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

Tomohiro Oda[1], Gaël Dur[2], Stéphane Ducasse[3], and Hugo Daniel Macedo[4]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] Creative Science Unit (Geoscience), Faculty of Science, Shizuoka University (dur.gael@shizuoka.ac.jp)
[3] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 – CRIStAL, France (stephane.ducasse@inria.fr)
[4] Aarhus University, DIGIT, Department of Electrical and Computer Engineering, (hdm@ece.au.dk)

Abstract. Formal specification is a basis for rigorous software implementation. VDM-SL is a formal specification language with an extensive executable subset. Successful cases of VDM-family including VDM-SL have shown that producing a well-tested executable specification can reduce the cost of the implementation phase. This paper introduces and discusses the reversed order of specification and implementation. The development of a multi-agent simulation language called RE:MOBIDYC is described and examined as a case study of defining a formal specification after initial implementation and reflecting the specification into the implementation code.

1 Introduction

Lightweight formal methods are partial applications of formal methods so that a formal specification of the whole or a part of the software system provides the implementation with clear goal conditions of its functionality [5,1]. A formal specification consists of concise and unambiguous definitions of the system's functions which allow implementors to focus on the correctness and efficiency of the implementation. VDM-SL [6] is one of the oldest formal specification languages which are still in practical use. VDM-SL has an extensive subset that can be executed by interpreters, which enables unit testing of specifications. Successful cases of the VDM-family [4,8] have shown that producing a well-tested executable specification benefits the implementation phase. Unit testing ensures that the algorithms defined in the specification work as expected in the test cases. Test cases for the VDM specification can also be rewritten in implementation languages so that the implementation works the same as the executable specification.

While the specification is deemed to be written before the implementation in traditional views of software lifecycles, the implementation sometimes precedes the specification in practice. Development of a GUI application is sometimes driven by GUI prototypes before the developers capture whole the required functionality of the application. This then raises the following questions. Does it worth writing a formal specification even if the implementation is already running? How much does it cost to write a concise specification based on exploratory implementation?

This paper describes a development where the formal specification was written after the implementation. RE:MOBIDYC [9] is a multi-agent simulation system for population dynamics in biology. GUI-based construction of the model instead of textual coding and the operational semantics of the modeling language in VDM-SL were planned from the beginning of the development. The operational semantics can be viewed as the formal specification of the interpreter. The implementation of a GUI-based IDE including an interpreter was started without the formal semantics. The implementation was started first because the GUI-based construction of the model was needed by biologists to design the modeling language. Language features such as life-history stages, moulting and reproduction were implemented in exploratory manners involving biologists. The formal specification of the modeling language was defined after the language features had become stable. The formal specification was refactored into a more concise presentation and the implementation was improved by reflecting the refactored specification.

In Section 2, the design and implementation of re:mobidyc and its modeling language are explained. Its development process will be described in Section 3. Findings from the development will be discussed in Section 4, and Section 5 concludes the paper.

2 RE:MOBIDYC and its modeling Language

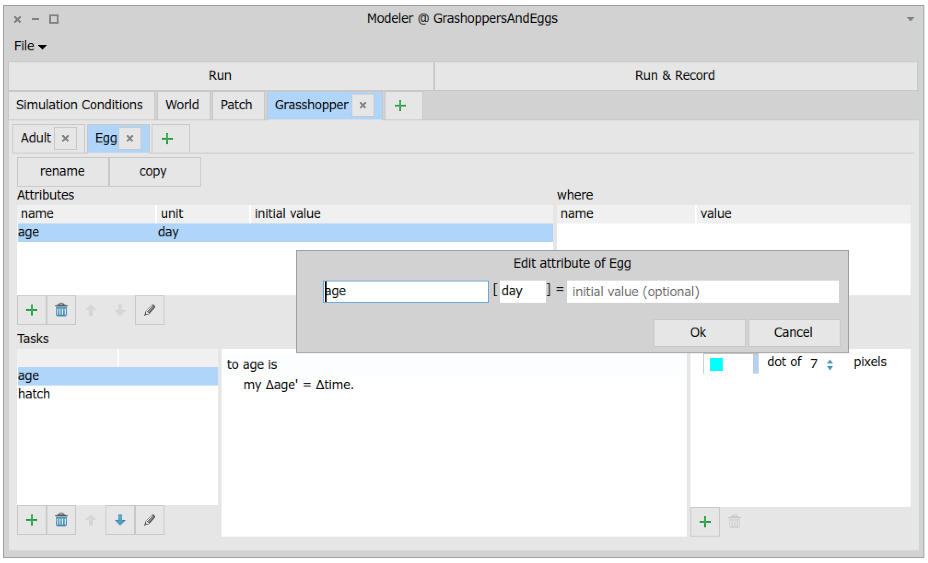
RE:MOBIDYC is a multi-agent simulation platform for the scientific study of biology and ecology, built upon Pharo [9]. In multi-agent simulation, agents with simpler definitions interact with each other to exhibit complex phenomena.

Followings are RE:MOBIDYC's major design principles to support the user's tasks.

- DP1. The language should be declarative and should not impose imperative programming.
- DP2. The language must guarantee the computation always terminates or aborts.
- DP3. The system should provide GUI-based interfaces to define a model.
- DP4. The language should provide semantic checking based on measurement units.
- DP5. The system should record all attribute variables of all agents at all time steps.
- DP6. The system should produce the same result from the same model.

DP1 and DP2 are to support biologists and ecologists who are not necessarily familiar with imperative programming languages. In general, it is hard to verify whether or not an imperative program with assignments and conditional loops implements the mathematical model at hand. The termination property is also hard to verify. DP2 implies that the language will not be Turing complete. In return, the model will be free of incidental infinite loops. DP3 and DP4 help the user to reduce the burdens of syntax and semantic errors in the model. Fig. 1 shows the GUI-based modeler for DP3. The basic idea of DP4 is to use measurement units as static types to detect semantically ill-formed expressions in the model. DP5 and DP6 are for the verifiability and reproducibility of simulation results to make the simulation more useful as a tool for scientific research.

RE:MOBIDYC is built upon Pharo [3,2], a modern dynamic object-oriented language derived from Smalltalk, which provides with flexible and immersive programming environment on macOS, Linux and Windows systems. Fig. 2 illustrates the software configuration of the RE:MOBIDYC simulation environment. It shows three major UI elements,



Attributes

name	unit	initial value
age	day	

Edit attribute of Egg: age [day] = initial value (optional)

Tasks: age, hatch

```
to age is
   my Δage' = Δtime.
```

dot of 7 pixels

Fig. 1. Screenshot of RE:MOBIDYC's GUI-based modeler

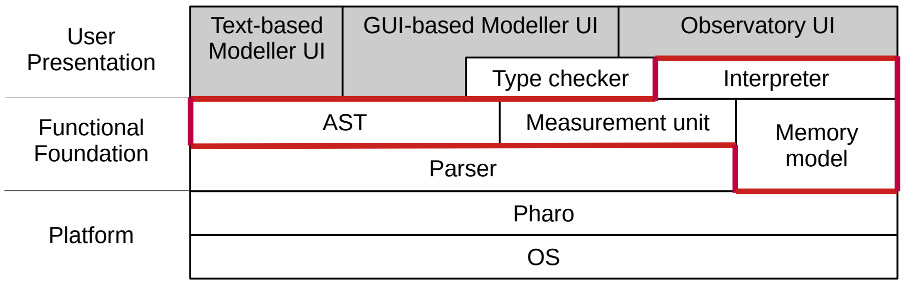


Fig. 2. Major components of RE:MOBIDYC simulation environment

namely text-based modeler, GUI-based modeler and Observatory which runs a simulation and visualises the result. Functional components operated by the user such as type checker and interpreter are also built on the fundamental elements such as the abstract syntax tree (AST), measurement units and the memory model. Among the UIs, text-based modeler UI is provided as a backup to GUI-based modeler UI in the case that the model definition files are broken by some reason. The scope of the formal specification in VDM-SL was limited to the components enclosed by the red lines in Fig. 2. The specification phase was started after their first implementation in Pharo had been done and the language had become stable enough.

The development of RE:MOBIDYC required a GUI-based modeling environment and exploratory design of the modeling language because the language must be validated by experts from biology and ecology. Pharo's dynamic nature enables agile prototyping to quickly develop the GUI-based modeler and the interpreter. The modeling language and its interpreter including the memory model are developed with the implementation in Pharo first, and then its formal semantics was defined in VDM-SL. The development is still ongoing in both implementation and formal semantics as an open-source project.

2.1 Overview of the Language Design

RE:MOBIDYC is designed friendly to mathematical modelers in biology and ecology, less demanding programming skills[9]. The models in RE:MOBIDYC are constructed in the declarative manner so that the models look more like definitions of numeric sequence in the form of recurrence relation than a series of imperative statements.

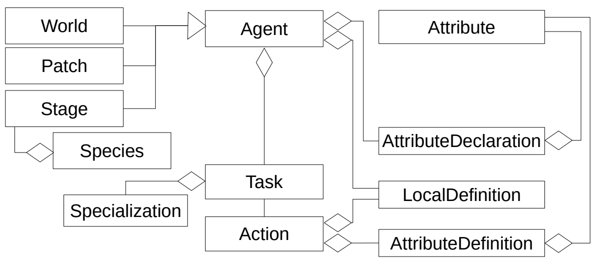


Fig. 3. Class diagram of re:mobidyc's modeling language AST

Fig. 3 illustrates the major constructs of the modeling language. Agent, action, task and attribute are the most significant constructs in the modeling language. RE:MOBIDYC has three kinds of agents: World, Patch, and Stage. World is the agent that represents the global environment, and a Patch represents the local environment. A stage is also called *animat*, which has two attributes x and y so that they can move around in the two-dimensional space. A stage represents a life-history stage of an individual life such as larva, juvenile, and adult which together represent an individual. An individual is modeled as a *species*.

Each agent has a set of *attribute declarations* each of which specifies the identifier, the measurement unit and optionally the initial value of the attribute allocated in the persistent heap memory. An interaction among agents is modeled as an *Action* which consists of a set of *attribute definitions*, each of which modifies the next value of the attribute of an animat that participates in the interaction. Local variables are defined in the *utility definitions* so that right-hand expressions in attribute declarations and attribute definitions are concisely presented. A task definition binds an action to an agent so that each individual of the agent will do the action at every time step in the simulation.

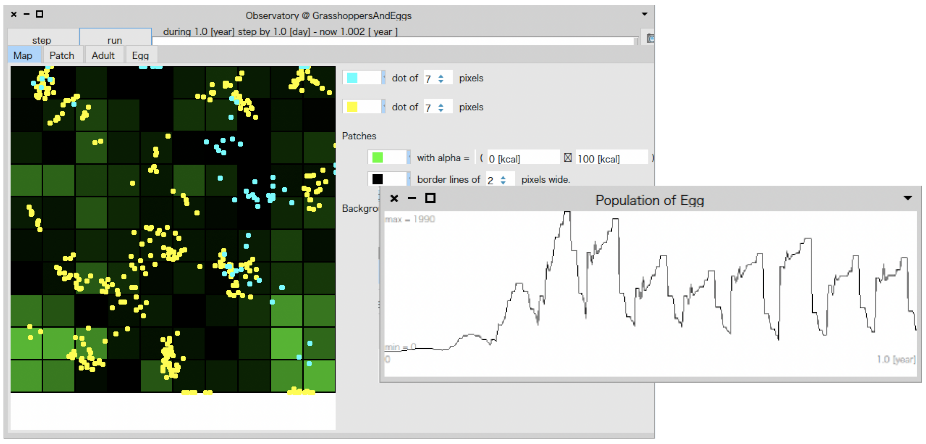

Observatory @ GrasshoppersAndEggs
step
run
during 1.0 [year] step by 1.0 [day] - now 1.002 [year]
Map
Patch
Adult
Egg
dot of 7 pixels
dot of 7 pixels
Patches
with alpha = 0 [kcal] 100 [kcal]
border lines of 2 pixels wide.
Backgro
Population of Egg
max = 1990
min = 0
0
1.0 [year]

Fig. 4. Screenshot of re:mobidyc's Observatory

Fig. 4 shows a screenshot of RE:MOBIDYC running the eggs and grasshoppers, a simple model to explain the basic features of RE:MOBIDYC. The space is divided into rectangular patches, visualized as green rectangles in the figure, each of which has grass on it. A grasshopper, rendered as yellow dots in the figure, moves to the cell with the richest grass and eats it. Eating grass reduces the grass in the patch. The grasshopper also stores energy from food and metabolism reduces the energy at a certain rate. If the energy is below a threshold, the grasshopper starves to death. The grasshopper spawns eggs when it gets matured. An egg, rendered as cyan dots in the figure, does not do anything until it eventually hatches after a certain duration, and becomes a grasshopper. A chart of the population of eggs by time is displayed to the right. Details of these language constructs are explained later.

2.2 Agents and Attributes

An agent has a set of attributes, each of which consists of an identifier, a measurement unit, and an optional initial value. The measurement unit is used for type checking to be described in Section 2.5 and also to display numbers in visualization such as tables and charts. An attribute is a variable that stores a floating point number. All numbers in attributes and expressions of RE:MOBIDYC are in the SI unit. Stages implicitly have the x and y attributes in m to hold the position of the agent. World represents the global environment and Patch represents a local environment of a square region of the space.

```
Adult is Grasshopper with
    age [day].
Egg is Grasshopper with
    age [day] = 0 [day].
```

Fig. 5. Example definitions of two stages

Fig. 5 shows the concrete syntax of the agent definitions in the eggs and grasshoppers model. The first and the second lines define the Adult stage of the Grasshopper species with the age attribute in day. The third and the fourth lines define the Egg stage of the Grasshopper species also with the age attribute in day initialized with 0 [day].

2.3 Actions and Tasks

Actions are core constructs of a model that represent interactions among agents as modifications to their attribute variables. RE:MOBIDYC employs discrete-event simulation with *synchronous* updates on memory, which holds all modifications to attributes during the computation at a time step.

To implement the synchronous updating memory, RE:MOBIDYC allocates three memory slots for an attribute: the *value* slot, the *next* slot, and the *delta* slot.

- The value slot is for the read access.
- The next slot and the next-delta slot are for the write access.

The interpreter can overwrite the next slot with a new value of the attribute for the next time step. The value will not be read until the next time step. The delta slot accumulates the increments/decrements to the attribute. The increments/decrements to an attribute across different actions will not affect the attribute until the next time step but are accumulated in the delta slot. At the end of the time step, the value slot is updated with the sum of the next slot and the delta slot. The details of the synchronous updating memory are explained in Section 2.6.

The synchronous updating memory was chosen due to DP5: *The system should record all attribute variables of all agents at all time steps*. The record of attribute variables at every time step carries all information about the simulation because the values of the attribute variables can change only at the transitions between time steps. Also, the synchronous updating semantics makes the semantics of actions closer to definitions of numeric sequence in the form of recurrence relation because neither the execution order of actions nor the order of agents affects the result of simulation. All changes are accumulated and reflected regardless of the order of execution.

```
to age is
    my Δ age′ = Δ time.
to move is
    my d/dt x′ = cos(theta)*r
    my d/dt y′ = sin(theta)*r
where
    theta = the heading
    r = the speed.
```

Fig. 6. An example definition of actions

Fig. 6 shows the definitions of three actions, namely age, move and hatch to illustrate the taste of the RE:MOBIDYC modeling language. The age action has one attribute definition to add Δtime, the simulation time step defined as a simulation setting, to the age attribute of the agent that performs the action. Please note that which agent will perform this action is not specified yet. An agent and an action are bound by a task definition, which is described later in this section.

The move action modifies the attributes x and y of the performing agent using two utility variables theta and r defined below the where clause. A utility variable is a kind of temporary variable scoped within the action. the heading and the speed are placeholders which will be replaced with concrete expressions in a task definition. Placeholder allows an action to have a generic parameter whose concrete argument may vary by the performer agent. Above the where clause, two attribute definitions on the x and y attributes are placed. An attribute definition with the d/dt decorator is equivalent to the attribute definition with Δ operator and the right-hand side multiplied by the simulation time step Δtime. For example, my d/dt x′ = cos(theta)*r is equivalent to my Δ x′ = cos(theta)*r*Δtime.

```
Adult move
where
    the speed -> uniform 0 [km/day] to 0.5 [km/day]
    the heading -> direction neighbor's grass.
```

Fig. 7. An example definition of a task

As stated in DP1: *The language should be declarative and should not impose imperative programming*, the action definition is in a declarative style. Assignments are listed at the top level of an action definition, and the right-hand sides are expressions which do not contain assignments. The only kind of expression that causes side effects is a random number generator. Expressions such as arithmetics and built-in functions trivially correspond to their counterparts in mathematics. All values are floating point numbers. Neither booleans nor symbols are expressions. No loop can appear in expressions including finding the local patch or nearby agents. Finding a peer for interaction is done by the interpreter.

2.4 Formal Semantics of the Language

Although the language is intended to be easy to learn without prerequisite programming skills, its semantics should be defined without ambiguity as a tool for scientific research. For example, the internal state of the random number generator depends on the number of random numbers generated so far even when the same seed is specified. A precise process of execution must be specified to clarify, for example, when the right-hand side of a utility definition is evaluated and how many times.

```
types
  AttributeDefinition ::
    variable : AttributeVariable | Placeholder
    decorator : Decorator
    expression : Expression;
  AttributeVariable ::
     agent : [Identifier]
     identifier : Identifier;
  Decorator = <assign> | <delta> | <differential>;
  Expression =
    Variable | Literal | Casting | Apply | Arithmetics | ...;
```

Fig. 8. AST definition of actions in VDM-SL

Syntactical definitions. Fig. 8 shows the definition of attribute definition's AST, and Fig. 9 defines how variable references in expressions are evaluated. Triple dots in the VDM-SL specification indicate omission. The main objectives of defining the formal semantics of RE:MOBIDYC are not in mathematical proof of certain properties of the language, but to provide a clear reference of the language for understanding models and for ensuring compatibility of ported interpreters in future.

Execution semantics. The definitions of major operations related to evaluating expressions and attribute definitions are shown in Fig. 9. RE:MOBIDYC provides expressions such as variable references, literals, built-in functions, arithmetics and so on. The definition body of the evalExpression operation is a huge cases statement that defines how to evaluate each kind of expression. The semantics of variables with regard to read and write access is a core feature of RE:MOBIDYC.

The reference to a utility variable is defined in three steps; (1) try read first and return if successful, (2) evaluate the right-hand side of the utility definition, and (3) store the result and return it. Thus, utility variables are evaluated on demand and written

```
evalExpression : AST`Expression ==> real
evalExpression(expr) ==
  cases expr:
    mk_AST`UtilityVariable(identifier) ->
      let val = Interpreter`readUtility(identifier)
      in
        (if val <> nil then return val;
        let newval = evalExpression(
          Interpreter`getUtilityDefinition(identifier))
        in
          (Interpreter`writeUtility(identifier, newval);
          return newval)),
    mk_AST`AttributeVariable(agent, identifier) ->
      return Memory`read(
        Interpreter`getAttributeAddress(agent, identifier)),
    ...
  end;

readVariable : [AST`Identifier] * AST`Identifier ==> real
readVariable(agent, identifier) ==
  return Memory`read(
    Interpreter`getAttributeAddress(agent, identifier));

evalAttributeDefinition : AST`AttributeDefinition ==> ()
evalAttributeDefinition(attributeDefinition) ==
  let
    value = evalExpression(attributeDefinition.expression),
    agent = attributeDefinition.variable.agent,
    identifier = attributeDefinition.variable.identifier
  in
    cases attributeDefinition.decorator:
      <assign> -> writeVariable(agent, identifier, value),
      <delta> -> writeDeltaVariable(agent, identifier, value),
      <differential> ->
        writeDeltaVariable(agent,identifier,value*deltaTime())
    end;

writeVariable:[AST`Identifier]*AST`Identifier*real==>()
writeVariable(agent, identifier, data) ==
  Memory`write(
    Interpreter`getAttributeAddress(agent, identifier),
    data);

writeDeltaVariable:[AST`Identifier]*AST`Identifier*real==>()
writeDeltaVariable(agent, identifier, data) ==
  Memory`writeDelta(
    Interpreter`getAttributeAddress(agent, identifier),
    data);
```

Fig. 9. Semantics of variable references in VDM-SL

once for each evaluation of the action. The reference to an attribute variable defined by the readVariable operation states that the interpreter is also responsible for resolving the address of the attribute variable of an agent. It is also clear that the values of the utility variable are managed by the Interpreter module while the attribute variables are allocated in the Memory module.

The definitions of the writeVariable and writeDeltaVariable operations define the write access to the next slot and the delta slot accordingly, and the interpreter resolves the address of the specified attribute of the agent. The definition of the evalAttributeDefinition operation specifies the mapping between the decorators and memory's write-access API.

2.5 Type System

The modeling language of RE:MOBIDYC has a unique type system based on measurement units. All attribute variables, utility variables, and expressions in RE:MOBIDYC are floating point numbers with measurement units. All values are computed in SI units, and a pair of types are compatible when their measurement units have the same dimension. For example, an expression 10 [km] / 3 [h] is typed [km/h] with dimension [m/s]. 10 [km] + 3 [h] is a type error because the $+$ operator requires the both arguments in compatible types and $[km]$ and $[h]$ are not compatible. Two kinds of type casting expressions are provided; the en-unit conversion to attach a measurement unit to a non-dimensional number typed [], and the de-unit conversion to detach the measurement unit to generate a non-dimensional number. These type castings are useful when dealing with expressions with exponentials and logarithms.

```
types
  Unit :: dimension:seq of (SIBaseUnit * int) scale : real
    inv mk_Unit(us, -) ==
      card {u | mk_(u, -) in seq us} = len us
      and (forall mk_(-, o) in set elems us & o <> 0);
  SIBaseUnit = <kg>| <m>| <s>| <degreeC>| <K>| <degreeF>| <rad>|
               <mol>;
```

Fig. 10. The definition of measurement units in VDM-SL

Fig. 10 shows a snippet from the definition of measurement units in VDM-SL. The $Unit$ type has two fields: the dimension field which has a sequence of SI base units and their orders, and the scale field for unit conversions in the type casting expressions and literal values with non-SI units. For example, the unit of speed $[km/h]$ is represented as $mk\_Unit([mk_(< m >, 1), mk_(< s >, -1)], 0.27777...)$.

```
types
     Address = nat1;

state Memory of
    vals : map Address to real
    next : map Address to real
    delta : map Address to real
    ...
    valuesStorage : seq of (map Address to real)
    ticks : nat
init s == ...
end

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

  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;

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

  load : nat1 ==> ()
  load(t) ==
    (vals := valuesStorage(t);
    ...
    next := vals;
    delta := {a |-> 0 | a in set dom vals};
    ...
    ticks := t)
  pre  t <= len valuesStorage and t <= len animatsStorage;
```

Fig. 11. The definition of read and write access to the memory in VDM-SL

2.6 Memory

As explained in Section 2.3, RE:MOBIDYC employs synchronous updates on memory. All modifications to attribute variables are delayed until the end of the time step. The values of all attributes are recorded to storage as required by DP5: *the system should record all attribute variables of all agents at all time steps*.

Fig. 11 shows the specification of memory access in VDM-SL. The memory holds three mappings from $Address$ to $real$ as the memory's internal state. The state variables $vals$, $next$ and $delta$ hold the value slots of values, next and delta accordingly. The three operations $read$, $write$ and $writeDelta$ define the three kinds of access to the memory. Because operations are not first-class objects in VDM-SL, callers to the $write$ and $writeDelta$ operations can be statically analyzed to ensure the evaluation of expressions does not reach the $write$ and $writeDelta$ operations.

The values of all attributes are stored into storage. The store operation appends the sum of the next slot and the delta slot into the state variable $valuesStorage$ typed as a sequence of real numbers. The load operation initializes the values slots and the next slots with the stored values, and sets the delta slots to zero. The store and load operations are called at transitions of simulation time steps to synchronize the three slots and the storage. These two operations abstract the implementation of the storage $valuesStorage$. The $load$ operation takes the time step as the argument, which allows the simulation not only to precede forward but can be re-winded to past and replay simulation. In Pharo, the Observatory UI shown in Fig. 4 has a slider to control the simulation time step back and forth.

The formal specification of the memory brings the significant benefit of enabling multiple storage back-ends. In Pharo, two kinds of storage are implemented. One is on-memory storage which simply stores the values as an ordered collection object which runs fast but has the limitation of capacity. Small models with short simulation time run efficiently on the on-memory storage. Another is file-based storage which dumps the values into CSV files. The file-based storage uses less memory regardless of the number of time steps in the simulation, at the cost of the reading and writing files. It is also possible to implement a storage back-end on RDBMS shared by multiple users. In Pharo, an abstract class for storage defines the public API of storage classes. The formal specification in VDM-SL adds the semantic requirements of the API so that the user can safely choose any concrete storage.

3 Development Process

The development of RE:MOBIDYC is hosted on the github organization [5]. Pharo source code and VDM-SL specification are together in the same repository.

Table 1 shows the progress of the development in the sizes of implementation and specification. The *Pharo LOC (all)* column shows the total number of lines of source code stored in the repository, including GUIs, parsers and type checkers. The *Pharo LOC (interp.)* column shows the number of lines of code for interpretation, *i.e.* AST,

[5] `https://github.com/ReMobidyc/`

Table 1. Source size during development time

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

evaluation and memory model. The *VDM LOC* column shows the number of lines of the VDM specification for the corresponding parts.

By reading the time factors, the specification went far quicker than the implementation. Implementation in Pharo started in Oct 2019 and is still ongoing. The specification in VDM-SL started in Aug 2022, about 3 years after the start of the implementation by the same engineer who implemented it in Pharo. The specification was developed on ViennaTalk [7] and was also refined on the Overture VSCode plugin [**?**,10]. The implementation for interpretation gradually increased throughout the development time and the specification caught up in four months. Please note that this should not be taken as the difference in productivity between Pharo and VDM. The difference reflects the cost of exploration to discover the language features validated by the domain experts. Building GUIs, designing concrete syntax, writing parsers, implementing various visualizations, creating type-checking algorithms, and communicating with domain experts are involved in the exploratory process.

The LOCs also indicate that the code in Pharo is about 10 times larger than its counterpart in VDM-SL. This also does not mean that the VDM is 10 times more productive than Pharo. Both Pharo and VDM provide high-level functions to manipulate abstract concepts such as finite sets and mappings. One source of the difference is the redundancy of Pharo's file format. Editing source files is not a typical way of programming in Pharo. Pharo provides powerful programming UIs that the programmer can interactively write code using automated tools. Accessor methods to read and write instance variables can be automatically generated by the tool, which takes a significant amount of lines when stored in a file.

4 Discussions

Successful cases of formal methods including VDM have shown that formal specification benefits the implementation phase in time, cost and quality [4]. The development of RE:MOBIDYC went in the reverse order of the specification and its implementation. The specification of RE:MOBIDYC plays two roles in the development: an intermediate product as input to the implementation, and the specification itself is a part of the final product. The language is designed as a tool for scientific research and the open specification of the interpreter and related components provides the rigorous semantics of the language.

Writing the specification from implementation went smoothly despite the difference in paradigms between Pharo and VDM-SL. Pharo is a dynamically-typed object-oriented language while VDM-SL is modularized definitions of types, values, functions, states, and operations without classes. There were minor mismatches in the way of the presentation of concepts between Pharo and VDM-SL. One is *kind-of* relations presented in a class hierarchy in Pharo. In VDM, they are presented as subtypes using union types. For example, an agent is either the world, a patch or a stage of individual life in RE:MOBIDYC. In Pharo, the classes for world definition, patch definition and stage definition are subclasses of the agent definition class. In VDM-SL, the AgentDefinition type is the union of WorldDefinition, PatchDefinition and AnimatDefinition. These subtype relations implemented in the class hierarchy are naturally rewritten as union types in VDM-SL.

An apparent difference in the presentation of implementation and specification is in how functions on AST nodes are defined. Although both Pharo and VDM-SL provide high-level functionalities such as sets and mappings, the specification in VDM-SL tends to be more compact than the source code in Pharo. The implementation class for agent definition has 24 methods and its subclass for stage definition has 15 methods while the AST module in the specification has only one function related to the agent definition, namely sizeOfAgent. Fields of a record value can be accessed by default as declared in the record type definition. In object-oriented programming, those accessor methods are considered good practice for specialization by inheritance, and Pharo's programming tools provide good UIs to operate over overridden small methods with less burden.

Besides accessor methods to access instance variables, Pharo's flexible object system allows the programmer to define control structures as methods. The attributeDeclarationsDo: method of the agent definition class iterates over the attribute declarations to evaluate the closure given as the argument. In the VDM specification, those enumerations on sequences are done by the language, *i.e.* the sequence-binds and for-in-do statements. A few functions like the sizeOfAgent function are needed to avoid repetitions.

The cost of the specification phase has been a small fraction in the development of RE:MOBIDYC. Case studies of lightweight formal methods indicate that the formal specification gains the quality and productivity of the implementation. The cost of specification in the RE:MOBIDYC was insignificant when the explorative tasks were done in the preceding implementation phase.

5 Concluding Remarks

This paper reported the development of RE:MOBIDYC as a case of the implementation-first approach of formal specification. The development time and amount of source text were significantly smaller than those of implementation. In successful cases of formal specification by conventional specification-first approach, specification took longer time than implementation. It is highly probable that the source of the cost of the specification case is more in the exploratory process than in presenting in formal specification languages. The exploration to understand the problem domain and to communicate with domain experts is required in the development of novel software. The developers need to pay the cost of exploration regardless of which language to use in the exploration, a specification language or a programming language. Formal specification is applicable in either case. The authors consider that the case of RE:MOBIDYC introduced in this paper mentions that the implementation-first approach with an agile programming language can be a good choice of lightweight formal methods. Further study is needed to understand the effect of formal specification applied in the implementation-first approach.

Acknowledgments

The authors thank the Pharo community for technically supporting the implementation platform.

References

1. Agerholm, S., Larsen, P.G.: A Lightweight Approach to Formal Methods. In: Proceedings of the International Workshop on Current Trends in Applied Formal Methods. Springer-Verlag, Boppard, Germany (October 1998)
2. Bergel, A., Cassou, D., Ducasse, S., Laval, J.: Deep Into Pharo. Square Bracket Associates (2013), `http://books.pharo.org`
3. Black, A.P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), `http://books.pharo.org`
4. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays. pp. 237–254. Springer, Lecture Notes in Computer Science, Volume 4700 (September 2007), iSBN 978-3-540-75220-2
5. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2), 3–11 (February 2008)
6. Larsen, P.G., Lausdahl, K., Battle, N., hn Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Peter W. V. Tran-Jørgensen, T.O., Chisholm, P.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
7. Oda, T., Araki, K., Larsen, P.G.: A formal modeling tool for exploratory modeling in software development. IEICE Transactions on Information and Systems 100(6), 1210–1217 (June 2017), `https://www.jstage.jst.go.jp/article/transinf/E100.D/6/E100.D_2016FOP0003/_article`
8. Oda, T., Araki, K., Larsen, P.G.: ViennaVM: a Virtual Machine for VDM-SL development. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 39–56. Newcastle University, School of Computing, Oxford (July 2018), TR-1524
9. Oda, T., Dur, G., Ducasse, S., Souissi, S.: re: Mobidyc-reconstructing modeling based on individual for the dynamics of community. In: International Conference on Practical Applications of Agents and Multi-Agent Systems. pp. 367–371. Springer (2021)
10. Rask, J.K., Madsen, F.P., a nd Leo Freitas, N.B., Macedo, H.D., Larsen, P.G.: Advanced vdm support in visual studio code. In: Macedo, H.D., Pierce, K. (eds.) Proceedings of the 20th International Overture Workshop. pp. 35–50. Overture (7 2022)