Transforming an industrial case study from VDM++ to VDM-SL

René S. Nilsson[1,2], Kenneth Lausdahl[3], Hugo D. Macedo[1], and Peter G. Larsen[1]

[1] Department of Engineering, Aarhus University, 8200 Aarhus N, Denmark
[2] AGCO A/S, Dronningborg Allé 2, 8930 Randers NØ, Denmark
[3] Mjølner Informatics A/S, Finlandsgade 10, 8200 Aarhus N, Denmark

Abstract. Normally transitions between different VDM dialects go from VDM-SL towards VDM++ or VDM-RT. In this paper we would like to demonstrate that it actually can make sense to move in the opposite direction. We present a case study where a requirement change late in the project deemed the need for distribution and concurrency aspects unnecessary. Consequently, the developed VDM-RT model was transformed to VDM++ and later to VDM-SL. The advantage of this transformation is to reduce complexity and prepare the model for a combined commercial and research setting.

Keywords: VDM, industrial application, model transformations

1 Introduction

The Vienna Development Method (VDM) is one of the most mature formal methods [2]. The method have been extended with multiple dialects over time, including the ISO standardised VDM Specification Language (VDM-SL) [3], VDM for object-oriented modelling (VDM++) [4] and VDM Real Time (VDM-RT) [12]. The choice of dialect highly depends upon the type of system or behaviour that must be modelled.

In this paper we present an industrial project involving optimization of the logistics in harvest operations. Such an operation is inherently distributed, as it involves a number of independent vehicles that need to coordinate and interact. Development guidelines for distributed real-time systems [7] were initially followed, resulting in a rather complex VDM-RT model [1].

A significant requirement change was introduced late in the project. Concretely, a specific communication protocol between vehicles as well as a specific hardware platform on each vehicle were imposed. Consequently the system architecture was changed to comprise a single centralised control algorithm and thin data-acquisition applications on each vehicle. The change of architecture diminished the need for distribution in the model, as the core functionality now only consisted of a single control algorithm and not a distributed control. In order to keep the model consistent with the modelled system and to reduce model complexity, distribution was removed from the model and thereby transformed to VDM++.

In this paper we will focus on a further transformation to VDM-SL, which was conducted to reduce model complexity and prepare the model for a combined commercial and research setting.

The evolution of the project is further described in Section 2. Afterwards Section 3 continues with concrete transformation examples and guidelines on transforming a VDM++ model to VDM-SL. Next, Section 4 presents an overview of the structural changes between the VDM++ and the VDM-SL models from the case study, while Section 5 provides an evaluation on the results obtained herein. Finally, Section 6 concludes on the findings of this work.

2 Case Study Project Evolution

The industrial case study presented in this paper originates from a research project named *Off-line and on-line logistics planning of harvesting processes*, involving Aarhus University and AGCO A/S.[4] Logistics optimisation in this setting includes both static and dynamic route planning of all involved vehicles. Specifically two tools were developed, 1) an *off-line simulation tool*, where a harvest operation can be optimized and simulated under the assumptions that no deviations occur, and 2) a *real-time guidance system*, that provides guidance to all drivers in the different vehicles involved and continuously monitors and reacts to possible deviations.

A common workflow in a harvest starts with a combine harvester harvesting the crop. The collected yield is unloaded into an in-field grain cart, which again unloads to an on-road truck, which delivers the yield to a drying or storage facility. This is illustrated in Figure 1, where parts of the optimized routes for each vehicle are shown.

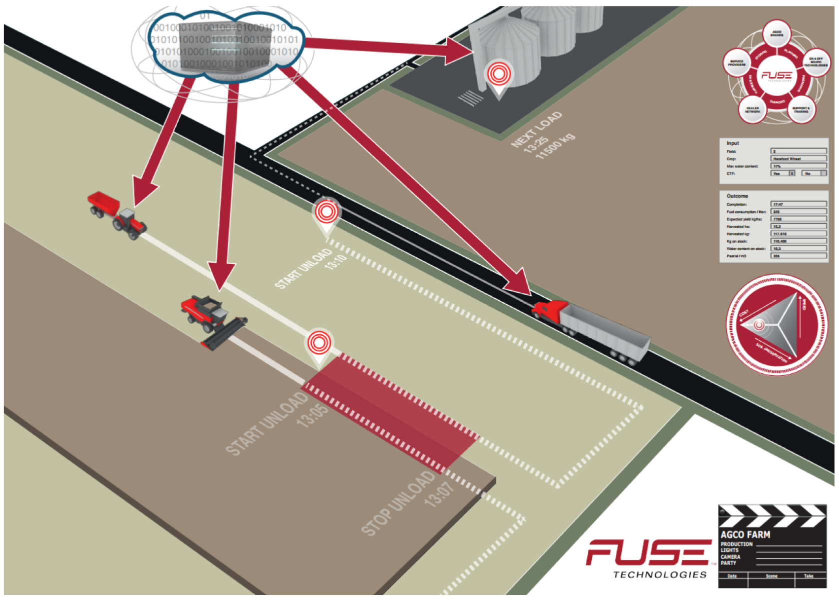

Fig. 1: Harvest logistics illustration.

[4] https://goo.gl/6tT8tK

During the four year span of the research project, the underlying VDM model has evolved and changed dialect a number of times, as depicted in Figure 2. The arrows and numbering illustrate how the model evolved over time. The arrows *2* and *4* highlights that substantial modifications were made to the VDM++ and VDM-RT model during the research project. These modifications are one of the main reason why the final VDM-SL model is different from the initial SL model. Note that the initial SL model was not kept up to date with the changes introduced into the VDM++ nor the VDM-RT model. Additionally, change in personnel had the side effect that knowledge of the initial SL model was lost. The following subsections further describe the motivation and reasoning behind each change of dialect.

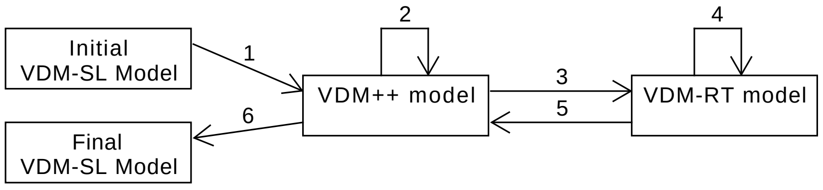

Fig. 2: Model evolution over time

2.1 Initial VDM modelling

The initial requirements for the project were defined based on the domain knowledge about the problem at hand. The system was considered a complex distributed system with embedded devices deployed in each vehicle. Therefore the development guideline for distributed real-time systems proposed by [7] was mostly followed. It involves a step-wise transition starting with a VDM-SL model, which is transformed to VDM++ and finally to VDM-RT, where more details of the system is included in each transition. The initial VDM-SL model captures the specification or the core functionality of the system. The transition to VDM++ adds concurrency and object-orientation, and the final transition to VDM-RT adds real time and deployment aspects.

For performance reasons, the Java bridge technology, offered by the Overture Tool, was leveraged [9]. This allowed the VDM model to invoke external Java components, such as existing Java graph libraries and proprietary performance optimized Java code [10, section 3.4]. Code-generation of the VDM model to Java further improved the performance [6], while easing the deployment process. Figure 3 shows how the VDM model and tests were connected to external components through a `Bridge` and how the same external components were integrated with the code-generated system through a corresponding `Delegate` class.

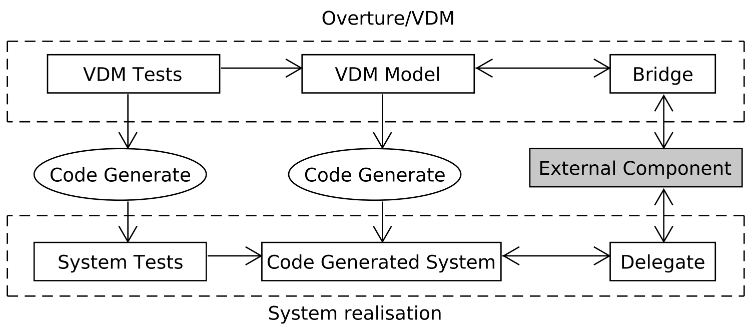

Fig. 3: Code generation and external components integration from [1].

2.2 Requirement changes

Late in the project a major requirement change was introduced. Initially the communication between vehicles had not been restricted in any way, and the hardware platforms were not constrained either. This allowed for an easy implementation of a distributed control algorithm. The requirement change meant that the system should work with existing hardware platforms and backend systems. All communication should now use a Publish/Subscribe (P/S) service, and the possibility to deploy software to the vehicles were highly constrained, as it should fit within an existing platform. As a consequence, it was decided to implement a centralized control algorithm in the cloud, and each vehicle should now only be responsible for real-time data acquisition and providing a user interface to the driver. This change towards centralized control diminished the purpose of including distribution in the model. Therefore, distribution was removed and replaced with a P/S interface and thereby transforming the VDM-RT model to a VDM++ model again.

2.3 Generalisation and commercialisation

At the end of the research project the model architecture was revised with the purpose of preparing the model for commercialisation. Additionally, a generalisation of the system towards other farming operations was envisioned. During the revision, focus was on the core functionality. One conclusion was that communication should be separated from the core model, hence removing the need for concurrency. This is enabled by the use of the Java bridge as described above.

Ideally the core optimisation and control algorithm could be achieved in a purely functional manner, taking the current state and incoming P/S events as input, and outputting a new state, including route plans for all vehicles.

In order to reduce complexity and best model the system, VDM-SL was chosen as the most appropriate dialect. From a research standpoint, the use of VDM-SL and a functional style should also allow new students to easily work on delimited parts of the model. Additionally it should enable formal proofs of certain model properties, which is not easily done in a VDM++ model. This led to the final transformation from VDM++ to VDM-SL, which is further described in the following sections.

3 Transformation guidelines

Transformation of a VDM-RT model to VDM++ is relatively simple since both dialects are object-oriented and we had limited use of VDM-RT specific constructs. However, transformation from VDM++ to VDM-SL is not straight forward, because the two languages does not share the same feature set, nor semantics [8]. Some of the main transformation challenges include concurrency, objects, hierarchy, operation overloading, and visibility.

In an attempt to create a systematic approach to transforming VDM++ models to VDM-SL a number of guidelines and concrete transformation rules are defined as described below.

3.1 Guideline 1: Concurrency

The VDM-SL dialect is a single threaded model and thus does not have any support for multiple threads nor coordination thereof. Therefore models cannot in general be converted into the VDM-SL dialect unless the nature of the problem is such that the multi-threaded behaviour can be moved out into an external Java component. This is in particular the case if the multi-threaded behaviour is present in order to facilitate communication where the data stream instead can be converted into a sequence of events, which can be consumed by the VDM-SL model sequentially.

3.2 Guideline 2: Visibility

In VDM++ the visibility of operations is declared using the access modifiers **`public`**, **`private`**, **`protected`** and **`static`**. In VDM-SL all definitions are static and the visibility is declared using the **`export`** and **`import`** constructs. Hence, if an operation should be visible in another module, the declaring module should export the operation including any internal referenced types, and the other module should import it.

State in VDM-SL is only visible within the module it is declared in. If the visibility needs to be extended, getters and setters can be implemented, which follows best OO practices.

3.3 Guideline 3: Operation overloading

Operation overloading is not supported in VDM-SL. All operations must be unique based on their name and any calls that relied on the overload behaviour must be guarded by an if statement to determine which operation to call.

3.4 Guideline 4: Objects and state

All object instances must be transformed into **`state`** components. This is described in the following transformation rule, and the accompanying example in Figure 3.

```
class A

types
Data : seq of char;

instance variables
data : Data;

operations
opX : () ==> ()
opX () == data := "ok";
```

```
class B

instance variables
objA : A;
data : real;

operations
opY : () ==> ()
opY () == (
  objA.opX();
);
```

(a) VDM++ classes

```
module A

types
ID = token;
S = seq of char;
InstanceMap = map ID to S;

state AST of
  a_m : InstanceMap
  init s ==  s = mk_AST({|->})
end

operations
opX : ID ==> ()
opX (id) == a_m(id) := "ok";



...
```

```
module B

types
ID = token;
S ::
  objA : [A`ID]
  data : real;
InstanceMap = map ID to S;

state BST of
  b_m : InstanceMap
  init s ==  s = mk_BST({|->})
end

operations
opY : ID ==> ()
opY (id) == (
  A`opX(b_m(id).objA);
);
```

(b) VDM-SL modules

Fig. 4: Transformation example: VDM++ classes translated to VDM-SL using guideline 4: *Objects and state.*

Transformation rule 1: Object instances map to state A class is transformed into a module that exports all functions and operations according to guideline 2. The class instance variables are translated into a module **`state`**. This is done by first defining a type that encapsulates all class instance variables e.g. `S`, where all objects are represented with an object id rather than an object reference. Secondly, a type `ID` for the object instances of the class itself is defined. The module state shall then comprise a mapping from instance id to state: **`map`** `ID` **`to`** `S`, where `S` could be a record. Finally, define an operation `newId : () ==> ID`, which creates a unique id and a new instance record of type `S` and add it to the instance map, while returning the id. In this way, duplication of `ID`s is avoided.

Once the transformation is completed, all modules now reference state "objects" by an `ID`, rather than having a direct object reference, which is the case in an OO setting. Rather than invoking operations directly on the object, modules now need to invoke operations on the "objects" module, passing the `ID` along.

3.5 Guideline 5: Inheritance

Inheritance in a VDM++ setting includes a number of features, such as inheriting instance variables and operations, overriding operations, and extending a class with new instance variables and operations. All of which are features that are not directly supported in VDM-SL. Possibly, a complex transformation might be able to support all inheritance features, but the resulting VDM-SL model will be very complex and not easily understood or maintained. Our general guideline is therefore to avoid constructs that mimic inheritance in VDM-SL if possible. However, if only a few features are used for a specific purpose, simpler transformations can be used, with reasonable results. Specifically, we present two transformations related to inheritance, which have been used in the case study.

Strategy design pattern: A strategy design pattern consist of two types of classes, a strategy interface class and concrete strategy classes [5]. In an OO setting, a strategy pattern will be used, by having an object reference defined by the interface class and invoking operations on the object. The strategy can be changed easily, by replacing the object reference with another object reference that implements the same interface. In VDM++ the strategy interface class would be implemented as a base class, and the concrete strategies would be subclasses hereof. This is not possible in VDM-SL, but a strategy design pattern can be transformed to VDM-SL using union types and cases expressions, as described in *Transformation rule 2* and the accompanying example in Figure 5.

Transformation rule 2: Strategy patterns map to a union type and cases expressions Each concrete strategy class is transformed to VDM-SL, following all necessary previously defined guidelines. The strategy interface class must define a union type `Type` with a type for each concrete strategy module. A parameter of type `Type` must be added to each operation defined in the interface class. Additionally, a **`cases`** expression on the type parameter must be added, with an entry for each concrete strategy type, which delegates the call to the concrete strategy module.

```
class X

operations
opX : nat ==> nat
opX (x) ==
  is subclass responsibility;
end X
```

```
class A is subclass of X

operations
opX : nat ==> nat
opX (x) == return x + 1;
...
end A
```

(a) VDM++ classes

```
module X

types
Type = <A> | <B>;

operations
opX : Type * nat ==> nat
opX (t, x) ==
  cases t:
    <A> -> return A`opX(x),
    <B> -> return B`opX(x),
    ...
    others -> exit "Unknown Type"
  end;
end X
```

```
module A

operations
opX : nat ==> nat
opX (x) == return x + 1;
end A



module B

operations
opX : nat ==> nat
opX (x) == return x * 2;
end B
```

(b) VDM-SL modules

Fig. 5: Transformation example: VDM++ classes translated to VDM-SL using guideline 5: *Inheritance and strategy pattern.*

Basic inheritance: *Transformation rule 3* along with the example in Figure 6 describes how some of the basic inheritance features can be transformed in a simple manner, if type information is available whenever operations are invoked on the instances. In an OO setting this information is known through the object reference, but in VDM-SL it must be handled explicitly. Embedding this information in the inheritance modules in VDM-SL greatly complicates the model, which is not desirable. Therefore, we suggest this reduced transformation, which can be used to extend a module with a relatively small effort. Specifically, this reduced transformation will mimic the following inheritance features: Extending a class with more instance variable and operations, operation overriding, and allow calls to a super class.

Transformation rule 3: Basic inheritance used for extendibility/reusability In each module define a type `S` that encapsulates all the instance variables of that class and its superclass. Additionally, in the base module, add a union type `S_UNION` that holds the `S` type from all the modules and add an instance map from a type `ID` to `S_UNION`, similar to Guideline 4. Note that all "object instances" of all the subclasses will be kept as state in the base module. Next, add getters and setters for state in the base module, such that all sub-modules can access the necessary state. Finally, add a `newId` operation as described in Guideline 4.

4 Structural changes to the VDM models

4.1 Existing VDM++ model

The VDM++ model supports both off-line simulation and real-time guidance of harvest operations, but the model presented here is simplified to ease the understanding and only includes the core functionality. As VDM++ is an OO language, the model contains common OO constructs, such as design patterns including the strategy and the template pattern [5]. This also means that many of the individual instances of classes have state information about themselves so references to these are frequently passed around. Figure 7 shows a simplified class diagram of the model, where `Harvi` is the top-level class of the control algorithm. The strategy pattern is used both for `UnloadStrategy` and `TrackSeqStrategy`, whereas the template pattern is used for the `Resource` class and its subclasses.

The core of the model is single-threaded, but the integration with the P/S framework introduces more threads and asynchronous callbacks. In the presence of concurrency this means that permission predicates on certain instance variables and operations are included in the model, and it seems that this can have a negative impact on the performance, since every time an operation is called the permission predicates must be analysed.

4.2 VDM-SL model

Given the transformations defined in Section 3, the VDM++ model was transformed to VDM-SL. In addition, the VDM-SL model was made more general in the sense that

```
class A

instance variables
x : nat;

operations
opX : () ==> nat
opX () == return x + 1;
end A
```

```
class B is subclass of A

instance variables
y : real;

operations
opY : () ==> real
opY () == return y + 1.5;
end B
```

(a) VDM++ classes

```
module A

types
ID = token;
S :: x : nat;
S_UNION = S | B`S;
InstanceMap = map ID to S_UNION;

state AST of
  a_m : InstanceMap
  init s ==  s = mk_AST({|->})
end

operations
opX : ID ==> nat
opX (id) ==
  return a_m(id).x + 1;

getState : ID ==> S_UNION
getState (id) ==
  return a_m(id);
end A
```

```
module B

types
S :: x : nat
     y : real;

operations

opY : A`ID ==> real
opY (id) ==
  return A`getState(id).y + 1.5;
end B




...
```

(b) VDM-SL modules

Fig. 6: Transformation example: VDM++ classes translated to VDM-SL using guideline 5: *Basic inheritance.*

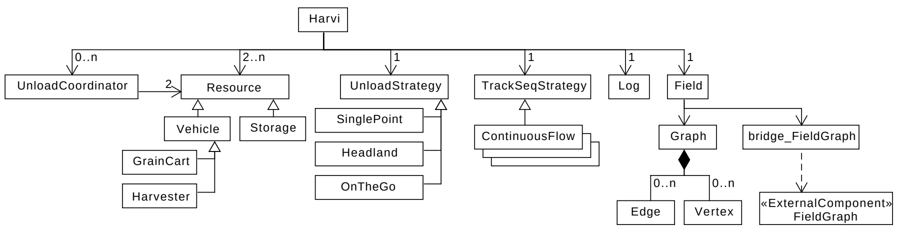

Fig. 7: Simplified class diagram of VDM++ model.

the VDM++ model was only able to cope with one plan, whereas the VDM-SL model has been prepared to be able co cope with multiple plans. A simplified overview of the most important modules in the VDM-SL model can be found at Figure 8. Note how the level of plans simply is added as a layer above the other modules. The `GrainHarvest` module is similar to the `Harvi` class from the VDM++ model. The four modules below that, all include state information organised as mappings from identifiers to data about them. In this way the state information is centered at specific places and the `MQTT` module represent all the P/S communication with the centralised cloud service. This is realised in Java using the bridge technology explained in Section 2. This has its own thread of control but since this is outside the actual VDM model the model complexity is significantly reduced.

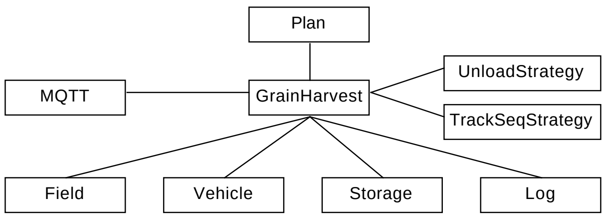

Fig. 8: Simplified overview of the VDM-SL modules.

5 Evaluation

Although most of the listings presented above indicate that the VDM-SL version is larger than the corresponsing VDM++ model, the transformation from VDM-RT to VDM-SL resulted in a new smaller model as shown in Table 1. However, the transformation process led to the discovery of cases where the responsibilities of modules were mixed. This discovery was made because of the explicit imports added in the process. The many dependencies between the modules is likely an artefact of the initial distributed system where each vehicle had more control over its own behaviour oppose to the current approach where a more "functional" approach is taken. The new model aims to provide an operation which can perform all required computation to consume events received from the vehicles and in turn produce new or updated routes. All communication have been removed from inside the planning operation and converted into a sequence of events that is then consumed as part of the plan generation. The ideal function would have looked like illustrated in Listing 1.1, but due to the caching of the large graphs, used to represent the field, it had to be modelled as an operation and thus keeping state in many modules.

	VDM++	VDM-SL
Lines Of Model	3701	3041

Table 1: Lines of model in VDM++ and VDM-SL implementations, excluding libraries and tests.

```
planRoutes :  FieldPartition *
              seq of Route *
              FieldProgress *
              seq of Event -> seq of Route
```

Listing 1.1: Ideal route planning function.

The model transformation was carried out manually partly using the transformation principles from Section 3 that in itself can be quite error prone due to human factors. To make this even worse the testing framework *VDMUnit* only provided support for OO based models and thus could not be used for the new SL model. To overcome this issue and provide some validation against the source model a new extension to *VDMUnit* was developed to mimic the unit test behaviour for SL models as described in [11]. The test validation did provide the required basis for comparison with the source model but also showed a limitation of a VDM module based approach where all modules have mutable state. The primary issue is that all operations are directly imported and since VDM does not have *operation values* there is no way to provide true module based testing of each module in isolation using stubs that respect the pre-, post-conditions of the imported operations.

To assess if the newly created VDM-SL model performs similarly to the VDM-RT model the execution time of the full test suite for both models are compared in Table 3 and for one of the test scenarios in Table 2[5]. It shows that some of the improvements done during the transformations likely had a positive impact on the performance. During the transformation multiple places in the model were identified that constructed

[5] All experiments were performed on a server hosting a 64 bit VM configured with 6 x Intel Core Processor @ 2.0 GHz and 15 GB RAM.

route sequences in a way that grew exponentially in time in relation to the route length. One example was looping over a route to check relevant sequence elements, by using **`hd`** and **`tl`** expressions, rather than an index. By using the **`tl`** expression, the route was internally cloned for every loop iteration, causing poor performance both in the interpreter and in the generated code.

The comparison in Table 2 clearly shows that the original model had scalability issues.[6] As a result it was not possible to determine the full execution time of the full test suite for the interpreted model as shown in Table 3, where the experiment was turned off after 7 days execution running at 100% CPU load.

	VDM++	VDM-SL	Difference
Interpreted	2246.65 s	358.16 s	-84%
Code generated	40.36 s	19.46 s	-52%

Table 2: Performance comparison between VDM++ and VDM-SL implementations for one big scenario test.

	VDM++	VDM-SL	Difference
Interpreted	> 7 days	150 min	> -98%
Code generated	91 min	7 min	-92%

Table 3: Performance comparison between VDM++ and VDM-SL implementations for all tests.

It should be noted that the generated code does not perform any pre-, post-condition, or invariant checks. This is one of the primary reasons to why the test framework was upgraded to support VDM-SL. The usage of the *VDMUnit* for the OO model have revealed especially pre-condition errors that were not revealed by the tests at the generated code level.

6 Concluding remarks

In the new VDM-SL model the main focus is on the calculation of routes for the different vehicles. Compared to the VDM++ model there is also a cleaner separation between the planning aspects and the event-based communication carried out via the Publish/-Subscribe server connection to the cloud. Actually it was a positive surprise for us that transforming the model from an imperative style to a more functional style had the side effect of increasing the performance. However, it also turned out that the transition

[6] The difference is calculated as: Difference $= \frac{a-b}{b} * 100\%$, where $a =$ VDM-SL performance and $b =$ VDM++ performance

rules suggested were not sufficient to take care of all refactorings. In particular in relation to inheritance there is a tendency that there is an overhead of what needs to be written in a VDM-SL setting. Therefore, we do not see a possibility for automating the transformation from a VDM++ model to VDM-SL.

Initial work targeting the use of this VDM-SL model in a combined commercial and research context have started and this looks promising, but no conclusion can be drawn from this yet. Finally, it is worth noting that it is possible to use traditional unit testing in a VDM-SL model with the use of a newly extended VDMUnit testing framework.

Future Plans Transitions from VDM++ to VDM-SL are not something one would embed in a methodology like the one proposed by [7], as it is not a desired transition. Ideally one would not end up in a situation where such a transformation is necessary. However, requirement changes are a common phenomena and might lead one into such a situation. We see the transformation guidelines proposed in this papers as a help or inspiration for others who might face the same needs as we did.

Theoretically it might be possible to include all features of VDM++ into similar guidelines, but it is our belief that this would be at the expense of model complexity and readability. Therefore we do not propose to go this way, and we do not suggest automating the transformation either, as especially the inheritance transformations might be somewhat use case specific.

Acknowledgments We would like to thank the Danish Innovation Foundation for funding this project and to our colleagues that have worked with us in this project. We would also like to pay special thanks to the anonymous reviewers who have helped improving the quality of the paper.

References

1. Couto, L.D., Tran-Jørgensen, P.W.V., Larsen, P.G.: Enabling continuous integration in a formal methods setting. In: Submitted for publication (2018)
2. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc.
3. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
4. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://overturetool.org/publications/books/vdoos/`
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
6. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)

7. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
8. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: The VDM-10 Language Manual. Tech. Rep. TR-2010-06, The Overture Open Source Initiative (April 2010)
9. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-30885-7_19`, ISBN 978-3-642-30884-0
10. Tran-Jørgensen, P.W.V.: Enhancing System Realisation in Formal Model Development. Ph.D. thesis, Aarhus University (Sep 2016)
11. Tran-Jørgensen, P.W.V., Nilsson, R.S., Lausdahl, K.: Enhancing Testing of VDM-SL models. In: Proceedings of the 16th Overture Workshop (July 2018)
12. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)