Specification and Animation of a Bank Transfer

Y. Ledru
Laboratoire de Génie Informatique
IMAG, B.P. 53
F-38041 Grenoble cedex, FRANCE

Abstract

The development of formal specifications may benefit from prototyping activities. The production of an executable model for a given description helps bridging the gap between this specification and the corresponding reality. The KIDS/VDM system, based on the KIDS environment, provides these prototyping facilities for the model-based specification language of VDM. This paper illustrates its use in the specification of a bank transfer operation. It shows how animation may be helpful at several stages of a specification process based on a series of refinements of an initial abstract specification.

1 Introduction

Recent reports [1, 2] have shown that model-based formal methods like Z [3] or VDM [4] are becoming part of industrial practice for critical applications, especially at the specification stage. The specification languages associated to these methods are mainly mathematical notations that allow the precise description of a given system.

A first generation of tools has been proposed to support these formalisms. It is based on the syntax and static semantics of these languages, and ensures the syntactical conformance of a given text to the definition of the formalism. But syntactical conformance to a mathematical notation does not guarantee the high quality of a specification, which is the expected benefit of these techniques. For example, the mathematical formula $x \geq 5 \wedge x < 1$ can not be verified by any x value. It corresponds thus to a specification that can not be fulfilled.

A formal method like VDM provides several techniques to check that a specification has interesting mathematical properties (invariant preservation and implementability). For example, it defines a necessary condition for the implementability of specifications. The application of these techniques allows to produce a consistent piece of mathematics but does not address an important topic in the validation of specifications: does it correspond to some real world problem?

One way to get further convinced of this correspondence is to produce an executable model of the specification and to confront it to the real world, e.g. by demonstrating it to a customer. This may be achieved by using an executable specification language, which is not the case for VDM and Z. The pros and cons of executable specifications have already been debated, among others in [5, 6]. As far as VDM is concerned, several attempts have been performed towards executable specifications [7, 8]. Two strategies may be followed:

- to concentrate on a subset of executable constructs of the language. This turns the specification language into a programming language. It may lead the specifier to overlook the non-executable constructs of the specification language and to specify unnecessary details for executability concerns.
- to informally translate the specification into a programming language close to the specification language. This distinction between languages prevents overspecification, but the link between specification and prototype is only informal and errors may result from the translation process. Moreover, the production of two descriptions, the specification and its prototype, increases the cost of the development.

This latter approach may be improved by an adequate tool support which
- ensures the correspondence between specification and prototype;
- automatizes parts of the development.

KIDS/VDM [9] provides such a solution for the production of prototypes from VDM specifications. It exploits two knowledge-based software engineering tools: the REFINE[1] language [10] and the KIDS system [11]. This paper will illustrate its use on a case study and show how these techniques may be helpful in the specification process.

Section 2 presents the case study, introduces the VDM method and proposes a first specification. Section 3 features two refinements of the initial specification and shows the benefit of this tool support. Finally, section 4 will draw the conclusions of this work.

2 Abstract specification of a bank transfer

At the 16th International Conference of Software Engineering, several participants of the Workshop on Software Engineering and Artificial Intelligence were asked by M. Feather to work on examples taken from the "Barclay's Code of Business Banking"[12]. This document explains the general principles of the operation of this bank. The bank transfer example presented here is based on common knowledge about business banking, combined with several elements from that document. Its objective is to illustrate how a VDM developer could use his tools for this kind of specification activity. Another response to this problem has been proposed by Chung and his colleagues [13]: it focuses on non-functional requirements and the support of change.

[1] REFINE is a trademark of REASONING SYSTEMS Inc.

2.1 Reuse of an existing specification

A good starting point in specification is to reuse an existing description of a bank transfer. The following VDM specification is taken from the teaching notes of [4].

Types and state variables A VDM specification describes an "abstract machine", i.e. global variables associated to operations. The definition of state variables starts with the introduction of types:

- $Acno$ (account number) is a simple natural number

$$Acno = \mathbb{N}$$

- $Acdata$ (account data) groups in a record the name of the owner of the account (own), the current balance (bal), i.e. the amount of money available, and the absolute value of the overdraft limit (od), i.e. the limit on the possible negative balance; an invariant on the type $Acdata$ constrains the balance to be greater than the overdraft.

$$\begin{aligned} Acdata :: \; own &: char^* \\ bal &: \mathbb{Z} \\ od &: \mathbb{N} \end{aligned}$$

$$\text{inv } (mk\text{-}Acdata(own, bal, od)) \triangleq bal \geq \text{-}od$$

Once types have been defined, the state variables are introduced. Here, the state of the specification is made up of a single global variable am (accounts map) that stores a map from account numbers to account data.

$$Bank :: am : Acno \xrightarrow{m} Acdata$$

Operations on the state In VDM, each operation is defined in terms of a pair of predicates that specify its pre- and post-conditions. In this case study, operations include: creation of a new account, deletion of an account, modification of account data such as the owner or the overdraft limit, and operations on the amounts stored on the accounts. In this paper, we will concentrate on the transfer operation ($TRANSF$).

- It takes three arguments as input: fr_ac, the account which will be debited, to_ac, the account which will be credited, and a, the amount of money transferred. This last parameter is a natural number strictly greater than zero.
- The operation has read and write access to the global variable am.
- A pre-condition states that both accounts given as input must be in the domain of the am map, that they must be different, and that the balance and overdraft associated to fr_ac must allow the transfer.
- The post-condition expresses the new values of the state variables after the execution of the operation. It states that am is overwritten by a new map which only modifies the balances of both accounts given as parameters.

$$TRANSF\ (fr\_ac{:}\, Acno, to\_ac{:}\, Acno, a{:}\, \mathbb{N}_1)$$

$$\text{ext wr } am : Acno \xrightarrow{m} Acdata$$

$$\begin{aligned} \text{pre } & fr\_ac \in \text{dom } am \wedge to\_ac \in \text{dom } am \\ & \wedge fr\_ac \neq to\_ac \\ & \wedge bal(am(fr\_ac))\text{-}a \geq \text{-}od(am(fr\_ac)) \end{aligned}$$

$$\begin{aligned} \text{post } am = \overleftarrow{am} \dagger \{ & fr\_ac \mapsto \mu(\overleftarrow{am}(fr\_ac), \\ & \quad bal \mapsto bal(\overleftarrow{am}(fr\_ac))\text{-}a), \\ & to\_ac \mapsto \mu(\overleftarrow{am}(to\_ac), \\ & \quad bal \mapsto bal(\overleftarrow{am}(to\_ac)) + a)\} \end{aligned}$$

Some VDM notations need further explanations:

- the "hook" symbol ($\leftharpoondown$), used in the post-condition, denotes the initial value of the state variable;
- The overriding operator (†) is a binary operator on maps which returns the union of the second map with the pairs of the first map which are outside its domain.
- The μ function takes two arguments: a record and an association of a record field and a value. It returns the value of its first argument where the specified field has been replaced by the given value.

In the proposed ISO VDM standard, the specification of the initial state is a component of the state specification. Here, we use a slightly different (but equivalent) approach where the initial state is specified as an operation. In this case study, the post-condition of $INIT$ specifies an initial state with three accounts. For example, account 1 is owned by John, has a positive balance of 25000 and an overdraft limit of 10000.

$$INIT$$

$$\text{ext wr } am : Acno \xrightarrow{m} Acdata$$

$$\text{pre } true$$

$$\begin{aligned} \text{post } am = \{ & 1 \mapsto mk\text{-}Acdata(\text{John}, 25000, 10000), \\ & 2 \mapsto mk\text{-}Acdata(\text{Paul}, 0, 0), \\ & 3 \mapsto mk\text{-}Acdata(\text{George}, 5000, 10000)\} \end{aligned}$$

Proof obligations Once a VDM specification has been stated, it must be proved to fulfill several mathematical properties. First, the invariants and types must be preserved by the operations. Second, the operation must be implementable, i.e. there must exist a final state that satisfies the post-condition provided the pre-condition was satisfied.

In this case, implementability is easy to demonstrate since the post-conditions provide the new values of the state variable. The main point is to check that the invariant associated to type $Acdata$ is preserved by the operation, i.e. that the balance remains greater than the overdraft limit. This is obvious for $INIT$, since all balances are positive. As far as $TRANSF$ is concerned, this is guaranteed by the last conjunct of the pre-condition.

The careful examination of these proof obligations may be stated more formally. Several books [4, 14] provide the necessary detailed framework to do so. It may even be supported by a theorem prover [15]. Nevertheless, in practice, this proof activity is most often performed very informally.

Further validation of the specification may be carried out by cross-reading or inspections. But a specification like the one of $TRANSF$ involves a mathematical jargon that may only be read (and understood) by a VDM expert. This process guarantees that the specification is a consistent piece of mathematics but does not assess that this mathematical model corresponds to the described reality.

Validation of the specification In order to achieve a high quality specification, this specification should be validated by a bank specialist, who will presumably be unable to read the mathematical text of the specification. A dialog must thus take place between him and the developer. One way to facilitate this dialog is to prototype the specification.

2.2 KIDS/VDM

The original idea of KIDS/VDM [9, 16] is to exploit the similarities between the specification language of VDM and the executable language REFINE [10]. Both languages share the same data types and data structures (sets, sequences, maps) and REFINE provides a wide spectrum of control constructs (functional, declarative, imperative). Experiments have shown that a VDM specification can be informally transformed into a REFINE program [17].

The Kestrel Institute Development System [11] supports the synthesis of REFINE programs from functional specifications, written in a language named REGROUP. This synthesis process corresponds to the application of design tactics (e.g. global search, divide and conquer), simplifications by a theorem prover, and code optimizations. Every synthesis step is performed under the control of the environment which guarantees the correctness of the code with respect to its specification.

KIDS/VDM exploits the capabilities of the KIDS environment to

- control and support the transformation of VDM specifications into REFINE code [9];
- help discharging proof obligations of the VDM specification [16].

This second feature will not be considered in this paper which focuses on the use of KIDS/VDM to animate specifications.

The original KIDS environment supports two major modes:

- the *program development* mode includes tools for the synthesis of programs from specifications;
- the *theory development* mode proposes tools to help the expression of REGROUP specifications and the development of inference rules for the theorem prover.

The KIDS/VDM environment [9] adds a new mode to KIDS to support transformations of VDM specifications (Fig. 1). A typical development of a VDM specification goes through three main steps:

1. transformations of the original VDM specification, in the VDM mode; these transformations are automatic and rewrite several language constructs (e.g. subtyping) into a form closer to REGROUP;
2. translation of the resulting VDM specification into a combination of REGROUP specifications and REFINE programs;
3. synthesis of REFINE programs from these specifications, in the program development mode.

The principles of this process have already been detailed [9]. It is essentially automatic: interactions mainly occur in the KIDS development mode in order to select the design tactics or guide the optimization and simplification activities. Recent work on this system has kept the underlying principles of the translation; it has increased the coverage of the VDM specification language, added design and proof

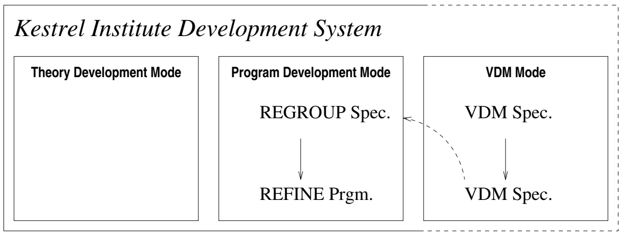

Figure 1: Prototyping a specification within KIDS

knowledge in order to better support the kind of developments involved, and improved the automatization of the process.

In this case study, the following REFINE program has been automatically synthesized for $TRANSF$. It is a side-effecting function that modifies the AM variable by assigning it the result of the TRANSF-F function.

```
function TRANSF(FR-AC: integer,
                TO-AC: integer,
                A: integer) : any-type
= let(TRANSF-TUPLE:
        tuple(AM: map(integer,
                      tuple(OWN: seq(char),
                            BAL: integer,
                            OD: integer)))
      = TRANSF-F(FR-AC, TO-AC, A, AM))
  AM <- TRANSF-TUPLE.1
```

TRANSF-F (see Fig. 2) is a pure function that takes the initial value of state variable AM as an argument and returns the new value of this variable. Its specification is automatically generated from the VDM specification. It is then transformed into the REFINE code of Fig. 2 by the application of the KIDS tactics "spec-to-code". This tactics extracts the new value of AM from the equality in the post-condition. It also checks that several typing constraints are verified by the result: INV-NATURAL-4 and INV-NATURAL-1-4 respectively check that the account numbers and overdraft are natural values, INV-BAL-OD checks that the typing invariant of $Acdata$ is verified. A further KIDS optimization step has introduced the local variables C-1, C-2, and C-3 in order to avoid computing C-1 four times.

The REFINE language does not provide a map overriding operator. To support the translation of VDM specifications, it has thus been defined as a function in a general purpose KIDS theory:

```
function OVERRIDES(m1,m2)
                  : map (any-type,any-type)
= {| x -> (if defined?(m2(x)) then m2(x)
                                 else m1(x))
  |(x) x in (domain(m1) union domain(m2))|}
```

This definition only provides a way to compute map overriding. In order to perform code simplifications, the theory should be complemented with axioms and theorems about this operator.

```
function TRANSF-F (FR-AC: integer, TO-AC: integer, A: integer,
                   OLD-AM: map(integer,tuple(OWN: seq(char), BAL: integer, OD: integer))
                   | ...)
returns
 (TRANSF-TUPLE: tuple(AM: map(integer, tuple(OWN: seq(char), BAL: integer, OD: integer)))
   | ... )
= let(C-3: tuple(OWN: seq(char), BAL: integer, OD: integer) = OLD-AM(TO-AC),
      C-2: tuple(OWN: seq(char), BAL: integer, OD: integer) = OLD-AM(FR-AC))
  let(C-1: map(any-type, any-type)
      = OVERRIDES(OLD-AM,
         {| FR-AC -> (let (var XXX: tuple(OWN: seq(char), BAL: integer, OD: integer) = C-2)
                      XXX.BAL <- C-2.2 - A; XXX),
            TO-AC -> (let (var XXX: tuple(OWN: seq(char), BAL: integer, OD: integer) = C-3)
                      XXX.BAL <- C-3.2 + A; XXX) |}))
  if INV-NATURAL-4(C-1) & INV-NATURAL-1-4(C-1) & INV-BAL-OD-2(C-1)
  then <C-1> else undefined
```

Figure 2: The `TRANSF-F` function

In summary, the synthesis of the prototype proceeds mainly automatically. The significant interactions with the environment are the choice of the spec-to-code tactics and the optimization of the resulting code. The resulting REFINE code has then been compiled into executable form.

The KIDS/VDM development may be carried on further in order to simplify the resulting program: in the context of the pre-condition, it may be proved that the guard of the `if ... then ... else` construct is always true and that the program may be simplified to `<C-1>`. Such a simplification not only speeds up the execution of the prototype, but also provides a proof of the implementability of the specification because `TRANSF` may no longer return `undefined`. Yet, performing this simplification is a proof process that may involve extensive user interaction.

2.3 Execution of the prototype

2.3.1 A typical execution

The execution of the prototype may give the following scenario:

- At the beginning, the am variable exists but its content is undefined.

```
.> am
RE:*UNDEFINED*
```

- The $INIT$ operation is performed and the content of am is checked. Actually, since no output is specified for the $INIT$ or $TRANSF$, they return the new value of the state variable.

```
.> (init)
((1 "John" 25000 10000)
 (2 "Paul" 0 0)
 (3 "George" 5000 10000))
.> am
((1 "John" 25000 10000)
 (2 "Paul" 0 0)
 (3 "George" 5000 10000))
```

- A transfer of an amount of 1000 is executed between accounts 1 and 2. It may be checked that it modifies the accounts of John and Paul, and does not affect other accounts.

```
.> (transf 1 2 1000)
((1 "John" 24000 10000)
 (2 "Paul" 1000 0)
 (3 "George" 5000 10000))
```

- An illegal transfer is then attempted: Paul has no permitted overdraft and tries to transfer more than his balance. First, the pre-condition, prototyped by a similar process, may be evaluated:

```
.> (pre-transf 2 1 10000 am)
NIL
```

  Then, an attempt to perform the transfer results in:

```
.> (transf 2 1 10000)
Error: Received signal number 11 (Seg-
mentation violation)
```

  Actually, the semantics of VDM operations mean that if an operation is started with a false pre-condition, anything may happen.

2.3.2 The VDM Animator

In order to facilitate this animation, the KIDS/VDM environment provides a graphical interface to the prototype (Fig. 3). The "VDM animator" visualizes the metaphor of the abstract machine. The abstract machine stores internal variables that are affected by its operations. The "State Variables" window displays the current contents of the internal state (here, variable AM). The operations of the abstract machine appear as mouse-sensitive words in the "Buttons" window. The user may click on these buttons to apply the operations to the machine. The contents of the

"State Variables" window are updated accordingly. Operations may also take additional input parameters and deliver some output values through the "Input" and "Output" windows.

The VDM animator is automatically instantiated from the text of the corresponding VDM specification. The buttons and variables are then linked to the REFINE variables and functions produced by the prototyping process.

In practice, the VDM animator provides an easy way to experiment the resulting prototype. The user has a comprehensive look at all variables of its specification, provided that these fit into a single window, and has a user-friendly access to its operations.

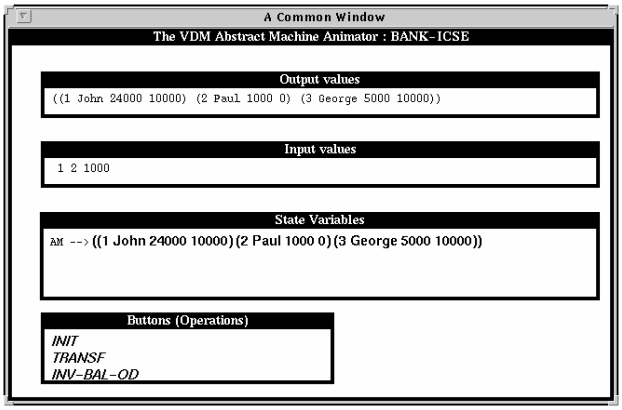

Figure 3: The VDM animator

3 Refinements of the specification

The VDM method allows to refine abstract specifications into more concrete ones. This refinement mechanism is usually considered as a way to link programs to specifications but it may also be used to further develop specifications. In this case study of bank transfer, both refinements of control structures and data structures are presented. First the debit and credit operations are introduced, then the data structure representing an account is further developed.

3.1 Refining the control flow

The specification of the bank transfer proceeds with the decomposition of $TRANSF$ into $DEBIT$ and $CREDIT$. This decomposition is intended to model the following statement of the bank code:

> *When you pay a cheque into your bank account at the branch where you have your account, the amount of the cheque will be shown on your statement as being credited to your account on that day. So the value of the cheque appears to be available for you to use immediately.*

Therefore, the transfer may be specified as a sequential combination of operations:

$$TRANSF(fr, to, a) \triangleq \\ CREDIT(fr, to, a); DEBIT(fr, to, a)$$

In order to make the amount of the cheque directly available, the credit operation is performed first; the debit takes place when the cheque has been processed completely. This models the corresponding statement:

> *The cheque has to be sent to the branch of the bank on which it is drawn.*

The specifications of $CREDIT$ and $DEBIT$ result from the split of the pre- and post-conditions of $TRANSF$. The pre-condition of $CREDIT$ states that the credited account number must exist and is different from the debited account. Its post-condition expresses the modification of the am map.

$$CREDIT\ (fr\_ac\colon Acno, to\_ac\colon Acno, a\colon \mathbb{N}_1)$$
$$\textsf{ext wr}\ am\ :\ Acno \xrightarrow{m} Acdata$$
$$\textsf{pre}\ to\_ac \in \textsf{dom}\ am \wedge fr\_ac \neq to\_ac$$
$$\textsf{post}\ am = \overleftarrow{am} \dagger \{to\_ac \mapsto \mu(\overleftarrow{am}(to\_ac), \\ bal \mapsto bal(\overleftarrow{am}(to\_ac)) + a)\}$$

Similarly, the pre-condition of $DEBIT$ ensures that the debited account exists, that it is different from the credited account, and that its balance and overdraft limit allow the transfer. The post-condition expresses the modification of the am map.

$$DEBIT\ (fr\_ac\colon Acno, to\_ac\colon Acno, a\colon \mathbb{N}_1)$$
$$\textsf{ext wr}\ am\ :\ Acno \xrightarrow{m} Acdata$$
$$\textsf{pre}\ fr\_ac \in \textsf{dom}\ am \wedge fr\_ac \neq to\_ac \\ \wedge\ bal(am(fr\_ac))\text{-}a \geq \text{-}od(am(fr\_ac))$$
$$\textsf{post}\ am = \overleftarrow{am} \dagger \{fr\_ac \mapsto \mu(\overleftarrow{am}(fr\_ac), \\ bal \mapsto bal(\overleftarrow{am}(fr\_ac))\text{-}a)\}$$

Using Hoare Logic or the logic of VDM, it is possible to show that the sequential combination of these operations is a valid implementation of $TRANSF$. But before performing this proof, the developer may want to convince himself that this refinement has the expected behaviour by animating it. So the transfer of 1000 between accounts 1 and 2 corresponds to the following sequential execution.

```
.> am
((1 "John" 25000 10000) (2 "Paul" 0 0)
 (3 "George" 5000 10000))
.> (credit 1 2 1000)
((1 "John" 25000 10000) (2 "Paul" 1000 0)
 (3 "George" 5000 10000))
.> (debit 1 2 1000)
((1 "John" 24000 10000) (2 "Paul" 1000 0)
 (3 "George" 5000 10000))
.>
```

This combination of operations produces the same result as the single execution of $TRANSF$. Being convinced of the correctness of his refinement, the developer may then proceed with its formal proof.

3.2 Refining the data structures

A careful examination of the trace of the previous animation reveals that, after the execution of $CREDIT$, the total amount of money recorded on the accounts of John and Paul (26000) is greater than the amount effectively stored in the bank (25000). The developer may suspect some error in his specification, as confirmed by a careful reading of the bank rules:

> *The money is not immediately transferred from the account of the writer of the cheque to your own account. Instead, the cheque has to be sent to the branch of the bank on which it is drawn. Only when the account of the writer of the cheque is debited is the value of the cheque transferred to your account - this process is called 'clearing' the cheque. While the cheque is being processed it is described as 'uncleared for interest'; when you actually receive the value of the cheque it is described as 'cleared for interest'.*
>
> ...
>
> *What is important for you is that if you withdraw the money from your account before the cheque is 'cleared for interest', you may become overdrawn for the purpose of calculating interest and you will then be charged interest on this overdraft.*

Instead of performing a further refinement of the control flow, the developer decides to refine his data structures in order to introduce an "effective balance" field in $Acdata$. This effective balance corresponds to the amount cleared for interest.

$Acno = \mathbb{N}$

$$\begin{array}{lrcl} Acdata :: & own & : & char^* \\ & bal & : & \mathbb{Z} \\ & ebal & : & \mathbb{Z} \\ & od & : & \mathbb{N} \end{array}$$

$$\textsf{inv}\ (mk\text{-}Acdata(own, bal, ebal, od)) \triangleq \\ \qquad bal \geq -od \wedge bal \geq ebal$$

The invariant associated to this new data type still enforces the relation between balance and overdraft. It also states that the effective balance is always lower than or equal to the balance. This means that the customer never has more on his account than reported, but may have less money than he thinks and subsequently might pay unexpected interest charges.

In order to prove that this refinement is correct, the developer must provide a "retrieve function" which returns the abstract representation from the concrete one. In this case, the retrieve function is obvious: it only masks the $ebal$ field of the concrete representation. The developer must then prove several mathematical properties of this function (e.g. for every valid abstract value, a concrete representation may be found). Here, the animation facilities of the KIDS/VDM environment don't really help.

Still, the refinement of the data structures also impacts on the specification of the associated operations. $CREDIT$ and $DEBIT$ should be refined in order to conveniently update the $ebal$ field of the accounts. Two alternate refinements may be considered at this stage.

- In the first refinement the $CREDIT1$ operation simply affects the balance. The effective balance remains unchanged; its modification only takes place when $DEBIT1$ is performed.

$$CREDIT1\ (fr\_ac : Acno, to\_ac : Acno, a : \mathbb{N}_1)$$
$$\textsf{ext wr}\ am\ :\ Acno \xrightarrow{m} Acdata$$
$$\textsf{pre}\ to\_ac \in \textsf{dom}\ am \wedge fr\_ac \neq to\_ac$$
$$\textsf{post}\ am = \overleftarrow{am} \dagger \{to\_ac \mapsto \mu(\overleftarrow{am}(to\_ac), \\ \qquad bal \mapsto bal(\overleftarrow{am}(to\_ac)) + a)\}$$

$$DEBIT1\ (fr\_ac : Acno, to\_ac : Acno, a : \mathbb{N}_1)$$
$$\textsf{ext wr}\ am\ :\ Acno \xrightarrow{m} Acdata$$
$$\textsf{pre}\ fr\_ac \in \textsf{dom}\ am \wedge fr\_ac \neq to\_ac \\ \quad \wedge\ bal(am(fr\_ac)) - a \geq -od(am(fr\_ac))$$
$$\textsf{post}\ am = \overleftarrow{am} \dagger \{fr\_ac \mapsto \mu(\mu(\overleftarrow{am}(fr\_ac), \\ \qquad bal \mapsto bal(\overleftarrow{am}(fr\_ac)) - a), \\ \qquad ebal \mapsto ebal(\overleftarrow{am}(fr\_ac)) - a), \\ \quad to\_ac \mapsto \mu(\overleftarrow{am}(to\_ac), \\ \qquad ebal \mapsto ebal(\overleftarrow{am}(to\_ac)) + a)\}$$

The animation gives thus the following trace[2]: $CREDIT1$ only modifies Paul's account and $DEBIT1$ modifies both. As far as the own, bal, and od field are concerned, the results of both abstract and concrete operations are identical, which is a necessary condition for a valid refinement.

```
.> (init)
((1 "John" 25000 25000 10000)
 (2 "Paul" 0 0 0))
.> (credit1 1 2 1000)
((1 "John" 25000 25000 10000)
 (2 "Paul" 1000 0 0))
.> (debit1 1 2 1000)
((1 "John" 24000 24000 10000)
 (2 "Paul" 1000 1000 0))
```

- In the second possible refinement, the transferred amount is immediately debited from the effective balance of fr_ac when $CREDIT2$ takes place. It is only credited to to_ac at the execution of $DEBIT2$. As far as the effective balance is concerned $CREDIT2$ is a debit operation and $DEBIT2$ is a credit: the transfer is in fact a sequence of debit followed by credit.

$$CREDIT2\ (fr\_ac : Acno, to\_ac : Acno, a : \mathbb{N}_1)$$
$$\textsf{ext wr}\ am\ :\ Acno \xrightarrow{m} Acdata$$
$$\textsf{pre}\ to\_ac \in \textsf{dom}\ am \wedge fr\_ac \neq to\_ac$$
$$\textsf{post}\ am = \overleftarrow{am} \dagger \{fr\_ac \mapsto \mu(\overleftarrow{am}(fr\_ac), \\ \qquad ebal \mapsto ebal(\overleftarrow{am}(fr\_ac)) - a), \\ \quad to\_ac \mapsto \mu(\overleftarrow{am}(to\_ac), \\ \qquad bal \mapsto bal(\overleftarrow{am}(to\_ac)) + a)\}$$

[2] For clarity sake, Georges's account, which remains unchanged, is not displayed in this trace.

$DEBIT2\ (fr\_ac{:}\ Acno, to\_ac{:}\ Acno, a{:}\ \mathbb{N}_1)$

ext wr $am\ :\ Acno \xrightarrow{m} Acdata$

pre $fr\_ac \in \mathrm{dom}\ am \land fr\_ac \neq to\_ac$
$\land\ bal(am(fr\_ac))\text{-}a \geq \text{-}od(am(fr\_ac))$

post $am = \overleftarrow{am} \dagger \{fr\_ac \mapsto \mu(\overleftarrow{am}(fr\_ac), bal \mapsto bal(\overleftarrow{am}(fr\_ac))\text{-}a), to\_ac \mapsto \mu(\overleftarrow{am}(to\_ac), ebal \mapsto ebal(\overleftarrow{am}(to\_ac))+a)\}$

The animation confirms this understanding of the specification: after $CREDIT2$ the total amount of money recorded on the effective balances is lower than the amount of money stored in the bank. Once again, one may check the validity of the refinement.

```
.> (init)
((1 "John" 25000 25000 10000)
 (2 "Paul" 0 0 0))
.> (credit2 1 2 1000)
((1 "John" 25000 24000 10000)
 (2 "Paul" 1000 0 0))
.> (debit2 1 2 1000)
((1 "John" 24000 24000 10000)
 (2 "Paul" 1000 1000 0))
```

Deciding which of these alternate concrete refinements is the right one is out of the scope of this paper. Presumably, this decision, which is mainly a business concern, will result from a discussion between the developer and a representative of the bank. The animation of the specification may help the communication between these persons.

During this dialog, new problems may arise like the following one. This trace shows how the interleaving of two transactions may lead Paul to a negative effective balance.

```
.> (init)
((1 "John" 25000 25000 10000)
 (2 "Paul" 0 0 0)
 (3 "George" 5000 5000 10000))
.> (credit2 1 2 15000)
((1 "John" 25000 10000 10000)
 (2 "Paul" 15000 0 0)
 (3 "George" 5000 5000 10000))
.> (credit2 2 3 10000)
((1 "John" 25000 10000 10000)
 (2 "Paul" 5000 -10000 0)
 (3 "George" 15000 5000 10000))
.> (debit2 2 3 10000)
((1 "John" 25000 10000 10000)
 (2 "Paul" 5000 -10000 0)
 (3 "George" 15000 15000 10000))
.> (debit2 1 2 15000)
((1 "John" 10000 10000 10000)
 (2 "Paul" 5000 5000 0)
 (3 "George" 15000 15000 10000))
```

It reveals a problem in the code of the bank: Paul will have to pay interest charges but, since he has no permitted overdraft, he never signed any agreement with the bank on the charged interest rates!

4 Conclusions

4.1 Uses of prototyping

The presentation of this case study has tried to demonstrate how a specification process may benefit from a prototyping tool. The use of this tool may take several forms:

Dialog support. It supports the dialog between a specifier and his customer. The execution of a prototype is usually closer to the customer's concerns than the mathematical statement of the specification. In the bank transfer case study, this interaction with the customer is needed to choose between $CREDIT1$ and $CREDIT2$.

Understanding. It helps the developer understand his specification because it leads him to take a different viewpoint at his specification than the purely mathematical viewpoint. For example, the animation of $CREDIT$ leads to discover that there is more money on the accounts than in the bank.

Error detection. The development or the execution of the prototype may fail and reveal an error in the specification.

Refinement support. Section 3 has shown that it allows to compare the behaviour of a refinement with respect to the corresponding abstract specification. This provides clues on the correctness of the refinement. Provided that the prototyping activity is fast and cheap, it becomes part of a Refine-Prototype-Proof process. In this process, proof activities are only undertaken when the developer is confident in the correctness of his refinement.

Data refinement has received less attention here than operation refinement. One way to support the refinement of data structures is to synthesize executable code for the retrieve function (here, a mask on the $ebal$ field). At animation time, this function is used to extract the abstract information from its concrete representation.

Tests. A record of the execution traces of the prototype may provide raw material for the definition of test scenarios for the final product.

The rigorous exploitation of these facilities necessitates methodological guidelines and support that go beyond the scope of this work.

In section 2.3.2, the VDM animator has been presented. It is an attempt to ease these activities by a user-friendly interface. The effective use of this tool would require further extensions like:

- the systematic recording of the execution sequences with a facility for replay; this would be the first step of an evolution towards a test environment;
- the possibility to execute in parallel a specification and its refinement in order to compare their behaviours.

4.2 Comparison with the IFAD toolbox

The IFAD toolbox [7] is currently the most mature tool to execute VDM specifications. It is based on an executable subset of VDM, where implicit specifications in terms of pre- and post-conditions must be associated to explicit executable code. Its scope is thus different from the one of KIDS/VDM.

The bank transfer case study does not use this explicit style, and therefore can not be executed with this toolbox. It started from an off-the shelf specification that was not meant to be prototyped. A translation of this specification into an explicit one would not be difficult. But, as stated in the beginning of this paper, there is a definite risk to focus on executability concerns at specification time. We believe that the interactive design process of KIDS/VDM allows to delay the introduction of these details until the synthesis activity that takes place in the KIDS program development mode.

4.3 Contribution of Knowledge-Based tools

The prototyping facility is based on two tools which contribute to its success:

- REFINE brings in a programming language that supports the VDM data structures and most operations on these structures. Experience with the KIDS/VDM system on about 100 small specifications showed that it was close enough to VDM to provide a good target language. The bank transfer case study showed that the overriding operator was missing. But this was not a major problem since it could be easily programmed.

- KIDS brings in an environment where every step is controlled. This guarantees the correspondence between the specification and its prototype. It also allows to record a development and to replay it afterwards.

  KIDS encapsulates design knowledge to synthesize code from specification and to further optimize code. Our experiments showed that good results could be reached in the prototyping process with a very basic design tactics (spec-to-code). This is due to the short semantical distance between VDM and REFINE and to the fact that many VDM specifications are closer to code than intended.

  Finally, KIDS provides theorem proving facilities that can be used to prove properties of VDM specifications. Experiments have shown the feasibility of these proof activities, but it is necessary to further tailor the theorem prover to the kind of proofs involved in VDM specifications.

In summary, both KIDS and REFINE provide powerful and reusable tools to KIDS/VDM.

References

[1] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, U.S. National Institute of Standards and technology, 1993.

[2] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. Technical Report PRG-TR-5-92, Oxford University Computing Laboratory, 1992.

[3] J.M. Spivey. *The Z notation - A Reference Manual (Second Edition)*. Prentice Hall, 1992.

[4] C. B. Jones. *Systematic Software Development Using VDM (Second Edition)*. Prentice-Hall, London, 1990.

[5] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE, Software Engineering Journal*, 4(6):320–338, November 1989.

[6] Peter Gorm Larsen. Response to "the formal specification of safety requirements for storing explosives". *Formal Aspects of Computing*, 6(5):565–568, 1994.

[7] R. Elmstrom, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox : a practical approach to formal specifications. *ACM SIGPLAN Notices*, 29(9):77–80, 1994.

[8] A. Kans and C. Hayton. Using ABC to prototype VDM specifications. *ACM SIGPLAN Notices*, 29(1):27–37, 1994.

[9] Y. Ledru and M.-H. Liégeois. Prototyping VDM specifications with KIDS. In *Proceedings of the 7th Knowledge-Based Software Engineering Conference*, pages 50–59. IEEE Computer Society Press, 1992.

[10] D.R. Smith, G.B. Kotik, and S.J. Westfold. Research on Knowledge-Based Software Environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, 11(11):1278–1295, 1985.

[11] D.R. Smith. KIDS: a semi-automatic program development system. *IEEE Transactions on Software Engineering — Special Issue on Formal Methods*, 16(9):1024–1043, 1990.

[12] Barclays Bank PLC. The Barclays code of business banking. Barclays Bank, Commercial Banking Division, 1991.

[13] L. Chung, B. Nixon, and E. Yu. Using non-functional requirements to systematically support change. In *RE '95 - Second IEEE international symposium on Requirements Engineering*, March 1995.

[14] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[15] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *Mural: A Formal Development Support System*. Springer-Verlag, 1991.

[16] Y. Ledru. Proof-based development of specifications with KIDS/VDM. In M. Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 214–232. Springer-Verlag, 1994.

[17] Y. Ledru and M.-H. Liégeois. Integrating REFINE prototypes in a VDM development framework. In B. Möller, editor, *Proceedings of the IFIP TC2 Working conference on Constructing Programs from Specifications*, pages 243–265. North-Holland, 1991.