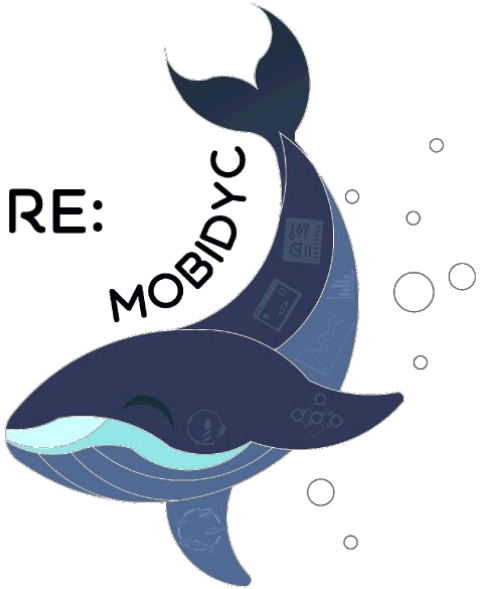
Implementation-First Approach of Developing Formal Semantics of a Simulation Language in VDM-SL



Tomohiro Oda, Software Research Associates, Inc.
Gael Dur, Shizuoka University
Stephane Ducasse, University of Lille
Hugo Daniel Macedo, Aarhus University

re:mobidyc

the overview

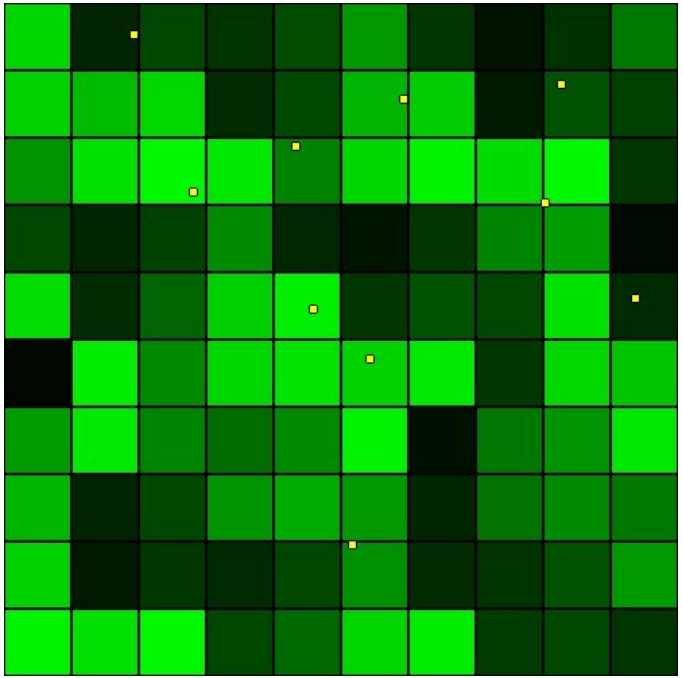
re:mobidyc

A multi-agent simulation platform for population dynamics in biology and ecology.

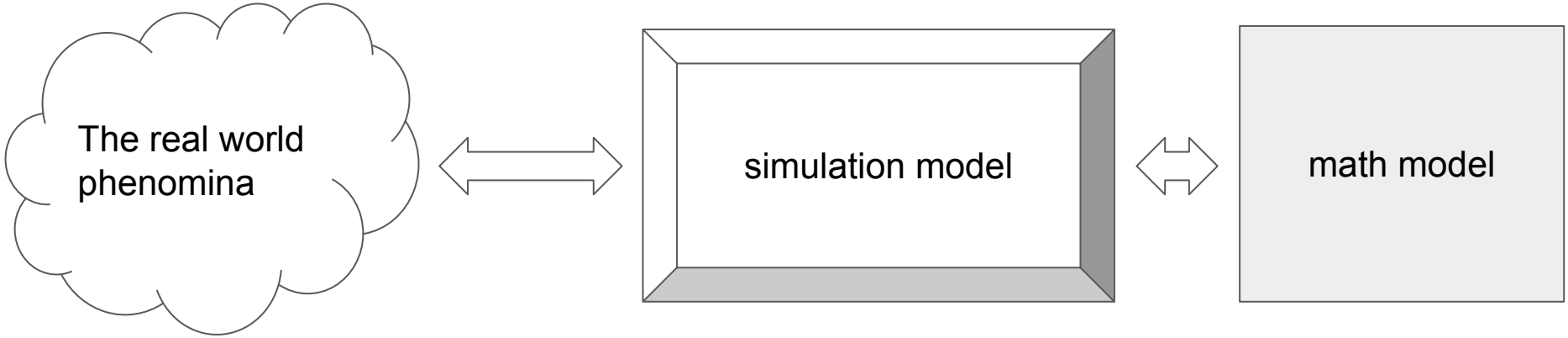
Major competitors = variants of LOGO

re:mobidyc

- supports domain-specific language features
- demands less "programming" skills
- serves as a tool for science



The execution of simulation model is not the goal, but the math model is.



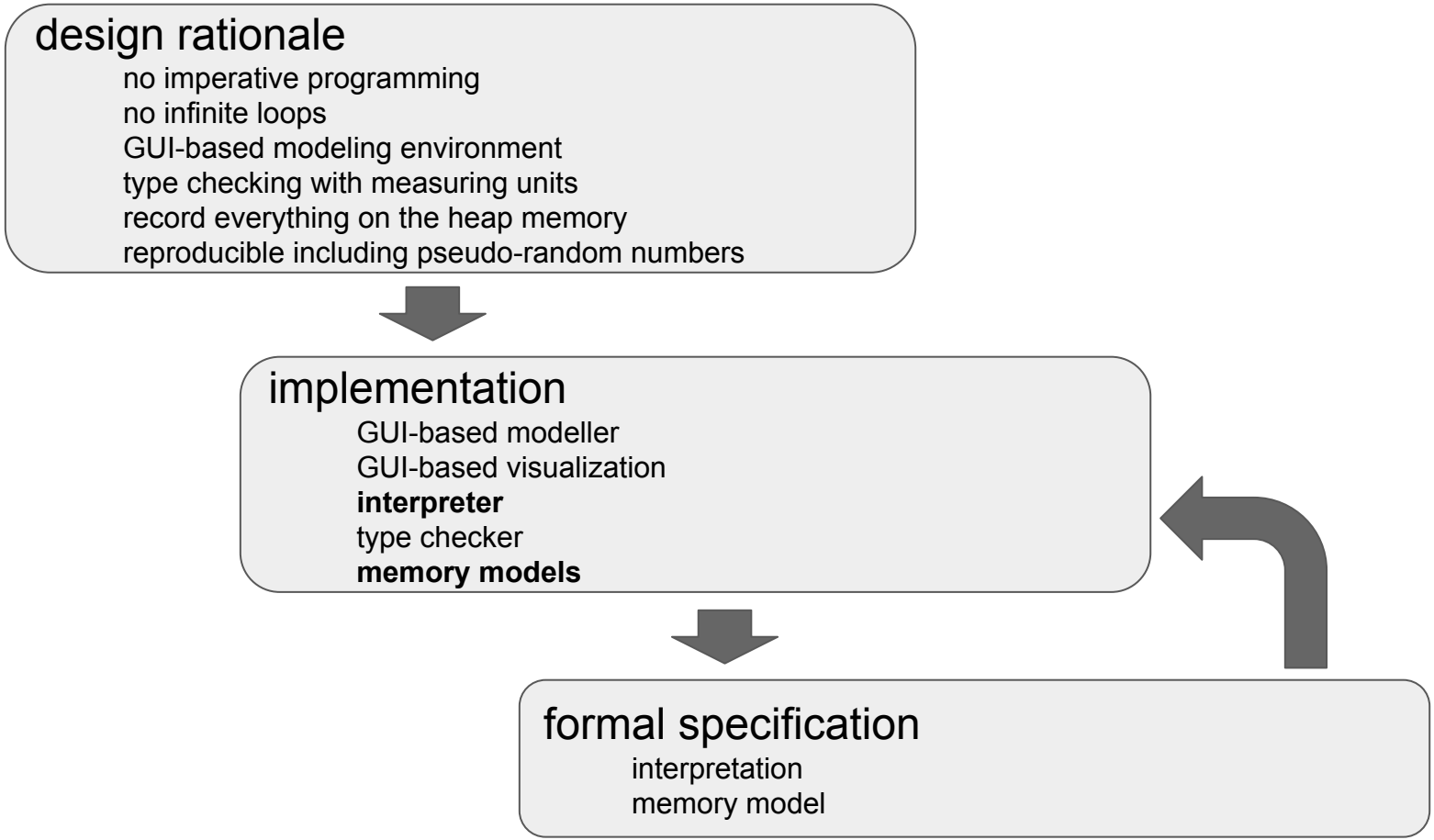
The simulation model reproduces and explains the real world phenomina.

The simulation model implements the math model.

Implementation First

the semantics and its interpreter

development process of re:mobidyc



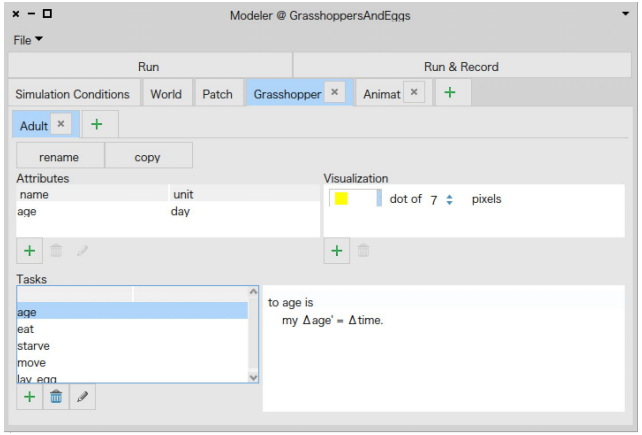
Why Formal Semantics?

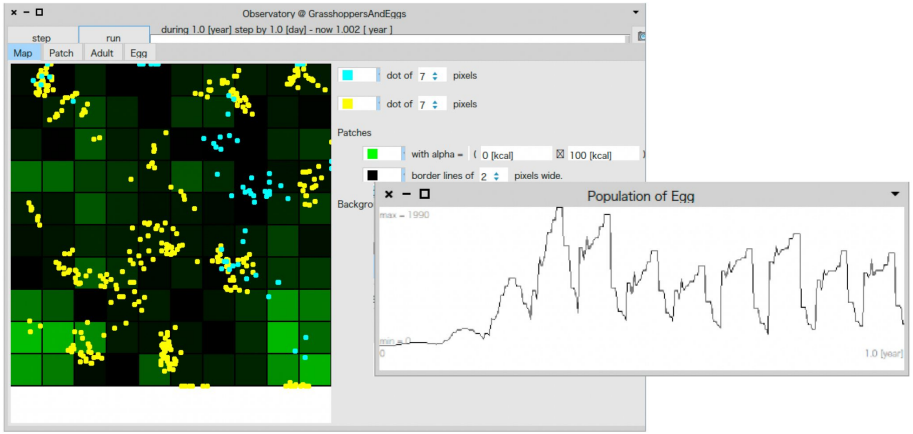
- grounding to mathematics
  - The goal of the user is to develop and validate a MATH MODEL in biology.
- portability
  - The formal semantics makes it easy to implement a fully compatible interpreters/transpilers.
  - The re:mobidyc modeling language should not depend on our implementation.
- uncommon memory model
  - The memory model with synchronous updates for time-series data is not common among existing programming languages and therefore needs concise and unambiguous definition.

with expectations to improve code quality

Why not semantics first?

- GUI-based modeling environment and execution/analysis tools





- **while** true **do**

  (Computer_scientists`implement_a_language_feature();
  Biologists`try_it())

Specification

VDM-SL modules

- AST.vdmsl (ASTTest.vdmsl)
  - 50 records, 20 unions and 19 constants
  - 7 functions to manipulate AST
- Evaluation.vdmsl (EvaluationTest.vdmsl)
  - 9 eval operations, 3 variable access operations and 10 primitive operations
- Interpreter.vdmsl (InterpreterTest.vdmsl)
  - 27 operations to manage simulation models, evaluation contexts and random seeds
- **Memory.vdmsl** (MemoryTest.vdmsl)
  - read/write operations, synchronous updates and snapshotting
- Random.vdmsl (RamdomTest.vdmsl)
  - Fishman-Moore random number generator
- Unit.vdmsl (UnitTest.vdmsl)
  - Measurement units
- MATH.vdmsl
- UnitTesting.vdmsl

state definition of the memory model

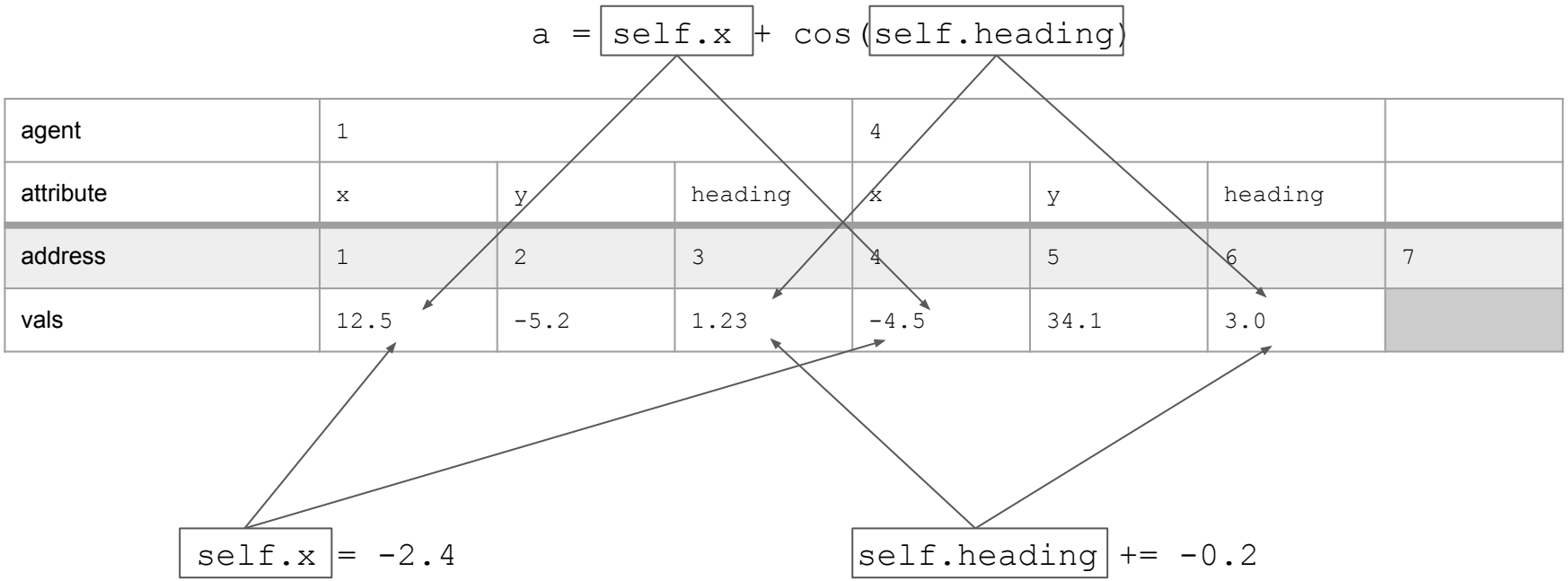
```
state Memory of
    vals : map Address to real
    next : map Address to real
    delta : map Address to real
    nextAvailableSlot : [Address]
    world : Address
    patchBase : [Address]
    xDivisions : [nat1]
    yDivisions : [nat1]
    animats : map Address to (AST`Identifier * nat1)
    newBorns : map Address to (AST`Identifier * nat1)
    deads : set of Address
    valuesStorage : seq of (map Address to real)
    animatsStorage : seq of (map Address to (AST`Identifier * nat1))
    ticks : nat
init s == s = mk_Memory({|->}, {|->}, {|->}, nil, 1, nil, nil, nil, {|->}, {|->}, {}, [], [], 0)
end
```

Heap memory
with synchronous update

Agent allocations

Time-series storage

Conventional memory model



The resulting value of `a` depends on the order of execution of the scripts.

three cells per address

```
read : Address ==> real
read(address) ==
    if address in set dom vals
    then return vals(address)
    else exit ADDRESS_ERROR
```

a = my x + cos(my heading)

agent		1			4			
attribute		x	y	heading	x	y	heading	
address		1	2	3	4	5	6	7
vals	read	12.5	-5.2	1.23	-4.5	34.1	3.0	
delta	write	12.5	-5.2	-0.2	-4.5	34.1	-0.2	
next	add	-2.4	0.0	0.0	-2.4	0.0	0.0	

my x' = -2.4[m]

my Δheading' = -0.2[rad]

```
write : Address * real ==> ()
write(address, data) ==
    next(address) := data
```

```
writeDelta : Address * real ==> ()
writeDelta(address, data) ==
    if address in set dom delta
    then delta(address) := delta(address) + data
    else exit ADDRESS_ERROR;
```

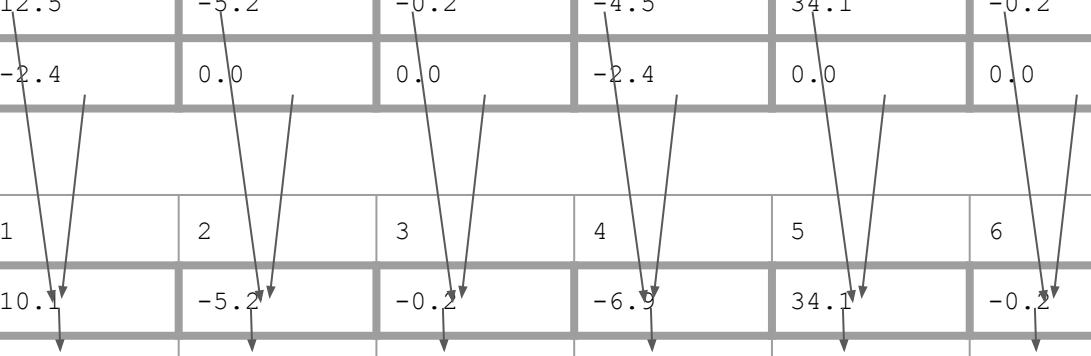
synchronous update

time 354

address		1	2	3	4	5	6	7
vals	read	12.5	-5.2	-0.2	-4.5	34.1	3.0	
next	write	12.5	-5.2	-0.2	-4.5	34.1	-0.2	
delta	write	-2.4	0.0	0.0	-2.4	0.0	0.0	

time 355

address		1	2	3	4	5	6	7
vals	read	10.1	-5.2	-0.2	-6.9	34.1	-0.2	
next	write	10.1	-5.2	-0.2	-6.9	34.1	-0.2	
delta	write	0.0	0.0	0.0	0.0	0.0	0.0	



{addr |-> next(addr) + (if addr in set dom delta then delta(addr) else 0) | addr in set dom next \ deads}

time-series memory

time 355

address		1	2	3	4	5	6	7
vals	read	10.1	-5.2	-0.2	-6.9	34.1	-0.2	
next	write	10.1	-5.2	-0.2	-6.9	34.1	-0.2	
delta	write	0.0	0.0	0.0	0.0	0.0	0.0	

```
store : () ==> ()
store() ==
    (valuesStorage := valuesStorage
        ^ [{a |-> next(a) + (if a in set dom delta then delta(a) else 0)
            | a in set dom next \ deads}];
    animatsStorage
        := animatsStorage ^ [deads <-: animats munion newBorns])
pre  ticks = len valuesStorage and ticks = len animatsStorage;
```

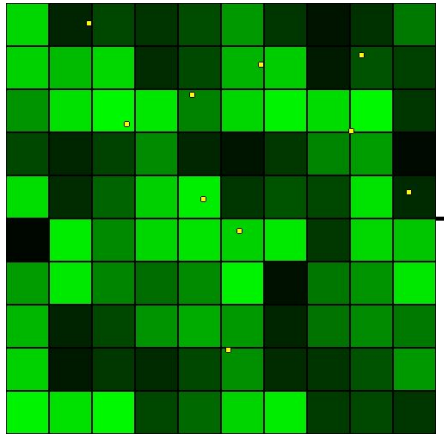
backend storage (on-memory, file system, null, ...)

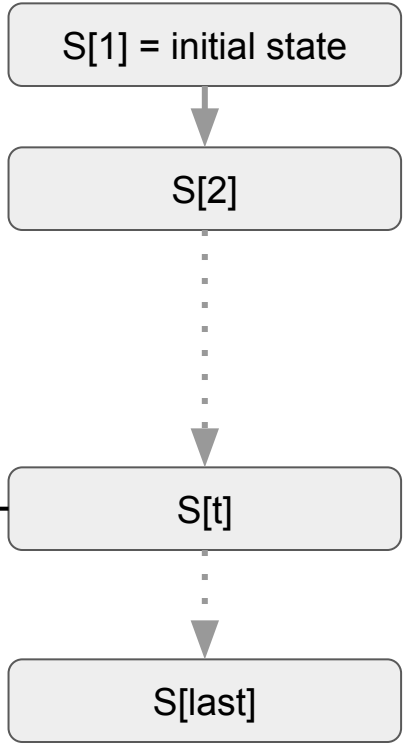
address \ time	1	2	3	4	5	6	...
...	...	...	...	...	...	...	...
350	22.1	-5.2	-4.2	5.1	34.1	-1.5	...
351	19.7	-5.2	-4.2	2.7	34.1	2.3	...
352	17.3	-5.2	3.2	0.3	34.1	5.1	...
353	14.9	-5.2	-5.2	-2.1	34.1	-5.2	...
354	12.5	-5.2	1.23	-4.5	34.1	3.0	...
355	10.1	-5.2	-0.2	-6.9	34.1	-0.2	...
356							
...							

Memory model for re:mobidyc

The objective of re:mobidyc is to enable the user to analyze
- what happens
- why it happens
- how it happens

We need a memory model with
- lightweight snapshot
  - dump heap space at every timestep
  - to trace cause and effect
- synchronous update
  - delay write to memory to the interval of timestep
  - to isolate effects of each action and
  - to eliminate intermediate state



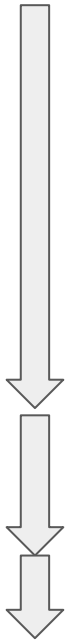


Implementation First Revisited

the development process

development of re:mobidyc in numbers

event	date	Pharo LOC (all)	Pharo LOC (interp)	Pharo tests	VDM LOC	VDM tests
Implementation started	Oct 2019	-	-	-	-	-
	Jan 2020	1,499	1,150	21	-	-
	Jan 2021	12,936	7,268	214	-	-
	Jan 2022	19,474	9,526	276	-	-
Specification started	Aug 2022	26,330	11,990	320	0	0
	Dec 2022	30,114	13,205	338	1,364	113



Thoughts on implementation-first Lightweight Formal Method

- The first exploration in Pharo took long time.
- Just writing a formal specification did not cost much.
  - The specification took only 4 months in 3 years development so far.
- The specification in VDM was compressed into 10% of the implementation
  - 13,205 LOC in Pharo → 1,364 LOC in VDM

What if we started with the specification phase first?
... In the development of ViennaVM,

- The first exploration in VDM took long time and just writing C did not cost much.

Explorative Development Process

Front loading effect:

> Developing a formal specification reduces the cost of implementation because semantic errors of functionalities are eliminated at the specification phase and the implementers can get focused on implementation issues.

re:mobidyc project:

> Developing a prototypical implementation reduced the cost of specification because mis-assumptions on the problem domain were eliminated at the prototyping phase and the specifiers could get focused on semantic issues.

Conclusion

Just writing a formal specification is not costly, but the exploration process is.

- Learning the problem domain
- Finding affordable solution
- Planning for realisation

Implementation-first approach:
Models in VDM as a summary of the exploratoration by implementation languages, and also as a pivot to the next iteration of the development cycle.

- Some of software developers are afraid of **cost and risk** of adopting formal specification.
- The apparent cost of formal specification can be compressed. (10% in re:mobidyc)
- Even if VDM modeling does not go well, the development can go without a formal specification.

Thank you!