Modeling and Validating Distributed Embedded Real-Time Systems with VDM++

Marcel Verhoef – CHESS & Radboud University Nijmegen (NL)
Peter Gorm Larsen – Engineering College of Aarhus (DK)
Jozef Hooman – ESI & Radboud University Nijmegen (NL)

CHESS iha.dk Embedded Systems INSTITUTE Radboud Universiteit Nijmegen

Contents of this talk

- Motivation – early life-cycle system architecting
- VDM++ for distributed embedded real-time
- Case study: In-Car Radio Navigation System
- The role of formal methods
- Conclusions and future work

Why System Architecting is hard (1)

- Early phases system life-cycle are extremely volatile
- Many unknowns (not just technical)
- Nevertheless the *key decisions* need to be made early on
- Often *out-of-phase* system development occurs
- Design is typically *mono-disciplinary* organized
- “Shooting at a moving target”

Why System Architecting is hard (2)

- Task of the System Architect is to
  - Increase confidence in the system
  - To reduce project and product risks
  - While dealing with uncertainty
  - Working under high time pressure

- The System architect needs to *bridge the gap* between the disciplines **and** *deal with the design complexity* in a very *cost-effective* way

Can Formal Methods Help?

- Yes, FM offer great opportunities to beat complexity
- No, FM typically take too much effort and resources (*If you need to mow the lawn, don't take scissors*).

Is there a middle way? We believe there is.

Its been around for a while and it is called VDM.

VDM++ for distributed embedded real-time?

- VDM++ has excellent industrial track record
  [BVW1999], [HA2000], [FLMPV2005], [KON2005]

- Industry strength tools are available
  VDMTools → **`http://www.vdmtools.jp/en`**

- UML coupling, code generation, round-trip engineering

- But: new application domain for CSK Systems

On the use of VDM++ in industry

- Felicia Networks (Sony Corporation)
- Formal specification of firmware for mobile phone IC
- 150 man year project (50 people, 3 years)
- 100.000 lines, 700 page specification written in VDM++
- Validated by 10.000.000 test cases using VDMTools
- *Measured* quality improvement due to formal modeling
- Project on-time, within budget
- Product roll-out Q4-2006: 10.000.000 ICs

[source: Overture workshop, FM'06, Shin Sahara, CSK Systems]

VDM++ useful for distributed real-time?

- Timed VDM++ was our starting point
- Evaluate language and tools in this application domain
- Changes to notation and tool support were needed
- Prototype these changes and validate
- Challenge: keep changes minimal

VDM++ in a nutshell (1)

- Class
- Values
- Types
- Instance variables
- Functions
- Operations
- Threads
- *Synchronization*

```
class Buffer

instance variables
  val : [nat] := nil

operations
  public Set: nat ==> ()
  Set (pv) == val := pv;

  public Get: () ==> nat
  Get () ==
    ( dcl res : nat := val;
      val := nil;
      return res )

sync
  per Set => val = nil;
  per Get => val <> nil;
  mutex (Get, Set)

end Buffer
```

VDM++ in a nutshell (2)

- Class
- Values
- Types
- Instance variables
- Functions
- Operations
- ***Threads***
- Synchronization

```
class Producer

instance variables
  public theBuf : Buffer;
  private theVal : nat := 0

operations
  public Producer: Buffer ==> Producer
  Producer (pBuf) == theBuf := pBuf

thread
  while true do
    ( theBuf.Set(theVal);
      theVal := theVal + 1 )

end Producer
```

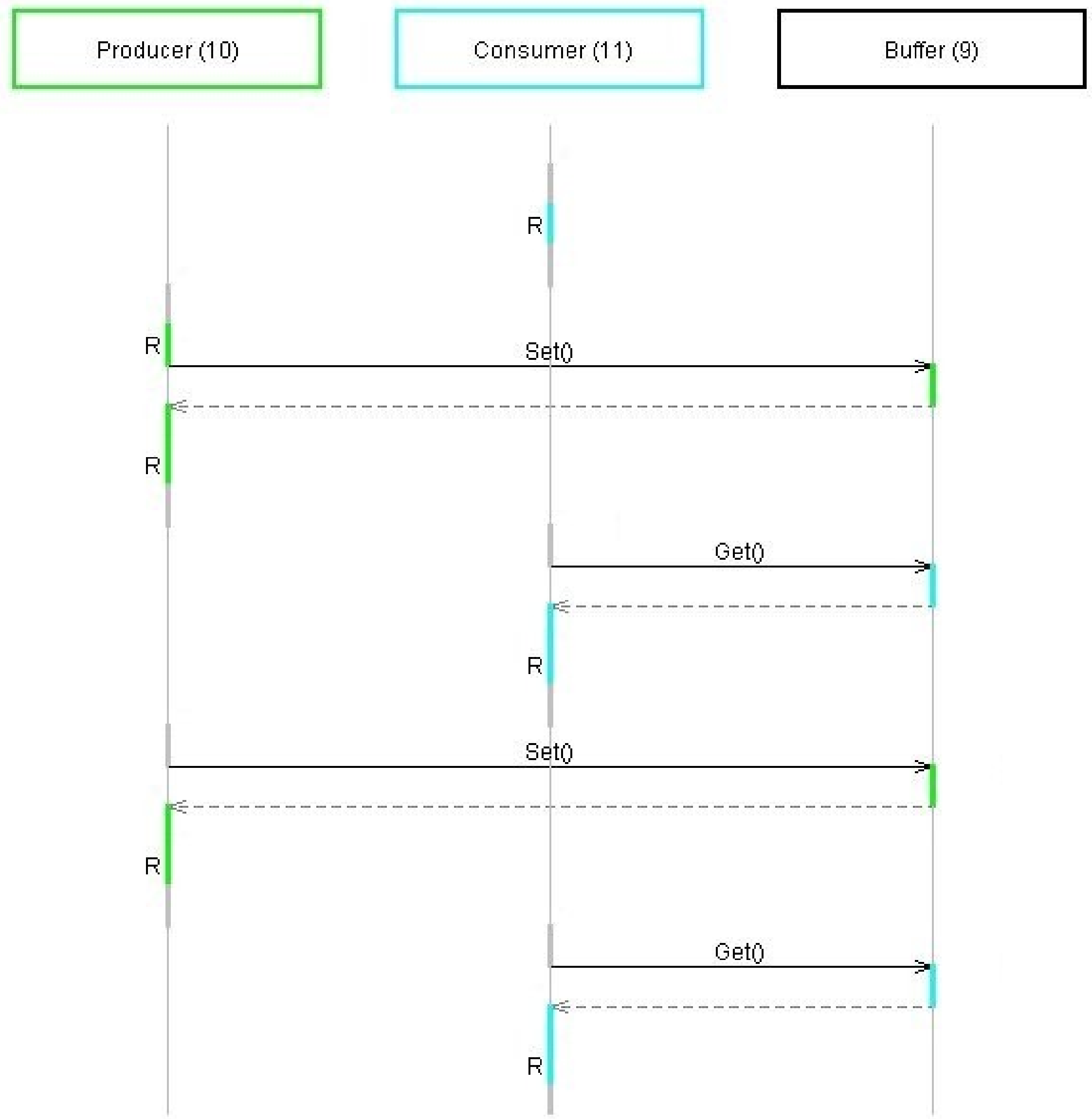
caveat: scheduling policy can be specified

VDM++ in a nutshell (3)

- Class
- Values
- Types
- Instance variables
- Functions
- Operations
- ***Threads***
- Synchronization

```
class Consumer

instance variables
  public theBuf : Buffer;
  private theVal : nat := 0

operations
  public Consumer: Buffer ==> Consumer
  Consumer (pBuf) == theBuf := pBuf

thread
  while true do
    theVal := theBuf.Get()

end Consumer
```

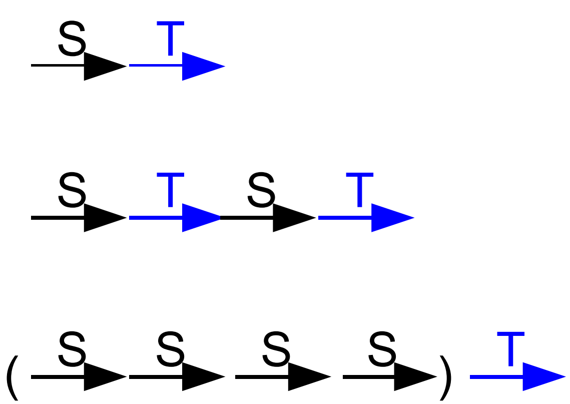
VDM++ in a nutshell (4)



Timed VDM++ (1)

- VDM++ plus
- ***(default) duration***
- periodic

informal semantics



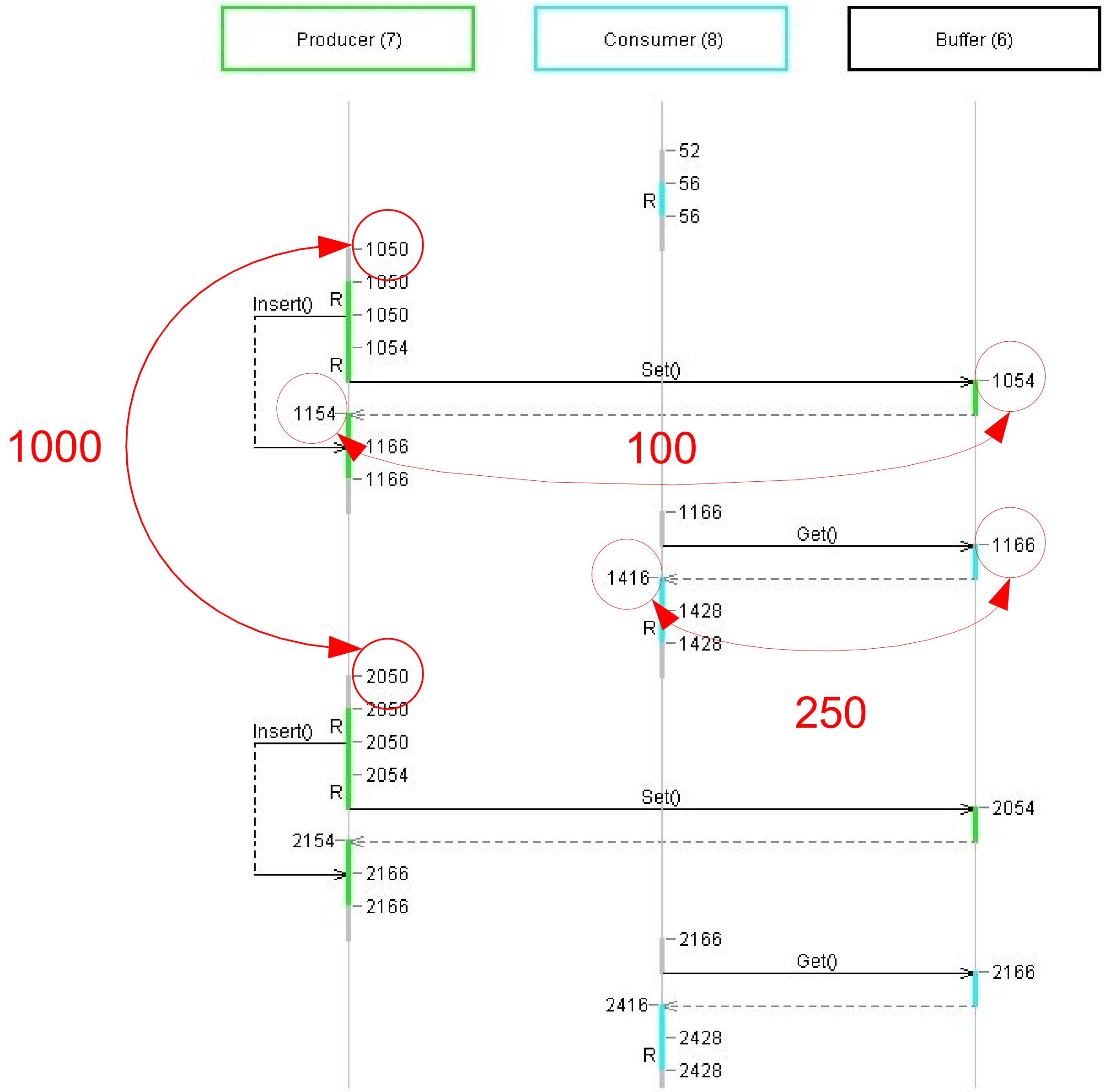

```
class Buffer

instance variables
   val : [nat] := nil

operations
   public Set: nat ==> ()
   Set (pv) ==
     duration (100) (val := pv);

   public Get: () ==> nat
   Get () ==
     duration (250)
        ( dcl res : nat := val;
          val := nil;
          return res )

sync
   per Set => val = nil;
   per Get => val <> nil;
   mutex (Get, Set)

end Buffer
```

Timed VDM++ (2)

- VDM++ plus
- (default) duration
- *periodic*

```
class Producer

instance variables
  public theBuf : Buffer;
  private theVal : nat := 0

operations
  public Producer: Buffer ==> Producer
  Producer (pBuf) == theBuf := pBuf;

  public Insert: () ==> ()
  Insert () ==
    ( theBuf.Set(theVal);
      theVal := theVal + 1 )

thread
  periodic (1000)(Insert)

end Producer
```

Timed VDM++ (3)



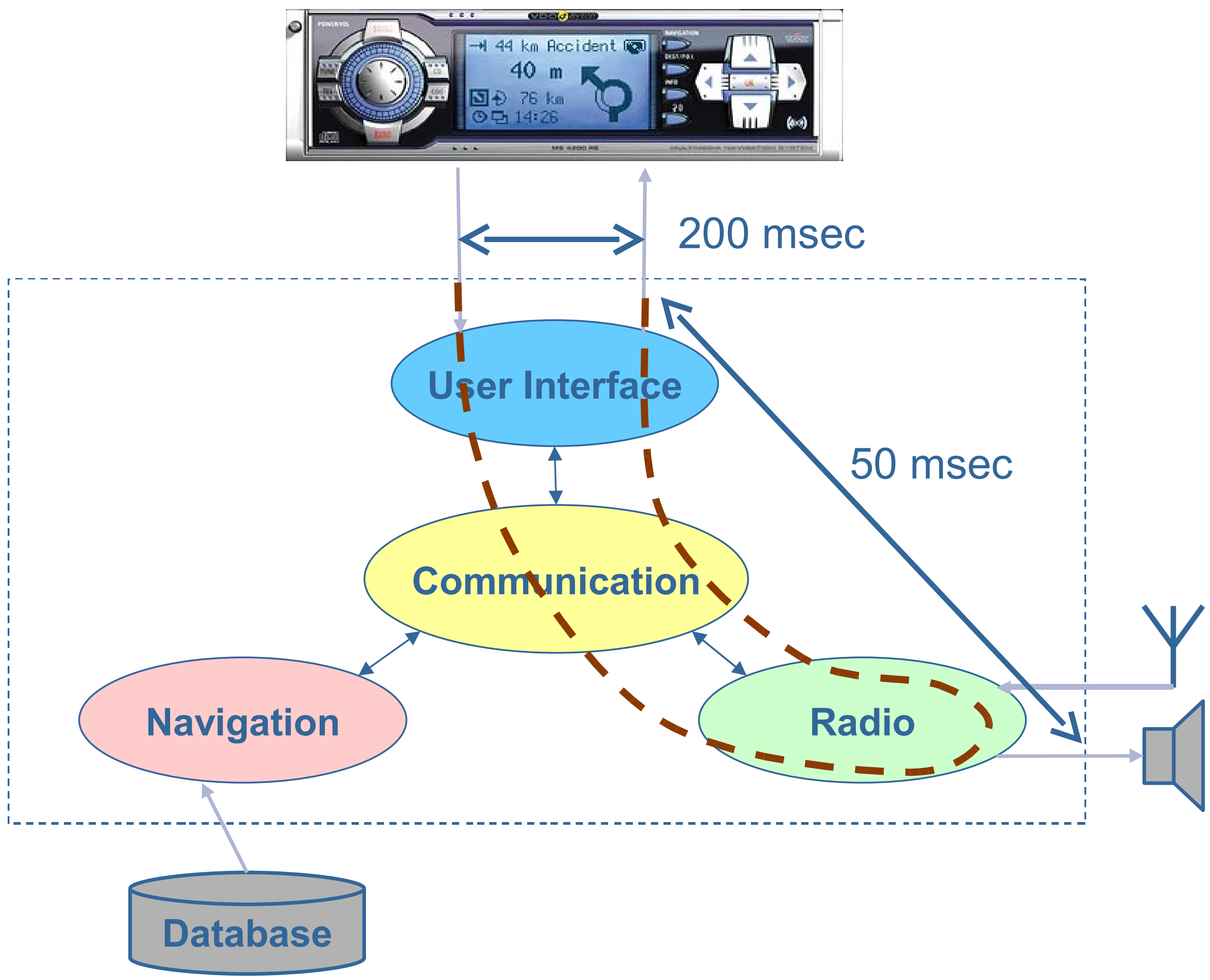
Producer (7)
Consumer (8)
Buffer (6)
52
56
R
56
1050
1050
Insert()
R
1050
1054
R
Set()
1054
1154
1000
1166
100
1166
1166
Get()
1166
1416
1428
R
1428
250
2050
2050
Insert()
R
2050
2054
R
Set()
2054
2154
2166
2166
2166
Get()
2166
2416
2428
R
2428

The In-Car Radio Navigation System

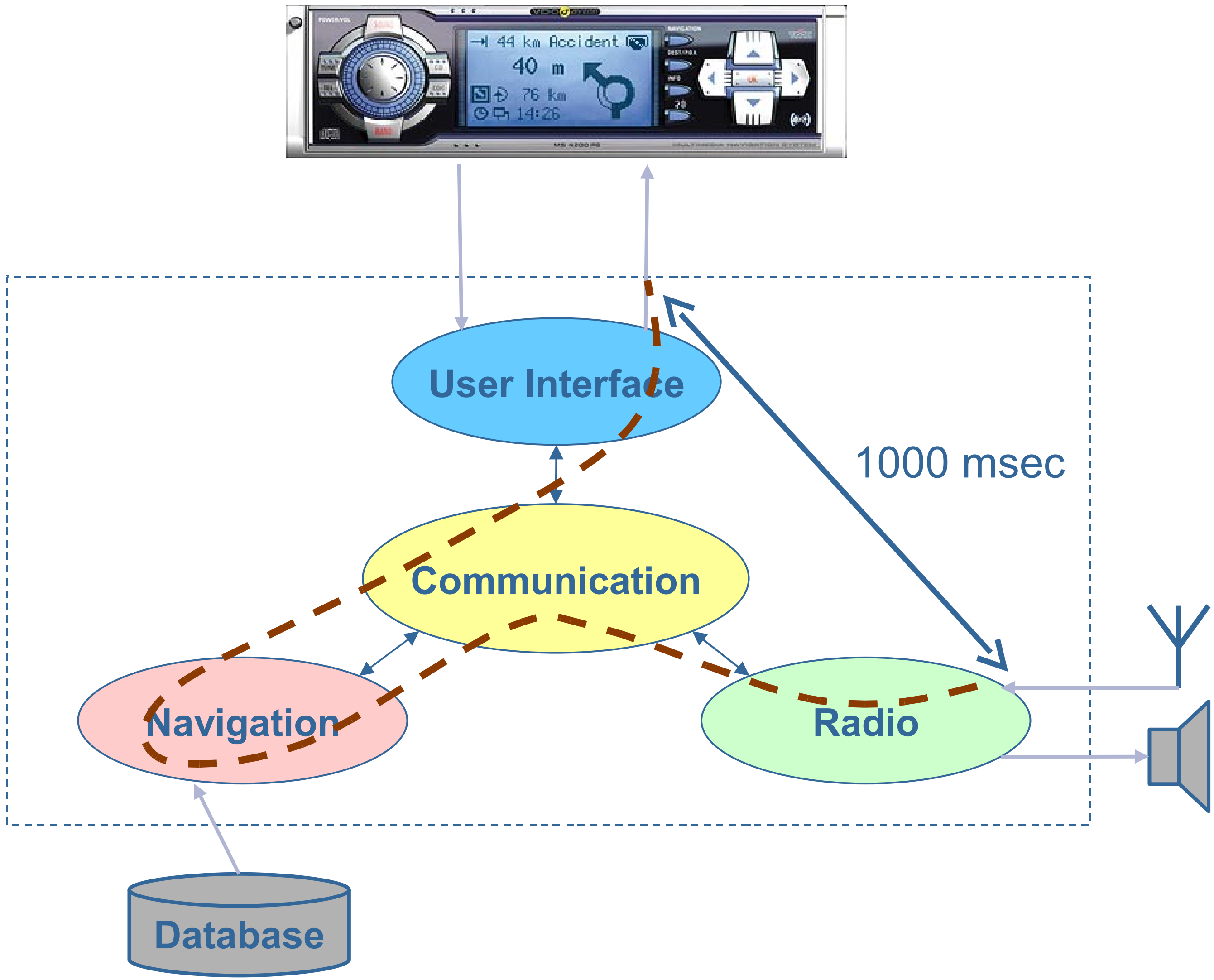
- Car radio with a built-in navigation system
- User-interface needs to be responsive
- TMC messages must be processed in a timely way
- Several applications may execute concurrently

http://people.ee.ethz.ch/~leiden05/data/pset/p2.pdf

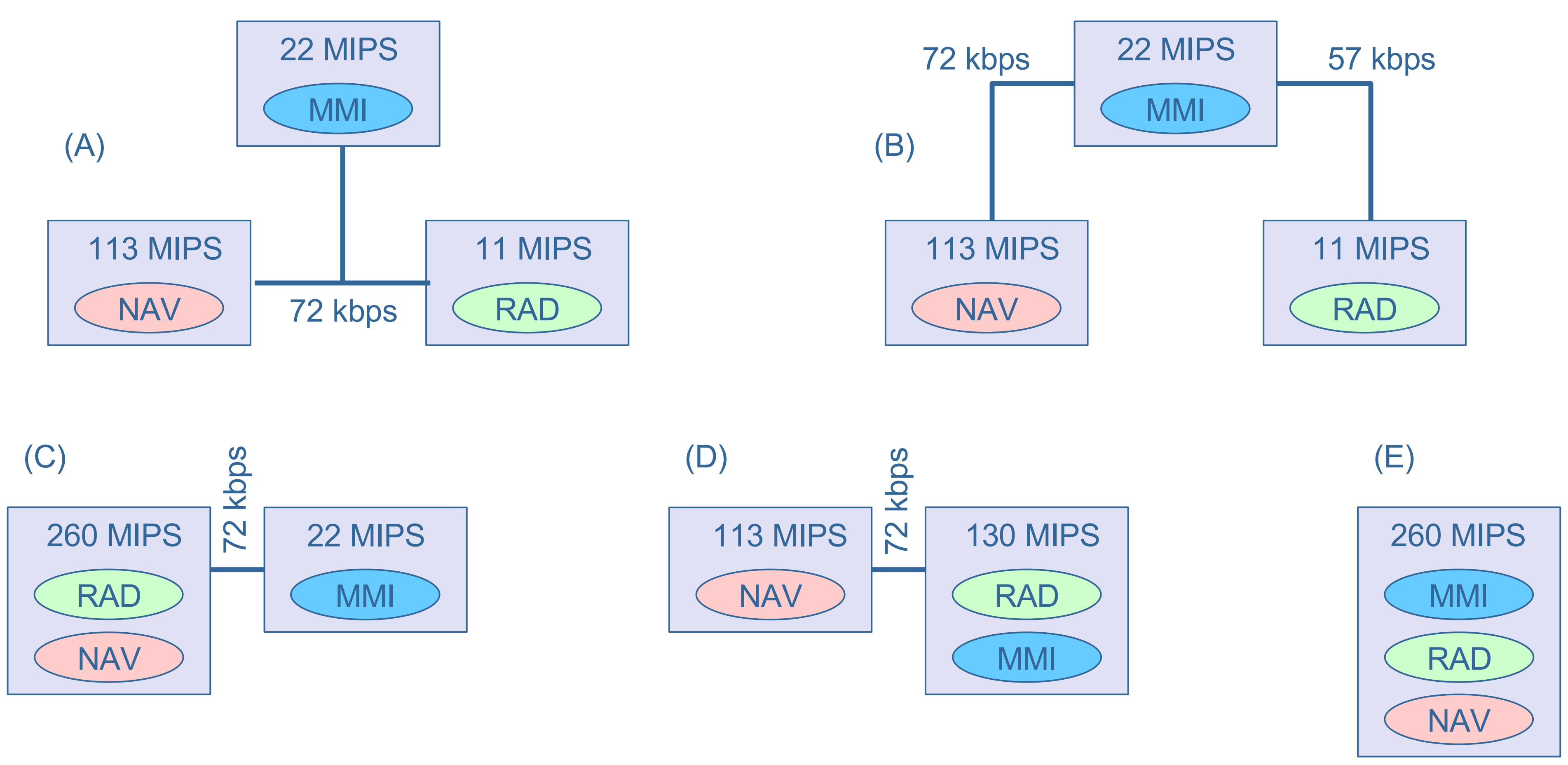
Change Volume Application



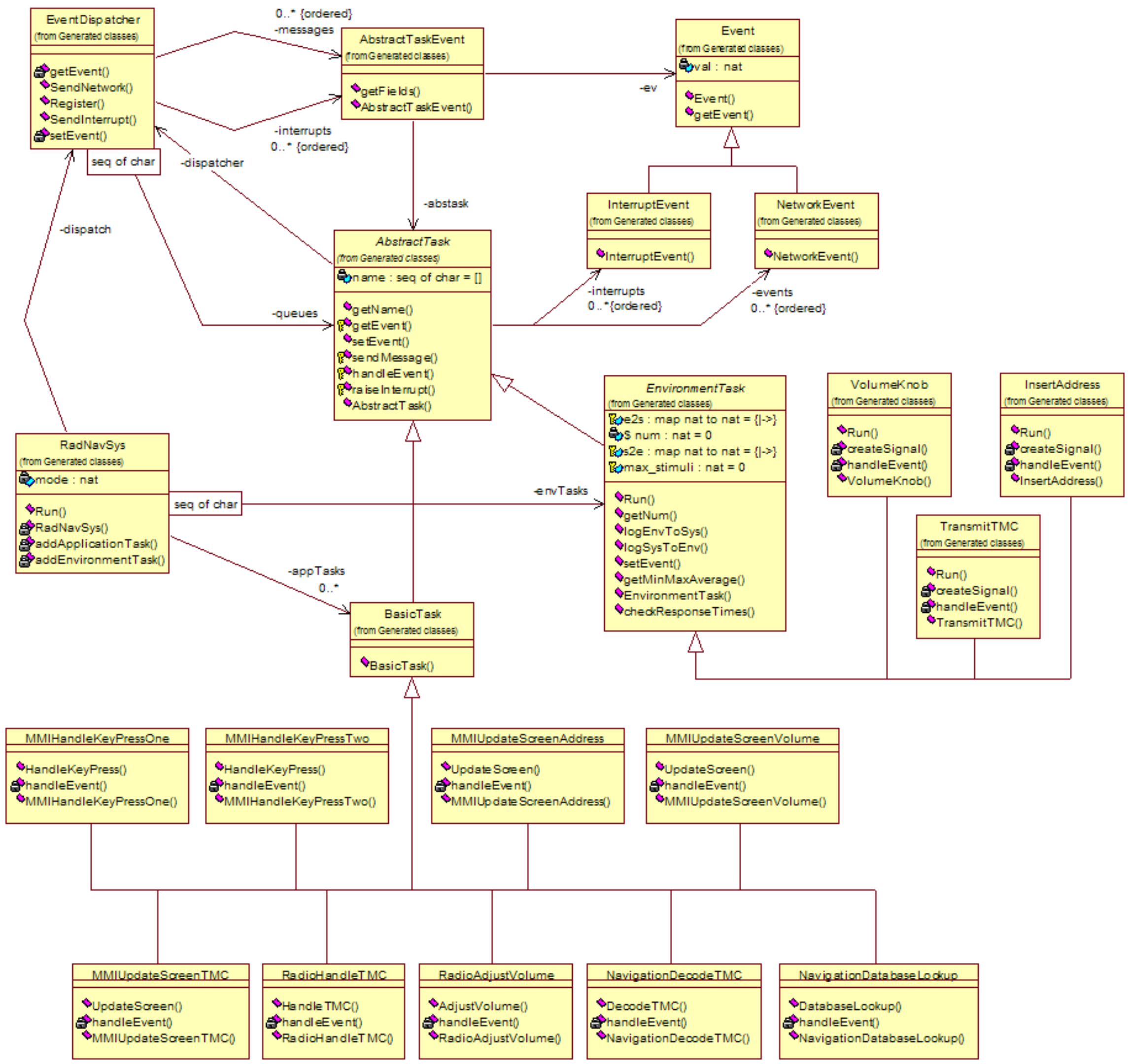
Handle TMC Application



Proposed Architectures



Modeling the case study in Timed VDM++



Problems encountered

- MoC (uni-processor multi-threading) is too restrictive
- Only synchronous operation calls are supported
- **duration** cannot be specified relative to capacity
- There is no means to capture the distributed architecture
- There is no notion of deployment
- Strict periodic behavior is assumed (no jitter)

Solutions proposed

- MoC: communicating multi-processor multi-threading
- Introduce asynchronous operations ("**async**")
- Introduce context aware time penalties ("**cycles**")
- Introduce **BUS** and **CPU** as first class citizens
- Class instances can be deployed on a specific **CPU**
- Introduce the **system** class to capture the architecture
- Allow non-strict periodic behavior (*p, j, d, o*)

Absolute and relative elapse time

```
class Radio

operations

  async public AdjustVolume: ... ()
  AdjustVolume (pno) ==
     ( duration (150) skip;
       RadNavSys`mmi.UpdateVolume(pno) );


  async public HandleTMC: nat ==> ()
  HandleTMC (pno) ==
     ( cycles (10000) skip;
       RadNavSys`navigation.DecodeTMC(pno) )

end Radio
```

CAN BE REPLACED BY AN ARBITRARY COMPLEX STATEMENT

Absolute and relative elapse time

```
class Radio

operations

  async public AdjustVolume: nat ==> ()
  AdjustVolume (pno) ==
     ( duration (150) skip;
       RadNavSys`mmi.UpdateVolume(pno) );


  async public HandleTMC: nat ==> ()
  HandleTMC (pno) ==
     ( cycles (10000) skip;
       RadNavSys`navigation.DecodeTMC(pno) )

end Radio
```

Composing the distributed architecture (1)

```
system RadNavSys

instance variables

  -- create the class instances
  static public mmi := new MMI();
  static public radio := new Radio();
  static public navigation := new Navigation();
```

Composing the distributed architecture (2)

```
...

  -- create the computation resources
  CPU1 : CPU := new CPU(<FP>, 22E6, 0);
  CPU2 : CPU := new CPU(<FP>, 11E6, 0);
  CPU3 : CPU := new CPU(<FP>, 113E6, 0);

  -- create the communication resource
  BUS1 : BUS := new BUS(<FCFS>, 72E3, 0,
                              {CPU1, CPU2, CPU3})
```

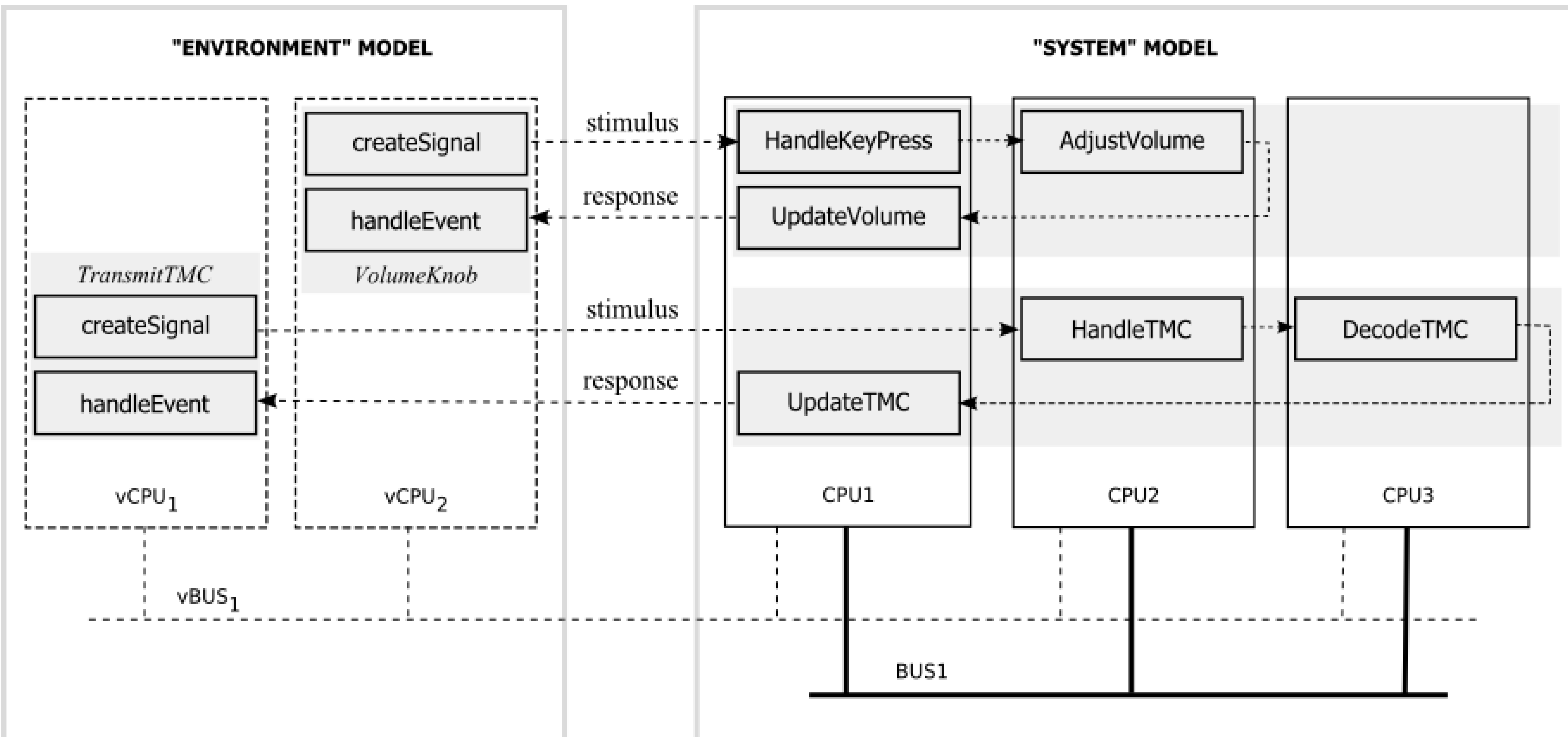
Composing the distributed architecture (3)

```
...

operations

  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( CPU1.deploy(mmi);
      CPU2.deploy(radio);
      CPU3.deploy(navigation) )

end RadNavSys
```

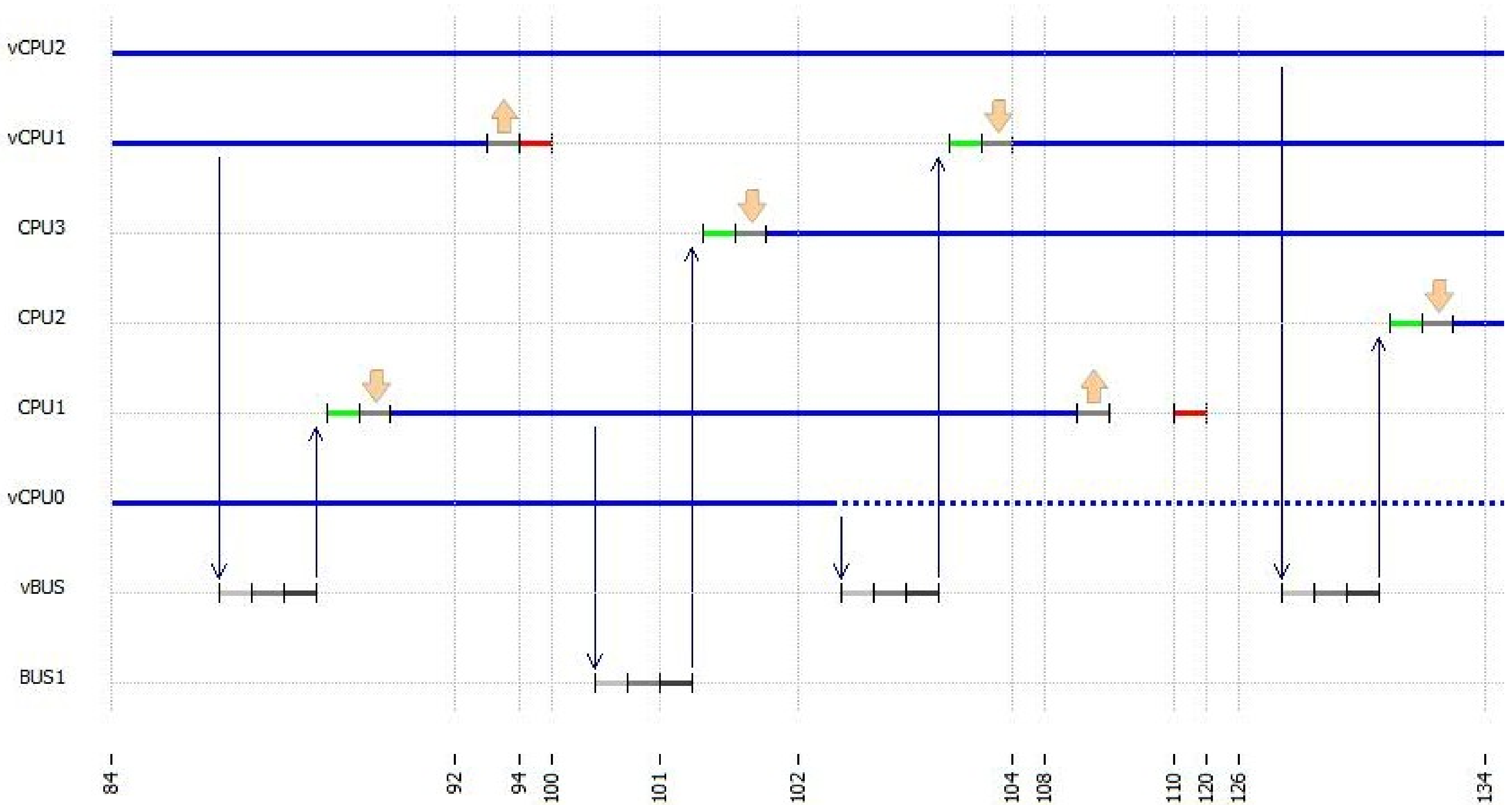
Modeling the environment

```
class TransmitTMC

...

operations
  async public handleEvent: nat ==> ()
  handleEvent (pev) == s2e := s2e munion {pev |-> time}
    post forall idx in set dom s2e &
           s2e(idx) - e2s(idx) <= 1000;

  async createSignal: () ==> ()
  createSignal () ==
    ( dcl num : nat := getNum();
      e2s := e2s munion {num |-> time};
      RadNavSys`radio.HandleTMC(num) )

thread
  periodic (3000, 4500, 1000, 0) (createSignal)

end TransmitTMC
```

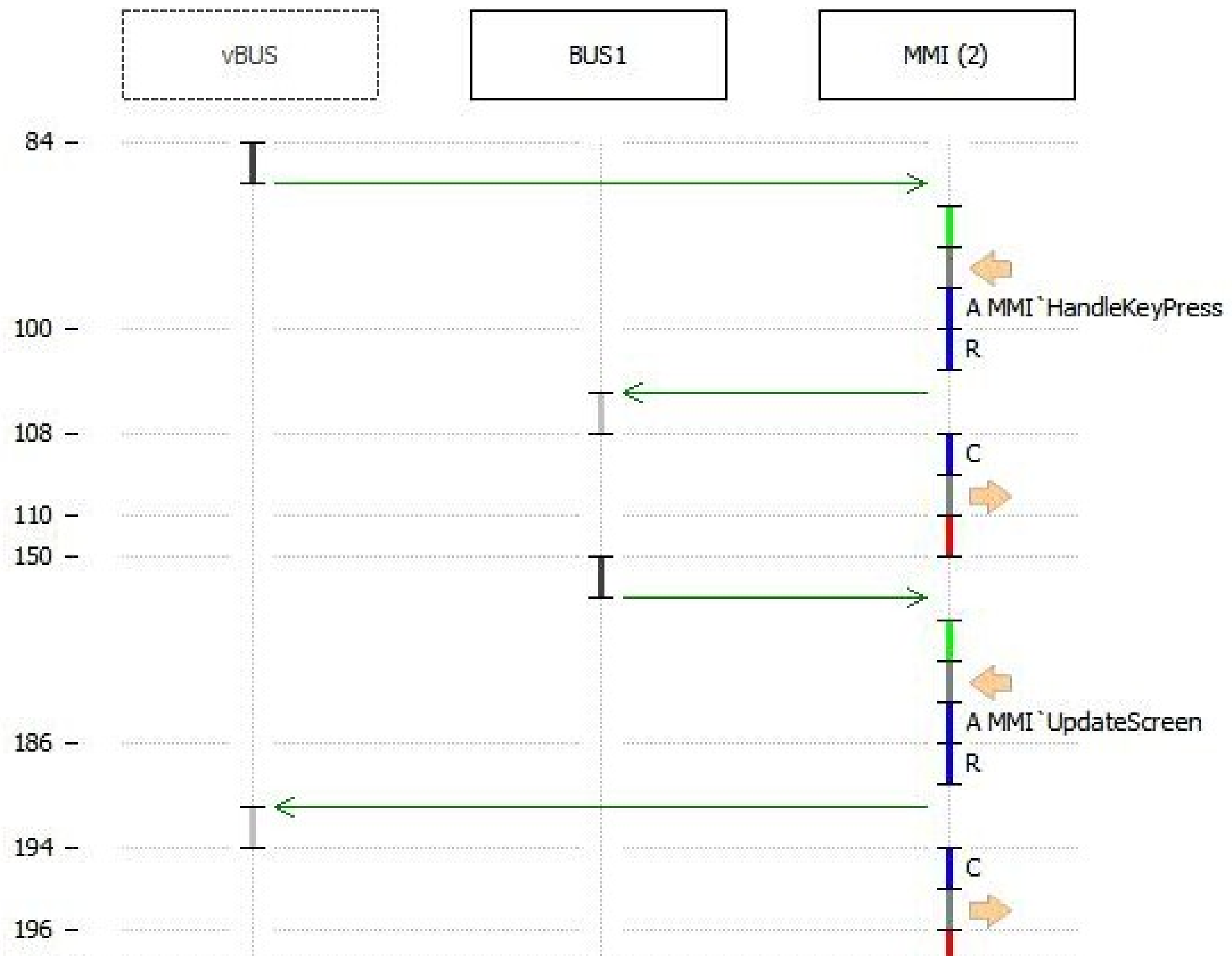
Case study – Summary and overview



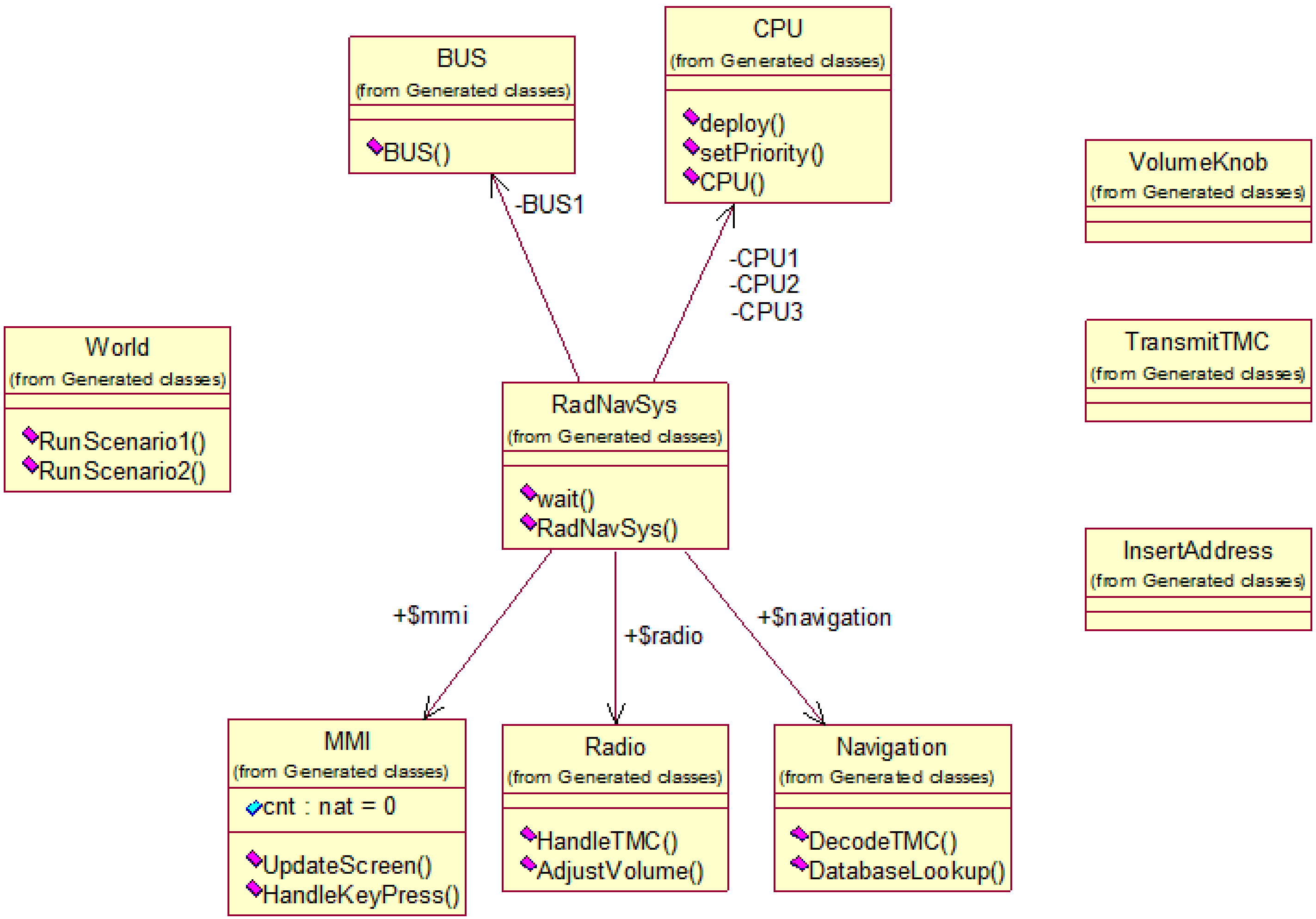
Symbolic execution (1)



Symbolic execution (2)



Some complexity removed...



On the use of formal techniques

- Abstract formal operational semantics for the new MoC
- *Not* specific to VDM++
- Machine checked with PVS
  `http://www.cs.ru.nl/~hooman/FM06.html`
- Executable Constructive Operational Semantics (COS) specified in VDM++
- Validated using VDMTools
  `http://www.cs.ru.nl/~marcelv/vdm/`
- COS merged into the VDMTools operational semantics specification (proprietary)

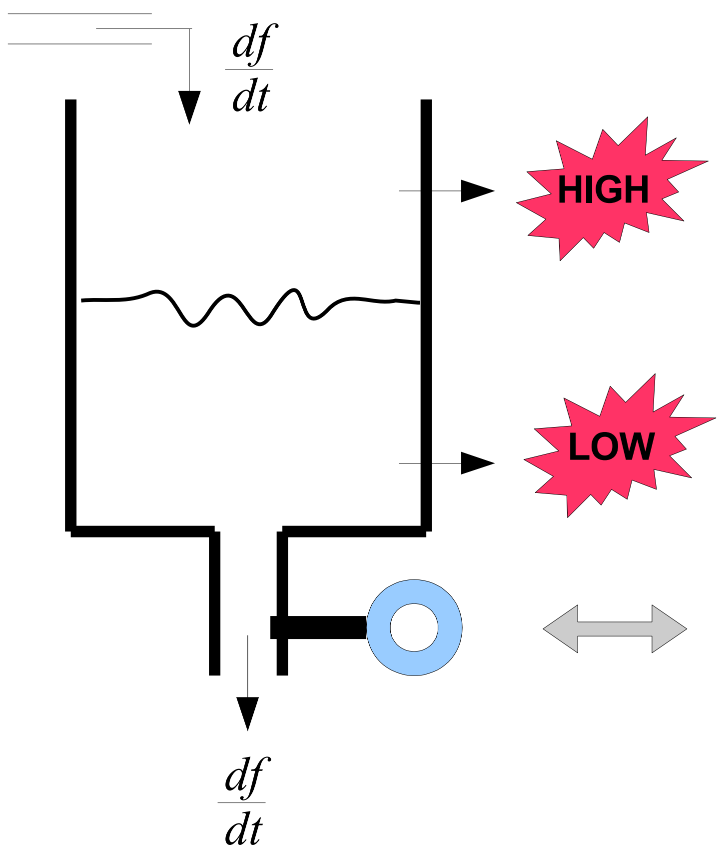
Results

- Significant decrease in model size
- Improved expressiveness, better domain applicability
- Minor syntactic changes – rather intuitive
- Major semantic changes – but “backwards compatible”
- Early exploration of deployment is now possible

Future work (1)

- On the notation
  - Duration as intervals, probabilities
  - From validation towards verification
  - History aware synchronization primitives
  - Explicit support for time-outs
  - Predicates over traces
- On the case study
  - Comparison to other techniques (MPA, UPPAAL, ...)
  - Comparison to measurements on real system

Future work (2)




```
class Controller
instance variables
  level : real := 0.0;
  valve : bool := false

operations
  public async open: () ==> ()
  open () == valve := true;

  public async close: () ==> ()
  close () == valve := false;

  public async update: () ==> ()
  update () ==
    if level < 2.0 then close()
    else if level > 3.0 then open()

threads
  periodic (0, 1000, 0, 0) (update)

end Controller
```