Efficient and Scalable Runtime Monitoring for Cyber–Physical System

Xi Zheng, *Member, IEEE*, Christine Julien, *Senior Member, IEEE*, Rodion Podorozhny, Franck Cassez, *Member, IEEE*, and Thierry Rakotoarivelo

***Abstract*—Our reliance on cyber–physical systems (CPSs) is increasingly widespread, but scalable methods for the analysis of such systems remain a significant challenge. Runtime verification of CPSs provides a reasonable middle ground between formal verification and simulation approaches, but it comes with its own challenges. A runtime verification system must run directly on the deployed application. In the CPS domain, it is therefore critical that a runtime verification system exhibits low overhead and good scalability so that the verification does not interfere with the analyzed CPS application. In this paper, we introduce *Brace*, a runtime verification system whose focus is on ensuring these performance qualities for applications in the CPS domain. *Brace* strives to bound the computation overhead for CPS runtime verification while preserving a high level of monitoring accuracy in terms of the number of false positive and false negative reports. *Brace* is particularly suitable to systems in which scheduling is distributed across networked CPS components. We evaluate *Brace* to determine how effectively and efficiently it can detect injected errors in two existing real-life CPS applications with distributed scheduling. Our results demonstrate that *Brace* efficiently detects those errors and a few true bugs and is able to bound both the memory and computation overhead even in systems with large numbers of observed events.**

***Index Terms*—Collaborative work, distributed computing, formal verification, publish-subscribe, real-time systems, runtime.**

I. Introduction

CYBER–PHYSICAL systems (CPSs) entail digital devices interacting with analog ones, the surrounding world, and humans in that world. Many systems we use daily are CPS, including autonomous vehicles, automated health care, smart grids, structural health monitoring, and other mission-critical applications. As a consequence, we expect these systems to be reliable. There have been many efforts to improve and ensure CPS reliability. Formal methods, simulation, and testing have been applied to increase the quality of parts of CPS, like the operating system [27], but it is still a challenge to formally verify complete CPS [60]. This challenge stems from the following: 1) CPS contains analog components that are modeled, for example, via differential equations, making a complete formal model of a CPS a hybrid system, which are known to be hard to formally analyze and 2) assumptions made to enable tractable verification are often too restrictive; defects occur when the system is deployed in a real environment in which the assumptions are no longer satisfied [36].

Because formal verification of an end-to-end CPS is challenging, a second option is to execute the CPS and detect anomalies at runtime. Runtime verification monitors the system execution with respect to some specified properties and issues warnings when a property is violated. Since CPS are often real-time hybrid systems, the properties range from qualitative ones (e.g., safety and liveness) to quantitative ones (e.g., responsiveness). This variety often requires monitoring many events and capturing the time that elapses between events (e.g., using clocks). Runtime verification of CPS first requires the ability to collect events needed to detect anomalies; this is done by instrumenting the application, as is done in our recent work using lightweight distributed runtime monitors [59]. Even so, the number of events is extremely large. Others' previous work has demonstrated that runtime checking of complicated (quantitative) properties with a large number of events imposes a high overhead on system performance [11]. Reining in this overhead is essential, especially in CPS, which are often constrained in terms of memory and CPU resources.

Consider an example CPS, which we use throughout this paper: A multiagent application coordinates a fleet of autonomous vehicles in a global monitoring task. Application correctness requires ensuring that a set of provided waypoints is navigated (a safety property). Another (quantitative) property is determining whether all waypoints can be navigated within τ time (a liveness property). Runtime checking of even these two properties involves many variables. Considering there might be hundreds of properties to check in a complex CPS, overhead can skyrocket, potentially impact the observed application [59]. The state of the art in online monitoring based on formal specifications [2], [3], [34] focuses on very fine-grained management of monitor state updated continuously by the application. However, these approaches neither provide nor guarantee bounds for the impact of memory and computational overhead on the observed application.

Runtime verification of CPS is also made challenging because the systems are large-scale distributed systems or complex systems of systems. Monitoring properties requires collecting and synchronizing events from nodes spread across

Manuscript received April 6, 2016; revised July 11, 2016 and August 24, 2016; accepted September 6, 2016. Date of publication October 21, 2016; date of current version May 2, 2018. This work was supported in part by the National Science Foundation under Grant CNS-1239498.

X. Zheng is with the School of Information Technology, Deakin University, Geelong, VIC 3216, Australia (e-mail: xi.zheng@deakin.edu.au).

C. Julien is with the Center for Advanced Research in Software Engineering, The University of Texas at Austin, TX 78712 USA (e-mail: c.julien@mail.utexas.edu).

R. Podorozhny is with the Department of Computer Science, Texas State University, San Marcos, TX 78666 USA (e-mail: rp31@txstate.edu).

F. Cassez is with the Department of Computing, Macquarie University, Sydney, NSW 2109, Australia (e-mail: franck.cassez@mq.edu.au).

T. Rakotoarivelo is with the Network Research Group, National Information and Communications Technology Australia, Eveleigh, NSW 2015, Australia (e-mail: thierry.rakotoarivelo@nicta.com.au).

Digital Object Identifier 10.1109/JSYST.2016.2614599

the distributed environment. Consider the multiagent CPS example; one property to be checked is whether the total number of messages required for the agents to reach a consensus on task assignment is bounded [44]. Checking this *global property* requires collecting and correlating information about messages and their sizes and numbers from the many distributed nodes (e.g., unmanned vehicles). Stating correctness properties of CPS requires at least the expressiveness of quantitative temporal and first-order logic [13], [59]; efficient and scalable online monitoring of global properties with the required expressiveness remains an open challenge [11].

To make online runtime verification of CPS more efficient and scalable while preserving accuracy, we introduce the *Brace* runtime verification framework, which relies on a novel linear optimization model and load balancing at runtime to guarantee bounded computational and memory resources employed for runtime monitoring of local properties (i.e., those involving just a single computational node in a CPS). *Brace* allows developers to directly specify thresholds for memory and running time allocated to the verification components. *Brace*'s load balancing uses additional dedicated monitor nodes as delegates for runtime checking of the computational nodes in a CPS. Our solution is the first to seek balance among CPU, memory, network bandwidth, and additional computation units to provide bounds for the overhead for runtime checking of CPS applications. To monitor global properties, we create event transformation algorithms that derive essential events from the observed traces so that we can guarantee efficient monitoring of global properties while reducing complex distributed monitoring to a standard decision problem for testing membership of a trace in a regular language [20]. *Brace* builds on existing work in runtime monitoring that solves the problem of *expressing* local and global properties [59]; this paper's fundamental contributions are on the mechanisms that enable efficiently *checking* these properties at runtime. Specifically the following.

1) We combine lightweight event transformation, linear programing, and load balancing to enable efficient online monitoring of CPS. Our approach is applicable to many applications, including those with distributed scheduling.
2) We evaluate *Brace* in two test beds with real CPS applications, demonstrating that it is efficient and scalable.

We introduce our approach in Section II. In Section III, we discuss our empirical study design and present the results in Section IV. We overview the state of the art and practice in Section V and relate it to *Brace*'s contributions. Section VI summarizes the paper and discusses some future research.

II. Brace Approach

In this section, we describe *Brace*, its architecture, and details of its components. We assume that local clocks for the underlying distributed systems are sufficiently synchronized (i.e., that the worst case drift is below a very small and acceptable σ); this is achievable using established algorithms [15], [35]. We assume low-level monitors generate event traces over which *Brace* operates. We also assume facilities for translating between high-level property specifications and implementable property models (i.e., automata); our own work [59] achieves this for expressive CPS properties. *Brace* improves runtime verification of high-level properties related to application logic, assumptions made about a distributed deployment environment, temporal constraints, and scheduling, all of which are otherwise hard or impossible to check at runtime. We presume the low-level systems aspects (e.g., overflow, limit cycle, stability, and minimum phase) have been thoroughly verified (e.g., through simulation), though the impacts of such properties on application logic, deployment environment assumptions, timing, and scheduling, can also be checked in *Brace*. Dealing with unreliable communication is important in CPS; for now we assume a reliable message passing mechanism (MQTT) [22], thereby delegating the responsibility to a lower layer. Recent work on a real-time data distribution in CPS over slow or unstable networks [25] can further mitigate this concern. In the future, *Brace* could integrate more sophisticated handling of these unreliabilities, allowing developers to state properties relating to communication reliability.

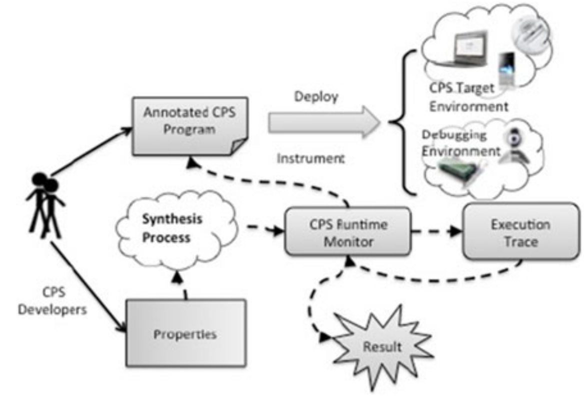


Fig. 1. CPS monitor architecture.

A. Overview

Runtime monitoring of a CPS is shown in Fig. 1. An annotated CPS program is deployed to an instrumented environment. A CPS developer specifies correctness properties that guide the collection of an execution trace (to limit what is collected to what is necessary to check properties of interest). At runtime, the CPS runtime monitor checks the execution trace against the stated properties.

Brace is built using existing CPS runtime verification tools. Existing CPS runtime verification methodologies [59] require developers to write properties using formal specifications (e.g., temporal logics), and the runtime monitors are synthesized from the specifications. Efficient runtime verification tools [34], [59] often involve using the synthesized monitors to instrument the binary code of the monitored application to improve monitoring efficiency. These tools are all *event-triggered*, where events are monitored as they occur. Instead, *Brace* uses *time-triggered* online monitoring [10], [54], where runtime monitors are activated periodically to avoid unnecessary overhead. We use time-driven monitoring instead of event-driven monitoring to better deal with the stochastic nature of CPS. For example, a CPS rover might be deployed to an environment in which it is possible (and often likely) that event generation will trigger events at unexpected times or frequencies and event spikes may occur randomly, making event-triggered monitoring less predictable. In comparison, time-triggered approaches have more predictable behavior, better enabling an approach whose overhead's potential impact on the application can be mitigated. A time-triggered monitoring system that can dynamically allocate the monitoring time slots can handle the inherent stochastic effects with a performance overhead that is more predictable than an event-driven approach. While time-driven monitoring is particularly suitable for multiagent systems (which are the most frequently used architecture for self-adaptive CPS [38]) where there are a lot of consensus messages/events to be made, we acknowledge that when events are extremely sudden and not frequent, event-driven monitoring might be more suitable.

Therefore, instead of determining a fixed activation period for runtime monitors statically, we introduce a linear optimization model that is evaluated at runtime. The linear model can determine a dynamic activation period for instructing the runtime monitor when to start the next monitoring cycle. The linear model requires executing an efficient solver at runtime to avoid additional overhead. While it is quite hard to formally bound the execution time of the solver as the number of variables grows, our approach performs well empirically. We leave the exploration of further optimization for future work; these optimizations may include interior-point optimization algorithms and iterative refinements within a time bound or may convert the problem into one solvable in constant time.

The linear optimization observes the application and runtime monitors (e.g., the application's signal generation speed and the runtime monitors' signal processing speeds), and determines how long a monitor should sleep during each cycle and how long it should remain active to process events to meet the required balance of CPU and memory overheads. When the model cannot produce an optimal solution, for example, due to an unexpectedly large number of events, *Brace* uses internal load balancing to offload local monitoring to shared *monitor nodes* (which are also used to monitor global properties). To minimize the penalty for responsiveness while load balancing, *Brace* relies on low-latency, scalable, and reliable message passing protocols [19], [31]. While this is dependent on the communication channels available in deployment environments, increasingly fast and reliable communications (e.g., 5G mobile networks) make this much less of a constraint on the runtime verification system.

Brace's monitors capture the execution of a CPS 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, we assume the instrumentation also captures timing. 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. The timing sequence respects *monotonicity* $(\tau_i < \tau_{i+1})$ and *progress* $(\exists i \in \mathbb{N} \quad \forall j \in \mathbb{R}, \tau_i > j)$.

Unlike a traditional timed system where each observation can be a simple tuple (eid, τ) (where eid is the unique identifier for "something happened" and τ is a timestamp), observations for a CPS require additional semantic information. The event observed is often associated with one or more keys. For example, consider the property "the total number of messages required to achieve consensus on task assignment is bound by m." Observation of the implicit event "when an agent is assigned a task" contains the taskID, and the observation of each "consensus message" also contains the taskID. This key, i.e., taskID, connects the implicit event with the messages, which is required to check the property. The property monitor may require additional information to resolve the property. Consider the property "after an agent is assigned a task, the atomic actions of the task shall eventually be in the agent's schedule," which must observe the agent's local schedule. To accommodate increased expressiveness in events, a program trace records event observations using a generic event template that can include key fields, numeric fields (that are aggregatable), and non-numeric fields. The event template is in the form $(\text{eid}, \tau, \text{nvk}, \text{nvn}, \text{nvnn})$ where the last three fields each contain a list of zero or more name-value pairs. nvk stores keys associated with the observation, nvn records numeric fields (to be aggregated by *Brace*), and nvnn records non-numeric fields (e.g., a list of tasks).

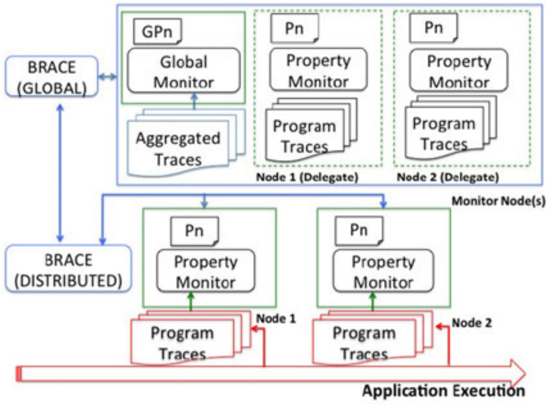


Fig. 2. Brace architecture—overview.

Our prior work [59] extended an existing Agile specification to cover the essential semantics of Event Clock Automata [1], generated timed traces of the system's execution using customized annotations inside the implementation, and synthesized runtime monitors to check property violations against the generated timed traces. We demonstrated that this approach had both low overhead and high accuracy in capturing wild bugs and injected ones. However, the approach is offline (the timed trace is generated as the program executes, but the synthesized monitor is checked against the timed trace offline). In the offline approach, property violations are only detected after the fact, making fault detection much more challenging. In this paper, we use similar fundamentals but check properties online. We check *local properties* at each application node and also check *global properties* that cross multiple application nodes. Most importantly, *Brace* bounds the computational overhead or property checking and delivers fault notifications close in time to the occurrence of the fault.

B. *Brace Architecture*

Fig. 2 shows an overview of *Brace* runtime monitoring. Node 1 and Node 2 are distinct computational nodes in the observed CPS. Each node hosts a *Property Monitor* that parses program traces collected from the application execution; each *Property Monitor* is associated with one local correctness property (Pn); a single application node may host more than one *Property Monitor*. BRACE (DISTRIBUTED) represents the *Brace* executable on each computational node; it determines whether to turn ON and OFF *Property Monitor*s according to the solution of the local optimization model, which is evaluated periodically. This component also aggregates events required for checking global properties and sends these aggregates to global monitors via the underlying message passing framework. Global monitors are hosted on *monitor nodes*, which are designated for monitoring global properties and load-balancing monitoring of local properties (i.e., in situations when the (local) *Property Monitor* is turned OFF to avoid overhead). Each *monitor node* contains (proxy) *Property Monitor*s and a dedicated buffer to hold a program trace for each computational node (*node delegate* in the figure). Each *node delegate* is by de-

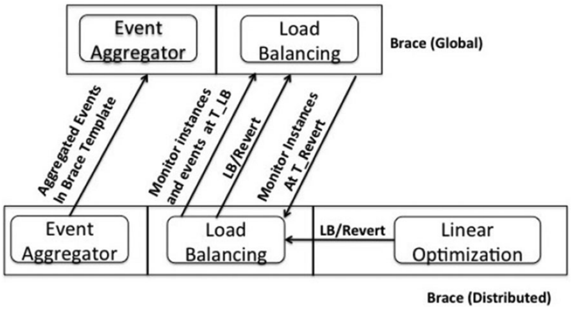


Fig. 3. Brace architecture—components.

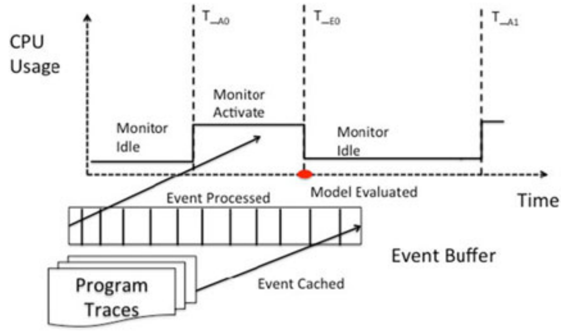


Fig. 4. Brace-timed-triggered monitoring.

fault deactivated and activated only when the corresponding computational node requires load balancing. BRACE (GLOBAL) is the *Brace* executable on each *monitor node*. It listens to load balancing requests from computational nodes and activates the corresponding *node delegate*s. BRACE (GLOBAL) also listens to event aggregation data, further filters and aggregates events, and places aggregated events into a dedicated buffer (Aggregated Traces). These aggregated events are checked by the *global monitors* against formally specified Global Properties (GPn). The *monitor node*s are external to the deployed CPS. The requirement for the computational power of these nodes is determined by the number of application nodes and the number of local properties and global properties to be monitored. In our current implementation, we retrieve the requirement empirically from a given evaluation application. We leave the automation of requirement assessment as future work.

Fig. 3 shows the internal components of BRACE (DISTRIBUTED) and BRACE (GLOBAL) and the interactions between the two. BRACE (DISTRIBUTED) has three components with distinct roles. The Linear Optimization observes each *Property Monitor* and the associated program traces periodically evaluates its linear programing model, and uses the solution to determine when to activate (and deactivate) *Property Monitor*s. The Linear Optimization also determines how to allocate the memory buffer required to hold unprocessed events while the *Property Monitor* is OFF. Whenever the linear programing model cannot generate an optimal solution, it turns OFF the associated *Property Monitor* and generates an *LB* (i.e., Load Balancing) signal, which is sent to the Load Balancing component. The Linear Optimization component continues to keep a track of the *Property Monitor* and program traces and continues to evaluate the linear programing model. If it finds an optimal solution, a specified number of times in a row, it generates and sends a *Revert* signal (i.e., Revert to monitoring properties locally) to Load Balancing. After Load Balancing returns the signal "OK" (indicating that the latest monitor instances from *monitor node*s are restored), Linear Optimization turns ON the local *Property Monitor*. Load Balancing forwards any *LB* signal to BRACE (GLOBAL), takes a snapshot of *Property Monitor*, and publishes the snapshot along with events from the program traces to BRACE (GLOBAL). Afterward, Load Balancing periodically publishes a snapshot of new events from the program traces along with the latest timestamp (*T_LB*). Upon receiving a *Revert* signal, Load Balancing sends the signal to BRACE (GLOBAL) and, then, waits for the corresponding *node delegate* to return the latest instance of *Property Monitor*. Finally, Event Aggregator is used to locate, aggregate, and publish to BRACE (GLOBAL) aggregates of events using the *Brace* format from the original program traces.

BRACE (GLOBAL) has two components. The Event Aggregator receives published aggregate events from BRACE (DISTRIBUTED), synchronizes events across the distributed computational nodes, further filters and aggregates the events, and generates the final aggregated signals into *Aggregated Traces* for the *Global Monitors*. The Load Balancing component receives snapshots of *Property Monitor* instances and events from computational nodes that delegate monitoring tasks to the *monitor node*. Load Balancing then activates the corresponding *node delegate* and sends received events to the queue holding program traces. Upon receiving a *Revert* signal from a BRACE (DISTRIBUTED) instance, Load Balancing takes a snapshot of the *node delegate*'s *Property Monitor* and sends this snapshot back to BRACE (DISTRIBUTED). In this paper, we focus primarily on Linear Optimization (see Section II-C) and Event Aggregator (see Section II-D).

C. CPS Specific Linear Programing Model

Brace uses time-triggered online monitoring [10], [39], [54] to reduce computational and memory overhead. However, instead of statically analyzing a linear programing model to determine the monitor's sampling and sleeping cycle, *Brace* allows CPS developers a finer granularity of control (i.e., to specify lower bounds on the sampling and sleeping states of the monitor and an upper bound for the active time (AT) of the monitor). Fig. 4 shows how *Brace* uses the linear programing model for time-triggered online monitoring, targeted specifically for CPS, where the linear optimization is evaluated dynamically, depending on the evolution of the system.

Each *Property Monitor* is deactivated during a computed *Monitor Idle* time (IT). To prevent losing events, *Brace*'s event buffer caches events from the underlying program traces. When the property monitor is activated, it processes events collected during the IT. The state of the art is to determine the first time for activation (τ_{A0}) statically and subsequently activate and deactivate the monitor on regular intervals, i.e., the next activation time τ_{A1} is statically determined to be $2 * \tau_{A0}$. The dynamic nature of CPS quickly invalidates any statically determined monitor activation schedule. CPS are often characterized by surges of events that such a static schedule cannot handle. Instead, to create dynamic activation schedules, we measure characteristics of the CPS and use them as inputs to the linear programing model. First, we measure event creation speed (*CS*, in events/second) simply by observing the underlying program traces. This gives us a good indication how much computational and memory overhead will be incurred in the *Property Monitor*. Second, we measure the event processing speed (*PS*, also in events/second) by observing the *Property Monitor* itself. This measurement is a good indicator of whether the *Property Monitor* is overloaded in spite of a reasonable *CS*. Our linear programing model relies on these two inputs; *Brace* evaluates the linear programing model each time the *Property Monitor* is deactivated to determine the next activation time (i.e., when to wake up the sleeping monitor). Since part of the runtime monitor's job is to check the program trace, retrieving *CS* has trivial overhead [59]. The overall overhead of using *Brace* is proven to be very well bounded as shown in Section IV.

In addition to enabling dynamic activation, CPS developers also want to maximize the monitor's IT and minimize its AT, reducing CPU overhead and the potential impact of the monitor on the CPS application, of course without impacting its ability to detect violations. *Brace* allows CPS developers to specify a minimum value for *monitor IT* ($\mathrm{IT}_{\mathrm{qos}}$) and a maximum value for *monitor activation time* ($\mathrm{AT}_{\mathrm{qos}}$) of each *Property Monitor*. To limit the memory required to store the idle period's events in the event buffer, the linear programing model searches for an optimal solution to minimize the memory usage while finding a satisfying IT and AT. *Brace*'s linear programing model's objective function is, therefore,

$$\mathrm{minimize}(\mathrm{CS} * \mathrm{IT} + (\mathrm{CS} - \mathrm{PS}) * \mathrm{AT}). \quad (1)$$

This model is also framed by additional constraints imposed by practicalities of the CPS environment. First, and simply: $\mathrm{IT}_{\mathrm{qos}} \leq \mathrm{IT}$ and $\mathrm{AT}_{\mathrm{qos}} \geq \mathrm{AT}$ ensures that the application's observed IT, i.e., the time when no monitoring is occurring, meets the required minimum IT and maximum AT provided by the CPS developer. To bound memory overhead further, we add the constraint: $\mathrm{CS} * \mathrm{IT} \leq maxMemory$ to restrict how much memory can be allocated to store unprocessed events during each segment of IT (i.e., when the program is generating events but the *Property Monitor* is not processing them because it is deactivated). The value of maxMemory is provided by the CPS developer. CS captures the average event generation speed in the most recent sampling period, which *Brace* uses as a prediction for the generation rate in the next sampling period.[1]

The optimal solution, if found, balances CPU overhead with responsiveness and memory overhead. If the optimal solution is not found, the node resorts to load balancing, which seeks a balance between local computational overhead and network overhead. *Brace*'s load balancing uses publish/subscribe [31] to coordinate computational nodes running *Brace* software components. BRACE (DISTRIBUTED) uses a *bundle* to combine events and other data to save network overhead. Each bundle contains a message header with the unique id of the application node, a timestamp of the bundle's creation, and a unique sequence number. Because we only want one monitor node to be responsible for each computational node, we implement a cancel protocol on the publish/subscribe mechanism using a second topic shared among monitor nodes. When a monitor node receives a bundle and intends to process it, it publishes the bundle's node id and sequence number, along with the timestamp at which it was received. If there are duplicates, the monitor node that received the bundle first according to the reception timestamp keeps and processes the bundle while others discard it. When a monitor node keeps and processes a bundle, it takes responsibility for the load balanced *Property Monitor*, instantiating the relevant *node delegate* and subsequently receiving and processing events destined for the delegated *Property Monitor*. The *node delegate* continues to do this until it receives a signal from the *Property Monitor*'s computational node indicating that the node is ready to retake responsibility for its own *Property Monitor*.

D. Event Aggregation and Transformation

Aggregating events to evaluate global properties requires coordinated actions among distributed (local) computational nodes and (global) monitor node(s). Observations in program traces from each local computational node adhere to the same *Brace* event template (see Section II-A). We treat all event observations with the *Brace* event template as "instructions" for the distributed nodes as to which events to collect and send to global property monitors. Reusing the *Brace* event template for this purpose requires an additional field to differentiate whether the event is for a local property, a global property, or both. The *Event Aggregator* inside each BRACE (DISTRIBUTED) instance aggregates events that have the same *eid* (i.e., those to be sent to the same global property monitor) to reduce network overhead and ultimately simplify the checking of global properties that depend upon these events. The aggregation algorithm groups observations with the same eid using the following rules.

1) If any observations contain nvk (i.e., an observation is associated with one or more keys), we add a count field to contain the total number for each unique key.
2) If any observations contain nvn (i.e., an observation has one or more variables with numeric fields), we add $\max$, $\min$, and sum fields for each unique numeric field across the aggregated observations.
3) If any observations contain nvnn (i.e., an observation is associated with one or more variables with non-numeric fields), we simply aggregate the values into a linked list for each unique non-numeric field.
4) Instead of recording the timestamp (τ) of each observation, we record $\tau_{\min}$ and $\tau_{\max}$ across all of the observations to capture a first and last timestamp for the aggregate.

Aggregated events are combined into bundles, which are sent to BRACE (GLOBAL) at an application-specific interval.

As bundles containing aggregated events are received at the (global) monitor nodes, they are synchronized and transformed to generate events in the aggregated traces used for checking global properties. This process has two levels of synchronization. The first is to synchronize events received from a single computational node. *Brace* simply checks the unique sequence number inside each bundle to guarantee a total ordering of messages from each application node and to detect missing bundles. The second level is to synchronize events across distributed nodes. *Brace* uses the timestamps of the events and known maximum network delay to detect out-of-order delivery across distributed nodes. Since the timestamp of each aggregate event is a min and a max, we use the max. To check quantitative properties that require finer granularity, we could introduce a field in the *Brace* event template that keeps an event's individual timestamp. The synchronization algorithms are similar to existing literature [47].

[1] Other approaches for estimating a future event generation rate exist, for example, using a running average over a window of sampling periods or using other context information to identify similar situations; we use this simple approach in our prototype and leave the exploration of other heuristics to future work.

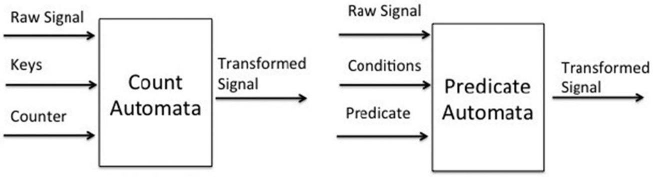


Fig. 5. Event transformation automata.

After events are fully synchronized and stored in an internal lock-free queue, *Brace* uses two automata (shown in Fig. 5) to transform the observed events into those checked by the global monitor. The Count Automata transforms multiple aggregatable events with the same event identifier (eid) and matching keys into a single event with all the information. It accepts three inputs: a raw signal containing the aggregated event from the internal queue, a list of keys that should be matched across signals, and a counter indicating the number of raw signals required to generate the transformed signal, which is the output. As an example, consider the property: "after all tasks are assigned, all tasks shall be completed within τ time units;" the input is the event id for "an agent is assigned a task;" there is one key ("taskID"); and the counter is the total number of tasks that should be assigned (which is an application-level correctness constraint).[2] The generated signal represents the event "all tasks are assigned," which is the event of interest to the *Property Monitor*.

Checking CPS properties requires first-order logic, where a function implementing the predicate is invoked; the Predicate Automata serves this purpose in *Brace*. It accepts three inputs: a raw signal containing the aggregated event, a Boolean expression (conditions) providing a required relationship among events in the aggregated trace, and the predicate. For the property "the number of messages required to reach the consensus of which agent shall be assigned a given task is bounded by m," the signals are events representing each consensus message; there is a single condition, which represents "an agent is assigned a task;" and the predicate accepts a list of aggregated consensus messages with each event object containing the total number of consensus messages from a specific vehicle regarding the task(s) that it is assigned. *Brace* invokes the function with the required input and generates the output signal if the function evaluates to TRUE. The aggregation and filtering of the raw signal is bounded by the time span of those events in the conditions. The aggregation relies on the key-value pairs collected from the events (i.e., group by the unique node id, event id, and key value).

The aggregation algorithms are largely dependent on the decision procedure they preserve, and in turn, of the properties that can be verified on the aggregated data. We presume it is intuitive for the engineer using *Brace* to understand that the aggregation process might lose information and prevent the verification of certain classes of properties. Intuitively, we introduce event aggregation and event transformation to reduce the number of messages transmitted (thus saving network bandwidth) and the number of states/events to be checked (thus lowering performance overhead and increasing scalability). We will show this first analytically and, then, with a case study on two test beds with real-life CPS applications.

E. Combinatorial Analysis

Lemma II.1: The runtime complexity for the *Event Aggregator* algorithm at each (local) computational node is $O(|e|)$, and the space complexity is also $O(|e|)$, where $|e|$ is the number of observed events in the program trace.

Proof: The *Event Aggregator* algorithm iterates over each event to create an aggregated event. The complexity of the aggregation process is determined by the number of keys, number of numeric fields, and number of non-numeric fields, which are constant. So the runtime complexity is $O(|e|)$. The extra space required for the aggregated event is an additional "count" field for each key field, and three additional fields ("min," "max," "sum") for each numeric field, which are also constant. So the space complexity is $O(|e|)$. ■

Lemma II.2: The runtime complexity for the *Event Aggregator* algorithm at the global *monitor node* is $O(|e|^2)$, and the space complexity is $O(|e|)$, where $|e|$ is the number of observed (raw) events.

Proof: The runtime complexities for the two levels of event synchronization are both $O(|e|^2)$ (we assume the implementation uses an ordered link list). The runtime complexity of storing events into an internal queue (another ordered link list) is also $O(|e|^2)$. Each Count Automata requires one iteration to loop through the internal queue created above, the runtime complexity is $O(|e|)$ for each of the automata, and there is a fixed constant number of automata. The Predicate Automata requires one iteration to loop through the internal queue of raw events and another internal loop over the conditions for the aggregated program trace (another ordered link list); the runtime complexity is therefore $O(|e|^2)$. Thus, the runtime complexity overall for the event aggregator is $O(|e|^2)$. The space complexity is two internal queues for event synchronization, one internal queue for raw signals, and the queue for the program trace, which is, in total, $O(|e|)$. ■.

III. Empirical Design

Brace's main design goals are to improve efficiency and scalability of runtime monitoring of CPS properties while maintaining effectiveness. To these ends, we measure the CPU and memory overheads for monitoring local and global properties of CPS, including for a large number of monitored properties. We also measured the false positives and false negatives in monitoring both local and global properties.

Our prior work [59] introduced runtime monitors that check formally specified properties with the expressiveness of a quantitative temporal logic [42]. This monitoring was performed offline and only checked local properties. In this paper, we generate global property monitors using the same monitor synthesis process and also handle complexities of distributed monitoring (e.g., event synchronization and generating necessary event signals) within the *Brace* framework. *Brace* essentially reduces the distributed monitoring problem into a local one as long as we provide the required input and output for Count Automata and Predicate Automata.

We implemented our time-triggered online monitoring with our underlying linear optimization model, which uses SCPSolver [46]. We implement publish/subscribe communication using a combination of *Really Small Message Broker* [12]

[2] In our implementation, we automatically synthesize these automata based on natural language specifications from developers [59].

and the Paho MQTT client [40]. All the queues required for the program trace, event buffer, and aggregated program traces are implemented using lock-free lists [49].

We apply our implementation to two existing multiagent CPS applications based on the Soft Real Time Agent Control Architecture [21] and GPGP coordination architecture [30], whose software architectures are representative of CPS systems including those with distributed scheduling. In both cases, we acquired the application from other researchers in the CPS domain in an attempt to mitigate biases introduced in using applications that we build just for evaluating *Brace*.

The first evaluation application is a Rover Patrol application that was built in Android and deployed to a Samsung Galaxy S3 phone to control a Rover 5 Robot Platform.[3] In this application, each robot in a team is assigned a set of tasks by a global scheduler; each task is effectively a set of waypoints that the robot must visit. We used the language defined in [59] to state six (local) correctness properties for this application, then we used the *Brace* implementation described in this paper to monitor those properties at runtime. The properties are as follows.

LP1 After a vehicle is assigned a task, the atomic actions of the task shall eventually be in the vehicle's schedule.

LP2 The completion time for a given task is bounded.

LP3 The assignment of a task to a vehicle reflects the optimal choice.

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

LP5 The duration of the main control loop is bounded.

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

We assess *Brace*'s computational and memory overhead and its effectiveness at checking these local properties which are otherwise hard to detect. We intentionally selected the above properties as ones that are associated with the distributed algorithm, timing requirements, and the control algorithm. We deployed and executed this application in a laboratory environment on real rovers. We used three surfaces to mimic different physical deployment environments (wood, grass, and linoleum[4]). The test bed also contains an overhead camera that is used for both positioning and for recording the tests.

The second application models a utility-based distributed scheduling multiagent system. In this instantiation of the application, the rovers performing the global patrol task share a blackboard that is updated concurrently by scheduling and execution components of agents on those nodes. The complexity of this application lies in its concurrency and correctness of the scheduling algorithms, which are very difficult to check. In this application, we state and check 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.

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

We also check 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.

GP3 The global patrol must eventually finish (i.e., all waypoints must be visited).

Through this application, we aim to assess *Brace*'s effectiveness in checking *global* CPS properties and its efficiency in *online* monitoring, including in the face of surges of events.

In the first application, we injected ten logical errors that cause deliberate violations of each of the six properties. We profiled the CPU and memory utilization using TraceView [48] and MAT [33], respectively, and recorded the number of false positives and false negatives in property detection by analyzing the property monitor trace files and comparing them to the overhead camera recording of the rovers' behavior.

We deployed and ran the second application on a replica of the ORBIT testbed [43] with 40 physical machines (i.e., *nodes*) based on a VIA 1 Ghz, 512 MB board [24], [56], interconnected via multiple networking technologies, including two wired 1 GB Ethernet networks, which separate control traffic and experiment/data traffic. The nodes' clocks are synchronized via NTP, which ensures offsets smaller than 5 ms. We used OMF [41] to systematically describe and orchestrate our Orbit experiments. We divided the nodes into ten subsets of four nodes, then ran our experiment on each subset (in parallel). Each running instance has its own parameters. We evaluated with an increasing number of tasks: 4 (default), 48, and 384. We use this as a baseline before rerunning the annotated application, instrumented with varying numbers of local and global properties to measure the impact on CPU and memory overhead (computed from Orbit's performance log data). We then hold the number of tasks constant but increase the number of injected local events (i.e., we introduce artificial events observable by the property monitors to ascertain the scalability of monitors in the face of CPS event surges). We then execute the application with injected logical errors to determine whether *Brace* can detect the errors, recording the number of false positives and negatives by analyzing property monitor trace files and comparing with Orbit's application log files. We ran each experiment multiple times to enable a statistically meaningful analysis.

For *Brace* settings in both evaluations, we used an AT (maximum activation time) of 4 s, an IT of 10 s (minimum IT), and maxMemory of 2500 events. These settings were selected as representative of normal behavior of our applications on the computers in the given testbed.

IV. Results and Discussion

We next evaluate the efficiency, scalability, and effectiveness of *Brace*'s. We also briefly discuss some threats to validity.

Efficiency of Brace *Runtime Monitoring:* To measure efficiency of using *Brace* for runtime monitoring of CPS applications, we measured its CPU and memory usage. Fig. 6 shows results for running our second application in the Orbit testbed in terms of CPU overhead (percentage increase relative to running the application without any instrumentation) for increasing numbers of tasks. With increasing numbers of properties monitored (which increases the number of distinct executing *Property Monitors*) and increasing number of tasks monitored (which increases the number of events monitored), the CPU overhead is controlled: the average overhead for monitoring one local property ranges from a 0.77% to 1.7%; and the average overhead for monitoring three local properties ranges from 0.40% to 2.98%. This finding is consistent with our results for the first application executing in the Android Test Bed, which we omit for

[3] [Online]. Available: https://www.sparkfun.com/products/10336

[4] Sample videos for the three surfaces. [Online]. Available: https://goo.gl/PkxDa4, https://goo.gl/Vv3fF2, and https://goo.gl/s097Ag.

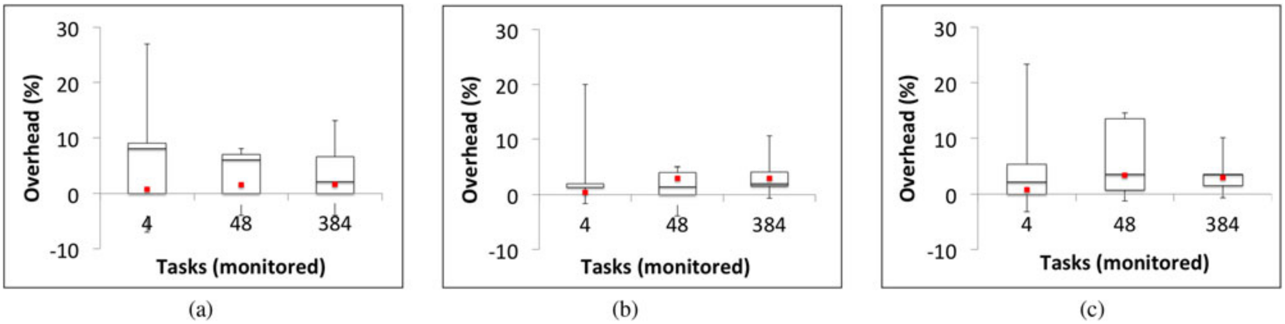


Fig. 6. CPU overhead—orbit test bed. (a) One local property. (b) Three local properties. (c) Three local, three global properties.

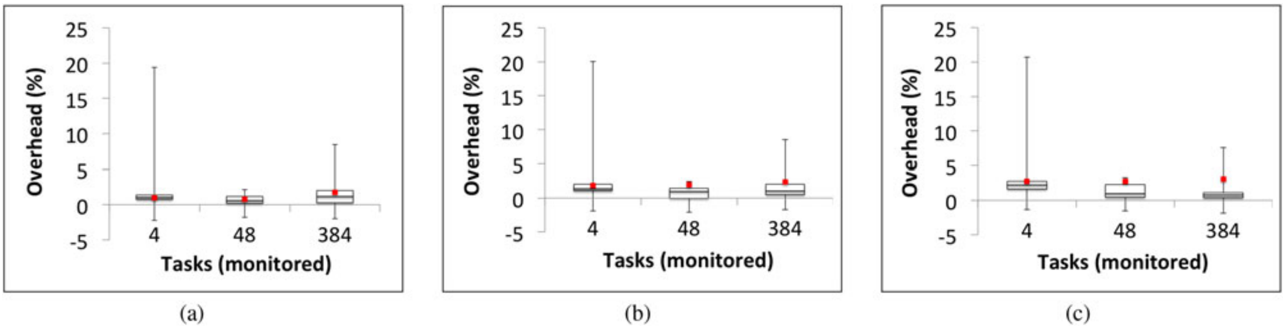


Fig. 7. Memory overhead—orbit test bed. (a) One local property. (b) Three local properties. (c) Three local, three global properties.

brevity. These relatively low overheads are largely attributed to *Brace*'s time-triggered online monitoring. Another interesting finding is that the additional CPU overhead for monitoring three global properties in addition to the three local properties is only (on average) 0.3%, which is consistent with our combinatorial analysis.

Fig. 7 shows the results for memory overhead, again in comparison to a baseline with no instrumentation. One interesting finding is when there is a relatively small number of events (i.e., tasks) to monitor, the memory overhead is larger than the CPU overhead. We believe the cost of building a lock-free queue to hold unprocessed events (as required for the time-triggered online monitoring) has a minimum cost that is amortized as the number of tasks increases.

In the Android Test Bed, we compare results of *Brace* with our previous work [59] (which does not employ time-triggered online monitoring or load balancing). In our previous approach, our instrumented event monitoring significantly changed the behavior of rovers (e.g., navigating from one point to another took a lot longer and sometimes the rovers did not move at all). The elements of *Brace* introduced in this paper make it demonstrably better suited for resource constrained CPS applications where the computational overhead of runtime monitoring has to be maintained to be very low.

In the application in the Orbit testbed, we also notice a tendency for some of the execution times to vary substantially between runs. After some investigation of the issue, we found one of the wild bugs detected by *Brace* is related to a concurrency issue in the application. This bug causes significant fluctuations in CPU and Memory usage in some cases. Though the number of nodes used in the Orbit test bed (40 nodes) and the number of trials used (ten times) allow us to get fairly accurate mean value, the variance captures these fluctuations.

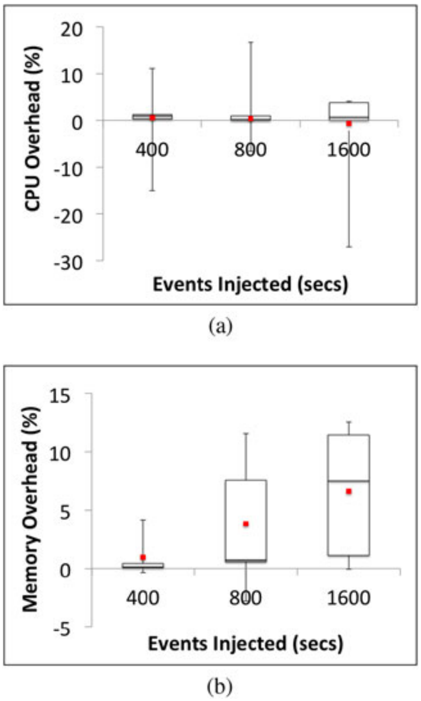


Fig. 8. Scalability—orbit test bed. (a) CPU Scalability. (b) Memory Scalability.

Scalability of Brace *Runtime Monitoring:* To evaluate *Brace*'s scalability, we again measured CPU and memory overheads, but this time in the presence of increasing numbers of injected events. Fig. 8 shows the results in the Orbit testbed; the values shown are relative to those in Figs. 6 and 7 with 384 tasks. With

TABLE I
LINOLEUM LOCAL PROPERTY ERROR REPORT

Property	*Found/Injected Errors*	*Wild Errors (Confirmed)*
LP1	10 / 10	0 (0)
LP2	10 / 10	7 (6)
LP3	10 / 10	22 (22)
LP4	10 / 10	1 (0)
LP5	10 / 10	2 (0)
LP6	10 / 10	0 (0)

TABLE II
ORBIT GLOBAL PROPERTY ERROR REPORT

Property	*Found/Injected Errors*	*Wild Errors (Confirmed)*
GP1	40 / 40	2 (2)
GP2	40 / 40	14 (10)
GP3	40 / 40	8 (4)

an increasing number of injected events (mimicking an increasing number of properties to monitor), CPU usage is basically constant (and actually decreases slightly). However, memory usage increases. In the scenario with 800 injected events, *Brace* periodically invokes load balancing on the global monitor nodes to guarantee the local runtime monitoring behavior; in the scenario with 1600 injected events, *Brace* uses the load balancing mechanism almost exclusively. This explains the flat to slight decline in (local) CPU overhead. The increase in memory overhead is explained by the storage of the events; even with load balancing, these events must be maintained before being sent to monitor nodes. We configured the experiment to send aggregate bundles to the monitor node every 10 s; if we increased the frequency, we expect memory usage to drop but CPU usage to increase. This result demonstrates that *Brace* style time-triggered online monitoring (with linear optimization[5] and load balancing combined) can guarantee the runtime monitor behavior with large number of events for complex CPS applications.

Effectiveness of Brace Runtime Monitoring: To measure the effectiveness of *Brace* in its support for runtime monitoring of CPS applications, we measured the degree to which it could find both injected and wild bugs in the applications under test. In the rover test bed, *Brace* detected all of the injected logic errors in all three environments. *Brace* also detected errors in these properties that are related to data collected from the rovers' sensors and, thus, related to the physical deployment. These errors were actual bugs discovered in the CPS program we were given. As one example, Table I shows the errors that *Brace* found for the run on the linoleum surface. As shown, *Brace* found an additional 22 wild errors for **LP3**. We verified with the CPS application developers that these were in fact legitimate (previously unknown) errors related to the scheduling algorithm that resulted in suboptimal solutions when relying on actual position data from the sensors. In the case of **LP2**, *Brace* found an additional seven errors over the ten 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. For brevity, we omit the results for the grass and wood floor deployments, but in both environments, *Brace* was able to detect all of the injected errors and to report additional true wild bugs (e.g., we found bugs related to the fact that the rover unexpectedly weaves on the grass surface due to the unexpected high level of friction).

Table II shows the results of our experiment on *Brace*'s effectiveness for the second application, with 40 randomly injected logic errors in the application's distributed algorithms. *Brace* detected all of the injected errors; again *Brace* also detected wild errors. For the first global property (**GP1**), all of the additional errors detected were confirmed by the application developers; they were related to a concurrency error in the scheduling algorithm. For the other two properties, *Brace* detected some confirmed wild bugs. *Brace* also detected errors that the application developers deemed false positives (especially for **GP3**). The logs show that this is a direct result of "bad" instrumentation in the application. To emulate a true environment, we asked the CPS application developers to perform the annotations that generate program traces. In some cases, this annotation is nontrivial. If the annotation is in a thread-critical location, then the required event observations are not immune from thread safety issues (in **GP3**, some waypoint acknowledgement messages/events are overwritten nondeterministically due to a subtle race condition); in this instance, incorrect instrumentation resulted in events that carried "false" data. This large number of false positives could be avoided by ensuring that the annotations are in the correct place(s), either by moving them outside of thread-critical sections or, when it is necessary to check these critical sections, to use static analysis to verify the annotation placement.

Validity Discussion: We did not conduct the same experiments for efficiency and scalability in the rover test bed as we did in the Orbit Test Bed. In Android, when we injected a large number of events per second, the profiling tools available to measure the CPU and memory usage incurred too much overhead, which changed the application's behavior. Better profiling tools for Android could make this evaluation possible; investigating such tools is an avenue for future work. This is not a deficit of *Brace*, but it made us unable to evaluate the overheads of *Brace* at a large scale on Android.

In the Orbit test bed, we did not analyze network transmission data. 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 uses. As a result, we do not know an exact value of how much additional network overhead is required for running the CPS application with *Brace* versus without it, especially for load balancing and checking global properties. Our *Brace* implementation did use our data aggregation, event filtering, and package compression schemes, but we cannot ascertain the degree of their impact on the observed application. Again, this does not point to a deficit of *Brace* but of the evaluation environment; one option is to use a dedicated network channel for *Brace* to avoid the impact on the hosted application.

We did not measure the CPU and memory overheads for checking global properties on monitor nodes in the Orbit test bed because we did not evaluate global properties in isolation from local ones (so the monitor node was always engaged in some amount of load balancing). However, the global monitor is exactly the same as the local monitor (using the same monitor synthesis algorithms in our implementation), it just executes on a

[5] We evaluated the overhead by measuring the total cost of running the linear optimization model in the experiment; the running time of each run is always less than 1 ms.

different (aggregate) trace. The event transformation algorithms required for checking global properties on the monitor nodes are proven quadratic in runtime complexity and linear in space complexity. We conducted a few experiments that demonstrated that the runtime overhead of running a global monitor is similar to running a local monitor. In the worst case, we can have a cluster of more computationally powerful monitor nodes in the debugging environment that we dedicate for use for the monitor nodes.

V. State of Research and Practice

Runtime verification is dynamic program analysis that aims to prove the correctness of programs at runtime. There are three key elements:

1) the language to specify properties to check;
2) the generation of traces on which to check properties;
3) algorithms that check the properties against the traces.

We target runtime verification for CPS by reducing computational and memory overheads of the approaches for these resource constrained applications.

A few efficient online monitoring algorithms for checking a formal specification have been proposed [2], [3], [5], [17], [18], and some are incorporated into tool sets [9], [26]. Among them, monitor oriented programming (MOP) [34] is one of the most efficient approaches, and some recent work is explicitly designed to improve monitoring efficiency in MOP, for example, by jointly exploring shared parameters and events across multiple specifications [23], or by improving the index trees and caches used to manage monitors' internal states [32]. However, when there is a large number of events monitored, MOP cannot guarantee a bounded computational and memory overhead; moreover, MOP does not provide any means to monitor global properties, which are essential in checking correctness of combined behavior of CPS components.

Eagle [2] is a monitoring logic that provides explicit operators (e.g., fix-point) to be extended to capture qualitative and quantitative constraints required for CPS. RuleR [3] improves Eagle by replacing the fix-point operators with flexible rules activated in response to observed events. Though Eagle and RuleR have the expressiveness needed for CPS and provide developers finer control of the monitor state, which leads to more efficient monitoring, they still are unable to guarantee bounded overheads. Moreover, these logics do not provide efficient means to check global properties. Adaptive runtime verification [4] uses an offline state estimation of the probability of a property being violated to assign a *critical level* to each monitor, which it uses to turn the monitor ON and OFF at runtime to control an overhead. Though this work is effective in reducing overhead, the resulting false positive and false negative rates are not acceptable for mission critical CPS.

Much of the overhead for runtime monitoring comes from trace generation [11], a finding consistent with our observations in our prior work [59]. While the above approaches have improved the inner state of the monitors or algorithms, we are also interested in controlling the overhead of instrumentation required to extract traces from an observed application.

Traditional runtime verification instrumentation calls a runtime monitor whenever an event associated with an observed property occurs [10], [54]. This can lead to *unpredictable* overhead whenever there are *bursts* of monitored events. Instead, in time-triggered online monitoring, a monitor is invoked periodically to sample events stored in a buffer. In [10], for instance, a linear programing model finds the longest sampling period with minimum event history buffer for time-triggered online monitoring. The work in [54] integrates event- and time-triggered monitoring to balance responsiveness and performance overhead. In both cases, the linear programing models are based on a statically analyzed control-flow graph (CFG); as a result, these approaches are intended for sequential programs. While they might work for more traditional embedded systems, they are not applicable to CPS, which are generally mobile, distributed, stochastic, and multithreaded. Our solution complements this work by introducing a linear programing model with finer granularity of control over overhead. We invoke the model using runtime inputs and complement the model with load balancing for backup. This is more resilient to stochastic changes in the number and frequency of observed events and, thus, more suitable to CPS.

In runtime monitors for distributed systems, online monitoring that allows LTL specifications to be distributed and observed by each component can: 1) reduce communication overhead (though some communication is still required to resolve nonlocal properties) and 2) increase responsiveness in detecting violations at the expense of performance overhead on the local component [6]. Such an approach is similar to ours in that it explores the use of distributed monitoring, but it assumes perfect synchrony among components and that the sets of events on different components are unique. In many CPS, perfect synchrony cannot be guaranteed (and is very unlikely), and the sets of events on different components often intersect. We use event transformation algorithms to reduce distributed online monitoring into a simpler problem of checking language membership, which handles network noise and delay and has no restriction on event set similarity.

In passive distributed assertions [45], distributed assertions are instrumented in the application and evaluated at a single global node. Properties used for checking are collected passively from the network. Though this provides lower communication overhead, it has little or no control over the computational and memory overhead of checking distributed properties. Moreover, many CPS properties are locally observable, and checking them only at a global node reduces responsiveness and significantly restricts the prospects of runtime feedback and recovery, which is an essential piece of what makes runtime verification attractive for CPS.

We focus on ensuring performance of a runtime verification system while not sacrificing monitoring accuracy (in terms of false positives and false negatives) and minimizing the impact on the behavior of the monitored application.

Our work is complementary to model-based verification [14] where the combination of Simulink and Simulink Design Verifier are used to check (low-level) properties of the models in simulation. Such a model-based approach can be applied in the earlier stages of CPS application verification to reduce later verification costs, but our approach provides the last line of defense where the actual implementation is verified at runtime against the deployment environment.

Our study also differs from hybrid verification of embedded software [7], in which a function call graph derived from an underlying C program is verified against LTL properties using software model checkers (e.g., CBMC [29] and ESBMC [37]). For those functions that are too complex to be verified (i.e., *Marked Function*), hybrid verification launches a simulation in which a SystemC [8] model derived from the embedded software is connected to the SystemC Temporal Checker [53]. The simulation model is executed in parallel with the temporal checker, which monitors the inputs and outputs of each *Marked Function* to detect violations. Our study verifies properties more suitable for CPS at runtime, which are distributed (e.g., across multiple application nodes) and complex (e.g., quantitative constraints and predicate logics).

In [57] and [58] combined static and dynamic information to predict future behaviors of an observed CPS application is used. The goals are complementary to ours; in fact, we could use these approaches within *Brace* to validate a developer's event instrumentation and improve monitoring accuracy. However, these approaches use either a CFG or a program dependency graph to gather dynamic information. For distributed large-scale CPS, the inherent lack of support for interactions among distributed processes make these approaches suboptimal [55]. In [50], [52], and [51], multitier context information (e.g., car, road, and weather conditions) and cloud computation detect malicious behaviors and predict future behavior in vehicular CPS. We intend to combine these works and predictive semantics in future work to *predict* violation of safety-critical properties. Further, we will explore connecting with *automatic program repair* to recover (at runtime) from violated properties, where a genetic programing approach [16] can search for repair candidates as variants of original source codes with different sequences of edits. To make this approach more effective in CPS, we can infuse the notion of allowed statespace [28], where each repair candidate must function within acceptable physical limits (e.g., landing altitude and pitch angle for an autonomous aircraft).

VI. Conclusion

In this paper, we present *Brace* as an approach to improve efficiency and scalability of runtime monitoring of CPS. *Brace* uses a combination of a linear programing model created specifically for time-triggered online monitoring for CPS, load balancing, and lightweight event aggregation and transformation algorithms to minimize the overhead of runtime monitors. With a thorough case study both on a real rover application and a complex simulation on distributed robotic planning, we prove *Brace* as efficient, effective, and scalable for CPS runtime verification.

References

[1] R. Alur *et al.*, "A determinizable class of timed automata," in *Proc. Comput.-Aided Verification*, 1994, pp. 1–13.

[2] H. Barringer *et al.*, "Program monitoring with LTL in EAGLE," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2004, pp. 3617–3624.

[3] H. Barringer *et al.*, "Rule systems for run-time monitoring: From EAGLE to RULER," *J. Logic Comput.*, vol. 20, pp. 675–706, 2010.

[4] E. Bartocc *et al.*, "Adaptive runtime verification," in *Proc. Runtime Verification*, 2012, pp. 168–182.

[5] A. Bauer *et al.*, "Comparing LTL semantics for runtime verification," *J. Logic Comput.*, vol. 20, no. 3, pp. 651–674, 2010.

[6] A. Bauer and Y. Falcone, "Decentralised LTL monitoring," in *Proc. Formal Methods*, 2012, pp. 85–100.

[7] J. Behrend *et al.*, "Scalable and optimized hybrid verification of embedded software," *J. Electron. Test.*, vol. 31, no. 2, pp. 151–166, 2015.

[8] D. C. Black *et al.*, "*SystemC: From the Ground Up: From the Ground Up*. New York, NY, USA: Springer, 2009.

[9] E. Bodden, "A lightweight LTL runtime verification tool for java," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2004, pp. 306–307.

[10] B. Bonakdarpour *et al.*, "Time-triggered runtime verification," *Formal Methods Syst. Des.*, vol. 43, no. 1, pp. 29–60, 2013.

[11] L. Pike, "Study on run time assurance for complex cyber physical systems," Air Force Res. Lab., Tech. Rep. ADA585474, Apr. 2013. [Online]. Available: http://www.cs.indiana.edu/∼lepike/pubs/RTA-CPS.pdf

[12] I. Craggs, "Really small message broker. (2013). [Online]. Available: http://www.alphaworks.ibm.com/tech/rsmb/

[13] J. C. Eidson *et al.*, "Distributed real-time software for cyber–physical systems," *Proc. IEEE*, vol. 100, no. 1, pp. 45–59, Jan. 2012.

[14] J. Etienne *et al.*, "Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain," in *Proc. Complex Syst. Des. Manage.*, 2010, pp. 61–72.

[15] C. Fetzer and F. Cristian, "An optimal internal clock synchronization algorithm," in *Proc. 10th Annu. Conf. Comput. Assurance Syst. Integrity, Softw. Safety Process Security*, 1995, pp. 87–196.

[16] C. L. Goues *et al.*, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[17] K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *Proc. 8th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2002, pp. 342–356.

[18] K. Havelund and G. Roşu, "Efficient monitoring of safety properties," *Int. J. Softw. Tools Technol. Transfer*, vol. 6, no. 2, pp. 158–173, 2004.

[19] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2013.

[20] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*. Noida, India: Pearson Educ. India, 1979.

[21] B. Horling *et al.*, "The soft real-time agent control architecture," *Auton. Agents Multi-Agent Syst.*, vol. 12, no. 1, pp. 35–91, 2006.

[22] U. Hunkeler *et al.*, "MQTT-S—A publish/subscribe protocol for wireless sensor networks," in *Proc. 3rd Int. Conf. Commun. Syst. Softw. Middleware Workshops*, 2008, pp. 791–798.

[23] J. Dongyun, M. P. O'Neil, and R. Grigore, "Scalable parametric runtime monitoring," Illinois Digital Environ. for Access to Learning and Scholarship, Tech. Rep. 2012-04-24, 2012. [Online]. Available: http://hdl.handle.net/2142/30757

[24] G. Jourjon *et al.*, "From learning to researching, ease the shift through testbeds," in *Proc. 6th Int. ICST Conf. TridentCom 2010*, 2010, pp. 496–505.

[25] W. Kang, K. Kapitanova, and S. H. Son, "RDDS: A real-time data distribution service for cyber-physical systems," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 393–405, May 2012.

[26] M. Kim *et al.*, "Java-MaC: A run-time assurance approach for java programs," *Formal Methods Syst. Des.*, vol. 24, no. 2, pp. 129–155, 2004.

[27] G. Klein *et al.*, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, 2014, Art. no. 2.

[28] C. M. Krishna and I. Koren, "Adaptive fault-tolerance fault-tolerance for cyber-physical systems," in *Proc. Int. Conf. Comput., Netw. Commun.*, 2013, pp. 310–314.

[29] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *Proc. 20th Int. Conf., Held Part Eur. Joint Conf. Theory Practice Softw.*, 2014, pp. 389–391.

[30] V. Lesser *et al.*, "Evolution of the GPGP/TAEMS domain-independent coordination framework," *Auton. Agents Multi-Agent Syst.*, vol. 9, no. 1, pp. 87–143, 2004.

[31] D. Locke, "Mq telemetry transport (MQTT) v3. 1 protocol specification," *IBM developerWorks Tech. Library*, 2010.

[32] Q. Luo *et al.*, "RV-monitor: Efficient parametric runtime verification with simultaneous properties," in *Proc. 5th Int. Conf. Runtime Verification*, 2014, pp. 285–300.

[33] Memory analyzer. [Online]. Available: http://www.eclipse.org/mat/, 2015.

[34] P. Meredith *et al.*, "An overview of the MOP runtime verification framework," *Softw. Tools Technol. Transfer*, vol. 14, no. 3, pp. 249–289, 2011.

[35] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Trans. Commun.*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991.

[36] S. Mitra *et al.*, "Verifying cyber-physical interactions in safety-critical systems," *IEEE Security Privacy*, vol. 11, no. 4, pp. 28–37, Jul./Aug. 2013.

[37] J. Morse *et al.*, "ESBMC 1.22," in *Proc. 20th Int. Conf. Held Part Eur. Joint Conf. Theory Practice Softw.*, 2014, pp. 405–407.

[38] H. Muccini *et al.*, "Self-adaptation for cyber-physical systems: A systematic literature review," in *Proc. 11th Int. Symp. Softw. Eng. Adaptive Self-Managing Syst.*, 2016, pp. 75–81.

[39] S. Navabpour *et al.*, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *Proc. 2nd Int. Conf. Runtime Verification*, 2012, pp. 208–222.

[40] Paho. (2014). [Online]. Available: http://eclipse.org/paho/clients/java/

[41] T. Rakotoarivelo *et al.*, "Designing and orchestrating reproducible experiments on federated networking testbeds," *Comput. Netw.*, vol. 63, pp. 173–187, Apr. 2014.

[42] J. F. Raskin and P. Y. Schobbens, "State clock logic: A decidable real-time logic," in *Proc. Int. Workshop Hybrid Real-Time Syst.*, 1997, pp. 33–47.

[43] D. Raychaudhuri *et al.*, "Overview of ORBIT radio grid testbed for evaluation of next-generation wireless network protocols," in *Proc. Wireless Commun. Netw. Conf.*, 2005, pp. 308–309.

[44] A. Rogers *et al.*, "Bounded approximate decentralised coordination via the max-sum algorithm," *Artif. Intell.*, vol. 175, no. 2, pp. 730–759, 2011.

[45] K. Romer and J. Ma, "PDA: Passive distributed assertions for sensor networks," in *Proc. Int. Conf. Inf. Process. Sensor Netw.*, 2009, pp. 337–348.

[46] Scpsolver—An easy to use java linear programming interface. (2015). [Online]. Available: http://scpsolver.org//

[47] A. S. Tanenbaum and M. Van S, *Distributed Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2007.

[48] Traceview. [Online]. Available: http://developer.android.com/tools/debugging/debugging-tracing.html, 2015.

[49] J. D. Valois, "Lock-free linked lists using compare-and-swap, in *Proc. 14th Annu. ACM Symp. Principles Distrib. Comput.*, 1995, pp. 214–222.

[50] J. Wan *et al.*, "Cyber-physical systems for optimal energy management scheme of autonomous electric vehicle," *Comput. J.*, vol. 56, no. 8, pp. 947–956, 2013.

[51] J. Wan *et al.*, "Context-aware vehicular cyber-physical systems with cloud support: Architecture, challenges, and solutions," *IEEE Commun. Mag.*, vol. 52, no. 8, pp. 106–113, Aug. 2014.

[52] J. Wan *et al.*, "VCMIA: A novel architecture for integrating vehicular cyber-physical systems and mobile cloud computing," *Mobile Netw. Appl.*, vol. 19, no. 2, pp. 153–160, 2014.

[53] R. Weiss *et al.*, "Efficient and customizable integration of temporal properties into systemc," in *Proc. Appl. Specification Des. Lang. SoCs*, 2006, pp. 101–114.

[54] C. Wu *et al.*, "Reducing monitoring overhead by integrating event-and time-triggered techniques," in *Proc. Runtime Verification*, 2013, pp. 304–321.

[55] B. Xin *et al.*, "Lightweight task graph inference for distributed applications," in *Proc. 29th IEEE Symp. Rel. Distrib. Syst.*, 2010, pp. 82–91.

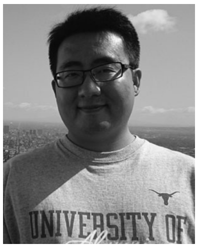
[56] H. Yoon *et al.*, "Mobility emulator for DTN and manet applications," in *Proc. 4th ACM Int. Workshop Experimental Eval. Characterization*, 2009, pp. 51–58.

[57] K. Yu *et al.*, "A predictive runtime verification framework for cyber-physical systems," in *Proc. 2014 IEEE 8th Int. Conf. Softw. Security Rel.-Companion*, 2014, pp. 223–227.

[58] X. Zhang *et al.*, "Runtime verification with predictive semantics," in *Proc. 4th Int. Conf. NASA Formal Methods*, 2012, pp. 418–432.

[59] X. Zheng *et al.*, "Braceassertion: Runtime verification of cyber-physical systems," in *Proc. IEEE 12th Int. Conf. Mobile Ad-Hoc Sensor Syst.*, 2015.

[60] X. Zheng *et al.*, "Verification and validation in cyber physical systems: Research challenges and a way forward," in *Proc. 2015 IEEE/ACM 1st Int. Workshop Softw. Eng. Smart Cyber-Phys. Syst.*, 2015, pp. 15–18.



(James) Xi Zheng (GS'14–M'15) received the Bachelor's degree in computer information systems from FuDan University, Shanghai, China, in 2001. He received the Master's degree in information science from the University of New South Wales, Sydney, NSW, Australia, in 2006, and the Ph.D. degree in software engineering from The University of Texas at Austin, Austin, TX, USA, in 2015.

He is a Lecturer/Assistant Professor in computer science with Deakin University, Burwood, VIC, Australia. Before pursuing the Ph.D. degree in the USA, he had been working as a Senior Principal Consultant and a Solution Architect in Sydney for more than eight years. His current research focuses on the design and implementation of middlewares for cyber physical systems (CPS) and Internet of Things in general. His Ph.D. thesis looks into a practical way of bringing formal methods (e.g., temporal logics and automata theories) and physical models (in terms of real-time simulation) into CPS runtime verification.

Christine Julien (M'06–SM'14) received the B.S. degree (with majors in computer science and biology), and the M.S. and the D.Sc. degrees from Washington University, St. Louis, MO, USA, in 2000, 2003, and 2004, respectively.

She is a Professor with the Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA. Her research focuses on the intersection of software engineering and dynamic, unpredictable networked environments and develops models, abstractions, tools, and middleware whose goals are to ease the software engineering burden associated with building applications for pervasive and mobile computing environments. Her research has been supported by the National Science Foundation, the Air Force Office of Scientific Research, the Department of Defense, Freescale Semiconductors, Google, and Samsung, and the results have appeared in many peer reviewed journal and conference papers.

Rodion Podorozhny received the M.Sc. degree in computer science from the University of Massachusetts Amherst, Amherst, MA, USA, in 1997, and the Ph.D. degree in software engineering from The University of Texas at Austin, Austin, TX, USA, in 2004.

He joined Texas State University–San Marcos, San Marcos, TX, USA, in 2004. His research interests include the area of software engineering, in particular in software process and formal methods. In addition, he has done some work in analysis of multiagent systems and application of the multiagent technology to software process enactment.

Franck Cassez (M'08) is an Associate Professor with the Department of Computing, Macquarie University, Sydney, NSW, Australia. From 2012 to 2014, he was a Principal Researcher with National Information and Communications Technology Australia. From 2008 to 2011, he was a Marie Curie Research Fellow (7th framework programme of the European Community). Prior to 2008, he was a Research Scientist with the French National Centre for Scientific Research. His main topics of interests are the specification, verification of software systems, static analysis, model checking, and controller synthesis for infinite and/or timed systems.

Thierry Rakotoarivelo received the Ph.D. degree in cotutelle from the University of New South Wales, Sydney, NSW, Australia and the Institut National Polytechnique de Toulouse, Toulouse, France, for his work on the distributed discovery of alternate Internet paths with enhanced Quality-of-Service.

He is a Senior Research Scientist with the Network Research Group, CSIRO Data 61, NICTA, Eveleigh, NSW, Australia. His research focuses on large scale cyber-physical systems such as IoT systems, sensor networks, or Internet-wide testbeds. He has been working on the design and orchestration of these systems, and their use in experiment-driven evaluations and measurement campaigns. He is also interested in systems to collect and create the produced data, and the analysis and algorithms to extract meaningful information from these data. Prior to the Ph.D. degree, he was a Senior Research Engineer with the Motorola Australian Research Centre (MARC, now closed), where he was involved with QoS oriented MAC protocols for wireless LANs.