenge. For instance, at runtime, the input and output parameters of a vehicle kinematic model (in Open Dynamic Engine [143]) may need to be connected with a

[3] `http://au.mathworks.com/products/simulink-real-time/`

[4] `https://www.dspace.com/en/inc/home/support/suptrain/systems.cfm`

controller model (in Simulink) and a few sensor models (e.g., accelerator model built in SystemC [144]). Function Mock-up Interface (FMI) [66] provides interfaces to integrate physical components across different simulation platforms. However, FMI requires simulation platform vendors to strictly support FMI function calls, which is not practical. Moreover, the two fundamental challenges in integrated simulation, namely time synchronization and data integration are left for developers to handle, which is complex and error prone. In my solution, CPS developers just need specify the input and output interface for each physical model, and *BraceBind* converts the simulation models into executable modules, handling time synchronization and data integration across models automatically with only microseconds delay.

My work is in line with Hardware-in-the-Loop testing [43, 73, 79, 191], which is becoming increasingly common in practice. In [79], a real-time simulation environment with microseconds 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. In comparison, my solution is built on top of a generic message passing framework, and is therefore not designed for any specific hardware or tied to any specific processors. Work in co-design [74, 129] validates device function for a wide range of operating conditions. In this approach, the real device communicates to mathematical models which are implemented as FPGA circuits or processor instructions; the models are executed in real time to simulate the interacting environment. In comparison, I use off-the-shelf simulation platforms (e.g., Simulink and LabView) and existing physical models for the given CPS application (which are created anyway during prototype phase). Also as I bring real-time

simulation into CPS runtime verification, I provide glue that enables switching the test environment from a physical setting to a real-time simulation setting, even enabling test settings that consist of combinations of physical devices and simulation models. In [135], a virtual prototype of an entire cyber physical system is built inside the SystemC simulation environment. Besides the large amount of work required to create the necessary different kinds of models, the cyber part is also prototyped. Such an approach comes with the common disadvantage of pure simulation: the fact that the prototyped CPS will likely not match the actual deployed CPS. In comparison, *BraceBind* integrates (existing) simulation models from (popular) simulation platforms to test real implementations of CPS applications based on existing work in runtime verification.

5.3 Interface Specification for Physical Models and the Cyber

To establish and connect a real time simulation to the cyber portion of a CPS, *BraceBind* requires a few essential pieces of information from the physical models and the cyber part of the CPS. I design an XML Schema Definition (XSD) through which the CPS developer provides the required input, output, and a few other basic pieces of information for each physical model, and I use a specific annotation class to define input (to actuators) and output (from sensors) connecting the cyber part of the system to the real time simulation and the physical environment. I use the physical models for an electric vehicle used for the evaluation as examples throughout the section.

5.3.1 Model Interface Specification

For each model required in the real-time simulation, the developer uses XML to provide information of the input, output, and the basic information for the model. For instance, to establish a real-time simulation for an electric vehicle traveling across different terrains, I would like to replace the actual vehicle with a simulation of the vehicle, based on a physical model of its expected behavior. To do this, I need a dynamic body model for the vehicle. No matter how complex the model is, it requires the input for current angle change and drive force, and the model outputs the current position and angle of the vehicle. *BraceBind* also requires some basic information about the model and its realization in order to be able to connect it to the CPS implementation. This information includes the simulation platform the model is currently created under (e.g., Simulink), the file name for the model (e.g., slx or mdi file), the solver algorithm (e.g., Runge-Kutta, Dormand-Prince, or Euler), and a few other essential pieces of information. *BraceBind* uses this information to generate executables (C or C++) from the existing model file. To help control the simulation and its integration with the cyber part of a CPS, the CPS developer provides a minimum and maximum for the simulation's step size and the name of the *TimeZone* the model belongs to. The settings for the step size are used by *BraceBind* to find an optimal value for the fixed step size based on the deployment machine. *TimeZone* is used by *BraceBind* to unite physical models with similar timing requirements into a group to achieve time synchronization across all the models while not sacrificing required accuracy within the group. This is described in more detail in Sec. 5.4.5.

The following XML is part of the specification for the aforementioned vehicle's dynamic body model. In the sample input, I use "ModelSpecific" to include those input parameters that may be set statically or empirically (e.g., Vehicle Mass), and use "AppSpecific" to include those input parameters whose values come from other models in the same simulation environment (e.g., current angle). One important thing to note is the value of current angle change ("Angle_Change") comes from a model called "CyberDelegate". This model is created automatically by *BraceBind* as a facade between the cyber part of the system and the real-time simulation. The input and output for the model are defined in the next section. For brevity, I skip the discussion of the schema and refer readers to `https://goo.gl/7YzvmQ` for more details.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Model>
  <Name>DynamicModel</Name>
  <Platform>Simulink</Platform>
  <ModelType>Dynamic</ModelType>
  <FilePath>/user/james/simulink/dynamicmodel.slx</FilePath>
  <OutputPath>/user/james/rtse/dynamicmodel.cpp</OutputPath>
  ...
  <Input>
      <Integration>
            <FixedStepSize><Min>0.00001</Min><Max>0.0001</Max>
            </FixedStepSize>...
      </Integration>
      <ModelSpecific>
            <Param>
                <DataType>Double</DataType><Name>Vehicle_Mass</Name>
                <Value>1.8118</Value>
            </Param>
      </ModelSpecific>
      <AppSpecific>
              <Param>
                     <ModelName>CyberDelegate</ModelName>
                     <Name>Angle_Change</Name>
               </Param>...
      </AppSpecific>
  </Input>
  <Output>
       <DataPack>
            <DataType>Double</DataType><name>X_Position</name>...
       </DataPack>
```

```
  </Output>...
</Model>
</xs:schema>
```

5.3.2 Annotations for physical aspects

A primary motivation of my work is to provide "glue" to connect cyber components to values that reflect the physical environment, whether these values come from real-time simulation or from physical transducers (e.g., sensors and actuators). In *BraceBind*, this glue comes in the form of annotations that provide a unified view of the physical system and specify how the simulation or transducers connect to that system model.

In my previous work on CPS runtime verification [195,196], I have developed an intuitive specification language based on a determinizable timed automaton [9] called *BraceAssertion*. This assertion language is coupled with a practical online runtime verification framework, that is efficient, effective, and specifically designed for CPS. In its current form, *BraceAssertion* uses a Behavior Driven Development (BDD) [26] style annotation to connect implementation with specification. In this work, *BraceBind* extends the BDD annotation to connect the cyber portion of the implementation with the real-time simulation via the "CyberDelegate" model provided by *BraceBind*. More specifically, I add a BDD annotation class called *PhysicalVariable*, which is used to differentiate those variables in the cyber part of the implementation, values of which either come from sensors or are used (often as a result of a control algorithm) to command actuators. When connecting the variable to a real-time simulation, the physical variable has a property "Type," which deter-

mines whether the value of the variable is used as an input to the "CyberDelegate" model ("Actuator") or as an output from the "CyberDelegate" model ("Sensor")[5].

For instance, to supply "Angle_Change" input to the "CyberDelegate" model, the developer provides the annotation in Fig. 5.1. *BraceBind* provides an instrumentation process to locate the variable ("angle") and insert a statement after the value assignment. The statement will report the changed data to the "CyberDelegate" model using *BraceBind*'s lightweight and very low latency messaging service (Sec. 5.4.3). While the example shows the annotation attached to a particular method, it can also be provided at the class level[6]. 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 the value "Physical" means the variable is connected to a real device. There is also a project level "mode" which can connect the entire cyber part with a real time simulation ("Simulation") or with a physical environment ("Physical").

5.4 BraceBind

In this section, first I briefly explain the *BraceBind* architecture, and then explain in detail the key algorithms inside *BraceBind*. Throughout, I demonstration how *BraceBind* address the challenges identified in Sec. 5.2.

---

[5]To simplify the example, I connect the "CyberDelegate" model directly with vehicle Dynamic model; in the experiment, there are accelerometer and gyrometer models that supply data to the "CyberDelegate" model, and there is a motor actuator model that receives data from the "CyberDelegate" model.

[6]In future work I will explore extending *BraceBind*'s annotation to both the the project and statement level.

```
@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);
}
```

Figure 5.1: Sample use of physical variable annotations

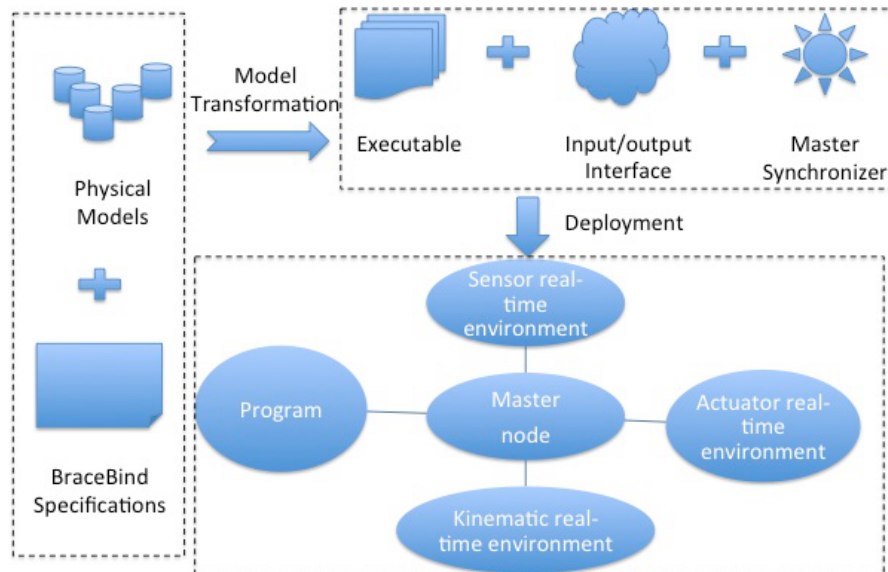

Figure 5.2: BraceBind: Real-time Simulation Integrator

5.4.1 BraceBind Architecture Overview

Fig. 5.2 shows how CPS developers use the *BraceBind* approach to integrate real-time simulation of physical portions of the CPS into the development process. A CPS developer uses the XML specification in Sec. 5.3 to specify the input and output parameters for each needed physical model; because of the heterogeneity of

many CPS environments and preferences by CPS developers, these physical models may be running on heterogeneous simulation platforms. *BraceBind* then automatically transforms the model to create an executable for each model; generates the input and output programming interfaces for each model; and creates a *master synchronizer*, which coordinates the data exchange and time synchronization among the models, and between the models and the CPS program. CPS developers can then manually deploy (or *BraceBind* can deploy automatically) these synthesized components, enabling a real-time simulation environment as part of the CPS testing environment. In connecting the *BraceBind*-assisted real-time simulation to the CPS testing, not only are idiosyncrasies related to timing and synchronization handled on the developer's behalf, but sensor models, actuator models, kinematic models, the master synchronizer, and the CPS system can be deployed to their own machine(s) and devices. In the *BraceBind* approach, the only input required for CPS developers to provide are (1) the specification XML to guide the entire automation process (manual deployment is possible but is reserved for hand-optimizing the testing environment) and (2) the annotations to connect a selected number of *PhysicalVariable*s or the entire cyber part of the system with the real-time simulation. Compared with the *de facto* standard in [66], flexibility, intuitiveness and most importantly practicality are the key advantages and my main contributions.

The main algorithms in *BraceBind* can be divided into four categories: *Model Transformation*, *Data Integration*, *Time Synchronization*, and *Master Synchronizer*. I will walk through each category in the remainder of this section.

5.4.2 Model Transformation

Algorithm 2 shows *BraceBind*'s main algorithm to generate, instrument, and deploy the real-time simulation modules using the specified physical models. The algorithm iterates over each model in the XML specification, uses code generation scripts provided by the simulation platforms (e.g., lvcg from LabView C Generator or rtwbuild from Simulink) to generate C/C++ code automatically and to deploy it to the designated path (line 3). Then the algorithm instruments the generated code (e.g., for Simulink generated code, it is necessary to change the specified location of a few referenced header files and the access permissions for a few variables in preparation for *BraceBind*'s later tasks) (line 4). The algorithm then instruments the generated code with data integration code and copies over to the deployment path a few library files required for the data integration (line 5); more details on the data integration process are provided in Sec. 5.4.3. Afterwards, the algorithm instruments the generated code with time synchronization code and copies over to the deployment path a few library files for time synchronization (line 6); similarly, more details on the time synchronization algorithm are given in Sec. 5.4.4. Later the algorithm compiles the generated code into an executable (line 7), and registers the model with the deployment path of the executable to the master synchronizer (MS), which coordinates and monitors the real- time simulation environment (line 8).

Algorithm 2 Model Transformation

```
1: models ← lookupModelSpec
2: for modelinmodels do
3:     dpath ← generateCode(model)
4:     instrument(dpath)
5:     addDataStub(dpath)
6:     addTimeSynchronizationStub(dpath)
7:     compile(dpath)
8:     register(dpath, model)
```

5.4.3 Data Integration

In *BraceBind*, the generated executable for each model communicates with each other and with the master synchronizer (MS) using a Message Passing System built on top of ZeroMQ [88], which provides a scalable solution and a message delivery latency less than 1 millisecond when the message size is less than 100KB [169]. When evaluated in my testbed, the latency for messages of size less than 50KB in the same machine is less than 65 microseconds and less than 400 microseconds among computers in the same network (connected via 1G Ethernet).

In the process of generating executable from each model, *BraceBind* generates a class implementing the interface `DataIntegrator` shown in Fig. 5.3. The *DataIntegrator* has four key functions:

- `startPublisher` is called to start up a *ZeroMQ* agent that listens to subscription for the model output values from *ZeroMQ* agents for other models in the same real-time simulation. For each subscription request, the publisher retrieves the most current data from a lock free concurrent queue (*OutputQueue*). The data in the queue is provided by a separate function discussed later; this approach avoids locks and keeps the output data "fresh". To enable this behavior, the publisher also registers its IP address and port number with the Master Synchronizer (MS).

The MS provides a directory service for each real-time executable (identified by its publisher). This enables publishers and subscribers to send messages directly, which is essential for efficient message exchange for real-time simulation.

- `startSubscribers` is called to start up one or more *ZeroMQ* agents to act as subscribers to publishers for the model's required input values. The subscribers contact the master synchronizer to find the needed publishers. The subscribers subscribe to the input data with a much smaller (e.g., 4 times faster) interval than the fixed time step specified for the physical model they are serving. This ensures that the input data for the physical model is current in the face of network latency and noise. The retrieved input data along with its timestamp is stored inside a lock free concurrent queue (*InputQueue*).
- `getRealTimeInputValueFromQueue` is called before current physical model performs a time step and retrieves the most recent input data from the `InputQueue`.
- `addResultToQueue` is called when the current physical model completes a time step; this function stores the output data into the `OutputQueue`.

```
class DataIntegrator {
 public:
  virtual void startPublisher(string modelName,
                              unsigned int portNumber) = 0;
  virtual void startSubscribers(vector<string>& inputModels) = 0;
  virtual HashMap getRealTimeInputValueFromQueue() = 0;
  virtual void addResultToQueue(HashMap currentResult) = 0;
  // other methods omitted for brevity...
}
```

Figure 5.3: `DataIntegrator` Interface

5.4.4 Time Synchronization

In real-time simulation, it is essential to guarantee that the fixed step size (τ) in simulation is equal to the actual time (T). This is especially important in *BraceBind*, as some physical components may be simulated but others may not be. In [43], the clock frequency of a timer in the host computer is obtained at the beginning and end of the step function of a simulation model; the difference of these two values is taken as the actual time required for the model to perform one step (T). The difference between T and τ (generally, τ is much bigger than T) is used as an input to a fine-tuned delay function to maintain the time synchronization between τ and T. However, this previous work is constrained to a single host platform (the Windows operating system) and cannot deal with clock drifts across different computation platforms. In this chapter, I define a hierarchical approach to handle time synchronization challenges in real-time simulation. In this section, I describe the challenges and my solutions to guarantee accurate local time synchronization for each model. In Sec. 5.4.5, I will explain my solution for time synchronization across models and finding an optimal solution for each model's step size in the deployment environment.

First, in generating a real-time executable for each model, I automatically deploy a Network Time Protocol (NTP) [130] daemon into each host machine. This step guarantees about one millisecond time synchronization accuracy across nodes in the real time simulation (intra-net) [27].

Secondly, in the same process, *BraceBind* generates a class (*TS*) implementing the interface `TimeSynchronizer` in Fig. 5.4. The *TimeSynchronizer* has four

key functions:

- `initialize` retrieves the assigned step size from the master synchronizer and stops time for running the model in the deployment environment.
- `moreSteps` controls the loop that the current physical model executes each time step. For each physical model in the real-time simulation, I can terminate it either from the master synchronizer or assign a fixed number of time steps to execute.
- `startTimeSync` takes the correct time stamp of the deployment machine before the physical model executes a step function.
- `endTimeSync` takes another correct time stamp of the deployment machine after the physical model execute a step function. The difference between this time stamp and the previous one is the actual time this step T. If assigned step size τ is bigger than or equal to T, an accurate nanoseconds level wait function is called for the period of $\tau - T$, otherwise, a timeout error is reported.

```
class TimeSynchronizer {
  public:
    virtual void initialize(string modelName, float stopTime,
                            float stepSize) = 0;
    virtual bool moreSteps() = 0;
    virtual void startTimeSync() = 0;
    virtual void endTimeSync() = 0;
    // other methods omitted for brevity ...
}
```

Figure 5.4: `TimeSynchronizer` interface

Besides the apparent difference that the fixed step size is assigned from the master synchronizer, the time synchronizer (TS) implements similar functionality as in [43]. However, my TS not only provides additional auto detection of the deployment environment (*BraceBind* currently supports Windows, Ubuntu, and Mac OS)

and activates required platform-specific library calls (e.g., `host_get_clock_service` in Mac OS and `QueryPerformanceFrequency` in Windows), but also provides more accurate wait functions. For instance, in my initial experiment, I have tried different types of system wait or sleep functions. I found even the most accurate wait function provided by the system (`nanosleep` in Linux and Mac) does not wait for the exact time, which is not acceptable in real-time simulation. For instance, one of the physical models in my evaluation application exhibits the following properties has a value that sometimes (legitimately) fluctuates greatly. If the data for a previous time step is mistakenly used as input for the current time step, crucial physical elements (in this case, the angle of the vehicle dynamic body model) will be (potentially dangerously) misrepresented. Instead of relying on a sleep function provided by the system, which is heavily dependent on the operating system version and configuration, I define my own more reliable wait function that uses a hard loop and polls a platform-specific accurate nano-seconds timing function (e.g., `clock_get_time` in Linux and Mac). From my experiments, this results in highly consistent values for the real-time simulation.

5.4.5 Master Synchronizer

As the kernel of *BraceBind*, the master synchronizer (MS) provides the following functions to make establishing real time simulation possible: 1) directory service for the underlying message passing system; 2) time synchronization across models in the real time simulation; 3) monitor for the the real-time simulation environment, 4) logging for models' output subscribed by the application for debugging

analysis. These functions are achieved inside Algorithm 3.

Algorithm 3 Master Synchronizer - Main

```
1: if modelsTransformed(modelSpec) then
2:     currentMode ← Trial
3:     for model ∈ activatedModels do
4:         remoteStartModel(model)
5:     currentMode ← RegoCompleted
6:     outputCurrentMode()
7:     currentMode ← Optimize
8:     for model ∈ activatedModels do
9:         createTimeZoneAgent(model)
10:    optimizeStepSize()
11:    currentMode ← ready
12:    generatePhysicalVariableBinding()
13:    while !receiveStopFromRV() do
14:        if receiveInputFromRV() ∧ currentMode = ready then
15:            currentMode ← start
16:            startModels()
17:            startMonitor()
18:    shutDownModels()
```

In the algorithm, after all the physical models in the XML specification are transformed into executables and deployed into designated path, the MS enters the *Trial* state (lines 1-2), which remotely activates the models in the real-time simulation. After all of the models register with the MS (as described previously), the MS enters the "RegoCompleted" state and publishes the mode to all of the models in the real-time simulation. The models can then look up the IP and port number for models providing their required input data (lines 3-6). Then MS enters the "Optimize" state.

In real-time simulation, it is intuitive that 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. However, to save processing power and memory, it is common to use a low-order Ordinary Differential Solver (ODE) with a smaller step size to reduce local (per time step) and global error to an

acceptable level [108]. This makes the intuitive mechanism not possible in real-time simulation due to the unacceptable level of local and global error (i.e., larger step sizes introduce larger errors). To deal with this challenge, instead of the *de facto* solution of deploying all the models with different timing requirements into a dedicated yet expensive server with the smallest step size (which is expensive and wastes resources), I introduce *TimeZone*. The CPS developer can group physical models with similar timing requirements into a single *TimeZone*. For instance, a CPS developer can deploy the vehicle dynamic model and the environment model in the evaluation application into one *TimeZone* that allows microsecond level time synchronization. Other, less stringent models, can be placed in a different *TimeZone* with a larger time step.

Ideally all the models in a given *TimeZone* are deployed into one machine whose capabilities are matched to the requirements of the *TimeZone*. Alternatively, the models may be able to execute in an intra-net with a clock synchronization algorithm with sufficient accuracy [98]. For each *TimeZone*, the MS needs to create a dedicated timezone agent that subscribes to input data and publishes output data for the simulation models inside the associated *TimeZone*, and more importantly performs each time step while fully synchronizing with other timezone agents. This requires the timezone agent to wait for the slowest *TimeZone* at the end of each time step. The creation of specified *TimeZone* and timezone agent, and assigning each model to a timezone are achieved in lines 8-9 of Algorithm 3. The MS then sets an optimized fixed step size for each *TimeZone* (line 10); I elaborate on this optimization algorithm later.

After the optimization step, each physical model enters "sleep" mode to await further commands from the MS. The MS enters the "ready" state and creates the "CyberDelegate" facade to connect the cyber part of the system to the real time simulation. As part of the process for creating the delegate, the MS creates the `PSAgent`, which reports the start and end of the CPS application to the MS, which is used to control the life span of the real-time simulation environment (lines 11-12). Once CPS developer starts the CPS application (e.g., using a runtime verification framework like *Brace* [196]), the communication between the cyber part of the system and the real-time simulation starts. When the MS detects the "start" signal from the *PSAgent* and the current mode is "ready", the MS notifies all the physical models to activate and start executing their step function (lines 13-16). Afterwards the MS monitors the runtime behavior of each model by subscribing to the output for all models. If an error is reported for a given model, the MS stops all the models and reports the error back to CPS developer (line 17). After the MS detects the "end" signal from *PSAgent*, it shuts down all of the real-time models executables and generates log files for each physical variable.

Besides serving as the facade between the cyber and the real time simulation, "CyberDelegate" also has other two functionalities. First, it contains lock-free concurrent queues for the exchange of data between *PhysicalVariable*s in the cyber portion of the CPS application and models in the real-time simulation. Second, it enables logging of those physical variables data for further debugging analysis. For instance, when the runtime verification framework detects a violation of a CPS property, the CPS developer can look at the trajectories for those logged physical

variables in the property to help debugging subtle errors involving physical processes and/or regarding the deployment environment.

I return now to the question of how I can optimize the step size for each *TimeZone*. I am motivated to optimize these time steps because it is known that of I can reduce the step size of an ODE solver by a factor of λ, I am able to reduce local error (per time step) by approximately $\lambda * n + 1$ and reduce global error by approximately $\lambda * n$, where n is the order of the ODE solver [108]. However, reduction of step size can incur additional computational costs. Because most simulation platforms only support fixed step sizes once the simulation starts, it is important to set the step size wisely from the start. I aim to achieve a balance between accuracy and computational efficiency by computing the best step size for each *TimeZone*. Algorithm 4 shows how I achieve this optimization for my real-time simulation models. I first iterate through each zone to test for each model whether the max step size (default) works for the deployed machine (i.e., that τ is bigger than T). If not, the algorithm reports an error (which indicates that the deployment machine is not suitable for the model or the max step size is not set correctly) (lines 3-4). I then find the largest, second largest, and smallest step size (as specified in the models' XML specifications) in the current zone (line 5-7). I then try to (iteratively) find whether I can reduce the maximum step size for the *TimeZone* to the smallest possible maximum step size of models in the same zone without causing timeout for any models in the *TimeZone* (lines 8-14).

Algorithm 4 Find Optimal StepSize

```
1: for ∀zone ∈ TimeSyncZones do
2:     for ∀model ∈ zone.models do
3:         if !run(model.maxSize, model) then
4:             reportTimeOutError(model)
5:     existingMaxSize ← findMax(zone)
6:     maxSize ← findMax(zone, existingMaxSize)
7:     maxMinSize ← findMaxMin(zone)
8:     for maxSize > maxMinSize do
9:         for ∀model ∈ zone.models do
10:            if model.maxSize > maxSize∧!run(maxSize, model) then
11:                goto nextTimeZone
12:        existingMaxSize ← maxSize
13:        maxSize ← findCandidateMax(zone, existingMaxSize)
14:    nextTimeZone:
```

5.4.6 The Case Study

I evaluate *BraceBind* using a case study application that existed before the creation of *BraceBind*; this is the application I have used for examples.

5.4.7 The Case Study

I used an existing robot planning system, which is a distributed version of Generalized Partial Global Planning (GPGP) [116]. This system's planning algorithm is a version of Anytime A* customized to distributed planning for a group of mobile vehicles. Specifically, a group of vehicles is assigned patrols that must visit a set of way-points. The vehicles negotiate, and each derives a schedule that contains a subset of the way-points. The schedules are chosen to optimize the combined utility of the vehicles.

My case study vehicle is based on the Rover 5 Robot Platform[7], which uses four independent motors, each with a hall-effect quadrature encoder and gearbox. I

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

equipped the vehicle with accelerometer and orientation sensors. The evaluation application is built in Android and deployed to a Samsung Galaxy S3 phone to control the rover. I conduct the experiments in a laboratory environment, where there are three surfaces to mimic different physical deployment environments (wood[8], grass[9], and linoleum[10]) and an overhead camera for both positioning and recording the tests. Before the experiment, there are existing thoroughly tested physical models for the vehicle including motor, vehicle dynamic, environment, accelerometer, and gyrometer.

5.4.8 Research Questions (RQs)

My evaluation answers the following research questions:

RQ1: What is the accuracy of the real-time simulation established by *BraceBind* compared with the physical models in their original simulation platforms?

RQ2: What is the accuracy of the real-time simulation established by *BraceBind* compared with the real deployment environment?

RQ3: How effective is the runtime verification that uses *BraceBind*'s real-time simulation compared with the runtime verification in physical deployment environment?

[8] A sample video for wood is at `https://goo.gl/PkxDa4`
[9] A sample video for grass is at `https://goo.gl/Vv3fF2`
[10] A sample video for linoleum is at `https://goo.gl/s097Ag`

5.4.9 Experiment Design

To answer these questions, I implemented a prototype for *BraceBind* in C++. I then ask developers familiar with the existing physical models to create *BraceBind* specifications for these models, and ask developers familiar with the evaluation application to annotate the application code to identify *PhysicalVariable*s using supported BDD annotation. Then I use *BraceBind* to automatically establish the real-time simulation environment. The real-time simulation environment is running 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 are running in the Mac machine and other models are running in Ubuntu.

I ask application developers to annotate the code using customized annotation class to identify variables and predicate functions required for the runtime verification framework [195] to check the following six application correctness properties: 1) when a vehicle is assigned a task, the vehicle shall have this task in its local scheduler (*P1*); 2) the completion time for a given task is bounded (*P2*); 3) when a vehicle is assigned a task, the schedule for this task is optimal (*P3*); 4) the integral of cross track error is bounded (i.e., the vehicle is not weaving) (*P4*); 5) the duration of the main control loop is bounded (*P5*); 6) the number of messages for each task negotiation is bounded (*P6*). These properties are hard to verify by hand because they are related to the distributed algorithm, timing requirement, and the control algorithm.

I report results for the following experiments, which are specifically designed

to answer the above questions:

- *Simulation-platform VS Real-time Simulation (E1)*: I compare each model in the real-time simulation environment (RTSE) with the model in its original simulation platform. I create a big simulation model combining all the physical models for the application and run it in Simulink. I feed 20 random inputs to the model and RTSE (with the cyber part disconnected and replaced with a program providing random inputs and recording the outputs from RTSE) with 30 minutes running time (which is the maximum time it takes for all vehicles to complete all the tasks in each physical deployment scenario), and compare the output (which is the trajectory of the rover position including x, y, velocity of x, velocity of y, and the vehicle's angle).

- *Real-time Simulation VS Physical Deployment (E2)*: I deploy the evaluation application on two physical rovers in the controlled lab with three floor settings (wood, grass, and linoleum). I then create an RTSE with the evaluation application (with two identical sets of vehicle related models). For each type of floor, I change the environment model input (specifically the co-efficiency of drag) to reflect the friction level on each physical setting. I conduct one experiment for each floor setting with a separate task issuer assigning 180 tasks (waypoints) for each experiment. I record x and y position of each rover along with the time stamp, time of reaching each waypoint both for the physical deployment and the RTSE.

- *Real-time Simulation + Runtime Verification VS Physical Deployment + Runtime Verification (E3)*: I conduct the same experiment as in *E2* but with the

runtime monitors embedded inside the application to monitor my six application correctness properties. I also inject errors in the application to intentionally violate those properties. I record number of errors found by the runtime verification framework (real + injected) both for the physical deployment and the RTSE.

I report results for each experiment and also discuss a few unexpected and interesting findings during the experiments. I close this section with a discussion of key features of my framework, and of some validity discussions.

5.4.10 BraceBind VS Simulink (E1)

Fig. 5.5 plots the difference between each output parameter using the *BraceBind* real-time simulation vs. direct simulation in Simulink for the 20 different runs. The figure shows the average difference for each output parameter, along with the maximum and minimum errors, over the 20 runs. These differences are *very* small (the maximum error, across all runs and all variables, was 0.029%), demonstrating that the real-time simulation environment established by *BraceBind* is highly faithful to the original simulation environment. The accurate result is due to my highly accurate time synchronization, low latency data integration, and step size optimization; I omit the details demonstrating the individual contributions of these components for brevity.

5.4.11 BraceBind VS Physical Deployment (E2)

Fig. 5.6 shows the trajectories of one of the test vehicles reaching 19 assigned waypoints, both in the Linoleum environment and in the *BraceBind*. This trajectory

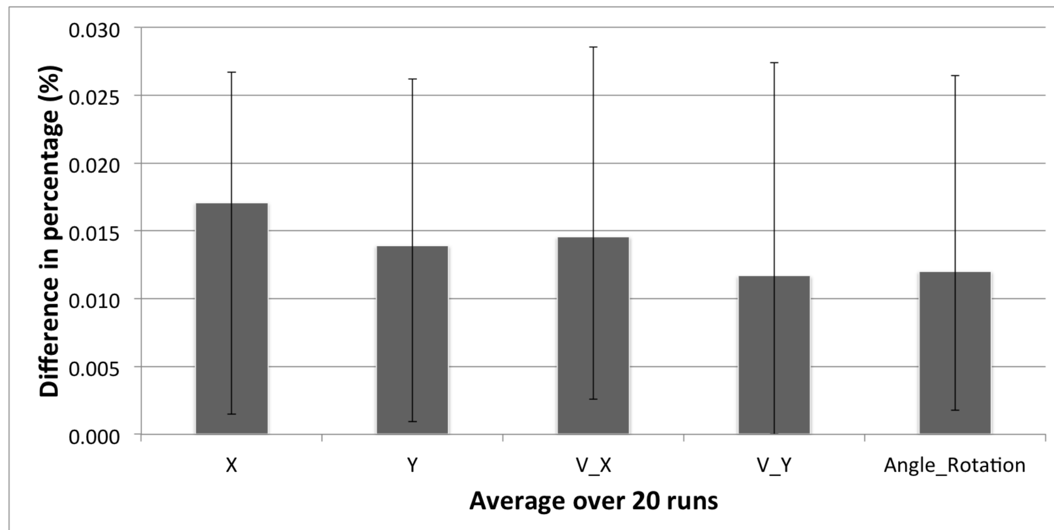

Figure 5.5: BraceBind VS Simulink

is representative of the trajectories for both vehicles across all 180 waypoints. The Y axis shows the distance from the vehicle to the next waypoint. The Waypoints Range series shows the acceptance range of each waypoint (e.g., if the vehicle is within 25 centimeters of a given waypoint, the vehicle is considered to have reached the waypoint). The figure shows that the simulation result from *BraceBind* is highly consistent with the real deployment environment with average error (over both vehicles and across all trajectories) between 5.76% to 11.20%. my analysis determined that the errors result from:

- the physical models used in the experiments are thoroughly tested, they still contain small errors with respect to their representation of the physical world;
- in my testbed, the positioning of the vehicle is recorded by an overhead camera, which can process three frames per second; this results in small positioning errors that impact both the *BraceBind* deployment and the physical deployment; and

- I did not build a model of the battery into the real time simulation environment; I attempted to experiment with fully powered batteries in each experiment, but the battery dissipation does have a small impact on the error between the two deployments.

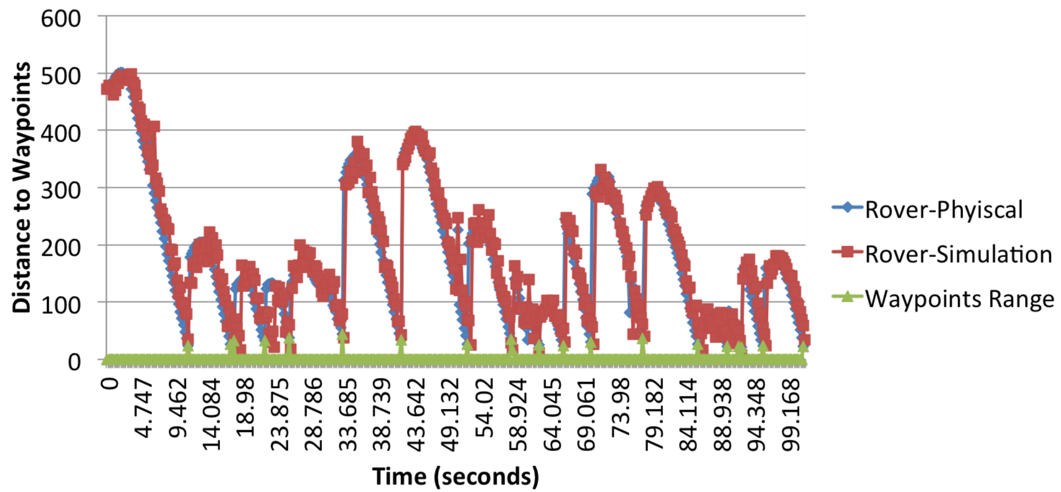

Figure 5.6: *BraceBind* VS Physical Deployment – Linoleum

Figure. 5.7 shows another physical deployment, where the surface is changed to wood. Compared to the Linoleum environment, there are a few bumps on the floor. As the figure shows, though the trajectories of the vehicle are quite consistent, there are spikes now and then in the physical environment deployment that are not exhibited in the simulation environment. my analysis indicates that these spikes are due to the bumps causing the vehicle astray from the planned track. The error range for the real time simulation in this test is from 5.7663% to 11.3299%. This shows that, even with unexpected physical conditions (e.g., bumps), *BraceBind* can still guarantee an acceptable error range. Further, a more detailed model of the physical environment (e.g., one that models bumps in the floor surface) would give real-time

simulation results that a more faithful representation of the physical deployment (although the *trajectories* themselves may not line up if the arbitrary bumps in the real-time simulation do not end up in the same locations as in the physical deployment).

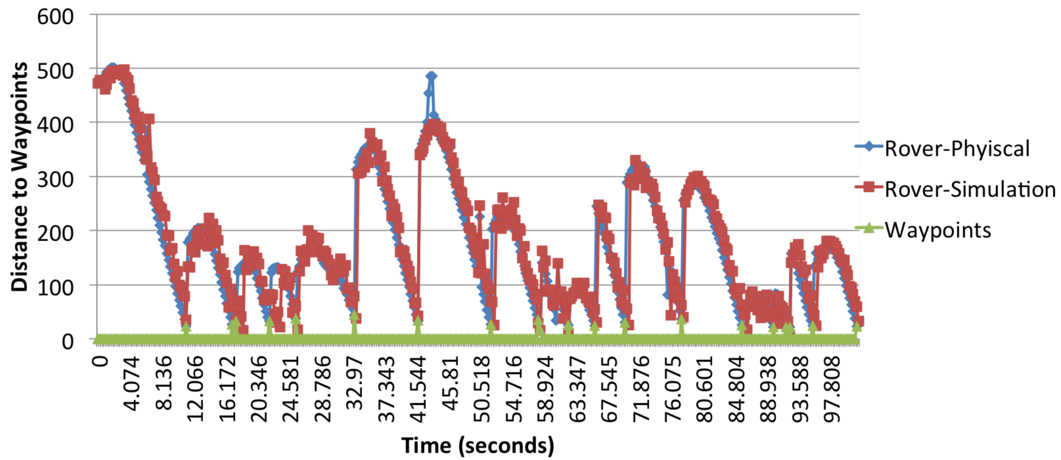

Figure 5.7: *BraceBind* VS Physical Deployment – Wood

During the trial tests before conducting *E2*, I discovered that the co-efficiency of drag provided for the physical model of the grass environment is not accurate (likely because the grass surface in the lab is synthetic grass instead of real grass). However, the results for the grass environment are still interesting because they measure how the use of real-time simulation within *BraceBind* responds if there are relatively large errors for parameters of the physical models. Therefore, in addition to having an incorrect co-efficiency of drag, I also reduced the weight of vehicles by 1/3 without changing the weight parameter in the models. Fig. 5.8 shows that the resulting error range is quite large (from 17.3215% to 81.7556%). However, I can still have a pretty faithful representation of the trajectories, which shows that *BraceBind* is still able to give a rough feel about how the CPS application might

behave in such “uncertain” environments.

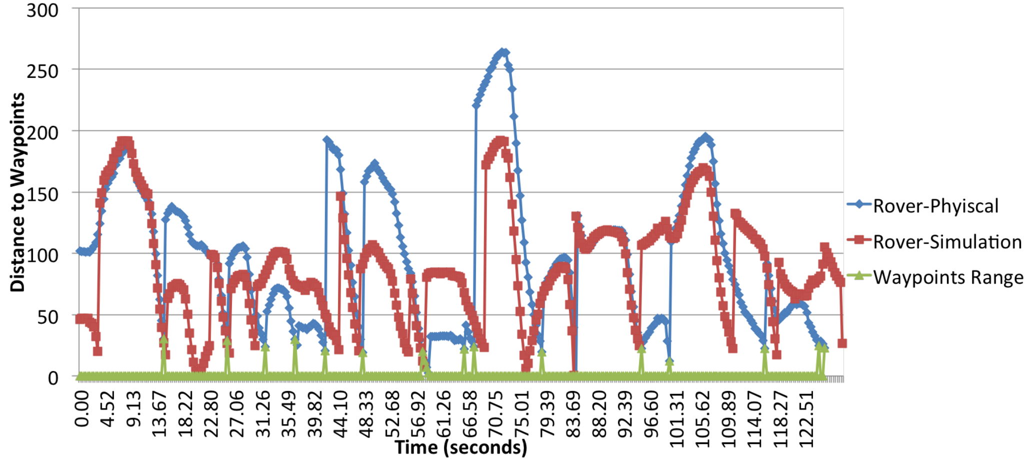

Figure 5.8: *BraceBind* VS Physical Deployment – Grass

5.4.12 BraceBind VS Physical Deployment in Runtime Verification (E3)

In *E3*, I injected 10 errors for each of the six application correctness properties. Since injected errors are all logical errors, not surprisingly these injected errors are detected (which was expected based on my previous work [195,196]; correct error detection is not a contribution of *BraceBind*). Fig. 5.9 shows the additional errors (non-injected ones) that I were able to detect in both the physical deployment and in *BraceBind* as supported by real-time simulation. Six of the errors (for *P2*) were confirmed by the CPS developers to be actual bugs in the implementation. The remaining errors are assumed to be true positives if they were found in the physical deployment and false positives if they were not.

Summarizing the above and the results in Fig. 5.9, runtime verification supported by *BraceBind* instead of the physical deployment had 0 false negatives (i.e., no (known) errors were not detected) and 1 to 2 false positives (i.e., errors that were

"detected" based on the real-time simulation that were not actual errors).

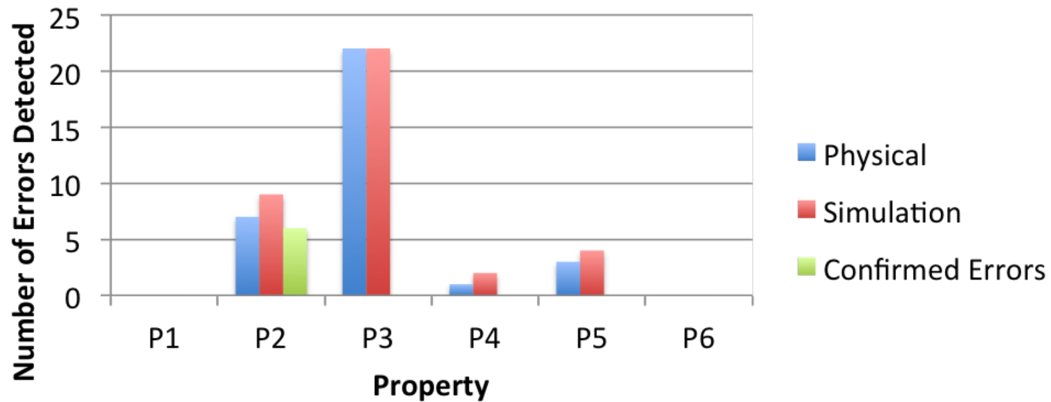

Figure 5.9: *BraceBind* VS Physical Deployment – Linoleum

Fig. 5.10 shows the results from the same experiment on the wood surface. Again, both scenarios found all of the injected errors. Here, however, *BraceBind* backed by real-time simulation actually missed some confirmed errors (for *P2*) and some assumed errors (i.e., those found in the physical deployment for *P4* and *P5*). This implies a likely small false negative rate in the *BraceBind* scenario and a false positive rate that is on par with the prior results. I believe these errors are again caused by the bumps in the wood floor that are not accounted for in the physical model.

For brevity, I omit the results for the grass surface. As expected, there are larger but still reasonable number of false positives (i.e., 5 for *P2*, 2 for *P4*, 3 for *P5*) again due to the (intentional) inaccuracies in the grass model.

5.4.13 Discussion

In addition to the experimental results I have reported here, I have tested the impact of removing time synchronization or using a less accurate time synchro-

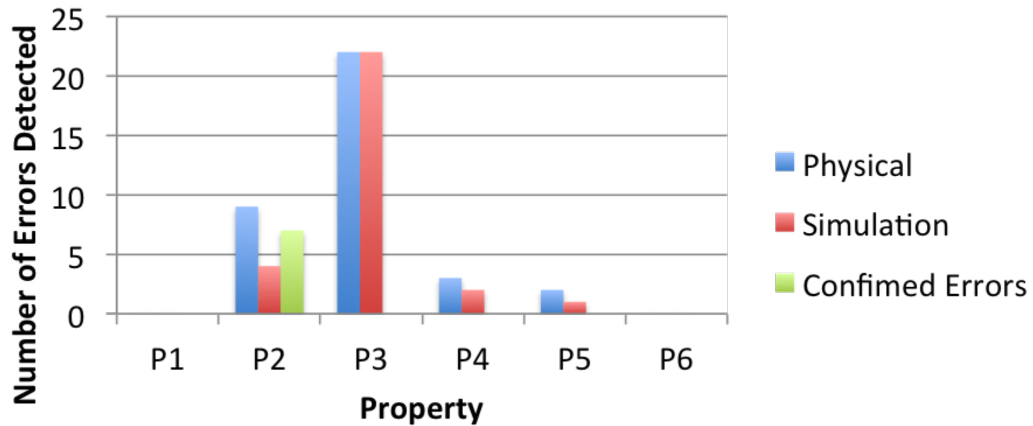

Figure 5.10: *BraceBind* VS Physical Deployment – Wood

nization in *BraceBind*. The result is that vehicles in the simulated environments often get stuck in an endless control loop while trying to reach a waypoint even with an accurate version of PID controller. I found that the position data can become completely out of sync, and the vehicle is not able to reach the waypoint before being given a piece of incorrect position data. I also tried using MQTT[11] instead of ZeroMQ; however, the latency of message delivery with MQTT is much higher, meaning that the output data from the dynamic real-time simulations can not reach the models fast enough for the vehicle's view of the (simulated) physical world to remain in sync.

BraceBind is also able to support LabView models in addition to Simulink models. For instance, I 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 requires much greater computational power than the Simulink converted C++

[11] `http://mqtt.org/`

code, and if I run the combined real-time simulation for more than 30 minutes, the CPU utilization in my test machine quickly climbs over 100%. I believe I need a much more powerful host machine to carry out experiments with the LabView models. My future work will investigate how to measure the requirements and make recommendations of host machine capabilities.

Compared with the state of the art using dedicated simulation servers running all simulation models together, *BraceBind* provides a more scalable and cheaper solution for a real-time simulation environment. Even the suggested somewhat more powerful host machine for LabView models is much more affordable than a dedicated simulation server. Moreover, *BraceBind* combines the real time simulation with the runtime verification seamlessly which is not provided by any off-the-shelf products and is one of the first kind.

The empirical results in *E3* use mainly my previous runtime verification framework as the baseline. As proven in [195, 196], my runtime verification framework has a very low number of false positives with no confirmed false negatives. Moreover, for *P2* where I do have existing logs to confirm the real number of errors, I provide the confirmed errors to prove the trustworthiness of the runtime verification framework and increase the accuracy of the test results.

Though a relatively simple and low speed physical rover is used as the testing vehicle, the physical models I use are quite representative of any electric vehicle models both in terms of number of sub-models/subsystems and types (e.g., sensors, actuators, environment, and dynamic models). As my future work, I am looking into applying *BraceBind* for industrial level CPS (e.g., to express a train control

system in China).

5.5 Research Contributions

In this chapter, I made the following contribution:

Research Contribution 4: I create *BraceBind* as a middleware that enables real time simulation of physical models from heterogeneous simulation platform to inform CPS runtime verification. With effective CPS runtime verification middleware, this work allows detection of subtle bugs originated from developers' uncaught wrong assumptions regarding physical processes in CPS application without repetitive physical deployment which is either too expensive or not feasible. Since *BraceBind* allows two channels of debugging data, one from physical deployment environment (e.g.,sensors and actuators) and another from highly faithful real-time simulation, this capability provides the required flexibility, repeatability, and cost control by CPS runtime verification. Based on this, I am able to provide more efficient and expressive runtime verification middleware.

5.6 Chapter Summary

In this chapter, I created *BraceBind* to provide real-time simulation to inform CPS runtime verification. Firstly, I defined a simple yet expressive model interface specification in XML; I then extended *BraceAssertion* annotations to differentiate physical variables from cyber variables in the implementation and allow physical variables to take values either from sensors (physical devices) or from models (real-time simulation). *BraceBind* is able to automatically convert physical models from

various simulation platforms into executables. I also create highly capable algorithms to grant the generated executables with accurate time synchronization and efficient data integration essential to creating a real-time simulation environment. The extensive case study proves that the real-time simulation created by *BraceBind* is highly accurate. Combined with my prior work on CPS runtime verification, *BraceBind* is able to detect subtle bugs CPS as effectively as in the physical deployment environment without all of the commensurate difficulties of debugging *in situ*.

Chapter 6

Brace: A Middleware for Practical On-Line Monitoring of Cyber-Phyiscal System Correctness

The previous works combined allow detection of bugs both in Cyber and Physical worlds. However, developing and debugging CPS remain significant challenges. Formal Methods, testing, simulation and other state of the art techniques are not sufficient to guarantee required correctness. Runtime Verification (RV) provides a perfect complement to above approaches. However, the state of the art RV tools suffer from either lack of expressiveness or lack of efficiency. Moreover, a few unique requirements of CPS (e.g., predictability and intuitiveness) exacerbate the problem. In this chapter, building on my prior work, I present *Brace*, a middleware for practical on-line monitoring that abstracts and controls the monitoring of local and global correctness properties for CPS. *Brace* provides time synchronization for the events and attributes collected from distributed CPS application nodes and performs event filtering on behalf of applications, while guaranteeing predictable behavior of runtime monitoring in the presence of surges of events from the underlying monitored CPS. Using an extensive case study on a real-world multi-agent CPS deployed both on physical rovers and a simulation testbed, I demonstrate *Brace* as an efficient, effective, and practical runtime monitoring middleware for CPS.

6.1 Introduction

Cyber-Physical Systems (CPS) have been widely applied in various domains, including but not limited to autonomous vehicles, automated health care, smart grid, structural health monitoring, and many other mission-critical applications. However, recent research has demonstrated significant gaps in my ability to perform quality verification and validation of CPS [102,194]. In fact, the *de facto* state of the art for generating correct CPS programs is to deploy the entire system and attempt to debug *in situ* [194]. This is some *ad hoc* form of *runtime verification*, which enjoys some popularity in the wider software engineering community. However, in the context of CPS, significant research challenges remain.

Consider a canonical example of a CPS: an autonomous vehicular system. I consider a multi-agent patrol application that executes atop a fleet of such autonomous vehicles. In this CPS, a set of unmanned vehicles coordinates to achieve a provided global monitoring task. To verify whether this system performs correctly, one must verify that the fleet together achieves the *global* task; to accomplish this, one must also verify *local* tasks assigned to each vehicle. If the global task is specified as the fleet visiting a set of assigned waypoints given some schedule, then each agent's local tasks will consist of the set of waypoints the agent must visit. An important local property to be checked in such an application is whether the choice of a particular vehicle to perform a set of waypoints (as its local task) is optimal relative to a chosen system utility function

As a CPS is often distributed (in the case of my example at a minimum across the multiple communicating vehicles), correctness is also expressed via global

correctness properties. For instance, in my example application, I consider the following three global properties: (1) given that a particular vehicle is assigned a particular task (containing one or more waypoints), no other vehicles are assigned the same task; (2) the number of messages required to be exchanged among the vehicles for them to reach consensus about a task assignment is bounded; and (3) all the tasks must be completed within a given time.

In my prior work [198], I introduced *BraceAssertion*, by which developers can specify local correctness properties in the form of assertions that are associated with the individual components of the CPS program. I also built an offline monitoring framework to check these assertions over traces of program executions. In this work, I make two significant advances: first, I extend the approach to capture *global* assertions, which requires collecting events and properties from distributed CPS nodes, synchronizing time across these events from distributed nodes, and new constructs for expressing quantitative constraints at a global level. Second, I ensure an efficient, effective runtime monitor with *predictable* behavior (e.g., CPU and Memory). As the common approach to ensuring the correctness of a CPS is to execute the system *in situ*, runtime verification is a natural approach to codifying the validation approach. However, since CPS developers are often domain experts and not software engineers (or even trained programmers), overly formal approaches are not palatable to them [194]. Given that runtime verification is my chosen approach and my audience requires both an effective but an intuitive approach to creating specifications, I must take care to balance efficiency and effectiveness, which have been shown to be conflicting in prior work on middleware for runtime verification [18]. For instance,

on one hand efficient synchronization and monitoring algorithms are required to check global properties; on the other hand false negatives (i.e., missed errors) and false positives (i.e., wrongly reported errors) must be minimized. As a further challenge, CPS applications are often deployed to embedded devices or smart phones or in real-time or interactive application domains, and these applications cannot afford any significant impact on the application's timing behavior (especially for devices without a real-time operating system) and thus any performance impact of runtime verification must be carefully mitigated. Existing approaches to runtime verification do not emphasize this aspect; for instance, widely used time triggered online monitoring frameworks use long sampling periods (during which runtime monitors sit idle) to ensure predictability [34]. While CPS require a more flexible approach to setting the sampling period, that alone is not enough; instead, CPS developers need to be able to specify the resources (e.g., time and memory) to allocate to the runtime verification process. In summary, cyber-physical systems demand a practical middleware for runtime verification that is able to address these challenges collectively. More explicitly, I make the following concrete contributions:

- I extend *BraceAssertion* to allow programmers to specify global properties based on events and attributes collected across distributed CPS nodes.
- I introduce *Brace*, a middleware that enables on-line runtime verification of cyber-physical systems and is able to detect violations both in local properties and global properties.
- Within *Brace*, I design a novel linear programming model and load balancing algorithms to guarantee predictable behaviors of runtime monitors even with

unpredictable surges of events.

- I design new synchronization algorithms to guarantee the correct ordering of events and attributes collected from distributed CPS nodes and use these events to check global properties.
- As a backend for *Brace*, I create a novel model backed by automata to filter and aggregate events. This is important for reducing the processing overhead of checking global correctness specifications.
- I extensively evaluate *Brace* with a real-world autonomous vehicle application.

6.2 The Formal Specification: BraceAssertion

I start with my *BraceAssertion* model [198] that allows specifying local properties in an intuitive way using Behavior-Driven Development (BDD) [26], which is designed to be intuitive to developers. To enable off-line verification using *BraceAssertion*, I translated the intuitive BDD-style specifications into State Clock Logic (SCL), which I extended with customized annotations. I then check the resulting predicates using Event Clock Automata (ECA) [9]. In this prior work, *BraceAssertion* supported local properties in components of a CPS and checked them off-line, against traces collected of the CPS's behavior. As a brief summary of the approach, I model 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. For each observation, I also record the timing information that allows us to construct a timing sequence $\Theta = (\bar{\delta}, \bar{\tau})$, which respects *monotonicity* and *progress* (i.e., $\tau_i < \tau_{i+1}$ and $\exists i \in N, \forall j \in R, \tau_i > j$). Using this

prior work, I can specify the property "whether the choice of an agent is optimal after assigned a task" as:

```
When [Assigned a Task]
Then [Schedule is optimal
      WITH schedule
      AND  task]
```

This prior work supports neither the checking of *global* properties, which specify the joint correctness of multiple components of a CPS, nor *on-line* monitoring, or checking the CPS *in situ*, while it is executing. With respect to the specification language, I define three extensions to the semantics in this work.

Defining Sets of CPS Components. To write specifications that reference multiple components of the CPS I must extend the *BraceAssertion* notation to reference properties across sets. In my extended *BraceAssertion*, the *Set* construct groups multiple distributed CPS application nodes into a logical unit that can be referenced in global properties. For instance, in my second example global property (i.e., that the number of messages required to negotiate task assignment is bounded), I need a way to reference a *Set* of active vehicles (in order to check whether they reach a consensus). I add three keywords into *BraceAssertion*: *Set*, *Joins*, and *Quits*. Properties can specify that a given node *Joins* a *Set* after one or more specified local events and *Quits* a set after one or more specified local events. The complete syntax is: "*Set* [set name], *Join* [events], *Quit* [events]". The *Set* definition is uniform across the entire CPS. The events defined for *Joins* and *Quits* are connected with the implementation via annotation, and each generated event is associated with the unique id of the CPS application node generating it. The

events collected by *Brace* help manage *Set* memberships at runtime (Section 6.3.4). Within the "[events]" blocks, multiple events are connected by logical operators (e.g., *And* and *Or*). As a concrete example, for the motivating application, I define the following set: "*Set* [ActiveAgents], *Join* [assigned task]".

Defining Global Events and Distributed Properties. In *BraceAssertion*, the specifications are effectively synthesized into automata, which process events generated at runtime to monitor the correctness of the specified property. The first step of defining a global property is for the user to specify the global event(s) that trigger checking a particular global property. Such a global event is built by aggregating local events that occur across distributed nodes. I create a few constructs in *BraceAssertion* to capture this semantics. To define global events, I extend the original *BraceAssertion* language with a *When* definition specifically for marking a global event. Its format is:

```
When [Exactly (default)/More Than/Less Than]
     [number of required local events]
     [name of local component]
     (set alias)
     [description of (distributed) local event(s)]
     (distributed attribute name)
```

In the definition of global event, "set alias" implicitly defines a set of components based on the specified local components and required local events. The description of the local event is the CPS developer's natural language description; as described in the next section, the developer must use this description in annotations that connect the specification to the source code of the CPS program. The "distributed attribute name" refers to one or more objects or attributes with which

the local events are associated, for instance, "the task" in the assigned task event for an agent.

Consider the first two properties I gave as examples in the introduction. Both make use of a global event that references the state when exactly one agent has been assigned the task of patrolling one or more given waypoints. Such a global event can be specified by the CPS developer as:

```
When [Exactly]
     one
     Agent
     (AssignedAgent)
     assigned to complete a task
     (ChosenTask)
```

In this example, "AssignedAgent" defines an implicit set (as opposed to being defined explicitly via the *Set* definition), where the agent joins the set after the agent experiences the "assigned to complete a task" event (Section. 6.3.4). "ChosenTask" defines the task attribute in the local event "assigned to complete a task". When a CPS developer associates the local event with the implementation using *BraceAssertion* annotations, the developer must also specify a mapping from the attribute "ChosenTask" to a variable in the implementation. The following code-snippet shows how this is done using the *BraceAssertion* annotation. As shown, this process adds two new annotation types to the existing *BraceAssertions*: `DistributedEvent`, which binds a distributed event to a method invocation (or a single statement) and `DistributedAttribute`, which identifies a distributed attribute inside a method or a class.

```
//somewhere inside the class Agent
```

```
@DistributedEvent(name="assigned to complete a task")
@DistributedAttribute(name="ChosenTask",
                      type="Object",
                      variableName="task")
public synchronized void assignTask(Task task){
  ....
}
```

In the above event specification and annotations, *Distributed Attributes* refers to local attributes in each CPS application node that are used to specify global properties. In CPS, such distributed attributes are ubiquitous. For instance, in the motivating application, there is an attribute that records the task(s) assigned to each agent, and there is another distributed attribute that captures the number of messages needed to reach a particular consensus. In other applications, distributed attributes may capture the power usage of each node, the pressure reading of a particular tire, etc. Aggregate functions (predicates) are often associated with distributed attributes to collectively specify global properties. To support this, I add four built-in predicates to *BraceAssertion*: *Sum*, *Average*, *Max*, and *Min*. Moreover, CPS developers can express complex and application specific global properties not only with existing predicate logic support, but also with "set" (explicitly defined), "distributed attribute name", "set alias" (implicitly defined).

Defining Global Properties. This finally enables us to specify global properties by combining these components. For instance, to express my first example property, which requires that exactly one vehicle is assigned a particular task, I write the following *BraceAssertion* specification:

```
Given [Globally]
When [Exactly]
```

```
     one
     Agent
     (AssignedAgent)
     assigned to complete a task
     (ChosenTask)
Then [no other vehicles have the same task
      WITH ChosenTask in AssignedAgent
      AND ExecutionPlan in ActiveAgents]
```

In this specification, "Globally" explicitly differentiates a global *BraceAssertion* from a local one, the text description in *Then* but before "With" is a predicate definition that is associated with a customized function in the implementation via yet another *BraceAssertion* annotation [198]. The predicate evaluates whether any other vehicle has the same task with respect to two quantified parameters "ChosenTask" and "ExecutionPlan". The parameters are quantified using a combination of the *BraceAssertion* annotation (using the "DistributedAttribute" annotation), which determines the values the parameters will have at runtime, and the set these two parameters belong to ("AssignedAgent" and 'ActiveAgent", respectively) which determines the distributed nodes (agents) these parameters belong to.

As another example, my second sample global property would have the following specification:

```
Given [Globally]
When [Exactly]
     one
     Agent
     (AssignedAgent)
     assigned to complete a task
     (ChosenPoint)
Then [Sum (messages required to reach consensus
      WITH ChosenTask in AssignedAgent
      AND Consensus in ActiveAgents)
      Less Than 10]
```

In this specification, "Sum" is the pre-defined predicate introduced previously, "Less Than" (along with "Equal" and "More Than") are newly introduced keywords to support pre-defined predicates. The contents of the parentheses (i.e., "messages required to reach consensus...") are associated with a customized function in the implementation using an annotation that returns a count of the number of messages sent to reach a consensus on the assignment of the task to the selected agent.

6.3 The Brace Middleware

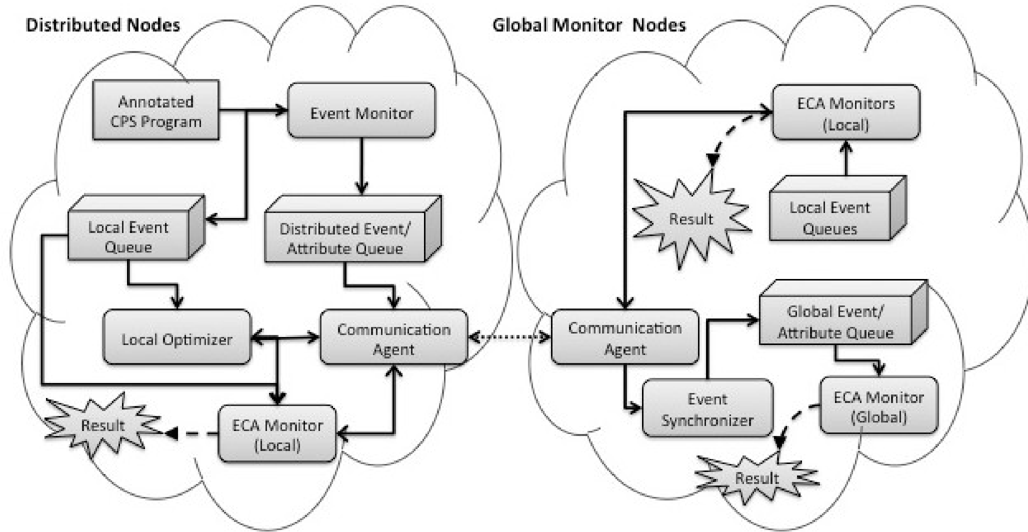

Figure 6.1: Brace Architecture

Fig. 6.1 shows the overall *Brace* architecture, including the cyber components (both hardware and software) of the CPS on the left hand side and the dedicated monitoring components on the right hand side. Within the local CPS nodes, the *Event Monitor* and *ECA Monitor*[1] are synthesized automatically from *BraceAsser-*

[1] I use ECA monitor to refer my property monitor built on top of Event Clock Automata.

tion specifications, as described in [198]. However, I extend the role of the *Event Monitor* to also monitor distributed events and attributes used for the specification of global properties. In addition to using the monitored events to check the local properties, the *Event Monitor* stores the information required for distributed events and attributes in the *Distributed Event/Attribute Queue* (DQ), which can then facilitate sharing this event and attribute information for the purpose of monitoring global properties.

To enable reliable transfer of distributed events and properties from each node to the global monitor nodes, I introduce a *Communication Agent* both at the local CPS application nodes and at the global monitor nodes. I use MQ Telemetry Transport (MQTT) [124] to implement the communication among the different distributed components.[2] Each *Communication Agent* on a local CPS node aggregates numeric data to minimize network overhead and creates *bundle*s to send data in batches. On the other side, the *Communication Agent* at each global monitor node passes received distributed events and attributes to the *Event Synchronizer*, which synchronizes events and attributes received from the distributed CPS nodes. The result is the fully ordered *Global Event/Attribute Queue*, which feeds an event automata that filters and aggregates these events to reduce the number of events that must be processed by the *Global ECA Monitor*.

More details about *Global ECA Monitor* and these *Event Automata* in Sec. 6.3.4. As described in detail below, I allow the local monitors to be transitioned to the

[2]As MQTT is known to have milliseconds message latency and guarantee message delivery with specific settings.

global monitoring nodes when the burden of monitoring becomes too great for operating on the CPS devices. To support this load balancing, each global monitor node keeps a copy of local monitors and can dynamically assign a local event queue for a distributed node when that node requires load balancing.

Because a main goal in this work is to guarantee predictable behavior of runtime monitoring, I introduce a *Local Optimizer* on the local CPS application nodes, which uses a novel linear programming model to find an optimal solution for the local monitor automata. To deal with situations in which the local optimizer cannot find an optimal solution, I also introduce load balancing through the *Communication Agent*, which stops the local monitors and delegates local monitoring tasks to the global monitor nodes. Given this overview of the *Brace* architecture, I next present the details of the components of Fig. 6.1, including implementation aspects.

6.3.1 Local Optimization Controller

The online local property monitor (ECA Monitor) that sits in each distributed node is a time-triggered automaton [34, 185] to guarantee predictability of the observed system. In this timed-triggered mode, the monitors are activated periodically according to a pre-defined sample period to read events from the local event queue, make the appropriate state transitions, and detect any resulting violations of the monitored properties. In [34, 185], the sampling period is a constant determined by an off-line linear optimization model specially designed for sequential programs. In *Brace*, I give CPS developers more granular control over the behavior of runtime

monitors by allowing them to specify the minimum requirement for the idle time where the runtime monitor is not activated (IT), the maximum time for a runtime monitor to run after activated but before putting into "sleep" again during a CPS execution (AT), and the maximum memory for the *Brace* middleware to allocate to storing unprocessed events while the runtime monitor is "sleeping" (BS). Moreover, to handle the often unpredictable nature of monitored events in CPS (e.g., events in CPS may often come in bursts), I determine the optimized values for IT, AT, and BS dynamically, based on a linear programming model. More specifically, the *Local Optimizer* in each distributed node continuously monitors the rate at which the local events are generated, i.e., the event creation speed (y events/second) and the rate at which the local property monitors can process the events, i.e., the event processing speed (x events/second). From these two statistics and the ranges specified by CPS developers, I devise the following linear programming model to find the optimal solution.

my Linear Programming model's objective function is:

$$minimize(y * IT + (y - x) * AT) \tag{6.1}$$

That is, the objective is to minimize the required buffer size for the next Idle Time and Active Time combined. The linear optimization model is further constrained by the minimum Idle Time and maximum Active Time specified by the CPS developer. I use an off-the-shelf Linear Programming solver [168] that is fast and lightweight[3] to find optimal solution.

[3]From trial and the testbed in the chapter, the running time for the solver is within 2 milliseconds

I now frame some of the key constraints that underlie my linear programming model. First, and simply:

$$IT \geq mIT \tag{6.2}$$

This constraint is used to guarantee that the application's observed idle time, i.e., the time when no monitoring is occurring, meets the required minimum idle time (i.e., mIT) provided by the CPS developer.

Second, and similarly:

$$AT \leq maxPT \tag{6.3}$$

This constraint guarantees the maximum observed period of time in which the run-time monitors are allowed to execute is constrained by the CPS developer's specified bound. $maxPT$ is also configurable for each CPS application.

Thirdly:

$$y * IT \leq BS \leq maxMemory \tag{6.4}$$

This constraint gives the constraints over how much memory can be allocated to store unprocessed events during each segment of Idle Time. The required maximum memory (i.e., $maxMemory$) is provided by the CPS developer. Recall that y captures the average event generation speed[4].

The optimal solution, if found, provides a balance between how much memory is allocated, how much time is given to the monitors for execution, and how

[4] *Brace* records the event generation speed from the latest sampling period as a prediction for the next idle time.

much time is given to applications to run on their own without being affected by monitors. If there is no optimal solution, the local optimizer switches to a *Delegation* mode that turns off the local monitor and instructs the communication agent to send the local monitoring tasks to global monitoring nodes. In such situations, the local optimizer continues to monitor the event queue and calculate the optimal solution but with the last constraint changed to:

$$BS \leq safeBuffer * y * IT \tag{6.5}$$

This loosens the upper bound for memory overhead to attempt the application-specific buffer value *safeBuffer* to make sure that the length of the event queue is large enough to hold events for the give number of idle periods. The constraint is used to minimize the possibility that while in the transitions between *Delegation* mode and *Local* mode, there are no lost (overwritten) events in the buffer.

While the local monitor is in *Delegation* mode, if after N (configurable per application) sampling periods, the updated linear programming model can *always* find an optimal solution, the local optimizer switches back to *Local* mode. The local optimizer instructs the local communication agent to send a *Restore to local* message to the global monitor nodes, waits for latest instance of local monitors to be returned, and restores and reactivates the monitor instances locally.

6.3.2 Communication Agent

As described in the overview above, the *Communication Agent* in each of the CPS local nodes and in the global monitor nodes is responsible for coordinating

checking of global properties and for checking delegated local properties. To support my implementation of the *Communication Agent*, I rely on MQTT. MQ Telemetry Transport (MQTT) [124] is a scalable and light-weight messaging protocol based on publish/subscribe; Its capabilities of minimizing network bandwidth and ensuring reliable message delivery make it a good candidate for devices and connections like those found in CPS. In my prototype, I use Paho (an MQTT client library) [147] and Really Small Message Broker (RSMB), an open-source messaging server implementing MQTT with very small overhead (80KB of storage space and 200KB or less of memory) [52]. In essence, the communication agent uses a lock-free linked list to store the data related to local events (if load balancing is required), distributed events, and distributed attributes, including their time stamps. The agent periodically aggregates data in a bundle (batch) to minimize network overhead. The bundle contains a message header with a unique id (e.g., mac-address) of the CPS application node and a creation timestamp and sequence number (later used for detecting missing bundles and for message synchronization across bundles).

Because MQTT is built on the publish/subscribe paradigm, each bundle is published to a shared topic subscribed to by all of the global monitor nodes. Because I only want one global monitor node to be responsible for each bundle, I implement a cancel protocol on top of MQTT's publish subscribe mechanism. Specifically, I use a second MQTT topic shared among the global monitor nodes. When a global monitor receives a bundle and intends to process it, it publishes the bundle's id and sequence number, along with the timestamp at which it was received. If there are duplicates, the global monitor node that received the bundle first according to the

reception timestamp would keep the bundle while others just simply discard it.

The other role of the *Communication Agent* on the local CPS application nodes is to handle load balancing of local property checking. When the node's local optimizer decides to switch the monitoring mode back to *Local*, its *Communication Agent* sends a control package of *Restore to local* to all reachable global monitoring nodes. If one of the global monitor nodes replies with a snapshot of the current instance of the application's local monitors, the *Communication Agent* passes the snapshot to local optimizer to restore it. At the global monitor node, the *Communication Agent* passes the received distributed events and attributes to the *Event Synchronizer* for further processing. The global monitor's *Communication Agent* also receives request to accept delegation of local property monitoring. After receiving a *Delegation* request from a CPS application node, the global monitor's *Communication Agent* activates replicas of the specified local monitors and restores local events to another concurrent queue. While it is responsible for a delegated local monitor, the global monitor's *Communication Agent* maintains the local monitor's event queue using events received from the *Communication Agent* on the associated local CPS application node. When the global monitor receives a *Restore to local* request, it ceases loading the local monitor's event queue. When the local event queue is exhausted, the global monitor's *Communication Agent* deactivates the indicated local monitor, takes a snapshot of current instance, and sends it back to the CPS application node.

For communication efficiency, I combine the tasks of the *Communication Agent*, sending network packages that contain both distributed events and infor-

Algorithm 5 Data Preparation

```
1: t ← currentTime
2: for ∀e ∈ DQ do
3:     if e.time ≤ t then
4:         eList.add(e)
5: addToCQ(eList)
6: if delegateMon then
7:     if init then
8:         for ∀m ∈ activeLocalMonitors do
9:             monList.add(m)
10:        init ← false
11:        addToCQ(monList)
12:    else
13:        for ∀le ∈ localEventQueue do
14:            if le.time ≤ t then
15:                leList.add(le)
16:        addToCQ(leList)
```

mation about delegated local monitoring. Algorithm 5 shows the raw data that is gathered by the *Communication Agent* at a CPS application node. The *Communication Agent* starts by marking the current time. It then selects all of the distributed events and attributes (bounded by the marked current time) from the distributed event and attribute queue (DQ) and adds them to the *Communication Queue* (CQ) (lines 1-5). The CQ is monitored by an MQTT agent and published periodically. If the local optimizer has decided to delegate local monitoring (line 6), initially the *Communication Agent* adds the instance of each active local monitor to a linked list that it also puts in the CQ (lines 7-11). Finally, if the node is in delegate mode and not just starting delegate mode, the *Communication Agent* adds all the events in the local queue (bounded by the current time) to CQ (lines 13-16).

Algorithm 6 shows the algorithm behind how the data is published by the *Communication Agent* in a CPS application node; it is called in a thread separate from the main CPS application execution. The *Communication Agent* iterates through each item in the communication queue (CQ), if the item is either a local

event or local monitor instance, no further processing is required, and the *Communication Agent* simply sends the item in its raw form (lines 3-4). If the item in the *CQ* is a distributed event and contains event objects, the data cannot be aggregated and is therefore sent directly (lines 7-9). Otherwise the events with the same name are aggregated into a single event with two time stamps. The first time stamp records the first occurrence of the event while the second one records the last one; the value of the aggregate is the number of occurrences of the event (lines 11-12). If the item is a distributed attribute and of numeric value, the data is aggregated in a similar way, but the aggregate also includes the average, maximum, and minimum value of all attributes with the same name (lines 13-16). Finally, a non-numeric attribute is not aggregatable and is instead sent directly (line 18). In the end, the *Communication Agent* creates a bundle that contains all of this data with a message header containing the node's unique id of the distributed node and an increasing sequence number.

6.3.3 Event Synchronizer

As distributed events and attributes are received at the global monitor nodes, they must be synchronized so that the required properties can be checked against the generated events. The purpose of the *Event Synchronizer* in a global monitor node is to guarantee a total ordering of distributed events and attributes collected from across the distributed CPS application nodes, which is essential to being able to check global properties at runtime. The *Event Synchronizer* achieves this using three levels of synchronization.

Algorithm 6 Main publish algorithm (Local Node)

```
1: sendList ← newList()
2: for ∀c ∈ CQ do
3:     if c.type ∈ {localEvent, monitor} then
4:         sendList.add(c)
5:     if c.type = distributedEAs then
6:         for ∀de ∈ c.list do
7:             if de.isEvent then
8:                 if de.hasEventObjects then
9:                     sendList.add(de)
10:                else
11:                    ale ← aggEvent(de.name, de.time)
12:                    sendList.add(ale)
13:            if de.isAttribute then
14:                if de.isNumeric then
15:                    ala ← aggAttribute(de.name, de.time, de.value)
16:                    sendList.add(ala)
17:                else
18:                    sendList.add(de)
19: seqNumber ← newSeqNumber()
20: nodeId ← getMacAddress()
21: publish(sendList, seqNumber, nodeId)
```

6.3.3.1 Synchronizing the Events of a Single Node

To guarantee a total ordering of messages from each individual CPS application node and to detect missing bundles as a result of network losses, the *Event Synchronizer* checks the sequence number embedded inside each bundle. Algorithm 7 gives the algorithm that the *Event Synchronizer* uses to synchronize bundles received for a single CPS application node. First, based on the bundle's unique node id and sequence number, the *Event Synchronizer* knows the next sequence number expected from the node (lines 1-4). If the *seqNumber* of the next bundle received is less than the expected next sequence number, the bundle is a duplicate and the *Event Synchronizer* just discards it (lines 5-6). If the *seqNumber* of the received bundle is larger than the expected next sequence number, the network package is stored in an Out-of-Sync Queue (OSQ) reserved for future use (lines 7-8). Finally,

Algorithm 7 Network Package Synchronization

```
1: bundle ← receivePublication
2: nodeID ← bundle.nid
3: seqNumber ← bundle.sno
4: nextSeq ← getNextExpected(nodeID)
5: if seqNumber < nextSeq then
6:     return
7: else if seqNumber > nextSeq then
8:     storeOutSyncQueue(bundle, nodeID)
9: else
10:    SQ.add(bundle)
11:    nextSeq ← seqNumber + 1
12:    storedList ← retrieveQueue(nodeID)
13:    for ∀bd ∈ storedList do
14:        if bd.seqNumber = nextSeq then
15:            SQ.add(bd)
16:            nextSeq ← seqNumber + 1
17: storeNextExpected(nodeID, nextSeq)
```

if the received *seqNumber* is the expected one, the bundle's contents are pushed into the Synchronization Queue (SQ) (line 10). In this last case, the *Event Synchronizer* also looks into the OSQ (which is ordered by time) to find any subsequent bundles that were received out of order and pushes them to the SQ (lines 11-16).

6.3.3.2 Synchronizing Events across Multiple Nodes

Merely guaranteeing a total ordering for events and attributes within each CPS application node is not enough; the *Event Synchronizer* also needs to synchronize events and attributes *across* multiple CPS application nodes, for which it uses the *Synchronization Queue* (SQ). I rely on clock synchronization achieved using NTP. Also, as discussed next, I introduce a synchronization error buffer, which, in addition to other purposes described below, diminishes the need to cater to NTP's error bound. However, it is still not possible to simply sort the events and attributes in the global queue by their timestamps because network latencies may cause a bundle from some node N_x, which contains an earlier event to have not arrived before

the bundle from some node N_y, even if the event(s) in the latter occurred later. The challenge is to prevent the events from N_y from being placed in the global queue and processed by the global monitor before the events from N_x arrive.

To address this challenge, I built a buffer inside the *Event Synchronizer* to pre-sort the arrival events and attributes by timestamp, and I use Algorithm 8 to guarantee time synchronization of events and attributes that are output to the global queue for processing by the global monitor(s). The algorithm assumes that the maximum delay (δ_{max}) for a network packet is known and bounded practically, this can be achieved experimentally in a set-up period. The algorithm starts by detecting whether the buffer has reached a pre-determined size limit or a pre-set flushing period is complete (line 1). If the result is positive, the synchronization process enters the *aboutToFlush* stage if there are events in the buffer (lines 7-8). The process records the last event (*lastEvent*) (line 11) and starts a count down period that is bounded by δ_{max} (lines 9-11), which waits for any potentially missing bundles, which could potentially contain events that occurred before *lastEvent*. When this waiting period ends, all the received events up to *lastEvent* are ordered (by timestamp) and stored in the *Global event/attribute Queue* (GQ) (line 4) before the process is re-initialized (line 5).

The above synchronization algorithm guarantees the total ordering of messages across the CPS application nodes. I also built into the *Event Synchronizer* to logically synchronize distributed events with distributed attributes. Distributed events cause state transitions in the global monitor, while distributed attributes are used to generate corresponding events that determine whether the automaton ends

Algorithm 8 Synchronization Across Nodes

```
1: if flushTimeUp ∨ bufferSize ≥ maximumSize then
2:     if aboutToFlush then
3:         if countDown ≥ flushTime then
4:             flushUntil(lastEvent)
5:             initBuffer
6:     else
7:         if buffer.size ≥ 1 then
8:             aboutToFlush ← True
9:             countDown ← currentTime
10:            flushTime ← currentTime + δ_max
11:            lastEvent ← getBufferEnd
```

up in an accepting state or rejecting state. I refer to the latter events as *Global Predicate Events* (GPE). For instance, in the sample global property where the number of messages to reach a consensus of task assignment shall be bounded, the number of messages shall be bounded, as specified in the "Then", is treated as a GPE as it is a predicate defined globally and its evaluation result is treated as an event.

6.3.3.3 Synchronizing and Aggregating Events and Attributes

In this final step, the *Event Synchronizer* aggregates distributed events from one or more CPS application nodes specified in *BraceAssertion* specification into global events (which are the atomic events recognized by the global monitor). Another task of the *Event Synchronizer* at this stage is to locate corresponding distributed attributes, which is essential to generate the *Global Predicate Events* (GPEs), which triggers state transitions either into accepting or rejecting states detectable by the global monitor.

To aggregate distributed events into a global event[5], the *Event Synchronizer*

[5]Based on the global specifciation. For instance, if the specification is "When [More than two agents is assigned to reach a waypoint]", then the first three arrived events "Assigned to reach a waypoint" will be aggregated at the global monitor node as a global event.

assigns an event id for the global event, and the *Event Synchronizer* monitors the first and last occurrence of the contributing distributed events, and records the timestamps for these two occurrences as τ_{start} and τ_{end} (If multiple distributed events are aggregated into one global event, I take the earliest and latest timestamps as the time duration). The two timestamps define the time span of the global event. Since *Brace*'s global events are often associated with distributed attributes, it is essential to synchronize the timing between a global event and its matching distributed attributes. For instance, in my first example property, the global event "one Agent assigned to complete a task" is associated with the distributed attributes "ChosenTask" and "ExecutionPlan". The time span of the distributed attributes is determined *collectively* by the time span (τ_{start} and τ_{end}) of the matching global event and any timing constraint (τ_c) specified in the specification. Timing constraints from specifications may indicate application-level timing requirements, for example that an event occurs within a specified time of another event. Distributed attributes for each global property are not only filtered temporally using the above timespan but also based on the CPS application nodes they originated from. The *Event Synchronizer* determines the originating node for each distributed attribute using the unique node ID in the bundle. If there are duplicate distributed attributes in the same time span for the same node, an aggregation process is invoked to create one aggregated attribute. The *Event Synchronizer* guarantees that, for a given correctness predicate that each distributed attribute from a set of distributed nodes contains a unique or aggregate value and that the entire collection of values falls into the right time span.

6.3.4 Global Property Monitors and their Automata

Brace's global property monitors detect violations of specified properties in much the same way that the local monitor does [198]. Several additional challenges exist in making the global monitor work correctly, specifically so that it can simply read an event queue and make state transitions without worrying about the details of how these events are generated. To support this, I create a *Set Manager* and invent two event automata.

Set Manager. As described previously, a *Set* consists of nodes, each of which *Joins* after a specific event occurs to the node and *Quits* after another specific event occurs. A *Set* manager monitors distributed events associated with *Set* definitions and dynamically manages each *Set* defined by the CPS application through the *BraceAssertion* framework. [6] The *Set* manager is essential in the generation of Global Predicate Event (GPE).

Global Event Automata. Each global event is determined based on the contributing distributed events collected from the CPS application nodes as described above. Each distributed event can be used by more than one global event. To generate global event efficiently, I create a single state machine for generating each global event based on the contributing distributed events. The state machine has an initial state (*Not Firing*) and an accepting state (*Firing*). The arriving of the accepting state triggers an action to put the global event along with the timestamp of reaching the accepting state to the internal event queue (*Global Event Queue*)

[6] *Set* manager sits in the global monitor node.

read by the global property monitor. Each global property monitor may consume multiple global events; these events are each generated by their own global event automata. The intermediate states of the global event automata are determined by the number of distributed events specified in the *BraceAssertion* and the constraints on and relationships between those distributed events, also specified in the *BraceAssertion*.

For example, in the global properties of the motivating application, I define one global event as:

```
When [Exactly]
     one
     Agent
     (AssignedAgent)
     assigned to complete a task
     (ChosenTask)
```

In this case, there is one intermediate state that can be reached from the initial state upon receiving an event indicating that one of the CPS application node's agent was assigned a task. If "*More Than*" is used, then there are two intermediate states[7] and the accepting state can only be reached upon receiving two specified distributed events, each from a unique node. When a global event is generated, the timespan of the participating distributed event is used to generate a Global Predicate Event (GPE) as discussed next; and the only purpose of creating GPE is to filter events so that the global monitor can only accept those limited number of event signal to avoid creating a big state transition table before hand and effectively avoid state explosion issue commonly see in applying automata theory.

[7]There is no order for events, any event of "assigned a task" is considered as an intermediate state. The automata is pretty much like a count-down automata

GPE Automata In BraceAssertion, a user can create a boolean predicate which accepts any number of specified distributed attributes as parameters, and evaluates the predicate at runtime. If the evaluation result is *True*, an associated event (GPE) along with the current timestamp are pushed into the Global ECA Queue (GQ). User is also allowed to create an application-specific predicate with required return value for a pre-defined predicate (e.g., *Sum*) and compare the output of the pre-defined predicate with a user specified value using a predefined set of operators (e.g., *Less Than*), both defined in Sec. 6.2). The boolean result is processed the same way as user-defined boolean predicate. For each GPE, I also create a automata with similar reasons as Global Event. The GPE automata is activated by the generation of the corresponding global event (e.g., defined in *When*); and the inputs are distributed attributes that are time-bounded by the corresponding global event, and value-bounded by the *Set*(s) as each distributed attribute is always associated with a *Set* (default *All*, which refers to all nodes). The starting state is *Not Firing*, and there is one accepting state and another rejecting state. The accepting state is triggered when the predicate is evaluated as *True* and the corresponding GPE along with the current timestamp are pushed into GQ; and the rejecting state is triggered when the predicate is evaluated as *False* and a complemented GPE (created during global monitor synthesis) along with the current timestamp are pushed into GQ. The intermediate states are determined by the number of parameters (distributed attributes) required by the user defined predicate; and unlike the intermediate states for global event state machine, the intermediate states for GPE are ordered. For instance, for *Global Property 2* for the motivating application, GPE is defined as

"*Then* Sum of (total messages required to reach consensus with ChosenTask in AssignedAgent And the consensus in Active Agents) Less Than xx". The user defined predicate is "total messages required to reach consensus", that is associated with the implementation using specifically designed annotation [198], the first intermediate state in the generated state machine is for "ChosenPoint" (as constrained by the *Set* - AssignedAgent), and the second intermediate state is for "the consensus" (as constrained by the *Set* - Active Agents). The transitions to these intermediate states are triggered by completing the calculation of the value for each distributed attribute (if corresponding distributed attributes are aggregatable, aggregate distributed attributes for each node in the specified *Set* in the time span; otherwise take the last value for each node in the *Set* in the time span). From the last intermediate state, there are two transitions, one to the accepting state and another to the rejecting state. The two transitions are triggered by passing the value of each distributed attribute in the order to the specified predicate, and evaluate the predicate (if the predicate is boolean) or passing the predicate result to the built-in predicate and the return value is compared against a user specified number using a built-in operator. The result is a boolean and determines which transition to be activated. When reaching either accepting or rejecting state, the corresponding event and timestamp are pushed into GQ.

In [196], using combinatorial analysis I prove the runtime cost of checking global and local properties (both *Local* and *Delegation* mode) is polynomial. I also implemented a Java prototype of *Brace* and conducted a thorough empirical study on a real multi-agent patrol system to analyze the capabilities of *Brace* to deal with

identified challenges in CPS runtime verification.

6.4 Case Study and Evaluation

I evaluate the *Brace* middleware using an existing robot planning system, which is a distributed version of Generalized Partial Global Planning (GPGP) [116]. This system's planning algorithm is a version of Anytime A* that is customized to distributed planning for a group of mobile vehicles; this is the application I have used for examples.

6.4.1 The Case Study

To use *Brace* to test different aspects of the system, I used two evaluation applications that already existed before *Brace* based on the robot planning system and deploy the applications to two different testbeds. I used *Brace* to find bugs that I intentionally injected, but along the way, I also uncovered bugs in the applications that were unknown to the developers.

The first evaluation application is a Rover Patrol application that is built in Android and deployed to a Samsung Galaxy S3 phone to control a Rover 5 Robot Platform[8]. This platform uses four independent motors, each with a hall-effect quadrature encoder and gearbox. I asked the application developers to annotate the code using my customized annotation class to identify variables and specify the predicate functions required to check six application correctness properties:

LP1 when a vehicle is assigned a task, the vehicle shall have this task in its local

[8] `https://www.sparkfun.com/products/10336`

scheduler;

LP2 the completion time for a given task is bounded;

LP3 when a vehicle is assigned a task, the schedule for this task is optimal;

LP4 the integral of cross track error is bounded (i.e., the vehicle is not weaving);

LP5 the duration of the main control loop is bounded; and

LP6 the number of messages for each task negotiation is bounded.

All of these properties are *local* to a single CPS application node, but these properties are hard to verify by hand because they are related to the distributed algorithm, timing requirement, and the control algorithm. I conduct the experiments in a laboratory environment (*Android Test Bed*), where there are three surfaces to mimic different physical deployment environments[9] (wood[10], grass[11], and linoleum[12]). The environment also contains an overhead camera that is used for both positioning and for recording the tests. Through this evaluation application, I aim to assess *Brace*'s *effectiveness* at detecting local properties related to control algorithms and timing, both of which may be significantly affected by different physical environments.

The second evaluation application models a utility-based distributed scheduling multi-agent system for the robotic patrol application. It follows a blackboard architecture in that a task structure is shared among multiple coordinating CPS application nodes; this blackboard updated concurrently by the scheduling and execution components agents on those nodes. Each agent maintains a snapshot of its

[9]The controller algorithm, for instance, has different behavior in each environment.
[10]A sample video for wood is at `https://goo.gl/PkxDa4`
[11]A sample video for grass is at `https://goo.gl/Vv3fF2`
[12]A sample video for linoleum is at `https://goo.gl/s097Ag`

current state of execution and a combined task structure. The use of a structurally rich, current, combined task structure for representing the agent's execution state allows interleaving of atomic actions that belong to different tasks, avoiding redundancy, and rescheduling only the portion(s) of a task that have not been completed yet. The complexity of this application lies in its concurrency and the correctness of the scheduling algorithms, which are very difficult to check. In this application, I test three local properties:

LP7 when a vehicle is assigned a task, the vehicle shall have this task in its local scheduler

LP8 an agent accomplishes an assigned task within a time bound; and

LP9 each agent's schedule is optimal for that agent.

Besides these local properties, I also test three global properties:

GP1 there are no duplicate task assignments (i.e., each task is assigned to only one agent);

GP2 the number messages required to reach a consensus is bounded; and

GP3 the global patrol must eventually finish.

I deployed and ran my evaluations for this second application on a smaller replica of the ORBIT testbed [161] (*Orbit Test Bed*). This testbed has 40 physical ORBIT-like machines (i.e. *nodes*) based on a VIA 1Ghz, 512 MB board [99, 189]. These nodes are interconnected via multiple networking technologies, including two separate wired 1GB Ethernet networks, which guarantees the separation between control

traffic and experiment/data traffic in my evaluations. Moreover the clocks of these nodes are also synchronized via NTP on the same LAN, which ensures offsets between nodes smaller than 5ms. Through this evaluation application, I aim to assess *Brace*'s *effectiveness* in checking global CPS properties and its *efficiency* in online monitoring, including in the face of surges of CPS events.

I discuss the empirical results for each testbed, and concluded with a validity discussion.

6.4.2 Android Test Bed

In the Android test bed application, I injected ten random logical errors that caused violations in each of the six local properties. *Brace* detected all of these logic errors. I also found that *Brace* was able to detect more errors in these properties that related to data collected from the rovers' sensors and thus related to the physical deployment.

As one example, Figure 6.2 shows the errors that *Brace* found for the trail run on the Linoleum surface. As shown, *Brace* found an additional 22 errors (over the 10 I injected) for ***LP3***. I verified with the CPS application developers that these were in fact legitimate (previously unknown) errors related to the scheduling algorithm that resulted in sub-optimal solutions when relying on actual position data from the sensors. In the case of ***LP2***, *Brace* found an additional seven errors over the 10 injected ones, six of which were confirmed to be legitimate errors related to an inefficient implementation of the controller loop. The seventh was a false positive.

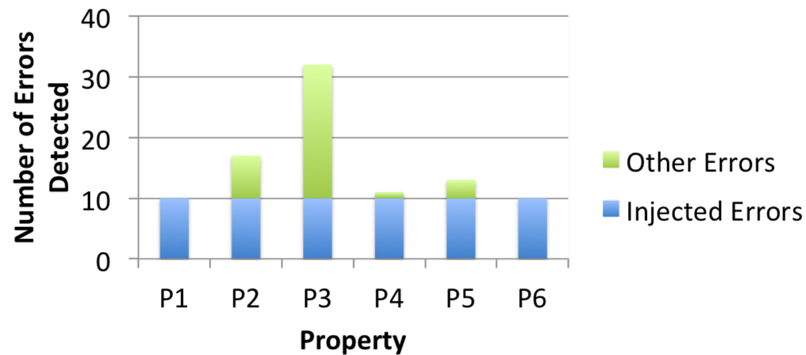

Figure 6.2: Linoleum Deployment

For brevity, I skip the results of Grass and Wood floor deployment. In summary, in both environments, *Brace* was able to detect all of the injected errors and to report additional true bugs in the application (e.g., I found bugs related to the fact that the rover unexpectedly weaves on the grass surface due to the unexpected friction level).

6.4.3 Orbit Test Bed

To evaluate *Brace*'s efficiency using the Orbit testbed,I designed the following four experiments:

- *Baseline*: I run the original application with an increasing number of statically determined tasks: 4 (default), 48, and 384 (to increase the number of concurrently executing monitors).
- *Experiment 1*: I re-run *Baseline* with the application annotated and instrumented with an increasing number of specifications (in terms of local and global properties) (the first local property, three local properties, and three local and three global properties); this again results in an increase in the num-

ber of concurrently running monitors, both in the CPS application nodes and in the global monitor nodes.

- *Experiment 2*: I re-run *Baseline* of 384 tasks with the application annotated and instrumented with all local specifications. I run the experiment with an increasing number of injected locally monitored events per second: 400, 800, and 1600, which is representative of an increased sampling rate of the sensors in the CPS application.
- *Experiment 3*: I re-run *Baseline* of 384 tasks with the application annotated and instrumented with all of the local specifications (i.e., ***LP7***, ***LP8***, and ***LP9***). I run the experiment with 40 randomly injected errors.
- *Experiment 4*: I re-run *Baseline* of 3072 tasks with the application annotated and instrumented with all global specifications. I run the experiment with 40 randomly injected errors.

I used the OMF framework [157] to systematically describe and reproducibly orchestrate my experiments on the testbed. In that regard, I translated the above experiments into an OMF experiment description, which is versioned and available on github[13]. I divided the total available nodes into subsets of four nodes, then deployed and ran a separate instance of my experiment on each subset of nodes (in parallel). Each running experiment instance has its own set of parameters as described in my experiment design. For each set of parameters, I ran 10 replicas of the experiment in order to collect sufficient results for a statistically meaningful analysis.

[13]https://github.com/mytestbed/experiments/tree/master/brace2015

Figure. 6.3 shows the result of *Experiment 1* in terms of CPU overhead as compared to Baseline. The figure plots the percentage increase in CPU usage. In the figure, the **L1** bar indicates the overhead when there is a single local property checked; **L3** checks all three local properties, and **G3** checks the three local properties and three global properties. The figure shows that, with increasing numbers of properties to be monitored, the CPU overhead is reasonably under control thanks to efficient event filtering and event automata used to verify these properties.

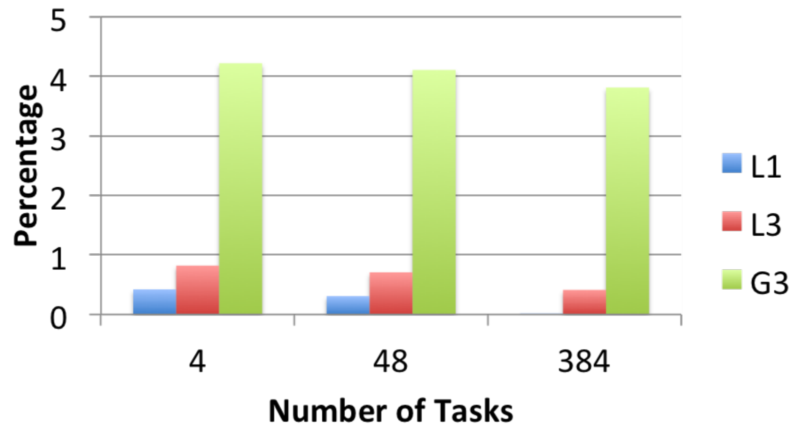

Figure 6.3: Efficiency - CPU Overhead

Figure 6.4 shows the result of *Experiment 1*, plotting the percentage increase in memory usage relative to the Baseline experiment. The memory overhead, as expected, is higher than the CPU overhead. First of all, to guarantee predictable behavior, *Brace*'s local optimizer constantly monitors the behavior of the monitor (which is reflected in the CPU overhead) and adjusts the buffer size to give the hosted application more frequent and longer time windows without monitors. These increasing buffers impact the memory overhead. Secondly, the particular application chosen here exhibits a high degree of concurrent behavior, and thread safety of the

data structure becomes a primary thing to check against in my properties. To do this, I need to allocate memory in *Brace* to store the list of objects to deal with dirty reads and dirty writes, which is crucial to guarantee sufficient effectiveness (in terms of reducing the number of false positives and false negatives).

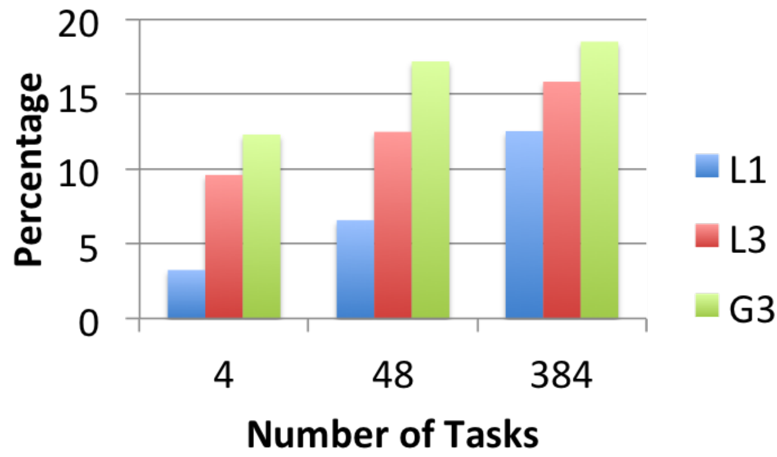

Figure 6.4: Efficiency - Memory Overhead

Figure 6.5 shows the results of *Experiment 2* with different number of injected events; the values shown are relative to the results of *Experiment 1* with 384 tasks. The figure shows with increasing number of injected errors, there is an increasing CPU usage (from 3.3091% with 400 injected events, to 5.4508% with 1600 injected events). The memory usage has similar increase range (from 2.68% with 400 injected events to 5.51% with 1600 injected events). From the log I noticed that, in the scenario with 800 injected events, *Brace* periodically turns on and off the load balancing mode to guarantee the local runtime monitoring behavior. This demonstrates that the local optimization algorithm can guarantee the runtime monitor behavior with unexpected surges of events. However, when there are a large

number of event surges (and the local optimizer cannot produce optimal solutions), load balancing works well, too.

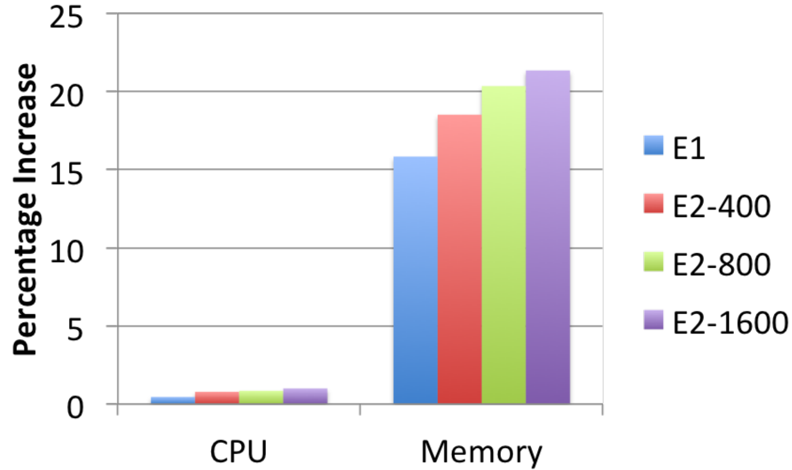

Figure 6.5: Predictability

For brevity, I omit the discussion of the results of *Experiment 3*. The results are consistent with the findings I have in the *Android Test Bed* as all the injected local errors are detected.

Figure 6.6 shows the result of *Experiment 4* with 40 randomly injected logic errors in distributed algorithms. The figure shows that *Brace* can always detect all of the injected logic errors; again *Brace* also detects other (previously unknown) errors. For the first global property ***GP1***, all the additional errors I detected were confirmed by the CPS application developers; they were related to a concurrency error in the scheduling algorithm. For the other two properties, each detected 4 errors, which the CPS application developers later deemed to be false positives. From analyzing server logs from the testbed, I discovered that, though my time synchronization does give a time buffer in which it matches global events with required distributed

attributes, concurrency errors caused data to be overwritten for a few distributed attributes. This is a direct result of the location chosen for the annotations for those distributed attributes in the implementation. If the annotation is in thread critical place, then any variables required are not immune from thread safety issues. I could resolve this by storing more instances of every single property, but the resulting significant increase in the memory usage motivates us to instead sacrifice a small number of false positives.

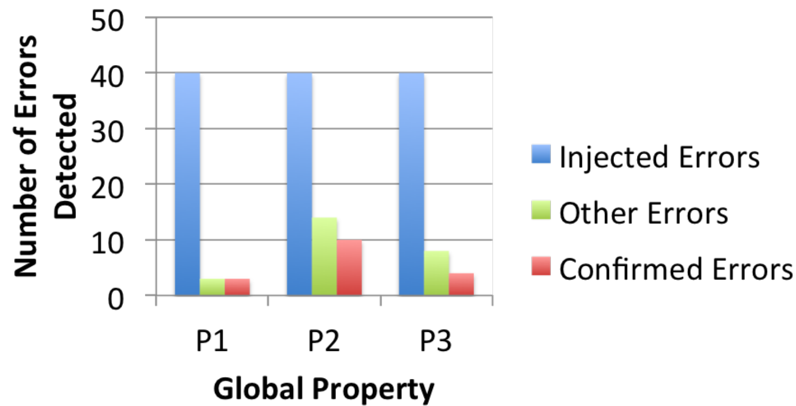

Figure 6.6: Global Properties

6.4.4 Validity discussion

In the Android test bed, I did not measure the CPU and memory usage as I found it is very difficult to do so reliably on the Android platform. I tried third party profile tools, but the overhead of running these tools changed the rover and applications' behavior.

In the Orbit test bed, I did not analyze the network transmission data. I found the evaluation application has a nondeterministic way of sending network packets due to the nature of the complex negotiation and scheduling algorithm and

the message passing system the application chose to use. As a result, I do not know an exact value of how much additional network overhead is required for running the CPS application with *Brace* vs. without it, especially for load balancing and global properties. I did implement data aggregation, event filtering, and package compression in the *Brace* middleware. In the worst case scenario, I can have a dedicated network channel for *Brace* to avoid the impact on the hosted application.

6.5 Related Work

There are a few recent work in RV to improve efficiency and scalability. In [96], shared parameters and events of multiple specifications are explored to increase runtime and memory performance, and RV-Monitor [126] introduces a few improvements for the index tress and caches commonly used for runtime monitoring to improve efficiency. While these techniques provide good solution in efficiency, they suffer from a lack of predictability (especially when there are surges of events) and expressiveness (i.e., none of them provides capability to specify quantitative constraints). These deficits make them unlikely candidates for CPS run-time verification. MapReduce has also been used to process multiple trace fragments in parallel to improve scalability [17]. However, these parallel monitors are offline and are therefore not adequate for CPS applications, which in general require more responsive on-line monitoring. In comparison, my solution is based on on-line monitoring and resolve efficiency issue by introducing Event Monitor at local node (existing work) and Event Automata at global node (introduced in this chapter) to filter and aggregate "raw" events.

Copilot introduces a functional specification language based on Haskell to address hard-constraints in monitoring ultra-critical embedded systems [151]. However, the specification language requires a non-trivial learning curve, which in addition to being more expensive may also introduce errors in specifications. Further, Copilot's expressiveness is not intended for CPS; for instance, the specification language is not expressive enough to capture the local and global properties in the motivating application. In comparison, my solution is based on intuitive specification language (*BraceAssertion*) and I have introduced in this chapter new syntax and semantics in *BraceAssertion* and novel algorithms in *Brace* to support checking global properties.

Adaptive runtime verification (ARV) uses an offline state estimation of the probability of a property being violated to assign a *critical level* to each monitor instance [20]. ARV then uses the critical level of each monitor to turn the monitor on and off at runtime to control overhead. Though this work is effective in reducing runtime overhead, the number of false positives and false negatives that result from approximation errors is not acceptable for mission critical CPS applications. In comparison, besides using Event Monitor (for local properties) and Event Automata (for global properties), I use dynamic linear optimization model and load balancing to guarantee predictable runtime overhead.

In the context of runtime monitors for distributed systems, a distributed on-line monitoring algorithm that allows LTL specifications to be distributed and observed by each component has been shown to 1) reduce the communication overhead (though some communication is still required to resolve non-local properties)

and 2) increase the responsiveness in detecting violations at the expense of performance overhead on the local component [24]. This line of work is similar to ours in that it also explores the use of distributed monitoring. However, this work presumes perfect synchrony among components (e.g., there is no network noise or delay) and that the sets of events on different components are unique. In many CPS applications, perfect synchrony cannot be guaranteed (and is very unlikely) and the sets of events on different components are often overlapping. In my work, I use novel time synchronization algorithms to deal with network noise and delay and have no restriction of sets of events on different components.

In time-triggered on-line monitoring, a monitor is invoked periodically to sample events stored in a buffer. In [34], for instance, a linear programming model is introduced to find the longest sampling period with minimum event history buffer for time-triggered online monitoring. The work in [185], on the other hand, integrates event- and time-triggered online monitoring based on a linear programming model to balance responsiveness and performance overhead. However, in both cases, the linear programming models are based on a Control-Flow Graph (CFG) that is statically analyzed; as a result, these approaches are intended for sequential programs, while most CPS applications are distributed and entail concurrently executing components. In my work, I use dynamic linear model to control each running time for monitors, idle time without monitors, and maximum memory allocated. my approach is more fine grained and suitable for CPS.

Perhaps most similar to my work in terms of capturing global properties is Passive Distributed Assertions (PDA) [163], though in the context of wireless sen-

sor networks. In PDA, distributed assertions are instrumented in the application and evaluated at a single sink node. Properties used for the assertions are collected passively from the network. The distributed assertion has the expressiveness of first-order logic but is not applicable to CPS applications due to its lack of expressiveness of event ordering and real-time properties (e.g., timeouts). In my work, the underlying speciciation language has the power to express metric temporal logic [159] and first-order logic, and my online runtime verification framework is more tailored for CPS applications in terms of efficiency, effectiveness, and being predictable.

6.6 Research Contributions

In this chapter, I made the following contribution:

Research Contribution 5: Built on top of previous works, I create *Brace* as a practical, efficient, and effective online runtime verification middleware specifically designed for CPS applications.

6.7 Chapter Summary

In this chapter, I present *Brace* as a middleware for practical on-line monitoring of Cyber-Physical System Correctness. I first extend *BraceAssertion* to allow programmers to specify global properties, then I enable online violations detection both in local properties and global properties. With a novel linear programming model combined with load balancing, I guarantee predictable behaviors of the runtime monitors. With a thorough case study both on real rover application and a complex simulation on distributed robotic planning, I prove *Brace* as efficient,

effective, and practical runtime verification middleware for Cyber-Physical System.

Chapter 7

Conclusion

With a three-pronged investigation of verification and validation in CPS through a literature survey, an on-line survey, and interviews. We found there are significant research gaps in applying simulation and model checking in CPS verification and validation. Many CPS developers now use trial and error as a de facto standard to debug CPS. To improve the state of the art in CPS verification and validation, I proposed a multi-component tool-set to define and support CPS assertions. My first step is to create BraceForce as a supporting infrastructure to get sensing data from heterogeneous platforms. Then I create *BraceAssertion* as an intuitive yet powerful specification language with capabilities to express metric temporal logic and first order logic. Based on *BraceForce* and *BraceAssertion*, I create *BraceBind* which is not only able to establish a real-time simulation environment for physical models from various simulation platforms, but also allows two ways data input to inform CPS debugging, one from debugging sensor and another from system models. Finally, I complete the picture by creating *Brace* as an efficient and effective online runtime verification middleware which can guarantee predictable behavior for runtime monitors even with unpredictable surge of (random) events. With *BraceForce*, *Brace* can be easily deployed to different test environment to access various debugging sensors. With *BraceAssertion*, CPS developers can write simple yet powerful

specification to capture interesting properties in CPS application. With *BraceBind*, *Brace* is able to detect bugs in a highly accurate real time simulation environment without need for laborious, expensive and often infeasible physical deployment.

We evaluate each component in the Brace suite with specifically designed empirical study, and more specifically using increasingly more complex and suitable real CPS application of smart agent systems to test each component in the *Brace* suite. The evaluation application evolves from a relatively simple simulation application with simulated agents, simulated rovers, and simulated networks, to a testbed with agents running in their own dedicated machine communicating with each other through real network, and accomplish tasks using distributed negotiation algorithms (with simulated rovers), finally to physical rovers executing real tasks in three different real physical environment with a dedicated server assigning tasks.

Appendix

Appendix A

BraceAssertion

A.1 SCL Syntax and Semantics

SCL [160] is defined in the context of timed state sequences and extends LTL (with past operator) with two modal operators $\rhd$ (*prophecy*) and $\lhd$ (*history*). These modalities can be annotated by constraints on clock variables. For instance, the formula $\rhd_{\leqslant 2}\ \phi$ is true if ϕ becomes true within 2. A *formula* of SCL is composed of (atomic) proposition symbols in a finite set AP, boolean connectives $\vee$ and $\neg$, qualitative temporal operators U (Until) and S (Since), $\rhd$ as prophecy operator, $\lhd$ as history operator. SCL is as defined by the following grammar:

$$\begin{aligned}\phi ::= \rho \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \circ\phi \mid \ominus\phi \mid \phi_1 U \phi_2 \mid \phi_1 S \phi_2 \mid \\ \rhd_{\backsim c}\ \phi \mid \lhd_{\backsim c}\ \phi \qquad \text{(A.1)}\end{aligned}$$

where $\rho \in 2^{AP}$, ϕ_1, ϕ_2 are SCL formulas and $\backsim \in \{<, \leqslant, =, \geq, >\}$. The semantics of SCL is inductively defined on timed traces. Given a timed trace $\theta = (\bar{\sigma}, \bar{\tau})$, a formula ϕ in SCL, a *position* $i \in \mathbb{N}$, the *satisfaction relation* $\models$ is inductively defined

by:

$$
\begin{aligned}
&(\theta, i) \models \rho \text{ iff } \rho \subseteq \sigma_i;\\
&(\theta, i) \models \neg\phi \text{ iff not } (\theta, i) \models \phi;\\
&(\theta, i) \models \phi_1 \vee \phi_2 \text{ iff } (\theta, i) \models \phi_1 \text{ or } (\theta, i) \models \phi_2;\\
&(\theta, i) \models \circ\phi \text{ iff } (\theta, i+1) \models \theta;\\
&(\theta, i) \models \ominus\phi \text{ iff } i > 0 \text{ and } (\theta, i-1) \models \phi;\\
&(\theta, i) \models \phi_1 U \phi_2 \text{ iff there exists } \ j \geq i \text{ such that } (\theta, j) \models \phi_2\\
&\quad \text{and for all } k, i \leqslant k < j, (\theta, k) \models \phi_1;\\
&(\theta, i) \models \phi_1 S \phi_2 \text{ iff there exists } j, 0 \leqslant j \leqslant i, \text{ such that}\\
&\quad (\theta, j) \models \phi_2 \text{ and for all } k, j < k \leqslant i, (\theta, k) \models \phi_1;\\
&(\theta, i) \models \rhd_{\backsim c}\, \phi \text{ iff there exists } j > i, \text{ such that } (\theta, j) \models \phi\\
&\quad \text{for all k, } i < k < j, (\theta, k) \not\models \phi \text{ and } \tau_j - \tau_i \backsim c;\\
&(\theta, i) \models \lhd_{\backsim c}\, \phi \text{ iff there exists } 0 \leqslant j < i, \text{ such that}\\
&\quad (\theta, j) \models \phi \text{for all k, } j < k < i, (\theta, k) \not\models \phi \text{ and}\\
&\quad \tau_i - \tau_j \backsim c.
\end{aligned}
\tag{A.2}
$$

A formula ϕ is satisfied by a sequence θ iff $(\theta, 0) \models \phi$. Based on the SCL grammar A.1, and the semantics A.2, it is straightforward to extend SCL with additional boolean and temporal operators. Here we listed a few examples in Rule A.3 and the

full list is in [160].

$$
\begin{aligned}
\text{boolean: } & \top \equiv \neg\phi_1 \vee \phi_1, \bot \equiv \neg\top, \phi_1 \wedge \phi_2 \equiv \neg(\neg\phi_1 \vee \neg\phi_2), \\
& \phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2; \\
\text{future: } & \Diamond\phi_1 \equiv \top U \phi_1 \text{ - eventually;} \\
& \Box\phi_1 \equiv \neg\Diamond\neg\phi_1 \text{ - always;} \\
& \rhd_{[l,u]}\ \phi \equiv \rhd_{\geq l}\ \phi \wedge \rhd_{\leqslant u}\ \phi \text{ - metric intervals;} \\
\text{past: } & \lhd_{[l,u]}\ \phi \equiv \lhd_{\geq l}\ \phi \wedge \lhd_{\leqslant u}\ \phi \text{ - metric intervals.}
\end{aligned}
\tag{A.3}
$$

In [160], SCL is shown to be decidable in *PSPACE* with a simple decision process based on ECA.

A.2 BraceAssertion Syntax (in BNF)

The following BNF gives the syntax of the fragment of *BraceAssertion* to support qualitative constraints. *¡State¿* is any text description of a given state, and *All* refers to any given states and is used to specify global invariants for the system/component.

```
<Given Fragment> ::= Given <condition>
<condition> ::= All | (<State Clause>)
<State Clause> ::= (Not) <State>
<Then fragment> :: = Then|Else <Then statement>
<Then statement> ::= (<TemporalOp>) <State>
<TemporalOp> ::= Always | Eventually | Eventually Permanent |
    Never
```

The following BNF gives the syntax of the fragment of *BraceAssertion* to support quantitative constraints. *Number* refers to any real-value[1] and *Event* is a text description of either an event happened in the system or a predicate.

[1] Unit of time can be seconds, minutes, hours

```
<When Fragment> ::= When( <TransitionCondition> )
<TransitionCondition> ::= <InputSymbol> |
    <InputSymbol>  { <LogicOp> <InputSymbol> }
<InputSymbol> ::= (<temporal condition>) (<Event Clause>)
<Event Clause> ::= (Not) <Event>
<temporal condition> ::=   (<ClockQ>) (<ClockT>)
<ClockT> ::= Before | After
<ClockQ> ::= Between <number> And <number> |
    Within <number> | Less Than <number> |
    More Than <number> | Exactly <number> | Immediately
<LogicOp> ::= And | Or
```

The following BNF gives the syntax of the fragment of *BraceAssertion* to support predicate logics. *Simple Event* is a text description of an event, *Predicate* is a text description of predicate, while *Param* is a text description of a parameter which is used in the evaluation of the predicate.

```
<Event> ::= <Simple Event> | <Predicate Event>
<Predicate Event> ::= <Predicate> With <Params>
<Params> ::= <Param> | <Param> {<Connector> <Param>}
<Connector> ::= And
```

A.3 BraceAssertion Semantics

Table A.1 lists a few keywords in *BraceAssertion* and their semantics in terms of SCL temporal operators, for complete syntax (in Backus-Naur Form) and semantics (in terms of SCL operators), see [197].

Table A.2 gives some examples of SCL representation of *BraceAssertion*. The translation from the full *BraceAssertion* to SCL is defined inductively on *BraceAssertion* and given in Table 4.2 (we assume that the ϕ formulas are atomic in the table).

in Table A.2, the first expression presents typical functional requirement of *bounded time response*; the second and fourth ones are to specify *exact response*

Table A.1: Selected Mappings between *BraceAssertion* Keywords and SCL Temporal Operators

Keyword	**Temporal Operator**
Always	$\Box$
Eventually	$\Diamond$
Eventually Permanent	$\Diamond\Box$
Never	$\neg\Box$
Before	$\rhd$
After	$\lhd$

Table A.2: Comparison of representative temporal expressions

SCL	**BraceAssertion**
$\Box(p \rightarrow \rhd_{\leq 5}\, q)$	*Given* p *When* Within 5 After p *Then* q
$\Box(p \rightarrow \rhd_{=3}\, q)$	*Given* p *When* Exactly 3 After p *Then* q
$\Box(p \rightarrow (\rhd_{>5}\, p) \vee (\circ\Box\neg p))$	*Given* p *When* More Than 5 After p *Then* p *Else* Never p
$\Box((\lhd_{=3}\, q) \rightarrow p)$	*Given* q *When* Exactly 3 After q *Then* p
$\Box((\boxminus_{<3} \neg p) \rightarrow q)$	*Given* Not p *When* Less Than 3 After *Then* Always q
$\Box(p \rightarrow \lhd_{=3}\, q)$	*Given* All *When* Exactly 3 Before p *Then* q

time; the third one is to define *periodicity of events*; the fifth is to specify *alarm*; the last one is to specify *minimum distance between events*.

A.4 Algorithms in BraceAssertion

A.4.1 AspectJ Instrumentation

Algorithm 9 shows the primary pieces of the algorithm the *Event Monitor* uses to instrument the program by injecting AspectJ *pointcuts*. The *given*, *when*, *then*, *else*, and *whenE* parameters refer to those events in the *Given*, *When*, *Then*, *Else*, and complemented *When* for the component indicated by the system-generated id, *cid*.

Algorithm 9 Instrumentation (AspectJ Pointcuts)

```
 1: for ∀e ∈ given ∪ when ∪ then ∪ else ∪ whenE do
 2:     if e.hasClockConstraints then
 3:         cm ← newclockmanager(e)
 4:         register(cm)
 5:     if e.isAggregatedEvent then
 6:         em ← neweventmanager(e)
 7:         register(em)
 8:     if e.isPredicate then
 9:         for ∀p ∈ e.parameters do
10:             if locateParam(p) then
11:                 genPointCut(p)
12:             else
13:                 reportParamMissing
14:         if locateExecution(e.exeuction) then
15:             if locatePredicate(e.predicate) then
16:                 genCombinedPointCut(e)
17:             else
18:                 reportPredMissing
19:         else
20:             reportExeMissing
21:     if e.isEvent then
22:         if locatEvent(e) then
23:             genPointCut(e)
24:         else
25:             reportEventMissing
```

In Algorithm 9, a clock manager is created and registered with the Event Monitor to track the recording and predicting clock values for each event bound with clock constraints (lines 2-4). For each aggregated event, an event manager is created and registered with the Event Monitor; an event manager is essentially a

state machine that records what are the (atomic) events associated with the aggregated event (lines 5-7). For each event bound with a predicate, the algorithm locates the parameters in the implementation and creates AspectJ pointcuts (lines 9-13); afterwards the algorithm locates the annotations for each execution and predicate annotation, and generates AspectJ pointcut to evaluate annotated predicate function with values of parameters collected at runtime via parameters pointcuts (lines 8-20). All cases other than *Predicates* are much simpler as the algorithm generates an Aspect pointcut with either a method invocation, a class constructor, or a single statement annotated with *Event* (lines 21-25).

A.4.2 Monitor Synthesis (Main)

In identifying events in *Given-When-Then* structure, we differentiate between *Proactive* and *Reactive* events. All the clauses defined in *Then* are *Proactive* events where the component makes actions proactively to trigger a state transition. All the clauses defined in *When* are either *Reactive* events where the component receives input signals either from the environment (e.g., via sensors) or other components[2], or simply just temporal constraints for a state transition.

We create the Algorithm 10 to identify events in BraceAssertion specifciation, generate an aggregated event for a set of events in the same structure (e.g., one aggregated event for the whole set of *When* statements, another aggregated event is generated for the whole complemented set of *When* statements if *Else* statements are

[2]BraceAssertion is designed to work on those CPS applications relying on shared variables/message to deal with concurrency and distribution

specified), generate a state for each aggregated event, and synthesize ECA monitor and Event Monitor using the following steps: (1) reads the *BraceAssertion* specification file to construct a set of *BraceAssertion* specifications in line with the BNF syntax (line 1), this step also deals with challenges posed by *And* and *Or* logical operators in *When* fragment, which enables more expressive transitions. The rule is for each event after *And*, a new *When* clause will be created; and for each event after *Or*, a separate specification are created (e.g., a state transition with *When A Or B* are split into a state transition with *When A* and another with *When B*, each of which maintains remaining events in the original *when* clauses); (2) for each *BraceAssertion* specification, assigns a unique id for the component specified, which is used to generate component-unique aggregated events later (line 3); for the set of events in each segment (e.g., *Given*, *When*, *Then*) in the specification, generates an aggregate event for the set, generates a state for the aggregated event, and adds the aggregated state and event along with the ordinal set of events to the event records (ers) (lines 4-5); and uses the collected event records (ers) to synthesis Event Monitor (line 6) and ECA Monitor (line 7).

Algorithm 10 Monitor Synthesis (Main)

1: $specs \leftarrow readInputFile$
2: **for** $\forall s \in specs$ **do**
3: $\quad cid \leftarrow generateComponentId$
4: $\quad$ **for** $\forall es \in s.gi \cup s.when \cup s.then \cup s.whenE \cup s.else$ **do**
5: $\quad\quad ers \leftarrow ers \cup aggregate(es, cid)$
6: $SynthesisEventMon(ers)$
7: $SynthesisECAMon(ers)$

A.4.3 Event Monitor Synthesis

Algorithm 11 is the main algorithm to synthesize Event Monitors using the following steps: (1) for each aggregated event in the *BraceAssertion* substructures (e.g., *When*), creates an event manager (lines 1-2); (2) from a set of events associated with the aggregated event, adds each event as input event to the event manager and adds the aggregated event as the output event to the event manager (lines 3-5); (3) adds the event manager to the global maintained event manager collection (line 6).

Algorithm 11 Event Monitor Synthesis

1: **for** $\forall ae \in ers.gi \cup ers.when \cup ers.then \cup ers.whenE \cup ers.else$ **do**
2: $\quad em \leftarrow new\ eventmanager()$
3: $\quad$ **for** $\forall e \in ae.events$ **do**
4: $\quad\quad em.addInputEvent(e)$
5: $\quad em.addOutEvent(ae)$
6: $\quad updateEMs(em)$

A.4.4 ECA Monitor Synthesis

Algorithm 12 is the main algorithm to synthesize run-time ECA monitors using the following steps: (1) builds a state transition, that consists of a starting state, input events, constraints (if have), and an end state, for *Given*; the starting state is retrieved by finding a state in a globally shared transition table that can transit to the aggregate state for the *Given*, if no such state exists, uses the default start state (that represents any state) (lines 1-6); (2) builds a state transition for *When*, state transition creation logic is similar to *Given*, except the clock constraints are aggregated and added to the state transition (line 8) and the starting state is the aggregate state for *Given* (line 9); another state transition is created for alternative path of *When* (if *Else* is specified) (lines 10-11); (3) builds a state transition for *Then*,

and creates accepting conditions (*accStates*) and rejecting conditions (*rejStates*) for the automata; the temporal operator (e.g. *Eventually*) determines different way of creating the state transition, accepting conditions, and rejection conditions (lines 15-34). We skip the processing of *Else*, which is almost identical to *Then*; (4) with the set of state transitions generated for the current *BraceAssertion* specifciation, updates the global transition table (line 35); (5) creates a runtime monitor which keeps track of the starting state, accepting conditions, and rejecting conditions for the specification (line 36); (6) adds the runtime monitor to the global maintained runtime monitor collection (line 37).

A.5 The Case Study - Full Version

We used an existing robot planning system, which is a distributed version of Generalized Partial Global Planning (GPGP) [116]. This system's planning algorithm is a version of Anytime A* that is customized to distributed planning for a group of mobile vehicles. Specifically, a group of vehicles is assigned patrols that must visit a set of specified waypoints. The vehicles negotiate, and each derives a schedule that contains a subset of the waypoints. The schedules are chosen to optimize the combined utility of the vehicles.

Each vehicle hosts an intelligent agent that can optimize its actions, autonomously and interactively. Each agent includes a local scheduler, which derives a schedule based on a current set of tasks assigned for execution; a negotiator, which coordinates with other agents to derive the schedule; and an execution system. An agent's *task structure* captures its knowledge about how to accomplish

Algorithm 12 ECA Monitor Synthesis

```
1: if ers.given ≠ null then
2:     startState ← trs.find(ers.given.state)
3:     if startState = null then
4:         startState = defaultStartState
5:     trs ← trs ∪ new tr(startState, ers.given)
6:     givenState = ers.given.state
7: if ers.when ≠ null then
8:     clockCons ← new clockCons(ers.when)
9:     trs ← trs ∪ new tr(givenState, ers.when, clockCons)
10:    if ers.whenE ≠ null then
11:        trs ← trs ∪ new tr(givenState, ers.whenE, clockCons)
12:    whenState = ers.when.state
13: if ers.then ≠ null then
14:    tempOp ← getTempOp(ers.then)
15:    if tempOp = null then
16:        trs ← trs ∪ new tr(whenState, ers.then)
17:        trs ← trs ∪ new tr(whenState, ers.thenC)
18:        accStates ← whenState ∪ ers.then.state
19:        rejStates ← whenState ∪ ers.thenc.state
20:    if tempOP = Eventually then
21:        trs ← trs ∪ new tr(anystate, ers.then)
22:        accStates ← whenState ∪ ers.then.state
23:    if tempOp = EventuallyPermanent then
24:        trs ← trs ∪ new tr(anystate, ers.then)
25:        trs ← trs ∪ new tr(anystate, ers.thenC)
26:        rejStates ← whenState ∪ ers.then.state ∪ ers.thenc.state
27:    if tempOp = Always then
28:        trs ← trs ∪ new tr(whenState, ers.then)
29:        trs ← trs ∪ new tr(anystate, ers.thenC)
30:        accStates ← whenState ∪ ers.then.state
31:        rejStates ← whenState ∪ ers.thenc.state
32:    if tempOp = Never then
33:        trs ← trs ∪ new tr(whenState, ers.then)
34:        rejStates ← whenState ∪ ers.then.state
35: updateGT(trs)
36: rms ← rms ∪ new rm(startState, accStates, rejStates)
37: updateRMs(rms)
```

certain tasks. A task structure's root corresponds to a single task. The leaves of the task structure correspond to atomic actions that are performed in various ways to accomplish the task; these combinations are determined by the structure itself. Each agent has a collection of task structures that determines the agent's overall capabilities. An external event corresponding to a request to perform a task triggers an agent's reasoning about whether and how it can accomplish that task. The result

of that reasoning is a schedule that “interweaves” instances of atomic actions from the agent’s tasks in a time- oriented partial order. The schedule can be changed dynamically in response to the changing situation.

To accomplish a global task, agents negotiate over multiple attributes (dimensions). Consider a simple example with two agents. Agent A asks whether agent B can perform task T by time 10, and A requests a minimum quality of 8 for the task. B replies that it can do task T by time 10 but only with a quality of 6; if A can wait until time 15, it can get a quality of 12. A considers which choice is better for the global system. The negotiation considers both completion time and achieved quality, and thus the scope of the search space for negotiation is increased, improving the chance of finding a solution that increases the combined utility; clearly negotiation is more sophisticated when more agents are involved. The cooperative negotiation process can also have many outcomes depending on the agents’ expended effort. They may find a solution that leads to the maximum combined utility, they may simply find a solution that increases the combined utility from their current state, or they may find that there is no solution that increases the combined utility (at least not with the resources available for the search). In this paper, we used an instance in which two vehicles cyclically move along their generated waypoint sets. The utility of visiting a waypoint can change dynamically, which may change the agents’ schedules.

Bibliography

[1] E. Abrahám-Mumm, U. Hannemann, and M. Steffen. Verification of hybrid systems: Formalization and proof rules in PVS. In *Proc. of the 7th IEEE Int'l Conference on Engineering of Complex Computer Systems*, pages 48–57, 2001.

[2] R. Akella and B. M. McMillin. Model-checking BNDC properties in cyber-physical systems. In *Proc. of COMPSAC*, pages 660–663, 2009.

[3] A. T. Al-Hammouri. "A comprehensive co-simulation platform for Cyber-Physical Systems". *Computer Communications*, 36(1):8–19, 2012.

[4] R. Albert, I. Albert, and G. L. Nakarado. Structural vulnerability of the north american power grid. *Physical review E*, 69(2):025103, 2004.

[5] J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. "Making software verification tools really work". In *Automated Technology for Verification and Analysis*, pages 28–42. Springer, 2011.

[6] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. volume 104, pages 2–34, 1993.

[7] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[8] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *JACM*, 1996.

[9] R. Alur, L. Fix, and T. A. Henzinger. A determinizable class of timed automata. In *Computer Aided Verification*, 1994.

[10] R. Alur, R. P. Kurshan, and M. Viswanathan. Membership questions for timed and hybrid automata. In *Proc. of RTSS*, pages 254–263, 1998.

[11] Android sensor framework. `http://developer.android.com/guide/topics/sensors/sensors_overview.html`.

[12] ASM. `http://asm.ow2.org/`.

[13] F. Baccelli, N. Khude, R. Laroia, J. Li, T. Richardson, S. Shakkottai, S. Tavildar, and X. Wu. On the design of device-to-device autonomous discovery. In *Proc. of COMSNETS*, pages 1–9, 2012.

[14] R. Baheti and H. Gill. Cyber-physical systems. *The Impact of Control Technology*, pages 161–166, 2011.

[15] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[16] J. Barnes. *High integrity software: the SPARK approach to safety and security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[17] B. Barre, M. Klein, M. Soucy-Boivin, P. A. Ollivier, and S. Hallé. Mapreduce for parallel trace validation of ltl properties. In *Runtime Verification*, 2013.

[18] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM 2012: Formal Methods*, pages 68–84. 2012.

[19] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation*, 2004.

[20] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. S. D., Z. E., and J. Seyster. Adaptive runtime verification. In *Proc. of RV*, 2012.

[21] D. Basin, F. Klaedtke, and S. Müller. Monitoring security policies with metric first-order temporal logic. In *Proc. of SACMAT*, 2010.

[22] D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proc. of LIPICS*, 2008.

[23] A. Basu, M. Bozga, and J. Sifakis. "Modeling heterogeneous real-time components in BIP". In *Proc. SEFM*, pages 3–12, 2006.

[24] A. Bauer and Y. Falcone. Decentralised ltl monitoring. In *Proc. of FM.* 2012.

[25] A. Bauer, M. Leucker, and C. Schallhart. "Runtime verification for LTL and TLTL". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[26] Introducing behavior-driven development. `http://dannorth.net/introducing-bdd`, 2006.

[27] D. Becker, R. Rabenseifner, and F. Wolf. Implications of non-constant clock drifts for the timestamps of concurrent events. In *Proc. of Cluster Computing*, pages 59–68, 2008.

[28] R. Bellman. “Dynamic programming and Lagrange multipliers”. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767, 1956.

[29] N. Bjoner, Z. Manna, H. Sipma, and T. Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253(1):27–60, February 2001.

[30] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. “Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software”. In *The Essence of Computation*, pages 85–108. Springer, 2002.

[31] E. Bodden. *Verifying finite-state properties of large-scale programs.* PhD thesis, McGill University, 2009.

[32] C. D. Bodemann and F. De Rose. The successful development process with matlab simulink in the framework of ESA’s ATV project. In *Proc. of the* 55th *Int’l. Astronautical Congress*, 2004.

[33] B. Boehm. Verifying and validating software requirements and design specifications. In *IEEE Software*, 1984.

[34] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, 2013.

[35] J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul, and M. Spichkova. "On the correctness of upper layers of automotive systems". *Formal Aspects of Computing*, 20(6):637–662, 2008.

[36] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of MobiSys*, 2003.

[37] J. P. Bowen and M. G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *Proc. of Formal methods for industrial critical systems*, pages 8–16, 2005.

[38] D. Broman, P. Derler, and J. Eidson. "Temporal issues in cyber-physical systems". *Journal of the Indian Institute of Science*, 93(3):389–402, 2013.

[39] W. Brunette, R. Sodt, R. Chaudhri, M. Goel, M. Falcone, J. V. Orden, and G. Borriello. Open data kit sensors: a sensor integration framework for android at the application-level. In *Proc. of MobiSys*, pages 351–364, 2012.

[40] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li. Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior. *ACM SIGBED Review*, 8(2):7–10, 2011.

[41] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. In *Readings in*

hardware/software co-design, 2001.

[42] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Proc. of TACAS*. 2005.

[43] H. X. Chen. Simulink and vc-based hardware-in-the-loop real-time simulation for ev.

[44] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *Trans. on Database Systems*, 1995.

[45] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.

[46] R. Clarisó and J. Cortadella. "The octahedron abstract domain". In *Static Analysis*, pages 312–327. 2004.

[47] E. M. Clarke, B. Krogh, A. Platzer, and R. Rajkumar. Analysis and veri cation challenges for cyber-physical transportation systems. 2008.

[48] E. M. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *Automated Technology for Verification and Analysis*, pages 1–12. 2011.

[49] R. Colgren. Efficient model reduction for the control of large-scale systems. In *Efficient Modeling and Control of Large-Scale Systems*, pages 59–72. 2010.

[50] E. COMPUTING. Cyber-physical systems. 2009.

[51] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the* 4th *ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[52] I. Craggs. Really small message broker. `http://www.alphaworks.ibm.com/tech/rsmb/`.

[53] J. W. Creswell. *Qualitative inquiry and research design: Choosing among the five traditions.* Sage publications, 2012.

[54] J. W. Creswell and A. L. Garrett. "The" movement" of mixed methods research and the role of educators". *South African Journal of Education*, 2008.

[55] D. Crockford. The application/json media type for javascript object notation (json). 2006.

[56] Sony dynamic android sensor hal. `https://github.com/sonyxperiadev/DASH`.

[57] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Hybrid Systems III*, pages 208–219. 1996.

[58] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling cyber–physical systems. *Proc. of the IEEE*, 100(1):13–28, 2012.

[59] A. Deutsch. "Static verification of dynamic properties". *PolySpace White Paper*, 2004.

[60] V. D'silva, D. Kroening, and G. Weissenbacher. "A survey of automated techniques for formal software verification". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[61] P. Edara, A. Limaye, and K. Ramamritham. Asynchronous in-network prediction: Efficient aggregation in sensor networks. *ACM Trans. on Sensor Nets.*, 4(4), 2008.

[62] J. El-khoury, F. Asplund, M. Biehl, F. Loiret, and M. Törngren. "A Roadmap Towards Integrated CPS Development Environments".

[63] H. Elmqvist, S. E. Mattsson, and M. Otter. Modelica-a language for physical system modeling, visualization and interaction. In *Proc. of the IEEE Int'l Symposium on Computer Aided Control System Design*, pages 630–639, 1999.

[64] P. Emanuelsson and U. Nilsson. "A comparative study of industrial static analysis tools". *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[65] C. Fetzer and F. Cristian. An optimal internal clock synchronization algorithm. In *Proc. of COMPASS*, 1995.

[66] Functional mock-up interface. `http://www.fmi-standard.org`.

[67] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Strategies for incorporating formal specifications in software development. *Comm. of the ACM*, 1994.

[68] P. Fritzson and P. Bunus. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pages 365–380. IEEE, 2002.

[69] P. Fritzson and V. Engelson. Modelica - a unified object-oriented language for system modeling and simulation. In *Proc. of ECOOP*. 1998.

[70] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. "CADP 2011: a toolbox for the construction and analysis of distributed processes". *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[71] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI*, pages 1–11, 2003.

[72] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC language: A holistic approach to networked embedded systems". In *Acm Sigplan Notices*, volume 38, pages 1–11, 2003.

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

[74] D. Goswami, R. Schneider, and S. Chakraborty. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *Proc. of ASP-DAC*, 2011.

[75] L. Grunske and P. Zhang. "Monitoring probabilistic properties". In *Proc. ESEC/FSE*, pages 183–192, 2009.

[76] H. Gupta, V. Navda, S. Das, and V. Chowdhary. Efficient gathering of correlated data in sensor networks. *ACM Trans. on Sensor Networks*, 4(1), 2008.

[77] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. “Verification of real-time systems using linear relation analysis”. *Formal Methods in System Design*, 11(2):157–185, 1997.

[78] Z. Han and B. Krogh. Reachability analysis of hybrid control systems using reduced-order models. In *Proc. of American Control Conference*, volume 2, pages 1183–1189, 2004.

[79] M. Harakawa, H. Yamasaki, T. Nagano, S. Abourida, C. Dufour, and J. Bélanger. Real-time simulation of a complete pmsm drive at 10 μs time step. In *Proc. of IPEC*, 2005.

[80] J. Harrison and J. Harrison. *Theorem proving with the real numbers.* 1998.

[81] N. He, P. Rümmer, and D. Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Proc. of DAC*, 2011.

[82] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[83] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *Computer aided verification*, pages 460–463, 1997.

[84] J. Hill and D. Culler. A wireless embedded sensor architecture for system-level optimization. Technical report, UC Berkeley, 2002.

[85] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, 2000.

[86] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. of AOSD*, 2004.

[87] A. Himmler. Openness requirements for next generation hardware-in-the-loop testing systems. In *AIAA Modeling and Simulation Technologies Conference, National Harbor, Maryland*, 2014.

[88] P. Hintjens. *ZeroMQ: Messaging for Many Applications.* 2013.

[89] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. 2006.

[90] G. J. Holzmann. Basic spin manual, 1980.

[91] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 2001.

[92] J. Hsiang and M. Rusinowitch. *On word problems in equational theories.* 1987.

[93] R. Huuck, A. Fehnker, S. Seefried, and J. Brauer. "Goanna: Syntactic software model checking". In *Automated Technology for Verification and Analysis*, pages 216–221. Springer, 2008.

[94] V. Jakkula and D. Cook. Mining sensor data in smart environment for temporal activity prediction. In *Proc. of KDD (Poster)*, 2007.

[95] Z. Jiang, M. Pajic, and R. Mangharam. Cyber–physical modeling of implantable cardiac medical devices. *Proc. of the IEEE*, 100(1):122–137, 2012.

[96] D. Jin, P. O. Meredith, and G. Rosu. Scalable parametric runtime monitoring. 2012.

[97] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. "Why don't software developers use static analysis tools to find bugs?". In *Proc. ICSE*, pages 672–681, 2013.

[98] T. Jones and G. A. Koenig. A clock synchronization strategy for minimizing clock variance at runtime in high-end computing environments. In *Proc. of SBAC-PAD*, 2010.

[99] G. Jourjon, T. Rakotoarivelo, and M. Ott. From learning to researching, ease the shift through testbeds. In *Proceedings of Tridentcom 2010*, pages 496–505, 2010.

[100] jsonbeans. `http://code.google.com/p/jsonbeans/`.

[101] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *Proc. of the* 6th *ACM conference on Embedded network sensor systems*, pages 99–112, 2008.

[102] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. "Comprehensive formal verification of an OS microkernel". *ACM Transactions on Computer Systems (TOCS)*, 32(1):2, 2014.

[103] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. "seL4: Formal verification of an OS kernel". In *Proc. SOSP*, pages 207–220, 2009.

[104] N. Kosmatov, G. Petiot, and J. Signoles. "An optimized memory monitoring for runtime assertion checking of C programs". In *Runtime Verification*, pages 167–182, 2013.

[105] R. Koymans. "Specifying real-time properties with metric temporal logic". *Real-time systems*, 2(4):255–299, 1990.

[106] E. Kuleshov. Using ASM framework to implement common bytecode transformation patterns. In *Proc. of AOSD*, 2007.

[107] H. J. La and S. D. Kim. "A service-based approach to designing cyber physical systems". In *Proc. ICIS*, pages 895–900, 2010.

[108] Labview user manual. `http://autnt.fme.vutbr.cz/lab/FAQ/labview/SimulationModule_UserManual_371013c.pdf`.

[109] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int'l Journal on STTT*, 1(1):134–152, 1997.

[110] G. T. Leavens, A. L. Baker, and C. Ruby. "JML: a Java modeling language". In *Formal Underpinnings of Java Workshop (at OOPSLA)*, 1998.

[111] E. A. Lee. Cyber-physical systems-are computing foundations adequate. In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, volume 2, 2006.

[112] E. A. Lee. "Computing foundations and practice for cyber-physical systems: A preliminary report". *University of California, Berkeley, Tech. Rep. UCB/EECS-2007-72*, 2007.

[113] E. A. Lee. Cyber physical systems: Design challenges. In *Proc. of ISORC*, 2008.

[114] K. R. M. Leino. "Efficient weakest preconditions". *Information Processing Letters*, 93(6):281–288, 2005.

[115] K. R. M. Leino and W. Schulte. "A Verifying Compiler for a Multi-threaded Object-Oriented". *Software System Reliability and Security*, 9:351, 2007.

[116] V. Lesser, K. Decker, T. Wagner, et al. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 2004.

[117] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[118] P. Levis. Experiences from a decade of TinyOS development. In *Proc. of OSDI*, 2012.

[119] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of ASPLOS*, pages 85–95, 2002.

[120] F. Lin, A. Rahmati, and L. Zhong. Dandelion: A framework for transparently programming phone-centered wireless body sensor applications for health. In *Proc. of Wireless Health*, 2010.

[121] J. Lin, S. Sedigh, and A. Miller. Towards integrated simulation of cyber-physical systems: a case study on intelligent water distribution. In *Proc. of DASC*, pages 690–695, 2009.

[122] G. Lipovszki and P. Aradi. Simulating complex systems and processes in LabVIEW. *Journal of mathematical Sciences*, 132(5):629–636, 2006.

[123] C. Liu, K. Wu, and J. Pei. An energy-efficient data collection framework for wireless sensor networks by exploiting spatiotemporal correlation. *IEEE Trans. on Parallel and Distributed Systems*, 18(7):1010–1023, 2007.

[124] D. Locke. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks Technical Library*, 2010.

[125] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. Technical report, 2006.

[126] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Roşu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In *Runtime Verification*, 2014.

[127] J. Magnetto Neff. *Under Construction: Working at the Intersections of Composition Theory, Research, and Practice*, chapter Grounded Theory: A Critical

Research Methodlogy. 1998.

[128] R. Mattolini and P. Nesi. "An interval logic for real-time system specification". *IEEE Transactions on Software Engineering*, 27(3):208–227, 2001.

[129] B. Miller, F. Vahid, and T. Givargis. Application-specific codesign platform generation for digital mockups in cyber-physical systems. In *Proc. of ESLsyn*, 2011.

[130] D. Mills. Network time protocol (version 3) specification, implementation and analysis. 1992.

[131] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. on Comm.*, 1991.

[132] A. Miné. "The octagon abstract domain". *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[133] S. Mitra, T. Wongpiromsarn, and R. M. Murray. Verifying cyber-physical interactions in safety-critical systems. *IEEE Security & Privacy*, 11(4):28–37, 2013.

[134] T. A. Moehlman, V. R. Lesser, and B. L. Buteau. Decentralized negotiation: An approach to the distributed planning problem. *Group decision and Negotiation*, 1992.

[135] W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale. Virtual prototyping of cyber-physical systems. In *Proc. of ASP-DAC*, 2012.

[136] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. “The design of bug fixes”. In *Proc. ICSE*, pages 332–341, 2013.

[137] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Using models to address challenges in specifying requirements for medical cyber-physical systems.

[138] S. Nadimi and B. Bhanu. Physics-based models of color and ir video for sensor fusion. In *Proc. of MFI*, pages 161–166, 2003.

[139] S. Nath. ACE: exploiting correlation for energy-efficient and continuous context sensing. In *Proc. of MobiSys*, pages 29–42, 2012.

[140] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar. Model-based integration platform for fmi co-simulation and heterogeneous simulations of cyber-physical systems. In *Proc. of International Modelica Conference*, 2014.

[141] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* 2002.

[142] U. Nordström, J. D. Lopez, and H. Elmqvist. Automatic fixed-point code generation for modelica using dymola. *The Modelica Association*, 2006.

[143] Open dynamics engine. `http://www.ode.org`.

[144] Open systemc initiative. `http://www.systemc.org`.

[145] J. Ouaknine and J. Worrell. “Some recent results in metric temporal logic”. In *Formal Modeling and Analysis of Timed Systems*, pages 1–13. 2008.

[146] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[147] Paho - mqtt client. `http://eclipse.org/paho/clients/java/`.

[148] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. Safety-critical medical device development using the UPP2SF model. *ACM Transactions on Embedded COmputing Systems*, 2014. (to appear).

[149] D. L. Parnas. Really rethinking'formal methods'. *Computer*, 43(1):28–34, 2010.

[150] L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In *Runtime Verification*, 2012.

[151] L. Pike, N. Wegmann, S. Niller, and A. Goodloe. Copilot: monitoring embedded systems. *Innovations in Systems and Software Engineering*, 2013.

[152] A. Pnueli. "The temporal logic of programs". In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[153] A. Pnueli, A. Zaks, and L. Zuck. "Monitoring interfaces for faults". *Electronic Notes in Theoretical Computer Science*, 144(4):73–89, 2006.

[154] R. Purandare, M. B. Dwyer, and S. Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *Proc. of ISSTA*, pages 280–290, 2013.

[155] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: An open-source robot operating system. In *Proc. of the Open Source Software Workshop of ICRA*, 2009.

[156] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *Proc. of DAC*, pages 731–736, 2010.

[157] T. Rakotoarivelo, G. Jourjon, and M. Ott. Designing and orchestrating reproducible experiments on federated networking testbeds. *Computer Networks*, 63:173–187, Apr. 2014.

[158] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. of the* 3rd *Int'l Conf. on Embedded networked sensor systems*, pages 255–267, 2005.

[159] J. F. Raskin and P. Y. Schobbens. State clock logic: A decidable real-time logic. In *Proc. of HART*, 1997.

[160] J. F. Raskin and P. Y. Schobbens. The logic of event clocks: decidability, complexity and expressiveness. *IFAC*, 1998.

[161] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *Wireless Communication and Networking Conference (WCNC 2005)*, 2005.

[162] U. Raza, A. Camerra, A. Murphy, T. Palpanas, and G. Picco. What does model-driven data acquisition really achieve in wireless sensor networks? In

Proc. of PerCom, pages 85–94, 2012.

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

[164] U. Sammapun, I. Lee, and O. Sokolsky. "RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties". In *Proc. RTCSA*, pages 147–153, 2005.

[165] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. "Taming dr. frankenstein: Contract-based design for cyber-physical systems*". *European journal of control*, 18(3):217–238, 2012.

[166] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. of the* 9th *ACM/IEEE Int'l Conf. on Information Processing in Sensor Networks*, pages 186–196, 2010.

[167] T. Schreiber. Android binder: Android interprocess communication. Master's thesis, Ruhr-Universitat Bochum, 2011.

[168] Scpsolver - an easy to use java linear programming interface. `http://scpsolver.org//`.

[169] A. Shakhimardanov, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar. Analysis of software connectors in robotics. In *Proc. of IROS*, 2011.

[170] J. Shi, J. Wan, H. Yan, and H. Suo. A survey of cyber-physical systems. In *Proc. of WCSP*, pages 1–6. IEEE, 2011.

[171] S. Sierla, B. M. OHalloran, T. Karhela, N. Papakonstantinou, and I. Y. Tumer. Common cause failure analysis of cyber–physical systems situated in constructed environments. *Research in Engineering Design*, pages 1–20.

[172] J. Sifakis. "A framework for component-based construction". In *Proc. SEFM*, pages 293–299, 2005.

[173] N. Simulator. "ns-2", 1989.

[174] A. P. Sistla, M. Žefran, and Y. Feng. "Runtime monitoring of stochastic cyber-physical systems with hybrid state". In *Runtime Verification*, pages 276–293, 2012.

[175] H. Siy and L. Votta. Does the modern code inspection have value? In *Proc. of ICSM*, page 281, 2001.

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

[177] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global views of distributed program execution. In *Proc. of the* 7th *Int'l Conf. on Embedded networked sensor systems*, pages 141–154, 2009.

[178] R. A. Thacker, K. R. Jones, C. J. Myers, and H. Zheng. Automatic abstraction for verification of cyber-physical systems. In *Proc. of ICCPS*, pages 12–21, 2010.

[179] A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In *Hybrid Systems: Computation and Control*, pages 465–478. 2002.

[180] J. Tretmans and E. Brinksma. "Torx: Automated model-based testing". 2003.

[181] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 329–348. 1996.

[182] K. Wan, D. Hughes, K. L. Man, and T. Krilavicius. "Composition challenges and approaches for cyber physical systems". In *Proc. NESEA*, pages 1–7, 2010.

[183] K. Wan, K. Man, and D. Hughes. Specification, analyzing challenges and approaches for cyber-physical systems (cps). *Engineering Letters*, 18(3), 2010.

[184] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *CSUR*, 2009.

[185] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister. Reducing monitoring overhead by integrating event-and time-triggered techniques. In *Runtime Verification*, 2013.

[186] C. Yang and R. Cardell-Oliver. An efficient approach using domain knowledge for evaluating aggregate queries in WSN. In *Proc. of ISSNIP*, 2009.

[187] C. Yang, R. Cardell-Oliver, and C. McDonald. Combining temporal and spatial data suppression for accuracy and efficiency. In *Proc. of ISSNIP*, 2011.

[188] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proc. of the* 5th *Int'l Conf. on Embedded networked sensor systems*, pages 189–203, 2007.

[189] H. Yoon, J. Kim, T. Rakotoarivelo, and M. Ott. Mobility emulator for dtn and manet applications. In *4th ACM Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WiNTECH)*, 2009.

[190] Y. Zhang, I.-L. Yen, F. B. Bastani, A. T. Tai, and S. Chau. "Optimal adaptive system health monitoring and diagnosis for resource constrained cyber-physical systems". In *Proc. ISSRE*, pages 51–60, 2009.

[191] Z. Zhang, J. Porter, E. Eyisi, G. Karsai, X. Koutsoukos, and J. Sztipanovits. Co-simulation framework for design of time-triggered cyber physical systems. In *Proc. of ICCPS*, 2013.

[192] X. Zheng. Physically informed assertions for cyber physical systems development and debugging. In *Proc. of Percom*, 2014.

[193] X. Zheng. Physically informed assertions for cyber physical systems development and debugging. In *Proc. PERCOM Workshops*, pages 181–183, 2014.

[194] X. Zheng, C. Julien, S. Khurshid, and M. Kim. On the state of the art in verification and validation in cyber physical systems. *FSE submitted*, 2014.

[195] X. Zheng, C. Julien, R. Podorozhny, and F. Casses. Braceassertion: Runtime verification of cyber-physical systems. In *Proc. of MASS*, page under review. IEEE, 2015.

[196] X. Zheng, C. Julien, R. Podorozhny, F. Casses, and R. Thierry. Brace: A practical on-line monitoring framework for cps. In *Proc. of Middleware*, page under review. IEEE, 2015.

[197] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez. Braceassertion: Behavior-driven development for cps application. `http://goo.gl/XpTksg`.

[198] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez. Braceassertion:behavior-driven development for cps applications. Technical Report UTARISE-2015-002, 2014.

[199] X. Zheng, D. Perry, and C. Julien. Braceforce: A middleware to enable sensing integration in mobile applications for novice programmers. In *ACM/IEEE First International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, page to appear. IEEE, 2014.

[200] X. Zheng, D. Perry, and C. Julien. Braceforce: Software engineering support for sensing in cps applications. In *ACM/IEEE 5th International Conference on Cyber-Physical Systems (ICCPS)*, page to appear. IEEE, 2014.