Abstract

The paper presents a case study of application of the VDM formal method to specification and verification of a simple real-time kernel. Specifications of selected external services of the kernel are presented. Then the verification methodology is introduced by demonstrating its basic steps in relation to verification of a selected function - a process waiting for a signal on a condition variable. The experience from the study is subsumed in the last section.

1 Introduction

Typically, embedded systems are based on a small kernel which provides process and interrupt handling services. This article reports on a case study made using the VDM [3] notation, a mathematical specification language, to specify and verify a kernel which supports the concurrent programming laboratory. Previously the kernel was used to support a telecommunication switching system.

The kernel supports the following services:

- Process Management: creation, start and termination of processes,
- Time Management: time-out and pausing of processes,
- Receiving Interrupts: real-time clock ticks and device interrupts,
- Process Scheduling: assigning and changing process priorities,
- Synchronisation Support: mutual exclusion in monitors, signalling in monitors.

The kernel implements a simple process scheduling principle: At any time, active is a ready process with the highest priority; in the case of equal priorities the First-In-First-Out rule is in effect.

The synchronisation services offered by the kernel support implementation of monitors - internally synchronised modules with procedural interfaces.

In the initial documentation, the states that a single process may occupy and the possible transitions between them (caused by a kernel call from the process, or by external interrupt) were represented as a finite state machine. Although diagrams like this are commonly in use, they convey little information that a programmer can use while developing programs that rely on the kernel and in fact they reveal much information that is irrelevant, e.g. they suggest that the kernel is a set of finite-state machines, one per process. However it does not show process interactions and makes it difficult to show combined states (e.g. a ready process with a time-out set). To handle this problem the number of states would grow exponentially.

It was decided to replace the initial (informal) documentation by a formal one in order to avoid problems in communicating the semantics of the kernel to programmers.

The goal of the study was twofold: (1) develop a precise external specification of the kernel that could then be used by students to develop their laboratory programs, (2) formally verify the kernel design against this specification to increase confidence in its reliability and robustness.

2 VDM - a short introduction

The Vienna Development Method (VDM) [3] is a formal development method for software. It is based on the VDM specification language which provides constructs for mathematical notions like sets, sequences, mappings and compound types together with a subtyping mechanism (by giving invariants on a type). Using those well understood mathematical notions, a system designerspecifier can define data types suitable for the description of the
intended system without being forced to decide how these are to be represented in the implementation language. The specification is built around definition of the state of the specified object. The state can be manipulated by (possibly nondeterministic) operations. The specification can also define functions which are purely applicative and can not alter the state. Both functions and operations can be defined implicitly, without giving any implementation details, by providing pre- and post-conditions. When applied during development, the method starts from an initial (most abstract) specification and the subsequent development steps involve the choice of more concrete (i.e. closer to an implementation) representations of the data types and redefinition of the functions and operations as to correspond to the new data types. The relationship between the subsequent specifications is formally represented by the retrieve function which uniquely associates each element of the more concrete data type to elements of the more abstract one. This provides a formal framework within which the (abstract and more concrete) specifications can be compared. The method requires the discharging of certain proof obligations arising from these steps which ensure that the correspondence between data types, functions and operations is correct. The development process may involve several layers of specification before the final implementation is achieved. Compositionality of the method ensures that once the proof obligations for each layer and between layers have been discharged, the final implementation is correct relative to the initial specification.

Tools are essential in a formal development to avoid type errors and enable proofs of realistic systems. There are currently only a few industrialised tools for formal methods although there are much more that are at an advanced stage of development. It is essential that any tools used are trustworthy, but their verification and validation is difficult because they are often large programs. There are also problems with the reliability of hosting systems such as UNIX. Tools can be general, i.e. addressing several methods, specific to a particular method or addressing only a particular phase within a given method. At present stage, the tools which are practically applicable to realistic problems are mainly from the third of the above classes. During our study we applied SpecBox [5], an interactive window-based tool which supports creation and maintenance of VDM specifications. The \texttt{\LaTeX} generator of SpecBox produces a source file for the \texttt{\LaTeX} document preparation system that prints the specification in the BSI protostandard mathematical syntax. The attempts to use MURAL [4], the system designed to support the construction of formal specifications and refinements, and to aid the verification and validation of these specifications were rather unsuccessful, mainly because at that time it did not support the full set of the BSI-VDM notation.

### VDM notational conventions

The full description of the VDM specification language and its semantics is given in [3].

Symbols used in this paper are described here:

- \( \mathbb{Z} \) the set of integers
- \( \mathbb{B} \) \{ true, false \}
- \( T_1 \mid T_2 \) union type
- \( m \{ ... \} \) composite object generator
- \( \mu(o, s_1 \leftarrow v) \) modify component \( s_1 \) of object \( o \) with value \( v \)
- \( T \text{-set} \) all finite subsets of \( T \)
- \( \{ x \mid f(x) \} \) finite sequences of elements of \( T \)
- \( \text{hd } s \) head of sequence \( s \)
- \( \text{tl } s \) tail of sequence \( s \)
- \( \text{elements } s \) elements of sequence \( s \)
- \( \text{length } s \) sequence enumeration
- \( \text{finite map} \) finite map
- \( \text{one-to-one map} \) one-to-one map
- \( \text{domain of map} m \) domain of map \( m \)
- \( \{ d_1 \leftarrow r_1, d_2 \leftarrow r_2, \ldots \} \) map enumeration
- \( m_1 \vdash m_2 \) overwriting \( m_1 \) by \( m_2 \)

Basically, a VDM specifications define a collection of state variables (which contents are given in terms of the above mathematical constructs) and a set of operations which manipulate the state. The operations are defined in terms of the pre- and post-conditions which refer to two states. Hooked variables are used for the before state and unhooked variables are used for the after state in the conditions defining an operation.

### 3 Specification of kernel state

The kernel specification begins with the description of the state space, the collection of variables with associated types and the invariant relationship which
additionally restricts possible values for the state variables.

\[
\text{state } \text{Kernel of} \\
\begin{align*}
\text{a} & : \text{Process-name} \\
\quad & \text{-- active process} \\
\text{fn} & : \text{Process-name-set} \\
\quad & \text{-- free names} \\
\text{na} & : \text{Process-name-set} \\
\quad & \text{-- suspended processes} \\
r & : \text{Process-name}^* \\
\quad & \text{-- ready processes} \\
p & : \text{Process-name} \rightarrow \text{Priority}^* \\
\quad & \text{-- processes' priorities} \\
w & : \text{Condition} \rightarrow \text{Process-name}^* \\
\quad & \text{-- waiting processes} \\
pa & : \text{Process-name} \rightarrow \text{Time} \\
\quad & \text{-- pausing processes} \\
tou & : \text{Process-name} \rightarrow \text{Time} \\
\quad & \text{-- process with time-out set} \\
wto & : \text{Process-name} \rightarrow \text{B} \\
\quad & \text{-- 'time-out elapsed' flag of a process} \\
ts & : \text{Process-name}^* \\
\quad & \text{-- real time synchronisation queue} \\
wtk & : \text{B} \\
\quad & \text{-- real time tick flag}
\end{align*}
\]

Processes are named by \text{Process-names} which belong to a fixed interval \([0,\text{Maxname}]. \) It is assumed that process 0 exists initially and its activity eventually gives rise to other processes. Thus, the initial condition can be expressed as follows:

\[
\text{init mk-Kernel}(a,fn,na,r,p,w,pa,tou,wto,ts,wtk) \triangleq \\
a = 0 \land fn = \{0\} \land na = \{} \land \\
r = [0] \land p = [0 \rightarrow \text{Maxprior}] \land w = \{} \land \\
pa = \{} \land tou = \{} \land wto = \{0 \rightarrow \text{false}\} \land \\
ts = [\text{false}] \land wtk = \text{false}
\]

The state invariant defines the meaning of state consistency, e.g. excludes the situation where a given process is simultaneously ready and waiting for a signal. A simplified version of the invariant is shown below.

\[
\text{inv mk-Kernel}(a,fn,na,r,p,w,pa,tou,wto,ts,wtk) \triangleq \\
(a = \text{hd} r) \land \\
\exists \text{waiting-proc} : \text{Process-name-set} \\
\quad \text{waiting-proc} = \{ pr \mid pr : \text{Process-name} \land \\
\quad \exists e : \text{Condition} \land pr \in \text{elems} w(e) \} \\
\land \\
\forall \text{pn} : \text{Process-name} \land \\
\quad (\text{pn} \in \text{elems} r) \Rightarrow (\text{pn} \notin (fn \cup na \cup \text{elems} ts \land \\
\quad \emptyset \cup \text{dom} pa \cup \text{waiting-proc} \land \\
\quad \land \text{pn} \in \text{dom} p \land \text{len} p(\text{pn}) \geq 1)
\]

The invariant asserts that at any time active is the process at the top of the ready queue and that a ready process can not be simultaneously suspended, e.g. waiting for a synchronisation signal or pausing for a given time interval.

4 Specification of kernel services

A process comes to existence by a call to the \text{CREATE} service by the active process.

\[
\text{CREATE}(pr : \text{Priority}) \text{ name : Process-name} \\
\text{pre} \\
\quad fn \neq \{\} \\
\text{post} \\
\exists e : \text{Process-name} \land \\
\quad e \in fn \land \\
\quad fn = fn \setminus \{e\} \land \\
\quad name = e \land \\
\quad na = na \cup \{e\} \land \\
\quad p = \overline{p} \cup \{e \mapsto [pr]\} \land \\
\quad wto = \overline{wto} \cup \{e \mapsto \text{false}\};
\]

It can then be moved into the scheduler algorithm by call to the \text{START} service:

\[
\text{START}(\text{name}:\text{Process-name}) \\
\text{pre} \\
\quad \text{name} \in na \\
\text{post} \\
\quad na = na \setminus \{\text{name}\} \land \\
\quad r = \text{insert}(\overline{T},\text{name},p) \land \\
\quad a = \text{newactive}(\overline{T},\text{name},p);
\]

An active process can \text{PAUSE} for a prescribed amount of real-time clock ticks, by calling the \text{PAUSE} service:

\[
\text{PAUSE}(tm:\text{Time}) \\
\text{pre} \\
\quad tm > 0 \land \\
\quad \text{existnext}(r) \land \\
\quad \text{len}(p(a)) = 1 \\
\text{post} \\
\quad pa = \overline{pa} \cup \{\overline{a} \mapsto \text{tm}\} \land \\
\quad r = \text{tl} \overline{T} \land \\
\quad a = \text{nextready}(\overline{T});
\]
While pausing, the process looses its active/ready attribute and releases its control over CPU.

An active process may SET-TIME-OUT to itself:

\[
\text{SET-TIME-OUT}(tm:\text{Time})
\]

\[
\text{pre}
\]

\[
 tm > 0
\]

\[
\text{post}
\]

\[
\text{tou} = \text{tou}'\{a \leftarrow tm\} \land
\]

\[
\text{uto} = \text{uto}'\{a \leftarrow \text{false}\};
\]

and then may check if the time-out elapsed by calling the WAS-TIME-OUT service:

\[
\text{WAS-TIME-OUT() wt:B}
\]

\[
\text{post}
\]

\[
 wt = \text{uto}(a);
\]

Real time behaviour of the kernel is driven by the TIME-INTERRUPT service activated by the hardware generated clock interrupt.

A set of kernel services is associated with mutual exclusion and conditional signalling in monitors which are assumed as the basic interprocess communication mechanism employed while developing application software. The corresponding services are

\text{MONENTRY, MONEXIT} for mutual exclusion,

\text{WAIT, SIGNAL} for conditional synchronisation.

External specification of the WAIT service is presented below. Specification of other services can be found elsewhere [1].

\[
\text{WAIT}(\text{sig:Condition})
\]

\[
\text{pre}
\]

\[
\text{sig} \in \text{dom w} \land
\]

\[
\text{existnext}(r) \land
\]

\[
\text{len}(\ p(a)) > 1
\]

\[
\text{post}
\]

\[
 w = \overline{w}\{\text{sig} \leftarrow \text{append}(\overline{w}(\text{sig}),\overline{w})\} \land
\]

\[
 r = \text{tl}\overline{r} \land
\]

\[
a = \text{nextready}(\overline{r});
\]

5 The verification methodology

After developing a formal specification of the kernel (external) services (which we called Abstract Specification) this specification was used as the target in verification of the kernel implementation. The implementation was realised in Modula2 (this was the third implementation of the kernel, the previous implementations were in assembler and in C) and have been already in use for 2 years.

The verification process was split into three phases:

- Specification of the realisation – this involved capturing (in the VDM notation) the semantics of the Modula2 program. In effect we obtained a VDM based specification which we called Realisation Specification.

- Relating the two specifications (i.e. the Abstract and the Realisation ones) one to another by means of the \text{retrieval function retr}, which maps states of the Realisation Specification into states of the Abstract Specification.

- Development of a formal argument that Realisation Specification is consistent with the specification of the kernel services. This involved definition of proof obligations for demonstration of state consistency and operation realisation consistency. The state consistency condition looks as follows, the realisation state space is distinguished by a prime (‘):

\[
\forall s' : \text{state}' \bullet \exists s : \text{state} \bullet s = \text{retr}(s')
\]

An example proof obligation for the WAIT service is shown below, the identifiers which correspond to the realisation specification are distinguished by primes (‘):

\[
\forall s', p : \text{Sig} \bullet \text{pre-WAIT}(p, \text{retr}(s'))
\]

\[
\Rightarrow \text{pre-WAIT}'(p, s')
\]

\[
\forall \overline{s'}, s' : \text{state}', p : \text{Sig} \bullet (\text{pre-WAIT}(p, \text{retr}(\overline{s'}))
\]

\[
\land \text{post-WAIT}(p, \overline{s'}, s')
\]

\[
\Rightarrow \text{post-WAIT}'(p, \text{retr}(\overline{s'}), \text{retr}(s'))
\]

6 Deriving the Realisation Specification

The objective of this task is to create a specification which reflects all the properties of the program which are given by its source code. The principal source of information in this task is the program code and the definition of the semantics of the programming language the program is written in. The program documentation is less useful as it could cause the person involved in the verification to specify what
the program is supposed to do (i.e. what the documentation says it does) instead of what it really does.

This task has been fulfilled by capturing the semantics of the Modula2 program using the VDM notation. This involved VDM representation of data types and program instructions and then derivation of the specification of the whole program by iterative application of composition rules. The formalisation of the Modula2 language semantics was done to the extent which was necessary to convert the program implementing the kernel. Those features of the language which were not used were left unspecified.

Capturing the VDM based semantics of Modula2 involved the following steps:

- data structures specification: all program data structures were transformed into VDM state definition:

  state program-state of
  mem : NeNilProcessPtr
  Smem : Sig
  ReadyQueue : ProcessPtr
  ...
  where proc is an abstract type of the following structure:

  proc :
    Name : N
    Next : ProcessPtr
    CondQueue : ProcessPtr
    PauseQueue : ProcessPtr
    PauseTime : Z
    Prior : Z
    MonLev : Z
    OldPrior : Z
    TimeOutCount : Z
    TONext : ProcessPtr
    WasTimeOut : B;

  and similarly SigVar type:

  SigVar :
    First : ProcessPtr
    Last : ProcessPtr;

  In case of pointers we distinguish "valid" pointers' values (pointing to some variable in memory) from nil pointer:

  ProcessPtr = NeNilProcessPtr | nil

- simple statement definition: each programming language statement was specified as an operation, by giving its pre- and post-conditions on the associated state. For example a Modula2 statement:

  \texttt{cc^.CondQueue := Process;}

  is specified in the following way:

  pre \texttt{cc \in dom mem}
  post \texttt{mem = mem'} { cc \mapsto \mu (mem'(cc), }
  \texttt{CondQueue \mapsto Process)}

- composed statements specification: for example the semantics of a sequence

  \texttt{S_1; S_2}

  is described by following rule:

  \{P_1\} S_1 \{P_2 \land R_1\}, \{P_2\} S_2 \{R_2\}
  \{P_1\} S_1; S_2 \{R_1 \lor R_2\}

  where \{P\}S\{R\} stands for

  \texttt{S}
  pre \texttt{P}
  post \texttt{R}

  and for \(\overline{\sigma}, \sigma \in \text{state}\):

  \[ R_1 \lor R_2 (\overline{\sigma}, \sigma) \triangleq \exists \sigma_1 : \text{state} \bullet R_1 (\overline{\sigma}, \sigma_1) \land P(\sigma_1) \land R_2 (\sigma_1, \sigma) \]

  Similar rules were used for other composed statements, for example FOR and WHILE. The general approach followed the classical work on axiomatic definition of a programming language presented in [6].

  The language definition was then used to develop formal specification of the whole program. For example, the WAIT service of the kernel was implemented by the following Modula2 procedure:

  \texttt{PROCEDURE Wait( VAR C : Sig);}
  \texttt{VAR}
  \texttt{Process, cc : ProcessPtr;}
  \texttt{BEGIN}
  \texttt{GetReadyQueue( Process );}
  \texttt{IF c^.First = NIL THEN}
  \texttt{c^.First := Process;
The language is as follows:

\[
\begin{align*}
&\text{wait} \quad (\text{obtained by applying the VDM-based definition of the language}) \\
&\text{wa \ldots t} \\
&\text{next step was to relate it to the specification of the external services of the kernel. It was achieved by defining the retrieval function ret} \\
&\text{then the proof obligations for particular kernel services were fulfilled by formal proving process.}
\end{align*}
\]

A part of the \texttt{ret} function is the \texttt{wait-ret} function which returns the \textit{w} component of the abstract state (waiting processes queues):

\[
\begin{align*}
\text{wait-ret} : & \quad \text{program-state} \\
& \quad \rightarrow \text{Kernel} \\
& \quad \text{if \ } s = \text{nil} \quad \text{then} \\
& \quad \text{else} \\
& \quad \text{sig-list}(s, m) \triangleq \\
& \quad \text{sig-list}(m(s).\text{Name} | p \in \text{dom} s.Smem) \\
\end{align*}
\]

where function \texttt{sig-list} is defined in the following way:

\[
\begin{align*}
\text{sig-list} : & \quad \text{ProcessPtr} \times (\text{NoNilProcessPtr} \rightarrow \text{Process-name}^*) \\
& \quad \rightarrow \text{Process-name} \\
& \quad \text{if \ } s = \text{nil} \\
& \quad \text{then} \\
& \quad \text{else} \\
& \quad \text{sig-list}(m(s).\text{Name} | p \in \text{dom} s.Smem)
\end{align*}
\]

In the above definition we assume the following equivalence between the abstract and the representation types:

\[
\begin{align*}
\text{Name} & = \text{Process-name} \\
\text{Sig} & = \text{Condition}
\end{align*}
\]

Proof obligations for \texttt{WAIT} operation were given in section 5.

The proofs were developed manually, without tool support. Initial attempts to use Mural [4] were unsuccessful, the tool was unable to support analysis of specifications of that size. For handling of the specifications and to perform some static analysis of their correctness the SpecBox [5] tool proved to be useful.

\section{The verification process}

Having developed the realisation specification, the next step was to relate it to the specification of the external services of the kernel. It was achieved by defining the retrieval function \texttt{ret} which maps the states of the realisation specification into states of the external specification. Then the proof obligations for particular kernel services were fulfilled by formal proving process.

\section{Results}

The formal verification process resulted in the following:

- Several discrepancies between the Abstract Specification and the actual implementation were revealed. Those were the result of some earlier
maintenance actions which were not reflected in the formal specification of the kernel external services.

- Three faults were discovered in the implementation which could lead to incorrect behaviour of the processes.
- Few errors and inconsistencies of the external specification of the kernel were discovered.

**Discrepancies between the source code and its specification**

For instance, it has been found that the MONENTRY and MONEXIT operations which ensured mutual exclusion in monitors were implemented in a more restrictive way comparing to what was said in the Abstract Specification. In particular, although the specification allowed to assign priorities to monitors (by calling MONENTRY with a suitable parameter), the implementation re-defined this parameter to a system-wide constant.

**Software errors in the kernel**

1. Although there was no explicit notion of the father-son relationship between processes in the Abstract Specification, the implementation assumed that the memory for a newly created process is allocated from the pool being previously allocated to its creator. Consequently, after the father terminated, the memory used by his sons could be re-allocated again which could lead to tricky and difficult to repeat run-time errors.

2. Termination of a process with an active time-out was not correctly implemented. The Abstract Specification states it clearly: the time-out of a terminating process has to be canceled. However, this was not implemented in the program. Consequently, from the point of view of the time-out handling procedures, the process was still in existence and the memory previously implementing its internal data structures could have been then modified, while being allocated to some other process.

3. The program did not check whether the allocation of conditional variables was accomplished properly. This could have caused problems when system went out of memory.

**Abstract Specification errors**

The following faults of the abstract specification have been detected:

1. The local definition of a variable waiting-proc in the state invariant was incorrect.

2. Wrong operator was used in TERMINATE operation.

3. The state invariant assumes that PAUSE operation must not be called from a monitor. However, this was not consistent with the pre-condition of the PAUSE operation.

4. Similarly, the state invariant restricts that WAIT operation must not be called from outside a monitor, which was not reflected in the pre-condition of the operation.

Although the kernel has been already in use for a long time, the above faults were never discovered. In our opinion, the results obtained from our case study confirm a potential of formal methods to increase confidence in software and to add extra reliability to it. This can be particularly important within the context of critical computer applications. However, one have to be prepared for paying a price for the increased quality. A detailed account of the effort spent for the verification can be found elsewhere [2].

**References**


