Métodos Formais na Especificação de Interfaces com o Utilizador: a Linguagem VDM++ e o Tratamento de Eventos

Ana Cristina Ramada Paiva(1)

(1) Faculdade de Engenharia da Universidade do Porto, Porto, Portugal.

apaiva@fe.up.pt

João P. Faria(1), Raul M. Vidal(1), José N. Oliveira(2)

(2) Universidade do Minho, Braga, Portugal

rmvidal@fe.up.pt; jpf@fe.up.pt; jno@di.uminho.pt

Resumo

Este artigo faz uma avaliação da utilização de uma linguagem geral de especificação, *VDM++*, para especificação de interfaces com o utilizador analisando em particular uma das suas limitações que condicionam a sua aplicabilidade (o tratamento de eventos). Começa por uma classificação geral dos Métodos Formais existentes de acordo com os aspectos em que se baseiam (modelos, propriedades ou comportamentos) referindo as diferentes abordagens de aplicação destes métodos à especificação de interfaces. De seguida, é apresentado o caso de estudo onde se cria uma especificação, em *VDM++*, para um diálogo (validação de um utilizador e a sua palavra-chave) por um processo de engenharia reversa do código *C#* necessário para a implementação desse diálogo, avaliando-se a utilização da linguagem e apresentando-se uma solução para o tratamento de eventos.

Palavras chave: Engenharia de Software, Métodos Formais, Especificação de Software, Interfaces, *VDM++*.

1 Introdução

Os Métodos Formais são "Técnicas baseadas na matemática para descrever propriedades de um sistema" [Wing 1994]. Estas técnicas incluem notações precisas para construir modelos matemáticos de um sistema (*software*). Estas notações são usadas para especificar (em vez de desenhar e implementar) o sistema requerido, e preocupam-se com "o que é feito" pelo sistema em vez de "como é feito". Os Métodos Formais são precisos e abstractos, têm vindo a ganhar força dentro dos sistemas de desenvolvimento de *software* e podem ser aplicados em diversas fases de um processo de engenharia de *software*.

O principal objectivo dos Métodos Formais é orientar a produção de *software* segundo os critérios de qualidade exigidos por outras engenharias; por exemplo, podem ser usados para aumentar a confiança na correcção de software através de provas, processos de refinamento[1] e testes. Como regra geral, estes métodos dividem a produção de *software* nas fases de

[1] Processo de transformação do modelo do sistema com o objectivo de o tornar mais detalhado e operacional sem alterar as propriedades do modelo original.

especificação, em que um modelo matemático é construído a partir dos requisitos identificados pelo cliente, e de implementação, em que o modelo é de alguma forma convertido num programa executável. Avanços recentes da investigação em Métodos Formais mostram que as implementações podem ser efectivamente calculadas a partir de especificações. Assim, a tecnologia do *software* está a tornar-se uma disciplina madura que adopta a estratégia de "resolução de problemas universais".

A introdução da matemática e cálculo algébrico na engenharia de *software* surge como uma evolução natural, quando comparada com o desenvolvimento histórico das bases científicas de outras áreas da engenharia (ex.: engenharia mecânica e civil) que, há alguns séculos atrás, começaram a pôr de parte provas geométricas complicadas a favor de raciocínios algébricos. O grau de sofisticação dos métodos depende do tipo de matemática em que se baseiam: aritmética simples, sistemas de equações lineares, equações diferencias ou integrais e agora, no cálculo de *software*, sistemas de equações (recursivos) em espaços de domínio.

Os Métodos Formais podem ser classificados de acordo com o aspecto em que se baseiam: modelo, propriedades ou comportamento [MacColl e David 1997]. Os Métodos baseados em modelos e propriedades preocupam-se com as estruturas de dados subjacentes e as suas operações enquanto que os Métodos baseados no comportamento se focam no comportamento do sistema observável exteriormente:

1.1 Métodos baseados em modelos

Definem o comportamento do sistema através da construção de um modelo usando estruturas matemáticas como por exemplo os conjuntos. As notações baseadas em modelos para sistemas sequenciais incluem *Z* [Z], *VDM* [Jones 1990][Fitzgerald e Peter 1998] e são adequadas à especificação de *software* que envolva estruturas de dados complexas e operações uma vez que os tipos de dados são modelados explicitamente.

1.2 Métodos baseados em propriedades

Incluem notações como *OBJ* [OBJ] ou *Larch* [Larch] que modelam tipos de dados implicitamente definindo o comportamento do sistema através de um conjunto de propriedades. Estes Métodos podem ser também classificados em axiomáticos e algébricos: os primeiros baseiam-se em predicados da lógica de primeira ordem e os segundos usam axiomas equacionais.

1.3 Métodos baseados no comportamento

Definem sistemas em termos de sequências possíveis de estados em vez de tipos de dados e são normalmente usados para especificar sistemas concorrentes e distribuídos. Estes métodos

incluem Redes de *Petri* [PetriNets], *Calculus of Communicating Systems (CCS), Communicating Sequencial Processes (CSP)* [Alexander 1990] e Diagramas de Estados.

2 Características que tornam uma especificação formal

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.

3 Métodos Formais na especificação de interfaces

Os Métodos Formais têm vindo a ganhar força dentro dos sistemas de desenvolvimento de *software* mas a sua aplicação na especificação de interfaces para/com o utilizador não é tão comum. A especificação de interfaces é muitas vezes fornecida através de protótipos ou recorrendo a outras técnicas não formais. Destas técnicas não formais resultam especificações que apresentam ambiguidades e que conduzem a diferentes interpretações pelos diferentes intervenientes no processo — clientes, utilizadores e programadores. Sem uma técnica de especificação formal os mal entendidos vão surgir e a interface resultante será inválida e, muitas vezes, não usável. A especificação formal de interfaces poderá ajudar a descobrir inconsistências e problemas de utilização antes do desenvolvimento da própria interface, o que resulta numa economia de tempo e recursos.

Escrever uma especificação antes da implementação pode ser útil na medida em que permite estudar vários cenários sem ter que os codificar. A especificação deve descrever a estrutura e o comportamento da interface do ponto de vista do utilizador mas não deve restringir a sua implementação. Neste sentido, as linguagens de especificação de interfaces devem permitir construir e manter especificações que facilitem o estudo e análise de um conjunto alargado de possíveis cenários. A especificação deve ser expressiva e comunicável e poderá servir de base para uma discussão com os clientes e potenciais utilizadores.

Em termos gerais, poder-se-á dizer que as linguagens de especificação de interfaces devem ser capazes de expressar os requisitos para a interface e transmiti-los, com sucesso, aos seus leitores.

4 Linguagens de especificação de interfaces

A especificação de interfaces descreve o diálogo entre um sistema interactivo e os seus utilizadores. As linguagens de especificação, aplicadas às interfaces, permitem verificar se os requisitos identificados para o diálogo são cumpridos, e analisar os aspectos práticos e funcionais da utilização das interfaces [Chesson].

Nos últimos dez anos, várias têm sido as abordagens seguidas para especificação de interfaces com o utilizador [Harrison e Duke 1995]. Estas abordagens incluem:

- aplicação directa de linguagens gerais de especificação;
- desenvolvimento de novas linguagens a partir da semântica já existente em linguagens gerais de especificação;
- desenvolvimento de novas linguagens com definição de semântica própria.

Na primeira abordagem podemos encontrar os modelos abstractos *PIE* e *RED-PIE* em [Dix e Runciman 1985] que usam notações matemáticas genéricas para descrever propriedades da interface com o utilizador. A dificuldade em fazer determinadas especificações resulta do facto de estas notações terem um nível de abstracção elevado, que nem sempre permite expressar características específicas das aplicações. As questões temporais também são de difícil enquadramento nestas notações. Outro exemplo destas abordagens pode ser encontrado em [Johnson e Harrison 1992], onde a especificação da interface usa lógica de primeira ordem e lógica temporal. Outros obstáculos surgem quando se tentam adicionar características comportamentais às linguagens baseadas em modelos e são apresentados em [Galloway e Stoddart 1997]. Em resumo, a aplicação directa de linguagens gerais de especificação fica condicionada pelas características da linguagem usada mas beneficia do facto de já existirem diversas ferramentas para auxiliar a verificação e realizar os testes.

A segunda abordagem é a mais comum e é aquela em que se desenvolvem novas linguagens a partir da semântica de linguagens gerais já existentes. O objectivo consiste em seleccionar as partes da semântica de cada uma das linguagens naquilo em que são mais fortes e obter um conjunto híbrido mas com aplicabilidade mais vasta. Na selecção da semântica distinguimos as linguagens comportamentais (são preferíveis para os aspectos dinâmicos) das linguagens baseadas em modelos e propriedades (são boas para formalizar aspectos estáticos) [MacColl e David 1997]. Um exemplo comum é complementar o poder expressivo das linguagens algébricas para a perspectiva comportamental com as abordagens baseadas em modelos para a especificação dos dados e processos. Por exemplo, em [Galloway e Bill 1997] cria-se uma nova linguagem híbrida, *ZCCS*, que integra aspectos da linguagem *Z* baseada em modelos e a álgebra de processos *CCS*. Em [MacColl e David 1999] é apresentada uma solução que integra *Object-Z*

e *CSP*. A linguagem *Object-Z,* baseada em modelos, é usada para definir a funcionalidade e apresentação enquanto que a linguagem *CSP,* baseada no comportamento, é usada para a definição explícita e análise de sequências de operações. A integração de *Object-Z* e *CSP* permite o uso das classes *Object-Z* como processos *CSP*. Em [The RAISE 1995] surge a linguagem de especificação *RSL* que possui características da linguagem *VDM* baseada em modelos, métodos algébricos como *ACT ONE* e *OBJ* e álgebras de processo *CSP* e *CCS*. O inconveniente desta abordagem é a necessidade de desenvolver as ferramentas de suporte que verifiquem o resultado da combinação das linguagens e da maior dificuldade de comprovar os testes.

Por último, a abordagem da construção de novas linguagens tem o esforço de um desenvolvimento de raiz assim como o perigo da criação de mais uma linguagem de características específicas que depois só poderá ser aplicada a um grupo restrito de problemas. Neste grupo podemos inserir a linguagem de especificação *JML* [Leavens, Albert e Clyde 1999]. As expressões desta linguagem podem ser intercaladas no código *Java* para auxiliar a verificação e *debugging*. Esta linguagem aproveita algumas ideias para a sintaxe presentes no *Eiffel* e para a semântica aproveita algumas ideias das linguagens baseadas em modelos.

5 VDM++

Os Métodos Formais precisam do desenvolvimento de ferramentas que apoiem a sua utilização favorecendo o seu uso efectivo nas empresas. O surgimento de novas linguagens de especificação nem sempre tem este objectivo como preocupação principal. A linguagem de especificação *VDM++* (um caso na primeira abordagem da classificação da secção anterior), além de ter um bom conjunto de ferramentas de suporte, é uma linguagem de fácil apreensão e por isso tem uma boa probabilidade de vir a ser largamente usada no âmbito empresarial. A linguagem de especificação *VDM++* é baseada na norma *ISO/VDM-SL*, é orientada aos objectos e tem um conjunto de ferramentas, *VDMTools*, de suporte que permitem o desenvolvimento e análise precisa de modelos de sistemas de computação. Estas ferramentas permitem a verificação automática e validação dos modelos dos sistemas expressos em *VDM++* antes da sua implementação. A maior diferença entre o *VDM++* e o *VDM-SL* é a orientação por objectos e as extensões para suporte de concorrência.

As ferramentas *VDMTools* incluem: verificadores sintácticos; verificadores de tipos através de um mecanismo de inferência de tipos para detectar o uso errado de valores e/ou operadores; interpretador e *debugger*; facilidades de teste que permitem exercitar a especificação com um conjunto de testes; geração automática de código para *C++* e *JAVA*; uma *API CORBA* que

permite que outros programas acedam à ferramenta e uma ligação ao *Rational Rose*[1] que permite a integração de *UML* e *VDM++*.

Esta linguagem *VDM++* suporta funções polimórficas, funções como argumento, herança, tratamento de excepções, sincronização e *threads*. A sincronização é implementada através de sentinelas

pre A => B

Se B se verificar então A pode ocorrer, ou seja, B é condição necessária para ocorrer A mas não é condição suficiente.

Contudo, esta linguagem apresenta como principal limitação a não existência de suporte para eventos.

6 Caso de Estudo

Neste caso de estudo pretendemos estudar e construir uma especificação *VDM++* para um diálogo comum de validação de utilizador e sua palavra-chave.

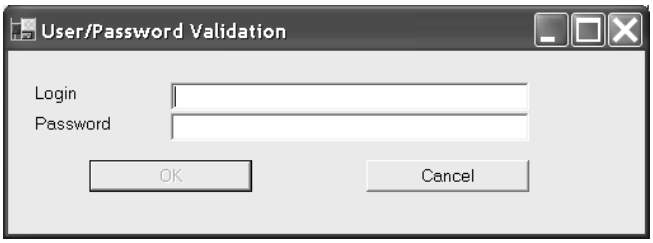

Figura 1 – Diálogo de validação de acesso.

Este diálogo deve obedecer aos seguintes requisitos:

R1: Este diálogo terá que permitir inserir duas palavras (utilizador e palavra-chave) e deverá ter dois botões, um para validar (*OK*) e outro para cancelar (*Cancel*);

R2: Pretende-se que o botão *OK* esteja inicialmente desactivado só permitindo interacção com o utilizador após preenchimento das duas caixas de introdução de dados;

R3: O botão *Cancel* tem como função fechar o diálogo.

R4: A função de validação do utilizador e sua palavra-chave deve ser invocada quando o utilizador pressionar o botão *OK*. Se o utilizador for válido apresenta-se a mensagem "User Valid" caso contrário, apresenta-se a mensagem “User not Valid”.

[1] Ferramenta de modelação visual em UML.

A especificação em *VDM++* para o diálogo de interacção e os seus requisitos definidos por R1, R2, R3 e R4 é apresentada na Figura 2:

```
1:   class LoginDialog is subclass of Form
2:   instance variables
3:     public TBLogin: TextBox;
4:     public TBPassword: TextBox;
5:     public BOk: Button;
6:     public BCancel: Button;
7:    inv --derivation
         BOk.Enabled = TBLogin.Text<>"" and TBPassword.Text<>"";

8:   operations
9:     --trigger
10:    public OnBOkClick() == if Validate(TBLogin.Text,TBPassword.Text)
                              then MessageBox`Show("User Valid")
                              else MessageBox`Show("User not Valid!");
11:    --trigger
12:    public OnBCancelClick() == Close();
13:  End LoginDialog
```

Figura 2 – Especificação *VDM++* para diálogo de interacção (especificação inicial).

- o requisito R1 é expresso nas variáveis de instância (3-6) porque se assume que os tipos *TextBox* e *Button* conferem às variáveis o comportamento dos componentes visuais, com os mesmos nomes, presentes na plataforma *.NET*;
- o requisito R2 é expresso sob a forma de um invariante (7);
- os requisitos R3 e R4 são expressos como operações a executar em resposta a eventos (8-12) segundo convenção de nomes On*ObjectEvent*.

Nesta especificação usam-se os tipos *Form*, *TextBox*, *Button* e *MessageBox* que pretendem modelar os componentes visuais, com os mesmos nomes, já existentes na plataforma *.NET*. Para podermos raciocinar formalmente sobre a especificação criada e para podermos testá-la de forma autónoma, foi efectuada uma especificação em *VDM++* dos componentes utilizados, por um processo de engenharia reversa, e foram modelados os eventos. O código, em *C#* (em anexo), necessário para implementar o diálogo de interacção do nosso caso de estudo, foi construído e analisado. Dessa análise extraíram-se as seguintes características:

- um diálogo pode ser realizado por um conjunto de objectos de interacção com o utilizador *Controls* (96-102). Neste caso específico temos um formulário que contém duas caixas de texto, dois botões e duas etiquetas;
- os objectos de interacção respondem a eventos. Cada evento tem uma sequência de funções associadas que serão executadas quando o evento ocorrer. O programador pode adicionar funções a essas sequências de forma a que também elas sejam executadas quando o evento ocorrer (86-90).

A biblioteca de classes construída em *VDM++* para modelar os objectos de interacção disponíveis no ambiente *.NET* é apresentada na Figura 3 e contém apenas as características relevantes para o exemplo a construir.

```
1.  class Control
2.  instance variables
3.    public Enabled: bool:= true;
4.  end Control

5.  class Message is subclass of Control
6.  instance variables
7.    public msg: seq of char;
8.  operations
9.  public static
      Show: seq of char ==> seq of char
      Show(s) == (msg:= s; return s);
10. end Message

11. class TextBox is subclass of Control
12. instance variables
13.   public Text: seq of char := [];
14.   public TextChangedHandlers:
          seq of Event`Handler := [];
15. operations
16. public TextChanged: () ==> Event`Any
    TextChanged () ==
    Event`ExecuteHandlers(TextChangedHandlers)
    Pre Enable = true;
17. end TextBox

18. class Label is subclass of Control
19. instance variables
20.   public Text: seq of char:= [];
21. end Label

22. class Button is subclass of Control
23. instance variables
24.   public Text: seq of char:=[];
25.   public ClickHandlers:
       seq of Event`Handler := [];
26.   public Enabled: bool:= true;
27. operations
28. public Click: ()==> Event`Any
     Click () ==
       Event`ExecuteHandlers(ClickHandlers)
      pre Enabled = true;
29. end Button

30. class Form is subclass of Control
31. instance variables
32.   public AcceptButton: Button;
33.   public CancelButton: Button;
34. operations
35. public Close:() ==> seq of char
     Close() == exit("END");
36. end Form
```

Figura 3– Biblioteca de classes em *VDM++* para modelar objectos de interacção (exemplo em estudo).

Os eventos foram modelados (Figura 4) da seguinte forma:

- criou-se uma classe *Event* com uma operação *ExecuteHandlers* que ao ser invocada executa a sequência de funções do tipo *Event`Handler* passadas como argumento;
- dentro de um objecto de interacção os eventos são definidos através de uma variável capaz de guardar uma sequência de funções (*Event`Handler*) e de uma operação que executa essa sequência quando o evento ocorre;
- para associar funções a um evento de forma a que sejam executadas aquando da sua ocorrência basta concatená-las à sequência de funções desse evento.

```
class Event
types
  public Any = bool | char | int | real | seq of char | Control;
  public Handler :: function: seq of Any -> Any
                    args: seq of Any;
operations
  public static ExecuteHandlers: seq of Event`Handler ==> Event`Any
    ExecuteHandlers ([mk_Event`Handler(f,s)]^t) ==
     if len t = 0 then return f(s) else return f(s)^ExecuteHandlers(t);
end Event
```

Figura 4 – Modelação de eventos em *VDM++*.

Depois de modelados os eventos e os objectos de interacção foi possível construir a especificação para o diálogo (Figura 5).

```
class BusinessLogic
functions
  public static Validate: seq of char * seq of char -> bool
  Validate(s1,s2) == s1="guest" and s2="guest"
end BusinessLogic

class LoginDialog is subclass of Form
instance variables
  public LLogin: Label := new Label();
  public TBLogin: TextBox := new TextBox();
  public Lpassword: Label := new Label();
  public TBPassword: TextBox := new TextBox();
  public BOk: Button:= new Button();
  public BCancel: Button:= new Button();

operations
  public LoginDialog: () ==> LoginDialog
  LoginDialog() ==
   (AcceptButton := BOk;
    CancelButton := BCancel;
    TBLogin.Text := "guest";
    TBPassword.Text := "guest";
    BOk.ClickHandlers := [mk_Event`Handler(OnBOkClick,[self])];
    Bcancel.ClickHandlers := [mk_Event`Handler(OnBCancelClick,[self])];
    TBPassword.TextChangedHandlers:=[mk_Event`Handler(OnTBTextChanged,[self])];
    TBLogin.TextChangedHandlers:=[mk_Event`Handler(OnTBTextChanged,[self])];
     );
  public SetOkEnabled: () ==> bool
    SetOkEnabled () ==
        (BOk.Enabled := (TBLogin.Text <>"" and TBPassword.Text <>"");
    return (TBLogin.Text <>"" and TBPassword.Text <>""););

functions
  public downcast: Control -> LoginDialog
    downcast(c) == c;
  public OnBOkClick: seq of Event`Any -> seq of Event`Any
   OnBOkClick ([s]) ==
   if BusinessLogic`Validate(downcast(s).TBLogin.Text,downcast(s).TBPassword.Text) = true
     then Message`Show("User Valid!") else Message`Show("User not valid!");

  public OnBCancelClick: seq of Event`Any -> seq of Event`Any
    OnBCancelClick ([s]) == downcast(s).Close();

  public OnTBTextChanged: seq of Event`Any -> seq of Event`Any
    OnTBTextChanged ([s]) == [downcast(s).SetOkEnabled()];
end LoginDialog
```

Figura 5 – Especificação em *VDM++* para diálogo de interacção (especificação final).

Esta especificação acrescenta detalhes à especificação inicial, alguns dos quais estão relacionados com características da linguagem *VDM++*, por forma a que possa ser executada e testada dentro do ambiente das ferramentas *VDMTools*. Por exemplo, para impor o invariante

```
7: inv BOk.Enabled = TBLogin.Text<>"" and TBPassword.Text<>"";
```

foi criada, à semelhança do que se passa no código *C#*, a função *OnTBTextChanged* para atribuir o resultado da condição *TBLogin.Text <>"" and TBPassword.Text <>""* à variável *Enabled* do botão *OK*.

Apesar da especificação final, suportada pela linguagem *VDM++*, não apresentar, no nosso ponto de vista, o nível de abstracção desejado, podemos registar duas diferenças entre a implementação em *C#* e a especificação em *VDM++*: o código *C#* é uma implementação e

como tal, tem código relativo à disposição dos elementos de interacção no diálogo (48-85) enquanto que a especificação em *VDM++* se preocupa com a transmissão sem ambiguidades dos requisitos pretendidos para o diálogo não incluindo esses detalhes de implementação.

7 Conclusão

Este artigo faz uma avaliação da utilização de uma linguagem geral de especificação, *VDM++*, para especificação de interfaces com o utilizador analisando em particular uma das suas limitações que condicionam a sua aplicabilidade (o tratamento de eventos).

No nosso caso de estudo criou-se uma especificação, em *VDM++,* para um diálogo de interacção resultante de um processo de engenharia reversa a partir do código *C#* necessário para a implementação do diálogo cumprindo os requisitos previamente definidos. Este exemplo serve para fazer uma reflexão sobre as potencialidades da especificação de interfaces usando a linguagem *VDM++*. Mostra-se também que, apesar da linguagem de especificação *VDM++* não possuir suporte para eventos, eles podem ser modelados como uma sequência de funções a executar aquando da sua ocorrência.

A especificação obtida para o diálogo de interacção suportada pela linguagem não tem o nível de abstracção desejado e, por isso, o nosso trabalho futuro será avaliar a hipótese de gerar esta especificação automaticamente a partir de uma especificação de mais alto nível. De futuro também pretendemos estudar a possibilidade de animar especificações em *VDM++* de diálogos interactivos.

8 Referências

8.1 Publicações

Alexander, Heather, *Formal Methods in human-computer interaction,* chapter 9: Structuring dialogues using CSP. Cambridge university press, 1990.

Chesson, Paul and Lorraine Johnston, *Objectives for User Interface Specification Languages*. Technical Report, September 1996.

Clark, Edmund M. and Jeannette M. Wing, *Formal Methods: State of the Art and Future Directions*, Carnegie Mellon University, 1996.

Dix, A.J. and C. Runciman, *"Abstract models of interactive systems"*, pp. 13-22 in People and Computers: Designing the Interface, ed. P. Johnson & S. Cook, Cambridge University Press, 1985.

Fitzgerald, John and Peter Gorm Larsen, *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.

Galloway, Andy and Bill Stoddart, *Integrated Formal Methods,* Institut de Recherche en Informatic de Nantes, 1997.

Gray, Phil and Chris Johnson, *Requirements For The Next Generation Of User Interface Specification Language*, In P. Palanque and R. Bastide, editors, Proceedings of The Design, Specification And Verification Of Interactive Systems. Springer Verlag, 1995.

Harrison, M.D. and D.J. Duke, *A Review of Formalisms for Describing Interactive Behaviour*, In Software Engineering and Human-Computer Interaction, Volume 896 of Lecture Notes in Computer Science, pp. 49-75, Springer Verlag, 1995.

Hartson, H. Rex and Deborah Hix, *Human-Computer Interface Development: Concepts and System for its Management*, ACM Computing Surveys, Vol 21, Issue 1, 5-92, 1989.

Johson, C.W. and M.D. Harrison, *"Using Temporal Logic to Support the specification of interactive control Systems."* In International Journal of Man-Machine Studies, 37, pp.357-385, 1992.

Jones, Cliff B., *Systematic Software Development using VDM* - The University, Manchester, England – Prentice Hall International, 2nd Ed. edition, April 1990.

Leavens, Gary T., Albert L. Baker and Clyde Ruby, *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, Department of Computer Science, 226 Atanasoff Hall Iowa State University, Ames, Iowa 50011-1040 USA, 1999.

MacColl, Ian and David Carrington, *User Interface Correctness,* Human Computer Interaction 3.3, Spring 1997.

MacColl, Ian and David Carrington. *Specifying Interactive Systems in Object-Z and CSP*. In K. Araki, A. Galloway and K. Taguchi, editors, Integrated Formal Methods, pages 335-352. Springer Verlag. London, June 1999.

Markopoulos, Panos, *A compositional model for the formal specification of user interface software*, PhD Thesis, Queen Mary and Westfield College, University of London, 1997.

Palanque, Philippe, Christelle Farenc and Rémi Bastide, *Human-Computer Interaction* – INTERACT'99, IOS Press, IFIP TC.13, 1999.

Rouff, Christopher, *Formal Specification of User Interfaces*, SIGCHI Bulletin, Vol.28 No.3, July 1996.

The RAISE Language Group, *The RAISE specification language.* BCS Practitioner Series. Prentice-Hall, 1995.

Took R., *Putting design into practice: formal specification and the user interface,* Cambridge University Press, pp. 63-96, In Harrison MD & Thimbleby H (Eds.) Formal Methods in Human-Computer Interaction, 1990.

Wing, Jeanette M., *Formal Methods*, In John J. Marciniak, editor, Encyclopedia of Software Engineering, pages 504-517. John Wiley & Sons, 1994.

8.2 Sítios Web

Larch - http://www.sds.lcs.mit.edu/spd/larch/ [conferido em 30/9/2002]

PetriNets - http://www.daimi.au.dk/PetriNets/ [conferido em 30/9/2002]

OBJ - http://archive.comlab.ox.ac.uk/obj.html [conferido em 30/9/2002]

Z - http://www.afm.sbu.ac.uk/z/ [conferido em 30/9/2002]

Anexo: Programa em C# relativo à janela de diálogo do caso de estudo.

```
1:  using System;
2:  using System.Drawing;
3:  using System.Collections;
4:  using System.ComponentModel;
5:  using System.Windows.Forms;
6:  using System.Data;
7:  using AliasToBL = BL.BusinessLogic;
8:  namespace BL
9:  {
10:     public class BusinessLogic {
11:         public static bool Validate(string login, string password) {
12:             return login=="guest" && password=="guest";
13:         }
14:     }
15: }
16: namespace WindowsApplication1
17: {
18:  public class LoginDialog : System.Windows.Forms.Form
19:  {
20:    private System.Windows.Forms.TextBox TBLogin;
21:    private System.Windows.Forms.TextBox TBPassword;
22:    private System.Windows.Forms.Label LLogin;
23:    private System.Windows.Forms.Label LPassword;
24:    private System.Windows.Forms.Button BOk;
25:    private System.Windows.Forms.Button BCancel;
26:    private System.ComponentModel.Container components = null;
27:    public LoginDialog() {
28:        InitializeComponent();
29:    }

30:    /// Clean up any resources being used.
31:    protected override void Dispose( bool disposing )  {
32:      if( disposing )  {
33:        if (components != null)  {
34:          components.Dispose();
35:        }
36:      }
37:      base.Dispose( disposing );
38:    }
39:    #region Windows Form Designer generated code
40:    private void InitializeComponent()  {
41:      this.TBLogin = new System.Windows.Forms.TextBox();
42:      this.TBPassword = new System.Windows.Forms.TextBox();
43:      this.LLogin = new System.Windows.Forms.Label();
44:      this.LPassword = new System.Windows.Forms.Label();
45:      this.BOk = new System.Windows.Forms.Button();
46:      this.BCancel = new System.Windows.Forms.Button();
47:      this.SuspendLayout();
48:      //  TBLogin
49:      this.TBLogin.Location = new System.Drawing.Point(160, 24);
50:      this.TBLogin.Name = "TBLogin";
51:      this.TBLogin.Size = new System.Drawing.Size(152, 22);
52:      this.TBLogin.TabIndex = 0;
53:      this.TBLogin.Text = "";
54:      // TBPassword
55:      this.TBPassword.Location = new  System.Drawing.Point(160, 64);
56:      this.TBPassword.Name = "TBPassword";
57:      this.TBPassword.Size = new  System.Drawing.Size(152, 22);
58:      this.TBPassword.TabIndex = 1;
59:      this.TBPassword.Text = "";
60:      // LLogin
61:      this.LLogin.Location = new  System.Drawing.Point(32, 24);
62:      this.LLogin.Name = "LLogin";
63:      this.LLogin.Size = new  System.Drawing.Size(104, 24);
64:      this.LLogin.TabIndex = 2;
65:      this.LLogin.Text = "Login";
66:      // LPassword
67:      this.LPassword.Location = new  System.Drawing.Point(32, 64);
68:      this.LPassword.Name = "LPassword";
69:      this.LPassword.Size = new  System.Drawing.Size(104, 24);
70:        this.LPassword.TabIndex = 3;
71:        this.LPassword.Text = "Password";
72:      // BOk
73:      this.BOk.Enabled = false;
74:      this.BOk.Location = new System.Drawing.Point(64, 104);
75:      this.BOk.Name = "BOk";
76:      this.BOk.Size = new System.Drawing.Size(80, 32);
77:      this.BOk.TabIndex = 4;
78:      this.BOk.Text = "Ok";
79:      // BCancel
80:      this.BCancel.DialogResult = System.Windows.Forms.DialogResult.Cancel;
```

```
81:      this.BCancel.Location = new  System.Drawing.Point(192, 104);
82:      this.BCancel.Name = "BCancel";
83:      this.BCancel.Size = new System.Drawing.Size(80, 32);
84:      this.BCancel.TabIndex = 5;
85:      this.BCancel.Text = "Cancel";
86:      // EventHandlers
87:      this.TBLogin.TextChanged += new System.EventHandler(this.OnTBTextChanged);
88:      this.TBPassword.TextChanged += new System.EventHandler(this.OnTBTextChanged);
89:      this.BOk.Click += new System.EventHandler(this.OnBOkClick);
90:      this.BCancel.Click += new System.EventHandler(this.OnBCancelClick);
91:      // LoginDialog
92:      this.AcceptButton = this.BOk;
93:      this.AutoScaleBaseSize = new System.Drawing.Size(6, 15);
94:      this.CancelButton = this.BCancel;
95:      this.ClientSize = new System.Drawing.Size(344, 160);
96:      this.Controls.AddRange(new System.Windows.Forms.Control[] {
97:          this.BCancel,
98:          this.BOk,
99:          this.LPassword,
100:         this.LLogin,
101:         this.TBPassword,
102:         this.TBLogin});
103:       this.Name = "LoginDialog";
104:       this.Text = "User/Password Validation";
105:       this.ResumeLayout(false);
106:     }
107:     #endregion
108:     [STAThread]
109:     static void Main()  {
110:       Application.Run(new LoginDialog());
111:     }

112:     private void OnTBTextChanged(object sender, System.EventArgs e) {
113:       BOk.Enabled = (TBLogin.Text != "" && TBPassword.Text != "")?true:false;
114:     }
115:     private void OnBCancelClick(object sender, System.EventArgs e)  {
116:       this.Close();
117:     }
118:     private void OnBOkClick(object sender, System.EventArgs e) {
119:       if (AliasToBL.Validate(TBLogin.Text, TBPassword.Text))   {
120:         MessageBox.Show("User Valid!");
121:       }
122:       else MessageBox.Show("User not valid!");
123:     }
124:   }
125:  }
```