Towards multi-refinement for the *LeanMachines* Framework

Danaël CARBONNEAU

June 3, 2025

Supervised by Frédéric Peschanski and Romain Demangeon

Introduction

The Event B method

- Formal method for state-based reactive systems

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

The Framework *LeanMachines*[Pes24]

Peschanski, Frédéric. « Stateful Functional Modeling with Refinement (a Lean4 Framework) ». Integrated Formal Methods.

- Inspired by the Event B method

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

The Framework *LeanMachines*[Pes24]

Peschanski, Frédéric. « Stateful Functional Modeling with Refinement (a Lean4 Framework) ». Integrated Formal Methods.

- Inspired by the Event B method
- Shallow embedding in *Lean*4

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

The Framework *LeanMachines*[Pes24]

Peschanski, Frédéric. « Stateful Functional Modeling with Refinement (a Lean4 Framework) ». Integrated Formal Methods.

- Inspired by the Event B method
- Shallow embedding in *Lean*4
  - Typed functional programming language,

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

The Framework *LeanMachines*[Pes24]

Peschanski, Frédéric. « Stateful Functional Modeling with Refinement (a Lean4 Framework) ». Integrated Formal Methods.

- Inspired by the Event B method
- Shallow embedding in *Lean*4
  - Typed functional programming language,
  - With dependent types

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

The Framework *LeanMachines*[Pes24]

Peschanski, Frédéric. « Stateful Functional Modeling with Refinement (a Lean4 Framework) ». Integrated Formal Methods.

- Inspired by the Event B method
- Shallow embedding in *Lean*4
  - Typed functional programming language,
  - With dependent types
  - A proof assistant

Introduction

The Event B method

- Formal method for state-based reactive systems
- Conceptualized by *Jean-Raymond Abrial*[Abr10]
- Correction by construction : proof obligations need to be discharged as the developpement goes

The Framework *LeanMachines*[Pes24]

Peschanski, Frédéric. « Stateful Functional Modeling with Refinement (a Lean4 Framework) ». Integrated Formal Methods.

- Inspired by the Event B method
- Shallow embedding in *Lean*4
  - Typed functional programming language,
  - With dependent types
  - A proof assistant
- Specfication, implementation, proofs of corrections, all in one go !

LeanMachines

Machines

- Type representing the internal state

LeanMachines

Machines

- Type representing the internal state
- Machine defined over this type by the fact that the internal state **must** respect an invariant (= **specification**)

LeanMachines

Machines

- Type representing the internal state
- Machine defined over this type by the fact that the internal state **must** respect an invariant (= **specification**)
- Top down approach : we specify several machines for the same system : starting with an abstract machine, we then concretize its behaviour through refinement

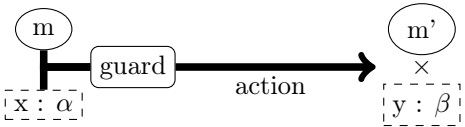
LeanMachines

Machines

- Type representing the internal state
- Machine defined over this type by the fact that the internal state **must** respect an invariant (= **specification**)
- Top down approach : we specify several machines for the same system : starting with an abstract machine, we then concretize its behaviour through refinement

Events

- Guarded transitions



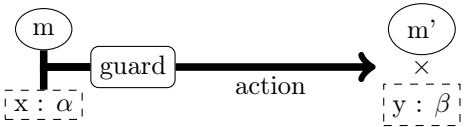
LeanMachines

Machines

- Type representing the internal state
- Machine defined over this type by the fact that the internal state **must** respect an invariant (= **specification**)
- Top down approach : we specify several machines for the same system : starting with an abstract machine, we then concretize its behaviour through refinement

Events

- Guarded transitions
  - guard (m : M) (x : α) : Prop
  - action (m : M) (x : α) (grd : guard m x) : $M \times \beta$



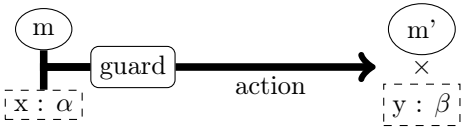
LeanMachines

Machines

- Type representing the internal state
- Machine defined over this type by the fact that the internal state **must** respect an invariant (= **specification**)
- Top down approach : we specify several machines for the same system : starting with an abstract machine, we then concretize its behaviour through refinement

Events

- Guarded transitions
  - guard (m : M) (x : α) : Prop
  - action (m : M) (x : α) (grd : guard m x) : $M \times \beta$
- Events **must** respect certain properties (**proof obligations** for the user)



Definition of a machine(formalisation)[1]

```
class Machine (CTX : Type u) (M) where
  context : CTX
  invariant : M → Prop
```

The Machine typeclass

[1]The code presented in these slides is a simplified version of the code of *LeanMachines*, complete version can be found here : https://github.com/lean-machines-central/lean-machines

Definition of a machine(formalisation)[1]

```
class Machine (CTX : Type u) (M) where
  context : CTX
  invariant : M → Prop
```

The Machine typeclass

context : static part of the system (constants, properties about the constants)

[1]The code presented in these slides is a simplified version of the code of *LeanMachines*, complete version can be found here : https://github.com/lean-machines-central/lean-machines

Definition of a machine(formalisation)[1]

```
class Machine (CTX : Type u) (M) where
  context : CTX
  invariant : M → Prop
```

The Machine typeclass

context : static part of the system (constants, properties about the constants)

invariant : properties the user wants to specify about the machine

[1]The code presented in these slides is a simplified version of the code of *LeanMachines*, complete version can be found here : https://github.com/lean-machines-central/lean-machines

Definition of a machine (example)

Model of a bounded buffer

- The *size* of the buffer must be bounded by *maxSize*

```
structure BufContext where
  maxSize : Nat
  maxSizeProp : maxSize > 0

structure B0 (ctx : BufContext) where
  size : Nat
```

Definition of a machine (example)

Model of a bounded buffer

- The *size* of the buffer must be bounded by *maxSize*
- We **specify** this by instanciating the typeclass Machine

```
structure BufContext where
  maxSize : Nat
  maxSizeProp : maxSize > 0

structure B0 (ctx : BufContext) where
  size : Nat

instance: Machine BufContext (B0 ctx)
    where
  context := ctx
  invariant b0 := b0.size ≤ ctx.maxSize
```

Definition of an event (example)

Adding an element to the buffer

Definition of an event (example)

Adding an element to the buffer

- size := size + 1

Definition of an event (example)

Adding an element to the buffer

- size := size + 1
- The guard needs to check that this operation will not exceed the bound !

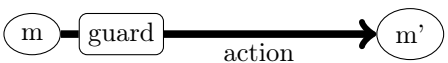
Definition of an event (example)

Adding an element to the buffer

- size := size + 1
- The guard needs to check that this operation will not exceed the bound !

```
def B0.Put : Event (B0 ctx) Unit Unit :=
{
    guard := fun b0 _ =>
         b0.size < ctx.maxSize
    action := fun b0 _ _ =>
         ((),{ size := b0.size + 1 })
}
```

Proof Obligation : *safety*

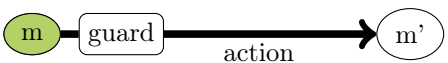


What is a Proof Obligation ?

- **Users** need to prove them when they specify the system
- The *safety* PO, for example, guarantees that the invariant won't be broken by the event's action.

safety (m : M) (x : α): Machine.invariant m $\rightarrow$ (grd : ev.guard m x) $\rightarrow$ Machine.invariant (ev.action m x grd).snd

Proof Obligation : *safety*

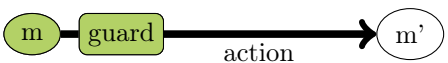


What is a Proof Obligation ?

- **Users** need to prove them when they specify the system
- The *safety* PO, for example, guarantees that the invariant won't be broken by the event's action.

safety (m : M) (x : α): Machine.invariant m $\rightarrow$ (grd : ev.guard m x) $\rightarrow$ Machine.invariant (ev.action m x grd).snd

Proof Obligation : *safety*

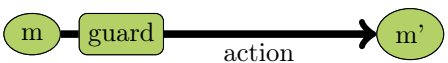


What is a Proof Obligation ?

- **Users** need to prove them when they specify the system
- The *safety* PO, for example, guarantees that the invariant won't be broken by the event's action.

safety (m : M) (x : α): Machine.invariant m $\rightarrow$ (grd : ev.guard m x) $\rightarrow$ Machine.invariant (ev.action m x grd).snd

Proof Obligation : *safety*



What is a Proof Obligation ?

- **Users** need to prove them when they specify the system
- The *safety* PO, for example, guarantees that the invariant won't be broken by the event's action.

safety (m : M) (x : α): Machine.invariant m $\rightarrow$ (grd : ev.guard m x) $\rightarrow$ Machine.invariant (ev.action m x grd).snd

Proof of the *safety* PO for Put

The steps of the proof

1. **Goal**:
   b0.size ≤ ctx.maxSize → (grd : b0.size < ctx.maxSize)
   → (((), size := b0.size + 1).snd).size ≤ ctx.maxSize

```
safety := fun b0 x =>
by
```

Proof of the *safety* PO for Put

The steps of the proof

1. **Goal**:
   $b0.size \leq ctx.maxSize \rightarrow (grd : b0.size < ctx.maxSize) \rightarrow (((), size := b0.size + 1).snd).size \leq ctx.maxSize$
2. **Goal** :
   $b0.size \leq ctx.maxSize \rightarrow (grd : b0.size < ctx.maxSize) \rightarrow b0.size + 1 \leq ctx.maxSize$

```
safety := fun b0 x =>
by

   simp
```

Proof of the *safety* PO for Put

The steps of the proof

1. **Goal**:
   $b0.size \leq ctx.maxSize \rightarrow (grd : b0.size < ctx.maxSize) \rightarrow (((), size := b0.size + 1).snd).size \leq ctx.maxSize$
2. **Goal** :
   $b0.size \leq ctx.maxSize \rightarrow (grd : b0.size < ctx.maxSize) \rightarrow b0.size + 1 \leq ctx.maxSize$
3. **Goal** : $b0.size + 1 \leq ctx.maxSize$
   - $hinv : b0.size \leq ctx.maxSize$
   - $hgrd : b0.size < ctx.maxSize$

```
safety := fun b0 x =>
by

    simp

    intros hinv hgrd
```

Proof of the *safety* PO for Put

The steps of the proof

1. **Goal**:
   b0.size $\leq$ ctx.maxSize $\rightarrow$ (grd : b0.size $<$ ctx.maxSize)
   $\rightarrow$ (((), size := b0.size + 1).snd).size $\leq$ ctx.maxSize
2. **Goal** :
   b0.size $\leq$ ctx.maxSize $\rightarrow$ (grd : b0.size $<$ ctx.maxSize)
   $\rightarrow$ b0.size + 1 $\leq$ ctx.maxSize
3. **Goal** : b0.size + 1 $\leq$ ctx.maxSize
   - hinv : b0.size $\leq$ ctx.maxSize
   - hgrd : b0.size $<$ ctx.maxSize

```
safety := fun b0 x =>
by

    simp

    intros hinv hgrd

    exact hgrd
```

Refinement between machines

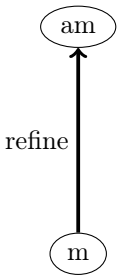
am

m

What is Machine Refinement ?

- Relation between specifications (one is the abstraction of the other)

Refinement between machines



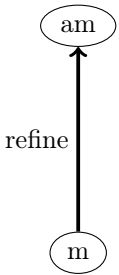
What is Machine Refinement ?

- Relation between specifications (one is the abstraction of the other)

  refine Property defining how the abstract machine is an abstraction of the concrete one

- refine : AM → M → Prop

Refinement between machines



What is Machine Refinement ?

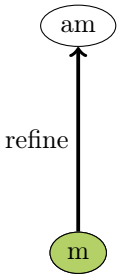
- Relation between specifications (one is the abstraction of the other)

  refine : Property defining how the abstract machine is an abstraction of the concrete one

  refine safe : Proof Obligation linking the abstract and the concrete invariant (the concrete invariant must imply the abstract one)

- refine : AM → M → Prop
- refine_safe (am : AM) (m : M) : Machine.invariant m
  → refine am m
  → Machine.invariant am

Refinement between machines



What is Machine Refinement ?

- Relation between specifications (one is the abstraction of the other)

  refine Property defining how the abstract machine is an abstraction of the concrete one

  refine safe Proof Obligation linking the abstract and the concrete invariant (the concrete invariant must imply the abstract one)

- refine : AM → M → Prop
- refine_safe (am : AM) (m : M) : Machine.invariant m
  → refine am m
  → Machine.invariant am

Refinement between machines

What is Machine Refinement ?

- Relation between specifications (one is the abstraction of the other)

  refine — Property defining how the abstract machine is an abstraction of the concrete one

  refine safe — Proof Obligation linking the abstract and the concrete invariant (the concrete invariant must imply the abstract one)

- refine : AM → M → Prop
- refine_safe (am : AM) (m : M) : Machine.invariant m
  → refine am m
  → Machine.invariant am

Refinement between machines

What is Machine Refinement ?

- Relation between specifications (one is the abstraction of the other)

  refine : Property defining how the abstract machine is an abstraction of the concrete one

  refine safe : Proof Obligation linking the abstract and the concrete invariant (the concrete invariant must imply the abstract one)

- refine : AM → M → Prop
- refine_safe (am : AM) (m : M) : Machine.invariant m
  → refine am m
  → Machine.invariant am

Example of refinement between machines

Abstract Buffer (B0) : counter

- The context consists of the bound ($ctx.maxSize$) and the property that it is non null ($ctx.maxSizeProp$)
- $size : \mathbb{N}$
- Invariant : $b0.size \leq ctx.maxSize$

Example of refinement between machines

Abstract Buffer (B0) : counter

- The context consists of the bound ($ctx.maxSize$) and the property that it is non null ($ctx.maxSizeProp$)
- $size : \mathbb{N}$
- Invariant : $b0.size \leq ctx.maxSize$

Concrete Buffer (B1) : stack of values

- same context as the abstract one
- $stack : List\ \alpha$
- Invariant : $b1.stack.length \leq ctx.maxSize$

Example of refinement between machines

Abstract Buffer (B0) : counter

- The context consists of the bound ($ctx.maxSize$) and the property that it is non null ($ctx.maxSizeProp$)
- $size : \mathbb{N}$
- Invariant : $b0.size \leq ctx.maxSize$

Concrete Buffer (B1) : stack of values

- same context as the abstract one
- $stack : List\ \alpha$
- Invariant : $b1.stack.length \leq ctx.maxSize$

The refinement relation

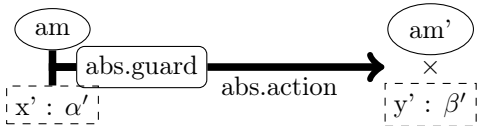
- Correspondance between the two machines : *size* corresponds to the length of *stack*.

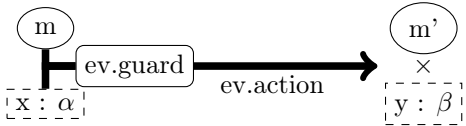
```
refine am m    := am.size = m.stack.length

refine_safe am m  :=
    by
      simp[Machine.invariant]
      intros hinv_ccr href
      rw[href]
      exact hinv_ccr
```

Refinement between events

What is Event Refinement ?

- Each event defined on the concrete machine must refine one of the abstract ones
- To establish such refinement we need :

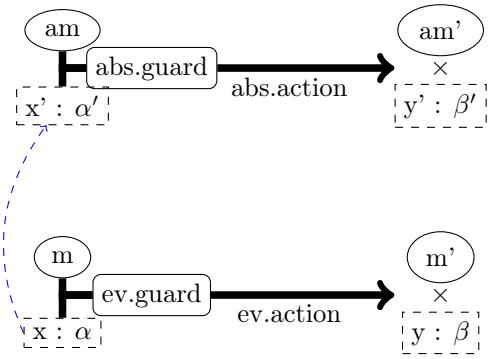




Refinement between events

What is Event Refinement ?

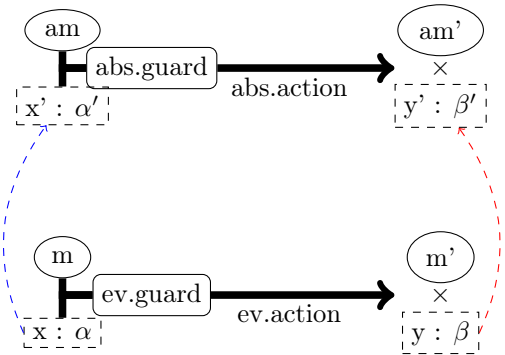
- Each event defined on the concrete machine must refine one of the abstract ones
- To establish such refinement we need :
  - A way to lift the input to the abstract type (*lift_in*)



Refinement between events

What is Event Refinement ?

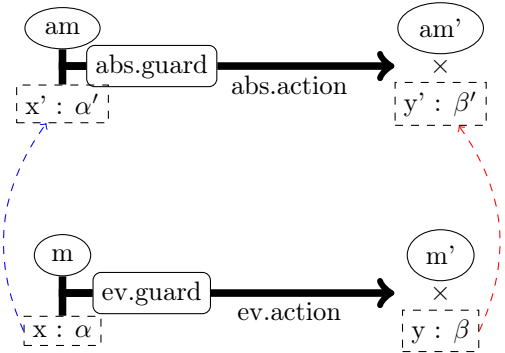
- Each event defined on the concrete machine must refine one of the abstract ones
- To establish such refinement we need :
  - A way to lift the input to the abstract type (*lift_in*)
  - A way to lift the output to the abstract type (*lift_out*)



Refinement between events

What is Event Refinement ?

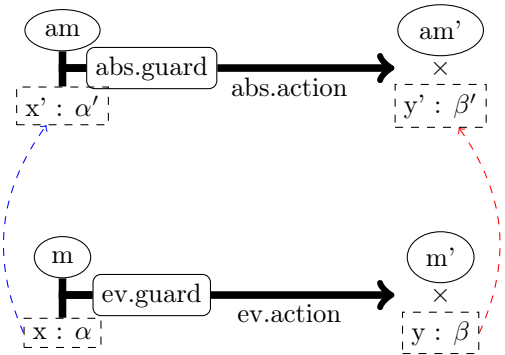
- Each event defined on the concrete machine must refine one of the abstract ones
- To establish such refinement we need :
  - A way to lift the input to the abstract type (*lift_in*)
  - A way to lift the output to the abstract type (*lift_out*)
  - A Proof Obligation about the guards (*strenghtening*)



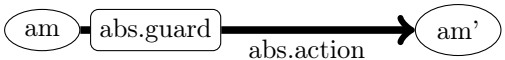
Refinement between events

What is Event Refinement ?

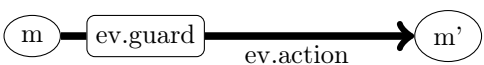
- Each event defined on the concrete machine must refine one of the abstract ones
- To establish such refinement we need :
  - A way to lift the input to the abstract type (*lift_in*)
  - A way to lift the output to the abstract type (*lift_out*)
  - A Proof Obligation about the guards (*strenghtening*)
  - A Proof Obligation about the action (*simulation*)



Proof Obligation : *strengthening*

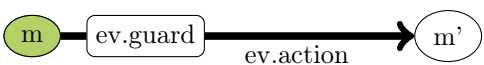


strengthening (m : M) (x : α):
Machine.invariant m $\rightarrow$ ev.guard m x $\rightarrow$ $\forall$
am, refine am m $\rightarrow$ abs.guard am (lift_in x)



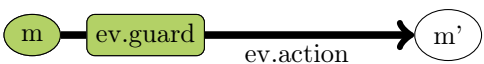
Proof Obligation : *strengthening*

strengthening (m : M) (x : α):
Machine.invariant m $\rightarrow$ ev.guard m x $\rightarrow$ $\forall$
am, refine am m $\rightarrow$ abs.guard am (lift_in x)

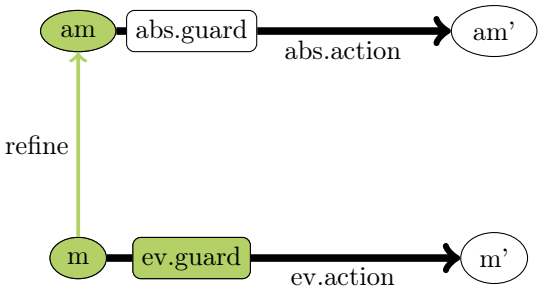


Proof Obligation : *strengthening*

strengthening (m : M) (x : α):
Machine.invariant m $\rightarrow$ ev.guard m x $\rightarrow$ $\forall$
am, refine am m $\rightarrow$ abs.guard am (lift_in x)

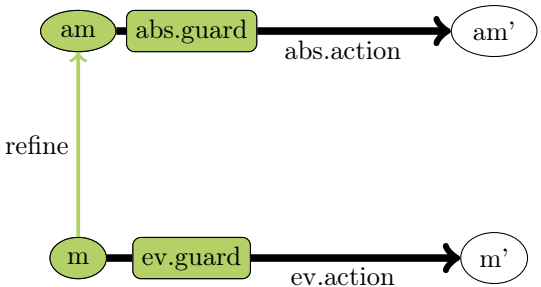


Proof Obligation : *strengthening*



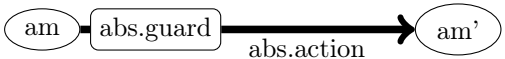
strengthening (m : M) (x : α):
Machine.invariant m $\rightarrow$ ev.guard m x $\rightarrow$ $\forall$
am, refine am m $\rightarrow$ abs.guard am (lift_in x)

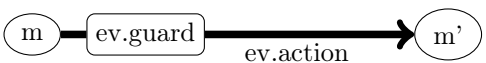
Proof Obligation : *strengthening*



strengthening (m : M) (x : α):
Machine.invariant m $\rightarrow$ ev.guard m x $\rightarrow$ $\forall$
am, refine am m $\rightarrow$ abs.guard am (lift_in x)

Proof Obligation: *simulation*

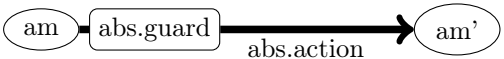


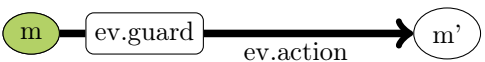


simulation (m : M) (x : α):
(Hinv : Machine.invariant m)
$\rightarrow$ (Hgrd : ev.guard m x)
$\rightarrow \forall$ am, (Href: refine am m)
$\rightarrow$ let (y, m') := ev.action m x Hgrd
let (z, am') := abs.action am (lift_in x)
(strengthening m x Hinv Hgrd am Href)

lift_out y = z $\wedge$ refine am' m'

Proof Obligation: *simulation*

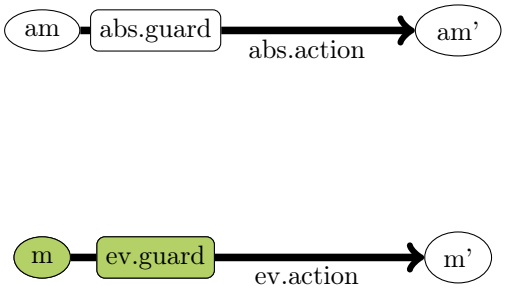




simulation (m : M) (x : α):
(Hinv : Machine.invariant m)
$\rightarrow$ (Hgrd : ev.guard m x)
$\rightarrow \forall$ am, (Href: refine am m)
$\rightarrow$ let (y, m') := ev.action m x Hgrd
let (z, am') := abs.action am (lift_in x)
(strengthening m x Hinv Hgrd am Href)

lift_out y = z $\wedge$ refine am' m'

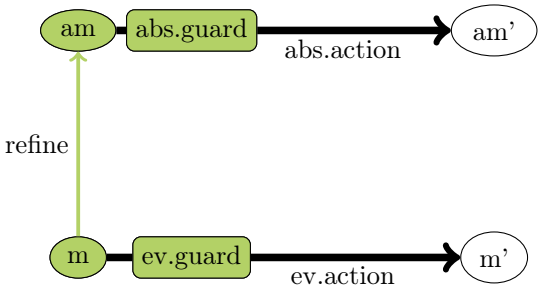
Proof Obligation: *simulation*



simulation (m : M) (x : α):
(Hinv : Machine.invariant m)
$\rightarrow$ (Hgrd : ev.guard m x)
$\rightarrow \forall$ am, (Href: refine am m)
$\rightarrow$ let (y, m') := ev.action m x Hgrd
let (z, am') := abs.action am (lift_in x)
(strengthening m x Hinv Hgrd am Href)

lift_out y = z $\wedge$ refine am' m'

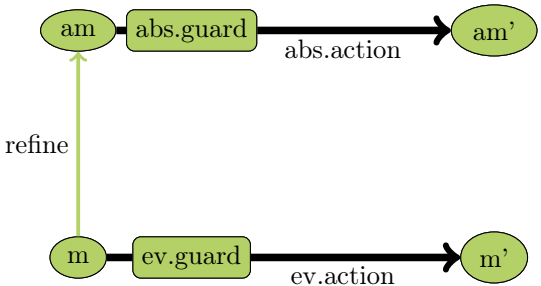
Proof Obligation: *simulation*



simulation (m : M) (x : α):
(Hinv : Machine.invariant m)
$\rightarrow$ (Hgrd : ev.guard m x)
$\rightarrow \forall$ am, (Href: refine am m)
$\rightarrow$ let (y, m') := ev.action m x Hgrd
let (z, am') := abs.action am (lift_in x)
(strengthening m x Hinv Hgrd am Href)

lift_out y = z $\wedge$ refine am' m'

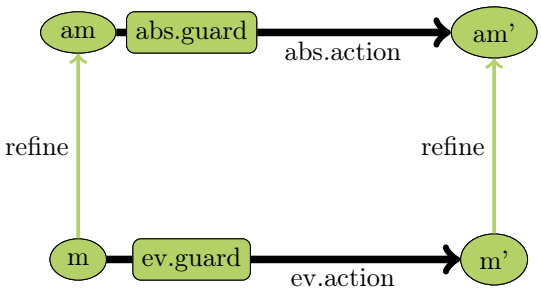
Proof Obligation: *simulation*



simulation (m : M) (x : α):
(Hinv : Machine.invariant m)
$\rightarrow$ (Hgrd : ev.guard m x)
$\rightarrow \forall$ am, (Href: refine am m)
$\rightarrow$ let (y, m') := ev.action m x Hgrd
let (z, am') := abs.action am (lift_in x)
(strengthening m x Hinv Hgrd am Href)

lift_out y = z $\wedge$ refine am' m'

Proof Obligation: *simulation*



simulation (m : M) (x : α):
(Hinv : Machine.invariant m)
$\rightarrow$ (Hgrd : ev.guard m x)
$\rightarrow \forall$ am, (Href: refine am m)
$\rightarrow$ let (y, m') := ev.action m x Hgrd
let (z, am') := abs.action am (lift_in x)
(strengthening m x Hinv Hgrd am Href)

lift_out y = z $\wedge$ refine am' m'

Example of Event refinement : Put

Add an element to the buffer (B0)

- size := size + 1
- *size* must be stricly below the bound

```
def B0.Put : Event (B0 ctx) Unit Unit :=
{
    guard := fun b0 _ =>
         b0.size < ctx.maxSize
    action := fun b0 _ _ =>
         ((),{ size := b0.size + 1 })
}
```

Add an element to the buffer (B1)

- Insertion at the head of the list
- The length of the list must be strictly below the bound

```
def B1.Put : Event (B0 ctx) α Unit :=
{
    guard := fun b1 _ =>
         b1.length < ctx.maxSize
    action := fun b1 x _ =>
         ((),{ stack := x :: b1.stack })
}
```

Example of Event refinement : Put

Lift of the Inputs/Outputs

- $lift\_in : \alpha \to Unit := \lambda\_.()$
- $lift\_out : Unit \to Unit := \lambda x.x$

Example of Event refinement : Put

Lift of the Inputs/Outputs

- $lift\_in : \alpha \to Unit := \lambda\_.()$
- $lift\_out : Unit \to Unit := \lambda x.x$

Strengthening

strengthening (m : B1) (x : α):
m.stack.length $\leq$ maxSize
$\to$ m.stack.length $<$ maxSize
$\to \forall$ am : B0, am.size = m.stack.length
$\to$ am.size $<$ maxSize

Example of Event refinement : Put

Lift of the Inputs/Outputs

- $lift\_in : \alpha \to Unit := \lambda\_.()$
- $lift\_out : Unit \to Unit := \lambda x.x$

Strengthening

strengthening (m : B1) (x : α):
m.stack.length $\leq$ maxSize
$\to$ m.stack.length $<$ maxSize
$\to \forall$ am : B0, am.size = m.stack.length
$\to$ am.size $<$ maxSize

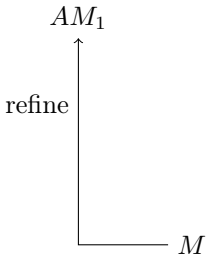
Simulation

simulation (m : B1) (x : α):
(Hinv : m.stack.length $\leq$ maxSize)
$\to$ (Hgrd : m.stack.length $<$ maxSize)
$\to \forall$ am : B0, (Href: am.size = m.stack.length)
$\to$ let (y, m') := ((), stack := x::m.stack)
let (z, am') := ((),size := size +1)

y = z $\wedge$ am'.size = m'.stack.length

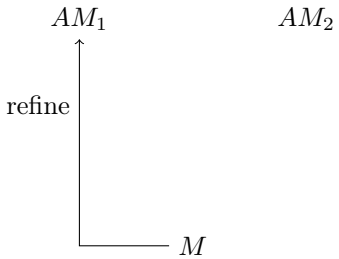
What is Multi-refinement

- With *LeanMachines*, we can specify systems through simple refinement (one abstract machine per concrete machine)



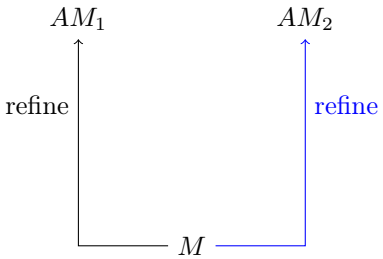
What is Multi-refinement

- With *LeanMachines*, we can specify systems through simple refinement (one abstract machine per concrete machine)
- Sometimes, multiple machines are an abstraction of the same concrete one



What is Multi-refinement

- With *LeanMachines*, we can specify systems through simple refinement (one abstract machine per concrete machine)
- Sometimes, multiple machines are an abstraction of the same concrete one
- We would like to allow multiple refinement in the framework in order to combine and **reuse** specifications



Multi-refinement : the pushBuffer

[Pes24]

Bounded buffer (B0)

State $size : \mathbb{N}$
Invariant : size $\leq$ SizeMax
Events : Push an element in the buffer

Button

State $pushed : \mathbb{B}$
Invariant : $True$
Events : push / release the button

Multi-refinement : the pushBuffer

[Pes24]

Bounded buffer (B0)

State $size : \mathbb{N}$
Invariant : size $\leq$ SizeMax
Events : Push an element in the buffer

Button

State $pushed : \mathbb{B}$
Invariant : $True$
Events : push / release the button

The pushBuffer

State $stack$: $List\ \alpha$, $pushed : \mathbb{B}$
Invariant : $stack.length \leq max$
Events : push the button (which pushes an arbitrary element of type α to in buffer)

Multiple refinement of Events (example)

Bounded buffer (B0) : Put

guard : $size + 1 \leq SizeMax$

action : $size := size + 1$

Button : Push

guard : $\neg pressed$

action : $pressed := true$

Multiple refinement of Events (example)

Bounded buffer (B0) : Put

guard : $size + 1 \leq SizeMax$
action : $size := size + 1$

Button : Push

guard : $\neg pressed$
action : $pressed := true$

The pushBuffer : PushAdd

guard : $stack.length + 1 \leq SizeMax \wedge \neg pressed$
action : $stack := x :: stack, pushed := true$

Previous version of the API (refinement)

```
structure _REventPO  [Machine ACTX AM] [Machine CTX M] [instR: Refinement AM M]
   (ev : _Event M α β)  (α' β')
   extends _EventPO ev where
  abstract : _Event AM α' β'
  lift_in : α → α'
  lift_out : β → β'
  strengthening (m : M) (x : α):
  /- [...] -/
  simulation (m : M) (x : α):
  /- [...] -/

structure OrdinaryREvent (AM) [Machine ACTX AM] (M) [Machine CTX M]
[instR: Refinement AM M]
  (α β) (α':=α) (β':=β) extends _Event M α β where
  po : _REventPO (instR:=instR) to_Event
  α' β'
```

The issue with this version

```
def PushBuffer.PushAdd : OrdinaryREvent
    (B0 ctx) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := B0.Put
         safety    := /- [...] -/
      }
}
```

```
def PushBuffer.PushAdd : OrdinaryREvent
    (Button) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := Button.Push
         safety    := /- [...] -/
      }
}
```

Limits of the use of `structure`

The issue with this version

```
def PushBuffer.PushAdd : OrdinaryREvent
    (B0 ctx) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := B0.Put
         safety    := /- [...] -/
      }
}
```

```
def PushBuffer.PushAdd : OrdinaryREvent
    (Button) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := Button.Push
         safety    := /- [...] -/
      }
}
```

Limits of the use of `structure`

- We need the define **everything** about the event (action, guard, safety, refinement PO) at once

The issue with this version

```
def PushBuffer.PushAdd : OrdinaryREvent
    (B0 ctx) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := B0.Put
         safety    := /- [...] -/
      }
}
```

```
def PushBuffer.PushAdd : OrdinaryREvent
    (Button) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := Button.Push
         safety    := /- [...] -/
      }
}
```

Limits of the use of `structure`

- We need the define **everything** about the event (action, guard, safety, refinement PO) at once
- It's not possible to **combine** structure definitions

The issue with this version

```
def PushBuffer.PushAdd : OrdinaryREvent
    (B0 ctx) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := B0.Put
         safety    := /- [...] -/
      }
}
```

```
def PushBuffer.PushAdd : OrdinaryREvent
    (Button) (PushBuffer ctx α) α Unit :=
{  guard m _     := /- [...]-/
   action m x _     :=/- [...]-/
   po := {   abstract := Button.Push
         safety    := /- [...] -/
      }
}
```

Limits of the use of `structure`

- We need the define **everything** about the event (action, guard, safety, refinement PO) at once
- It's not possible to **combine** structure definitions

⇒ Does not allow *multi-refinement* !

The use of *typeclass* (safe events)

```
class SafeEventPO [Machine CTX M] {α β} (ev : Event M α β)  where
  safety (m : M) (x : α):
    Machine.invariant m
    → (grd : ev.guard m x)
    → Machine.invariant (ev.action m x grd).snd
```

The use of *typeclass* (safe events)

```
class SafeEventPO [Machine CTX M] {α β} (ev : Event M α β)  where
  safety (m : M) (x : α):
    Machine.invariant m
    → (grd : ev.guard m x)
    → Machine.invariant (ev.action m x grd).snd
```

- Proof obligations are thus summed up in a *typeclass* specifying the properties (safety, refinement of events, etc.)

The use of *typeclass* (safe events)

```
class SafeEventPO [Machine CTX M] {α β} (ev : Event M α β)  where
  safety (m : M) (x : α):
    Machine.invariant m
    → (grd : ev.guard m x)
    → Machine.invariant (ev.action m x grd).snd
```

- Proof obligations are thus summed up in a *typeclass* specifying the properties (safety, refinement of events, etc.)
- We add steps to the definition of events :

The use of *typeclass* (safe events)

```
class SafeEventPO [Machine CTX M] {α β} (ev : Event M α β)  where
  safety (m : M) (x : α):
    Machine.invariant m
    → (grd : ev.guard m x)
    → Machine.invariant (ev.action m x grd).snd
```

- Proof obligations are thus summed up in a *typeclass* specifying the properties (safety, refinement of events, etc.)
- We add steps to the definition of events :
  1. Define its action and guard

The use of *typeclass* (safe events)

```
class SafeEventPO [Machine CTX M] {α β} (ev : Event M α β)  where
  safety (m : M) (x : α):
    Machine.invariant m
    → (grd : ev.guard m x)
    → Machine.invariant (ev.action m x grd).snd
```

- Proof obligations are thus summed up in a *typeclass* specifying the properties (safety, refinement of events, etc.)
- We add steps to the definition of events :
  1. Define its action and guard
  2. Instanciate the *typeclass* that correspond to what the user wants to prove about the event

The use of *typeclass* (refinement)

```
class SafeREventPO {α β α' β'}
   [Machine ACTX AM] [Machine CTX M]
   [instR: Refinement AM M]
   (ev : Event M α β)
   (abs : Event AM α' β')
   [instSafeAbs : SafeEventPO abs]
   [instSafeEv : SafeEventPO ev]
where
  lift_in : α → α'
  lift_out : β → β'
  strengthening (m : M) (x : α): /- [...]
    -/
  simulation (m : M) (x : α):    /- [...]
    -/
```

The use of *typeclass* (refinement)

```
class SafeREventPO {α β α' β'}
   [Machine ACTX AM] [Machine CTX M]
   [instR: Refinement AM M]
   (ev : Event M α β)
   (abs : Event AM α' β')
   [instSafeAbs : SafeEventPO abs]
   [instSafeEv : SafeEventPO ev]
where
  lift_in : α → α'
  lift_out : β → β'
  strengthening (m : M) (x : α): /- [...]
    -/
  simulation (m : M) (x : α):    /- [...]
    -/
```

- We require that *safety* was discharged through *typeclass* constraints (*instSafeAbs* and *instSafeEv*)

The use of *typeclass* (refinement)

```
class SafeREventPO {α β α' β'}
   [Machine ACTX AM] [Machine CTX M]
   [instR: Refinement AM M]
   (ev : Event M α β)
   (abs : Event AM α' β')
   [instSafeAbs : SafeEventPO abs]
   [instSafeEv : SafeEventPO ev]
where
  lift_in : α → α'
  lift_out : β → β'
  strengthening (m : M) (x : α): /- [...]
    -/
  simulation (m : M) (x : α):    /- [...]
    -/
```

- We require that *safety* was discharged through *typeclass* constraints (*instSafeAbs* and *instSafeEv*)
- There can be multiple instanciation of this *typeclass* for the same concrete event !

Coherence with previous API (Events)

```
structure OrdinaryEvent (M)
[Machine CTX M] (α β : Type) where
  guard (m : M) (x : α) : Prop
  action (m : M) (x : α)
   (grd : guard m x) : β × M
  safety (m : M) (x : α) :
    Machine.invariant m
    → (grd : guard m x)
    → Machine.invariant
      (action m x grd).2

instance instSafeEventPO_OrdinaryEvent
    [Machine CTX M]
  (ev : OrdinaryEvent M α β):
    SafeEventPO ev.toEvent
  where
  safety := ev.safety
```

Coherence with previous API (Events)

```
structure OrdinaryEvent (M)
[Machine CTX M] (α β : Type) where
  guard (m : M) (x : α) : Prop
  action (m : M) (x : α)
   (grd : guard m x) : β × M
  safety (m : M) (x : α) :
    Machine.invariant m
    → (grd : guard m x)
    → Machine.invariant
      (action m x grd).2

instance instSafeEventPO_OrdinaryEvent
    [Machine CTX M]
  (ev : OrdinaryEvent M α β):
    SafeEventPO ev.toEvent
  where
  safety := ev.safety
```

- We want to keep an API based on structures

Coherence with previous API (Events)

```
structure OrdinaryEvent (M)
[Machine CTX M] (α β : Type) where
  guard (m : M) (x : α) : Prop
  action (m : M) (x : α)
   (grd : guard m x) : β × M
  safety (m : M) (x : α) :
    Machine.invariant m
    → (grd : guard m x)
    → Machine.invariant
      (action m x grd).2

instance instSafeEventPO_OrdinaryEvent
    [Machine CTX M]
  (ev : OrdinaryEvent M α β):
    SafeEventPO ev.toEvent
  where
  safety := ev.safety
```

- We want to keep an API based on structures
- We are able to switch to the use of *typeclass* with *instSafeEventPO_ OrdinaryEvent*

Multiple refinement of Specifications (example)

Bounded buffer (B0)

State $size : \mathbb{N}$
Invariant : size $\leq$ SizeMax
Events : Push an element in the buffer

Button

State $pushed : \mathbb{B}$
Invariant : $True$
Events : push / release the button

Multiple refinement of Specifications (example)

Bounded buffer (B0)

State $size : \mathbb{N}$
Invariant : size $\leq$ SizeMax
Events : Push an element in the buffer

Button

State $pushed : \mathbb{B}$
Invariant : $True$
Events : push / release the button

The pushBuffer

State $stack$: $List\ \alpha$, $pushed : \mathbb{B}$
Invariant : $stack.length \leq max$
Events : push the button (which pushes an arbitrary element of type α to in buffer)

Multiple refinement of Events (example)

Bounded buffer (B0) : Put

guard : $size + 1 \leq SizeMax$
action : $size := size + 1$

Button : Push

guard : $\neg pressed$
action : $pressed := true$

Multiple refinement of Events (example)

Bounded buffer (B0) : Put

guard : $size + 1 \leq SizeMax$
action : $size := size + 1$

Button : Push

guard : $\neg pressed$
action : $pressed := true$

The pushBuffer

guard : $stack.length + 1 \leq SizeMax \wedge \neg pressed$
action : $stack := x :: stack, pushed := true$

Multiple refinement of Events (example)

```
def PushBuffer.PushAdd : OrdinaryEvent (PushBuffer ctx α) α Unit :=
  {
    guard m _    := ¬ m.pushed ∧ m.stack.length + 1 ≤ ctx.maxSize
    action m x _  := ((),{ stack := x::m.stack, pushed := true})
    safety := by simp[Machine.invariant]
  }
```

Method

Multiple refinement of Events (example)

```
def PushBuffer.PushAdd : OrdinaryEvent (PushBuffer ctx α) α Unit :=
  {
    guard m _    := ¬ m.pushed ∧ m.stack.length + 1 ≤ ctx.maxSize
    action m x _  := ((),{ stack := x::m.stack, pushed := true})
    safety := by simp[Machine.invariant]
  }
```

Method

1. We define the event PushAdd as an OrdinaryEvent (we use the structure API)

Multiple refinement of Events (example)

```
def PushBuffer.PushAdd : OrdinaryEvent (PushBuffer ctx α) α Unit :=
  {
    guard m _    := ¬ m.pushed ∧ m.stack.length + 1 ≤ ctx.maxSize
    action m x _  := ((),{ stack := x::m.stack, pushed := true})
    safety := by simp[Machine.invariant]
  }
```

Method

1. We define the event PushAdd as an OrdinaryEvent (we use the structure API)
2. Then we can instanciate the *typeclass SafeREventPO* twice (next slide !)

Multiple refinement of Events (example)

```
instance : SafeREventPO
  PushBuffer.PushAdd.toEvent
  (AM := Button)
  Button.Push.toEvent
  (instSafeEv :=
    instSafeEventPO_OrdinaryEvent
    PushBuffer.PushAdd)
  (instSafeAbs :=
    instSafeEventPO_OrdinaryEvent
    Button.Push)
  where
    lift_in := λ _ => () ; lift_out := id
    simulation    := /- [...] -/
    strengthening := /- [...] -/
```

```
instance : SafeREventPO
  PushBuffer.PushAdd.toEvent
  (AM := B0 ctx)
  B0.Put.toEvent
  (instSafeEv :=
    instSafeEventPO_OrdinaryEvent
    PushBuffer.PushAdd)
  (instSafeAbs :=
    instSafeEventPO_OrdinaryEvent B0.Put)
  where
    lift_in := λ _ => () ; lift_out := id
    strengthening    :=/- [...] -/
    simulation    :=/- [...] -/
```

Multi-refinement in Event B

- Sometimes discussed, sometimes wished for, never (to our knowledge) implemented for Event B tools...

Multi-refinement in Event B

- Sometimes discussed, sometimes wished for, never (to our knowledge) implemented for Event B tools...

T. S. Hoang, D. Basin, and J.-R. Abrial, *Specifying Access Control in Event-B*[HBA09]

- Model of a Bank following Event B method
- Use of two separate abstract models (transactions and security protocol)
- One concrete model (the secured system of transactions)
- Multi-refinement would be a good way to **combine** specifications
- Not possible with the Event B ecosystem (they found a "workaround")

Multi-refinement in Event B

- Sometimes discussed, sometimes wished for, never (to our knowledge) implemented for Event B tools...

T. S. Hoang, D. Basin, and J.-R. Abrial, *Specifying Access Control in Event-B*[HBA09]

- Model of a Bank following Event B method
- Use of two separate abstract models (transactions and security protocol)
- One concrete model (the secured system of transactions)
- Multi-refinement would be a good way to **combine** specifications
- Not possible with the Event B ecosystem (they found a "workaround")

V. Filou, M. Mosbah, and M. Tounsi, *Towards Proved Distributed Algorithms through Refinement, Composition and Local Computations*,[FMT13]

- "We think that a form of multirefinement would prevent the duplication of proofs and allow better automation of the approach"

Conclusions

Future works

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]

Future works

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement

Future works

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement
  - More modularity in the API (we can easily switch between the structural and the typeclass approach)

Future works

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement
  - More modularity in the API (we can easily switch between the structural and the typeclass approach)
  - Extensive use of *typeclass* and dependently typed programming in Lean4 in order to achieve that

Future works

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement
  - More modularity in the API (we can easily switch between the structural and the typeclass approach)
  - Extensive use of *typeclass* and dependently typed programming in Lean4 in order to achieve that

Future works

- Implement a bigger example from litterature (probably *Access Control*[HBA09])

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement
  - More modularity in the API (we can easily switch between the structural and the typeclass approach)
  - Extensive use of *typeclass* and dependently typed programming in Lean4 in order to achieve that

Future works

- Implement a bigger example from litterature (probably *Access Control*[HBA09])
- Find other ways of making the framework more modular (currently studying the product of machines)

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement
  - More modularity in the API (we can easily switch between the structural and the typeclass approach)
  - Extensive use of *typeclass* and dependently typed programming in Lean4 in order to achieve that

Future works

- Implement a bigger example from litterature (probably *Access Control*[HBA09])
- Find other ways of making the framework more modular (currently studying the product of machines)
- Study of the executable semantics of reactive programs modelled in *LeanMachines* with *Monadic Stream Functions*[PBN16].

Conclusions

- Presentation of the work of Frédéric Peschanski[Pes24]
- Presentation of my refactor of the framework in order to allow multi-refinement
  - More modularity in the API (we can easily switch between the structural and the typeclass approach)
  - Extensive use of *typeclass* and dependently typed programming in Lean4 in order to achieve that

Future works

- Implement a bigger example from litterature (probably *Access Control*[HBA09])
- Find other ways of making the framework more modular (currently studying the product of machines)
- Study of the executable semantics of reactive programs modelled in *LeanMachines* with *Monadic Stream Functions*[PBN16].
- Possible extension of this topic towards probabilistic reactive programming (Doctoral Research Project with Christine Tasson, Frédéric Peschanski and Romain Demangeon)

Bibliography I


Jean-Raymond Abrial.
Modeling in Event-B: System and Software Engineering.
Cambridge University Press, Cambridge, 2010.


Vincent Filou, Mohamed Mosbah, and Mohamed Tounsi.
Towards Proved Distributed Algorithms through Refinement, Composition and Local Computations.
In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 353–358, June 2013.
ISSN: 1524-4547.

Thai S. Hoang, David Basin, and Jean-Raymond Abrial.
Specifying Access Control in Event-B.
Technical report, ETH Zurich, 2009.
Artwork Size: 17 p. Medium: application/pdf.

Bibliography II


Ivan Perez, Manuel Bärenz, and Henrik Nilsson.
Functional reactive programming, refactored.
In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 33–44, New York, NY, USA, September 2016. Association for Computing Machinery.


Frédéric Peschanski.
Stateful Functional Modeling with Refinement (a Lean4 Framework).
In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods*, pages 109–127, Cham, 2024. Springer Nature Switzerland.