BraceAssertion: Behavior Driven Development for CPS Application

Xi Zheng and Christine Julien
The University of Texas at Austin
Email: jameszhengxi@utexas.edu, c.julien@utexas.edu

Rodion Podorozhny
Texas State University, San Marcos
Email: rp31@txstate.edu

Franck Cassez
Macquarie University, Sydney
Email: franck.cassez@mq.edu.au

***Abstract*—Cyber-Physical Systems (CPS) have gained wide popularity, 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 determinitic class of timed automata used for runtime monitoring. To reduce runtime overhead and support properties that reference predicate logic, we use a second monitor automaton to create filtered traces on which to run the analysis using the specification monitor. We 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.**

I. 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 [32]. 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 [23]. 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 [32]. Other verification techniques, including model-based testing [11] and simulation [16] have high learning curves, impractical development costs, and scalability issues. Domain specific tools (e.g., passive distributed assertions [29] and symbolic execution [30]), 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 [32]. 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 [12]) 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 [10]. The use of informal models as a transition has been recommended [15], and Behavior-Driven Development (BDD) tools are gaining popularity among CPS developers [32]. 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 [3] and to support predicate logic. We also design a monitor framework to automate verification based on BDD specifications. Our concrete contributions are:

- We bring Behavior-Driven Development to Cyber-Physical Systems (CPS) to enable formal specification of correct system behaviors using natural language.
- We 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, we implement an efficient dual monitor architecture, consisting of a synthesized *event monitor* that generates filtered traces and a synthesized *timed automata monitor* to verify quantitative properties on traces.
- We provide real-world case studies to evaluate the effectiveness and efficiency associated with the synthesized monitors derived from *BraceAssertions*.

II. Motivation And Overview

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

A. Motivating Application

Our work can be easily motivated by agent-based CPS [24], where the system's concurrency and distribution are handled

This work was supported in part by the NSF under grant CNS-1239498. The authors would like to thank Muhammad Shiraz for work on the implementation of the vehicular application.

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. We 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 [12], a state of the art runtime verification framework, is not able to capture these constraints. Instead, capabilities like those of Metric Temporal Logic (MTL) [20] and Metric Interval Temporal Logic (MITL) [2] 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) [6] 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 our work is based.

B. 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 [8]. 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, we 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 applications. Our 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, we 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. Our monitor synthesis algorithms can automatically generate runtime monitors that are timed automata obtained from the textual description.

C. Our Basic Formal Framework

Because CPS applications require reactions to timeouts or incoming events, our models must consider a dense time domain, for which we use non-negative real numbers. The *implementation* of such a time domain requires digital clocks; we 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 [14], [22].

We 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) [3] has many of the semantic capabilities required for modeling quantitatiev properties of CPS. We next provide a brief introduction to ECA and the associated State Clock Logic (SCL) [28], which is used to express ECA, and relate their capabilities to the goals of *BraceAssertion*.

D. ECA and SCL

Timed automata [1] are finite automata extended with real-time clocks. In timed automata [1], 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) [3] restrict the use of the clock and thus embody a determinizable class of timed automata. In ECA, for each event, a *recording clock*

[1] cukes.info, jbehave.org

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.

Fig. 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. 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).

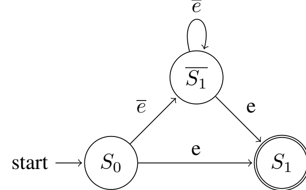

Fig. 1. Specification in ECA: *If a vehicle is selected for a waypoint, its schedule eventually contains that waypoint.*

As a *de-facto* standard, State Clock Logic (SCL) is used to express ECA [28]. 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 our 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. 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 [31] and by CPS developers especially [32] motivates us to create the *BraceAssertion* specification language on top of ECA.

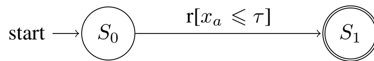

Fig. 2. Specification in ECA: *A vehicle will reach the locale of the waypoint within a bounded time τ.*

III. BraceAssertion

Our *BraceAssertion* language is expressive enough to specify quantitative and qualitative constraints that characterise 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, we introduce the essential syntax of *BraceAssertion* and show how to express standard real-time contraints. We then define the formal semantics in terms of SCL formulas.

A. 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, we 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 scheduler contains the task to reach that way point]." We 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 [28] 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, we 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 contraints 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

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

(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, we (1) add two new constructs and keywords to the *Given-When-Then* structure in the BDD specification, (2) we define additional semantics in the BDD *Then* fragment, and (3) we 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 our first specification) relies on a BDD annotation created specifically for this purpose. More generally, Table I lists our additional BDD annotation classes: *Event*, *Predicate*, *Param*, and *Execution*.

TABLE I. 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 $\forall$, i.e., each instance is verified against the predicate. Alternatively, the *mode* can be assigned *Single* to specify $\exists$, 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. We omit the details for brevity, but we 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 [33].

B. 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 II, 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 [33]. By provinding a translation (Table II) we provide a formal semantics to *BraceAssertion*. Our translation into SCL provides a correct-by-construction algorithm to build monitors to check *BraceAssertion* specifications. Indeed, one the result of [28] 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.

TABLE II. 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	$\Diamond \Box \phi_1$

IV. DUAL MONITOR ARCHITECTURE

One of our main concerns is to establish a tight integration between specification and implementation. At the conceptual level, we accomplish this through the annotations in the BDD-based *BraceAssertion*. From a practical perspective, we leverage *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, we devise a dual monitor architecture shown in Fig. 3. Our 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 analysed 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 our framework more flexible than the state of the art in runtime monitoring [6], [7], 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. Our framework synthesizes two monitors from each specification[3]. We first synthesize an Event Monitor that generates *pointcuts* and *advice* [17], monitors underlying events at runtime, and generates *filtered* execution trace. We then synthesize an ECA monitor to verify system correctness based on collected execution traces.

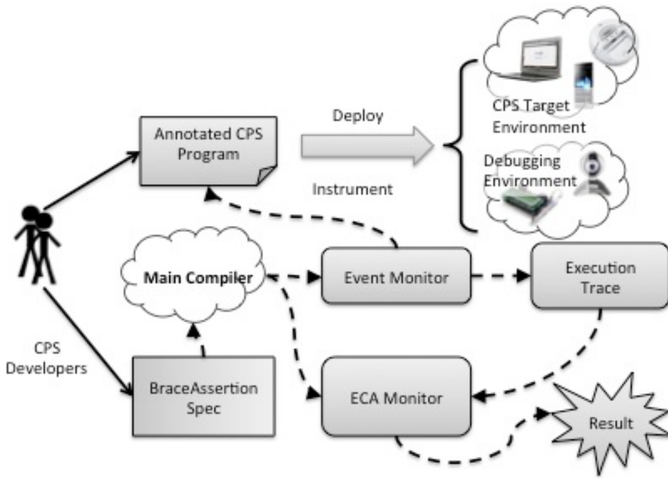

Fig. 3. The Dual Monitor Architecture

A. 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. Our 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. 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) [8]. 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., we 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, our 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, we 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, we omit the main algorithm the *Event Monitor* uses to instrument the program; refer to [33] for details.

[3]We omit the monitor synthesis algorithms, see [33] 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, we 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.

B. ECA Monitor

Given a *BraceAssertion*, we automatically construct an ECA that checks whether the specification is violated. While similar approaches are non-trivial [28], our 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, we 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. Our verification algorithm is based on a standard decision algorithm for testing membership of a trace in a regular language [18]. Testing membership of timed traces for general timed automata is NP-complete [4], but, as we prove below, because we restrict ourselves to a subclass of timed automata that are deterministic, testing membership can be done in polynomial time.

We also introduce two major enhancements. The first one is essential in dealing with the recording and predicting clocks for each *BraceAssertion*. Because we evaluate specifications over trace files, we 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.

Our second enhancement is essential in handling the fact that there may be a large number of *BraceAssertion* specifications for even a modest system, and we need a scalable solution for checking the correctness of these specifications. We use *lazy initialization* to activate an instance of a monitor only when the monitor's initial event is detected, and we terminate the monitor instance as soon as we 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 [33], we provide the complete ECA monitor synthesis algorithm. In essence, since SCL is used to represent ECA [28] 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.

C. Combinatorial Analysis

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

Lemma 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 $O(|e| + |p|)$. Our 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 4.2: 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, we reuse a shared state transition table, which has maximum number of transitions of $O(|e|)$. For each monitor, we also record accepted and rejected states. The total storage cost is therefore $O(n \cdot |e|)$. ■

Lemma 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: Our work is based on the membership test algorithm in [18]. The only difference is instead of checking words, we check timed words. So in the worst case scenario, for each timed word, we must traverse all state transitions ($|e|$) in the global transition table and check all clock constraints ($|c|$). In practice, the behavior of our 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 $O(|t|)$. ■

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

V. Case Study and Evaluation

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

A. The Case Study Briefly

We used an existing robot planning system, which is a distributed version of Generalized Partial Global Planning (GPGP) [21]. 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, we 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.

B. Research Questions (RQs)

Our 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]We get constant time performance using efficient hashing [25].

[5]We refer readers to the complete case study in [33]

C. Experiment Design

To answer these questions, we use our Java prototype and our case study application. We use the patrol application to generate traces as described in Section IV, and we check those traces for the three properties from Section III:

- "*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
}
```

We evaluate *BraceAssertion*'s ability to detect injected violations in traces, and we benchmark its performance. In this process, we also found violations in the patrol implementation other than those we injected. All of the experiments were performed on a PC with an Intel i5 CPU (2.30GHz, 2 cores 4 threads) and 4GB RAM. We 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.

We report results for the following experiments:

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

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

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

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

Experiment 3 (*E3*): we 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. We randomly inject errors to exercise the specification.

Experiment 4 (*E4*): we 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). We 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*): we use the same scaled trace file, using ECA monitors to verify the trace file while incrementally increasing the size of the trace buffer.

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

D. RQ1: The Efficiency of the Event Monitor

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

Fig. 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.

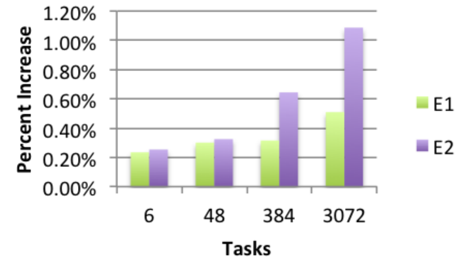

Fig. 4. CPU Overhead

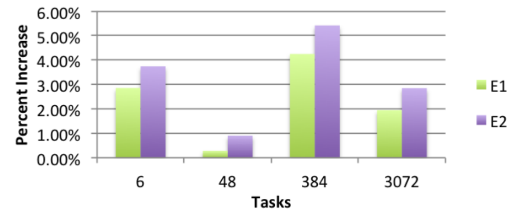

Fig. 5. Memory Overhead of the Event Monitor

Fig. 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 our 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.

We also measured the size of the trace files generated as a measure of the *BraceAssertion* overhead (Fig. 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.

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

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

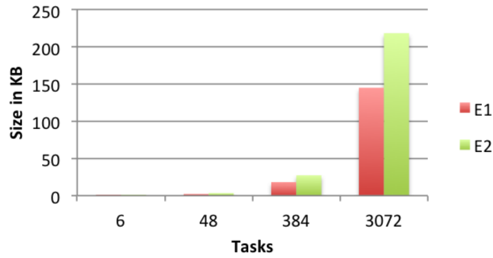

Fig. 6. Size of Traces Generated (in KB)

E. RQ2: The Effectiveness of BraceAssertion

For each trace from *E1*, *E2*, and *E3*, we used the synthesized ECA monitor to verify the trace and compared the number of reported violations with our injected errors. In all cases, our synthesized monitor was able to find the injected violations. In our first few runs of *E2*, our 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, we used an implementation with these bugs resolved for the results.

When we executed *E2* on the corrected version of the application, we successfully detected all of the injected violations, but we also detected one additional violation in the case of 384 tasks and two additional violations in the case of 3072 tasks. We devised and executed *E2-Wild* to adjust the timing constraint to attempt to find more subtle errors in the application. Fig. 7 shows that when we restrict the timing constraint from 40 seconds down to 10 and execute the dual monitors without injecting errors, we 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.

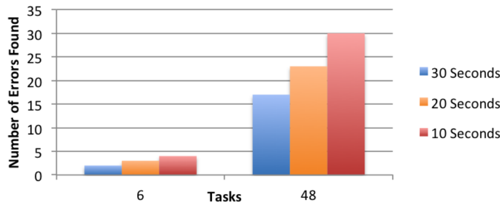

Fig. 7. *E2-Wild*: Finding Real Bugs

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*, we checked violations for the third specification with injected errors for 6 and 64 tasks; our ECA monitor found all injected violations.

F. 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, we used the largest trace from *E2* and increased its size by 10 times as a basis for the experiments here.

We 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. We measured the CPU usage and memory consumption with and without lazy initialization (Fig. 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. We believe the saving is mainly due to number of (specification related) concurrent data structures saved for monitors not activated or deactivated.

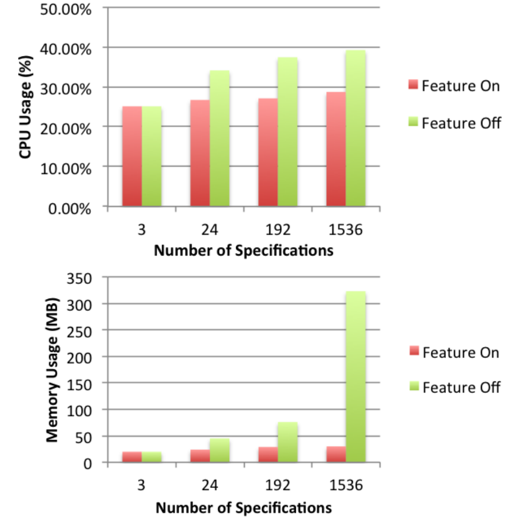

Fig. 8. Lazy Initialization

Finally, we 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., 10 to 30 timed words), the CPU and memory usage are quite high (Fig. 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.

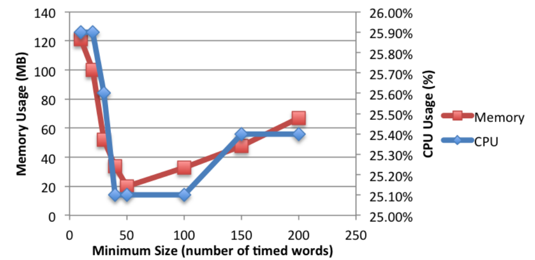

Fig. 9. Trace Buffer- ECA Monitor

G. Threats to Validity

A potential threat to the external validity of our work arises because we evaluated *BraceAssertion* using a single application. We have attempted to mitigate this issue by choosing a real world application that is representative of a broad class of cooperative CPS applications. We also chose correctness properties that exercise diverse aspects (e.g., qualitative, quantative, and first-order expressiveness) of the *BraceAssertion* framework. Though we used a limited number of specifications, our 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, we measured the CPU and memory overhead of our 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. We measured a decrease in runtime when running an increasing number of monitors, which is counter-intuitive. However, we believe that this is reasonable for CPS because the interdependencies among

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

multiple threads and events are not always deterministic and cannot be measured as in traditional software systems.

VI. 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) [20] and Metric Interval Temporal Logic (MITL) [2], which check execution traces for real-time properties. State Clock Logic (SCL) [28] includes *prophecy* and *history* clocks and is decidable by a simple decision procedure that relies on event clock automata. Eagle [5] 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) [13] adds first-order logic expressiveness and metrics to quantify timing constraints [7]. This work might be most similar to our 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 [19]. This is similar to our approach; we aggregate repetitive atomic events to significantly reduce the number of events to be processed, and we use a global transition table backed by an efficient data structure. Our approach is also similar to [27], 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 [9]. Since the underlying BDD for *BraceAssertion* requires manual instrumentation, this approach is orthogonal and complementary to our framework. There are a few existing efficient specification languages that we could use as the basis of our framework [26], however we believe the wide popularity and intuitiveness of BDD makes our work more accessible to real CPS practitioners.

VII. Conclusion

This paper brings Behavior-Driven Development (BDD) into runtime verification of CPS applications. To provide a balance between expressiveness and accessibility, we bridge the gap between BDD and Metric Temporal Logics (e.g., SCL) and first order logic. We support our 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.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

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

[20] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 1990.

[21] V. Lesser, K. Decker, T. Wagner, et al. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 2004.

[22] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. on Comm.*, 1991.

[23] S. Mitra, T. Wongpiromsarn, and R. M. Murray. Verifying cyber-physical interactions in safety-critical systems. *IEEE Security & Privacy*, 2013.

[24] T. A. Moehlman, V. R. Lesser, and B. L. Buteau. Decentralized negotiation: An approach to the distributed planning problem. *Group decision and Negotiation*, 1992.

[25] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[26] L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In *Runtime Verification*, 2012.

[27] 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.

[28] J. F. Raskin and P. Y. Schobbens. The logic of event clocks: decidability, complexity and expressiveness. *IFAC*, 1998.

[29] K. Romer and J. Ma. PDA: Passive distributed assertions for sensor networks. In *Proc. of IPSN*, 2009.

[30] R. Sasnauskas, O. Landsiedel, M. H. Alizai, et al. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. of IPSN*, 2010.

[31] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *CSUR*, 2009.

[32] X. Zheng, C. Julien, S. Khurshid, and M. Kim. On the state of the art in verification and validation in cyber physical systems. Technical Report UTARISE-2014-001, 2014.

[33] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez. Braceassertion: Behavior-driven development for cps application. http://goo.gl/XpTksg.

Appendix

A. SCL Syntax and Semantics

SCL [28] 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:

$$\phi ::= \rho \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \circ\phi \mid \ominus\phi \mid \phi_1 U \phi_2 \mid \phi_1 S \phi_2 \mid \rhd_{\backsim c}\ \phi \mid \lhd_{\backsim c}\ \phi \quad (1)$$

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:

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

A formula ϕ is satisfied by a sequence θ iff $(\theta, 0) \models \phi$. Based on the SCL grammar 1, and the semantics 2, it is straightforward to extend SCL with additional boolean and temporal operators. Here we listed a few examples in Rule 3 and the full list is in [28].

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

In [28], SCL is shown to be decidable in *PSPACE* with a simple decision process based on ECA.

B. 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[9] and *Event* is a text description of either an event happened in the system or a predicate.

```
<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
```

C. BraceAssertion Semantics

Table III 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 [33].

TABLE III. 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 IV 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 II (we assume that the ϕ formulas are atomic in the table).

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

D. Algorithms in BraceAssertion

1) AspectJ Instrumentation: Algorithm 2 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*.

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

TABLE IV. 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*

Algorithm 2 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 2, 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).

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[10], or simply just temporal constraints for a state transition.

We create the Algorithm 3 to identify events in BraceAssertion specfciation, 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 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 3 Monitor Synthesis (Main)

```
1: specs ← readInputFile
2: for ∀s ∈ specs do
3:     cid ← generateComponentId
4:     for ∀es ∈ s.gi ∪ s.when ∪ s.then ∪ s.whenE ∪ s.else do
5:         ers ← ers ∪ aggregate(es, cid)
6: SynthesisEventMon(ers)
7: SynthesisECAMon(ers)
```

3) Event Monitor Synthesis: Algorithm 4 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 4 Event Monitor Synthesis

```
1: for ∀ae ∈ ers.gi ∪ ers.when ∪ ers.then ∪ ers.whenE ∪ ers.else do
2:     em ← new eventmanager()
3:     for ∀e ∈ ae.events do
4:         em.addInputEvent(e)
5:     em.addOutEvent(ae)
6:     updateEMs(em)
```

4) ECA Monitor Synthesis: Algorithm 5 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

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

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* specfciation, 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).

Algorithm 5 ECA Monitor Synthesis

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

E. The Case Study - Full Version

We used an existing robot planning system, which is a distributed version of Generalized Partial Global Planning (GPGP) [21]. 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 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.