Métodos Formais em Engenharia de Software

Ana Paiva

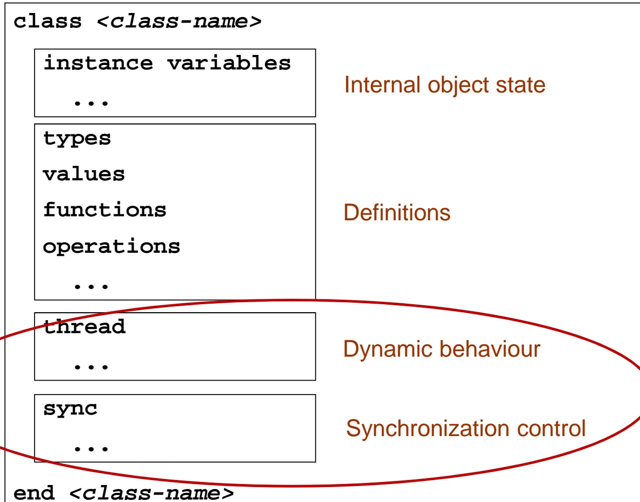
apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

Agenda

- VDMTools
- Características da linguagem VDM++
  - Classes; Variáveis de instância; Operações; Funções (polimórficas, de ordem superior, lambda, …); Tipos; Operadores; Expressões
  - Design-by-contact:
    - Definição de invariantes; pré e pós-condições
    - Ligação do VDM++ ao UML
- Consistência da especificação: obrigações de prova e teste
- Exemplo da Vending Machine
- Concorrência em VDM++

VDM++ Class Outline



Concurrency in VDM++

- Why use concurrency in specifications?
  - The real world is highly concurrent.
    Consequently models of the world are likely to be concurrent too.
  - For efficiency reasons in a multi processor environment.
- Objects can be:
  - **Passive**: Change state on request only, i.e., as a consequence of an operation invocation.
  - **Active**: Can change their internal state spontaneously without any influence from other objects. Active objects have their own thread of control.

Passive Objects

- Respond to requests (operation invocations) from active objects (clients).
- Supply an interface (a set of operations) for their clients.
- No thread.
- Can serve several clients.

Concorrência e sincronização em VDM++

- Concorrência: através da definição de objetos ativos que podem ser donos de "threads"
  - Classes de objetos ativos têm secção "thread" onde se especifica o comportamento do thread
  - Instrução "start" inicia "thread" num objeto previamente criado
  - Dois tipos de "threads": simples e periódicos

Concorrência e sincronização em VDM++

- Sincronização: através de restrições de sincronização no acesso a objetos partilhados (tipicamente passivos)
  - Restrições de sincronização são definidas de forma declarativa
  - Permitem limitar concorrência entre objetos ativos/threads
  - Restrições são indicadas na secção "sync" da definição da classe
  - Dois tipos de restrições/predicados: de permissão e de exclusão mútua

    sync per operation_name => guard-condition

    mutex (op1, op2)
  - Restrições são herdadas por subclasses

Threads simples (ou procedimentais)

- thread *statement(s)*
  - Secção da definição da classe que indica a instrução (normalmente uma operação) ou sequência de instruções a realizar pelo *thread*
  - O *thread* morre quando se completa a execução dessa(s) instrução(ões)
- start(*objRef*)
  - Instrução usada para iniciar um *thread* sobre o objeto indicado
  - O *thread* não é iniciado ao criar o objeto para permitir inicializações
  - Chamado de novo (mesmo sem acabar anterior), inicia novo *thread*
- startlist(*objRefSet*)
  - Instrução usada para iniciar um conjunto de *threads*
- threadid
  - Número natural que identifica univocamente o *thread* corrente

Threads periódicos

- thread periodic (*timeinterval*) (*opname*)
  - Executa repetidamente a operação de acordo com o intervalo de tempo especificado (em "unidades de tempo do sistema")
  - A operação deve executar em tempo inferior ao intervalo de tempo
  - Não são suportados pelas VDMTools Light

```
-- timer that periodically increments its clock, at every 1000 system time units
class Timer
 instance variables
   private curTime : nat := 0;
 operations
   private IncTime() == curTime := curTime + 1;
   public GetTime() res: nat == return curTime;
 thread
    periodic(1000)(IncTime)
 end Timer
```


Permission Guards

- Synchronization for objects is specified using VDM++'s **sync** clause:

```
sync
  per <operation-name> => <condition>
```

- The **per** clause is known as a permission guard. <condition> is a Boolean expression, which involves the attributes of the class, that must hold in order for operation-name to be invoked.
- Ex.: Permission guards reflecting the bounding of the buffer :

```
sync
  per GetItem => len buf > 0
  per PutItem => len buf < size
```

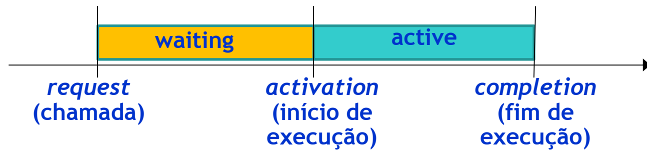

Predicados de permissão (1)

- per operation-name => guard-condition
  - Especifica condição a verificar para a operação poder ser executada
  - Se não se verificar no momento da chamada da operação, esta fica em espera
- Condição de guarda pode usar valores de variáveis de instância, bem como valores de contadores de execução de operações (ver a seguir)
- Condição de guarda é diferente de uma pré-condição
  - Não satisfação de pré-condição é um erro
  - Não satisfação de condição de guarda apenas coloca a chamada em espera

Predicados de permissão (2)

- Interpretador deteta e avisa eventuais situações de "deadlock"
- Só se pode especificar um predicado de permissão por operação
- Regras para reavaliação das condições de guarda:
  - Ocorre quando termina a execução duma operação (sobre mesmo objeto)
  - Teste da condição e (potencial) ativação da operação realizados atomicamente
  - Não está definido qual é o objeto cuja expressão de guarda é reavaliada 1º

Contadores de execução de operações



Expressão	Descrição
#act(*op- name*)	Nº de vezes que a operação foi ativada (iniciou execução) sobre este objeto.
#fin(*op- name*)	Nº de vezes que a operação foi concluída (terminou execução) sobre este objeto.
#active(*op- name*)	Nº de chamadas da operação que estão presentemente ativas sobre este objeto. #active(op-name) = #act(op-name) - #fin(op-name)
#req(*op- name*)	Nº de chamadas da operação sobre este objeto.
#waiting(*op- name*)	Nº de chamadas que estão presentemente em espera sobre este objeto. #waiting(op-name) = #req(op-name) - #act(op-name)

Predicados de exclusão mútua (mutex)

- mutex(*op-name1*, *op-name2*, …)
  - Operações não podem executar em simultâneo (sobre o mesmo objecto)
- mutex(all)
  - "all" refere-se a todas as operações definidas na classe e superclasses
- A mesma operação pode aparecer em múltiplos predicados mutex (e num predicado de permissão)
- Predicados mutex são implicitamente traduzidos para predicados de permissão

```
mutex(opA, opB);
mutex(opB, opC, opD);
per opD => someVariable > 42;
```

```
per opA => #active(opA) + #active(opB) = 0;
per opB => #active(opA) + #active(opB) = 0 and
           #active(opB) + #active(opC) + #active(opD) = 0;
per opC => #active(opB) + #active(opC) + #active(opD) = 0;
per opD => #active(opB) + #active(opC) + #active(opD) = 0
           and someVariable > 42;
```


Exemplo

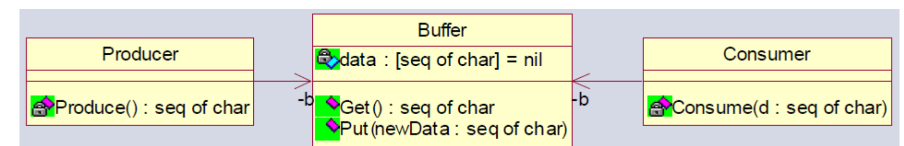
```
class Worker
 operations
   public doit() == (            -- standard VDM++ IO library
     dcl io : IO := new IO();
     dcl rc : bool;
     for i = 1 to 40 do
        rc := io.fwriteval[nat * nat]("out.txt",
                   mk_(threadid, i), <append>)
   );
   public wait_done() == skip;
   public static main() == (
     dcl w1 : Worker := new Worker();
     dcl w2 : Worker := new Worker();
     start(w1);  start(w2);
     w1.wait_done(); w2.wait_done()
   );      Com w1 outra vez também resultava
 thread doit()      wait_done só é invocada quando...
 sync per wait_done => #fin(doit) > #act(wait_done)
end Worker
```

out.txt

mk_(2, 1)	mk_(3, 9)	mk_(2, 37)
mk_(2, 2)	mk_(3, 10)	mk_(2, 38)
mk_(2, 3)	mk_(3, 11)	mk_(2, 39)
mk_(2, 4)	mk_(3, 12)	mk_(2, 40)
mk_(2, 5)	mk_(3, 13)	mk_(3, 21)
mk_(2, 6)	mk_(3, 14)	mk_(3, 22)
mk_(2, 7)	mk_(3, 15)	mk_(3, 23)
mk_(2, 8)	mk_(3, 16)	mk_(3, 24)
mk_(2, 9)	mk_(3, 17)	mk_(3, 25)
mk_(2, 10)	mk_(3, 18)	mk_(3, 26)
mk_(2, 11)	mk_(3, 19)	mk_(3, 27)
mk_(2, 12)	mk_(3, 20)	mk_(3, 28)
mk_(2, 13)	mk_(2, 21)	mk_(3, 29)
mk_(2, 14)	mk_(2, 22)	mk_(3, 30)
mk_(2, 15)	mk_(2, 23)	mk_(3, 31)
mk_(2, 16)	mk_(2, 24)	mk_(3, 32)
mk_(2, 17)	mk_(2, 25)	mk_(3, 33)
mk_(2, 18)	mk_(2, 26)	mk_(3, 34)
mk_(2, 19)	mk_(2, 27)	mk_(3, 35)
mk_(2, 20)	mk_(2, 28)	mk_(3, 36)
mk_(3, 1)	mk_(2, 29)	mk_(3, 37)
mk_(3, 2)	mk_(2, 30)	mk_(3, 38)
mk_(3, 3)	mk_(2, 31)	mk_(3, 39)
mk_(3, 4)	mk_(2, 32)	mk_(3, 40)
mk_(3, 5)	mk_(2, 33)	
mk_(3, 6)	mk_(2, 34)	
mk_(3, 7)	mk_(2, 35)	
mk_(3, 8)	mk_(2, 36)	

Example

- Concurrent threads must be synchronized
- Illustrate with a producer-consumer example
- Produce before consumption …
- Assume a single producer and a single consumer
- Producer has a thread which repeatedly places data in a buffer
- Consumer has a thread which repeatedly fetches data from a buffer



The Buffer class

```
class Buffer
 instance variables
     data : [seq of char] := nil
 operations
     public Put: seq of char ==> ()
     Put(newData) ==
      data := newData;

     public Get: () ==> seq of char
     Get() ==
      let oldData = data
      in
           ( data := nil;
            return oldData
           )
end Buffer
```


The producer class

```
class Producer
 instance variables
     b : Buffer
 operations
     Produce: () ==> seq of char
     Produce() == ...
 thread
     while true do
          b.Put(Produce())
end Producer
```


The consumer class

```
class Consumer
 instance variables
     b : Buffer
 operations
     Consume: seq of char ==> ()
     Consume(d) == ...
 thread
     while true do
          Consume(b.Get())
end Consumer
```


The Buffer Synchronized

Assuming the buffer does not lose data, there are two requirements:

1. It should only be possible to *get* data, when the producer has placed data in the buffer.
2. It should only be possible to *put* data when the consumer has fetched data from the buffer.

The following permission predicates could model these requirements:

```
per Put => data = nil
per Get => data <> nil
```

The Buffer Synchronized

- The previous predicates could also have been written using history counters:
- For example

  per Get => #fin(Put) - #fin(Get) = 1

Mutual Exclusion

- Another problem could arise with the buffer: what if the producer produces and the consumer consumes at the same time?
- The result could be non-deterministic and/or counterintuitive. VDM++ provides the keyword mutex

  mutex(Put, Get)
- Shorthand for

  per Put => #active(Get) = 0

  per Get => #active(Put) = 0

* A propósito: biblioteca padrão de IO

- Incluir ficheiro $TOOLBOXHOME/stdlib/io.vpp no projecto
- writeval[tipo](valor)
  - Função que escreve o valor do tipo indicado, em ASCII, no standard output
  - Exemplo: writeval[nat](20)
- fwriteval[tipo](ficheiro, valor, modo)
  - Função que escreve o valor do tipo indicado, em ASCII, no standard output
  - O modo pode ser <append> (acrescentar) ou <start> (criar)
  - Exemplo: fwriteval[nat]("output.txt", 20, <append>)
- echo(texto)
  - Operação que escreve o texto, possivelmente com sequências de escape, no standard output.
  - Exemplo: echo("ola\n")
- fecho(ficheiro, texto, [modo])
  - Idem, em ficheiro
- ferror()
  - Todas as funções/operações anteriores devolvem false em caso de erro. Esta operação devolve (e limpa) a string com a mensagem de erro correspondente

Bibliografia adicional

- *Applying Formal Specification in Industry*. P.G. Larsen, J. Fitzgerald and T. Brookes. Published in "IEEE Software" vol. 13, no. 3, May 1996
- *A Lightweight Approach to Formal Methods* S.Agerholm and P.G. Larsen. In Proceedings of the International Workshop on Current Trends in Applied Formal Methods, Boppard, Germany, Springer-Verlag, October 1998.
- *Applications of VDM in Banknote Processing* P. Smith and P.G. Larsen. + *Application of VDM-SL to the Development of the SPOT4 Programming Messages Generator*, A. Puccetti and J.Y. Tixadou + *Formal Specification of an Auctioning System Using VDM++ and UML, M.Verhoef et. al.*

  Published at the First VDM Workshop: VDM in Practice with the FM'99 Symposium, Toulouse, France, September 1999.

Exemplos de VDM++ na web

- http://www.vdmportal.org/twiki/bin/view/Main/VDMPPexamples
- http://www.overturetool.org/?q=node/13
- http://www.vdmportal.org/twiki/pub/Main/VDMPPexamples/cashdispenser_a4.pdf