Developing reactive systems in a VDM framework*

Y. Ledru

Unité d’Informatique, Université Catholique de Louvain, Place Sainte-Barbe 2, B-1348 Louvain-La-Neuve, Belgium

Abstract


This paper studies the detailed development of reactive systems, using an extension of VDM. The extension allows specification and proof of behavioural aspects to be expressed in the VDM framework. This is achieved by using traces of the input/output activities and introducing the notion of external entities whose behaviour is described by a state machine.

The major objective of this work is to improve understanding of the practical implications of the specification, design, and symbolic validation of machine-checked reactive systems.

1. Introduction

The specification, design, and validation of reactive systems are of growing interest. In particular, since these systems often play a major role in safety-critical applications, a lot of research has been carried out to enhance their validation. One of the key aspects of this validation is the establishment of a consistent correspondence between the software system under development and its specification. Moreover, if this specification is formal, it is possible to assess this link with machine-checked arguments.

A currently popular approach in this domain is the automated model checking of finite state systems [25]. The basic principle of this approach is to explore all the states of the reactive system and check whether they fulfill the formal specification. Unfortunately, these model-based techniques can only be applied to small systems due to the combinatorial explosion. In [9], Holzmann compares the size of one of his standard test protocols, which requires a 395-bit state vector (10^{119} states), to the state of the art of verification programs (10^7 states). Although recent

Correspondence to: Y. Ledru, Unité d’Informatique, Université Catholique de Louvain, Place Sainte-Barbe 2, B-1348 Louvain-La-Neuve, Belgium. Telephone: (+32-10) 473150. Fax: (+32-10) 450345. E-mail: yl@info.ucl.ac.be.

* Leibniz Project. This research has been funded in part by the Belgian SPPS under grant RFO/A1/15.
The present work extends the formal framework of VDM to the development of reactive systems. A methodological approach is proposed; it is based on a specification of the reactive system in the context of a description of its environment. This description is initially given as a state machine, and then translated into inference rules. The specification is thus expressed in the framework of VDM, which provides a single framework for both reactive and transformational aspects. The development itself may then be conducted as a series of VDM refinements and proofs. More than the design of yet another formal framework, the present work focuses on the practice of formal development and is the basis of an effective development method for reactive systems presented in [16]. By selecting VDM and state machine specifications (STATECHARTS in [16]), this work integrates industrial formalisms and allows the reuse of their associated infrastructure.

First of all, Section 2 discusses and defines the notion of reactive system. The methodological principles of the approach are given in Section 3. Section 4 shows how VDM can fit into this approach. Section 5 shows the application of the method to the development of the vending machine case study. Finally, Section 6 draws the conclusions of this study.

2. The notion of reactive system

The term “reactive system” was first introduced by Pnueli [7, 23] who distinguishes transformational and reactive systems. The fundamental characteristic of a reactive system is that it interacts with its environment.

The notion of environment leads to consider two basic entities in the development of reactive systems: the system and its environment. As will be seen later, it is
mandatory to make the distinction between these two clear. Moreover, most reactive systems are designed in order to operate in a precise environment. Therefore, the specification of a reactive system must include, at least implicitly and partially, a specification of its environment.

The important notion in reactive systems is the notion of interaction. Interaction may take place through input/output operations but also as a reaction to some elapsed time. It results in temporal dependencies between the reactive system and its environment.

3. The development method

The environment of a reactive system defines its relevant context with respect to specification and validation. It is thus natural to start the specification of a reactive system by stating the specification of its environment. For example, the specification of the vending machine starts with the specification of the customer's behaviour.

The environment of a reactive system is itself reactive. Specification languages for reactive systems can thus be used to model it. It appears from practice that state machine approaches, like the Statecharts [6], are well-suited for this purpose. However, since the case study developed here is relatively simple, a classical state automaton formalism is used to model this environment. Yet, the use of the Statecharts at this specification stage appears as a natural extension of the approach to the development of larger systems but involves more theoretical foundations. Once the environment is expressed in some state machine formalism, it is translated into inference rules which will fit into the syntactical framework of the rest of the development.

Another characteristic of the environment of reactive systems is that it may either cooperate with it or be hostile to it. Obviously, it is impossible for a reactive system to be productive in a hostile environment. All it can do is to enforce some safety properties. This justifies a two-fold structure for the specification of the reactive system:

- which goal the reactive system should achieve in a cooperative environment (whose behaviour is specified here by a state machine);
- which safety properties must be enforced if the environment no longer conforms to its state machine specification.

Once the context of the reactive system and its goal have been defined, it is necessary to specify the means that are available to meet this specification. This is the purpose of the target system specification which specifies the low level routines which allow interaction with the environment.

In summary, the specification of the reactive system is made up of:

- the specification of its environment;
the abstract specification of the system to develop, which states the goal to achieve in a cooperative environment and the safety properties to enforce otherwise; the specification of the target system.

Once this specification has been stated, the design and validation activities may begin. Their purpose is to establish a formal link between the abstract specification of the problem and the specification of the concrete target system in the context of the specification of the environment. This activity is carried out in the formal framework of VDM.

4. VDM aspects of the development

4.1. The VDM method

VDM is a mathematical verification method for software development: it has been developed and taught for many years, mainly by Jones and Bjørner [10].

A VDM development consists of several state descriptions at successive levels of abstraction and implementation steps which link the state descriptions. A state description is composed of:

- state variables;
- operations on the variables, specified in terms of pre- and post-conditions;
- an invariant on the state variables, which must be verified before and after the execution of any operation.

The implementation of an abstract state description $S_i$ by means of a more concrete one $S_{i+1}$ describes:

- either a data reification, i.e., a refinement in the data structures of the state variables;
- or an operation decomposition, i.e., how the operations of $S_{i+1}$ implement the ones of $S_i$.

The case study presented in Section 5 resulted in six state descriptions and thus five implementation steps.

At each stage of the VDM development, proof obligations must be verified to ensure that operations of a given state description are compatible with the state invariant and are implementable. Other proof obligations validate the implementation of one state description by means of a more concrete one. VDM provides rules to systematically derive these proof obligations.

4.2. Extension of VDM to reactive systems

VDM has been designed for transformational systems. Indeed, the specification of operations in terms of pre- and post-conditions establishes a link between the
Developing reactive systems in a VDM framework

*initial and final* configurations of the state variables. Moreover, VDM is concerned with *total correctness*: termination of an operation must be guaranteed. This appears insufficient for the specification of reactive systems.

1. Pnueli [23] mentions that some reactive systems never terminate. This is incompatible with the total correctness hypothesis of VDM.

2. The specification of reactive systems is not only concerned with initial and final states. The behaviour that is exhibited between these states is also relevant. In VDM, the only constraint that may be enforced *during* the execution of some operation is the state invariant of the most concrete state description.

3. The main activity of reactive systems is their interaction with the environment. Unfortunately, VDM is aimed at the specification of *closed* systems since it does not feature input/output primitives.

The proposed answers are the following ones.

1. Most reactive systems which never terminate actually repeat forever a limited number of operations. A VDM state description may be seen as a finite set of operations which are interleaved forever, provided that their pre-condition is verified when they are executed. In that sense, it is very similar to a UNITY specification [4], where simple operations, i.e., assignments, are applied repeatedly to global variables.

   An infinite execution is thus seen here as the infinite repetition of terminating activities. The proposed approach addresses the formal development of these terminating operations. It does not consider the problem of their composition into an infinite execution.

2. Since the specification only specifies the initial and final states, what happens during the execution of the operation will be recorded by some history variables. The post-condition will state properties of these variables when the final state of an operation is reached, specifying thus what happened *during* the operation. At the level of the most concrete state description, the state invariant will restrict the possible modifications of these history variables by the elementary operations (see Section 4.3).

   This technique is quite classical in the proof of CSP programs on the basis of trace variables (e.g. in [21]). It is also close to the concept of stream developed by Broy [2].

3. The reactive system and its environment can be seen together as one single closed system which falls within the scope of VDM.

   The first step in a VDM development is to list the *state variables* of the system. In this approach, the system is viewed as a combination of the reactive system and its environment. The variables which describe this global state are:

   • the *internal* variables of the reactive system;
• the internal variables of the environment (i.e., *external* with respect to the reactive system);
• the *interface* variables, i.e., the ones that may be accessed by both entities.

The internal variables of the reactive system are dependent on the application under development. The interface and external variables obey more systematic choices.

Since the environment is described in terms of a state machine (see Section 3), one of the external variables is *EnvState*, i.e., the current state of this machine. Another characteristic of the specification of the environment is that it only describes the normal behaviours. Therefore, a boolean variable named *BhrN* will record whether or not the environment keeps conforming to its state model. From the reactive system point of view, these two variables are external. This means that it cannot access these directly. Instead, interface variables are accessible to the reactive system which allow deduction to be made on the current state of the environment.

These interface variables are:

• *Intl* and *IntO*: the queues of respectively incoming and outcoming messages with respect to the reactive system. The elements of these queues are time-stamped (*ts*). The *Intl* queue may only be read by the reactive system and is written by the environment, and conversely for *IntO*.

The history of input and output messages is modeled as queues. This choice is motivated by proof concerns and does not mean that such queues correspond to the effective data structures of the target system. Actually, these history variables will only be partially accessed and do not require a full implementation.

• *Time*: a counter which measures the evolution of the "real time" common to both the system and its environment. This counter is read-only for the reactive system and its progress is achieved by the environment.

• *Istrd* records the position of the last incoming message read by the reactive system and the time of its last input attempt.

Figure 1 gives the VDM declaration of these variables. This figure adopts the notation of [11] with several enhancements such as the specification of the internal, interface, and external character of state variables. For readers unfamiliar with this notation, the specification expresses that *inMsg* and *outMsg* are composite types with fields *kind* and *ts*; *Intl* and *IntO* are sequences of respectively *inMsg* and *outMsg*.

These interface variables may not be accessed directly by the reactive system. This means that direct assignment or direct reading of these variables is not permitted. Access takes place through the low-level operations described in the specification of the target system.

The next step in stating a VDM specification is to define a *state invariant*. This invariant must be verified by every operation of the VDM state description. In this approach, the invariant is used to enforce the consistency between interface variables,
Developing reactive systems in a VDM framework

```plaintext
types
  inMsg :: kind : . . .
  ts   : N
outMsg :: kind : . . .
  ts   : N
state S of internal
  . . .
interface Intl :
  inMsg* lstrd :: nb : N
  time   : N
IntO :
  outMsg* Time : N
external
  BhvrN : B
  EnvState : {...}
end
```

Fig. 1. The state variables.

external variables, and the state machine specification of the environment. It also
enforces the internal consistency of the interface variables. The invariant features
three parts:

(I-1) If BhvrN is true, Intl and IntO must correspond to valid histories of input
and output messages with respect to the state machine description of the
environment SM_{Env}. Moreover, they must be compatible with the fact that
the machine has reached state EnvState at time Time. If BhvrN is false,
any behaviour recorded in the interface variables is permitted.

(I-2) The pointer nb(lstrd), which denotes the last input received by the reactive
system, must refer to some element of Intl.

(I-3) The numbers of the messages of Intl and IntO must be unique, monoto-
nically increasing, and consistent with the time-stamps used. Time must be
greater than any time-stamp already used.

The invariant may be written as:

\[ I_{\text{env}} \triangleq \text{consistent}(SM_{\text{Env}}, \text{EnvState}, \text{BhvrN}, \text{Intl}, \text{IntO}, \text{Time}) \]  \hspace{1cm} (I-1)
\[ \land \text{nb(lstrd)} \leq \text{len Intl} \]  \hspace{1cm} (I-2)
\[ \land \text{welltagged}(\text{Intl}, \text{IntO}, \text{Time}) \]  \hspace{1cm} (I-3)

The actual definitions of consistent and welltagged are not given. These predicates
are defined implicitly by the rules of Section 4.3.

The last part of a VDM specification is a list of operations. In this approach,
operations are applied to the global system. As classical VDM operations, they
specify the initial and final states of the global system in terms of pre- and post-
conditions. The actual behaviours of both entities during the execution of the
operations are recorded in the interface variables and constrained by the state invariant.
Reactive operations are the ones which include interface or external variables in their \textit{wr} (writable) variables.

4.3. \textit{The axiomatization}

The evolution of the interface and external variables obeys several rules which result from the invariant.

- \( B_{\text{hor}} \) is \textit{true} unless it gets \textit{false}. Once \textit{false}, the variable stays \textit{false} (\( \neg B_{\text{hor}} \) is a stable property).
  \[ \vdash \neg B_{\text{hor}} \Rightarrow \neg B_{\text{hor}} \]  (P-1)

- The allowed evolutions of \( \text{IntI} \), \( \text{Time} \), \( \text{IntO} \), and \( \text{lstrd} \) are restricted:
  - The interface queues may only be modified by appending new items. The beginning of the sequence must remain the same (P-3, 4).
    \( \text{Time} \) only flows into one direction, i.e., it may only increase (P-2).
  - \( \text{lstrd} \) may only increase, both as a pointer to \( \text{IntI} \) and as a time-stamp (P-11, 12). This variable is affected by any input operation.

These rules refer to the initial and final states of operations. The "hook" symbol \( - \) is a VDM notation which denotes the state in which the operation was started.

\[ \vdash \text{Time} \leq \text{Time} \]  (P-2)
\[ \vdash \exists x \in \text{inMsg}^* \cdot \text{IntI} = \text{IntI} \bowtie x \]  (P-3)
\[ \vdash \exists x \in \text{outMsg}^* \cdot \text{IntO} = \text{IntO} \bowtie x \]  (P-4)

The time-stamps are related to \( \text{Time} \):

\[ \vdash \forall i \in \{1..\text{len IntI}\} \cdot ts(\text{IntI}[i]) \leq \text{Time} \]  (P-5)
\[ \vdash \forall i \in \{\text{len IntI} + 1..\text{len IntI}\} \cdot ts(\text{IntI}[i]) \geq \text{Time} \]  (P-6)
\[ \vdash \forall i \in \{1..\text{len IntO}\} \cdot ts(\text{IntO}[i]) \leq \text{Time} \]  (P-7)
\[ \vdash \forall i \in \{\text{len IntO} + 1..\text{len IntO}\} \cdot ts(\text{IntO}[i]) \geq \text{Time} \]  (P-8)

The time-stamps are monotonically increasing:

\[ \vdash \forall i, j \in \{1..\text{len IntI}\} \cdot i < j \Rightarrow ts(\text{IntI}[i]) \leq ts(\text{IntI}[j]) \]  (P-9)
\[ \vdash \forall i, j \in \{1..\text{len IntO}\} \cdot i < j \Rightarrow ts(\text{IntO}[i]) \leq ts(\text{IntO}[j]) \]  (P-10)

Also, \( \text{lstrd} \) may only be modified in one way:

\[ \vdash time(\text{lstrd}) \leq time(\text{lstrd}) \]  (P-11)
\[ \vdash nb(\text{lstrd}) \leq nb(\text{lstrd}) \]  (P-12)

The evolution of the fields of \( \text{lstrd} \) must follow the next rule:

\[ \vdash nb(\text{lstrd}) \neq nb(\text{lstrd}) \Rightarrow time(\text{lstrd}) > \text{Time} \]  (P-13)
but also:

\[ \text{time}(lstrd) \leq \text{Time} \quad \text{(P-14)} \]

These rules are implicit post-conditions added to each operation. The implicit character of these rules is intended to make the post-conditions more readable.

These rules can also be considered as a proving context linked to the interface and external variables which results from the verification of the invariant. A supplementary rule may establish this link:

\[
\begin{array}{c}
I_{\text{env}} \\
P-1..P-14
\end{array}
\]

These implicit post-conditions result in additional proof obligations. Indeed, the post-conditions are involved in two kinds of proof obligations: the implementability and the proofs of implementations.

- As far as implementations are concerned, since these predicates are added to all operations and provided the interface variables are not affected by data reifications, the proofs are always satisfied. This assumes that the behaviour of the environment is never refined. This assumption will be discussed later.
- From the point of view of implementability, this is an additional proof obligation. But as it is discussed at the end of this paper, implementability does only make sense at the target level.

4.4. Observed behaviour

The actual values of the external variables (i.e. \(BhvrN\) and \(EnvState\)) may not be accessed directly by the reactive system. They must thus be inferred from the observation of the interface variables. Several inference rules which result from the specification of the environment can be used to establish the proofs about these variables.

An important notion is the observed correct behaviour of the environment, i.e. did the environment “look” normal up to now with respect to the specification of its normal behaviour. This notion can be encapsulated into a boolean function \(obsNormal\) which reports whether the observed behaviour is consistent with its specification. The arguments of \(obsNormal\) are:

\[obsNormal(SM_{\text{env}}, INTI, INTO, Time, lstrd)\]

This predicate has several interesting properties:

- If the behaviour is normal, then it is observed as normal: (O-1).
- If the behaviour was once observed as abnormal, then it remains abnormal forever: (O-2).
- If the behaviour was observed as normal at some time, the \(obsNormal\) function must remain true unless some new input operation is performed: (O-3).
The associated rules are:

\[
\begin{align*}
BhvN & \quad ObsN & & \quad \neg ObsN & \quad \neg ObsN & \quad \text{ObsN} \land \text{lstrd} = \text{lstrd} \\
(O-1) & & (O-2) & & (O-3)
\end{align*}
\]

with the following shortcuts:

\[
\text{ObsN} = \text{obsNormal}(SM_{Env}, \text{IntI}, \text{IntO}, \text{Time}, \text{lstrd}),
\]

\[
\neg \text{ObsN} = \text{obsNormal}(SM_{Env}, \neg \text{IntI}, \neg \text{IntO}, \neg \text{Time}, \text{lstrd}).
\]

Once \(\text{ObsN}\) is invalidated, it remains invalidated. It must thus be true at the initial time. This results in the following rule:

\[
\begin{align*}
\text{nb(lstrd)} = 0 & \land \text{time(lstrd)} = 0 \\
\text{ObsN} & \quad (O-4)
\end{align*}
\]

5. The vending machine case study

5.1. Specification

The case study discussed here is the vending machine, taken from [14]. In this instantiation of the vending machine, it dispenses chocolates when it has got either two quarter ECU coins (two times \(1q\)) or one half ECU (\(1h\)) coin. The vending machine is a typical example of reactive systems: its environment is the customer and interactions take the form of inputs of coins and outputs of chocolates.

A more detailed description of this case study may be found in [15].

5.1.1. The environment

The first step of the specification process models the environment of the reactive system (the customer) as a state machine diagram (Fig. 2).
Developing reactive systems in a VDM framework

The initial state of the customer is hungry. He may input either one half coin or two quarter coins in the reactive system. The choice between both is internal to the customer. He eventually reaches \textit{paid1h} or \textit{paid2q}.

The customer waits until the vending machine delivers him one chocolate. Once he has got the chocolate, he goes away.

The axiomatization of the environment

The state machine described in Fig. 2 can be translated into rules that can be used in VDM proofs. This translation process is still informal but appears to be quite systematic.

Inference rules (SM-1.1) to (SM-1.7) correspond to the normal cases. They all stand under the assumption that $BhvrN$ is and remains true.

Stable states

A notion of \textit{from} state is defined which corresponds to the last state where some interaction took place; but the state may have changed due to internal actions. Some states are stable, i.e. once the automaton has reached this state it can only leave it after the occurrence of some external event. This is the case for \textit{paid1h}, \textit{paid2q}, and \textit{going}.

The rules SM-1.1 to SM-1.3 express that \textit{paid1h}, \textit{paid2q}, and \textit{going} are stable.

\begin{align*}
\text{from } \textit{paid1h} & \quad \text{EnvState} = \textit{paid1h} & \text{(SM-1.1)} \\
\text{from } \textit{paid2q} & \quad \text{EnvState} = \textit{paid2q} & \text{(SM-1.2)} \\
\text{from } \textit{going} & \quad \text{EnvState} = \textit{going} & \text{(SM-1.3)} 
\end{align*}

The \textit{hungry} and \textit{paid1q} states are not stable, i.e. they can be left after an internal action. In these cases, \textit{from} means that the state is either the state itself or one of the following ones (with the corresponding values of $IntI$).

Transitions between states

Rule (SM-1.4) describes the normal behaviour at state \textit{paid1h}. It expresses that if:

1. the customer behaves normally,
2. he has reached state \textit{paid1h} at the start of some operation of the reactive system,
3. the operation appends one message of the kind \textit{chocolate} to the $IntO$ queue,

then the customer has reached state \textit{going} at the end of the operation. (SM-1.5) is a similar rule for \textit{paid2q}. 

Rule (SM-1.6) corresponds to the transition from \( \text{paid}1q \) to \( \text{paid}2q \). It states that if:

1. the customer behaves normally,
2. he has reached state \( \text{paid}1q \) at the start of the operation,
3. some new input has been read by the vending machine,

then he has output one quarter coin and has reached state \( \text{paid}2q \).

Similarly, rule (SM-1.7) expresses the transitions starting from the other unstable state, \( \text{hungry} \).

Abnormal observations

The next rules define the \( \text{ObsN} \) predicate. They thus refer to observations of the customer's behaviour from the vending machine point of view. (SM-2.1) states that if:

1. the system was supposed to have reached \( \text{hungry} \) before the execution of some operation,
2. this operation ends after a \( \text{time-out} \) amount of time and, in the meantime, no message was issued by the customer

then it has been observed that the environment behaves abnormally.
Developing reactive systems in a VDM framework

(SM-2.2) states the same property for state paid \(1q\). These rules express that the environment of a reactive system leaves unstable states within a predefined amount of time. This is mandatory to distinguish between bad and good behaviours within a finite period and thus ensures the termination of VDM operations.

Rules (SM-2.3) and (SM-2.4) express that paid\(1h\) and paid\(2q\) are stable states where the environment cannot issue messages to the system.

\[
\begin{align*}
\text{BhrN} & \Rightarrow \text{from paid}\,1h \\
\text{len IntI} & > nb(\text{lstd}) \\
\neg \text{ObsN} & \\
\text{(SM-2.3)}
\end{align*}
\]

\[
\begin{align*}
\text{BhrN} & \Rightarrow \text{from paid}\,2q \\
\text{len IntI} & > nb(\text{lstd}) \\
\neg \text{ObsN} & \\
\text{(SM-2.4)}
\end{align*}
\]

Rules (SM-2.5) and (SM-2.6) state which input operations are expected when the environment has reached hungry and paid\(1q\).

\[
\begin{align*}
\text{BhrN} & \Rightarrow \text{from hungry} \\
\text{len IntI} & > nb(\text{lstd}) \\
\text{kind(IntI}[nb(\text{lstd})+1]) & \neq 1q \\
\text{kind(IntI}[nb(\text{lstd})+1]) & \neq 1h \\
\neg \text{ObsN} & \\
\text{(SM-2.5)}
\end{align*}
\]

\[
\begin{align*}
\text{BhrN} & \Rightarrow \text{from paid}\,1q \\
\text{len IntI} & > nb(\text{lstd}) \\
\text{kind(IntI}[nb(\text{lstd})+1]) & \neq 1q \\
\neg \text{ObsN} & \\
\text{(SM-2.6)}
\end{align*}
\]

**Normal observations**

Rules (SM-2.7) to (SM-2.9) are used to assess a “normal” behaviour. Rule (SM-2.7) states that if:

1. the environment has reached state hungry and was observed as normal until the start of some operation,
2. the environment has issued messages during the operation,
3. the first message is a quarter coin message,
then it is observed as normal at the end of the operation.

Rules (SM-2.8) and (SM-2.9) state the same property for the other internal transitions.

\[
\begin{align*}
\text{ObsN} \land I_{\text{env}} \\
\text{BhrN} & \Rightarrow \text{from hungry} \\
\text{len IntI} & > nb(\text{lstd}) \\
\text{kind(IntI}[nb(\text{lstd})+1]) & = 1q \\
\text{ObsN} & \\
\text{(SM-2.7)}
\end{align*}
\]

\[
\begin{align*}
\text{ObsN} \land I_{\text{env}} \\
\text{BhrN} & \Rightarrow \text{from hungry} \\
\text{len IntI} & > nb(\text{lstd}) \\
\text{kind(IntI}[nb(\text{lstd})+1]) & = 1h \\
\text{ObsN} & \\
\text{(SM-2.8)}
\end{align*}
\]
This axiomatization of the state machine was not meant to be complete. Its goal is to provide a set of inference rules which is sufficient to prove the development. In the current state of this research, there is no systematic way of deciding whether enough rules have been extracted from the state machine specification of the customer, it is a matter of experience.

5.1.2. Abstract specification of the vending machine

The abstract specification of the vending machine groups the declaration of the state variables with an invariant and a list of operations. These are defined in the context of types and constants definitions. $MaxChoc$ is the maximum number of chocolates which may be stored in the vending machine; $SM_{Env}$ is the state machine of Fig. 2 which describes the behaviour of the customer.

$$
\text{constants } \begin{aligned}
\text{MaxChoc} & : \mathbb{N}, \\
\text{SM}_{Env} & : \text{StateMachine}
\end{aligned}
$$

$$
\text{types } \begin{aligned}
\text{inMsg} & :: \text{kind} : \{1q, 1h, undefined\} \\
\text{ts} & : \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
\text{outMsg} & :: \text{kind} : \{\text{chocolate, 1q, 1h, undefined}\} \\
\text{ts} & : \mathbb{N}
\end{aligned}
$$

$\text{VendingMachine}_0$, the global state of the system, groups the external and interface variables described in Section 4.2 with the internal variables of the reactive system, i.e. the number of chocolates left in the machine and the total amount of money collected. The invariant is extended to these internal variables and states the relationship between the number of chocolates sold and the money collected (measured in quarters). The $\text{init}$ field states the properties of the initial state.

$$
\begin{aligned}
\text{state } \text{VendingMachine}_0 \text{ of} & \\
\text{internal } \begin{aligned}
N\text{choc} & : \{0..\text{MaxChoc}\} \\
\text{Money} & : \mathbb{N}
\end{aligned} \\
\text{interface } \begin{aligned}
\text{IntI} & : \text{inMsg}^* \\
\text{Istrd} & :: \text{nb} : \mathbb{N} \\
\text{time} & : \mathbb{N}
\end{aligned} \\
\text{IntO} & : \text{outMsg}^* \\
\text{Time} & : \mathbb{N} \\
\text{external } \begin{aligned}
\text{BhvrN} & : \mathbb{B} \\
\text{EnvState} & : \{\text{hungry, paid}\,1q, \text{paid}\,2q, \text{paid}\,1h, \text{going}\}
\end{aligned}
\end{aligned}
$$
Developing reactive systems in a VDM framework

\[ \text{inv } \text{Env} \land \text{Money} = (\text{MaxChoc} - \text{Nchoc}) \times 2 \]

\[ \text{init } \text{Nchoc} = \text{MaxChoc} \land \text{Money} = 0 \land \text{IntO} = [] \land \text{nb}(\text{Istrd}) = 0 \land \text{time}(\text{Istrd}) = 0 \land \text{Env} \]

end

The specification features two abstract operations.

- **Transaction** is the actual interaction with the customer and the only reactive activity. The \text{ext wr} field lists the variables of the global state which may be affected by the operation. The structure of its post-condition is two-fold as expected:
  - if the customer is observed not to behave as specified ($\neg\text{ObsN}$), then the chocolates are not delivered;
  - if the customer had a normal behaviour, he must get a chocolate.

  \[ \text{Transaction} \]
  \text{ext wr} \text{Nchoc, Money}
  \text{wr} \text{IntI, Istrd, IntO, Time}
  \text{wr} \text{BhvrN, EnvState}
  \text{pre from hungry} \land \text{Nchoc} > 0
  \text{post} (\neg\text{ObsN} \Rightarrow \text{Nchoc} = \overline{\text{Nchoc}})
  \land (\text{BhvrN} \Rightarrow \text{EnvState} = \text{going} \land \text{Nchoc} = \overline{\text{Nchoc}} - 1)
  \land \text{Money} = (\text{MaxChoc} - \text{Nchoc}) \times 2 \land \text{Env} \]

- **Refill** refills the machine with chocolates and collects the money.

  \[ \text{Refill} \]
  \text{ext wr} \text{Nchoc, Money}
  \text{post} \text{Nchoc} = \text{MaxChoc} \land \text{Money} = 0

5.1.3. The target system

The target system specification is not detailed here and can be found in [15]. The main operations featured at this level are:

- **ReadCoin**(lc, res) which gets the next coin from the customer and puts it into a buffer (Payment); lc returns the kind of the next coin and res reports on the success of the read attempt;
- **StoreCoins** which empties the Payment buffer into some money storage (MoneyStore);
- **Payback** which returns the contents of the Payment buffer to the customer;
- **DeliverChoc** which delivers one chocolate to the customer;
- **RefillChoc** which refills the vending machine with chocolates and empties the money storage (MoneyStore).

As an example, here is the specification of **DeliverChoc**:
DeliverChoc

\text{ext \, wr \, Nchoc}
\text{\, wr \, IntI, \, IntO, \, Time}
\text{\, wr \, BhvrN, \, EnvState}
\text{pre \, Nchoc > 0}
\text{post \, Nchoc = Nchoc - 1}
\begin{align*}
\land & \text{\, \, \, \, \, \, \, \, \, kind(IntO[len \, IntO]) = \text{chocolate}} \\
\land & \text{\, \, \, \, \, \, \, \, \, len \, IntO = len \, IntO + 1} \\
\land & I_{env2}
\end{align*}

This specification states that one element of type \text{chocolate} has been appended to \text{IntO}.

This specification is quite typical of target systems operations. Unlike \text{Transaction}, the specification of \text{DeliverChoc} does not make use of the external variables in its pre- and post-conditions. It only accesses interface variables: the tail of the output sequence is modified. References to \text{BhvrN} and \text{ObsNorm} also disappear from the specification.

5.2. Development

The development now proceeds as a normal VDM development. It can be found, with the complete set of proofs in [15]. Several design steps link the abstract specification to the target system specification. These steps and the intermediate state descriptions they introduce are then validated by several proofs.

5.2.1. Design steps

Design steps involve operation decompositions and data reifications.

Operation decompositions

The abstract operations \text{Transaction} and \text{Refill} are refined as:
\begin{align*}
\text{Transaction} & \equiv \text{GetPayment} ; \text{GiveChoc} \\
\text{GetPayment} & \equiv \begin{dcases*}
\text{if \, ok \, then \{ReadCoin(lc, \, res);} \\
\text{\, \, \, \, \, \, \, \, \, \, \, \, if \, res \land \, lc = 1h \, then \, StoreCoins} \\
\text{\, \, \, \, \, \, \, \, \, \, \, \, else \, if \, res \land \, lc = 1q \, then \, Get1q} \\
\text{\, \, \, \, \, \, \, \, \, \, \, \, \, \, \, else \{PayBack; \, ok := false\}}
\end{dcases*} \\
\text{Get1q} & \equiv \begin{dcases*}
\text{ReadCoin(lc, \, res);} \\
\text{\, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, if \, res \land \, lc = 1q} \\
\text{\, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, then \, StoreCoins} \\
\text{\, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, \, else \{PayBack; \, ok := false\}}
\end{dcases*} \\
\text{GiveChoc} & \equiv \begin{dcases*}
\text{if \, ok \, then \, DeliverChoc}
\end{dcases*} \\
\text{Refill} & \equiv \text{RefillChoc}
\end{align*}

\text{ok} \text{ is a variable which is initially set to true.}
Data reifications

Most global variables are not modified by the development. The main reifications are:

- \( Money \xrightarrow{\text{reif}} \text{SumItems(Payment)} + MoneyStore \), i.e. the money is stored in both Payment and MoneyStore;
- the introduction of some intermediate variables: res, ok, and lc.

5.2.2. Proofs

The last stage of the development validates it through mathematical proofs. Since the interface variables which rule the behaviour have been integrated in the VDM framework, this proof process is a classical VDM activity. Yet, the set of available proof rules now includes the axioms and inference rules presented in Sections 4.3 and 5.1.1.

6. Conclusions

Three typical case studies have appeared in the field of reactive systems. The digital watch is dedicated to the illustration of specification languages. The alternating-bit protocol and the vending machine are better suited to illustrate the design and validation activities.

The development of the vending machine case study with this extended version of VDM does not guarantee that the approach can be applied to real-size problems (as discussed in Section 6.3). Nevertheless, it demonstrates that the behavioural aspects, which are the main characteristics of reactive systems, can be successfully handled by the approach.

The same case study has already been presented by other authors (e.g. Lamport in [14] or Misra in [20]). What distinguishes this particular treatment of the problem from other approaches is that the environment is seen here as the starting point of the development and provides a context for the subsequent activities. This development approach is further detailed in [16].

The vending machine case study has been treated completely. The resulting development may appear quite long: six state descriptions and sixty pages of proofs (including comments) [15]. It results from the efforts that were spent to keep the proofs very detailed to facilitate checking. The length also results from the choice of VDM which is a verbose notation. Moreover, although the proof was not machine-checked, it has reached a sufficient level of preciseness to allow its coding into some theorem prover.

6.1. Tool support

The primary goal of this work is to progress towards symbolic development of machine-checked reactive systems. This case study and previous experiments with
the B theorem prover \cite{12} have provided important insight in the requirements for tool support. Indeed, the additional amount of details required to allow automated checking of the proofs cannot be handled without effective help from the tool.

Fortunately, it appears from the case study that many proof steps are elementary propositional calculus steps which can be automatically performed by theorem provers. Similarly, many proofs related to the state machine properties are located at one node of the automaton and may be expected to be proved automatically. It may thus be expected that appropriate tool support would automatically handle the most clerical parts of the proofs. This would allow the developer to concentrate on the most insightful proof steps.

In this case study, all proofs were not conducted in a uniform manner. Some are goal-driven (backward-chaining), others start from the premises (forward-chaining). Most proofs are actually a mixture of these styles. Moreover, the notations for proofs may vary from one proof to the other. The tool must thus allow for flexibility in both the concrete syntax of the proofs and the way they are conducted.

6.2. Implementability

One of the proof obligations that a VDM state description must satisfy is the implementability of its operations. This proof obligation checks that there exists a configuration of the state variables which verifies the post-condition of the operation, provided this operation started in a state which satisfies both the pre-condition and the state invariant.

In this extension of VDM to reactive systems, implementability is quite easy to satisfy. For example, a configuration of the state variables where $BhrN$ is true and where all other variables are equal to their initial value (i.e. the system did nothing) satisfies the post-condition of $Transaction$. Unfortunately, although there is a state which satisfies the post-condition, there does not exist any sequence of actions of the reactive system such that $BhrN$ eventually becomes false, because it is an external variable. This proof obligation should thus be modified so that only “reachable” states are considered, i.e. states which can be reached by the execution of the target system operations. This also means that implementability is no longer a property of the state description itself but is a property concerning the target system. Moreover, such an implementability proof would probably turn out to be as complex as the proof of the actual development.

Another way to assess implementability is to actually implement the specification. The proof obligation is now reduced to:

- the proof of the development;
- the implementability proof of the target system operations.

Fortunately, the target system specification is supposed to model some real system. These low-level operations thus correspond to some actual implementation. Implementability of the target system can thus be considered as a reasonable
Developing reactive systems in a VDM framework

hypothesis of the formal development. The implementability proof obligation may still be useful at this stage to improve the developer's confidence in his specification.

6.3. Extensions of the approach

This paper has presented the basic ideas of an approach to the development of reactive systems in a VDM framework. By restricting the access to the interface and external values, and by using history variables constrained by the state invariant, it has addressed the problem of specifying and developing reactive operations. These ideas were explored from an engineering point of view which aimed at understanding the practical implications of the specification, design, and symbolic verification of machine-checked reactive systems. Further work is certainly needed to assess and complete the theoretical basis of the approach.

Moreover, this approach suffers from some limitations which prevent its application to real problems. Hopefully, other extensions to VDM may help overcome these limitations:

- The size of the applications is limited by both the state machine and VDM notations which do not allow the encapsulation into modules. This problem can be solved by allowing evolution of the state automaton notation towards more elaborate formalisms like the Statecharts and by using a modular version of VDM (e.g. [19] or RAISE [22]). In [16], the use of Statecharts instead of flat state machines is explored. The combined use of Statecharts and VDM is not new. In [18], Marshall used these languages for the specification of user interfaces. In her work, Statecharts diagrams specify the control flow of elementary operations specified in VDM.

- The final program of the reactive system is sequential because VDM does not allow parallel constructs. From this point of view, VDM+ [28] and RAISE are interesting substitutes to VDM.

- Another extension of the approach is required to allow refinement of the interface and external variables. For example, in a communication protocol it is sensible to speak about packets at the most abstract level and sequences of bits at the target level. Such a refinement would result in refinements of the implicit post-conditions of Section 4.3 and supplementary proof obligations to be satisfied.

Acknowledgement

I want to thank Michel Sintzoff for his remarks on a draft version of this paper as well as for his constructive comments during this work. Pierre-Yves Schobbens was of great help for detecting and solving the technical problems. The remarks of Pierre Collette and Philippe Massonet on several versions of this paper helped to make it more comprehensible. Finally I must thank the reviewers of both the

References

