Safety Enforcement for the Verification of Cyber-Physical Systems

Dionisio de Niz

May11, 2018

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

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.

DM18-0630

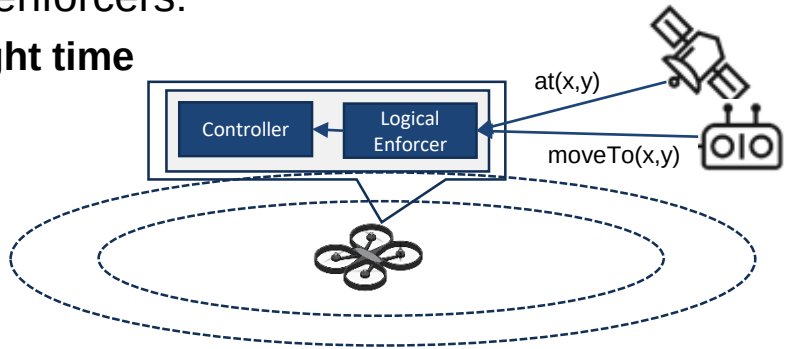
Verification of Cyber-Physical Systems

Challenge:

- Traditional Verification Does Not Scale
- Unpredictable Algorithms like machine learning (Autonomous CPS)
- Timely Interaction with Environment: correct **actions** at correct **time**

Solution:

- Add **simpler (verifiable)** runtime enforcer to make algorithms predictable
- Formally: specify, verify, and compose multiple enforcers:
  - Enforcer **intercepts/replaces** unsafe action at **right time**



Timeless Logical Model

Focus on Correct Values

Statespace

- $S = \{s\}$

Safe states

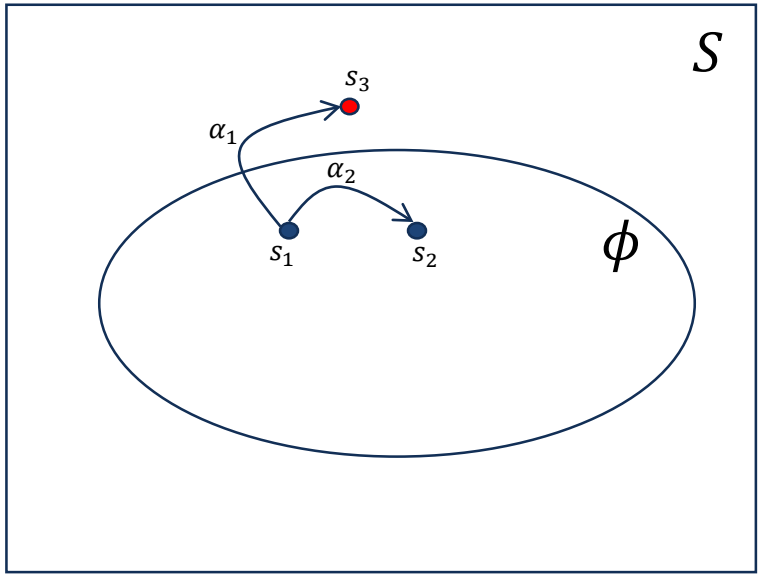
- $\phi \subseteq S$

Transitions triggered by actions

- Transition: $R(\alpha) \subseteq S \times S$
- Destination state: $R(\alpha, s) = \{s' | (s, s') \in R(\alpha)\}$

Safe actions from a state

- $SafeAct^*(s) = \{\alpha | R(\alpha, s) \in \phi\}$



Incorporating Time

Statespace

- $S = \{s\}$

Safe states

- $\phi \subseteq S$

Transitions triggered by actions

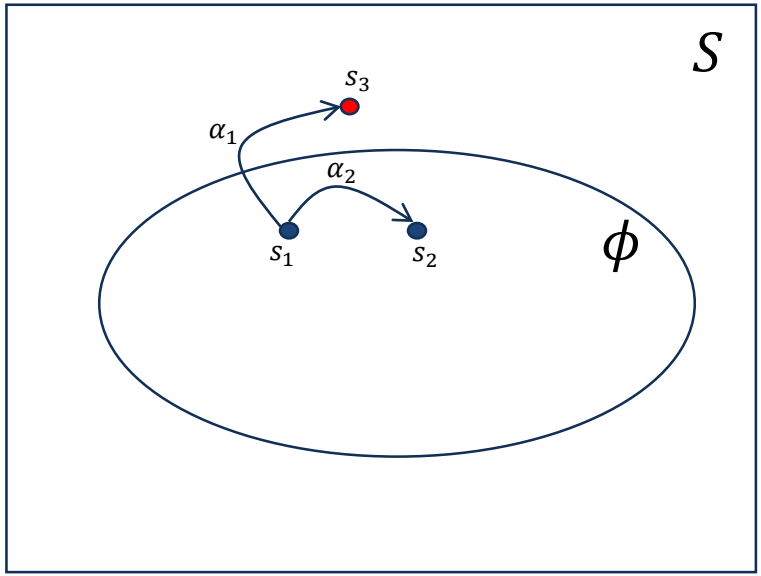
- Transition: $R(\alpha) \subseteq S \times S$
- Destination state: $R(\alpha, s) = \{s' | (s, s') \in R(\alpha)\}$

Safe actions from a state

- $SafeAct^*(s) = \{\alpha | R(\alpha, s) \in \phi\}$

Periodic Actions

- Occur every period with length P.
- $R_P(\alpha) \subseteq S \times S$
- Destination reached at P: $R_P(\alpha, s) = \{s' | (s, s') \in R_P(\alpha)\}$



Incorporating Inertia

Statespace

- $S = \{s\}$
- $\phi \subseteq S$

Periodic actions

- Transition: $R_P(\alpha) \subseteq S \times S$
- Destination state: $R_P(\alpha, s) = \{s' | (s, s') \in R(\alpha)\}$

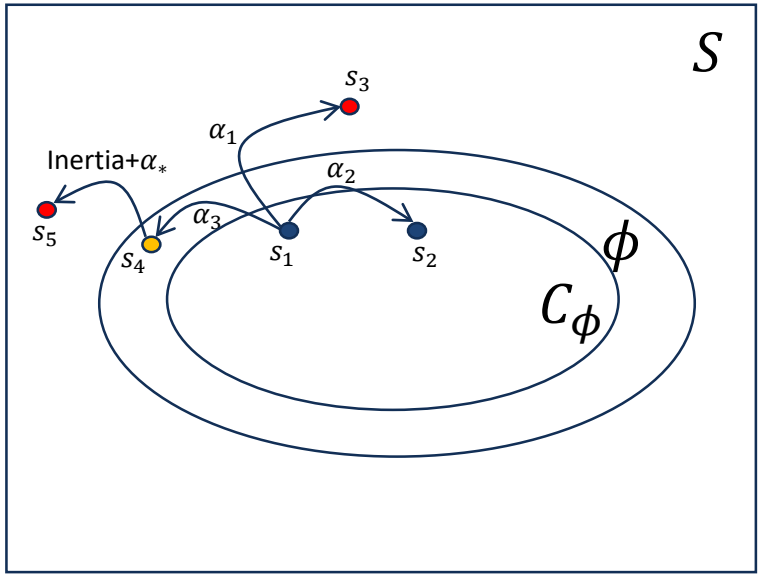
Identify states too close to safety border

- Inertia lead to unsafe state even if enforced
- Enforceable states:

$$C_\phi = \{s | \exists \alpha : R_P(\alpha, s) \in C_\phi\}$$

Safe actions:

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



Logical Enforcer

Statespace & actions

- $S = \{s\}, \phi \subseteq S$
- $R_P(\alpha) \subseteq S \times S;\ R_P(\alpha, s) = \{s' | (s, s') \in R(\alpha)\}$

Enforceable states

- $C_\phi = \{s | \exists \alpha : R_P(\alpha, s) \in C_\phi\}$

Safe actions:

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

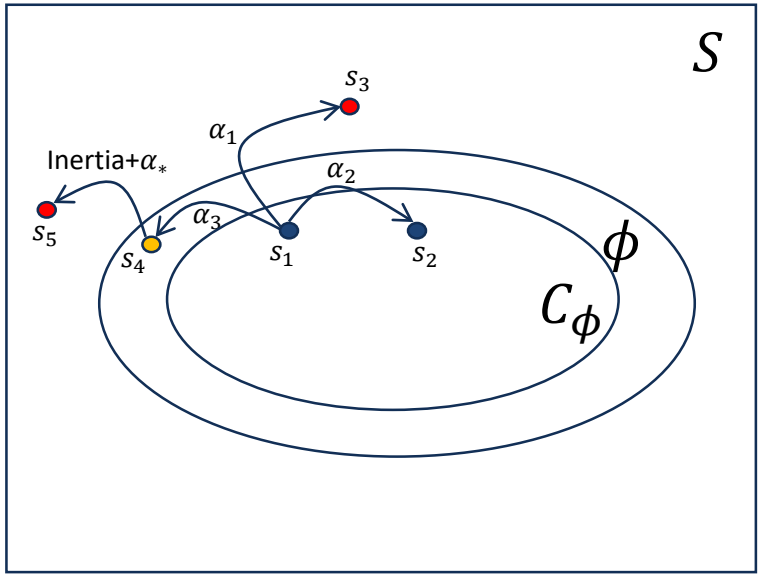
Logical Enforcer: $E = (P, C_\phi, \mu)$

- Set of safe actions:

$$\mu(s) \subseteq SafeAct(s)$$

- Monitor and enforce safe action:

$$\tilde{\alpha} = \begin{cases} \alpha, & \alpha \in \mu(s) \\ pick(\mu(s)), & otherwise \end{cases}$$

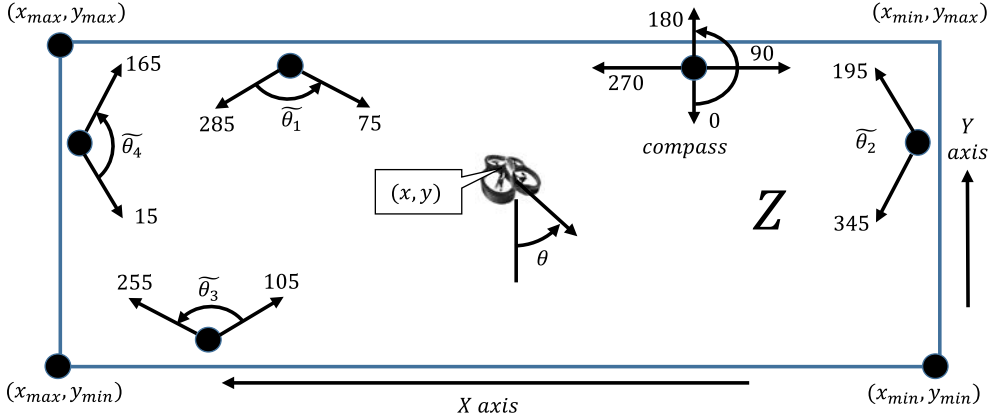


Verification with SMT (Z3)

Drone Example

Statespace

- $S = \{s = (x, y, \theta)\}$
- $\phi = \{(x, y, \theta) | \ (x, y) \in Z\}$



Enforceable states

- Max distance in one period P: δ_P
- Max distance in opposite direction of enforcement: δ_B
- $C_\phi = \{(x, y, \theta) | (x + \delta_B, y + \delta_B) \in Z \wedge (x - \delta_B, y - \delta_B) \in Z\}$

Action: constant speed at angle θ

Enforcement: $\tilde{\theta} = \begin{cases} \tilde{\theta} \in \tilde{\theta}_1, \ if \ Y_{max} - y \leq \delta_B + \delta_P \\ \tilde{\theta} \in \tilde{\theta}_2, \ if \ x - X_{min} \leq \delta_B + \delta_P \\ \tilde{\theta} \in \tilde{\theta}_3, \ \ if \ y - Y_{min} \leq \delta_B + \delta_P \\ \tilde{\theta} \in \tilde{\theta}_4, if \ X_{max} - x \leq \delta_B \theta + \delta_P \\ \qquad \theta, otherwise \end{cases}$

Are We Done Yet?

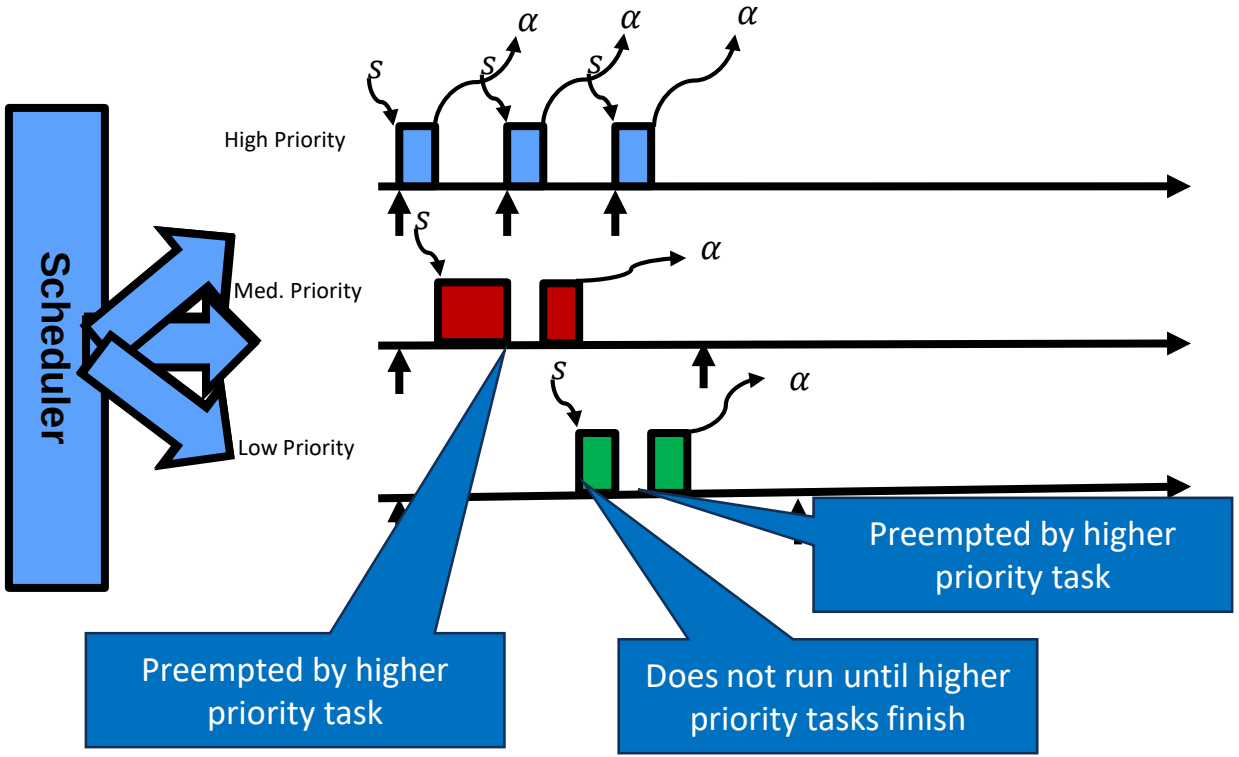
Timing Assumption:

- Unverified software finishes execution and enforcer evaluates output every P period.
- Software is guaranteed to finish executing by the next period (schedulable)
  - Unverified software executes for less than its Worst-Case Execution Time (WCET)
  - Other software running also executes for less than its WCET
  - Schedulability analysis successful

What can go wrong?

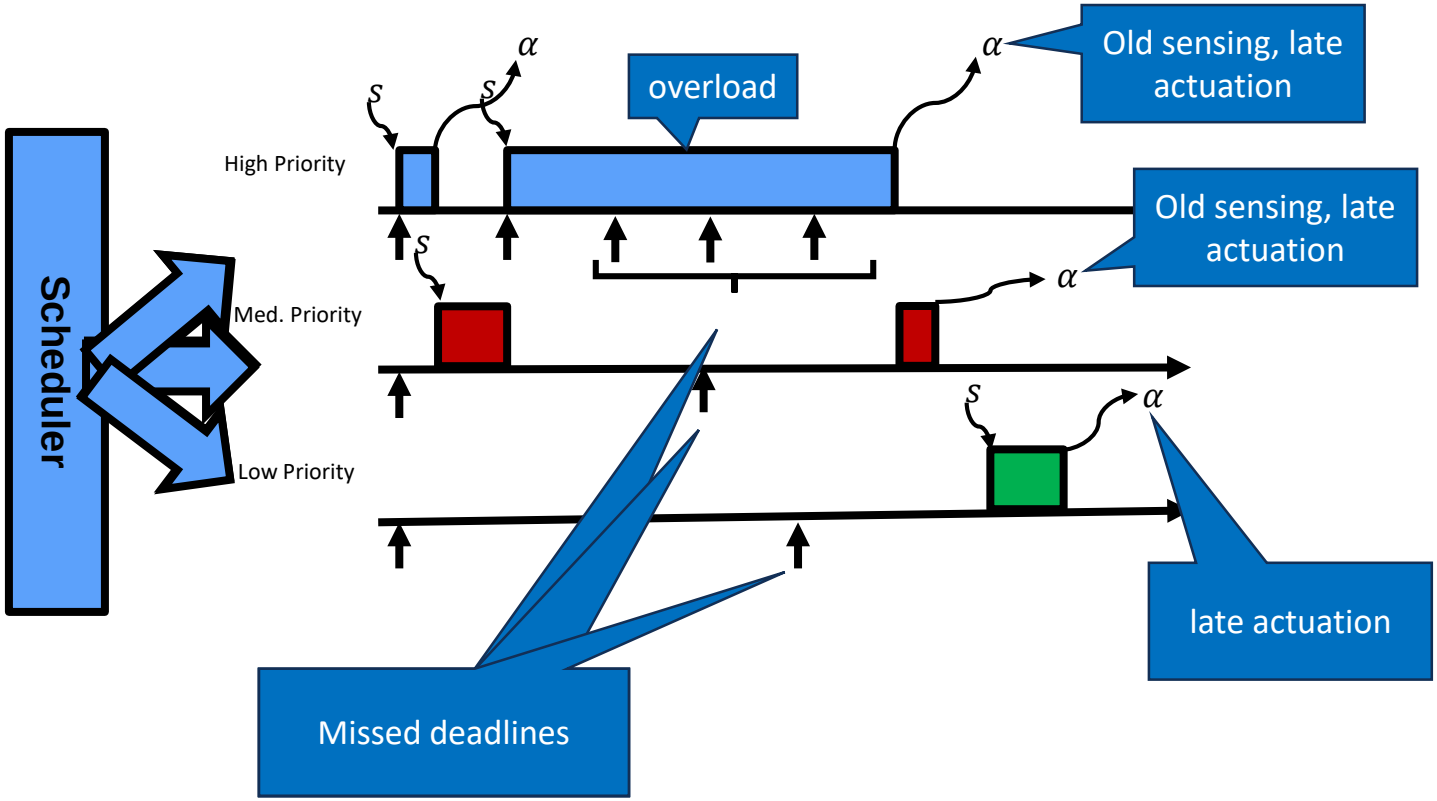
- Unbounded preemption
  - High priority software executes longer than WCET
  - Can make other software miss deadlines: late actions with old sensing
- Unbounded execution
  - Software executes longer than WCET
  - Misses its own deadline: Does **NOT** produce an output that can be evaluate by enforcer: late action + old sensing
    - **Inertia** takes it to **unsafe state**

Fixed-Priority Scheduling + Rate Monotonic



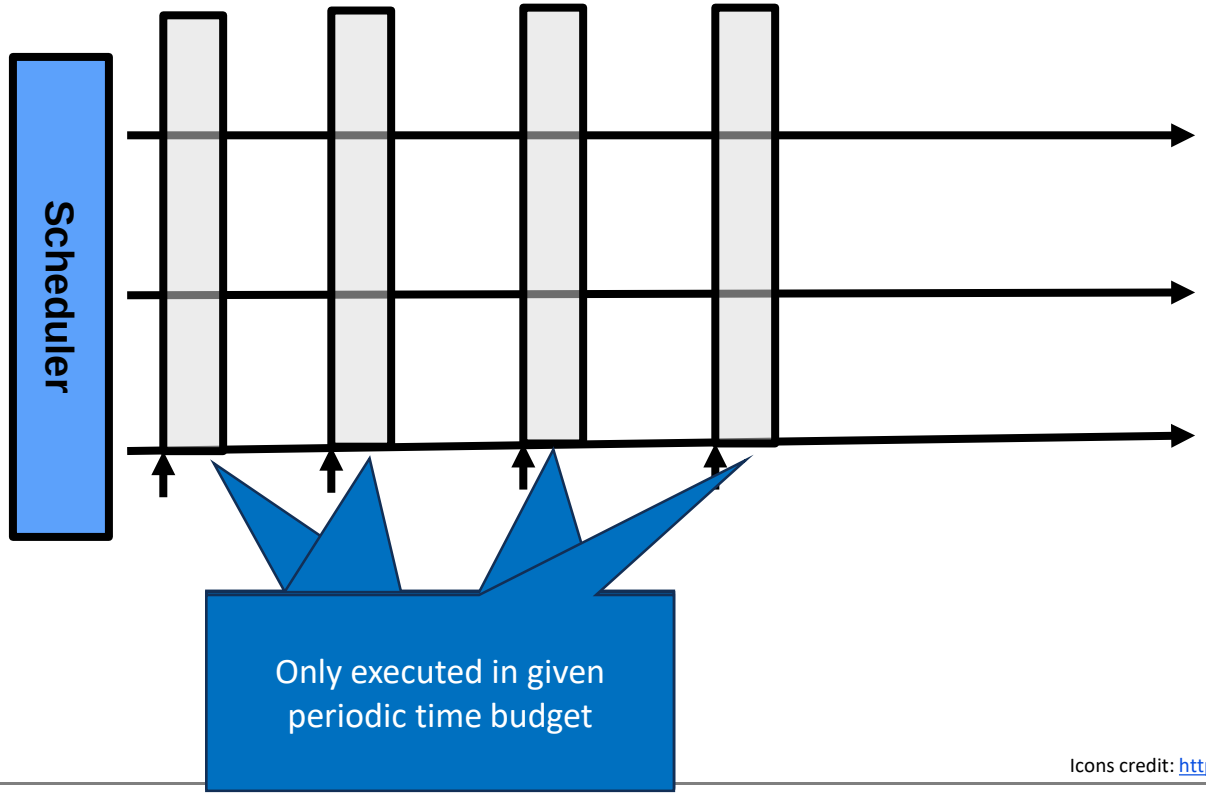
Icons credit: http://www.doublejdesign.co.uk

Overload -> old sensed data + late actuation



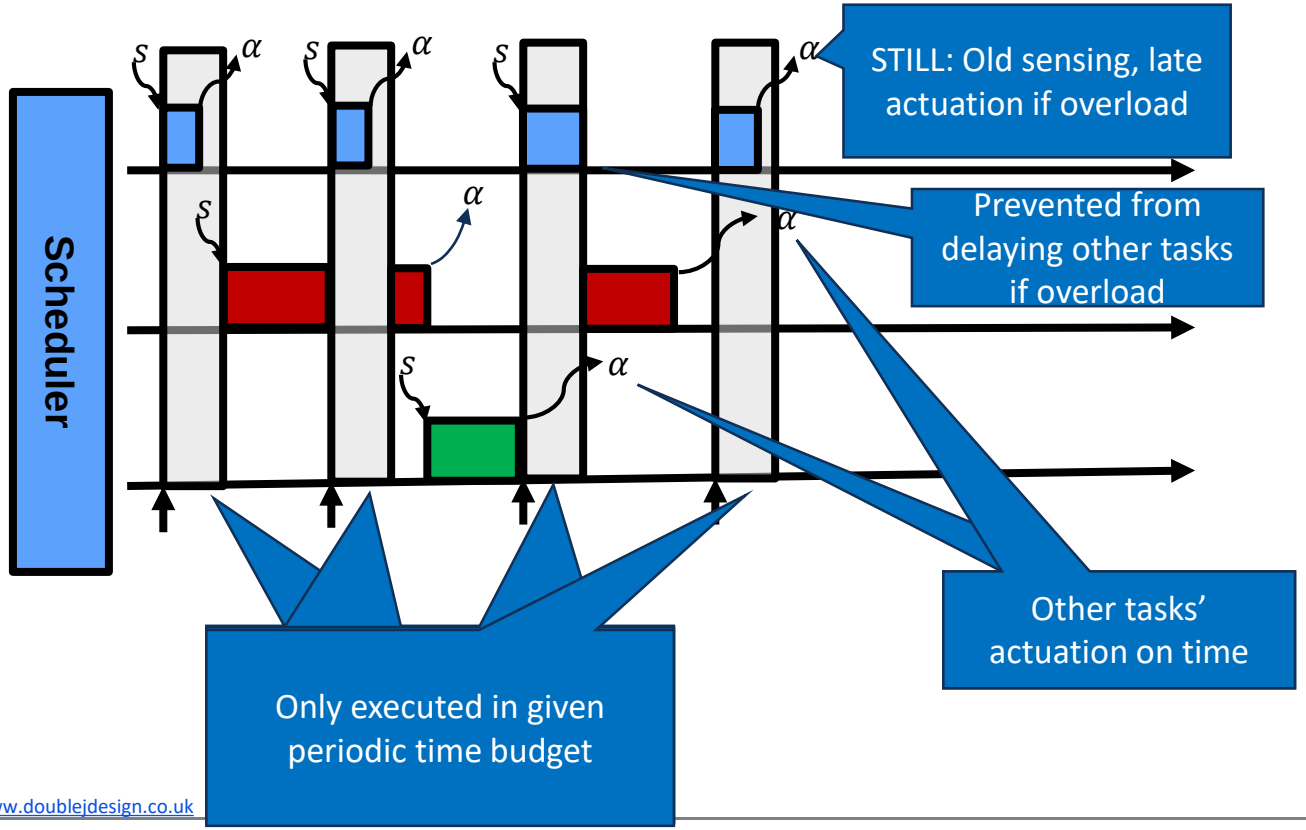
Icons credit: http://www.doublejdesign.co.uk

Unbounded preemption
Solution: Enforce timing budgets (timing enforcement)



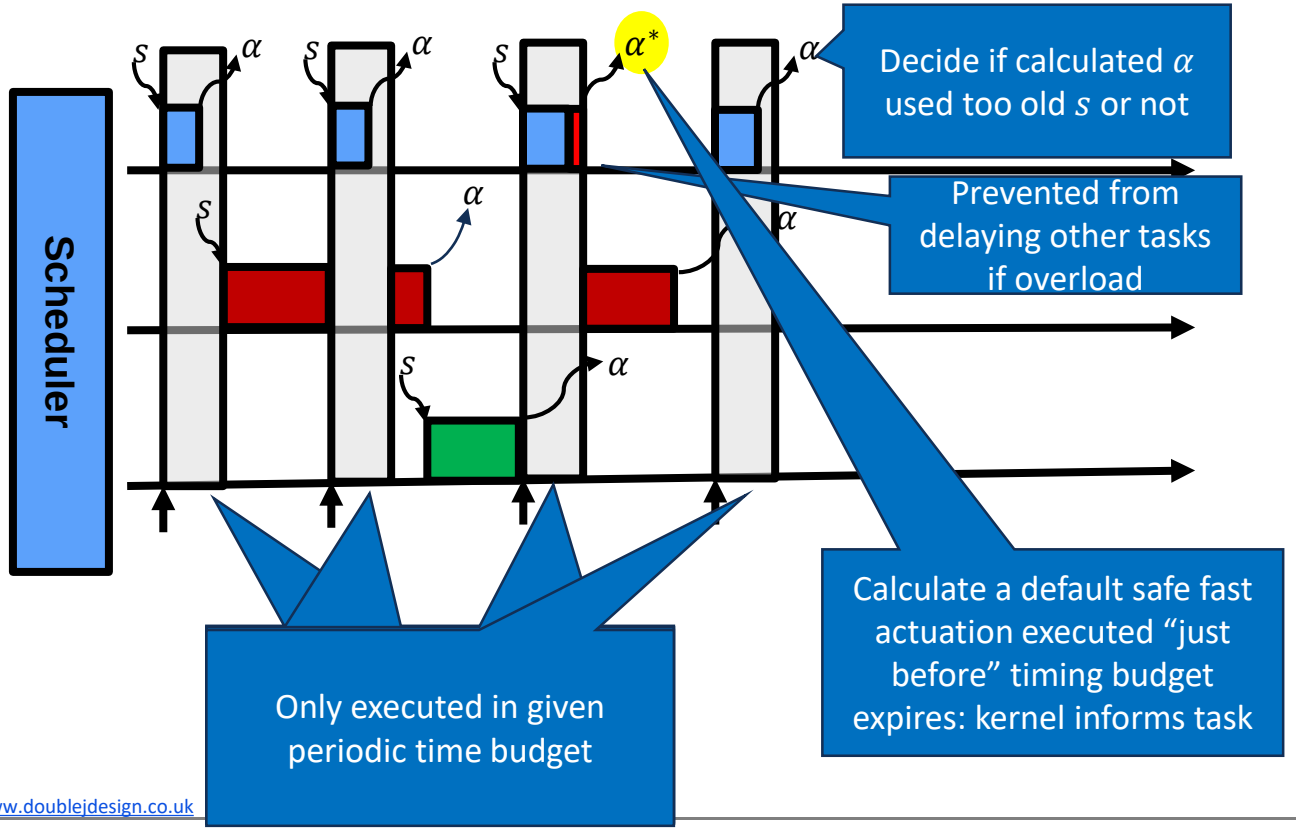
Icons credit: http://www.doublejdesign.co.uk

Unbounded preemption
Solution: Enforce timing budgets (timing enforcement)



Icons credit: http://www.doublejdesign.co.uk

Unbounded Execution:
Solution: safe actuation on timing enforcement



Icons credit: http://www.doublejdesign.co.uk

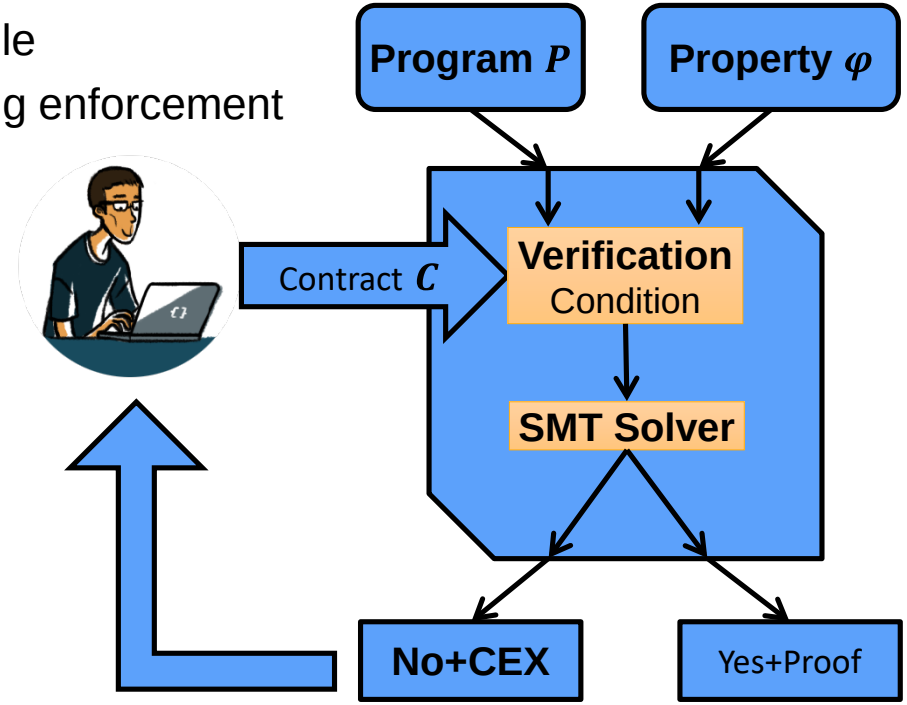
Who Verifies the Temporal Enforcer?

Implementation:

- Extension to Mixed-Criticality Zero-Slack Rate Monotonic Scheduler
- Implemented in C as a loadable kernel module
- Uses timer and clocks to implement the timing enforcement

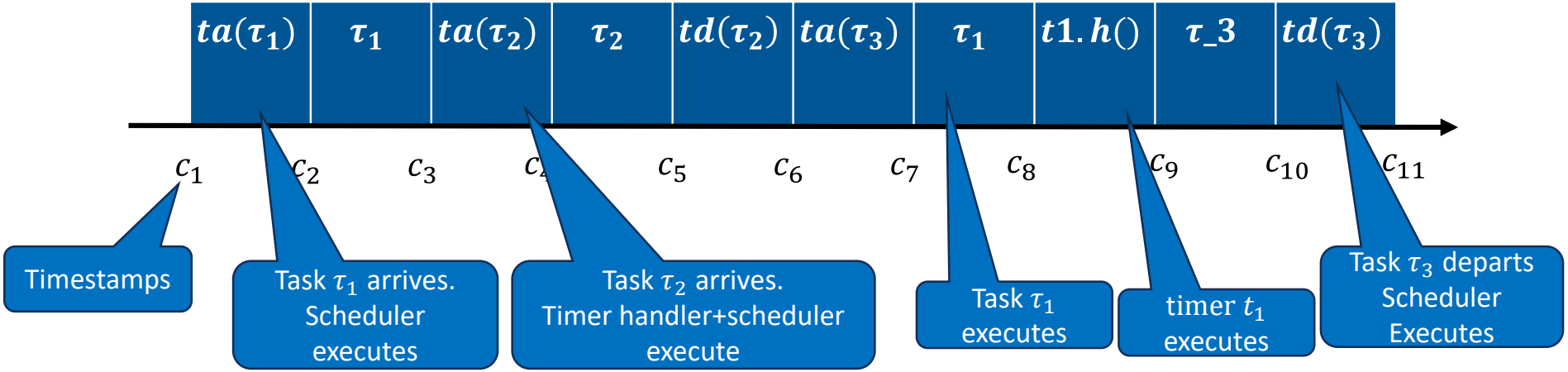
Verification of C code

- With FRAMA-C (auto-active verification)
- C code simplified to enable verification
- FRAMA-C encoding of timing model



Execution Timeline Model

Newtonian Time: flows monotonically, dense time

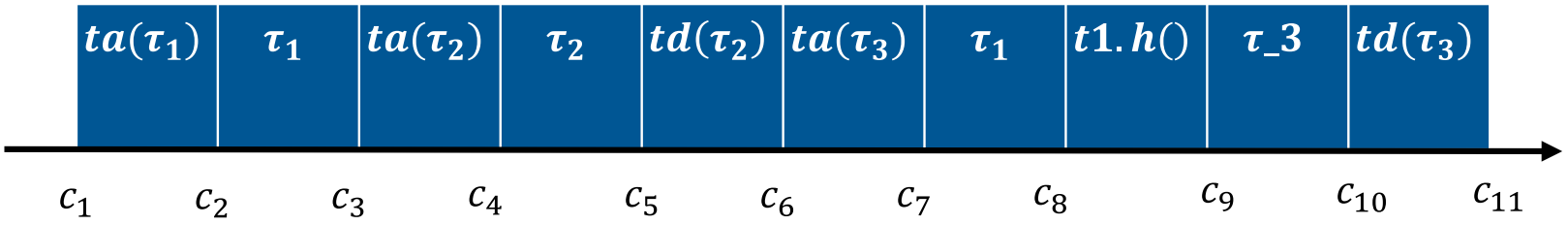


Execution $\pi = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$: α_i: action(enforcer|task)

State $s_i = (c_i, r_i, a_i) = (c_i$:timestamp, r_i:readyq, a_i:timerq)

$sched(s_i, \tau)$ means that τ is at the head of r_i

Task CPU Usage



$C(\pi, \tau_i)$ = total cpu usage by task τ_i over execution π

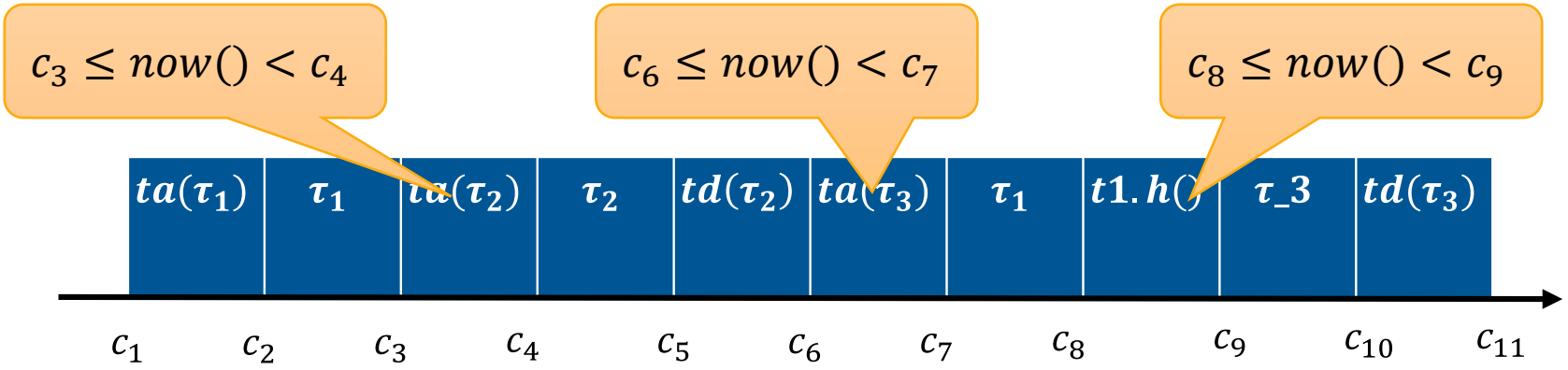
Add up durations of all the transitions labeled by τ_i

$C(\pi, \tau_1) = (c_3 - c_2) + (c_8 - c_7)$ *STOPPED by timer t_1

$C(\pi, \tau_2) = (c_5 - c_4)$

$C(\pi, \tau_3) = (c_{10} - c_9)$

Linking Time Model to Code



When task τ_1 starts executing

```
if (tau_1 ==readyq.head)
   tau_1.start = now()
```

When task τ_1 preempted/stopped

```
if (tau_1 ==readyq.head)
   tau_1.usage = now() − tau_1.start
```

Proving Tasks Never Exceed Budgets

Each task τ_i has a time budget $B(\tau_i)$

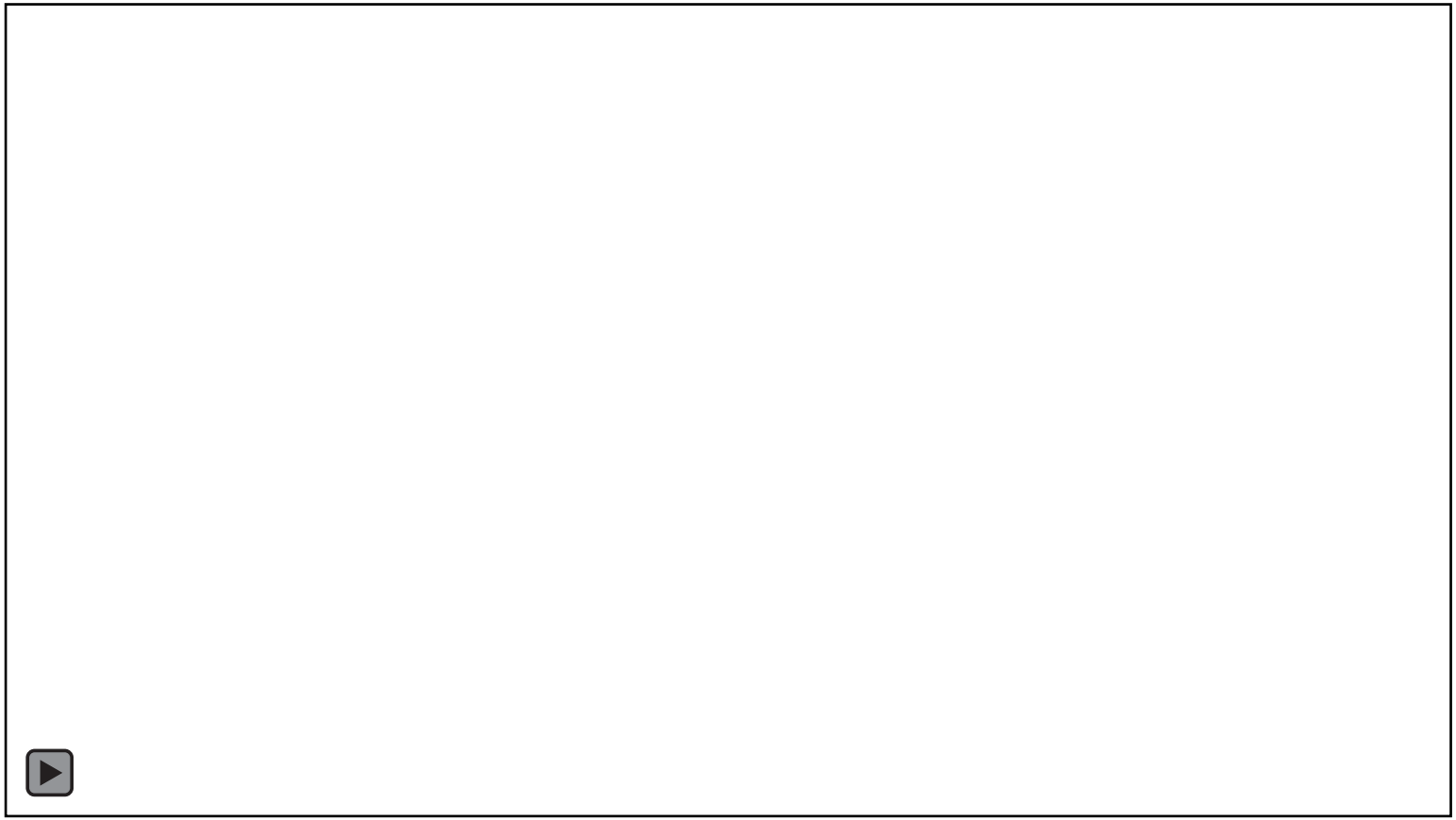
Definition 3 (Timer). *We say that a budget timer is always properly activated for a thread τ, denoted $Timer(\tau)$, if at the end of each execution $\pi = s_1 \xrightarrow{\alpha_1} s_2 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$ such that $\alpha_{n-1} \in F \wedge sched(s_n, \tau)$ there exists an active timer $t \in a_n$ such that $t.c \leq c_n + B(\tau) - \tau.usage$.*

Theorem 2. *For any thread τ, if $Timer(\tau)$, then τ never exceeds its budget, i.e., at the end of each execution π, we have $C(\pi, \tau) \leq B(\tau)$.*

Enforcer C implementation:

- Added ghost variables to follow time
- Verified correct “rounding” of timestamps (ticks to ns)
- Correct programming of timers
- Correct usage calculation

Drone Demo



Conclusions

Verification of Autonomous Systems is Challenging

- Verification technologies does not scale
- Behavior not fully defined at design time (machine learning)

Enforcement-Based Verification Makes it Possible & Practical

- Logical enforcement
- Temporal enforcement

Presented the formalization of enforcement-based safety verification

- Logical formulation: time-aware logic
- Temporal enforcement to deal with CPS inertia

Presented implementation in drone system