ULGEN: A Runtime Assurance Framework for Programming Safe Cyber–Physical Systems

Beyazit Yalcinkaya, Hazem Torfah, Ankush Desai, and Sanjit A. Seshia, *Fellow, IEEE*

***Abstract*—We present ULGEN, a runtime assurance (RTA) framework for programming safe cyber–physical systems (CPSs). In ULGEN, a system is implemented as a collection of asynchronous processes executing RTA modules which are generalizations of the well-known Simplex architecture. An RTA module is composed of a set of safe controllers (SCs), designed to guarantee certain safety specifications, and a set of advanced controllers (ACs), optimized for performance, each defined to run under the specific conditions of the operating environment, and a decision module implementing the switching logic between the controllers. A source of complexity in achieving safe CPS is that these systems often involve concurrently interacting components with different execution semantics. To this end, ULGEN allows for the definition of RTA modules with either event-driven or time-driven execution semantics and encapsulates such components into RTA modules. It further provides primitives for implementing priority-based communication between asynchronous processes, which is a necessary feature for task prioritization mechanisms, such as contingency plans and interrupt service routines. The framework also provides formal guarantees on the safe execution of RTA modules based on a formal definition of well-formedness. In ULGEN, a well-formed RTA module combines SCs and ACs in a way that guarantees the underlying safety specifications assured by the SCs while delivering the desired performance offered by the ACs. We compare the safety guarantees of ULGEN against other state-of-the-art RTA frameworks and demonstrate its efficacy in implementing safe and performant CPS by presenting an extensive experimental evaluation of five case studies both in a simulation environment and on a real robotic platform.**

***Index Terms*—Cyber–physical systems (CPSs), formal methods, robotics, runtime assurance (RTA), runtime verification.**

I. INTRODUCTION

DEPLOYING autonomous cyber–physical systems (CPSs) in safety-critical domains present several challenges, including the complexity of their operating environments and the growing trend of using learning-enabled components in tasks, such as perception, prediction, and planning [37]. It is, therefore, crucial to develop techniques for runtime assurance (RTA) of CPS, including runtime monitoring and safe fallback mechanisms. Autonomous CPS involves complex and uncertified software, such as learning-based components that can be hard to verify. Runtime monitors are, thus, necessary for maintaining situational awareness and triggering the right contingency plans and interrupt-service routines to maintain safety.

Programming safe CPS instrumented with runtime monitors requires careful design and implementation of their RTA architecture. Several frameworks have been presented in the literature introducing systematic ways for implementing and integrating RTA modules into CPS [7], [14], [20], [21]. One example is the SOTER framework [7], which provides a language for programming robotic systems with time-driven RTA modules. Another example is the ROSRV framework [20] building on the prominent robot operating system (ROS) [34]. ROSRV allows for the definition and integration of monitoring nodes that intercept commands and messages passing through the system when the underlying monitoring specification is violated. These and other frameworks have shown promising results in building safe CPS, but they are limited to certain communication and computation models. Moreover, autonomous CPS are complex systems composed of various concurrently interacting components often based on different execution semantics. Some are time-driven, i.e., they employ a fixed execution frequency. Others are event driven, i.e., their execution depends on the arrival of data from other components. Therefore, a general framework for programming safe autonomous CPS should provide flexibility and the right tools for implementing RTA modules with different communication and computation models while providing guarantees on the safety of the system behavior.

In this article, we present ULGEN,[1] a framework for designing and programming asynchronous autonomous CPS with both event-driven and time-driven RTA modules and with an underlying formalism for building provably safe RTA modules. In ULGEN, a program defines a set of asynchronously communicating processes interacting with each other using a message-passing communication model. Its language builds on *P* [8], an event-driven asynchronous programming language. ULGEN extends *P* with: 1) primitives for RTA modules used as *building blocks* for programming safe systems; 2) support for *both* time-driven and event-driven execution semantics for RTA modules; and 3) event priorities for task prioritization.

Manuscript received 6 August 2022; revised 19 December 2022; accepted 26 January 2023. Date of publication 17 February 2023; date of current version 20 October 2023. This work was supported in part by NSF under Grant 1545126 (VeHICaL) and Grant 1837132; in part by DARPA under Contract FA8750-18-C-0101 (AA); in part by Berkeley Deep Drive; in part by C3DTI; and in part by Toyota through the iCyPhy Center. This article was recommended by Associate Editor A. Shrivastava. *(Corresponding author: Beyazit Yalcinkaya.)*

Beyazit Yalcinkaya, Hazem Torfah, and Sanjit A. Seshia are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720 USA (e-mail: beyazit@berkeley.edu; torfah@berkeley.edu; sseshia@berkeley.edu).

Ankush Desai is with the Amazon Web Services, Amazon Inc, Cupertino, CA 95014 USA (e-mail: ankushpd@amazon.com).

Digital Object Identifier 10.1109/TCAD.2023.3246386

[1] In Turkic and Mongolian mythology, Ülgen is a creator-deity who assumes the protectorship of humankind against the god of evil and darkness.

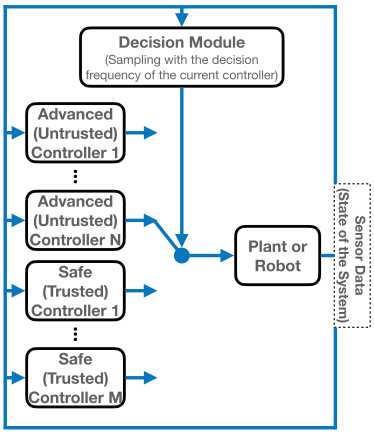


Fig. 1. RTA module architecture of the ULGEN framework.

An RTA module in ULGEN is a generalization of the widely used *Simplex architecture* [38]. In Simplex, an RTA module consists of three components: 1) an advanced (untrusted) controller (AC/UC); 2) a safe (trusted) controller (SC/TC); and 3) a decision module (DM). The AC is used for operating the system under nominal circumstances and is designed to achieve high performance. This controller may be an off-the-shelf (black-box) component that is not necessarily certified. The SC is a certified backup controller that takes over when, for example, abnormalities in the execution of the AC are observed or when the system exits the safe operational environment of the AC. The DM is a trusted component that implements the switching logic between the AC and the SC. The switching logic is based on a safety specification that if violated, triggers a switch in operation from the AC to the SC. If at some point the system is brought back to a safe state, the DM switches back to the AC. ULGEN generalizes the Simplex architecture by allowing an RTA module to comprise a set of two or more controllers (see Fig. 1). For example, different controllers may be designed for different environmental conditions (e.g., weather, time of the day, etc.) as those can have significant effects on the behavior of the system (e.g., see [16]). Thus, in practice, a DM should implement a switching logic between a larger set of controllers. Furthermore, in comparison to the standard Simplex architecture, where the DM operates periodically, according to a fixed frequency, thus, limiting the RTA module to time-driven semantics, the ULGEN language allows for the definition of both time-driven and event-driven RTA modules. The execution frequency of a DM in ULGEN is bound to the *decision frequency* of the current controller of the RTA module, which enables us to extend traditional RTA module architecture to event-driven semantics. Finally, an important feature of ULGEN is that it allows for the definition of priority-based communication, i.e., priority-based message passing, where each asynchronous process operates on its local event queue, and events are associated with different scheduling priorities. This mechanism is necessary for implementing task prioritization such as in interrupt-service mechanisms over different controllers and allows for the implementation of coordinated switching mechanisms between RTA modules.

The underlying formalism of ULGEN allows for the implementation of provably safe RTA modules. We present a set of *well-formedness* properties for ULGEN programs and show that satisfying these guarantees the safety of the system. We prove that if the initial state of a well-formed RTA module is within its operating region, then it will never leave this region.

We demonstrate the efficacy of ULGEN in five robotic case studies and show that ULGEN enables the modular design of robotic systems and can flexibly interface with well-established monitoring tools and robotic platforms. We showcase this by interfacing ULGEN with Reelay [41], a monitoring tool for metric temporal logic (MTL) [23], and by implementing the software stack of a real robotic platform.

We summarize our contributions as follows.

1) We introduce the ULGEN framework. ULGEN uses RTA modules as building blocks for asynchronous processes. It provides both event-driven and time-driven execution semantics for RTA modules, extends the traditional Simplex architecture to multiple controllers, and defines a priority-based communication mechanism for task prioritization.
2) We present a theoretical formalism for the well-formedness of systems designed with ULGEN and prove that such systems satisfy their safety specifications.
3) We demonstrate the efficacy of ULGEN in implementing safe and performant CPS by presenting an extensive experimental evaluation of case studies both in a simulation environment and on a real robotic platform.

The outline is as follows. Section II provides an overview, Section III is the formalism, Section IV presents our toolkit, Sections V–VII demonstrate case studies, Section VIII discusses the related work, and Section IX concludes this article.

II. OVERVIEW OF ULGEN

We present a case study to reveal high-level details of ULGEN and to introduce the ULGEN language. Further details and evaluation of the case study are given in Section VI-A.

A. Motivating Example

Consider a robot surveillance system, where a robot is tasked to visit a sequence of waypoints infinitely often while ensuring that the mission never fails due to battery deprivation. For simplicity, we assume that the environment is known to the robot a priori. To safely execute the mission, the robot must unceasingly satisfy the following four specifications: **(S1)** visit surveillance locations in the correct order; **(S2)** compute reachable plans between locations; **(S3)** closely follow a reference trajectory between waypoints; and **(S4)** never deplete all of the battery. While assuring the safety requirements, the robot also needs to provide adequate performance in terms of reaching its target waypoints as fast and as efficiently as possible. Advanced off-the-shelf software can be used to achieve such performance, e.g., for motion planning, third-party libraries such as OMPL [40]. However, the execution of plans computed by such components needs to be monitored and validated against the safety requirements, e.g., a plan should not lead to battery deprivation, i.e., a failure.

Implementing this system to satisfy the safety and performance requirements stated above requires careful design of the communication and execution of the several components

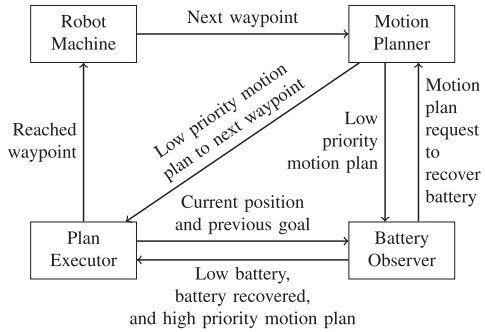


Fig. 2. Compositional RTA system, boxes are asynchronous components.

of the system. A high-level design of the software architecture is shown in Fig. 2. It consists of the following components.

(S1): A robot machine manages the tasks to be executed by the system and serves as the interface between an application layer and the motion planner.

(S2): The motion planner computes a motion plan for a task and forwards the plan to the plan executor.

(S3): The plan executor executes the corresponding motion primitives, i.e., low-level controls, to complete the motion plan.

(S4): Finally, the battery observer monitors the battery levels of the robot.

The robot machine is notified by the plan executor through a specific event indicating that the plant has reached the current waypoint, see the arrow from the plan executor to the robot machine in Fig. 2. Once a plan is executed, the next waypoint is forwarded to the motion planner. All components in Fig. 2 communicate asynchronously. For example, the battery observer concurrently communicates with both the motion planner and the plan executor to request battery charging plans when necessary, or to inform that a battery recovery procedure has been completed. Supporting both event-driven and time-driven RTA modules is a necessity for implementing our robot system. While the robot machine and motion planner are event-driven since motion planning should occur only when there is a location to go to, the plan executor and the battery observer need to be time driven. The plan executor executes certain motions, and the battery observer monitors the battery level in fixed time periods. All in all, we need a modular way to implement the different asynchronous components, we need to be able to define different execution behaviors for each component, and finally, we need a mechanism for priority-based event handling as the system has multiple concurrent objectives with different priorities, e.g., when the battery is critically low, recovering the battery has a higher priority than reaching the next waypoint.

B. Programming With ULGEN

The ULGEN language builds on *P*, an asynchronous event-driven programming language that unifies modeling, programming, and verification [8]. We start by giving some background on *P* and then explain the new features of ULGEN.

1) P Programming Language [8]: A *P* program consists of a set of state machines communicating asynchronously with each other in a message-passing fashion using events. In our

```
1  ////////// Event Declarations ////////////////
2  event eMotion: locationType;
3  event eMotionX priority 10: locationType;
4  event ePlanRequest priority 1: requestType;
5  event ePlanRequestX priority 10: requestType;
6  event eBatteryLow priority 10: machine;
7  ////////////// State Machines ///////////////
8  machine RobotMachine {... // Machine definition}
9  machine MotionPlanner {
10   var dest: machine;
11   var currLoc: locationType;
12   var goalLoc: locationType;
13   ... // Helper function definitions
14   fun f_DM(): string {//Controller switching logic
15     if (isInUCRegion(currLoc, goalLoc)) return "UC";
16     else return "TC";
17   }
18   fun UC(){send dest,ePlan,getUnTPlan(currLoc,goalLoc);}
19   fun TC(){send dest,ePlan,getTPlan(currLoc,goalLoc);}
20   fun handler(payload:requestType){ // Event handler
21     dest=payload.0; currLoc=payload.1; goalLoc=payload.2;
22   }
23   start state Init {entry {goto Run;}}
24   state Run {
25     rtamodule {
26       controller UC; // Untrusted controller
27       controller TC; // Trusted controller
28       decisionmodule f_DM @ {UC: 1, TC: 1};
29       on ePlanRequest, ePlanRequestX with handler;
30     }
31   }
32 }
33 machine PlanExecutor {
34   ... // Controllers, f_DM, and helper function definitions
35   state Run {
36     rtamodule {
37       controller SafeMotionC period 16 ms;
38       controller AdvMotionC period 16 ms;
39       decisionmodule f_DM @ {SafeMotionC:50,
40                              AdvMotionC:10};
41     }
42     on eMotion do appendToMotionsList;
43     on eMotionX do appendToHPMotionsList;
44     on eBatteryLow goto LowBatteryRun
45       with lowBatteryEventHandler;
46   }
47   state LowBatteryRun {...}
48 }
49 machine BatteryObserver {... // Machine definition}
```

Fig. 3. ULGEN Program implementing the robot surveillance case study.

example in Fig. 3, we declare four state machines for the robot machine (line 8), the motion planner (line 9), the plan executor (line 33), and the battery observer (line 49). Details are given for the motion planner and plan executor state machines. Events communicated between the state machines are declared at the beginning of the program (lines 2–6). Notice that between lines 2 to 6 only line 2 is native *P* syntax as *P* does not support event priorities. A state machine consists of several states. For example, the motion planner has two states (declared in lines 23 and 24), an initial state that switches to `Run` once the machine starts executing. Transitions between the states are defined using the `goto` keyword. Actions of a state or a transition to another state can also be triggered by the receipt of an event, e.g., in the `Run` state of plan executor, in line 44, upon receiving `eBatteryLow` a transition to `LowBatteryRun` is taken and a predefined procedure `lowBatteryEventHandler` is executed. Received events are placed into local FIFO queues and fetched from there.

P has a systematic testing engine for identifying incorrect behaviors of the model, e.g., deadlock detection. It can be translated to executable C code and is, therefore, a powerful language that bridges the gap between the high-level model definition and the low-level implementation. *P* has been used to develop the USB 3.0 driver inside Windows 8.1 [8] and also for building reliable distributed systems [9]. Thanks to its state machine-based asynchronous event-driven semantics, it has also been used for programming various robotic systems [6], [7], [10], [39]. For implementing the intended functionality of

the machines in our example above, P is, however, missing some features. In ULGEN, we build on P and extend it to a general language for building safe CPS by adding features for defining real-time behavior, event prioritization, and language primitives for the definition of RTA modules, as we show next.

2) ULGEN *Programming Language:* In ULGEN, we follow an RTA-based approach to implementing the states of a state machine. Specifically, the ULGEN language expands on P by adding new building blocks called RTA modules declared by the keyword `rtamodule` in the states. We call machines with RTA modules runtime-assured state machines (RTA-SMs). An example of an RTA module is given in line 25 in Fig. 3. An `rtamodule` block is defined over several controllers, each identified by the keyword `controller`, and a DM identified by the keyword `decisionmodule`. Implementations of the controllers and the controller switching logic are given earlier and are declared by the keyword `fun` (which defines a sequential procedure), as in lines 14–22. For this RTA module, we use an AC named `UC` that computes a plan using some off-the-shelf motion planning library, and an SC named `TC`, that uses a certified, yet not necessarily optimal planner. The controller switching logic is defined by the `f_DM` function (lines 14–17). The `f_DM` function (defining the controller switching logic) and the decision frequency mapping, together, form the DM (line 28), which is compiled into a procedure that runs according to the ULGEN semantics. Controllers are assigned to regions, i.e., subsets of system states. The union of all controller regions is called the operating region of the machine, which must cover the state space over which the RTA is defined, and moreover, it must partition this space. In motion planner, the RTA module checks whether the machine is within the region of `UC`, and if not switches to `TC`, and vice versa, as shown in lines 15 and 16. Each state of an RTA-SM can have at most one RTA module.

The RTA module in line 25 is an event-driven RTA module, i.e., the currently operating controller is executed once every time a `ePlanRequest` or `ePlanRequestX` event is processed. Upon receiving an event, `handler` is executed before running the current controller of the RTA module. The DM is executed in accordance with the execution of the operating controller. In the RTA module in line 25, the DM is run before every single execution of `UC` or `TC`, as defined in line 28. Thus, for this specific RTA module, every execution may trigger a controller switch depending on the current region.

ULGEN also supports time-driven RTA modules, e.g., the plan executor. The RTA module in line 36 is defined over `SafeMotionC` and `AdvMotionC` operating based on fixed frequencies, both with 16-ms periods in the example. As in the event-driven case, the DM is dependent on the decision frequency of the executing controller. Here, the DM executes every 50 executions of the `SafeMotionC` controller or every ten executions of `AdvMotionC`, as defined in lines 39 and 40. A time-driven RTA module does not allow an `on` statement as it operates independently from any events received.

Finally, events in ULGEN are declared with a corresponding priority. Priorities are needed to implement task prioritization. For example, if the battery observer issues a request to the motion planner to compute a plan to recharge the battery of the robot (`ePlanRequestX`), then this task needs to be executed before executing any plan requested by the robot machine (`ePlanRequest`). In our example, `ePlanRequest` (priority 1) has a lower priority than `ePlanRequestX` (priority 10). In ULGEN, priorities are handled by replacing the FIFO queues of P with priority queues. In this case, events with higher priorities are executed first, e.g., battery recovery plans.

3) ULGEN *Programming Framework:* To define an RTA-SM in ULGEN, the programmer defines a machine (e.g., lines 9–32), its local functions implementing the controllers (e.g., lines 18 and 19), the controller switching logic (e.g., lines 14–17), and finally declares either event-driven or time-driven RTA modules in desired states (e.g., lines 25–30). Since P allows foreign function calls to C/C++, programmers can utilize existing controller implementations and still get the RTA guarantees of ULGEN. By calling interface functions, programmers can generate runtime monitors for temporal specifications, log data to monitors, and check the latest statuses of monitors. In the controller switching logic (e.g., lines 25–30), monitors can be used to select the controller. The ULGEN toolkit provides an interface for the Reelay runtime monitoring tool [41], which can be extended to any state-of-the-art monitoring tool. The ULGEN type system ensures that RTA modules and RTA-SMs follow all formal definitions, e.g., each controller has an associated decision frequency, see Section IV for the details of the ULGEN language. A ULGEN program satisfying well-formedness properties formalized in Section III keeps the system safe, i.e., in the operating region.

III. FORMALISM

In this section, we give a formalization of ULGEN programs by defining syntax and semantics of: 1) RTA modules with multiple controllers; 2) RTA-SMs; and 3) compositional RTA-SM systems. We present the formalization for event-driven RTA modules and show that it can be easily extended to the time-driven case. Based on the given formalization, we define the well-formedness properties of an RTA module and an RTA-SM. We further show that a ULGEN program that is a composition of well-formed RTA-SMs always keeps the system safe, i.e., in its operating region. Finally, we conclude by emphasizing the correspondence between the presented formalism and the ULGEN language, and give a guideline for satisfying the well-formedness properties in practice.

We start by defining a few concepts that we will be using in the formalism. An *event* $e \in E$ is a name, where E is the universe of all events. Each event is associated with an *event priority* value given by a mapping $p : E \to \mathbb{N}$. For two events e_h and e_l, $p(e_h) > p(e_l)$ indicates that e_h has a higher priority than e_l. Let V denote the universe of all values, for simplicity, we assume that all *variables* share the same value universe V. For a set of variables X and a valuation $v : X \to V$, we use $v(x)$ to access the value of $x \in X$, we use $v[Y] \in Y \to V$ to get the valuation of $Y \subseteq X$ from v, and we use $v[x \leftarrow c]$ to denote a new valuation that updates the valuation of $x \in X$ with $c \in V$ and that is equal to v for all other variables in X. A *controller* $c : (X \to V) \to (X \to V)$ is a function running on a variable valuation and returning a variable valuation.

A. Runtime Assurance Modules

We formally define RTA modules (RTA modules), their *well-formedness* properties, the notion of execution for an RTA

module, and prove that a well-formed RTA module keeps the system in a region where its behavior is defined.

Definition 1 (RTA Module): An RTA module is defined as a tuple $\mathcal{M} = (C, X, \phi, \Delta, \text{Triggers})$, where:

1) C is the set of controllers executed by the RTA module;
2) X is the set of local variables. It particularly contains three *internal* variables: a) $x^{\mathcal{M}}_{\text{trigger}}$ stores the latest input event that triggered the execution of the RTA module; b) $x^{\mathcal{M}}_{c}$ stores the current controller (initially *null*), and c) $x^{\mathcal{M}}_{d}$ counts the executions of the current controller;
3) $\phi : C \rightarrow 2^{X \rightarrow V}$ is a mapping from controllers to the set of variable valuations, i.e., *regions* of controllers, s.t. for all $c_i, c_j \in C$, if $c_i \neq c_j$, then $\phi(c_i) \cap \phi(c_j) = \emptyset$;
4) $\Delta : C \rightarrow \mathbb{N}$ is a mapping from controllers to natural numbers representing *decision frequencies*;
5) Triggers $\subseteq E$ is the set of input events triggering the execution of the RTA module.

The mapping $\phi(.)$ corresponds to the controller switching logic used in the DM of an RTA module for deciding the current controller, e.g., in Fig. 3, the controller switching logic (implemented by the `f_DM` function in lines 14–17) symbolically defines the controller regions $\phi(.)$. For a controller $c \in C$, we refer to $\phi(c)$ as the *region* of c, i.e., $\phi(c)$ indicates the set of valuations required for *starting to run* c. Here, we assume that the set of variables of an RTA module describes the state of the RTA module adequately. Therefore, we define regions for controllers over variable valuations. For a valuation v and a controller c, if v is the current valuation and $v \in \phi(c)$, then we say that the system is in $\phi(c)$. Decision frequencies define the number of executions for each controller before checking the current region again, i.e., if the system is in $\phi(c)$, then c is executed $\Delta(c)$ times before checking the current valuation to change the controller. Notice that a controller is *not* only executed in its region, i.e., controller regions are *not* simply the regions in which we only run the corresponding controller. They are rather regions in which we decide to run the corresponding controller c for the *next* $\Delta(c)$ executions. An RTA module is run only when an event $e \in$ Triggers is received, and it is not executed otherwise. The notion of receiving events is precisely defined in Section III-B.

For $\mathcal{M} = (C, X, \phi, \Delta, \text{Triggers})$, we define the following.

1) *Reach:* $((X \rightarrow V) \times ((X \rightarrow V) \rightarrow (X \rightarrow V)) \times \mathbb{N}) \rightarrow (X \rightarrow V)$, s.t. $\text{Reach}(v, c, d)$ is the variable valuation obtained *after* executing c for d times starting from v. Reach(.) is a well-defined function since we define controllers as *mappings* from valuations to valuations.
2) *ReachSet:* $((X \rightarrow V) \times ((X \rightarrow V) \rightarrow (X \rightarrow V)) \times \mathbb{N}) \rightarrow 2^{X \rightarrow V}$, s.t. $\text{ReachSet}(v, c, d) = \bigcup_{i=1}^{d} \{\text{Reach}(v, c, i)\}$ is the set of reachable variable valuations *within* d executions by executing c starting from v. As a shorthand notation, we define $\text{ReachSet}(\phi(c), c, d) = \bigcup_{v \in \phi(c)} \text{ReachSet}(v, c, d)$.
3) $\Phi = \bigcup_{c \in C} \phi(c)$ is the *safety invariant* of $\mathcal{M}$, it also refers to the *operating region* of $\mathcal{M}$ since the behavior of $\mathcal{M}$ is undefined outside Φ. For a valuation v, if $v \in \Phi$, then we say that v is *safe*, and if v is the current valuation of $\mathcal{M}$, then we say that the system is *safe*.

When it is not clear from the context, we will refer to components of an RTA module $\mathcal{M}$ as $C_{\mathcal{M}}$, $X_{\mathcal{M}}$, $\phi_{\mathcal{M}}$, $\Delta_{\mathcal{M}}$, and $\text{Triggers}_{\mathcal{M}}$, respectively, and its safety invariant as $\Phi_{\mathcal{M}}$. In order for an RTA module to keep the system safe, i.e., in its operating region, it needs to satisfy a set of properties. Next, we capture these properties by introducing the notion of a well-formed RTA module, and we explain how a well-formed RTA module always keeps the system safe.

Definition 2 (Well-Formed RTA Module): Let $C = C_T \cup C_U$ be a set of controllers consisting of a set of trusted (safe) controllers (TCs) C_T and a set of untrusted (advanced) controllers (UCs) C_U. An RTA Module $\mathcal{M} = (C, X, \phi, \Delta, \text{Triggers})$ is said to be well-formed if it satisfies the following properties.

(P1) (Safety) *TCs must be safe*. For all $c \in C_T$, $\text{ReachSet}(\phi'(c), c, *) \subseteq \Phi$, where $\phi'(c) = \phi(c) \cup \bigcup_{c' \in C_U} \phi(c')$ and $*$ indicates any natural number, i.e., executing a TC in its region or in the region of any UC keeps the system in its operating region Φ.

(P2) (Liveness) *If the system is recoverable, then the set of TCs must eventually recover the system.* If a UC region is reachable from the current state of the system, then there exists $n \in \mathbb{N}$ s.t. for $i = 1, 2, \ldots, n-1$, $c_i \in C_T$, $c_n \in C_U$, and $\text{Reach}(\phi(c_i), c_i, \Delta(c_i)) \in \phi(c_{i+1})$, i.e., running a finite sequence of TCs leads to a UC region.

(P3) *UCs must be executed with caution*. For all $c \in C_U$, $\text{ReachSet}(\phi(c), *, \Delta(c)) \subseteq \Phi$, where $*$ is *any controller*, i.e., the region and the decision frequency of a UC must be assigned s.t. the system stays in the operating region after executing any controller $\Delta(c)$ times.

See Section III-D3 for a detailed discussion of achieving presented well-formedness properties in practice.

So far, we have only formally defined the notion of an RTA module. Observe how this definition relates to the architecture of an RTA module presented in Fig. 1. The set of controllers C corresponds to the controllers listed in the figure, the set of local variables X represents the variables used by the controllers for execution, ϕ and Δ are the user-defined parameters controlling the execution semantics of the RTA module (notice that we have not yet defined the notion of execution for an RTA module and its DM, but we will precisely define it in the following paragraph), and Triggers is the set of events that trigger the execution of the RTA module.

Next, to prove the safety of a well-formed RTA module, we need to define the notion of execution and prove that running a well-formed RTA module on a safe valuation will keep the system safe. In other words, the resulting variable valuations will be in the operating region. Similar to a controller, we define an *action* $a : (X \rightarrow V) \rightarrow (X \rightarrow V)$ as a mapping. We differentiate between the two to be consistent with state machine and RTA terminologies. Algorithm 1 is the *RTA action* for an RTA module $\mathcal{M}$ defining the notion of execution. Given a valuation, the procedure defined by Algorithm 1 first checks the execution count of its current controller, and if it is time to select a new controller, then it checks the current region of the system (this step corresponds to executing the `f_DM` function from Fig. 3) to update the current controller and the execution count variable. Notice that since controller regions are disjoint, line 5 is only executed once. After deciding the current controller, the procedure executes this controller, and returns the resulting new valuation. Observe that an RTA action can be interpreted as a

Algorithm 1 RTA Action of an RTA Module $\mathcal{M}$

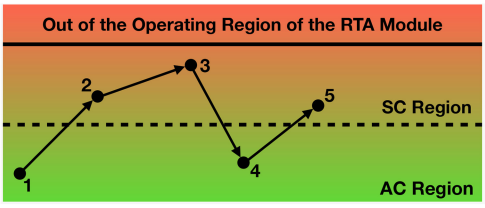
Require: $v \in X \to V$, where $X_{\mathcal{M}} \subseteq X$
Ensure: $v' \in X \to V$, where $X_{\mathcal{M}} \subseteq X$
1: $c, d \leftarrow v(x_c^{\mathcal{M}}), v(x_d^{\mathcal{M}})$ ▷ Save current controller and execution count
2: **if** $c = null \vee d \geq \Delta_{\mathcal{M}}(c)$ **then** ▷ Check if controller update is needed
3: **for each** $c_i \in C_{\mathcal{M}}$ **do** ▷ Iterate over controllers
4: **if** $v[X_{\mathcal{M}}] \in \phi_{\mathcal{M}}(c_i)$ **then** ▷ Check current region
5: $c, d \leftarrow c_i, 0$ ▷ Update current controller and reset counter
6: **break** ▷ Break since controller regions are disjoint
7: $v' \leftarrow (v \setminus v[X_{\mathcal{M}}]) \cup c(v[X_{\mathcal{M}}])$ ▷ Run controller and update valuation
8: **return** $v'[x_c^{\mathcal{M}}, x_d^{\mathcal{M}} \leftarrow c, d+1]$ ▷ Return the new valuation

supervisory controller for the controllers of an RTA module. For the set of RTA modules M and the set of actions A, $\alpha : M \to A$ maps RTA modules to RTA actions. As defined above, ReachSet$(v, \alpha(\mathcal{M}), d) \subseteq 2^{X \to V}$ is the reachable variable valuations *within* d executions starting from the valuation v running the RTA action $\alpha(\mathcal{M})$.

Theorem 1: For a well-formed RTA module $\mathcal{M}$ and an initial valuation of its variables v_0, where $v_0(x_c^{\mathcal{M}}) = null$, if $v_0 \in \Phi_{\mathcal{M}}$, then ReachSet$(v_0, \alpha(\mathcal{M}), *) \subseteq \Phi_{\mathcal{M}}$ holds.

Proof: (P1) and (P3) guarantee that for any $v \in \Phi_{\mathcal{M}}$, ReachSet$(v, c, \Delta_{\mathcal{M}}(c)) \subseteq \Phi_{\mathcal{M}}$ holds, where $c \in C_{\mathcal{M}}$ and $v \in \phi_{\mathcal{M}}(c)$, i.e., running the corresponding controller of a valuation will always keep the system safe. Algorithm 1 guarantees that $\alpha(\mathcal{M})$ will always update the current controller c after $\Delta_{\mathcal{M}}(c)$ executions with a new controller $c' \in C_{\mathcal{M}}$ s.t. $v' \in \phi_{\mathcal{M}}(c')$ for the new valuation v'. Therefore, $\mathcal{M}$ will always pick the corresponding controller for a valuation and that controller will always keep the system safe until the next controller decision of the RTA module. Thus, *ReachSet*$(v_0, \alpha(\mathcal{M}), *) \subseteq \Phi_{\mathcal{M}}$ holds for v_0 and $\mathcal{M}$. ■

Theorem 1 states that executing the RTA action of a well-formed RTA module from a valuation in its operating region will always lead to valuations in its operating region. Now that we precisely defined RTA modules, we conclude with an example clarifying the presented definitions for the reader.



Example 1: Consider a well-formed RTA module with an AC with a decision frequency of 4 and an SC with a decision frequency of 8. Let the figure above represent a small section of this RTA module's operating region, where the dotted line in the middle separates the controller regions, and the solid line bounds the operating region. The labeled points correspond to the executions of the RTA module. Let point 1 be a decision point for the RTA module (i.e., it is time to run the DM). Then upon receiving an event triggering the RTA module, at point 1, the RTA module checks the current region (e.g., runs the `f_DM` function from Fig. 3), sees that it is in AC's region, and then runs AC for the next four steps (i.e., until point 5).

In the given example, AC goes out of its own region, which is *perfectly* aligned with the definition of a well-formed RTA module, i.e., Definition 2. The crucial observation, here, is that the controller regions are regions in which we start running the corresponding controllers, and the decision frequencies are the values that tell us how many steps we need to run the controller before checking the system state again. However, it does *not* mean that ULGEN only executes controllers in their corresponding regions by immediately switching the controllers. The current controller can go into another controller's region before checking the system state again for a controller switch. This is a *safe* behavior, i.e., it satisfies the properties (P1) and (P2) given in Definition 2. (P1) defines the notion of safety for TCs, and (P3) bounds the controller regions and the decision frequencies of UCs so that they do not go out of the operating region. The crucial observation in this definition is that both (P1) and (P3) use Φ (which denotes the operating region of the RTA module, and it is the union of the RTA module's all controller regions), not $\phi(.)$ (which is a mapping defining the specific controller regions). These properties guarantee that a well-formed RTA module cannot go out of Φ, which means that even if a controller goes out of its own region, it will never leave the entire operating region, so it is, therefore, guaranteed that there is always a controller to select for the next controller switch. With constructs like controller regions and decision frequencies, we give more control to the programmer to define adequate controller behaviors for their applications.

B. Runtime Assured State Machines

Next, we define the integration of RTA modules with state machines by introducing RTA-SMs. We then formally define the notion of *well-formedness* for RTA-SMs, and prove that a well-formed RTA-SM will always keep the system in its operating region.

Definition 3 (RTA-SM): An RTA-SM is defined as a tuple $\mathcal{A} = (S, s_0, M, \rho, X, A, I, O, \delta)$, where:

1) S is the set of states;
2) $s_0 \in S$ is the initial state;
3) M is the set of RTA modules;
4) $\rho : S \to (M \cup \{null\})$ maps states to RTA modules;
5) X is the set of variables s.t. $\bigcup_{\mathcal{M} \in M} X_{\mathcal{M}} \subseteq X$;
6) $A \subseteq 2^{(X \to V)} \to (X \to V)$ is the set of actions;
7) $I \subseteq E$ is the set of input events;
8) $O \subseteq E$ is the set of output events;
9) $\delta : S \times (I \cup \{null\}) \to S \times (O \cup \{null\}) \times A$ is the transition function. $\delta(s, i) = (s', o, a)$ is a transition from s to s' receiving i, sending o, and executing action a.

Next, we define the well-formedness properties for RTA-SMs, which are the backbone of the proof of safety.

Definition 4 (Well-Formed RTA-SM): An RTA-SM $\mathcal{A} = (S, s_0, M, \rho, X, A, I, O, \delta)$ is said to be well-formed if it satisfies the following properties.

1) For all $\mathcal{M} \in M$, $\mathcal{M}$ must be a well-formed RTA module, and for all $\mathcal{M}_i, \mathcal{M}_j \in M$, $X_{\mathcal{M}_i} \cap X_{\mathcal{M}_j} = \emptyset$ or $\Phi_{\mathcal{M}_i} =$

$\Phi_{\mathcal{M}_j}$, i.e., if the sets of variables of two RTA modules intersect, they must have the same safety invariant.

2) For all $s_i, s_j \in S$, $\rho(s_i) = \rho(s_j)$ iff $\rho(s_i) = \rho(s_j) = null$, i.e., each state has a unique RTA module.
3) For any $\delta(s, i) = (s', o, a)$, we have the following.
   (D1) If $\rho(s) = null$ or $i \notin \text{Triggers}_{\rho(s)}$, then for all $x \in \bigcup_{\mathcal{M}\in M} X_{\mathcal{M}}$, and for all $v, v' \in (X \to V)$ s.t. $a(v) = v'$, $v(x) = v'(x)$ holds, i.e., v' is different than v only for variables that are not used by any RTA module.
   (D2) If $i \in \text{Triggers}_{\rho(s)}$, then we have $a = \alpha(\rho(s))$, i.e., if i is a triggering event of the RTA module of s, we run the corresponding RTA action of that RTA module.

Definition 4 enforces that RTA modules of a well-formed RTA-SM must have the same safety invariant if their variables intersect; thus, an RTA module cannot modify the variables of another RTA module for a different safety invariant. This is a crucial property to avoid unsafe behavior that might emerge from unnecessarily complex relationships between RTA modules of an RTA-SM. For example, consider the case of two RTA modules (of the same RTA-SM) with conflicting safety invariants on a set of variables. As the machine changes states and runs these RTA modules, actions of one would violate the safety invariant of the other, which would cause an unsafe behavior w.r.t. to the violated safety invariant. By disallowing such variable intersections, we make sure that such conflicts between RTA modules do not occur.

Before we define the execution semantics of an RTA-SM and prove the safety of its behavior, we highlight the correspondence between Definition 3 and Fig. 3. Observe that lines 9–32 of Fig. 3 is a programmatic description of an RTA-SM with an event-driven RTA module given in the ULGEN syntax. Specifically, the motion planner machine defined in lines 9–32 of the ULGEN program in Fig. 3 defines a set of states along with an initial state and dedicated RTA modules in certain states, declares a set of variables, and defines the transition function by describing the transition behavior in each state using input/output events and actions executed on the transition. That machine has only one RTA module; therefore, with cautious programming, it trivially satisfies the conditions over variables given in Definition 4, i.e., the first bullet point and (D1). The other conditions, i.e., the second bullet point and (D2), are enforced by the semantics of the language and the type checker, see Section IV for a detailed discussion of the ULGEN language and the toolkit.

Next, we continue with the execution semantics of an RTA-SM by defining the transition system for an RTA-SM, and we use this definition to prove the safety of its execution.

Definition 5 (RTA-SM Transition System): A TS for $\mathcal{A} = (S, s_0, M, \rho, X, A, I, O, \delta)$ is $\mathcal{T}(\mathcal{A}) = (Q, q_0, \to)$.

1) Q: $S \times (X \to V) \times (\mathbb{N} \to I)$, i.e., a state consists of a state of $\mathcal{A}$, a valuation of X, and an event queue for I.
2) $q_0 = (s_0, v_0, f_{\text{empty}})$ is the initial state, i.e., s_0 is the initial state of $\mathcal{A}$, v_0 is the initial valuation for X, where variables are initialized with their default values and $v_0(x_c^{\rho(s_0)}) = null$, and f_{empty} is the empty queue.
3) $\to$: $(Q \times (I \cup \{null\}) \to Q) \cup (Q \to (O \cup \{null\}) \times Q)$ is the transition function. We have three kinds of transitions.

Null $(s, v, f) \xrightarrow{o} (s', v', f)$ if there exists a transition $\delta(s, null) = (s', o, a)$ with $a(v) = v'$ and $v'(x_c^{\rho(s')}) = null$, i.e., a null transition not dequeuing any event.

EnQ $(s, v, (i_0, \ldots, i_n)) \xrightarrow{i} (s, v, (i_0, \ldots, i_{k-1}, i, i_k, \ldots, i_n))$, where $i \neq null$, for all $i' \in \{i_0, \ldots, i_{k-1}\}$, $p(i') \geq p(i)$, and for all $i'' \in \{i_k, \ldots, i_n\}$, $p(i'') < p(i)$, i.e., inputs are stored with respect to their priorities. We assume this transition type has a higher priority than others.

DeQ $(s, v, (i_0, i_1, \ldots, i_n)) \xrightarrow{o} (s', v', (i_1, \ldots, i_n))$ if there exists $\delta(s, i_0) = (s', o, a)$ with $i_0 \neq null$ and $a(v) = v'$. If $s \neq s'$, then $v'(x_c^{\rho(s')}) = null$.

A *run* of $\mathcal{A}$ is a sequence of states defined over $\mathcal{T}(\mathcal{A})$ as follows:

$$q_0 \xrightarrow{e_0} q_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} q_n \xrightarrow{e_n} \cdots$$

where $q_i \in Q$ and $e_i \in I \cup \{null\} \cup O$. Note that an input event can occur at any step of a run since it depends on the environment. The *operating region* of $\mathcal{A}$ is $\Phi_{\mathcal{A}} = \bigcup_{\mathcal{M}\in M} \Phi_{\mathcal{M}}$.

Theorem 2: For all runs of a well-formed RTA-SM $\mathcal{A}$, if $v_0 \in \Phi_{\mathcal{A}}$ holds for the initial state $(s_0, v_0, \emptyset)$, then for all states (s_i, v_i, f_i) appearing in a run, $v_i \in \Phi_{\mathcal{A}}$ holds.

Proof: **EnQ** do not change variable valuations by Definition 5, i.e., an enqueue transition will always keep the system in the operating region if it is taken from a safe valuation. Now, we consider other transition types, i.e., **Null** and **DeQ**. By (D1) in Definition 4, taking a transition on a *null* event (notice that a *null* event cannot trigger the execution of an RTA module by Definition 1), taking a transition in a state without an RTA module, and taking a transition on an event that does not trigger the RTA module of a state does not change the valuation of any variable used by an RTA module of $\mathcal{A}$. Hence, the safety invariant will be satisfied after taking such a transition if it is taken from a safe valuation. By Algorithm 1 and Theorem 1, running an RTA module starting from a safe valuation will keep the system safe, i.e., since Algorithm 1 only updates variables used by the RTA module of the state, and Theorem 1 guarantees that these modifications will keep the system in the operating region, we can conclude that running an RTA module will keep the system safe. The well-formedness properties for an RTA-SM in Definition 4 guarantees that this argument applies to RTA-SMs that have RTA modules with both the same safety invariant and different safety invariants. Thus, $\mathcal{A}$ will always satisfy its safety invariant starting from a safe valuation. ■

Theorem 2 states that if an RTA-SM is initially in its operating region, then it will always satisfy the safety invariant.

C. Compositional Runtime Assured State Machine Systems

Next, we extend our formalism to ULGEN programs. A *compositional RTA-SM system* (C-RTA-SM-S) $\mathcal{S} = \{\mathcal{A}_0, \ldots, \mathcal{A}_n\}$, which defines a ULGEN program, is the asynchronous composition of the set of RTA-SMs $\{\mathcal{A}_0, \ldots, \mathcal{A}_n\}$ under interleaving semantics, see [25] for the details of this composition. A C-RTA-SM-S (or a ULGEN program) $\mathcal{S}$ is said to be *well-formed* if for all $\mathcal{A}_i \in \mathcal{S}$, $\mathcal{A}_i$ is well-formed, and for all $\mathcal{A}_j \in \mathcal{S}$, if $\mathcal{A}_i \neq \mathcal{A}_j$, then $X_{\mathcal{A}_i} \cap X_{\mathcal{A}_j} = \emptyset$.

Observe that a C-RTA-SM-S (a ULGEN program) is simply a collection of RTA-SMs communicating through asynchronous events. Figs. 2 and 7 are high-level representations of such programs used in the case studies. In these figures, each box corresponds to an RTA-SM and arrows between the boxes present high-level descriptions of the events sent between the machines.

Theorem 3: For a well-formed ULGEN program $\mathcal{S}$, and for all $\mathcal{A}_i \in \mathcal{S}$, if $v_0^i \in \Phi_{\mathcal{A}_i}$ for the initial state $(s_0^i, v_0^i, f_{\text{empty}})$ of $\mathcal{A}_i$, then we have $v_j^i \in \Phi_{\mathcal{A}_i}$ for all (s_j^i, v_j^i, f_j^i) of $\mathcal{A}_i$.

Proof: Since RTA-SMs in $\mathcal{S}$ do not share any variables, they cannot change the truth value of another RTA-SM's safety invariant. Thus, by this observation and the semantics of asynchronous interleaving composition, each machine runs in isolation. Then by Theorem 2, we conclude that the overall system stays safe provided that it starts from a safe state. ■

Theorem 3 states that decomposing the overall safety requirements of a system into invariants and implementing a well-formed ULGEN program guarantees safety.

D. Remarks on Formalism and Well-Formedness

1) Time-Driven RTA Modules: So far, we have only considered the event-driven execution semantics of an RTA module. For the time-driven case, one can formalize an RTA module by having a dedicated triggering event instead of a set of events. Then a timer machine can be defined as a *timed automaton* [2], [3] to send timed events for triggering the time-driven RTA module. We can interpret such events as environment inputs; thus, the formalism trivially extends to the time-driven case.

2) Connections to the ULGEN *Language:* We remark on the correspondence between a ULGEN program and the presented formalism using the motivating example from Section II. The collection of ULGEN machines from Fig. 2, i.e., the ULGEN program, defines a C-RTA-SM-S, where each machine corresponds to an RTA-SM. Observe the one-to-one correspondence between Definition 1 and the RTA module declarations in Fig. 3. Just like the formal definition in Definition 1, RTA modules in ULGEN programs declare a set of controllers with decision frequencies along with a set of triggering events, operate on a subset of the local variables of their parent machines, and their DMs *logically* define controller regions. Thus, ULGEN is a native language for programming systems with the framework formalized in this section.

3) Well-Formedness in Practice: The correspondence between the formalism and the ULGEN language eases the programming; however, one must satisfy the well-formedness properties given in Definitions 2 and 4. Satisfying Definition 4 for an RTA-SM is somewhat trivial, i.e., it defines a unique RTA module in each state, and makes sure that either different RTA modules of an RTA-SM have the same safety invariant, or they have different safety invariants and they do not modify each other's local variables. Observe that this outlines a desirable and easy-to-follow modular programming practice. Furthermore, the semantics of the ULGEN language and type checks performed by the compiler guarantee the second bullet point and (D2) in Definition 4 whereas the other conditions [first bullet point and (D1)] can be easily satisfied by following a principled programming practice as mentioned. In Definition 2, (P1) and (P2) can be achieved by using reachability analysis [15], [22], [29] and controller synthesis frameworks [5], [18], [24], [35], respectively. To achieve well-formedness, we have to also decide on decision frequencies and regions for UCs to satisfy (P3). Synthesizing these for a UC, a controller that, in general, we cannot make any assumptions on besides a reasonable boundedness assumption is a challenging problem and outside the scope of this work. Still, we can provide a practical guideline for well-formedness. The decision frequency and the region of a UC define the tradeoff between performance and safety. As an empirical approach, one can start from the most conservative values, i.e., one for the decision frequency and a restricted region, and then iteratively relax them based on the requirements. Notice that the ULGEN language provides an easy interface for configuring these parameters. Specifically, the decision frequency parameter can be easily tuned by changing the given mapping in the program, e.g., line 28 of Fig. 3, and since the controller regions are defined symbolically, e.g., lines 14–17 of Fig. 3, one can relax and restrict the controller regions by modifying the constraints of this symbolic definition.

IV. ULGEN Programming Language and Toolkit[2]

We built the ULGEN toolkit by extending the *P* language framework. The toolkit has three components: 1) a compiler for compiling high-level ULGEN programs to executable C; 2) a C runtime for executing the ULGEN programs; and 3) a library for the Reelay monitoring tool [41] which can be used for the controller switching logic. This library provides functions for creating discrete and dense timed Reelay monitors, logging data to a monitor, and checking the specification.

We extended the C runtime of *P* to support the following features: 1) event priorities; 2) language primitives for RTA modules; and 3) time-driven semantics implemented using OS timers. To declare an RTA module, the programmer implements controllers and the controller switching logic as local functions of the state machine, which can be entirely implemented in the program or can call foreign C/C++ functions. The compiler guarantees that the well-formedness properties from Definition 4 are satisfied except for the first bullet point and (D1) which must be accomplished by the programmer. However, neither the compiler nor the runtime imposes the well-formedness properties of Definition 2. It is the programmer's responsibility to ensure that these properties are satisfied by following the guidelines presented in Section III-D3.

Our extensions to *P* are both syntactic as well as semantic as we added new language features. Specifically, time-driven execution semantics and event priorities are new features and cannot be expressed in the standard *P* language. For event-driven execution semantics of RTA modules, we made changes to the language semantics to assure safety. In particular, the standard *P* runtime defines dequeuing and event handling as separate atomic operations. However, for safety, we modified this behavior for RTA modules by defining their executions as atomic w.r.t. other machines, i.e., in ULGEN, dequeuing,

[2]Toolkit is available at https://github.com/BerkeleyLearnVerify/ULGEN.

event handling, and controller execution are all a single atomic operation which cannot be interleaved by other operations.

P offers a systematic testing engine for checking the correctness of *P* programs. Currently, our toolkit does not support systematic testing of ULGEN programs. We consider systematically testing a program with real-time behavior as an interesting problem on its own and leave it for future work.

Next, we note potential threats of our extensions to *P*.

1) Our backend uses OS timers for time-driven semantics, which is not reliable for hard real-time systems whereas it is performant enough for soft real-time systems as we demonstrate in the following sections.
2) Event priorities enable task prioritization, but one should note that it may expand the space of program behaviors.

Atomic execution of RTA modules is a restriction over standard *P* semantics, so it does not introduce any threats and reduces the space of program behaviors.

P is shown to be a low overhead and scalable language in various domains [7], [8], [10], [39]. Therefore, our toolkit is also low overhead, and it is scalable to realistic applications as we demonstrate in the case studies. Note that as our framework combines different controllers and DMs, its scalability is mainly limited by the scalability of these components.

V. EMPIRICAL EVALUATION: COMPARISON WITH SOTER

We present an empirical comparison of ULGEN with SOTER [7]. Before starting, notice that SOTER lacks most of the highlighted features of ULGEN: no event priorities, no multiple controllers, no event-driven RTA modules, just one AC, one SC, and time-driven RTA modules. Moreover, it uses standard *P*; thus, the time-driven behavior is implemented using explicit busy waits. Most importantly though, SOTER defines controllers and the DM of an RTA module as periodically executing nodes, and regardless of the current state of the system, both controllers are executed, but the DM enables/disables the outputs of the controllers through a globally shared data structure implemented at C level.

A. Case Study: Simple Robot Braking System

We consider a two-wheeled robot tasked to move forward and stop at a specific position. This system can be implemented with only one RTA module, where the AC moves forward, the SC stops the robot, and the DM monitors the position for a controller switch. We implement the system in ULGEN and SOTER, and we use the Webots [28], [42] simulator.

1) Evaluation: We answer the following question: "Do both frameworks assure safety?," where safety is defined as braking on time. The task only depends on the behavior of the frameworks and the environment dynamics, so we run 100 simulations with different random seeds, and monitor the position to detect safety violations. As *P* provides multithreaded execution by assigning jobs to worker threads, we run the experiments using both a single thread and two threads to understand and compare concurrent behaviors of frameworks.

Both single-threaded programs satisfy the specification. The two-threaded ULGEN program also satisfies the specification in all runs. However, the two-threaded SOTER program violates the specification in 82% of 100 simulations. This unsafe behavior of SOTER is due to the nondeterministic scheduling of threads, and the fact that the controllers and the DM are all asynchronously executing nodes, therefore, prone to ordering issues. Moreover, the formalism of SOTER does not define an ordering between the executions of the controllers and the DM when they ought to run at the same time. Thus, the unsafe behavior of SOTER in a multithreaded setting was expected both from a theoretical perspective and from a practical one. Crashes due to scheduling issues were also reported in the experiments of the SOTER work [7]. In ULGEN, such scheduling issues are irrelevant as we impose a well-defined execution order over components of an RTA module.

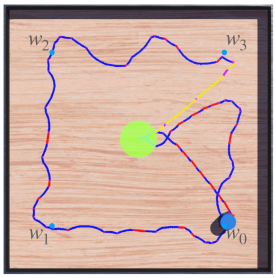


Fig. 4. Webots environment for Section VI-A.

We also compare SOTER and ULGEN programs by using the number of lines, a rough but arguably one of the few quantifiable metrics for program simplicity. The SOTER program consists of 100 lines whereas the ULGEN program only has 50. As we built most of our features into the language, the ULGEN program ends up being more readable and shorter. Finally, we report that the automatically generated C code from the ULGEN program has 1000 lines, which only has the C representations of the constructs defined in the program, and it is executed with the given semantics by our backend.

VI. EMPIRICAL EVALUATION: IN SIMULATION SETUP[3]

In this section, we present two case studies to evaluate the safety and performance of ULGEN programs.

A. Case Study: Robot Surveillance System

We investigate the motivating example from Section II-A in Webots [28], [42] and empirically evaluate the safety and performance of its ULGEN implementation using VerifAI [12], [13], a toolkit for the formal design and analysis of AI/ML systems. The task is to navigate a two-wheeled robot to visit waypoints w_0, w_1, w_2, and w_3, shown in Fig. 4, infinitely often while maintaining an adequate battery level by visiting the station (green area in Fig. 4) when the battery is critically low. We implement the case study in ULGEN using the design given in Fig. 2, where each concurrent component is implemented using an RTA-SM with either event-driven or time-driven RTA modules for assuring their specifications. To evaluate the safety, we introduce faulty behavior by implementing the UC of the plan executor as a controller which, regardless of the input, turns left with probability 20% or

[3]Complete source code, documentation for implementation details, and videos are available at https://github.com/BerkeleyLearnVerify/ULGEN.

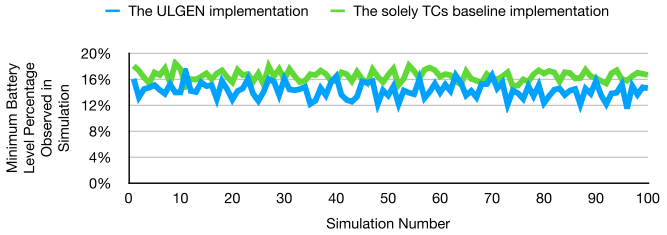


Fig. 5. Minimum battery level percentage observed in simulations versus the number of simulations. In this example, the robot executes the procedure for going to the station when the battery level drops below 20%.

moves forward with probability 80%. We refer to this specific controller as the *random controller* in the rest of this article. The TC is a controller designed for closely following an ideal trajectory between waypoints. To assure safety, the plan executor implements two states running time-driven RTA modules: 1) `LowBatteryRun` and 2) `Run`, the former defines a more conservative decision frequency and controller region for its UC. We use Reelay monitors for the controller switching logic. For details, see the ULGEN repository.

1) Evaluation: We answer:

(Q1) *Safety:* "Does ULGEN assure safety?"

(Q2) *Performance:* "Does ULGEN maximize the usage of UCs?"

We use the MTL falsifier of VerifAI with a cross-entropy sampler. The falsifier searches for a configuration that fails the mission due to a low battery level by sampling configurations with different initial locations for the robot and the charger. We ran the falsifier for 100 simulations with 150 000 simulation steps (> 5 h of systematic testing). Our baseline executes solely TCs.

a) (Q1) Safety: In all 100 simulations, the robot successfully performed surveillance. Fig. 5 presents the minimum battery level observed in simulations generated by VerifAI. The blue line is the ULGEN implementation and the green line is the baseline. The baseline keeps a higher battery level in general since ULGEN tries to execute the UCs whenever possible. Specifically, since the UC of the plan executor is the random controller, the robot loses the battery while moving to the charger. The falsifier finds cases with slightly lower battery levels, but it cannot falsify the system. Thus, our evaluation shows that ULGEN can be used to program safe systems.

b) (Q2) Performance: We measure the performance by analyzing the executions of the plan executor. We report the average execution percentages over 100 simulations for the RTA modules of `LowBatteryRun` and `Run` states. In `LowBatteryRun` and `Run`, the UCs execute 16% and 46% of all executions, respectively. This difference between the execution percentages was expected as `LowBatteryRun` is more conservative than `Run`. Fig. 4 presents the trajectory of the robot after one tour in the default environment. Blue and red represent executions of the UC and the TC of `Run`, respectively. Purple and yellow represent executions of the UC and the TC of `LowBatteryRun`, respectively. Fig. 4 shows as if the UC of `Run` runs more often than its TC as the TC is mostly used for fixing the orientation, and the UC is mostly used for moving forward. Even with the random controller, ULGEN maximizes the usage of UCs while assuring safety.

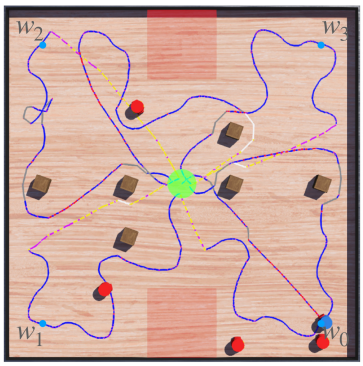


Fig. 6. Webots environment for Section VI-B.

B. Case Study: More Complex Robot Surveillance System

We present a more complex robot surveillance case study, where five robots of two different types are present in an environment (an ego robot and four secondary robots) with static obstacles and two geo-fenced regions. The secondary robots are tasked to randomly move in the environment while avoiding collisions. The task of the ego robot is to perform surveillance while satisfying several constraints: 1) waypoints w_0, w_1, w_2, and w_3 must be visited in the given order; 2) motion plans must consist of waypoints inside the environment; 3) motion plans must avoid geo-fenced regions; 4) motion plans must avoid static obstacles; 5) robot must closely follow a trajectory between waypoints; 6) robot must avoid collisions; 7) battery level must always be greater than zero; and 8) position/orientation estimation cannot deviate from the GPS/compass readings for more than a threshold. Fig. 6 is the environment, where the blue robot is the ego robot, red robots are secondary robots, brown boxes are obstacles, and red regions are geo-fenced. Only geo-fenced regions are known to the ego robot a priori. Fig. 7 presents the high-level design of the ego robot, where each node represents a concurrent component implemented as an RTA-SM with either event-driven or time-driven RTA modules to satisfy its safety specification. To emphasize safety, we use the random controller as the UC of the collision-aware plan executor machine, which turns left or moves forward with probabilities 20% and 80%, respectively. This machine also defines two TCs: 1) a safe trajectory controller (designed for closely following an ideal trajectory between waypoints) and 2) a collision avoidance controller. To assure safety when the battery is low, the collision-aware plan executor implements two states with time-driven RTA modules: 1) `LowBatteryRun` and 2) `Run`, where the former is more conservative. This case study is built on top of the motivating example, and as before, Reelay monitors are used for the controller switching logic. We refer the reader to the ULGEN repository for further details.

1) Evaluation: We conduct a similar evaluation to Section VI-A to answer (Q1) *safety* and (Q2) *performance* questions. We use the MTL falsifier of VerifAI with a cross-entropy sampler, which tries to falsify a specification stating that there must always be enough battery and the robot must always avoid geo-fenced regions. The falsifier searches for a configuration to fail the mission due to a low battery level or a violation of the geo-fences by sampling configurations

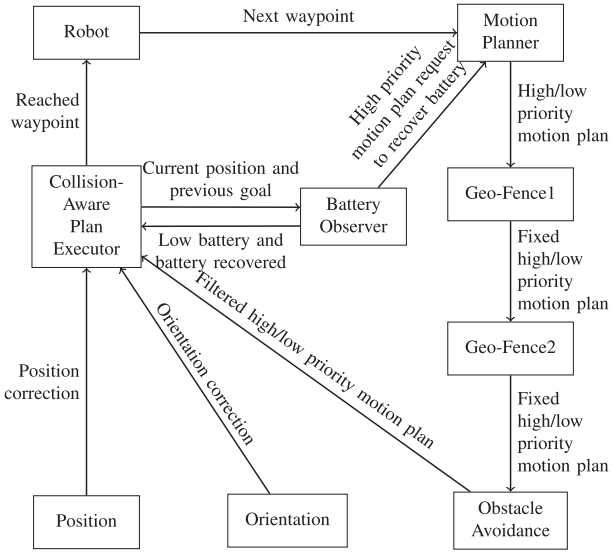


Fig. 7. ULGEN Program for Section VI-B, boxes are RTA-SMs.

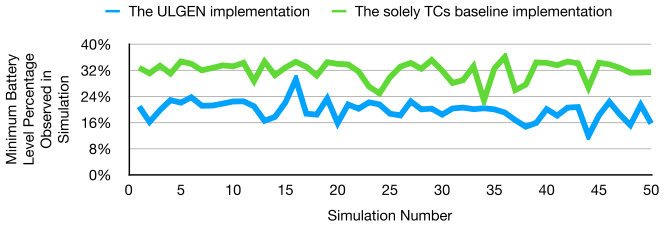


Fig. 8. Minimum battery level percentage observed in simulations versus the number of simulations. In this example, since the environment is larger than the previous example, and there are obstacles and other robots present in the environment, we increase the threshold for executing the battery recovery procedure, i.e., it is executed when the battery level drops below 40%.

with different initial locations for the ego robot, the charger, the secondary robots, and the boxes. We ran the falsifier for 50 simulations with 300 000 steps (> 7 h of systematic testing). Our baseline is an implementation executing solely TCs.

a) (Q1) Safety: In all 50 simulations, the robot successfully performed surveillance by keeping an adequate battery level and avoiding geo-fenced regions. Fig. 8 presents the minimum battery level observed in simulations. The blue line is the ULGEN implementation and the green line is the baseline. We observe a similar pattern to the evaluation given in Section VI-A, i.e., the baseline keeps a higher battery level in general since the ULGEN implementation tries to execute its UCs. Although the falsifier finds configurations where slightly lower battery levels are observed, it cannot falsify the system. Thus, our evaluation of a more complex system also shows that ULGEN can be used to program safe systems.

b) (Q2) Performance: We use the metric from Section VI-A for evaluating the performance by analyzing executions of the collision-aware plan executor. We report the mean of 50 simulations. In `LowBatteryRun`, the UC executes 42%, the safe trajectory controller executes 47%, and the collision avoidance controller executes 11% of all executions. In `Run`, the UC executes 45%, the safe trajectory controller executes 47%, and the collision avoidance controller executes 8% of all executions. Fig. 6 is the trajectory of the ego after one tour in the default environment. Different colors of the trajectory line indicate different controllers of the collision-aware plan executor. Purple, yellow, and white represent executions of the random, the safe trajectory, and the collision avoidance controllers of `LowBatteryRun`, respectively. Blue, red, and gray represent executions of the random, the safe trajectory, and the collision avoidance controllers of `Run`, respectively. We observe similar patterns in Fig. 6 and other simulations since TCs are used for recovery and UCs are run otherwise. These results show that even with the random controller, ULGEN maximizes the usage of UCs while assuring safety.

VII. Empirical Evaluation: On Robotic Platform[4]

We present two case studies to demonstrate an instantiation of ULGEN programs on a real robotic platform and to evaluate the safety and performance provided by our framework.

A. Case Study: Robot Surveillance System With Kobuki

We present a surveillance system on the iClebo Kobuki[5] robot base controlled with a Raspberry Pi 3 Model B+ running a 64-bit Ubuntu Server 20.04 LTS. We extend the ULGEN program presented in Section VI-A. The design is the same as Fig. 2 except that we do not have a battery observer as it is redundant for this mission. The UC of the plan executor's RTA module is a PID controller with empirically tuned parameters for performance, but its TC is the same as before. We used our own monitors for this case study since ULGEN *is agnostic to the selection of the monitoring tool*. We implemented a layer of helper functions to control the robot using the standalone Kobuki Core.[6] For details, see the ULGEN repository.

We assume the environment is safe, i.e., no obstacles. The task is to visit a sequence of waypoints. While performing the task, the robot must: 1) visit surveillance locations in the correct order; 2) compute safe motion plans between locations; and 3) follow the ideal trajectory without deviating more than 40 cm. The robot also has to optimize a performance objective: *minimize the average lap time throughout the mission.*

1) Evaluation: Similar to Section VI, we answer (Q1) *safety* and (Q2) *performance* questions. We run the ULGEN program on a Kobuki robot and record the mission details. We restrict the surveillance task to three tours in the environment, i.e., the robot starts from the station, makes three tours, and returns back to the station. Our baselines are implementations of the system with *solely TCs* and *solely UCs*.

a) (Q1) Safety: We focus on the specification of the plan executor, i.e., the robot must follow the ideal trajectory without deviating more than 40 cm. Fig. 9 presents the trajectories of the robot for all three setups. Both the solely TCs baseline and the ULGEN program satisfy the specification whereas the solely UCs baseline enters the unsafe red region. Due to the specific values of the decision frequencies tuned for performance, the DM does not immediately switch

[4]Complete source code, documentation for implementation details, and videos are available at https://github.com/BerkeleyLearnVerify/ULGEN.

[5]iClebo Kobuki's functional, hardware, and software details are available at http://kobuki.yujinrobot.com/about2/.

[6]Kobuki core is available at https://github.com/kobuki-base/kobuki_core.

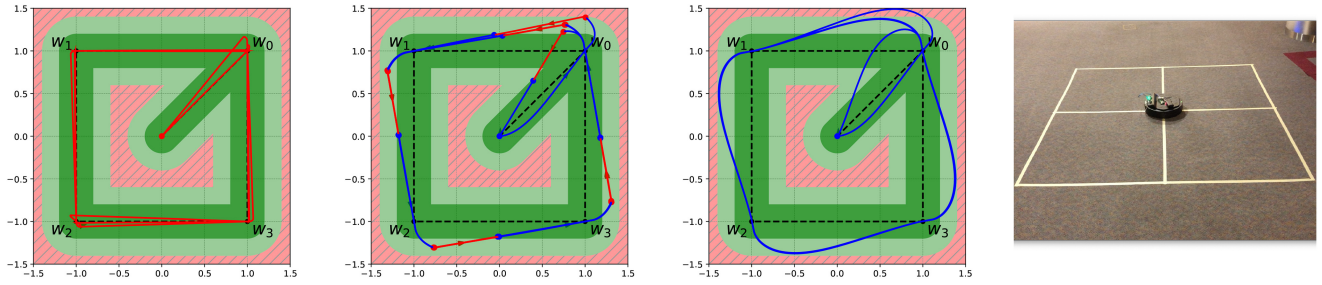


Fig. 9. Trajectories of the robot within three tours in the environment. The dashed black lines between waypoints represent the ideal trajectory. The red and blue lines indicate the execution of the TC and the UC of the plan executor, respectively. The light and dark green regions indicate the regions of the TC and the UC of the plan executor, respectively, and the red-striped regions indicate regions that violate the safety specification, i.e., deviating more than 40 cm. (a) Solely TCs (SCs). (b) ULGEN. (c) Solely UCs (ACs). (d) Environment.

TABLE I
MISSION TIMES FOR SECTION VII-A

	Lap 1 Time (s)	Lap 2 Time (s)	Lap 3 Time (s)	Mean Lap Time (s)	Mission Time (s)
Solely TCs	150	172	138	153	543
ULGEN	**95**	**91**	**91**	**92**	**310**
Solely UCs	50	49	50	50	171

TABLE II
MISSION TIMES FOR SECTION VII-B

	Exploration Time (s)	Surveillance Time (s)	Mean Lap Time (s)	Mission Time (s)
Solely TCs	181	185	183	431
ULGEN	**193**	**134**	**164**	**378**
Solely UCs	57	–	–	–

to TC in the light green region, but it switches before leaving the light green region. See the example given in Section III-A for further details about controller regions and decision frequencies.

b) (Q2) Performance: We measure and compare the lap times and the overall mission times for solely TCs, ULGEN, and solely UCs implementations to evaluate the performance. Table I reports the results. ULGEN provides a middle ground between the two extremes. The solely TCs baseline fails to optimize the performance, and the solely UCs baseline violates the specification whereas ULGEN optimizes the performance objective while assuring that the specifications are satisfied.

B. Case Study: Robot Exploration System With Kobuki

In this case study, we defer the safety assumption on the environment, i.e., we extend the mission to perform surveillance with unknown obstacles in the environment. This modification about our environment assumptions requires us to first perform an exploration tour to assure safety. We combine and improve the implementation of the surveillance systems presented in Sections VI-B and VII-A to accommodate the safe exploration of the environment. We use the same hardware and helper libraries as the ones presented in Section VII-A. For further implementation details, see the ULGEN repository.

The task of the robot is to safely explore and perform surveillance in an environment with unknown obstacles and a geo-fenced region. To complete the mission, the robot must: 1) visit surveillance locations in the correct order; 2) compute safe motion plans between locations; 3) obey geo-fencing constraints; 4) avoid obstacles; and 5) follow the ideal trajectory between waypoints without deviating for more than 30 cm while moving toward a waypoint, i.e., it is okay to enter the red region while avoiding a bumped obstacle. The robot should only execute its trusted controllers for the first tour to safely explore the environment. As the robot explores the environment, it must send the location of the identified static obstacles to a server through a TCP connection. The performance objective is to *minimize the average lap time*.

1) Evaluation: We answer two questions: (Q1) *safety* and (Q2) *performance*. We run the ULGEN implementation on a Kobuki robot and record the details. In our experiments, we restrict the task to two tours in the environment, i.e., the robot starts from the station, explores the environment by only executing its trusted controllers in the first tour, makes another tour for surveillance, and returns back to the station. Our baselines are implementations with *solely TCs* and *solely UCs*.

a) (Q1) Safety: We again investigate the specification of the plan executor. Fig. 10 presents the trajectories of the robot for all three setups. The solely TCs baseline and the ULGEN implementation keep the deviation from the ideal trajectory below the given threshold and complete the mission successfully. Due to the specific values of the decision frequencies tuned for performance, the DM does not immediately switch to the safe trajectory controller in the light green region, but it switches somewhere in that region. See the example given in Section III-A for further details about why we do not need an immediate switch between regions. The solely UCs baseline violates the specification by entering the red region several times. In the first lap, the solely UCs baseline deviates so much that the robot does not collide with the obstacle. In the second lap, it collides with the obstacle and fails.

b) (Q2) Performance: We measure and compare exploration, surveillance, and overall mission times. Table II reports the results. As we defer the safety assumption on the environment, in the exploration lap, the ULGEN implementation only executes its TCs. Therefore, there is no performance gain compared to solely TCs. As the solely UCs baseline collides with the obstacle and fails in the second lap, we only report its time for the first lap. ULGEN executes its UCs whenever the robot is in a *known* safe region; therefore, ULGEN optimizes the performance objective better than the solely TCs baseline.

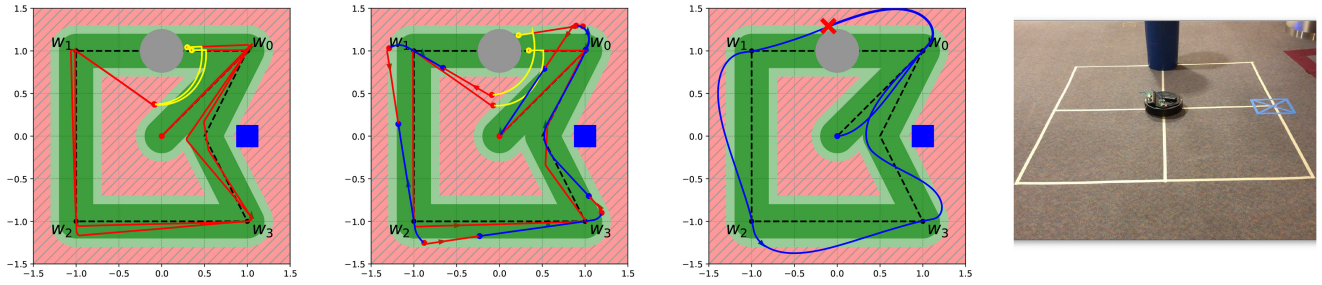


Fig. 10. Trajectories of the robot within two tours in the environment, i.e., exploration lap and surveillance lap. The dashed black lines between waypoints represent the ideal trajectory. The red, yellow, and blue lines indicate the execution of the safety trajectory controller, the obstacle avoidance controller, and the UC of the plan executor, respectively. The light and dark green regions indicate the region of the safe trajectory controller and the UC of the plan executor, respectively, and the red-striped regions indicate regions that violate the safety specification, i.e., deviating more than 30 cm from the ideal trajectory. The blue box at $(1, 0)$ is a geo-fenced region that is known a priori, and the gray circle at $(0, 1)$ is a static obstacle that is not known a priori by the robot. (a) Solely TCs (SCs). (b) ULGEN. (c) Solely UCs (ACs). (d) Environment.

VIII. Related Work

Various approaches have been proposed for the runtime monitoring and safe execution of safety-critical systems. Runtime verification techniques were used to monitor the safety of robotic systems in [6], [11], [19], [20], [26], [27], and [31]. Schierman et el. [36] examined the application of an RTA framework to the software stack of an unmanned aircraft system. Phan et al. [32] introduced a component-based Simplex architecture with assume-guarantee contracts for component-based CPS. The Simplex architecture has also been applied to sandboxing CPS [4], where automatic reachability-based approaches were presented for the controller regions. Akametalu et al. [1] proposed a reachability-based controller switching mechanism for reinforcement learning. ModelPlex [30] combines offline manual theorem proving-based verification with runtime validation of system executions [30]. Building on ModelPlex, Fulton and Platzer [17] developed an approach for provably safe learning of control strategies based on the differences between the reality and the offline verification model. The Neural Simplex architecture [33] is an RTA framework for neural controllers, which enables retraining of the AC after a switch to the SC. Desai et al. [7] introduced SOTER, a framework for programming safe robotic systems with RTA modules and bidirectional switching between SC and AC.

This article is complementary to much of this literature—we provide a general programming framework for RTA, wherein, for example, switching logic devised using an approach like ModelPlex could be implemented in ULGEN, and any state-of-the-art runtime monitoring tool could be used with our approach. The closest related work is SOTER [7], which only defines time-driven RTA modules with two controllers, and does not provide a publicly available tool. ULGEN defines event-driven and time-driven RTA modules with multiple controllers, provides priority-based communication, guarantees safety under formal well-formedness properties, enables the integration of state-of-the-art monitoring tools, and offers a ready-to-use toolkit. All in all, ULGEN provides a general RTA framework that can accommodate various state-of-the-art approaches and tools for programming safe CPS, and to the best of our knowledge, it is the first to offer such flexibility.

IX. Conclusion

We presented ULGEN, a framework for programming safe CPS. ULGEN defines event-driven and time-driven RTA modules with multiple controllers as building blocks for asynchronous processes along with a priority-based communication mechanism for task prioritization. We defined formal well-formedness properties guaranteeing a composition of SCs and ACs in a way that assures the underlying safety specifications provided by the SCs while delivering the desired performance offered by the ACs. We demonstrated the efficacy of ULGEN with five case studies both in a simulation setup and on a robotic platform. In the case studies, our evaluations show that ULGEN delivers the desired performance while assuring safety.

References

[1] A. K. Akametalu, J. F. Fisac, J. H. Gillula, S. Kaynama, M. N. Zeilinger, and C. J. Tomlin, "Reachability-based safe learning with Gaussian processes," in *Proc. 53rd IEEE Conf. Decis. Control*, 2014, pp. 1424–1431.

[2] R. Alur, "Timed automata," in *Proc. Int. Conf. Comput. Aided Verification*, 1999, pp. 8–22.

[3] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[4] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, "Sandboxing controllers for cyber-physical systems," in *Proc. IEEE/ACM 2nd Int. Conf. Cyber-Phys. Syst.*, 2011, pp. 3–12.

[5] B. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer, "Veriphy: Verified controller executables from verified cyber-physical system models," in *Proc. 39th ACM SIGPLAN Conf. Programming Lang. Design Implement.*, 2018, pp. 617–630.

[6] A. Desai, T. Dreossi, and S. A. Seshia, "Combining model checking and runtime verification for safe robotics," in *Proc. Int. Conf. Runtime Verification*, 2017, pp. 172–189.

[7] A. Desai, S. Ghosh, S. A. Seshia, N. Shankar, and A. Tiwari, "SOTER: A runtime assurance framework for programming safe robotics systems," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, 2019, pp. 138–150.

[8] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 321–332, 2013.

[9] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia, "Compositional programming and testing of dynamic distributed systems," in *Proc. ACM Program. Lang.*, 2018, pp. 1–30.

[10] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia, "Drona: A framework for safe distributed mobile robotics," in *Proc. 8th Int. Conf. Cyber-Phys. Syst.*, 2017, pp. 239–248.

[11] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods Syst. Design*, vol. 51, no. 1, pp. 5–30, 2017.

[12] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional falsification of cyber-physical systems with machine learning components," *J. Autom. Reason.*, vol. 63, no. 4, pp. 1031–1053, 2019.
[13] T. Dreossi et al., "VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems," in *Proc. Int. Conf. Comput.-Aided Verification*, 2019, pp. 432–442.
[14] A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, and V. Mascardi, "ROSMonitoring: A runtime verification framework for ROS," in *Proc. Annu. Conf. Towards Auton. Robot. Syst.*, 2020, pp. 387–399.
[15] G. Frehse et al., "SpaceEx: Scalable verification of hybrid systems," in *Proc. Int. Conf. Comput. Aided Verification*, 2011, pp. 379–395.
[16] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal analysis and redesign of a neural network-based aircraft taxiing system with VERIFAI," in *Proc. Int. Conf. Comput. Aided Verification*, 2020, pp. 122–134.
[17] N. Fulton and A. Platzer, "Safe reinforcement learning via formal methods: Toward safe control through proof and learning," in *Proc. AAAI Conf. Artif. Intell.*, vol. 32, 2018, pp. 6485–6492.
[18] S. L. Herbert, M. Chen, S. Han, S. Bansal, J. F. Fisac, and C. J. Tomlin, "FaSTrack: A modular framework for fast and guaranteed safe motion planning," in *Proc. IEEE 56th Annu. Conf. Decis. Control (CDC)*, 2017, pp. 1517–1522.
[19] A. G. Hofmann and B. C. Williams, "Robust execution of temporally flexible plans for bipedal walking devices," in *Proc. ICAPS*, 2006, pp. 386–389.
[20] J. Huang et al., "ROSRV: Runtime verification for robots," in *Proc. Int. Conf. Runtime Verification*, 2014, pp. 247–254.
[21] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "JavaMaC: A run-time assurance approach for Java programs," *Formal Methods Syst. Design*, vol. 24, no. 2, pp. 129–155, 2004.
[22] S. Kong, S. Gao, W. Chen, and E. Clarke, "dReach: δ-reachability analysis for hybrid systems," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2015, pp. 200–205.
[23] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.
[24] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Trans. Robot.*, vol. 25, no. 6, pp. 1370–1381, Dec. 2009.
[25] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Cambridge, MA, USA: MIT Press, 2017.
[26] H. X. Li and B. C. Williams, "Generative planning for hybrid systems based on flow tubes," in *Proc. ICAPS*, 2008, pp. 206–213.
[27] L. Masson, J. Guiochet, H. Waeselynck, K. Cabrera, S. Cassel, and M. Törngren, "Tuning permissiveness of active safety monitors for autonomous systems," in *Proc. NASA Formal Methods Symp.*, 2018, pp. 333–348.
[28] O. Michel, "Webots: Professional mobile robot simulation," *J. Adv. Robot. Syst.*, vol. 1, no. 1, pp. 39–42, 2004.
[29] I. M. Mitchell, A. M. Bayen, and C. J. Tomlin, "A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games," *IEEE Trans. Autom. Control*, vol. 50, no. 7, pp. 947–957, Jul. 2005.
[30] S. Mitsch and A. Platzer, "ModelPlex: Verified runtime validation of verified cyber-physical system models," *Formal Methods Syst. Design*, vol. 49, no. 1, pp. 33–74, 2016.
[31] O. Pettersson, "Execution monitoring in robotics: A survey," *Robot. Auton. Syst.*, vol. 53, no. 2, pp. 73–88, 2005.
[32] D. Phan, J. Yang, M. Clark, R. Grosu, J. Schierman, S. Smolka, and S. Stoller, "A component-based simplex architecture for high-assurance cyber-physical systems," in *Proc. 17th Int. Conf. Appl. Concurrency Syst. Design (ACSD)*, 2017, pp. 49–58.
[33] D. T. Phan, R. Grosu, N. Jansen, N. Paoletti, S. A. Smolka, and S. D. Stoller, "Neural simplex architecture," in *Proc. NASA Formal Methods Symp.*, 2020, pp. 97–114.
[34] M. Quigley et al., "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, vol. 3, 2009, p. 5.
[35] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *Proc. 18th Int. Conf. Hybrid Syst. Comput. Control*, 2015, pp. 239–248.
[36] J. D. Schierman et al., "Runtime assurance framework development for highly adaptive flight control systems," Barron Assoc., Charlottesville, VA, USA, Rep. AD1010277, 2015.
[37] S. A. Seshia, D. Sadigh, and S. S. Sastry, "Towards verified artificial intelligence," 2016, *arXiv:1606.08514*.
[38] L. Sha, "Using simplicity to control complexity," *IEEE Softw.*, vol. 18, no. 4, pp. 20–28, Jul./Aug. 2001.
[39] S. Shivakumar, H. Torfah, A. Desai, and S. A. Seshia, "SOTER on ROS: A run-time assurance framework on the robot operating system," in *Proc. Int. Conf. Runtime Verification*, 2020, pp. 184–194.
[40] I. A. Sucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robot. Autom. Mag.*, vol. 19, no. 4, pp. 72–82, Dec. 2012.
[41] D. Ulus, "Online monitoring of metric temporal logic using sequential networks," 2019, *arXiv:1901.00175*.
[42] "Open-source mobile robot simulation software." Webots. 2022. [Online]. Available: http://www.cyberbotics.com

Beyazit Yalcinkaya received the B.Sc. degree in computer engineering from Middle East Technical University, Ankara, Turkey, in 2020. He is currently pursuing the Ph.D. degree with the EECS Department, UC Berkeley, Berkeley, CA, USA.

His research interests include formal methods, robotics, and autonomous cyber–physical systems.

Hazem Torfah received the Doctoral degree in computer science from Saarland University, Saarbrücken, Germany, in 2019.

He is a Postdoctoral Researcher with the EECS Department, UC Berkeley, Berkeley, CA, USA. His research interests are the formal specification, verification, and synthesis of autonomous cyber–physical systems.

Ankush Desai received the Ph.D. degree in computer science from UC Berkeley, Berkeley, CA, USA, in 2019.

He is a Senior Applied Scientist with Amazon Web Services, Cupertino, CA, USA. He is working on building formal tools and techniques that help developers reason about the correctness of complex distributed services across AWS.

Sanjit A. Seshia (Fellow, IEEE) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 2005.

He is the Cadence Founders Chair Professor with the EECS Department, UC Berkeley, Berkeley, CA, USA. His research interests are in formal methods for dependable and secure computing, with a current focus on the areas of cyber–physical systems, computer security, machine learning, and robotics.

Dr. Seshia has several awards and honors, including the Donald O. Pederson Best Paper Award for IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.