Combining Symbolic Runtime Enforcers for Cyber-Physical Systems

Björn Andersson Sagar Chaki Dionisio de Niz

{baandersson,chaki,dionisio}@sei.cmu.edu

Abstract. The problem of composing multiple, possibly conflicting, runtime enforcers for a cyber-physical system (CPS) is considered. A formal definition of utility-agnostic and utility-maximizing CPS enforcers is presented, followed by an algorithm to combine multiple enforcers, and resolve their conflicts based on a design-time prioritization. To implement this combination in an efficient manner, enforcers are encoded symbolically using SMT formulas, and the combination is reduced to a set of SMT satisfiabilty and optimization operations. Further performance gains are achieved by using the SMT solvers incrementally. The approach is validated via experiments in an indoor area with Parrot minidrones. The incremental enforcer combination is shown to achieve an order of magnitude speedup, and no deadline misses.

1 Introduction

Cyber-Physical Systems (CPS) are "engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components"[1]. They 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. At the same time, modern CPS are incorporating advanced AI techniques, such as machine learning, to deliver more features and capabilities. Examples include driverless cars, intelligent patient monitors, and smart home appliances. 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.

Some form of "runtime verification" is therefore indispensable for achieving high assurance about the safe behavior of CPS. In this paper, we explore the concept of runtime assurance [15] (RA). Broadly, RA involves adding a runtime monitor (which we call the enforcer) that observes the behavior of the CPS, and intervenes to prevent specific CPS behaviors that could lead to catastrophic results. In particular, if we think of a CPS as a control system, then the enforcer observes, and modifies as needed, actuation signals emitted by the CPS software that determine the CPS's interaction with its physical environment.

In this paper, we address the challenge of soundly composing multiple enforcers that operate on the same set of actuators of a CPS. This is important since

a complex CPS has multiple safety-critical requirements involving the same set of actuators that are not guaranteed to be consistent under all situations. Consider two quadcopters (QCs) flying in an open area. Each QC has two enforcers: E_1 attempts to keep the QC within a specific GPS boundary (geo-fencing) and E_2 attempts to keep the QCs at a minimum distance from each other (collision avoidance). Then, in the situation where one QC is backed up against the geo-fence boundary by the other, the two-enforcers produce conflicting actuations – E_1 wants to move the QC away from the boundary (thus closer to the other QC), while E_2 wants to do the opposite. To address this challenge, we make the following contributions.

First, we formalize the logical behavior of enforcers. Intuitively, an enforcer is an algorithm that executes periodically and computes an appropriate actuation based on the observed system state and the command proposed by the application software. Next, we present an algorithm, called `select`, that combines multiple enforcers ordered by priority. Specifically, `select` computes an actuation that satisfies a subset of enforcers — determined by the priorities of enforcers. We also present a variant of `select`, called $\texttt{select}^*$ that is utility-maximizing, i.e., computes a legal actuation with maximal utility.

Next, we show how enforcers can be encoded logically using Satisfiability Modulo Theories (SMT) formulas, and how to implement `select` and $\texttt{select}^*$ using optimizing SMT solvers. Our symbolic encoding allows enforcers to be expressed succinctly, and enforcer compositions turn naturally into common logical operations. Thus, we bring into RA the benefits of symbolic reasoning, which have been leveraged in other verification domains over the past decades. CPS enforcers have to be very efficient in order to run with short periods (dictated by the application) and yet be guaranteed that each enforcer finishes execution before the enforcer is requested to execute again. To this end, we present *online* implementations of `select` and $\texttt{select}^*$, denoted by $\texttt{select}^\dagger$ and $\texttt{select}^{*\dagger}$, that use the SMT solver incrementally, "pushing" and "popping" formulas into dynamically created "contexts," as needed. We present pseudo-code for all variants of `select` and argue about their correctness.

We used our approach to implement the geo-fencing (a.k.a. tether) and separation enforcers targeting Parrot minidrones. We validated empirically that both enforcers work as expected, individually and in composition, in an indoor area. We also measured the performance of various threads under different variants of the `select` algorithm. These measurements indicate that our symbolic enforcers are quite efficient, with execution times in the range of tens of milliseconds. Furthermore, our online implementation of enforcers yields an order of magnitude speedup in execution time; thanks to this speedup, in our experiments, we observe zero deadline misses over tens of thousands of invocations.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents the formal definition of CPS enforcers and their composition. Section 4 presents the symbolic encoding and implementation of enforcers using SMT formulas and optimizing SMT solvers. Section 5 presents our experiments and results, and Section 6 concludes.

2 Related Work

A brief overview of runtime enforcement techniques is available in [9, 12]. Seto et al. [20] 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. [2] 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 [11, 10]. 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" [19] 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 [21, Section 4.3] studied the case that the enforcer must be decidable and Fong [8] studied the case where memory is limited. More recently, Ligatti at al. [13] 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. [17] 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. [7] 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 [18] 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 [5, 22] has also considered synthesizing monitors from a 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 [16, 3]. 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 [6] 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. [15] 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.

3 Enforcers

Let V_S be the set of state variables. Without loss of generality, we assume for each variable, its domain is D. A state $s : V_S \mapsto D$ is an assignment of values to state variables. Let $\mathbf{S}$ be the set of all states. Let V_Σ be the set of action variables. An action $\alpha : V_\Sigma \mapsto D$ is an assignment of values to action variables. Let Σ be the set of actuations, or actions. We consider a system comprising (i) a target system, (ii) a set of enforcers, and (iii) a set of applications. The enforcers and the applications are software. The target system is typically a physical system (but our paper allows the target system to be another software system as well) and the state s describes the state of the target system. Note that an application may have state itself and this is not described by s. Similarly, we envision the enforcer as a *controller* that executes periodically to sense the system's state, compute an appropriate action, and apply it. A user of an enforcer is an application.

The evolution of the system is modeled by the transition relation R, parameterized by the amount of time that elapses during the transition, and the actuation applied at the start of the transition. Formally $R_P(\alpha) \subseteq \mathbf{S} \times \mathbf{S}$ is the relation such that if the action α is applied to the system at time t when it is in state s and subsequently the system is in state s' at time is $t + P$, then $(s, s') \in R_P(\alpha)$. We write $R_P(\alpha, s)$ to mean the set of states related to state s by $R_P(\alpha)$, i.e., $R_P(\alpha, s) = \{s' \mid (s, s') \in R_P(\alpha)\}$.

We are interested in enforcing safety properties. Intuitively, a safety property ϕ states that something bad never happens. It is defined as a set of "safe" states.

In particular, we are interested in keeping the system within a set of states that are both safe and enforceable.

Formally, given a safety property ϕ, the set of all ϕ-enforceable states is denoted by C_ϕ and defined as the largest set of states satisfying the following two conditions: $C_\phi \subseteq \phi$ and $\forall s \in C_\phi \,.\, \exists \alpha \in \Sigma \,.\, R_P(\alpha, s) \subseteq C_\phi$.

We denote by $\mathit{SafeAct} : C_\phi \mapsto 2^\Sigma$ the mapping from ϕ-enforceable states to actuations that will ensure that the system remains enforceable, i.e.,

$$\mathit{SafeAct}(s) = \{\alpha \mid R_P(\alpha, s) \subseteq C_\phi\}$$

Note that by the definition of C_ϕ we know that: $\forall s \in C_\phi \,.\, \mathit{SafeAct}(s) \neq \emptyset$. Different actions in $\mathit{SafeAct}(s)$ may have different utilities, i.e., even though all of them would continue to keep the system in a ϕ-enforceable state, there may be additional reasons to prefer some of them over others. Our enforcers, which we define formally next, therefore support the notion of actuation utility.

Definition 1 (Enforcer). *An enforcer for safety property ϕ is a 4-tuple (P, C_ϕ, μ, U) where: (i) P is its period; (ii) C_ϕ is the set of all ϕ-enforceable states; (iii) $\mu : C_\phi \mapsto 2^\Sigma$ is a mapping from enforceable states to actuations such that: $\forall s \in C_\phi \,.\, \mu(s) \subseteq \mathit{SafeAct}(s)$; and (iv) $U : C_\phi \times \Sigma \hookrightarrow \mathbb{R}$ maps each ϕ-enforceable state and a corresponding actuation to its utility.*

Note that U is a partial function, and specifically, $\mathit{Dom}(U) = \{(s, \alpha) \mid (s \in C_\phi) \wedge (\alpha \in \mu(s))\}$. Also, the time period P is important since: (i) it specifies how frequently the enforcer must be executed; and (ii) it affects the precise value of C_ϕ. The smaller the value of P, the larger C_ϕ can be. In essence, P is needed because the physical dynamics of the platform and timeliness of enforcement action are crucially important for ensuring safety of a CPS.

Utility-Agnostic Enforcer Operation. Given a non-empty set X, let $\mathrm{pick}(X)$ denote an arbitrary element of X. The enforcer $E = (P, C_\phi, \mu, U)$ is executed periodically every P units of time. In each execution, it takes as input the current system state s and user actuation α, and produces an output actuation $\tilde{\alpha}$ defined as follows:

$$\tilde{\alpha} = \begin{cases} \alpha & \textit{if } \alpha \in \mu(s) \\ \mathrm{pick}(\mu(s)) & \textit{otherwise} \end{cases} \tag{1}$$

Note that if α is an enforcing action at state s, then the enforcer outputs α. Otherwise, it picks and outputs an arbitrary enforcing action α' at state s.

The construction of μ is the crucial step in defining the enforcer. We assume that it is done offline. It could be done via simulations, or analytically (if we have good models), or via some combination of the two.

Utility-Maximizing Enforcer Operation. In general, the enforcer does not have to output the user's command α even if α is enforcing. This is because, in a specific state, different enforcing actions have different utilities and the enforcer may choose the command that maximizes utility. Then the output of the utility maximizing enforcer is defined as follows:

$$\tilde{\alpha} = \begin{cases} \alpha & \textit{if } \alpha \in \mu(s) \wedge U(s, \alpha) = \max\limits_{\alpha' \in \mu(s)} U(s, \alpha') \\ \arg\max\limits_{\alpha' \in \mu(s)} U(s, \alpha') & \textit{otherwise} \end{cases} \tag{2}$$

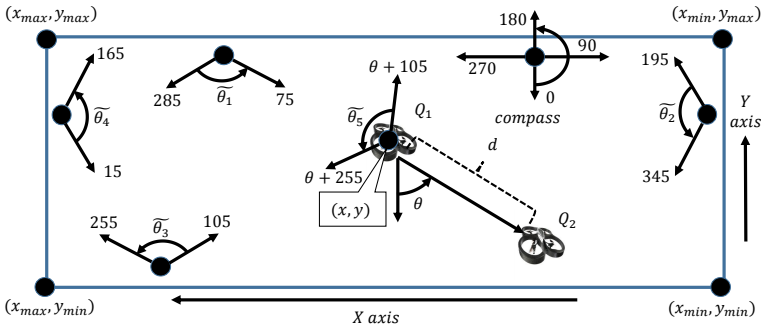

Fig. 1. An example with two enforcers.

Note that (1) is a special case of (2) when U is defined as follows:

$$U(s, \alpha) = \begin{cases} 1 \ if \ \alpha \in \mu(s) \\ 0 \ otherwise \end{cases} \tag{3}$$

3.1 Example

Consider two quadcopters, Q_1 and Q_2, moving in a two-dimensional rectangular zone Z, shown in Figure 1. The boundaries of Z are aligned to the X and Y axes, and the points (x, y) within Z are defined by $x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$. Directions are measured in degrees, using the reference shown on the right of Figure 1. Quadcopter Q_1 has two enforcers, defined as follows: (i) *Virtual Tether*: enforcer E_1 keeps Q_1 physically within the zone Z (safety property ϕ_1); (ii) *Separation*: enforcer E_2 keeps Q_1 at a minimum distance of D from Q_2 (safety property ϕ_2). The state variables are $V_S = \{x, y, \theta, d\}$ where (x, y) is the location of Q_1, and θ and d are the direction and distance from Q_1 to Q_2, respectively. An action α attempts to move Q_1 in a specific direction θ_α. Thus, $V_\Sigma = \{\theta_\alpha\}$. The specific range of values of θ_α that can be applied by each enforcer depends on the state of Q_1.

Operations on Angles. In the following, all operations on angles are performed modulo 360, and all angle values are in the range $[0, 360)$. In particular, given two angles θ and θ', we define $\theta \ominus \theta'$ to be the minimum of the two angles between θ and θ' (one measured clockwise, and the other anti-clockwise). For example, we evaluate $350 \ominus 5$ as follows. The clockwise angle between 350 and 5 is 15; the counter-clockwise angle between 350 and 5 is 345; thus $350 \ominus 5 = 15$. We write θ^{opp} to denote the angle opposite to θ, i.e., $\theta^{opp} = (\theta + 180) \bmod 360$. For example, $90^{opp} = 270$ and $300^{opp} = 120$.

Tether Enforcer Operation. If Q_1 is too close to the y_{max} boundary of Z, then E_1 applies an enforcement action with $\theta_\alpha \in \tilde{\theta_1}$, where $\tilde{\theta_1} = [285, 360) \cup [0, 75)$, as shown at the top of Figure 1. Similarly, if Q_1 is too close to the x_{min} boundary of Z, then E_1 applies an enforcement action with $\theta_\alpha \in \tilde{\theta_2}$ where $\tilde{\theta_2} = [195, 345]$,

as shown at the right of Figure 1. The other two possible enforcements by E_1, involving enforcement action ranges $\tilde{\theta_3} = [105, 255]$ and $\tilde{\theta_4} = [15, 165]$, are also shown in Figure 1. Let δ_{P1} be the maximum distance that Q_1 can travel along any one dimension in time P, and let δ_{B1} (the braking distance) be the maximum distance that Q_1 can travel in a direction opposing an enforcement action (e.g., toward y_{max} when $\theta_\alpha \in \tilde{\theta_1}$ is applied). Furthermore, we assume that the period P is sufficiently large that once the enforcement action is applied, the QC returns to a state in C_{ϕ_1} within P time units. Let $\delta_{PB1} \equiv \delta_{P1} + \delta_{B1}$. This means that an appropriate enforcement action must be applied as soon as the distance between Q_1 and a boundary of Z falls below δ_{PB1}. Clearly, we are interested in situations when Q_1 is too close to the four boundaries of Z. For this reason, we define b_1, b_2, b_3, and b_4 as

$$b_1 \equiv y_{max} - y \leq \delta_{PB1} \qquad b_2 \equiv x - x_{min} \leq \delta_{PB1}$$
$$b_3 \equiv y - y_{min} \leq \delta_{PB1} \qquad b_4 \equiv x_{max} - x \leq \delta_{PB1}$$

We specify μ_1 as: $\forall i \in [1,4] \,.\, b_i \implies \mu_1(x, y, \theta, d) \in \tilde{\theta_i}$. In addition, the ϕ_1-enforceable set of states are those corresponding to locations in Z such that there is enough space to prevent a violation, i.e.,

$$C_{\phi_1} = \{(x, y, \theta, d) \mid (x + \delta_{B1}, y + \delta_{B1}) \in Z \wedge (x - \delta_{B1}, y - \delta_{B1}) \in Z\}$$

The utility of an enforcement action is directly related to the degree to which it opposes the movement toward nearby boundaries. It is the sum of four utilities, one for each boundary. Formally,

$$U_1(x, y, \theta, d, \theta_\alpha) = U^1 + U^2 + U^3 + U^4 \text{ such that}$$
$$U^i = 75 - (\theta^i_{mid} \ominus \theta_\alpha) \text{ if } b_i \text{ and 0 otherwise, where}$$
$$\theta^1_{mid} = 0 \quad \theta^2_{mid} = 270 \quad \theta^3_{mid} = 180 \quad \theta^4_{mid} = 90$$

Separation Enforcer Operation. When Q_1 is too close to Q_2, E_2 applies an enforcement action to move it away. Recall that θ denotes the direction from Q_1 to Q_2. The enforcement is to move Q_1 in a direction θ_α that is opposite to θ. Specifically, we want $(\theta^{opp} \ominus \theta_\alpha) \leq 75$. In Figure 1, the range of possible values of θ_α is denoted by $\tilde{\theta_5}$. Let δ_{P2} be the largest possible reduction in the value of d within time P, and δ_{B2} be the largest possible reduction in the value of d once the separation enforcement is applied. Let $\delta_{PB2} \equiv \delta_{P2} + \delta_{B2}$. Define condition b_{sep} as $b_{sep} \equiv d \leq \delta_{PB2} + D$. We specify μ_2 as: $b_{sep} \implies \mu_2(x, y, \theta, d) \in \tilde{\theta_5}$. In addition, the ϕ_2-enforceable set of states with correct separation, i.e.,

$$C_{\phi_2} = \{(x, y, \theta, d) \mid d + \delta_{B2} \geq D\}$$

The utility of an enforcement action is directly related to the degree to which it is opposite to θ, specifically:

$$U_2(x, y, \theta, d, \theta_\alpha) = 75 - (\theta^{opp} \ominus \theta_\alpha) \text{ if } b_{sep} \text{ and 0 otherwise}$$

$$\texttt{select}(s, \langle E_1 \rangle, \alpha) = \begin{cases} \alpha & \textit{if } \alpha \in \mu_1(s) \\ \text{pick}(\mu_1(s)) & \textit{otherwise} \end{cases}$$

$$\texttt{select}(s, \langle E_1, \ldots, E_n \rangle, \alpha) =$$

$$\begin{cases} \alpha & \textit{if } s \in \bigcap_{i=2,\ldots,n} C_{\phi_i} \wedge \alpha \in \bigcap_{i=1,\ldots,n} \mu_i(s) \\ \text{pick}(\bigcap_{i=1,\ldots,n} \mu_i(s)) & \textit{if } s \in \bigcap_{i=2,\ldots,n} C_{\phi_i} \wedge \bigcap_{i=1,\ldots,n} \mu_i(s) \neq \emptyset \\ \texttt{select}(s, \langle E_1, \ldots, E_{n-1} \rangle, \alpha) & \textit{otherwise} \end{cases}$$

Fig. 2. Function `select` for utility-agnostic enforcer combination.

$$\texttt{select}^*(s, \langle E_1 \rangle, \alpha) = \begin{cases} \alpha & \textit{if } \alpha \in \mu_1(s) \wedge U_1(s, \alpha) = \max_{\alpha' \in \mu_1(s)} U_1(s, \alpha') \\ \arg\max_{\alpha' \in \mu_1(s)} U_1(s, \alpha') & \textit{otherwise} \end{cases}$$

$$\texttt{select}^*(s, \langle E_1, \ldots, E_n \rangle, \alpha) =$$

$$\begin{cases} \alpha & \textit{if } s \in \bigcap_{i=2,\ldots,n} C_{\phi_i} \wedge \alpha \in \bigcap_{i=1,\ldots,n} \mu_i(s) \wedge \\ & \sum_{i=1,\ldots,n} U_i(s, \alpha) = \max_{\alpha' \in \bigcap_{i=1,\ldots,n} \mu_i(s)} \left(\sum_{i=1,\ldots,n} U_i(s, \alpha') \right) \\ \arg\max_{\alpha' \in \bigcap_{i=1,\ldots,n} \mu_i(s)} \left(\sum_{i=1,\ldots,n} U_i(s, \alpha') \right) & \textit{if } s \in \bigcap_{i=2,\ldots,n} C_{\phi_i} \wedge \bigcap_{i=1,\ldots,n} \mu_i(s) \neq \emptyset \\ \texttt{select}^*(s, \langle E_1, \ldots, E_{n-1} \rangle, \alpha) & \textit{otherwise} \end{cases}$$

Fig. 3. Function $\texttt{select}^*$ for utility-maximizing enforcer combination.

3.2 Combining Multiple Enforcers

Consider n enforcers $E_1, \ldots, E_n$ such that $E_i = (P, C_{\phi_i}, \mu_i, U_i)$. Without loss of generality, assume that the enforcers are ordered by decreasing priority. When applied in state s, the combined utility-agnostic enforcer returns the action $\texttt{select}(s, \langle E_1, \ldots, E_n \rangle, \alpha)$, which is defined recursively as shown in Figure 2. Note that since $\texttt{select}(s, \langle E_1, \ldots, E_n \rangle, \alpha)$ is always in $\mu_1(s)$, the system always maintains property ϕ_1. Also note that the second and third cases in the definition of $\texttt{select}(s, \langle E_1, \ldots, E_n \rangle, \alpha)$ cannot be swapped without violating the condition that property ϕ_n must be maintained if at all possible. We assume that utilities are additive. Thus, when combining two enforcers, we assume that the overall utility achieved is the "sum" of the utilities of all the enforcers that are "activated". The utility-maximizing enforcer combination returns the action $\texttt{select}^*(s, \langle E_1, \ldots, E_n \rangle, \alpha)$, which is defined recursively as shown in Figure 3.

4 Enforcer Implementation

In this section, we focus on implementing enforcers in an efficient manner. Recall, from Definition 1, that an enforcer is a 4-tuple (P, C_ϕ, μ, U). The most critical

Enforcer Operation	Symbolic Implementation
$s \in \bigcap_{i=2,\ldots,n} C_{\phi_i}$	$\mathrm{sat}(V_S = s \wedge \bigwedge_{i=2,\ldots,n} C_{\phi_i})$
$\alpha \in \bigcap_{i=1,\ldots,n} \mu_i(s)$	$\mathrm{sat}(V_S = s \wedge V_\Sigma = \alpha \wedge \bigwedge_{i=1,\ldots,n} \mu_i)$
$\bigcap_{i=1,\ldots,n} \mu_i(s) = \emptyset$	$\neg\mathrm{sat}(V_S = s \wedge \bigwedge_{i=1,\ldots,n} \mu_i)$
$\mathrm{pick}(\bigcap_{i=1,\ldots,n} \mu_i(s))$	$\mathrm{soln}(V_S = s \wedge \bigwedge_{i=1,\ldots,n} \mu_i, V_\Sigma)$
$\sum_{i=1,\ldots,n} U_i(s,\alpha) = \max_{\alpha' \in \bigcap_{i=1,\ldots,n} \mu_i(s)} \left(\sum_{i=1,\ldots,n} U_i(s,\alpha') \right)$	$\neg\mathrm{sat}(\Omega(s,\alpha,n))$ where $\Omega(s,\alpha,n) = \left(\begin{array}{c} V_S = s \wedge V_\Sigma = \alpha \wedge k = \sum_{i=1,\ldots,n} U_i \wedge \\ V'_S = s \wedge \bigwedge_{i=1,\ldots,n} \mu'_i \wedge k' = \sum_{i=1,\ldots,n} U'_i \wedge \\ k < k' \end{array} \right)$
$\arg\max_{\alpha' \in \bigcap_{i=1,\ldots,n} \mu_i(s)} \left(\sum_{i=1,\ldots,n} U_i(s,\alpha') \right)$	$\mathrm{opt}(V_S = s \wedge \bigwedge_{i=1,\ldots,n} \mu_i, \sum_{i=1,\ldots,n} U_i, V_\Sigma)$

Table 1. Enforcer operations and their symbolic implementations.

parts to implement here are C_ϕ, μ and U. In particular, following Figure 3, such an implementation must support the operations shown in the left column of Table 1. In this paper, we propose to represent both C_ϕ and μ symbolically, as logical formulas, and implement the above operations using an SMT solver. We now present this in more detail.

Recall that a state is an assignment to state variables V_S, and an action is an assignment to action variables V_Σ. Thus, a logical formula F over V_S, denoted $F(V_S)$, represents a set of states S in the sense that a state s belongs to S iff the assignment to V_S corresponding to s makes $F(V_S)$ true. In the same way, a formula $F(V_\Sigma)$ represents a set of actions, and a formula $F(V_S, V_\Sigma)$ represents a relation over S and Σ. In addition, a state s is representable as a logical formula over V_S, and an action α is representable as a logical formula over V_Σ.

We therefore represent C_Φ as a logical formula over V_S, μ as a logical formula over $V_S \cup V_\Sigma$, and U as a function over $V_S \cup V_\Sigma$. Also, let $V_S = s$ denote the formula that assigns state variables to the specific state s, and $V_\Sigma = \alpha$ denote the formula that assigns action variables to the specific action α. We introduce fresh "primed" copies of variables in V_S and V_Σ, and denote them by V'_S and V'_Σ, respectively. We write μ' and U'_i to mean $\mu(V'_S, V'_\Sigma)$ and $U_i(V'_S, V'_\Sigma)$, respectively. Then, the right column of Table 1 shows the implementations of the necessary enforcer operations using the following logical primitives:

- $\mathrm{sat}(F)$ returns TRUE iff F is satisfiable, and FALSE otherwise;
- $\mathrm{soln}(F, V)$ returns a satisfying solution of F but restricts the assignment to variables in V, i.e., it returns a satisfying solution of $\exists(V_S \cup V_\Sigma \setminus V) \,.\, F$. If F is not satisfiable, then $\mathrm{soln}(F, V)$ returns a special value $\perp$.
- $\mathrm{opt}(F, \Psi, V)$ returns a satisfying solution of F that maximizes the objective function Ψ but restricts the assignment to variables in V. If F is not satisfiable, then $\mathrm{opt}(F, \Psi, V)$ returns a special value $\perp$.

```
1 proc select(s, α, n) {
2   if(n = 1) {
3     if (sat(V_S = s ∧ V_Σ = α ∧ μ_1))
4       return α;
5     return soln(V_S = s ∧ μ_1, V_Σ);
6   }
7   b := sat(V_S = s ∧ C^n);
8   if(¬b) return select(s, α, n − 1);
9   if(sat(V_S = s ∧ V_Σ = α ∧ μ^n))
10    return α;
11  α' := soln(V_S = s ∧ μ^n, V_Σ);
12  if(α' ≠ ⊥) return α';
13  return select(s, α, n − 1);
14 }
```

(a)

```
1 proc select*(s, α, n) {
2   if(n = 1) {
3     if (sat(V_S = s ∧ V_Σ = α ∧ μ_1)∧
4         ¬sat(Ω(s, α, 1))) return α;
5     return opt(V_S = s ∧ μ_1, U_1, V_Σ);
6   }
7   b := sat(V_S = s ∧ C^n);
8   if(¬b) return select*(s, α, n − 1);
9   if(sat(V_S = s ∧ V_Σ = α ∧ μ^n) ∧
10      ¬sat(Ω(s, α, n))) return α;
11  α' := opt(V_S = s ∧ μ^n, Σ_{i=1,...,n} U_i, V_Σ);
12  if(α' ≠ ⊥) return α';
13  return select*(s, α, n − 1);
14 }
```

(b)

Fig. 4. (a) Pseudo-code for utility-agnostic enforcer composition; $C^n \equiv \bigwedge_{i=2,\ldots,n} C_{\phi_i}$; $\mu^n \equiv \bigwedge_{i=1,\ldots,n} \mu_i$; (b) Pseudo-code for utility-maximizing enforcer composition; the formula $\Omega(s, \alpha, n)$ is defined in Table 1.

We assume that all logical formulas are over an underlying theory T for which $\text{sat}(F)$ is decidable, and $\text{soln}(F, V)$ and $\text{opt}(F, \Psi, V)$ are effectively computable. For our experiments, we use linear arithmetic over rationals, for which these two conditions hold. Specifically, we use the Z3 [14] SMT solver to implement $\text{sat}(F)$ and $\text{soln}(F, V)$, and νZ [4] solver to implement $\text{opt}(F, \Psi, V)$.

Example. Recall our example with two enforcers from Section 3.1. Enforcer E_1 is specified symbolically as follows:

$$\begin{aligned}
\mu_1 &\equiv (b_1 \implies (285 \le \theta_\alpha < 360) \vee (0 \le \theta_\alpha \le 75)) \wedge (b_2 \implies (195 \le \theta_\alpha \le 345)) \\
&\quad \wedge (b_3 \implies (105 \le \theta_\alpha \le 255)) \wedge (b_4 \implies (15 \le \theta_\alpha \le 165)) \\
C_{\phi_1} &\equiv (x_{min} + \delta_{B1} \le x \le x_{max} - \delta_{B1}) \wedge (y_{min} + \delta_{B1} \le y \le y_{max} - \delta_{B1}) \\
U_1 &\equiv U^1 + U^2 + U^3 + U^4 \text{ (Sec. 3.1)}
\end{aligned}$$

Similarly, enforcer E_2 is specified symbolically as follows:

$$\begin{aligned}
\mu_2 &\equiv d \le \delta_{PB2} + D \implies (\theta^{opp} \ominus \theta_\alpha) \le 75 \\
C_{\phi_2} &\equiv d + \delta_{B2} \ge D \qquad U_2 \equiv 75 - (\theta^{opp} \ominus \theta_\alpha)
\end{aligned}$$

Pseudo-Code. Figure 4(a) shows the pseudo-code for the utility-agnostic enforcer composition. The code follows from the definition of function `select` in Figure 2 and the definition of symbolic operations in Table 1. However, we have refactored various program statements to avoid calling $\text{sat}(V_S = s \wedge C^n)$ – an expensive operation – multiple times. Also, we eliminated the call to $\text{sat}(V_S = s \wedge \mu^n)$ since the result can also be obtained via the call to $\text{soln}(V_S = s \wedge \mu^n, V_\Sigma)$, which is needed in any case.

Similarly, Figure 4(b) shows the pseudo-code for the utility-maximizing enforcer composition. It follows from the definition of function `select`* in Figure 3 and the definition of symbolic operations in Table 1. Again, we have refactored

the program to avoid calling $\text{sat}(V_S = s \wedge C^n)$ multiple times. Also, we eliminated the call to $\text{sat}(V_S = s \wedge \mu^n)$ since the result can also be obtained via the (necessary) call to $\text{opt}(V_S = s \wedge \mu^n, \sum_{i=1,\ldots,n} U_i, V_\Sigma)$.

4.1 Optimized and Online Versions

We propose three additional optimizations to the pseudo-code shown in Figure 4. OPT1 replaces the recursion with iteration. This optimization is possible since both `select`() and `select`* are tail-recursive functions. It eliminates stack operations and limits the possibility of stack overflows. OPT2 precomputes commonly used formulas C^i and μ^i for $1 \leq i \leq n$, and uses them as needed, instead of creating and destroying them repeatedly. Finally, OPT3 uses the SMT solver in an online manner. Specifically, the online SMT solver maintains a stack of asserted formulas (a.k.a. the context), and updates it via operations push(F) and pop(F) as follows: (i) push(F) pushed formula F to the context; and (ii) pop() pops a formula from the context. Moreover, new operations $\text{sat}^\dagger$, $\text{soln}^\dagger$, and $\text{opt}^\dagger$ are defined as follows (here θ denotes the conjunctions of all the formulas in the current context):

$$\text{sat}^\dagger() \equiv \text{sat}(\theta) \qquad \text{soln}^\dagger(V) \equiv \text{soln}(\theta, V) \qquad \text{opt}^\dagger(\Psi, V) \equiv \text{opt}(\theta, \Psi, V)$$

Both Z3 and νZ support such online operation. In OPT3, the commonly used C^i's and μ^i's are always kept in contexts. Other formulas are pushed prior to an operation, and popped afterwards, as needed. We use the following shorthand:

$$b := \text{sat}^\dagger(\Gamma) \equiv \text{push}(\Gamma); b := \text{sat}^\dagger(); \text{pop}()$$
$$\alpha := \text{sat}^\dagger(\Gamma, V) \equiv \text{push}(\Gamma); \alpha := \text{soln}^\dagger(V); \text{pop}()$$
$$\alpha := \text{opt}^\dagger(\Gamma, \Psi, V) \equiv \text{push}(\Gamma); \alpha := \text{opt}^\dagger(\Psi, V); \text{pop}()$$

Optimizations OPT2 and OPT3 both trade-off memory in favor of time. Our experiments indicate that this is a good trade-off, yielding significant performance gain at the cost of a small memory footprint increase, due to the modest complexity of SMT formulas involved. Indeed, we expect this to be the general case since enforcer logics are expected to be much simpler than the target systems.

Figure 5 shows optimized versions of `select`() and `select`*() – denoted `select`†() and `select`*† respectively – with all three optimizations applied. For `select`†(), we use $2 \times n$ contexts. Contexts $(1,1), \ldots, (n,1)$ are used for operations involving μ_i's, while contexts $(1,2), \ldots, (n,2)$ are used for operations involving C_{ϕ_i}'s. All contexts are initialized once via `init`() at startup. We use subscripts for push(), pop(), $\text{sat}^\dagger()$, $\text{soln}^\dagger()$, and $\text{opt}^\dagger()$ to indicate the context involved. For example, $\text{push}_{i,j}(F)$ indicates a push of formula F to the (i,j)-th context. For `select`*†(), we use $3 \times n$ contexts, initialized via `init`*().

4.2 Correctness of Optimizations

Correctness of `select`†. We argue about the correctness of procedure `select`† by showing its correspondence to procedure `select`. Assume that the contexts

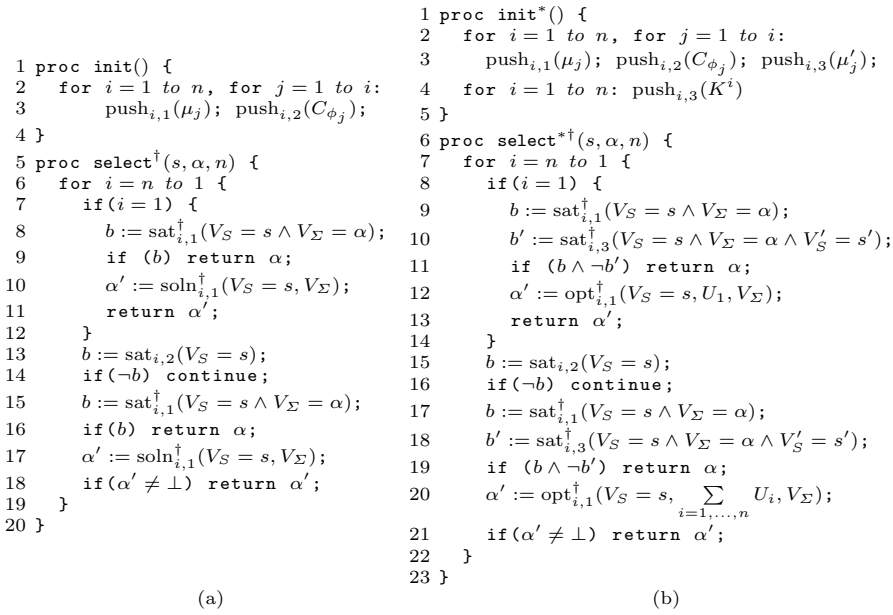

Fig. 5. (a) Pseudo-code for optimized utility-agnostic enforcer composition; (b) Pseudo-code for fully optimized utility-maximizing enforcer composition; $K^i \equiv (\sum_{j=1,\ldots,i} U_j) < (\sum_{j=1,\ldots,i} U'_j)$.

have been initialized via `init`. Then the lines of `select`† correspond to those of `select` as follows: (i) lines 7–12 correspond to lines 2–6 of `select`; (ii) lines 13–14 correspond to lines 7–8 of `select`; (iii) lines 15–16 correspond to lines 9–10 of `select`; and (iv) lines 17–18 correspond to lines 11–12 of `select`. Note that the loop invariant maintained by `select`† is that context $(i, 1)$ contains the formulas $\{\mu_j \mid 1 \leq j \leq i\}$, and context $(i, 2)$ contains the formulas $\{C_{\phi_j} \mid 1 \leq j \leq i\}$. To maintain this invariant, formulas pushed in each iteration are always popped before the next iteration, or before `select`† returns.

Correctness of `select`*†. We argue about the correctness of procedure `select`*† by showing its correspondence to procedure `select`*. Assume that the contexts have been initialized via `init`*. The the lines of `select`*† correspond to those of `select`* as follows: (i) lines 8–14 correspond to lines 2–6 of `select`*; (ii) lines 15–16 correspond to lines 7–8 of `select`*; (iii) lines 17–19 correspond to lines 9–10 of `select`*; and (iv) lines 20–21 correspond to lines 11–12 of `select`*. The loop invariant maintained by `select`*† is that context $(i, 1)$ contains the formulas $\{\mu_j \mid 1 \leq j \leq i\}$, context $(i, 2)$ contains the formulas $\{C_{\phi_j} \mid 1 \leq j \leq i\}$, and context $(i, 3)$ contains the formulas $\{\mu'_j \mid 1 \leq j \leq i\}$. To maintain this invariant, formulas pushed in each iteration are always popped before the next iteration, or before `select`*† returns.

5 Experimental Evaluation

To evaluate the proposed techniques, we created an indoor arena consisting of: (i) a laptop running Ubuntu 16.04; (ii) an Optitrack localization system with 8 cameras; (iii) two Parrot Travis mini-QCs; and (iv) two XBox gamepads. Optitrack sets up a coordinate system with six dimensions – X, Y, Z, roll, pitch and yaw – and multicasts the locations of the QCs in real-time at 120Hz over a wired LAN. The enforcer for each QC runs periodically on the laptop. It receives the locations from Optitrack via the LAN, and the user's command to the QC via the gamepad. Next, it uses the `select` algorithm, or one of its presented variants, to compute an appropriate actuation command and sends it to the Parrot QC via Bluetooth. We used existing open-source software for Bluetooth communication with the QCs, and with interface with the gamepads.

Thread Structure. Even though our running example is two-dimensional, for our experiments we implemented a three-dimensional enforcer. Specifically, the movement of a QC is specified via a pair of angles (α, α_1), where α is the same as in our example, and α_1 is an angle in the range $[-90, 90]$ w.r.t. to the horizontal plane. The enforcers are updated accordingly. The enforcement software running on the laptop for each QC consisted of the following periodic threads with fixed priorities: (i) T_{CL} receives and responds to commands from Optitrack; (ii) T_{FL} receives and records localization data from Optitrack; (iii) T_{EL} executes the `select` algorithm and computes the actuation; (iv) T_{RS} sends the command to the QC over Bluetooth; and (v) T_{Log} logs messages to a file. Table 2 shows the periods and priorities of all threads. The bottom 16 rows correspond to the T_{EL} thread executing different variants of `select`. Priorities were assigned rate-monotonically, i.e., shorter periods imply higher priorities. We made the periods as large as possible without sacrificing experiment quality. For example, further increasing the periods of T_{FL} and T_{EL} compromises unacceptably localization accuracy and enforcer responsiveness, respectively. Similarly, a higher period of T_{RS} causes the QCs to become unstable due to controller limitations.

Thread Scheduling. During experiments, all threads were bound to core 0, assigned their respective priorities, and the `SCHED_FIFO` scheduling policy. In this way, we achieve a single-core processing environment, and a pre-emptive fixed priority scheduler. Since we do not have precise worst-case execution time estimates for the threads, we do not analyze the system for schedulability. Instead, if a job misses its deadline (which always equals the task's period) the next job is delayed so that it starts at the next multiple of the task's period. For example, suppose a job of T_{RS} starts at time 10, and finishes at time 60. Since it misses its deadline (at time 50), the next job of T_{RS} starts at time 90. Clearly, we want to minimize deadline misses. While transient deadline misses are acceptable, a long series of deadline misses can cause the CPS to behave in an unsafe manner. In particular, a deadline miss by the enforcer thread reduces the effectiveness of the RA mechanism. We now present results demonstrating the effectiveness of the different enforcers presented earlier.

Enforcer Effectiveness. We implemented the tether and separation enforcers (E_1 and E_2) and evaluated them individually and in composition. Visually, both

Thread	Per	Prio	Flt-Time	#Jobs	DL-Miss	RespTime	ExecTime
T_{FL}	5	9	2358.22	530841	0	1.099/0.151/0.039	1.099/0.150/0.039
T_{CL}	50	2	2358.22	53059	26	250.994/0.146/4.281	0.101/0.014/0.008
T_{RS}	40	7	2358.22	64996	19	238.114/0.118/2.842	0.776/0.030/0.015
T_{Log}	1000	1	2358.22	2656	0	31.849/1.198/3.598	0.895/0.330/0.114
$\langle E_1 \rangle$	20	8	145.98	7743	572	83.626/5.579/7.709	39.164/5.415/7.453
$\langle E_1 \rangle^{\dagger}$	20	8	147.99	8397	0	7.196/0.323/0.439	3.553/0.322/0.434
$\langle E_1 \rangle^{*}$	20	8	197.11	8295	2564	33.910/9.798/10.237	32.722/9.558/9.984
$\langle E_1 \rangle^{*\dagger}$	20	8	353.03	19684	0	7.539/1.015/1.435	7.310/1.012/1.427
$\langle E_2 \rangle$	20	8	219.07	11338	660	45.079/5.752/7.515	42.942/5.611/7.329
$\langle E_2 \rangle^{\dagger}$	20	8	146.55	8368	0	2.732/0.361/0.480	2.732/0.361/0.480
$\langle E_2 \rangle^{*}$	20	8	188.14	8099	2327	36.035/9.940/10.264	34.776/9.705/10.018
$\langle E_2 \rangle^{*\dagger}$	20	8	234.75	13258	0	11.623/0.999/1.856	11.242/0.986/1.817
$\langle E_1, E_2 \rangle$	20	8	100.77	3479	2118	46.066/15.415/11.633	44.547/15.088/11.384
$\langle E_1, E_2 \rangle^{\dagger}$	20	8	101.23	5605	0	3.834/0.637/0.787	3.834/0.637/0.788
$\langle E_1, E_2 \rangle^{*}$	20	8	130.74	4396	2657	48.932/16.053/12.269	47.564/15.731/12.017
$\langle E_1, E_2 \rangle^{*\dagger}$	20	8	89.79	5009	0	13.640/1.815/2.579	13.157/1.796/2.537
$\langle E_2, E_1 \rangle$	20	8	55.61	2447	920	57.623/10.631/11.434	56.112/10.416/11.192
$\langle E_2, E_1 \rangle^{\dagger}$	20	8	81.71	4629	0	3.898/0.561/0.762	3.899/0.561/0.762
$\langle E_2, E_1 \rangle^{*}$	20	8	69.50	2795	1152	45.360/13.066/13.464	44.214/12.801/13.176
$\langle E_2, E_1 \rangle^{*\dagger}$	20	8	96.15	5315	0	16.940/2.656/3.770	16.371/2.586/3.647

Table 2. Response time and execution time measurements of various threads; Per= period in ms; Prio = priority; Flt-Time = total flight time (sec); DL-Miss = # of jobs that missed deadlines (i.e., response time > period); RespTime = response time (max/avg/stdev) in ms; ExecTime = execution time (max/avg/stdev) in ms; the superscript of $\langle \cdot \rangle$ denotes the variant of `select` used, e.g., $\langle E_1, E_2 \rangle^{*\dagger}$ means that `select`*† was used with the enforcer ordering $\langle E_1, E_2 \rangle$. Note that Flt-Time < Per × #Jobs since it does not include experiment times (at start and end) when the QCs are not airborne.

enforcers caused the QCs to behave as expected. When we flew a QC with only E_1, the enforcer prevented the QC from exiting the tether region even when the operator attempted to make it do so. When we flew both QCs together with E_2 running on each, the enforcers prevented the two QCs from crashing into each other even when the operators tried to make them crash. Next we flew QC_1 with both E_1 and E_2, given E_1 higher priority. We manually moved QC_2 closer to QC_1. We observed that this caused QC_1 to move further away till it reached the tether boundary. At this point, since E_1 has higher priority, QC_1 did not move away even if QC_2 was brought even closer to it. We ran a similar experiment by reversing the priorities of E_1 and E_2. This time, as expected, QC_1 continued to move away from QC_2 even if this caused QC_1 to violate the tether boundary.

Enforcer Efficiency. Each periodic execution of a thread is called a *job*. The *response time* of a job is the difference between its arrival and completion times. The *execution time* of a job is the amount of CPU time during which it actually executes (i.e., is not preempted by a higher priority thread). For each thread, we measured the response time and execution time of each job, and then computed their minimum, maximum, mean and standard deviation. Table 2 shows the results for the various threads in our system. As can be seen, our symbolic approach leads to quite efficient enforcers with execution times in the order of tens of milliseconds. In general, utility-maximizing enforcers are

slightly less efficient (and have more deadline misses) than their utility-agnostic counterparts. This is expected since maximizing utilities requires solving an optimization problem. Finally, `select`† and `select`*† deliver around 10x speedups in average execution times consistently, and sometimes almost 20x, e.g., $\langle E_2 \rangle^{\dagger}$ vs. $\langle E_2 \rangle$. Consequently, `select`† and `select`*† never miss deadlines while `select` and `select`* have frequent deadline misses. Overall, `select`*† is the best choice since it delivers optimal actuations in a timely manner. Finally, note that the execution times of enforcers have large standard deviations, indicating that overload conditions occur regularly. However, as can be seen by the modest number of deadline misses, ZSRM is able to handle overloads gracefully.

6 Conclusion

We addressed the problem of combining multiple runtime enforcers for a CPS that may produce conflicting actuation commands. We proposed an algorithm that resolves such conflicts at runtime by considering a design-time prioritization of the enforcers. Specifically, our algorithm produces an action that satisfies the maximum number of high-priority enforcers, ignoring the low-priority ones as needed. Our approach also supports a notion of utility-maximization that enables us to implement enforcers that yield the best possible actuation under any given situation. To enable efficient implementations, needed to meet tight schedulability and periodicity constraints, we encode the enforcers symbolically as SMT formulas, and compute their combination via incremental SMT solver operations. Experiments on a CPS testbed involving geo-fencing and collision avoidance among flying minidrones demonstrates the effectiveness of our approach. We see at least two important areas of future work: (i) supporting "skipping" of enforcement actions necessary due to extreme overload conditions; and (ii) supporting multiple enforcer threads operating in the same system.

Acknowledgment. Copyright 2017 Carnegie Mellon University[1].

References

1. NSF Definition of Cyber-Physical Systems, `https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503286`

2. Bak, S., Johnson, T., Caccamo, M., Sha, L.: Real-Time Reachability for Verified Simplex Design. In: Proceedings of the 35th Real-Time Systems Symposium (RTSS '14) (2014)
3. Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E.: Enforceable Security Policies Revisited. ACM Transactions on Information and System Security (TISSEC) (2013)
4. Bjørner, N., Phan, A., Fleckenstein, L.: νZ - An Optimizing SMT Solver. In: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '15). Lecture Notes in Computer Science (2015)
5. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: TACAS (2015)
6. deNiz, D., Lakshmanan, K., Rajkumar, R.: On the Scheduling of Mixed-Criticality Real-Time Task Sets. In: Proceedings of the 30th Real-Time Systems Symposium (RTSS '09) (2009)
7. Falcone, Y., Mounier, L., Fernandez, J.C., Ricier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design (FMSD) (2011)
8. Fong, P.: Access Control By Tracking Shallow Execution History. In: IEEE Security and Privacy (2004)
9. Havelund, K., Goldberg, A.: Verify your runs. In: VSTTE (2005)
10. Havelund, K., Rosu, G.: Monitoring Programs Using Rewriting. In: Proceedings of the 16th International Conference on Automated Software Engineering (ASE '01) (2001)
11. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS '99) (1999)
12. Leucker, M., Schallhart, C.: A brief account of runtime verification. In: JLAP (2008)
13. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. International Journal of Information Security (IJIS) (2005)
14. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08). Lecture Notes in Computer Science (2008)
15. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: monitoring embedded systems. Innovations in Systems and Software Engineering (ISSE) (2013)
16. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Timo, O.: Runtime Enforcement of Timed Properties. In: Proceedings of the 2nd International Conference on Runtime Verification (RV '12) (2012)
17. Pinisetty, S., Roop, P., Smyth, S., Tripakis, S., Hanxleden, R.: Runtime enforcement of reactive systems using synchronous enforcers (2016), coRR abs/1612.05030
18. Pinisetty, S., Tripakis, S.: Compositional Run-time enforcement. In: NFM (2016)
19. Schneider, F.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) (2000)
20. Seto, D., Krogh, B., Sha, L., Chutinan, A.: The simplex architecture for safe online control system upgrades. In: Proceedings of the American Control Conference (1998)
21. Viswanatha, M.: Foundations for the run-time analysis of software systems. Ph.D. thesis, University of Pennsylvania (2000)
22. Wu, M., Zeng, H., Wang, C.: Synthesizing runtime enforcer of safety properties under burst error. In: NFM (2016)