

On The Use of VDM++ for Specifying Real-time Systems

Marcel Verhoef (Chess Information Technology, NL)

Erik Gaal (Philips Applied Technologies, NL)



Radboud Universiteit Nijmegen



Contents of this talk

- To Extend or Not to Extend...
- Concurrency in VDM++
- Real-time in VDM++
- Lesson learnt
 - Implementation in VDMTools (VICE version 6.7.27 & 6.7.28)
 - Evaluating the proposed language extensions
- Conclusions & Outlook

To Extend or Not To Extend (1)

- How to bridge the gap between industrial practice and academic state of the art?
 1. OMG's Model Driven Architecture
Platform Specific Model (PSM) \leftrightarrow
Platform Independent Model (PIM)
 2. Integrate different languages
Circus: Z, (timed) CSP, refinement calculus
 3. Extend existing notation
VICE (VDM++ In a Constrained Environment)

To Extend or Not To Extend (2)

- Why are real-time systems so damn difficult?
 1. First focus on functionality, then worry about timing.
 - Performance is “second-class” citizen - ask any SW engineer.
 - Moore’s law doesn’t help, but the bad economy might!
 2. No commonly accepted notation for time
 - Commercial solutions are proprietary and often too restrictive
 - Academic proposals are generic and expressive but hard to apply
 - UML Schedulability, Performance and Time profile still unproven
 3. Current analysis techniques not very effective
 - RMA: can only deal with periodic task sets
 - EDF: optimal scheduling but inefficient implementation
 - TTA: predictable & compositional but heavily over-dimensional

Can VDM++ Be Useful for designing Real-Time Systems?

Let's investigate the VICE extensions!

Concurrency in VDM++

Edsgar Dijkstra's
Dining Philosopher Problem
Revisited

Dining Philosophers

- A number of philosophers meets for dinner
 - Each philosopher brings one fork to the table
 - They think and eat, but ...
 - They need two forks to eat
 - Forks are shared (and scarce) resources
-
- Pick any fork from table if it is free
 - Eat twice then leave table

VDM++ basics - Concurrency

- **Active objects** (philosopher)
 - Instances of classes *with* their own **thread of control**
 - Once started, they do not need external stimuli
- **Passive objects** (table)
 - Instances of classes *without* their own **thread of control**
 - Public operations are always called in the context of *one or more* threads of control of *other active* objects.
 - This requires special integrity protection of state variables against corruption.

VDM++ Basics – Active objects (1)

```
class Philosopher
```

```
instance variables
```

```
theTable : Table;  
turns : nat := 2
```

```
operations
```

```
public Philosopher : Table ==> Philosopher  
Philosopher (pt) == theTable := pt;
```

```
Think: () ==> ()  
Think () == skip;
```

```
Eat: () ==> ()  
Eat () == turns := turns - 1;
```

VDM++ Basics – Active objects (2)

```
thread  
  ( while (turns > 0) do  
    ( Think();  
      theTable.takeFork();  
      theTable.takeFork();  
      Eat();  
      theTable.releaseFork();  
      theTable.releaseFork() );  
    theTable.IamDone() )
```

```
end Philosopher
```

VDM++ Basics – Passive objects (1)

```
class Table
```

```
instance variables
```

```
forks : nat := 0;  
guests : set of Philosopher := {};  
done : nat := 0
```

```
operations
```

```
public Table: nat ==> Table  
Table (noGuests) ==  
  while forks < noGuests do  
    ( guests := guests union {new Philosopher(self)};  
      forks := forks + 1 )  
pre noGuests >= 2;
```

```
public takeFork: () ==> ()  
takeFork () == forks := forks - 1;
```

VDM++ Basics – Passive objects (2)

```
public releaseFork: () ==> ()  
releaseFork () == forks := forks + 1;
```

```
public IamDone: () ==> ()  
IamDone () == done := done + 1;
```

```
wait: () ==> ()  
wait () == skip;
```

```
public LetsEat: () ==> ()  
LetsEat () == ( startlist(guests); wait() )
```

```
sync  
  per takeFork => forks > 0;  
  per wait => done = card guests;  
  mutex(takeFork,releaseFork);  
  mutex(IamDone)
```

```
end Table
```

VDM++ Basics – permission predicates (1)

- 3-tuple ($\#req$, $\#act$, $\#fin$) is maintained for each operation x .
 - $\#req(x)$: how often is operation x called?
 - $\#act(x)$: how often is operation x executed?
 - $\#fin(x)$: how often has operation x finished?
- $\#active(x) = \#act(x) - \#fin(x)$
- $\#waiting(x) = \#req(x) - \#act(x)$

VDM++ Basics – permission predicates (2)

sync

mutex (x, y)

Is Equal To:

sync

per $x \Rightarrow \#active(x) + \#active(y) = 0;$

per $y \Rightarrow \#active(x) + \#active(y) = 0$

```
print new Table(3).LetsEat()
```

TOOL DEMO

VDM++ Basics – Concurrency (summary)

- permission predicates are powerful ...
- ... but very hard to get right
- deadlocks can be detected at *run-time* (during simulation) ...
- but permission predicates cannot be queried from the simulator user-interface and
- it is hard to “see” what goes on inside the system
- there is no support for advanced *compile-time* analysis (i.e. model checking or deductive proof)

Timed Philosophers

How To Introduce The Notion
of Time in VDM++

Timed Philosophers – Duration

Timing Things Once ...

VDM++ Extended – Duration (1)

statement = DURATION (*numeral*) *statement*
| *block statement*
| *dcl statement*
| ... ;

intuition duration: execution of *statement* takes *numeral* time units

VDM++ Extended – Duration (2)

Handling the forks takes time!

```
public takeFork: () ==> ()  
takeFork () ==
```

```
duration (5)  
    forks := forks - 1;
```

```
public releaseFork: () ==> ()  
releaseFork () ==
```

```
duration (5)  
    forks := forks + 1;
```

Eating and thinking too!

```
Think: () ==> ()  
Think () ==
```

```
duration (200)  
    skip;
```

```
Eat: () ==> ()  
Eat () ==
```

```
duration (200)  
    turns := turns - 1;
```

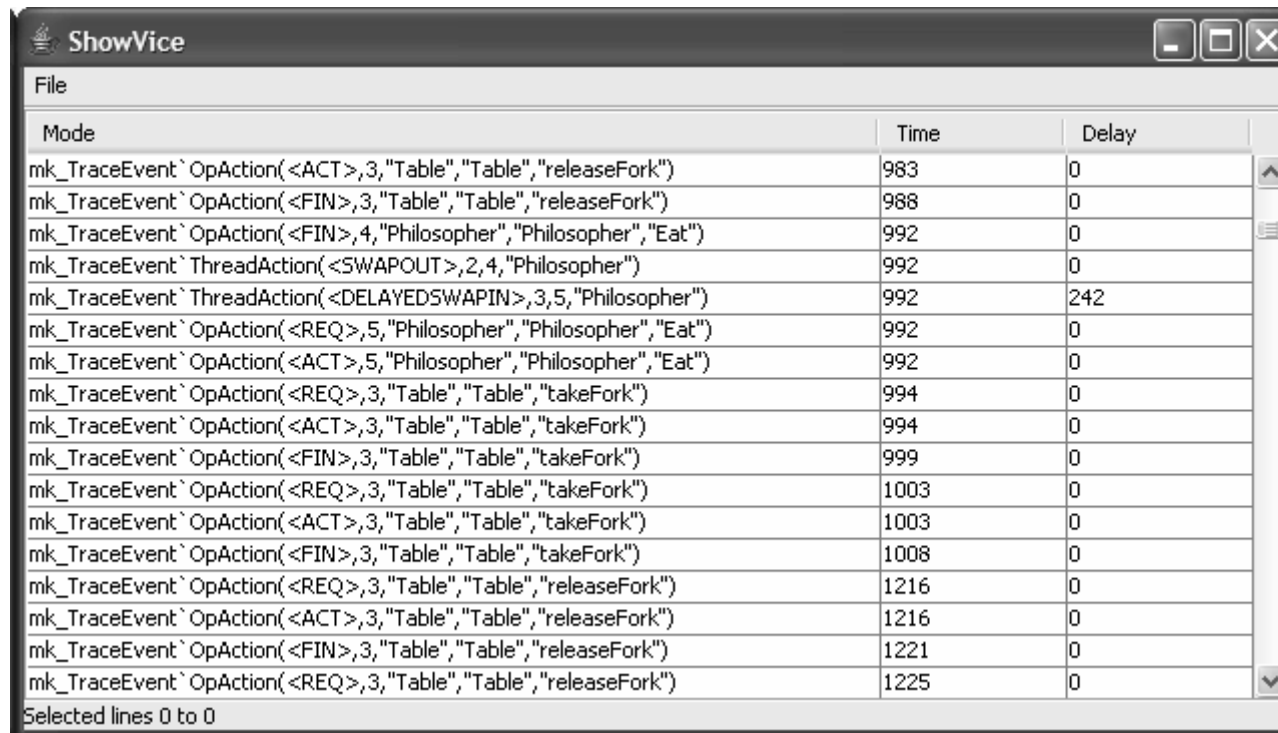
```
print new Table(3).LetsEat()
```

VICE 6.7.27 TOOL DEMO

VDM++ Extended – Trace files

```
req -> Op: Philosopher`Think  Obj: 4  Class: Philosopher  @ 156
act -> Op: Philosopher`Think  Obj: 4  Class: Philosopher  @ 156
fin -> Op: Philosopher`Think  Obj: 4  Class: Philosopher  @ 356
req -> Op: Table`takeFork     Obj: 3  Class: Table    @ 360
act -> Op: Table`takeFork     Obj: 3  Class: Table    @ 360
fin -> Op: Table`takeFork     Obj: 3  Class: Table    @ 365
req -> Op: Table`takeFork     Obj: 3  Class: Table    @ 369
act -> Op: Table`takeFork     Obj: 3  Class: Table    @ 369
fin -> Op: Table`takeFork     Obj: 3  Class: Table    @ 374
req -> Op: Philosopher`Eat    Obj: 4  Class: Philosopher  @ 376
act -> Op: Philosopher`Eat    Obj: 4  Class: Philosopher  @ 376
fin -> Op: Philosopher`Eat    Obj: 4  Class: Philosopher  @ 576
req -> Op: Table`releaseFork  Obj: 3  Class: Table    @ 580
act -> Op: Table`releaseFork  Obj: 3  Class: Table    @ 580
fin -> Op: Table`releaseFork  Obj: 3  Class: Table    @ 585
req -> Op: Table`releaseFork  Obj: 3  Class: Table    @ 589
act -> Op: Table`releaseFork  Obj: 3  Class: Table    @ 589
fin -> Op: Table`releaseFork  Obj: 3  Class: Table    @ 594
```

VDM++ Extended – ShowVice (1)

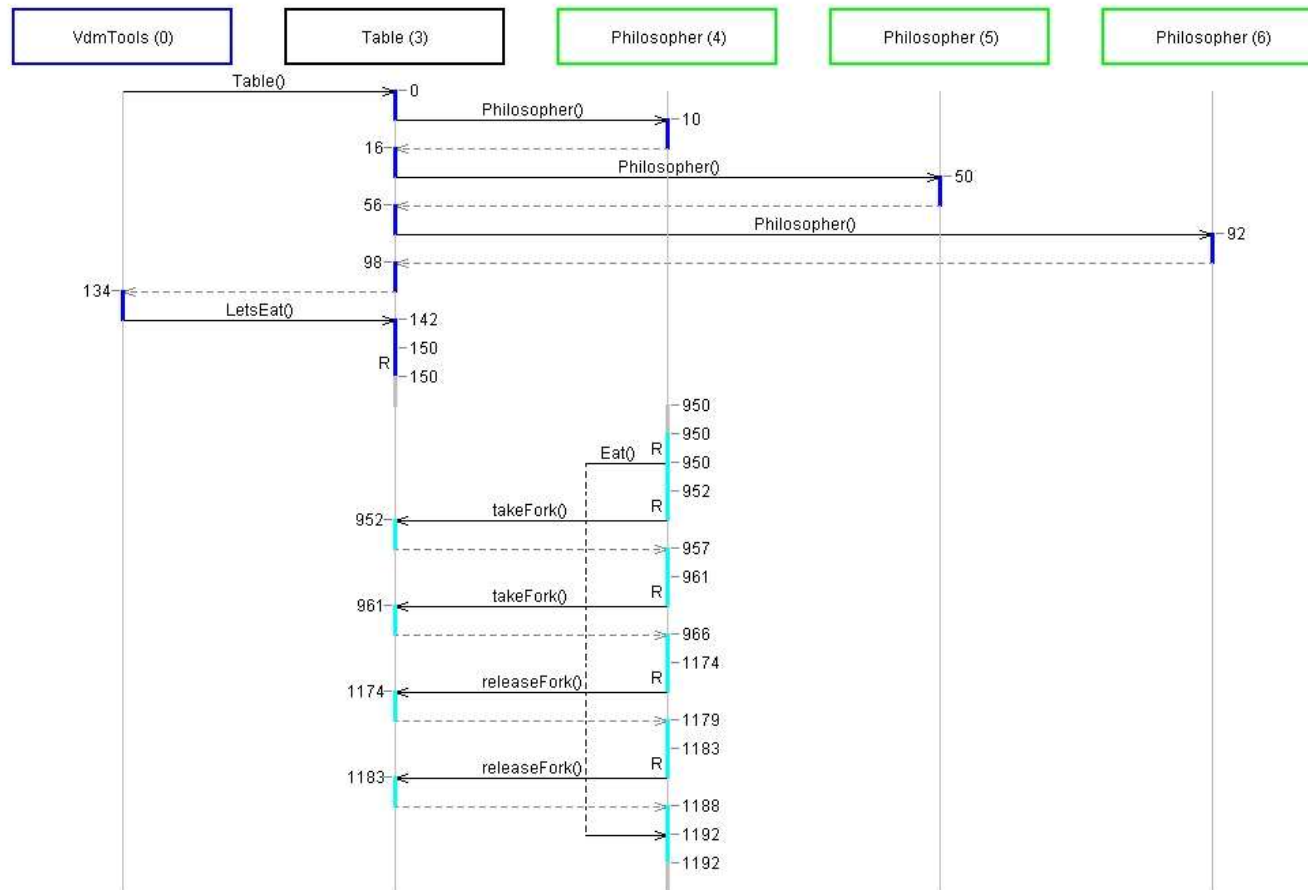


The screenshot shows a window titled "ShowVice" with a menu bar containing "File". Below the menu bar is a table with three columns: "Mode", "Time", and "Delay". The table contains 18 rows of trace events. The "Mode" column contains various actions such as "mk_TraceEvent`OpAction(<ACT>,...)", "mk_TraceEvent`ThreadAction(<SWAPOUT>,...)", and "mk_TraceEvent`ThreadAction(<DELAYEDSWAPIN>,...)". The "Time" column shows values ranging from 983 to 1225. The "Delay" column shows values ranging from 0 to 242. At the bottom of the window, it says "Selected lines 0 to 0".

Mode	Time	Delay
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","releaseFork")	983	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","releaseFork")	988	0
mk_TraceEvent`OpAction(<FIN>,4,"Philosopher","Philosopher","Eat")	992	0
mk_TraceEvent`ThreadAction(<SWAPOUT>,2,4,"Philosopher")	992	0
mk_TraceEvent`ThreadAction(<DELAYEDSWAPIN>,3,5,"Philosopher")	992	242
mk_TraceEvent`OpAction(<REQ>,5,"Philosopher","Philosopher","Eat")	992	0
mk_TraceEvent`OpAction(<ACT>,5,"Philosopher","Philosopher","Eat")	992	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","takeFork")	994	0
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","takeFork")	994	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","takeFork")	999	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","takeFork")	1003	0
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","takeFork")	1003	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","takeFork")	1008	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","releaseFork")	1216	0
mk_TraceEvent`OpAction(<ACT>,3,"Table","Table","releaseFork")	1216	0
mk_TraceEvent`OpAction(<FIN>,3,"Table","Table","releaseFork")	1221	0
mk_TraceEvent`OpAction(<REQ>,3,"Table","Table","releaseFork")	1225	0

Selected lines 0 to 0

VDM++ Extended – ShowVice (2)



VDM++ Extended – Duration (summary)

- duration statements are powerful ...
- ... but not powerful enough!
- execution time is often context dependent
 - upper and lower bound (BCET, WCET) pair
 - a function of a (set of) state variable(s)
- insight is improved by trace file and ShowVice but
 - only off-line analysis (should be integrated into VDMTools)
 - only permission predicates are logged, no instance variables

Duration – ReVICEd (proposal)

statement =

DURATION (*expression*) *statement*

| DURATION (*expression, expression*) *statement*

| *block statement*

| *dcl statement*

| ... ;

Timed Philosophers - Periodic

Timing Things Many Times ...

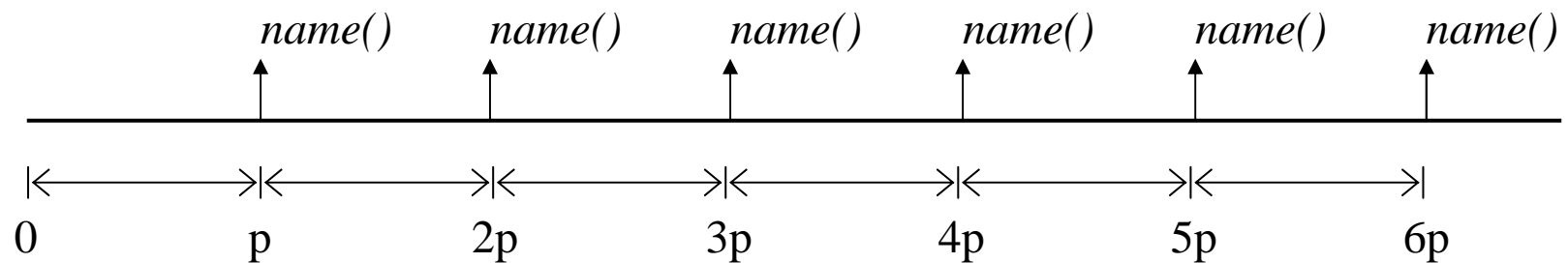
VDM++ Extended – Periodic (1)

thread definition =

THREAD *statement*

| THREAD PERIODIC (*numeral*) (*name*);

intuition periodic: invoke *name* **every** *numeral* time units



VDM++ Extended – Periodic (2)

```
class Philosopher
```

```
operations
```

```
Eat: () ==> ()
```

```
Eat () ==
```

```
  if turns > 0 then
```

```
    ( theTable.takeFork();
```

```
      theTable.takeFork();
```

```
      duration (200) turns := turns - 1;
```

```
      if turns = 0 then theTable.IamDone();
```

```
      theTable.releaseFork();
```

```
      theTable.releaseFork() )
```

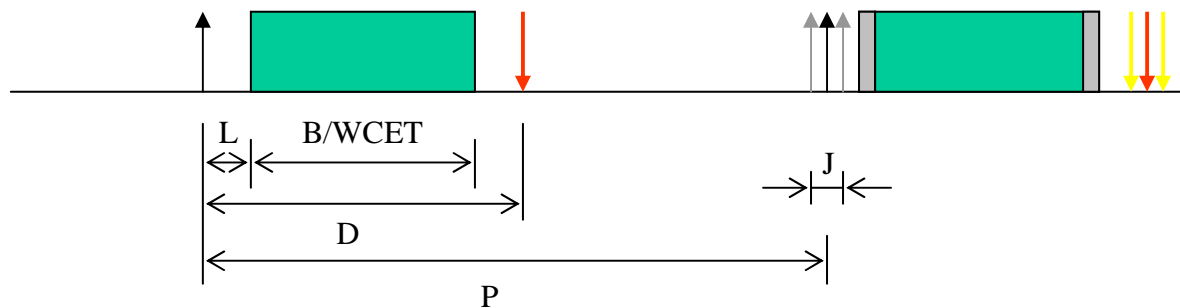
```
thread
```

```
  periodic (800) (Eat);
```

```
end Philosopher
```

VDM++ Extended – Periodic (summary)

- Hidden (implicit) assumptions
 - Periodic task can be **delayed** if another task is still running
 - Periodic task duration (execution time) \ll period
- Not sufficiently expressive, no constructs to
 - Specify maximum allowed task start latency (L)
 - Specify task deadline (D)
 - Specify task start jitter



Periodic – ReVICEd (proposal)

thread definition =

-- normal thread definition
THREAD *statement*

-- pure periodic thread
| THREAD PERIODIC (*numeral*) (*name*);

-- periodic thread with jitter
| THREAD PERIODIC (*numeral* , *numeral*) (*name*);

-- sporadic thread
| THREAD SPORADIC (*numeral*) (*name*);

Intuition sporadic : invoke *name* **at most** every *numeral* time units

Periodic – ReVICEd (prop. cont.)

New permission predicates are needed

- $\#age(n)$ = elapse time since $\#req(n)$
- $\#prev(n)$ = elapse time between
 $\#act(n) - \#act(n-1)$
where $n-1$ denotes the previous
invocation of n

Periodic – ReVICEd (prop. cont.)

```
class Philosopher
```

```
...
```

```
operations
```

```
  Eat: () ==> ()
```

```
  Eat () ==
```

```
    duration (200) turns := turns - 1
```

```
    pre #age(Eat) < 20 and -- task start latency
```

```
      #prev(Eat) < 1200 -- time passed since previous invocation
```

```
    post #age(Eat) < 300 -- task deadline
```

```
thread
```

```
  periodic (800) (Eat);
```

```
end Philosopher
```

Interrupted Philosophers

When It Is Time For a Beer (or Wine)

VDM++ Extended – Interrupt (proposal)

interrupt clause =

INTERRUPT `[` *interrupt definition list* `]` ;

interrupt definition list =

interrupt definition

| *interrupt definition list* `,` *interrupt definition* ;

interrupt definition =

quoted literal "`->" *name* ;

VDM++ Extended – Interrupt (prop. cont.)

statement =

```
SIGNAL `( ` quoted literal ` )`  
| ENABLE `( ` quoted literal | ALL ` )`  
| DISABLE `( ` quoted literal | ALL ` )`  
| ... ;
```

Philosophers catching interrupts

```
class Philosopher
```

```
operations
```

```
  drinkBeer: () ==> ()
```

```
  drinkBeer () == duration (20) skip
```

```
    pre #age(drinkBeer) < 5      -- maximum interrupt latency
```

```
    post #age(drinkBeer) < 50;  -- interrupt deadline
```

```
  drinkWine: () ==> ()
```

```
  drinkWine () == duration (30) skip
```

```
interrupt
```

```
  [ <BEER> -> drinkBeer, <WINE> -> drinkWine ]
```

```
end Philosopher
```

Throwing Interrupts

```
for all guest in set guests do  
  let s in set {<BEER>, <WINE>} in  
    guest.signal(s)
```

Conclusions

- VICE extensions are valuable first step towards specifying real-time systems
- Many deficiencies in both language and tools
- Major hurdle: interpreter only supports multi-threading, no multiprocessing
- Interpreter is not fast enough for large scale industrial application

Outlook

- Extend language as proposed in this talk
- Do *not* improve the interpreter, focus on code generation and use state-of-the-art DES (Rotalumis, SystemC, TrueTime, Ptolemy)
- Use program abstraction techniques to improve compile-time analysis (generate SPIN, FDR, μ CRL, UPPAAL, TIMES, PVS models instead)

Thank you for your attention!

Questions?

Marcel.Verhoef@chess.nl