Safety Enforcement for the Verification of Autonomous Systems

Dionisio de Niz[a], Bjorn Andersson[a], and Gabriel Moreno[a]

[a]Carnegie Mellon University SEI, 4500 Fifth Avenue, Pittsburgh, PA, USA

ABSTRACT

Verifying that the behavior of an autonomous systems is safe is fundamental for safety-critical properties like preventing crashes in autonomous vehicles. Unfortunately, exhaustive verification techniques fail to scale to the size of real-life systems. Moreover, these systems frequently use algorithms whose runtime behavior cannot be determined at design time (e.g., machine learning algorithms). This presents another problem given that these algorithms cannot be verified at design time. Fortunately, a technique known as runtime assurance can be used for these cases. The strategy that runtime assurance uses to verify a system is to add small components (known as enforcers) to the system that monitor its output and evaluate whether the output is safe or not. If the output is safe, then the enforcer will let it pass; if the output is unsafe, the enforcer replaces it with a safe output. For instance, in a drone system that must be restricted to fly within a constrained area (a.k.a. geo-fence) an enforcer will monitor the movement commands to the drone. Specifically, if a movement command keeps the drone within the geo-fence, the enforcer lets it pass, but if the command takes the drone outside of this area, the enforcer replaces it with a safe command (e.g., hovering). Given that enforcers are small components fully specified at design time, it is possible to use exhaustive verification techniques to prove that they can keep the behavior of the whole system safe (e.g., the drone flying within the geo-fence) even if the system contains unverified code.

Keywords: Autonomous Systems, Verification, Runtime Assurance, Enforcers

1. INTRODUCTION

Autonomous systems are systems controlled by software that interact with the physical world. This is in fact the key characteristic of a Cyber-Physical System (CPS). CPS play numerous safety-critical roles in our day-to-day lives, e.g., in the form of cars, airplanes, nuclear power plants, and medical devices. Verifying safe behavior of CPS is thus an important challenge. Autonomous systems, in addition, incorporate advanced AI techniques, such as machine learning, to deliver more features and capabilities. Examples of these systems include driverless cars, autonomous drones, and intelligent patient monitors. On the one hand, the added intelligence allows the CPS to operate more effectively and with less human supervision. On the other hand, it also makes static verification of the CPS inadequate since the system evolves during operation, and the complete set of its behaviors cannot be modeled precisely prior to deployment.

Runtime assurance allows the verification of systems whose behavior are not fully defined through the addition of small components called enforcers creating an *enforced system*. Enforcers monitor the system output and evaluate whether the output (e.g. drone command movements) is safe or not. If it is unsafe, the enforcer replaces the output with a safe one (e.g. hover) otherwise it lets is pass. The enforced system is then verified for safety by focusing only on the behavior of the enforcers in term of how they evaluate output safety and the replaced output.

In order to verify the correct behavior of a CPS we not only need to evaluate whether the software produces the correct actuation over the physical process but that also they are produced at the right time. For instance, to ensure that a drone keeps flying in the correct trajectory, the software needs to read the sensors (e.g., gyroscope and accelerometers) and evaluate the drone attitude and correct it if needed to compensate for perturbations such as variations to wind. This correction takes the form to variations in speed to the different propellers. These corrections need to happen in a timely manner to ensure that the drone remains stable. This timely interaction

dionisio@sei.cmu.edu, baandersson@sei.cmu.edu, gmoreno@sei.cmu.edu

happens through a periodic execution of a function and this periodicity (or period) is defined according to some control-theoretic model. Clearly, if the software takes too long to correct flight perturbations the drone can become unstable and crash. These timely interactions apply to different physical aspects. For instance, to ensure that a drone remains within a geo-fence area, it is necessary that the worst delay between sensing its position and correcting its movement is smaller than the time it needs to stop the drone and prevent it from leaving the geo-fence area in the worst-case.

The fact that a CPS requires timely interaction between the software and the physical process implies that it is not enough to wait forever for the system to produce an output and evaluate whether or not it is safe. Instead, we need to also ensure that if the system has not produced an output by a deadline (e.g., end of the period) we are able to produce an output that will keep the system safe. This is another form of enforcement that we name *temporal enforcer*. To avoid confusion we rename the enforcers that monitor the outputs as *logical enforcers*.

In this paper we present a runtime verification scheme to perform logical and timing verification of CPS with the use of logical and temporal enforcers and show how this is applied to the drone example. We will use this example across the paper to discuss the main concepts.

The rest of the paper is organized as follows. In Section 2 we present our logical model based on a set-theoretic models of control. Section 3 we present our temporal enforcement model and mechanisms. In Section 4 we present our drone example implementation. Section 5 presents the related work and in Section 6 we present our conclusions.

2. LOGICAL ENFORCEMENT

In this section we first define a timeless logical model that provides a gentle introduction of the system model and the definition of safe states and safe actions. Then we extend the model to include the periodic character of software actuation. Finally, we introduce the a model of the inertia of the physical system to have a complete model that can be use in CPS.

2.1 Timeless Logical Model

In order to verify the logical enforcer we first need to define what safe means. We do this by modeling all possible states the system can be in (a.k.a. *statespace*) and identifying what are the safe and unsafe states. In the geo-fenced drone example, the safe states are all the positions within the geo-fenced area and the unsafe states all the positions outside the geo-fenced area.

Formally speaking we say that the statespace S is the set of all possible states $\{s\}$ or $S = \{s\}$. Then the safe states ϕ is a subset of the state space: $\phi \subseteq S$, and everything outside ϕ is unsafe.

Once safe states are defined we then define the behavior of the system and identify safe and unsafe behavior. The behavior of the system is defined as the set of all possible transitions between different states of the system triggered by a system action (e.g. software actuation). Given this behavior then we can identify safe behavior as the set of all transitions from one safe state (the source state) of the system to another safe state (the target state). Similarly, the unsafe behavior is the set of all transitions whose target state is unsafe (the source state can be either a safe or an unsafe state). From the point of view of the geo-fenced drone, an unsafe transition can go from a position within the geo-fenced area to another position outside the geo-fence. In this case the actions of the system are drone movements that take the drone from one position to another. This model is based on set-theoretic methods of control as defined in [1].

The behavior can then be formalized as transitions that take an action α from one state to another: $R(\alpha) \subseteq S \times S$. Now, in order to formalize safe actions we first identify the set of states that a transition will take us into given a particular source state and an action as: $R(\alpha, s) = \{s' | (s, s') \in R(\alpha)\}$. Then we formalize safe actions as those that take us from one state into a safe state: $SafeAct^*(s) = \{\alpha | R(\alpha, s) \subseteq \phi\}$.

2.2 Time-aware Logical Model

The timeless model assumes that transitions take no time to execute and can be executed as frequently as desired. In reality, as we discussed in Section 1, the software in CPS is executed in a periodic fashion with a specific period. In order to reflect that we restrict transitions to only take place at every period P. The formalization of this concept is performed by only annotating the transitions with a P ($R_P(\alpha)$ and $R_P(\alpha, s)$) with the semantics that the system reaches the target state of the transition at the end of the period.

2.3 Modeling Inertia

The final step to capture all the characteristics of a CPS is to add the effect of inertia to our time-aware model. This means that the system continues to evolve between actuations while the period P elapses. To take this into account we identify the set of states that are too close to the unsafe states for which it is not possible to stop the system before it goes into an unsafe state. We use these states to define the set of states called enforceable states C_ϕ. These states are those for which there exists an action that triggers a transition into another enforceable state. For the geo-fenced drone the enforceable states are those positions within the geo-fence for which a drone command exists to keep the drone within the fence and within enforceable states. This is formalized as:

$$\forall s \in C_\phi | \exists \alpha | R_P(\alpha, s) \subseteq C_\phi \tag{1}$$

and create a final definition of a safe action as:

$$SafeAct(s) = \{\alpha | R_P(\alpha, s) \in C_\phi\} \tag{2}$$

2.4 Logical Enforcers

The time-aware logical model now let us formalize the concept of an enforcer E as a component that monitors the system at every period P, to enforce that the system remains within enforceable safe states $C_\phi \subseteq \phi$ with a specific set of safe actions for each enforceable states $\mu(s) \subseteq SafeAct(s)$. Putting it all together: $E(P, C_\phi, \mu)$.

The enforcer then intercepts the output α from the system and produces an output $\tilde{\alpha}$ as follows

$$\tilde{\alpha} = \begin{cases} \alpha & \text{if } \alpha \in \mu(s) \\ pick(\mu(s)) & \text{otherwise} \end{cases} \tag{3}$$

where *pick(X)* selects any one element from set X.

2.5 Geo-Fenced Drone Logical Sample Model

Let us now describe the geo-fenced drone example with our time-aware logical model. The elements of the model are presented in Figure 1.

In this example the state of the system is the position and angle of movement at constant speed (x, y, θ) and the actions is a simple movement at a constant speed in the direction of an angle θ. The safe states is basically any position (x, y) within the fence defined by the rectangle Z with corners $(X_{min}, Y_{min}), (X_{max}, Y_{max})$.

Then the inertia of the system is encoded as the maximum distance that the drone can travel δ_B in opposite direction of the enforcement action . This allows us to define the enforceable state as:

$$C_\phi = \{(x, y, \theta) | (x + \delta_B, y + \delta_B) \in Z \wedge (x - \delta_B, y - \delta_B) \in Z\} \tag{4}$$

The enforcer is then defined as angles of movement depending on how close they are to specific boundary. For this we define the maximum distance the drone can travel in any direction as δ_P then. Then, if it gets too close to any of the boundaries (sides of the rectangle) it selects an angle of movement from the set of enforcement angles according to closes boundary as presented in Figure 1 and defined as:

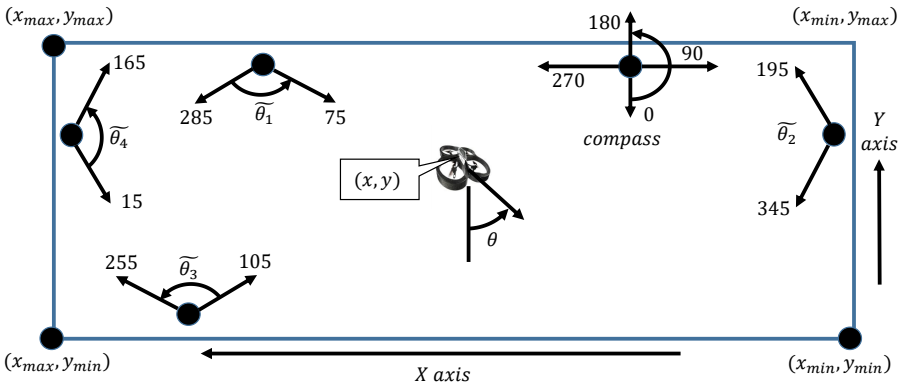

Figure 1. Geo-Fenced Example

$$\tilde{\theta} = \begin{cases} \tilde{\theta} \in \tilde{\theta_1} & \text{if } Y_{max} - y \leq \delta_B + \delta_P \\ \tilde{\theta} \in \tilde{\theta_2} & \text{if } x - X_{min} \leq \delta_B + \delta_P \\ \tilde{\theta} \in \tilde{\theta_3} & \text{if } y - Y_{min} \leq \delta_B + \delta_P \\ \tilde{\theta} \in \tilde{\theta_4} & \text{if } X_{max} - x \leq \delta_B + \delta_P \\ \theta & \text{otherwise} \end{cases} \tag{5}$$

3. TEMPORAL ENFORCEMENT

The time-aware logical model presented in Section 2 assume that the output is produced before the next periodic activation of the software occurs. Unfortunately, the only code we verify is the one in the enforcers. This means that the rest of the software may contain bugs that can potentially delay the execution (e.g., leading to an infinite loop) well beyond the end of the period. For this case we developed the temporal enforcement.

3.1 Real-Time Scheduling

Before describing in detail the temporal enforcement techniques we first introduced the basic concepts from real-time scheduling that we use. As mentioned in Section 1, real-time software uses functions that execute periodically to sense and actuate over a physical process to keep it under control. This functions live within tasks (or threads) that wait for the period to elapsed, execute the periodic function and go back to sleep waiting for the next period to elapsed (also known as the deadline of the current activation).

Given that a system typically interact with multiple physical processes, the software is organized as a set of task (or taskset) that is scheduled with a real-time scheduler within a Real-Time OS. One of the most common real-time schedulers is the Fixed-Priority Scheduler that is typically used with priorities assigned in a Rate-Monotonic fashion, i.e., tasks with shorter periods are assigned a higher priority. Tasksets scheduled in this fashion are said to be scheduled under Rate-Monotonic Scheduling (RMS).

Liu and Layland [2] develop one of the basic analysis techniques for tasks under RMS to verify if the tasks will always finish before the next activation. Specifically, if each task τ_i is characterized with a period T_i and a Worst-Case Execution Time (WCET) C_i we can calculate the utilization of each task as $\frac{C_i}{T_i}$ and the total utilization of the taskset of n tasks as the sum of the utilization of all the tasks:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \tag{6}$$

Lui and Layland proved that if the utilization is less than $ln(2) \approxeq .69$ then the tasks are always guaranteed to finish before its next activation, i.e.:

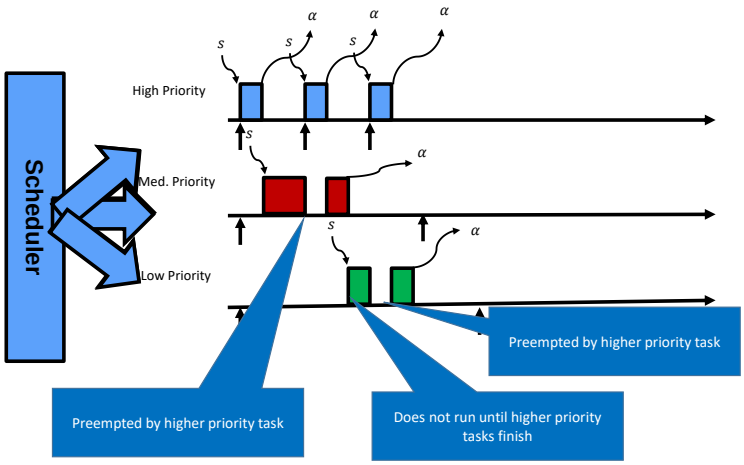

Figure 2. Preemptions in Rate-Monotonic Scheduling

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq .69 \tag{7}$$

A task running under RMS start executing as soon as no other higher-priority task is ready to run and keeps executing until either of two things happen (i) it finishes its periodic activation and goes to sleep waiting for the next period to elapsed or (ii) a higher-priority task becomes ready to execute. The interruption of the lower-priority task by a higher-priority task is known as a preemption and the analysis developed by Liu and Layland assumes that these preemptions are bounded by the WCET of these tasks. Figure 2 shows these preemptions.

Figure 2 shows how tasks of different priorities execute periodically reading sensor data s and producing actuation commands α at the end of their periodic activation. It is worth noting that preemptions delay the completion of tasks. For instance, the red task is delayed by the blue one and the green one is delayed by both the red and the blue. Liu and Layland analysis assume that the data provide to the analysis matches the behavior of the tasks, namely that the tasks do not activate more frequently than its period and that it does not execute more than the stated WCET. In this paper we assume that the periodicity is enforce by the RTOS. Hence, we focus on the problems due to the violation of the WCET assumption.

3.2 Temporal Errors

When a task executes beyond its WCET it can cause two temporal errors. First, the faulty task can cause a longer preemption on a lower-priority task than anticipated. This can be observed in Figure 3. In this figure we can see that when the blue task execute for a long time (beyond its WCET) it causes both the red and green tasks to miss their deadlines. We call this problem *unbounded preemption*.

The missing of deadlines caused by unbounded preemption means that the actuation (α) is not sent at the right time leading to errors like the drone getting out of the geo-fence in our example.

The second temporal error shown in Figure 3 is that the blue task also misses the deadline. We identify this problem as *unbounded execution*. The difference between the first and the second error is subtle but important. In the first error, the faulty task is not the one suffering the consequences (tasks red and green). In the second case, the faulty task also suffer the consequences. This difference requires different protection mechanisms discussed next.

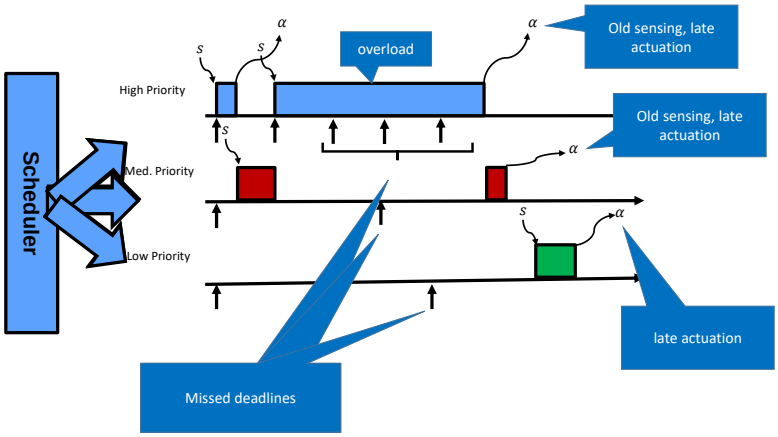

Figure 3. Temporal Errors

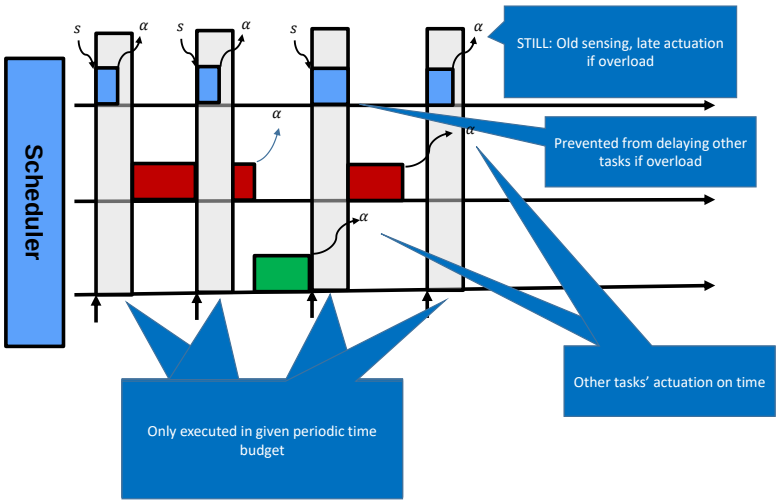

Figure 4. Budget Enforcement

3.3 Preventing Unbounded Preemption

A number of techniques used for unbounded preemption has been developed over the year for RMS. These are know as *processing servers* [3–5] and they were later incorporated into OS mechanisms known as resource reservation [6, 7] and generalized for other types of scheduling such as mixed-criticality scheduling [8].

Resource reservations are time budgets for the use of time-shared resource such as the CPU. More specifically, a time budget is assigned to a task and the OS monitors the time the task uses the CPU. If the task tries to use the CPU longer than the assigned budget then the OS stops the task (enforces the budget). CPU budgets for RMS scheduled tasks are replenished periodically, such that at each activation the task has a renewed budget to use during its current activation. The enforcement of CPU budgets allows a task to not execute beyond its WCET (implemented as a budget). This can be seen in Figure 4.

Figure 4 depicts the CPU reservation (budget) applied to the blue task as a gray rectangle for each activation. The enforcement of the budget is shown in the third activation of the blue task when it tries to continue executing but it is paused by the OS to be later resumed when its period elapses and it is able to complete its execution and output its actuation command α.

It is worth noting that, while the deadline misses and late actuation of the red and green task are prevented by the budget enforcement shown in Figure 4, the blue task still has a late actuation beyond the deadline of the

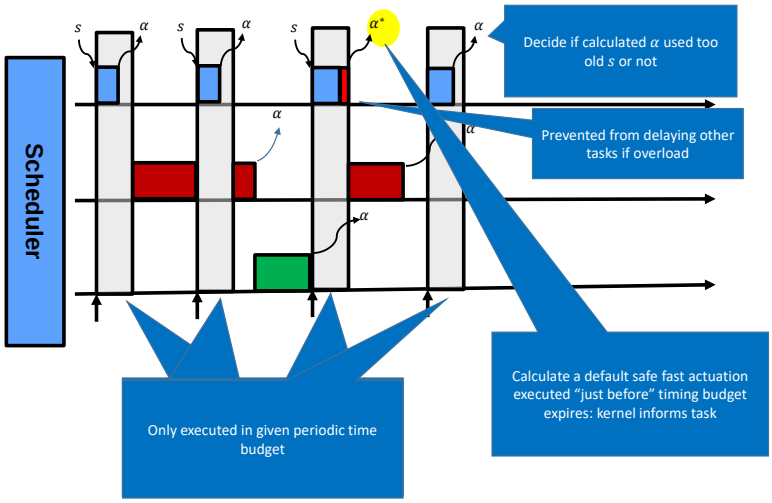

Figure 5. Temporal Enforcement

third activation. This is because we corrected the unbounded preemption problem but we have not corrected the unbounded execution. This is discussed next.

3.4 Preventing Unbounded Execution

Preventing an unbounded execution is trickier than unbounded preemption because we cannot force a task to execute faster if we have not control over the code it is executing. We solve this problem by adding a piece of code that is under our control that we call the temporal enforcer. The temporal enforcer is verified code that is executed at the last instant before we miss a deadline. This code is application specific and the purpose is to issue a very simple *safe actuation* command that prevent the CPS from violating a safety condition. For instance, for the case of our drone this could be a hovering command.

To implement the temporal enforcer we reserve a small part of the budget to be able to execute the enforcer. In other words, the new budget B_i of task τ_i is composed of the stated WCET of the task C_i and the enforcer WCET E_i such that $B_i = C_i + E_i$. However, the enforcement is set for C_i and a the temporal enforcer is called to execute for at most E_i. The call to the enforcer can be implemented as a signal handler.

The combination of the budget enforcement and the temporal enforcement is presented in Figure 5. This figure shows the execution of the temporal enforcement outputing a default actuation α^*.

It is worth noting that after the temporal enforcer has issued a default command it is necessary to re-evaluate what to do with the actuation command of the normal activation when it completes late (as presented in Figure 5). This is because such an actuation was calculated with sensing data from an old activation (one period or several periods old). This is application specific and a decision procedure must be devised to solve this issue.

4. A DRONE EXAMPLE IMPLEMENTATION

The drone example discussed throughout the paper was implemented using a parrot minidrone controlled with a laptop which in turn receive navigation commands from a user using an XBox game controller. It is worth noting that the user commands have the same unpredictable nature of a machine learning algorithm given that its behavior cannot be modeled at design time.

A logical controller was added to the drone controller to ensure that it stays within a virtual fenced area. To evaluate the current position of the drone indoors in our lab we use an Optitrack motion-capture system with six cameras. This camera system uses a desktop computer that process the video coming from the cameras and locates a set of reflective dots put on top of the drone. The position is then multicast to an IP multicast channel that is received by the laptop and used by the logical enforcer.

To enable the verification of the logical enforcer and the composition of multiple enforcers (not discussed in this paper) we implemented the enforcers as SMT formulae and executed them with the Z3 [9] verifier. A full discussion of this verification approach was presented in the Runtime Verification Conference in 2017 [10].

The temporal enforcer was implemented extending the verified budget enforcer presented in [11]. This was extended to implement budget enforcement signal handlers and take into account the enforcement budgets. Hovering was used as the safe actuation performed by the temporal enforcement.

We conducted experiments to test both the logical and temporal enforcement. The complexity of the Z3 verification process was a good test for the temporal enforcer. Specifically, a number of caching techniques were used to speed up the Z3 verification but we were never able to obtain a guaranteed WCET. As a result, when the verification took too long, the temporal enforcer kicked in and prevented the drone from leaving the virtual fenced area. As we fine tuned the caching techniques, the execution temporal enforcer was minimized. However, the experiment clearly highlighted the need of the temporal enforcer to ensure that the safety condition was never violated.

5. RELATED WORK

A brief overview of runtime enforcement techniques is available in [12,13]. Seto et al. [14] proposed the "Simplex" architecture for resilient control systems, where a monitor switches a system from a complex and more capable, but untrusted, controller C_{comp} to a simple but trusted controller C_{simp}, whenever the system is in danger of becoming unstable. The main focus of this work is on deciding the switching boundary based on control theory. Bak et al. [15] have developed a version of Simplex that combines offline analysis with hybrid reachability at runtime to further push the envelope of recoverability. We focus on efficient implementations of the switching logic, and combining multiple enforcers.

The idea of runtime monitoring has also been used in the context of formal verification [**?**, 16]. The key idea is to check for violations of a target safety property at runtime. This is more tractable than complete static verification since we are only analyzing the states that are reached during execution. Our approach is aimed at implementing runtime monitors using SMT solvers, and resolving conflicting actuation decisions.

In the domain of security, Schneider proposed "security automata" [17] as a formalism to express properties whose violations can be detected at runtime. Originally, security automata were passive, i.e., they only monitored the system for safety violations. Restricted versions of this has been considered: Viswanathan [**?**Section 4.3]viswanathan00] studied the case that the enforcer must be decidable and Fong [18] studied the case where memory is limited. More recently, Ligatti at al. [19] have generalized this idea to "edit automata" that can not only monitor system inputs and outputs, but also modify them as needed. Similarly, Pinisetty et al. [20] monitor and allow changing input and outputs for synchronous systems. This is similar in spirit to our enforcers. However, our enforcers also have real-time constraints (i.e., deadlines) since they are targeted toward CPS. Moreover, we focus on combining multiple enforcers, and efficient and incremental SMT-based implementations.

Falcone et al. [21] explore runtime verification of reactive systems where properties include finite and infinite sequences (i.e., Safety-Progress), and are expressed via (untimed) Streett automata. In contrast, we consider safety properties and consider enforcement in the context of the use of a real-time scheduler. The literature has also considered monitoring of multiple properties. Pinissety and Tripakis [22] use one monitor for each property, and enforce them either sequentially or in parallel. Instead, we construct a single monitor for multiple properties. Previous work [23, 24] has also considered synthesizing monitors from set of properties, assuming they are consistent. We focus on resolving such inconsistencies based on prioritization.

The role of timing in run-time verification deserves mentioning. Timing matters in the sense that the evaluation of the property that is monitored may be a function of the time of events. This is studied in [25, 26]. Another aspect of timing, however, is that regardless of whether evaluation of the property that is monitored depends on the timing of event, we would like to run the program that performs the enforcer at the right time; this is the aspect of timing that we have studied in this paper (using a real-time scheduler).

In the domain of real-time scheduling, enforcers have also been used widely, particularly to enforce CPU usage budgets by threads. For example, the ZSRM [8] mixed-criticality scheduler allocates CPU cycles to threads in

a way that respects their priorities (during nominal execution) and criticalities (during overload execution). To this end, it uses timers to intercept thread execution and take appropriate preemptive and budget enforcement steps. While we use ZSRM to ensure schedulability of enforcers, our main focus is on symbolic implementation and combination of logical enforcers.

Pike et al. [27] describe CoPilot, a runtime assurance approach for embedded systems. They focus on a single enforcer, which transforms a stream of application commands to commands that will ensure system safety. The enforcer is specified in a high-level domain specific language, from which efficient (but non-symbolic) implementations are automatically generated. A cyclic executive is used for scheduling both the enforcer and applications. We are inspired by this work, and take the same approach when defining the semantics of a single enforcers. However, our main contribution is on efficient symbolic implementations of enforcers using SMT solvers, and combination of multiple enforcers.

6. CONCLUSIONS

The verification of autonomous systems with unpredictable techniques like machine learning are specially problematic because their behavior cannot be defined at design time. This is further complicated by the fact that these systems are CPS that need to interact with physical processes in a timely manner. In this paper we presented a verification technique based on runtime assurance that uses verified enforcers to verify the safety of autonomous systems. Two type of enforcers were presented in this paper, a logical enforcer that monitors the output of the system and replaces it when it is deemed unsafe and a temporal enforcer that ensure that at the very least a safe default actuation is output to keep the system safe. Finally we discuss how this scheme was implemented in a drone example and how the two types of enforcers complement each other.

ACKNOWLEDGMENTS

 This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation. DM18-0393

REFERENCES

[1] Lucia, W., Sinopoli, B., and Franze, G., "A set-theoretic approach for secure and resilient control of Cyber-Physical Systems subject to False Data Injection Attacks," in [*Proceedings of Science of Security for Cyber-Physical Systems Workshop (SOSCYOS)*], (2016).

[2] Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM* **20**, 46–61 (Jan. 1973).
[3] Strosnider, J. K., Lehoczky, J. P., and Sha, L., "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers* **44**, 73–91 (Jan 1995).
[4] Sprunt, B., Sha, L., and Lehoczky, J., "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems* **1**, 27–60 (Jun 1989).
[5] Sprunt, B., Sha, L., and Lehoczky, J., "Scheduling sporadic and aperiodic events in a hard real-time system," Tech. Rep. CMU/SEI-89-TR-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1989).
[6] Mercer, C. W., Savage, S., and Tokuda, H., "Processor capacity reserves: operating system support for multimedia applications," in [*1994 Proceedings of IEEE International Conference on Multimedia Computing and Systems*], 90–99 (May 1994).
[7] Oikawa, S. and Rajkumar, R., "Portable rk: a portable resource kernel for guaranteed and enforced timing behavior," in [*Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*], 111–120 (1999).
[8] de Niz, D., Lakshmanan, K., and Rajkumar, R., "On the Scheduling of Mixed-Criticality Real-Time Task Sets," (2009).
[9] "Z3 prover." https://github.com/Z3Prover/z3. Accessed: 2018-03-20.
[10] Andersson, B., Chaki, S., and de Niz, D., "Combining symbolic runtime enforcers for cyber-physical systems," in [*Runtime Verification*], Lahiri, S. and Reger, G., eds., 68–84, Springer International Publishing, Cham (2017).
[11] Chaki, S. and Niz, D. D., "Formal verification of a timing enforcer implementation," *ACM Trans. Embed. Comput. Syst.* **16**, 168:1–168:19 (Sept. 2017).
[12] Havelund, K. and Goldberg, A., "Verify your runs," in [*VSTTE*], (2005).
[13] Leucker, M. and Schallhart, C., "A brief account of runtime verification," in [*JLAP*], (2008).
[14] Seto, D., Krogh, B., Sha, L., and Chutinan, A., "The simplex architecture for safe online control system upgrades," in [*Proceedings of the American Control Conference*], (1998).
[15] Bak, S., Johnson, T., Caccamo, M., and Sha, L., "Real-Time Reachability for Verified Simplex Design," (2014).
[16] Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O., "Formally specified monitoring of temporal properties," (1999).
[17] Schneider, F., "Enforceable security policies," (2000).
[18] Fong, P., "Access Control By Tracking Shallow Execution History," in [*IEEE Security and Privacy*], (2004).
[19] Ligatti, J., Bauer, L., and Walker, D., "Edit automata: enforcement mechanisms for run-time security policies," (2005).
[20] Pinisetty, S., Roop, P., Smyth, S., Tripakis, S., and Hanxleden, R., "Runtime enforcement of reactive systems using synchronous enforcers," CoRR abs/1612.05030 (2016).
[21] Falcone, Y., Mounier, L., Fernandez, J.-C., and Ricier, J.-L., "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," (2011).
[22] Pinisetty, S. and Tripakis, S., "Compositional Run-time enforcement," in [*NFM*], (2016).
[23] Bloem, R., Könighofer, B., Könighofer, R., and Wang, C., "Shield synthesis: runtime enforcement for reactive systems," in [*TACAS*], (2015).
[24] Wu, M., Zeng, H., and Wang, C., "Synthesizing runtime enforcer of safety properties under burst error," in [*NFM*], (2016).
[25] Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., and Timo, O., "Runtime Enforcement of Timed Properties," (2012).
[26] Basin, D., Jugé, V., Klaedtke, F., and Zălinescu, E., "Enforceable Security Policies Revisited," (2013).
[27] Pike, L., Wegmann, N., Niller, S., and Goodloe, A., "Copilot: monitoring embedded systems," (2013).