

Software Modelling

Ana Paiva

apaiva@fe.up.pt www.fe.up.pt/~apaiva

Model your system

- Build the model to meet user requirements
- Choose the right level of abstraction
- Choose a notation for modelling
- Once the model is written, ensure that it is accurate (validate and verify your model)

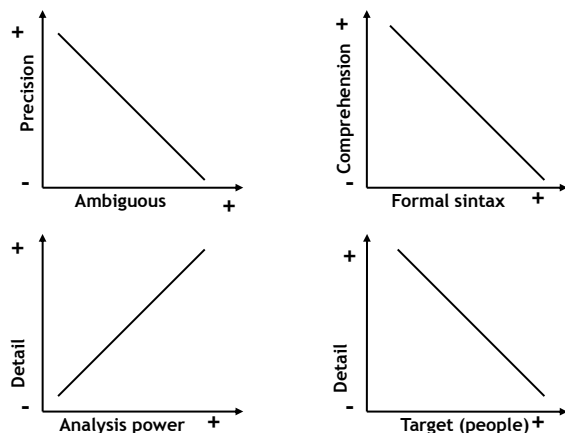
Models in the software process

- Forward engineering
 - To support the development (e.g., model-driven development)
 - Can be used in automatic code generation approaches
 - Used to derive test cases
 - ...
- Reverse engineering
 - Automatically/manually constructed from an existing application
 - Effort to understand legacy systems
 - May be needed to change systems platform
 - ...

Modelling Notations

- Graphical /textual
 - *Graphical* modelling languages use a diagram techniques with named symbols that represent concepts and lines that connect the symbols and that represent relationships and various other graphical annotation to represent constraints.
 - *Textual* modelling languages typically use standardized keywords accompanied by parameters to make computer-interpretable expressions.
- Executable / non-executable
 - Executable modelling languages are intended to amplify the productivity of skilled programmers
- Formal / semi-formal / informal
 - Formal notations are rigorous and unambiguous. Formal models are particular kind of mathematically-based techniques for the specification, development and verification of software and hardware systems
- Static / dynamic
 - A static model does not account for the element of time, while a dynamic model does
- ...

Modelling notations



Graphical notations

- UML (Unified Modelling Language) - is a general purpose language which supports thirteen different diagram techniques. UML 2.0, the current version
- Petri nets
- Statecharts
- Finite state machines
- Flowcharts
- ...

Formal notations

- Formal notations describe *what* the system should do, not (necessarily) *how* the system should do it
 - Model-based (or Pre/Post). E.g., VDM, Z, Spec#.
 - Behaviour-based (or transition-based). E.g., FSM, Petri nets.
 - Behaviour-based (or history-based). E.g., CSP, MSC.
 - Property-based (or functional-based). E.g., OBJ.
 - Hybrid approaches. E.g., RAISE.
 - ...
 - UML with OCL?
 - pre/post, Set, OrderedSet, Bag, Sequence, and associations among classes

Formal specification characteristics

- A expressão “**especificação formal**” pode ser entendida de diferentes formas [Rouff 1996]. No sentido comum, uma especificação é formal se for **escrita, comunicável, matemática, precisa, não ambígua** e sirva de **suporte à análise e permita raciocínio e predição**. Os matemáticos reforçam que uma especificação só será formal se suportar raciocínio e predição enquanto que o grupo ligado às questões humanas diz que tem que ser comunicável. Os engenheiros de *software* situam-se num campo intermédio concordando que a especificação tem que ser precisa, não ambígua e deve suportar análise. Todos concordam porém que uma especificação formal tem que ser não ambígua e comunicável/escrita.

Especificação versus implementação

- **Especificação** - “o quê” (problema, objectivo)
 - Domínio das linguagens e notações de especificação
 - Especificação **informal** - em linguagem natural
 - Exemplos: Comentários no código, documentos de requisitos
 - Principal problema: ambiguidade
 - Especificação **formal** - em linguagem formal
 - Principais vantagens: remover ambiguidade, suportar a verificação formal
 - Especificação **semi-formal** - caso intermédio
 - Exemplo: especificações baseadas em diagramas UML (com OCL?)
- **Implementação** - “como” (solução, algoritmo, programa)
 - Domínio das linguagens de programação
 - A implementação pode ser obtida por refinamentos sucessivos
 - Conformidade da implementação com a especificação pode ser verificada formalmente quando existe uma especificação formal

Especificações versus programas

- A linguagem de especificação não está restrita a expressar apenas **funções computáveis**
- As especificações não têm que ser **executáveis**
- As especificações podem ser **implícitas** (descrevem o resultado correcto sem descrever como atingir esse resultado)
 - Exemplo:

```
Seq<int> Sort (Seq<int> arg)
ensures Forall {i,j in result.Indices, i<j;
           result[i]<=result[j]};
{}

```
- Os programas devem ser **executáveis**

Especificações executáveis

- É possível enriquecer os modelos implícitos com especificações do corpo algorítmico das operações
 - Exemplo:

```
Seq<int> Sort (Seq<int> arg)
ensures Forall {i,j in result.Indices, i<j;
           result[i]<=result[j]};
{ // especificação de um algoritmo de ordenação
  // por exemplo, quick sort; merge sort; etc. }

```
- Obtém-se um **modelo executável**, que serve como protótipo executável do sistema, permitindo testar e validar precocemente os requisitos e opções de design

Model-based notation VDM-SL / VDM++

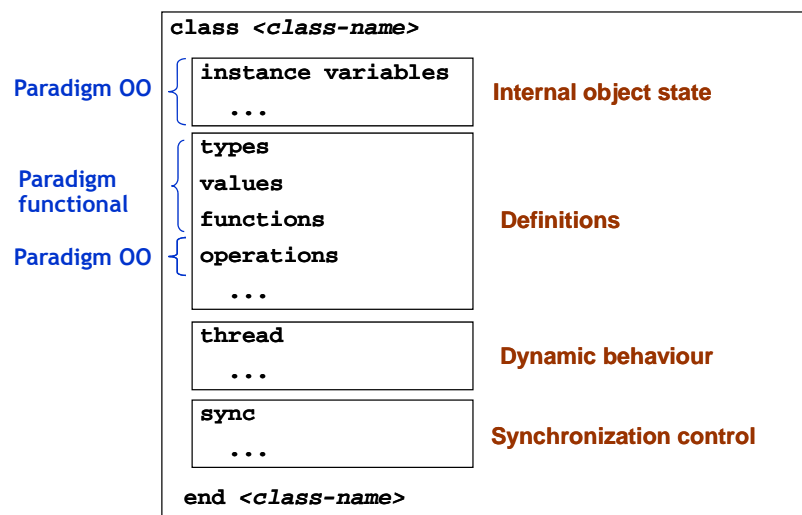
Model-based notations

- Examples
 - VDM-SL, VDM++; Z, Object Z; OCL (Object Constraint Language); Spec# - C# with contracts; JML - Java with contracts
- Advantages
 - Good for modelling state
 - More closed to programming languages
 - Design by contract (DbC)
 - ...
- Disadvantages
 - Not so appropriate for modelling concurrency
 - Not abstract enough
 - ...

VDM-SL / VDM++

- VDM (The Vienna Development Method) is one of the model based Formal Method which was developed in the middle of 1970s at the institution of IBM in Vienna. The formal specification language, *VDM-SL*, became the ISO standard language in 1996. (ISO/IEC 13817-1)
- VDMTools supports the analysis of precise models of computing systems which are expressed either in the VDM-SL language or in the object-oriented formal specification language *VDM++*.

VDM++ specification structure



VDM++: example

```

class Stack
instance variables
    stack : seq of int := [];
    maxLength : int := MAX;
    inv len stack < MAX
operations
    Stack : () ==> ()
    Stack () == stack := []
    post stack = [];
    Push : int ==> ()
    Push(i) == stack := [i] ^ stack
    post stack = [i] ^ ~stack;
    Pop : () ==> ()
    Pop() == stack := tl stack
    pre stack <> []
    post stack = tl ~stack;
    Top : () ==> int
    Top() == return (hd stack)
    pre stack <> []
    post RESULT = hd stack and stack = ~stack;
end Stack
    
```

Vending Machine: example

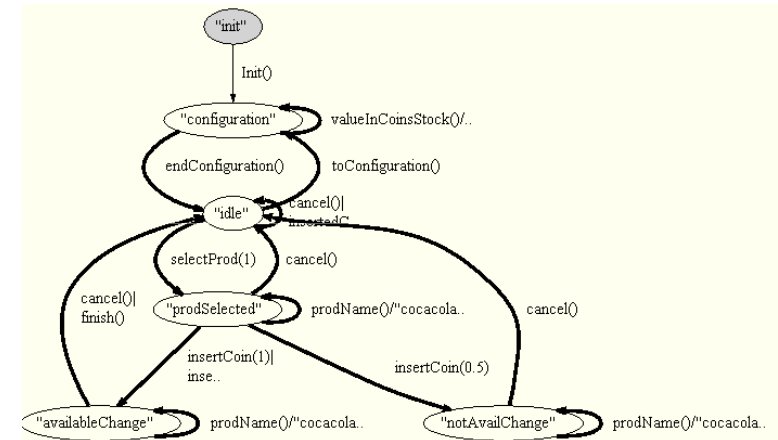
- Build a VDM++ model of a vending machine for products like drinks, snacks, cookies, etc. This machine accepts coins of 0.05; 0.1; 0.2; 0.5 and 1 euro
- There is a stock of coins inside the vending machine which is used to give the due change to clients who may need it
- The vending machine has also a stock of products and each one has its price
- The Product Vending Machine provides different services in two different possible states: in configuration or waiting for user interaction

While in configuration: it is possible to update the stock of products and coins

While waiting:

- The vending machine should show the products available to the clients
- The clients should select the product they want and then insert euro coins to pay for it
- The vending machine should give change every time it is possible (there are coins in stock for that purpose); otherwise it should give back all the money introduced by the client without selling the product

Vending Machine



Vending Machine - constants e types

values

```
public Capacity: real = 15;
```

types

```
public String = seq of char;
public Coins = nat
  inv c == c in set {100, 50, 20, 10, 5};
public Boxes = nat
  inv b == b in set {1,...,20};
```

```
public Products :: name: String
  quantity : nat
  price : nat1;
```

```
public ProductsInBoxes = map Boxes to [Products]
  inv pb ==
```

```
  forall b in set dom pb & pb(b) = nil or
  pb(b)<>nil => pb(b).quantity <= Capacity;
```

```
public State = <init> | <configuration> | <idle> | <prodSelected> |
  <availableChange> | <notAvailChange>;
```

instance variables

```
public stockProd: ProductsInBoxes;
public stockCoins: map Coins to nat;
public stateMachine: State := <init>;
public prodSelected: [Boxes] := nil;
public insertedCoins: seq of Coins := [];
public coinsTroco: seq of Coins := [];
```

Vending Machine - operações/funções

operations

Constructor

```
public VendingMachine : () ==> VendingMachine
VendingMachine () ==
(
  stateMachine := <configuration>;
  stockProd := { |-> };
  stockCoins := { |-> };
)
pre stateMachine = <init>;
```

Puts (replaces) a certain quantity of a product in a box. Machine should be in configuration state.

```
public SetStockProducts (num: Boxes, name: String, price: nat1, quant: nat1) ==
  stockProd := stockProd ++ {num |-> mk_Products(name, price, quant)}
pre stateMachine = <configuration> and
  quant <= Capacity and price mod 5 = 0;
```

Vending Machine - operations/functions

Updates the stock of coins. Machine in configuration state.

```
public SetStockCoins(novoStockCoins: map Coins to nat) ==
  stockCoins := stockCoins ++ novoStockCoins
  pre stateMachine = <configuration>;
```

Reads the quantity in stock of a product by its number.

```
public GetStockProducts(number: Boxes) res : nat1 ==
  return stockProd(number).quantity
  pre stateMachine = <configuration> and number in set dom stockProd;
```

Reads the price of a product by its number.

```
public GetPriceProduct(number: Boxes) res : nat1 ==
  return stockProd(number).price
  pre stateMachine = <configuration> and number in set dom stockProd;
```

End of configuration state.

```
public EndConfiguration() ==
  stateMachine := <idle>
  pre stateMachine = <configuration>;
```

[Pascoal Faria (FEUP)]

Vending Machine - operations/functions

Selects a product by its number. Then insert coins.

```
public SelectProduct(number : Boxes) ==
  prodSelected := number
  pre stateMachine = <idle> and number in set dom stockProd and stockProd(number) <> nil
  and stockProd(number) <> nil => stockProd(number).quantity > 0;
```

To know if enough money was already inserted.

```
public InsertedCoinsValue() res : nat ==
  (
    dcl sum:nat := 0;
    for all e in set inds insertedCoins do sum:=sum+insertedCoins(e); return sum;
  )
  pre prodSelected <> nil;
```

Insert a coin. Product should be already selected and enough money should not yet been introduced.

```
public InsertCoin(c: Coins) ==
  insertedCoins := insertedCoins ^ [c]
  pre prodSelected <> nil and InsertedCoinsValue() < stockProd(prodSelected).price;
```

[Pascoal Faria (FEUP)]

Vending Machine - operations/functions

Reads the name of the selected product.

```
public GetNomeProdSel() res : String == return stockProd(prodSelected).name
  pre prodSelected <> nil;
```

Total value of the set of coins

```
public Sum(troco: seq of Coins) res : nat ==
  (
    dcl sum:nat := 0;
    for all e in seq troco do sum := sum + e*troco(e);
    return sum;
  );
```

Cancel operations

```
public Cancelar() ==
  ( prodSelected := nil; insertedCoins := []; coinsTroco := [] )
  pre prodSelected <> nil;
```

Simulates the user getting the selected product

```
public RecolheProduto() == (
  stockProd(prodSelected).quantity := stockProd(prodSelected).quantity - 1;
  if (stockProd(prodSelected).quantity=0) then stockProd := {prodSelected} <-: stockProd ;
  coinsTroco := [];
)
  pre prodSelected in set dom stockProd;
```

[Pascoal Faria (FEUP)]

Vending Machine - operations/functions

If possible gives change to the cliente

```
public GiveChange() res: seq of Coins ==
  (
    dcl sortedCoins: seq of Coins := [100,50,20,10,5];
    dcl troco : nat := InsertedCoinsValue() - stockProd(prodSelected).price;
    coinsTroco := [];
    while (sortedCoins <> []) do
      (
        if (hd sortedCoins in set dom stockCoins and stockCoins(hd sortedCoins) > 0
          and hd sortedCoins < troco)
        then (
          coinsTroco := coinsTroco ^ [hd sortedCoins];
          stockCoins(hd sortedCoins) := stockCoins(hd sortedCoins)-1;
          troco := troco - hd sortedCoins;
        ) else sortedCoins := tl sortedCoins
      );
    if (Sum(coinsTroco) <> troco) then
      (
        for all e in coinsTroco stockCoins(hd sortedCoins) := stockCoins(hd sortedCoins)+1;
        coinsTroco := [];
      ) return coinsTroco;
  );
  pre InsertedCoinsValue() > stockProd[prodSelected].price;
```

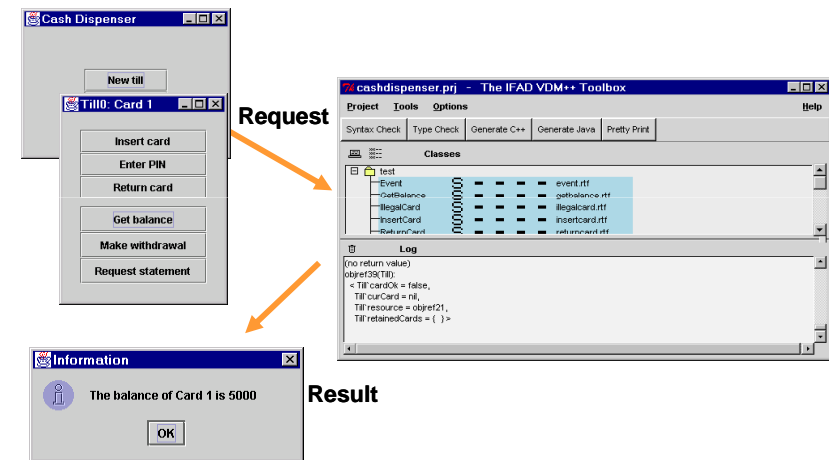
[Pascoal Faria (FEUP)]

VDMTools

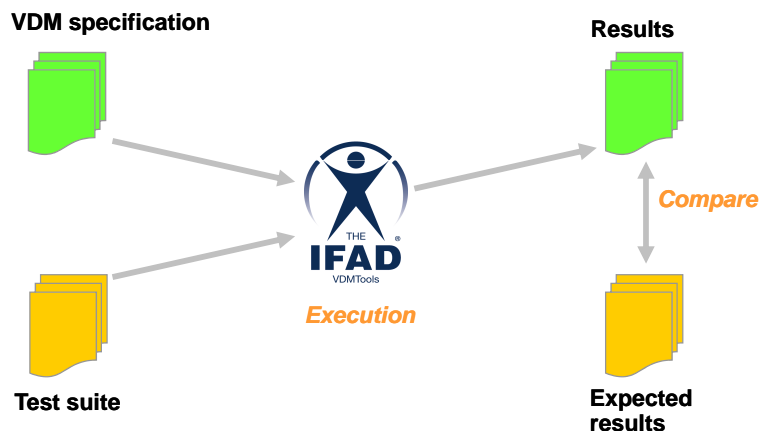
The VDMTools supports software development based on the specification which is written by formal specification language, VDM-SL or VDM++. VDMTools provides the various features which are described below.

- Syntax checker
- Type checker
- Integrity Examiner
- Interpreter and debugger
- Test coverage statistics tool
- Rose - VDM++ link
- Pretty Printer
- VDM++ to C++ code generator (Optional)
- VDM++ to Java code generator (Optional)
- Java to VDM++ generator (Optional)
- CORBA Compliant API (Optional)

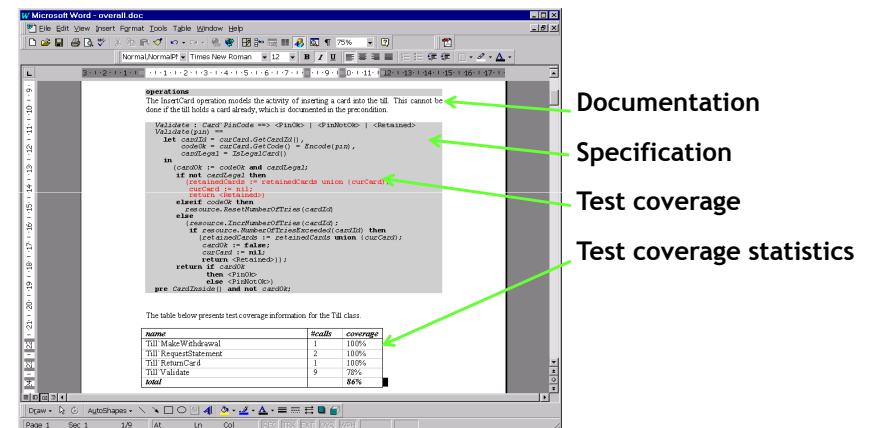
Toolbox API



Validation with VDMTools®



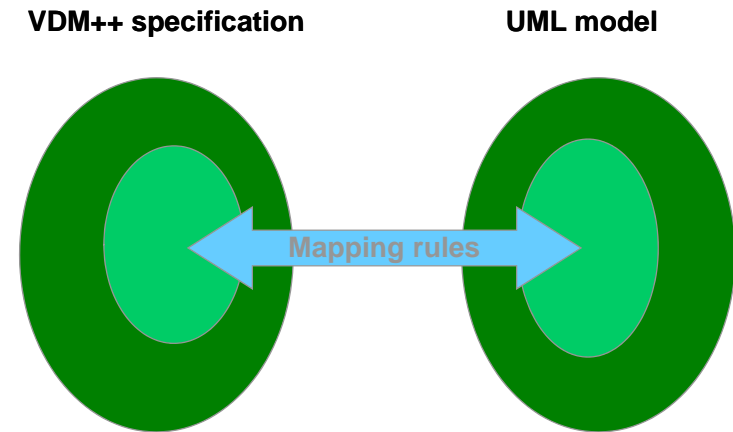
Documentation in MS Word/RTF



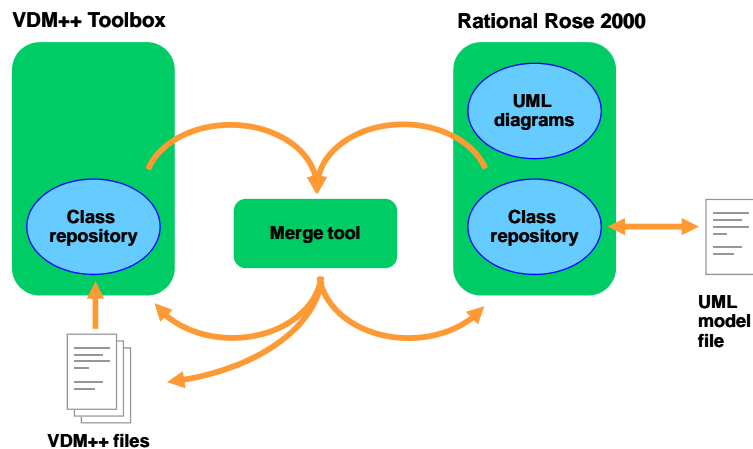
The Rose-VDM++ link

- Supports round-trip engineering with Rational Rose
- Offers the complementary benefits of the graphical notation UML and the textual formal notation VDM++
- Massive use of UML expected worldwide!

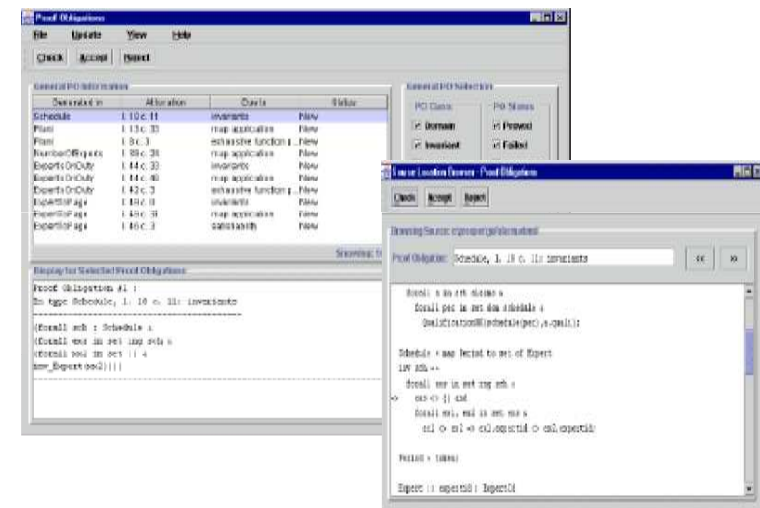
Integration principle



Architecture of link



Proof obligation generator



Model validation Proof obligations

Validação do modelo

- **Validação** é o processo de aumentar a confiança de que o modelo é uma representação fiel do sistema em análise. Há dois aspectos a considerar:
 1. Verificação da consistência interna do modelo.
 2. Verificar que o modelo descreve o comportamento esperado do sistema em análise.

Propriedades de integridade formal

- **Satisfabilidade** (existência de solução)
 - \exists combinação de valores finais de variáveis de instância e valor de retorno satisfazendo a pós-condição, \forall combinações de valores iniciais das variáveis de instância e argumentos obedecendo aos invariantes e à pré-condição
- **Determinismo** (unicidade de solução)
 - Sempre que os requisitos assim o indiquem, escrever uma pós-condição determinística (que admite uma única solução)
 - Mas, por exemplo, num problema de optimização, a pós-condição pode restringir as soluções admissíveis, sem chegar a impor uma solução única
- **Preservação de invariantes**
 - Se valores iniciais de variáveis de instância e argumentos obedecerem aos invariantes e pré-condição, a pós-condição garante invariantes no final
 - Depois de garantir que todas as operações respeitam os invariantes, pode-se desactivar a sua verificação (mais pesada que verificação incremental de pré/pós-condições)
- **Protecção de operadores parciais**
 - Inclusão de pré-condições que definam o domínio de valores em que os operadores podem ser chamados.

Consistência interna: obrigações de prova

- A colecção de todas as verificações a efectuar sobre um modelo VDM são designadas por *proof obligations*. Uma obrigação de prova é uma expressão lógica que se deve mostrar que é verdadeira antes de considerar que o modelo VDM é interna e formalmente consistente.
- Há que considerar três obrigações de prova em modelos VDM:
 - **Verificação dos domínios** (uso de operadores parciais)
 - **Satisfabilidade de definições explícitas**
 - **Satisfabilidade de definições implícitas**

} Relativas a invariantes

Verificação dos domínios

- O uso de um operador parcial for a do seu domínio é considerado erro por parte do modelador. Existem dois tipos de construções que não podem ser verificadas automaticamente:
 - aplicar uma função que tem uma pré-condição; e
 - aplicar um operador parcial.
- Algumas definições:
 - $f: T_1 * T_2 * \dots * T_n \rightarrow R$
 - $f(a_1, \dots, a_n) == \dots$
 - $pre \dots$
- Pode-se referir a pré-condição de f como uma função Boolean com a seguinte assinatura:
 - $pre_f: T_1 * T_2 * \dots * T_n \rightarrow bool$

Verificação dos domínios

- Se uma função g usa um operador $f: T_1 * \dots * T_n \rightarrow R$ no seu corpo, ocorrendo como uma expressão $f(a_1, \dots, a_n)$, então é necessário mostrar que a pré-condição de f :
 $pre_f(a_1, \dots, a_n)$
é satisfeita para qualquer a_1, \dots, a_n que ocorrer nesta posição.
- Exemplo:
AnalyseInput: Gateway -> Gateway
AnalyseInput(g) ==
if Classify(hd g.input) = <High>
then mk_Gateway(tl g.input,
g.outHi ^ [hd g.input],
g.outLo)
else mk_Gateway(tl g.input,
g.outHi,
g.outLo ^ [hd g.input])
- Obrigação de prova para verificação do domínio:
forall g:Gateway & pre_AnalyseInput(g) => g.input <> []

Verificação dos domínios

- Os operadores parciais podem ser protegidos por pré-condições:

```
AnalyseInput: Gateway -> Gateway
AnalyseInput(g) ==
if Classify(hd g.input) = <High>
then mk_Gateway(tl g.input,
g.outHi ^ [hd g.input],
g.outLo)
else mk_Gateway(tl g.input,
g.outHi,
g.outLo ^ [hd g.input])
pre g.input <> []
```

- Agora, a obrigação de prova verifica-se
 $pre_AnalyseInput(g) == g.input \neq []$

Verificação dos domínios

- Alternativamente, um operador parcial pode ser protegido incluindo uma verificação explícita no corpo da função, ex.:

```
AnalyseInput: Gateway -> (Gateway)
AnalyseInput(g) ==
if g.input <> []
then if Classify(hd g.input) = <High>
then mk_Gateway(tl g.input,
g.outHi ^ [hd g.input],
g.outLo)
else mk_Gateway(tl g.input,
g.outHi,
g.outLo ^ [hd g.input])
else nil
```

- Se se incluir esta verificação, há que retornar um valor especial de indicação de erro e assegurar que o tipo de retorno da função é opcional (para lidar com return nil).

Verificação dos domínios

- Pode ser difícil decidir o que incluir numa pré-condição.
 - Algumas condições são determinadas pelos requisitos.
 - Muitas condições são condições para garantir o correcto funcionamento de operadores e funções parciais.
- Quando se define uma função, há que lê-la sistematicamente, realçar o uso de operadores parciais, e assegurar que não haverá uso indevido desses operadores adicionando o conjunto apropriado de pré-condições.

Preservação de Invariante

- Todas as funções devem assegurar que o resultado não é só estruturalmente do tipo correcto, mas também que respeita o invariante do tipo resultado.
- Todas as operações devem assegurar que os invariantes nas variáveis de instância e nos tipos dos resultados são verificados
- Formalmente, a preservação de invariante deve verificar-se em todos os *inputs* que respeitam as pré-condições de funções e de operações
- Exemplo

```
AddFlight: Flight ==> ()
AddFlight (f) ==
    journey := journey ^ f
pre journey(len journey).destination = f.departure
```

Satisfabilidade de funções explícitas

- Função explícita sem pré-condição definida

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

diz-se **satisfiable** se, para todos os inputs, o resultado definido pelo corpo da função é do tipo correcto. Formalmente,

```
forall p1:T1,...,pn:Tn & f(p1,...,pn) : R
```

- Uma função explícita com pré-condição:

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

diz-se **satisfiable** se, para todos os inputs que satisfazem a pré-condição, o resultado definido pelo corpo da função é do tipo correcto. Formalmente,

```
forall p1:T1,...,pn:Tn &
pre_f(p1,...,pn) => f(p1,...,pn) : R
```

Satisfabilidade de funções implícitas

- Uma função *f* definida implicitamente como

```
f(a1:T1,...,an:Tn) r:R
pre ...
post ...
```

- diz-se **satisfiable** se, para todos os inputs que satisfazem a pré-condição, existe um resultado do tipo correcto que satisfaz a pós-condição. Formalmente,

```
forall p1:T1,...,pn:Tn &
pre_f(p1,...,pn) =>
exists x:R & post_f(p1,...,pn,x)
```

E.g.

```
f(x: nat) r:nat
pre x > 3
post r > 10 and r < 10
```

Não é possível encontrar um resultado do tipo nat que satisfaça

```
post_f
```

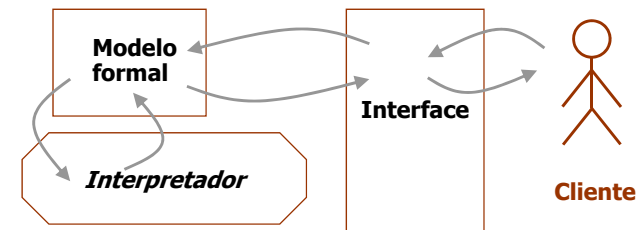
– então **é não é satisfazível**

Comportamento

- Outro aspecto da validação do modelo é assegurar que ele de facto descreve o comportamento esperado do sistema em análise.
- Há três abordagens possíveis:
 - **Animação** do modelo - funciona bem com clientes que não estão familiarizados com notações de modelação mas requer uma boa interface com o utilizador.
 - **Testar** o modelo - pode-se aferir a cobertura do modelo mas os resultados estão limitados à qualidade dos testes e o modelo tem que ser executável.
 - **Provar** propriedades sobre o modelo - Assegura uma excelente cobertura, não requer um modelo executável, mas o suporte de ferramentas é limitado.

Animação

- O modelo é **animado** através de uma interface com o utilizador. A interface pode ser construída numa linguagem de programação à escolha desde que tenha a possibilidade de ligação dinâmica (*dynamic link facility*) para interligação do código da interface ao modelo.



Teste sistemático

- O nível de confiança ganho com a animação do modelo depende do conjunto particular de cenários que se decidiu executar sobre a interface.
- No entanto, é possível um teste mais sistemático:
 - Definir a colecção de casos de teste
 - Executar esses testes no modelo formal
 - Comparar o resultado com o esperado
- Os casos de teste podem ser gerados manualmente ou automaticamente. A geração automática pode produzir um conjunto vasto de casos de teste.
- Técnicas de geração de casos de teste sobre programas funcionais também podem ser aplicados sobre modelos formais.

Validação por prova

- Systematic testing and animation are only as good as the tests and scenarios used. *Proof* allows the modeller to assess the behaviour of a the model for whole classes of inputs in one analysis.
- In order to prove a property of a model, the property has to be formulated as a logical expression (like a proof obligation). A logical expression describing a property which is expected to hold in a model is called a *validation conjecture*.
- Proofs can be time-consuming. Machine support is much more limited: it is not possible to build a machine that can automatically construct proofs of conjectures in general, but it is possible to build a tool that can check a proof once the proof itself is constructed. Considerable skill is required to construct a proof - but a successful proof gives high assurance of the truth of the conjecture about the model.

Validação por prova

Níveis de prova:

- **“Textbook”**: argument in natural language supported by formulae. Justifications in the steps of the reasoning appeal to human insight (“Clearly ...”, “By the properties of prime numbers ...” etc.). Easiest style to read, but can only be checked by humans.
- **Formal**: at the other extreme. Highly structured sequences of formulae. Each step in the reasoning is justified by appealing to a formally stated rule of inference (each rule can be axiomatic or itself a proved result). Can be checked by a machine. Construction very laborious, but yields high assurance (used in critical applications)
- **Rigorous**: highly structured sequence of formulae, but relaxes restrictions on justifications so that they may appeal to general theories rather than specific inference rules.

Revisão

- **Validação**: the process of increasing confidence that a model accurately reflects the client requirements.
- **Consistência interna**:
 - **domain checking**: partial ops and functions with precondition
 - Protect with preconditions or if-then-else
 - **satisfiability** of explicit and implicit function
 - Ensure invariants are respected
- **Checking accuracy**:
 - animation
 - testing
 - proof

increase cost
increase confidence

Pré/Pós-condições e herança

- Ao redefinir uma operação herdada da superclasse, não se deve violar o contracto (pré e pós-condição) estabelecido na super-classe
- A pré-condição pode ser enfraquecida (relaxada) na subclasse, mas não fortalecida (não pode ser mais restritiva)
 - qualquer chamada que se prometia ser válida na pré-condição da superclasse, deve continuar a ser aceite na pré-condição da subclasse
 - `pre_op_superclass => pre_op_subclass`
- A pós-condição pode ser fortalecida na subclasse, mas não enfraquecida
 - a operação na subclasse deve continuar a garantir os efeitos prometidos na superclasse, podendo acrescentar outros efeitos
 - `post_op_subclass => post_op_superclass`
- *Behavioral subtyping*

Pré/Pós-condições e herança

```
class Figura
types
  public Ponto :: x : real
                y : real;

instance variables
  protected centro : Ponto;

operations

  public Resize(factor: real)
==
  is subclass responsibility
  pre factor > 0.0
  post centro = centro~;
end Figura

class Circulo is subclass of Figura
instance variables
  private raio : real;
  inv raio > 0;

operations
  public Circulo(c: Ponto, r: real) res: Circulo
  == ( raio := r; centro := c; return self )
  pre r > 0;

  public Resize(factor: real) ==
  raio := raio * abs(factor)
  pre factor <> 0.0
  post centro = centro~ and
        raio = raio~ * abs(factor);
end Circulo
```

pre Figura `Resize(...) => pre Circulo `Resize(...)

post Figura `Resize(...) <= post Circulo `Resize(...)

Teste da especificação

- Uma especificação bem construída já tem verificações *built-in*
 - Invariantes, pré/pós-condições, outras asserções (invariantes de ciclos, etc.)
- Mas deve ser exercitada de forma repetível com testes automatizados
 - O objectivo é descobrir erros e ganhar confiança na correcção da especificação
 - Mais tarde, os mesmos testes podem ser aplicados à implementação
- Testar com entradas válidas
 - Exercitar toda as partes da especificação (medir cobertura com VDMTools)
 - Usar asserções para verificar valores devolvidos e estados finais
 - (Op) Derivar testes a partir de máquinas de estados (teste baseado em estados)
 - (Op) Derivar testes a partir de cenários de utilização (teste baseado em cenários)
 - (Op) Derivar testes de especificações axiomáticas (teste baseado em axiomas)
- Testar com entradas inválidas
 - Quebrar todos os invariantes e pré-condições, para verificar que funcionam ...

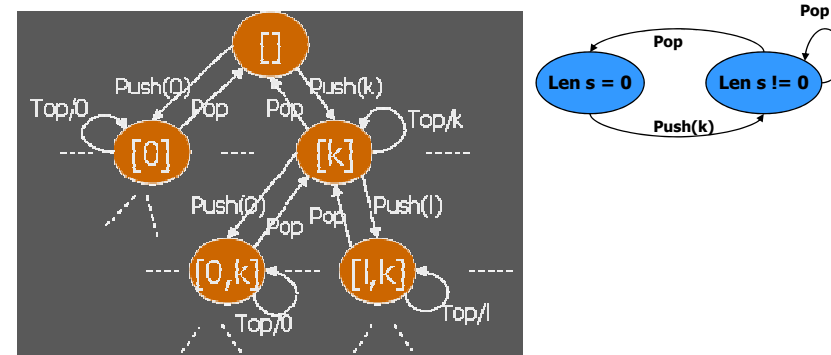
Behaviour-based notations FSM / CSP / temporal logic

Behaviour-based notations

- Examples:
 - Petri nets; Process algebras (e.g., CSP - Communicating Sequential Processes); state machines (e.g., statecharts); temporal logic
- Advantages
 - Particular useful for modelling concurrent systems
 - May be used by model checking techniques
 - More abstract than model-based notations
 - ...
- Disadvantages
 - Not so appropriate for modelling state
 - Not so closed to programming languages
 - ...

Transition-based: FSM

- Choosing the set of states is a critical step



- Statecharts are another option

CSP - Communicating sequential process

$x \rightarrow P$	Receive event x then behave like process P
$\alpha(P)$	Set of events that P can respond to
$(x \rightarrow P \mid y \rightarrow Q)$	Choose
$ch?x$	Receive x from channel ch
$ch!x$	Send x through channel ch
$\alpha(c) = \{v \mid c.v \in \alpha P\}$	Set of messages that channel c knows
$P \sqcap Q$	Or P or Q (non deterministic internal choose)
$P \sqbox Q$	Or P or Q (non deterministic external choose)
$P;Q$	Sequence
$P \Delta Q$	Interrupt
$P \Leftarrow b \rightarrow Q$	If b then P else Q
$b * P$	While b do P
$P \parallel Q$	Interleaving

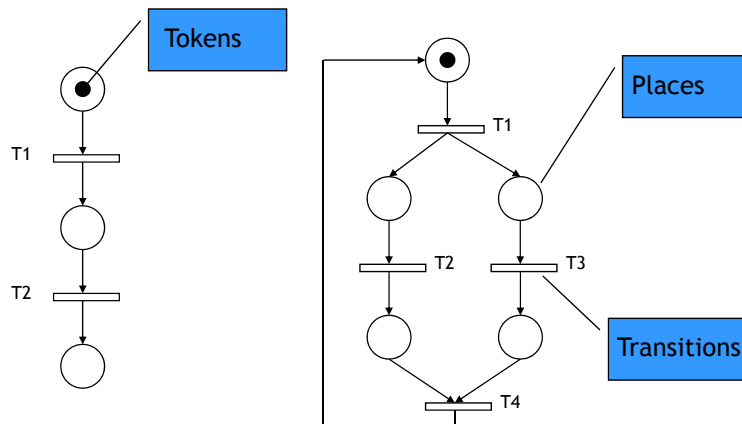
Behaviour-based: CSP

$\text{Stack}(\langle \rangle) = \text{push?}x:E \rightarrow \text{Stack}(\langle x \rangle)$
 $\text{Stack}(\langle y \rangle^{\wedge} s) = \text{push?}x:E \rightarrow \text{Stack}(\langle x \rangle^{\wedge} \langle y \rangle^{\wedge} s)$
 $\quad \mid \text{pop!}y \rightarrow \text{Stack}(s)$

OR

$\text{Stack}() = \text{push?}x;\text{Stack}(x)$
 $\text{Stack}(x) = \text{push?}y;\text{Stack}(y);\text{Stack}(x)$
 $\quad \mid \text{pop!}x;\text{Stack}()$

Petri nets

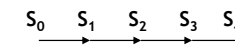


The state is modelled by the position of the tokens.

Temporal logic

Linear Temporal Logic (LTL)

Linear time.

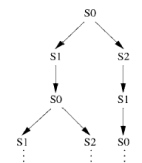


Time operators.

“Computation Tree Logic” (CTL)

Branching Time

Time operators + path formulae



“Timed CTL” (TCTL)

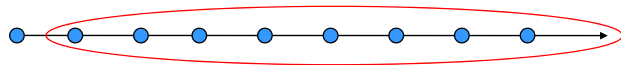
For time real systems

Linear Temporal logic

$\diamond p$ - Eventually p



$\square p$ - Globally p



op - Next p



pUq - p until q



Abbreviated formulas

- GFp (“always there will be a state such that p”)

(existirá sempre um estado em que se verifique p)

$$\boxed{\infty F p} \quad \text{“infinitely often”}$$

- FGp (“all the time for a certain time onwards”)

(a partir de um determinado estado, verifica-se sempre p)

$$\boxed{\infty G p}$$

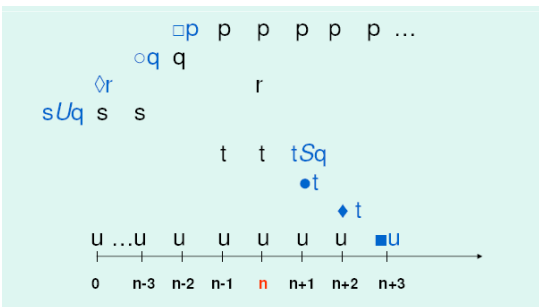
Time operators (past)

$\blacklozenge p$ - somewhere in the past.

$\blacksquare p$ - always in the past.

$\bullet p$ - In previous state.

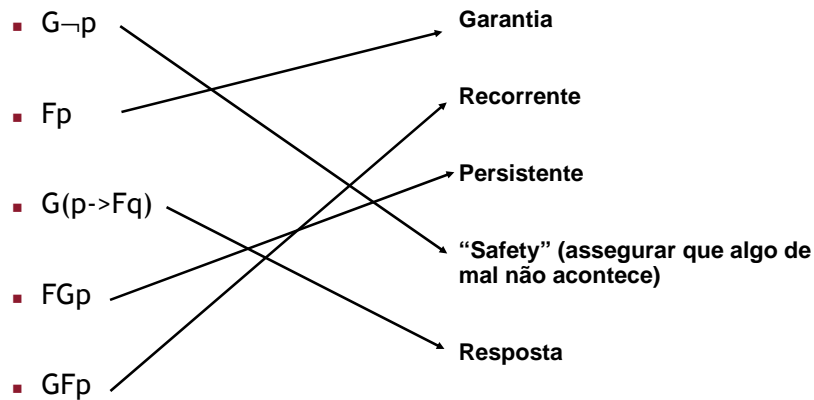
pSq - p since q.



Properties

- Reachability** property
 - states that some particular situation can be reached
- Safety** property
 - expresses that, under certain conditions, something never occurs
- Liveness** property
 - expresses that, under certain conditions, something will ultimately occur
- Fairness** property
 - expresses that, under certain conditions, something will (or will not) occur infinitely often.
- Deadlock-freeness** property
 - whatever the state reached may be, there will exist an immediate successor state.

LTL formulas



Quantifiers

A : Ao longo de todas as execuções

E : Ao longo de uma execução

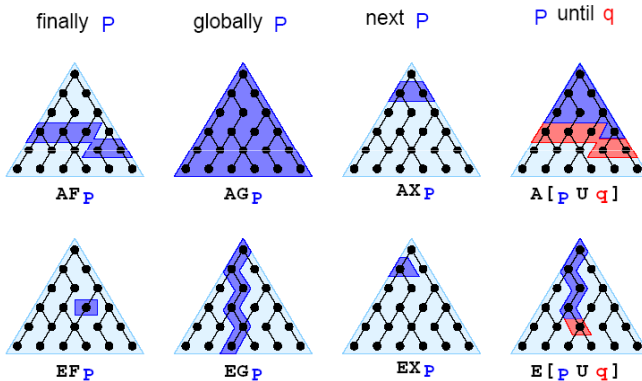
▪ Qual a diferença entre $AGFp$ e $AGEFp$?

$AGFp = A F p$: ao longo de todas as execuções (A), em todos os instantes de tempo (G), ir-se-á encontrar mais tarde (F) um estado que verifique p. Assim, p terá que ser satisfeita recorrentemente (*infinitely often*).

$AGEFp$: Em qualquer instante de qualquer execução deverá ser possível vir a satisfazer p, ou seja, p é sempre potencialmente atingível, mesmo que exista uma execução em que p nunca se verifique. Ao longo de todas as execuções, o segundo quantificador, E, permite exprimir o facto de que existem execuções alternativas que permitem obter comportamentos diferentes do sistema (ex.: em que se verifique p)

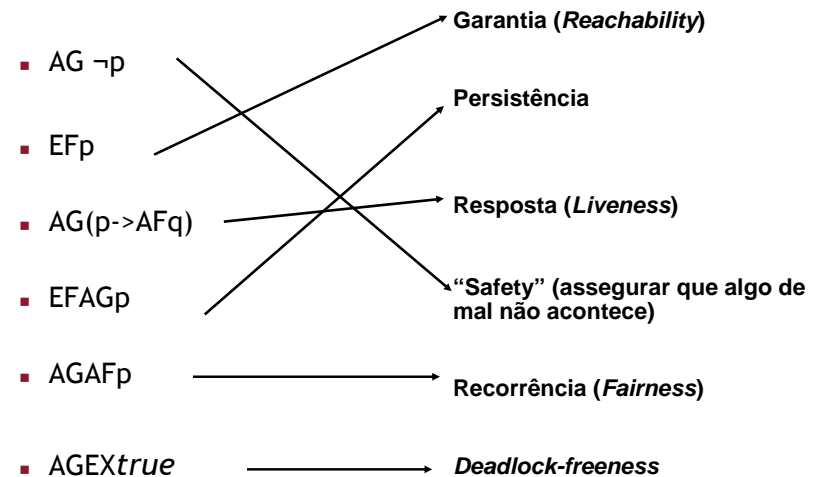
Computation Tree Logic (CTL)

Operadores temporais antecidos por quantificadores sobre caminhos.



AG p - Universal;
 AF p - Inevitável;
 EF p - É possível;
 AX p - No próximo estado;
 A[p U q] - p até q;
 E - Existe um caminho;
 A - Para todos os caminhos

Formulas



Propriedades - sumário

- **Reachability** (“atingível”) $EF\phi$
 - Um dado estado/situação é atingível
- **Safety** (“segurança”) $AG\neg\phi$
 - Dentro de determinadas condições, uma dada situação nunca ocorre
- **Liveness** (“certeza, resposta”) $AG(\text{req} \Rightarrow AF\text{sat}), AGEFinit$
 - Dentro de determinadas condições, uma dada situação irá ocorrer
- **Fairness** (“justiça, recorrência”) $AGAF\phi$
 - Dentro de determinadas condições, uma dada situação irá (ou não) ocorrer recorrentemente (*infinitely often*)
- **Deadlock-freeness** $AGEXtrue$
 - Para qualquer estado, existe sempre um estado sucessor.

Propriedades

Exemplos:

“Safety condition” (algo de mal não acontece)

Não podem estar duas luzes verdes acesas em duas ruas diferentes ao mesmo tempo

$$AG-(G1 \wedge G2)$$

“Fairness Condition” (algo de bom acontece)

No futuro, uma das ruas terá a luz verde acesa

$$EF(G1 \vee G2)$$

“Weak until” qualquer que seja o comportamento (A), o carro nunca iniciará a marcha (starts) enquanto (W) a chave não for colocada na ignição (key)

$$A\neg\text{starts } W \text{ key} = (\neg\text{starts } U \text{ key}) \vee G\neg\text{starts}$$

Timed temporal logic (TCTL)

▪ Gramática formal de TCTL

$\phi, \psi ::= P1 \mid P2 \mid \dots$ (proposições atómicas)

$\mid \neg\phi \mid \phi \wedge \psi \mid \phi \Rightarrow \psi \mid \dots$ (operadores Booleanos)

$\mid EF_{(-k)}\phi \mid EG_{(-k)}\phi \mid E\phi U_{(-k)}\psi$ (operadores de tempo)

$\mid AF_{(-k)}\phi \mid AG_{(-k)}\phi \mid A\phi U_{(-k)}\psi$

~ representa um símbolo de comparação {<, ≤, =, >, ≥} e k qualquer número racional \mathbb{Q}

$PU_{(<2)}Q$ significa que P é verdade até Q e que Q se verifica dentro de duas unidades de tempo a contar do presente instante

Problemas

▪ Explosão de estados: Soluções

Existem técnicas para representar e explorar os estados no espaço de estados de forma mais eficiente):

- **Abstracção** - Uma representação simplificada onde detalhes de baixo nível são ignorados.

- **Limitar o domínio** - o domínio das variáveis é limitado a um conjunto finito.

- “**Partial Order Reduction**” - quando temos caminhos independentes, a ordem pela qual os atravessamos não é relevante.

- “**Symbolic Model Checking**” - usa expressões simbólicas para representar caminhos de execução.

- **BDD** (Binary Decision Diagrams) - caso especial de “symbolic model checking” onde os estados são representados por fórmulas booleanas.

Exemplo

Um sistema que:

- Passa de pronto (*ready*) para ocupado (*busy*) por causa de um pedido (*request*) ou por outra razão não-determinística.
- Passa de ocupado (*busy*) para pronto (*ready*) de forma não-determinística.

Pretende-se verificar se

$$AG(request \rightarrow AF status = busy)$$

Sempre (AG) que recebido um pedido (*request*), ele será (AF) processado no futuro.

“Whenever (AG) a request arrives it will be processed eventually (some time in the future) (AF)”

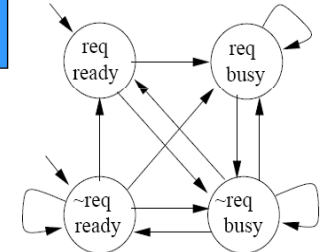
Exemplo

```
MODULE main
VAR
  request : boolean;
  status : {ready, busy};
ASSIGN
  init(status) := ready;
  next(status) :=
    case
      request : busy;
      1 : {ready, busy};
    esac;
SPEC
  AG(request -> AF status = busy)
```

InicializaStatus.
Request não é
inicializado por
isso pode tomar
qualquer valor.

Escolha não-
determinística

Secção para as
propriedades



Property-based notation OBJ

Property-based

- Examples
 - OBJ and OBJ-based (e.g., OBJ1, OBJ2, OBJ3, CafeOBJ, Maude, etc.); Larch
- Advantages
 - High level of abstraction
 - Particular useful for modelling abstract data types (ADT) and interfaces
 - ...
- Disadvantages
 - Difficult to know when the specification is finished
 - Too far from programming languages
 - ...

Property-based: OBJ

Spec: Stack;

Extend Nat by

Sorts: Stack;

Operations:

newstack: \rightarrow Stack

push: $\text{Stack} \times \text{Nat} \rightarrow \text{Stack}$

pop: $\text{Stack} \rightarrow \text{Stack}$

top: $\text{Stack} \rightarrow \text{Nat}$

Variables:

s: Stack; n: Nat

Axioms:

$\text{pop}(\text{newstack}) = \text{newstack}$;

$\text{top}(\text{newstack}) = \text{zero}$;

$\text{pop}(\text{push}(s,n)) = s$;

$\text{top}(\text{push}(s,n)) = n$;

Mais abstracta: semântica de operações especificada por axiomas equacionais, sem necessidade de escolher uma representação de dados interna!

Mas é mais difícil saber se a especificação está completa!

Uma instância do tipo definido é sempre representada por uma expressão de construção

$\text{push}(\text{push}(\text{push}(\text{newstack}, 1), 2), 3)$

Axiomas permitem simplificar/avaliar expressões

$\text{pop}(\text{push}(\text{push}(\text{newstack}, 1), 2))) = \text{push}(\text{newstack}, 1)$

$\text{top}(\text{pop}(\text{push}(\text{push}(\text{newstack}, 1), 2))) = \text{top}(\text{push}(\text{newstack}, 1)) = 1$

References and further reading

- <http://www.vdmtools.jp/en/>
- <http://www.usingcsp.com/>
- <http://plato.stanford.edu/entries/logic-temporal/>
- <http://www-cse.ucsd.edu/~goguen/sys/obj.html>
- <http://www.fmeurope.org/>