Type-Based Verification of Delegated Control in Hybrid Systems

Eduard Kamburjan[1]
Michael Lienhardt[2]
[1]University of Oslo
[2]ONERA

ABS'23, 04.10.23

Motivation

Modern Cyber-Physical Systems require Distributed Control and Cloud Systems

- Edge devices in IoT
- Digital Twins and Industry 4.0
- Networked devices, e.g., autonomous trains

Engineers can build these devices – but how do we verify them?

CPS verification, program verification and cloud modeling barely intersect.

Modeling - Hybrid Active Objects

UNIVERSITY OF OSLO

Abstract Behavioral Specification

(1) Modeling, (2) Specification and Verification, and (3) Simulation of Modular Systems with Active Objects.

Active Objects = objects
+ actor concurrency model
+ condition synchronization
+ explicit time

HABS: Hybrid ABS

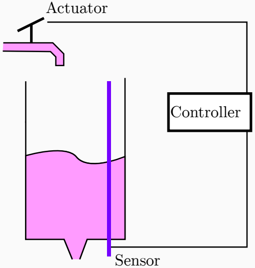
UNIVERSITY OF OSLO

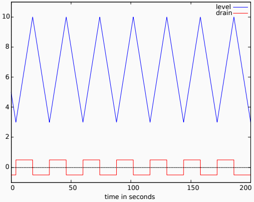
_A_bstract _B_ehavioral _S_pecification

(1) Modeling, (2) Specification and Verification, and (3) Simulation of Modular Hybrid Systems with Active Objects.

Hybrid Active Objects = objects
+ actor concurrency model
+ condition synchronization
+ explicit time
+ continuous behavior

Example: Water Tank






```
class CSingleTank(Real inVal){
    physical{
        Real lvl = inVal : lvl’ = flow;
        Real flow = -0.5 : flow’ = 0;
    }
    { this!low(); }
    Unit low(){
        await diff lvl <= 3 & flow <= 0;
        flow =  0.5; this!up();
    }
    Unit up(){
        await diff lvl >= 10 & flow >= 0;
        flow = -0.5; this!low();
    }
}
```

Is $3 \leq$ `lvl` ≤ 10 an invariant (if $3 \leq$ `inVal` ≤ 10)?

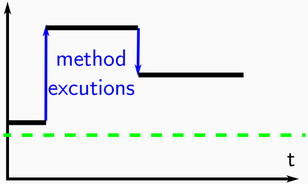
Verification - Post-Regions

Object Invariants

Proof Obligations with Dynamic Logic

In discrete systems, an object invariant I can be checked *modularly* with dynamic logic by showing that every method preserves I.

$$I \rightarrow [s]I \qquad \text{Proof Obligation for Discrete Systems}$$

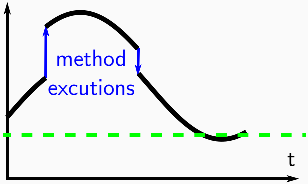


First, we need a logic for hybrid systems.

Proof Obligations with Dynamic Logic

In discrete systems, an object invariant I can be checked *modularly* with dynamic logic by showing that every method preserves I.

$I \rightarrow [\mathtt{s}]I$ Proof Obligation for Discrete Systems



First, we need a logic for hybrid systems.

UNIVERSITY OF OSLO

Differential Dynamic Logic

A logic for (algebraic) hybrid programs:

$$\phi ::= \forall x.\ \phi \mid \phi \vee \phi \mid \neg\phi \mid \ \dots \ \mid [\alpha]\phi$$

$$\alpha ::= ?\phi \mid \mathtt{v} := t \mid \mathtt{v} := * \mid \{\mathtt{v}' = f(\mathtt{v})\&\phi\} \mid \ \dots$$

UNIVERSITY OF OSLO

Differential Dynamic Logic

A logic for (algebraic) hybrid programs:

$$\phi ::= \forall x.\ \phi \mid \phi \vee \phi \mid \neg\phi \mid \ \dots \ \mid [\alpha]\phi$$
$$\alpha ::= ?\phi \mid \mathtt{v} := t \mid \mathtt{v} := * \mid \{\mathtt{v}' = f(\mathtt{v})\&\phi\} \mid \ \dots$$

Example

Set a variable to 0, let it raise with slope 1 while it is below 5 and discard all runs where it is above 5.

$$\big[\mathtt{x} := 0; \{\mathtt{x}' = 1\&\mathtt{x} \leq 5\}; ?\mathtt{x} \geq 5\big]\mathtt{x} \doteq 5$$

This formula is valid.

Preliminaries

- We assume that every method starts with an **await diff** statement. If it does not, add **await diff** `true`.
- The leading guard of a method `m` is denoted $\mathit{trig}_{\mathtt{m}}$.
- Only `Real` variables are manipulated.
- Weak negation is denoted $\tilde{\neg} e_1 \geq e_2 \iff e_1 \leq e_2$

Safety

An object is safe w.r.t. some formula ϕ, if its state is a model for ϕ (a) whenever a method starts and (b) whenever time advances.

For the beginning, we assume that all **await** are leading and no **get** or **duration** occur.

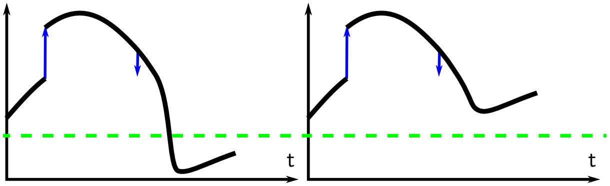
Theorem

Let C be a class with dynamics ode. Each object of C is safe w.r.t. inv and precondition pre if for every method the following holds:

$$\mathsf{inv} \to \left[?trig_{\mathtt{m}}; \mathsf{trans}(\mathtt{s_m})\right]\left(\mathsf{inv} \land [\mathsf{ode}\&\mathsf{true}]\mathsf{inv}\right)$$

And additionally for the constructor:

$$\mathsf{pre} \to \left[\mathsf{trans}(\mathtt{s_{init}})\right]\left(\mathsf{inv} \land [\mathsf{ode}\&\mathsf{true}]\mathsf{inv}\right)$$



Lemma

Let `C` be safe w.r.t. inv. Let $\texttt{C}^+$ be `C` with an added method and $\texttt{C}^-$ be `C` with a method removed.

- $\texttt{C}^-$ is safe
- To show safety of $\texttt{C}^+$, only the new method must be verified

- Very modular
- Imprecise: do not use additional information provided the structure
- Cannot verify our water tank
- Can verify self-stabilizing systems without control cycle

Locally Controlled Regions

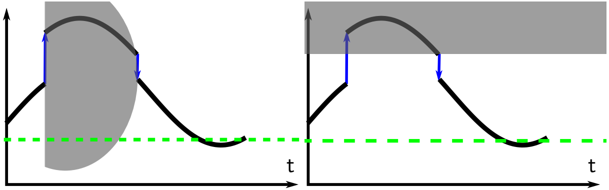
UNIVERSITY OF OSLO

Theorem

Let $\mathtt{C}$ be a class with dynamics $\mathtt{ode}$. For each method $\mathtt{m}$ let $\mathsf{CM}_\mathtt{n}$ be the set of methods which are guaranteed to called in every execution. Each object of $\mathtt{C}$ is safe w.r.t. inv if for every method $\mathtt{m}$ the following holds:

$$\mathsf{inv} \rightarrow [?\mathit{trig}_\mathtt{m}; \mathsf{trans}(\mathtt{s}_\mathtt{m})] \left(\mathsf{inv} \wedge \left[\mathtt{ode} \& \bigwedge_{\mathtt{m}' \in \mathsf{CM}_\mathtt{m}} \tilde{\neg} \mathit{trig}_{\mathtt{m}'} \right] \mathsf{inv} \right)$$

And analogously for the constructor.



Locally Controlled Regions

```
class LocalTank(){
physical{Real lvl = 5 : lvl' = flow; Real flow = -0.5 : ...}
{ this!low(); }
Unit low(){await diff lvl <= 3; flow = 0.5; this!up();}
Unit up(){await diff lvl >= 10; flow = -0.5; this!low();}
}
```

$$\mathsf{inv} \rightarrow [?\texttt{lvl <= 3}; \texttt{flow := 0.5}](\mathsf{inv} \land [\texttt{lvl' = flow}\&\texttt{lvl <= 10}]\mathsf{inv})$$

Locally Controlled Regions

```
class LocalTank(){
  physical{Real lvl = 5 : lvl' = flow; Real flow = -0.5 : ...}
  { this!timed(); }
  Unit timed(){
      await duration(1,1);
      if(lvl >= 9.5) -flow = 0.5;
      if(lvl <= 3.5) flow = 0.5;
      this!timed();
  }
}
```

```
class LocalTank(){
  physical{Real lvl = 5 : lvl' = flow; Real flow = -0.5 : ...}
  { this!timed(); }
  Unit timed(){
      await duration(1,1);
      if(lvl >= 9.5) -flow = 0.5;
      if(lvl <= 3.5) flow = 0.5;
      this!timed();
  }
}
```

What about systems that decouple control and dynamics?

UNIVERSITY OF OSLO

```
class Tank(Real inVal){              /*@ requires 3.5 <=inVal<= 9.5 @*/
 physical{ Real lvl' = flow; ...} /*@ invariant 3 <=lvl<= 10 && -0.5 <=flow<= 0.5 @*/
 Unit localCtrl(){
  if(lvl <= 3.5) flow =  0.5;
  if(lvl >= 9.5) flow = -0.5;}}
```

Limitations - External Control


```
class Tank(Real inVal){              /*@ requires 3.5 <=inVal<= 9.5 @*/
 physical{ Real lvl' = flow; ...} /*@ invariant 3 <=lvl<= 10 && -0.5 <=flow<= 0.5 @*/
 Unit localCtrl(){
  if(lvl <= 3.5) flow =  0.5;
  if(lvl >= 9.5) flow = -0.5;}}
```

Need to consider other objects to compute post-regions – is the tank controlled?

```
class Controller(){
  Unit timer(Tank t, Int time){
    await duration(1);
    if(time != 0) {
      t!localCtrl();
      this.timer(t, time - 1);}}}
```

UNIVERSITY OF OSLO

```
class Tank(Real inVal){                /*@ requires  3.5 <=inVal<= 9.5  @*/
 physical{ Real lvl' = flow; ...} /*@ invariant 3 <=lvl<= 10 && −0.5 <=flow<= 0.5 @*/
 Unit localCtrl(){
  if(lvl <= 3.5) flow =  0.5;
  if(lvl >= 9.5) flow = -0.5;}}
```

Need to consider other objects to compute post-regions – is there always one controller?

```
class Mobile {
 Unit run() {
  Tank t = new Tank(4);
  Controller c = new localCtrl(); Fut<Unit> f = c.timer(t, 40);
  await duration(40) & f;
  c = new Controller(); f = c.timer(t, -1); }}
```

Limitations - External Control

Subtle timing issues can violate specification

```
class Controller(){
  Unit timer(Tank t, Int time){
    await duration(1);
    if(time != 0) {
      t!localCtrl();
      this.timer(t, time - 1);}}}

class Mobile {
 Unit run() {
  Tank t = new Tank(4);
  Controller c = new localCtrl(); Fut<Unit> f = c.timer(t, 40);
  await duration(40) & f;
  c = new Controller(); f = c.timer(t, -1); }}
```

Limitations - External Control

Subtle timing issues can violate specification

```
class Controller(){
  Unit timer(Tank t, Int time){
    if(time != 0) {
      await duration(1);
      t!localCtrl();
      this.timer(t, time - 1);}}}

class Mobile {
 Unit run() {
  Tank t = new Tank(4);
  Controller c = new localCtrl(); Fut<Unit> f = c.timer(t, 40);
  await duration(40) & f;
  c = new Controller(); f = c.timer(t, -1); }}
```

External Control

Type-Checking External Control

Challenge

- Post-region cannot be computed locally
- External control must be globally ensured
- Obligation for external control can be delegated

Overview Solution

- Controllee gets specification: Temporal, externally controlled post-region (ECP)
- Controller gets specification: controlled objects
- Type system checks
  - For every object with an ECP there is always a controller
  - Each controller respects the ECP specification of its controllee
- From the controllee-view, post-region-based verification is unchanged

ECP Specification

```
class Tank(Real inVal){              /*@ requires 3.5 <=inVal<= 9.5 @*/
 physical{ Real lvl' = flow; ...} /*@ invariant 3 <=lvl<= 10 && −0.5 <=flow<= 0.5 @*/
 /*@ timed_requires 1 @*/
 Unit localCtrl(){
  if(lvl <= 3.5) flow =  0.5;
  if(lvl >= 9.5) flow = -0.5;}}

class Controller(){
 /*@ time_control: t.localCtrl = [1, 1] @*/
 Unit timer(Tank t, Int time){
```

- `timed_requires` specifies the period of repeated calls to this method
- `timed_control p.m= [a,b]` specifies what the method periodically calls
  - Periodic call to `p.m` with some ECP, where `p` is a parameter and thus invariant
  - The first time after `a` time units
  - After the last call, `b` time units remain until it must be called again

Sketch Type Analysis

ECP Analysis

Three step analysis

- Run a global time analysis, derive for each statement how much time it may require to execute it (non-modular, lightweight)
- Run type system, to make sure ECP are called correctly (non-modular, lightweight)
- Generate and verify all proof obligations with ddL/KeYmaera X (modular, heavyweight)

Type system operates on the level of locations.

- A `ceid` is a pair of location and method (e.g. `p,m`). Idea:
- Keep track of all `ceid`'s and when it must be called again during type checking
- Update maximal time left for each `ceid`'s to be called
- Check that this time is always positive
- Delegation only through method calls, i.e., tree like structure

$$\Gamma_l, \Gamma_d \vdash s : \Gamma_l', \Gamma_d'$$

- Γ_l registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(fid, t_{min}, t_{max}, t)$: Future *fid*, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

Type System

$$\Gamma_I, \Gamma_d \vdash s : \Gamma'_I, \Gamma'_d$$

- Γ_I registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(fid, t_{min}, t_{max}, t)$: Future fid, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\frac{}{\Gamma_I, \left[\texttt{ceid}_i \mapsto (fid^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \texttt{s} :}$$

Type System

UNIVERSITY OF OSLO

$$\Gamma_l, \Gamma_d \vdash s : \Gamma'_l, \Gamma'_d$$

- Γ_l registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(\mathit{fid}, t_{min}, t_{max}, t)$: Future *fid*, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\frac{}{\Gamma_l, \left[\texttt{ceid}_i \mapsto (\mathit{fid}^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \texttt{s} :}$$

Type System

$$\Gamma_I, \Gamma_d \vdash s : \Gamma_I', \Gamma_d'$$

- Γ_I registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(\mathit{fid}, t_{min}, t_{max}, t)$: Future fid, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\mathtt{TA(s)} = [t^-, t^+] \qquad C = \{i \mid i \in I \wedge t^i_{min} - t^+ < 0\}$$

$$\overline{\Gamma_I, \left[\mathtt{ceid}_i \mapsto (\mathit{fid}^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \mathtt{s} :}$$

UNIVERSITY OF OSLO

$$\Gamma_I, \Gamma_d \vdash s : \Gamma'_I, \Gamma'_d$$

- Γ_I registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(\mathit{fid}, t_{min}, t_{max}, t)$: Future *fid*, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\mathtt{TA(s)} = [t^-, t^+] \qquad C = \{i \mid i \in I \land t^i_{min} - t^+ < 0\}$$

$$\Gamma'_d = \left[\mathtt{ceid}_i \mapsto (\mathit{fid}^i, t^i_{min} - t^+, t^i_{max} - t^-, t^i)\right]_{i \in I \setminus C}$$

$$\overline{\Gamma_I, \left[\mathtt{ceid}_i \mapsto (\mathit{fid}^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \mathtt{s} :}$$

$$\Gamma_l, \Gamma_d \vdash s : \Gamma'_l, \Gamma'_d$$

- Γ_l registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(fid, t_{min}, t_{max}, t)$: Future *fid*, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\mathtt{TA}(\mathtt{s}) = [t^-, t^+] \qquad C = \{i \mid i \in I \wedge t^i_{min} - t^+ < 0\}$$

$$\Gamma'_d = \left[\mathtt{ceid}_i \mapsto (fid^i, t^i_{min} - t^+, t^i_{max} - t^-, t^i)\right]_{i \in I \setminus C}$$

$$\Gamma^1_l = \left[\mathtt{ceid} \mapsto \Gamma_l(\mathtt{ceid}) - t^+\right]_{\mathtt{ceid} \in \mathbf{dom}\Gamma_l}$$

$$\frac{}{\Gamma_l, \left[\mathtt{ceid}_i \mapsto (fid^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \mathtt{s} :}$$

Type System

$$\Gamma_I, \Gamma_d \vdash s : \Gamma'_I, \Gamma'_d$$

- Γ_I registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(\mathit{fid}, t_{min}, t_{max}, t)$: Future *fid*, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\mathtt{TA(s)} = [t^-, t^+] \qquad C = \{i \mid i \in I \wedge t^i_{min} - t^+ < 0\}$$

$$\Gamma'_d = \left[\mathtt{ceid}_i \mapsto (\mathit{fid}^i, t^i_{min} - t^+, t^i_{max} - t^-, t^i)\right]_{i \in I \setminus C}$$

$$\Gamma^1_I = \left[\mathtt{ceid} \mapsto \Gamma_I(\mathtt{ceid}) - t^+\right]_{\mathtt{ceid} \in \mathbf{dom}\Gamma_I}$$

$$\Gamma^2_I = \left[\mathtt{ceid}_i \mapsto (t^i + (t^+ - t^i_{min}))\right]_{i \in C}$$

$$\overline{\Gamma_I, \left[\mathtt{ceid}_i \mapsto (\mathit{fid}^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \mathtt{s} :}$$

$$\Gamma_l, \Gamma_d \vdash s : \Gamma'_l, \Gamma'_d$$

- Γ_l registers the `ceid`'s that the method under analysis must control, maps to a number
- Γ_d registers the `ceid`'s that we delegated control to and maps them to $(fid, t_{min}, t_{max}, t)$: Future fid, how long they control t_{min}, t_{max} and when control must be called afterwards t

Rule for time advance without calls (simplified)

$$\frac{\begin{array}{c}\mathtt{TA}(\mathtt{s}) = [t^-, t^+] \qquad C = \{i \mid i \in I \wedge t^i_{min} - t^+ < 0\} \\ \Gamma'_d = \left[\mathtt{ceid}_i \mapsto (fid^i, t^i_{min} - t^+, t^i_{max} - t^-, t^i)\right]_{i \in I \setminus C} \\ \Gamma^1_l = \left[\mathtt{ceid} \mapsto \Gamma_l(\mathtt{ceid}) - t^+\right]_{\mathtt{ceid} \in \mathbf{dom}\Gamma_l} \\ \Gamma^2_l = \left[\mathtt{ceid}_i \mapsto (t^i + (t^+ - t^i_{min}))\right]_{i \in C} \\ \Gamma'_l = \Gamma^1_l \cup \Gamma^2_l \qquad \forall \mathtt{ceid} \in \Gamma'_l.\ \Gamma'_l(\mathtt{ceid}) \geq 0\end{array}}{\Gamma_l, \left[\mathtt{ceid}_i \mapsto (fid^i, t^i_{min}, t^i_{max}, t^i)\right]_{i \in I} \vdash \mathtt{s} : \Gamma'_l, \Gamma'_d}$$

Type System

- Rule for method calls is responsible for two things:
  - If one delegates control, move `ceid` from Γ_l to Γ_d
  - If the method has a ECP, update Γ_l
- When object is created, all its methods with ECP are added to Γ_l
- Full system needs context-awareness and some other rules (see paper)

Rule for calls (simplified)

UNIVERSITY OF OSLO

- Rule for method calls is responsible for two things:
  - If one delegates control, move `ceid` from Γ_l to Γ_d
  - If the method has a ECP, update Γ_l
- When object is created, all its methods with ECP are added to Γ_l
- Full system needs context-awareness and some other rules (see paper)

Rule for calls (simplified)

$$\frac{(\mathtt{e}_1, \mathtt{m}) \in \mathbf{dom}\Gamma_l \qquad \Gamma'_l = \Gamma_l\big[(\mathtt{e}_1, \mathtt{m}) \mapsto \mathit{treq}(\mathtt{T}_1.\mathtt{m})\big]}{\Gamma_l, \Gamma_d \vdash \mathtt{e}_1!\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_n)}$$

UNIVERSITY OF OSLO

- Rule for method calls is responsible for two things:
  - If one delegates control, move `ceid` from Γ_l to Γ_d
  - If the method has a ECP, update Γ_l
- When object is created, all its methods with ECP are added to Γ_l
- Full system needs context-awareness and some other rules (see paper)

Rule for calls (simplified)

$$\frac{(\mathtt{e}_1, \mathtt{m}) \in \mathbf{dom}\Gamma_l \qquad \vdash \mathtt{e}_i : \mathtt{T}_i \qquad \mathit{tctrl}(\mathtt{T}_1.\mathtt{m}) = [\mathtt{p}_i, \mathtt{m}_j \mapsto [t_j, t'_j]]_{i \in I, j \in J} \quad \Gamma'_l = \Gamma_l\big[(\mathtt{e}_1, \mathtt{m}) \mapsto \mathit{treq}(\mathtt{T}_1.\mathtt{m})\big]}{\Gamma_l, \Gamma_d \vdash \mathtt{e}_1!\mathtt{m}(\mathtt{e}_2, \dots, \mathtt{e}_n)}$$

UNIVERSITY OF OSLO

- Rule for method calls is responsible for two things:
  - If one delegates control, move `ceid` from Γ_l to Γ_d
  - If the method has a ECP, update Γ_l
- When object is created, all its methods with ECP are added to Γ_l
- Full system needs context-awareness and some other rules (see paper)

Rule for calls (simplified)

$$\frac{(\mathtt{e}_1,\mathtt{m}) \in \mathbf{dom}\Gamma_l \qquad \vdash \mathtt{e}_i : \mathtt{T}_i \qquad \mathit{tctrl}(\mathtt{T}_1.\mathtt{m}) = [\mathtt{p}_i, \mathtt{m}_j \mapsto [t_j, t_j']]_{i\in I, j\in J} \\ \Gamma_l' = \Gamma_l\big[(\mathtt{e}_1,\mathtt{m}) \mapsto \mathit{treq}(\mathtt{T}_1.\mathtt{m})\big] \qquad \Gamma_l'' = \Gamma_l' \setminus \{\mathtt{e}_i, \mathtt{m}_j \mapsto \_\}_{i\in I, j\in J}}{\Gamma_l, \Gamma_d \vdash \mathtt{e}_1!\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_n)}$$

Type System

- Rule for method calls is responsible for two things:
  - If one delegates control, move `ceid` from Γ_l to Γ_d
  - If the method has a ECP, update Γ_l
- When object is created, all its methods with ECP are added to Γ_l
- Full system needs context-awareness and some other rules (see paper)

Rule for calls (simplified)

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{m}) \in \mathbf{dom}\Gamma_l \qquad \vdash \mathtt{e}_i : \mathtt{T}_i \qquad \mathit{tctrl}(\mathtt{T}_1.\mathtt{m}) = [\mathtt{p}_i, \mathtt{m}_j \mapsto [t_j, t_j']]_{i \in I, j \in J} \\ \Gamma_l' = \Gamma_l\big[(\mathtt{e}_1,\mathtt{m}) \mapsto \mathit{treq}(\mathtt{T}_1.\mathtt{m})\big] \qquad \Gamma_l'' = \Gamma_l' \setminus \{\mathtt{e}_i, \mathtt{m}_j \mapsto \_\}_{i \in I, j \in J} \\ \mathtt{TA}(\mathtt{T}_1.\mathtt{m}) = [t^-, t^+] \qquad \Gamma_l(\mathtt{e}_i.\mathtt{m}_j) \geq t_j\end{array}}{\Gamma_l, \Gamma_d \vdash \mathtt{e}_1!\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_n)}$$

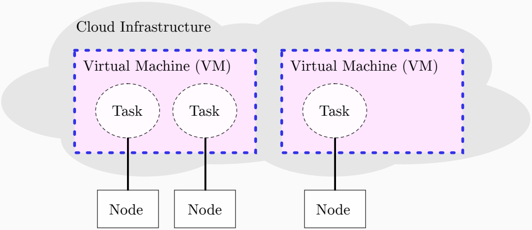
Type System

- Rule for method calls is responsible for two things:
  - If one delegates control, move `ceid` from Γ_I to Γ_d
  - If the method has a ECP, update Γ_I
- When object is created, all its methods with ECP are added to Γ_I
- Full system needs context-awareness and some other rules (see paper)

Rule for calls (simplified)

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{m}) \in \mathbf{dom}\Gamma_I \qquad \vdash \mathtt{e}_i : \mathtt{T}_i \qquad \mathit{tctrl}(\mathtt{T}_1.\mathtt{m}) = [\mathtt{p}_i, \mathtt{m}_j \mapsto [t_j, t'_j]]_{i \in I, j \in J} \\ \Gamma'_I = \Gamma_I\big[(\mathtt{e}_1,\mathtt{m}) \mapsto \mathit{treq}(\mathtt{T}_1.\mathtt{m})\big] \qquad \Gamma''_I = \Gamma'_I \setminus \{\mathtt{e}_i, \mathtt{m}_j \mapsto \_\}_{i \in I, j \in J} \\ \mathtt{TA}(\mathtt{T}_1.\mathtt{m}) = [t^-, t^+] \qquad \Gamma_I(\mathtt{e}_i.\mathtt{m}_j) \geq t_j\end{array}}{\Gamma_I, \Gamma_d \vdash \mathtt{e}_1!\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_n) : \Gamma''_I, \Gamma_d[(\mathtt{e}_i, \mathtt{m}_j) \mapsto (\mathit{fid}, t^-, t^+, t'_j)]}$$

- Type system works for Timed ABS
- If model can be separated into timed control structure and hybrid, HABS can be used
- Cyber-physical systems are only at the edge!



Conclusion

Conclusion

UNIVERSITY OF OSLO

Summary

- Post-regions for external control
- Type system ensures that control structures respect timing constraints
- Modular in time-analysis
- Modularity of deductive verification preserved

Future Work

- Implementation
- Beyond tree structured delegation
- Further post-region patterns

Conclusion

UNIVERSITY OF OSLO

Summary

- Post-regions for external control
- Type system ensures that control structures respect timing constraints
- Modular in time-analysis
- Modularity of deductive verification preserved

Future Work

- Implementation
- Beyond tree structured delegation
- Further post-region patterns

Thank you for your attention