# Os Métodos Formais na Análise de Orientação por Objectos

Ana M D Moreira* e Robert G Clark
*Department of Computing Science and Mathematics,*
*University Of Stirling,*
*STIRLING FK9 4LA, Scotland.*
*Email:* amm@uk.ac.stir.cs
*Email:* rgc@uk.ac.stir.cs

**Resumo**

O método ROOA (*Rigorous Object-Oriented Analysis*) introduz rigor no processo de análise orientado pelos objectos, oferecendo um conjunto de regras que permite produzir sistematicamente um modelo formal de análise orientado pelos objectos a partir dos requisitos originais. Este modelo, escrito na linguagem LOTOS, proporciona uma especificação precisa e não ambígua dos requisitos iniciais do sistema. Como a especificação é executável, podemos usar a prototipagem rápida para validar e refinar o modelo formal.

**Abstract**

The ROOA (Rigorous Object-Oriented Analysis) method introduces formality into the object-oriented analysis process by providing a set of rules which enables a formal object-oriented analysis model to be produced systematically from a set of requirements. This model is expressed in LOTOS and provides a precise and unambiguous specification of system requirements. As the specification obtained is executable, prototyping is used to support validation and refinement of the formal model.

## 1 Introdução

Desenvolver um sistema de *software* eficiente, fiável e de manutenção fácil, exige a adopção de uma estratégia que ajude os engenheiros de *software* a comunicar entre si sem ambiguidade. Quer dizer, os engenheiros de *software* que fazem o desenho devem ser capazes de perceber os resultados fornecidos pelos analistas e, ao mesmo tempo, ser capazes de dar aos programadores uma especificação sem ambiguidades, isto é, com uma só interpretação. Para isso ser possível é necessário dispor de um processo de desenvolvimento rigoroso. Este processo rigoroso deve incluir a construção de uma especificação formal que apresente os requisitos de um modo preciso e sem ambiguidade.

*Este trabalho tem sido apoiado pela Junta Nacional de Investigação Científica e Tecnológica (JNICT), Portugal.

Neste artigo, introduzimos o método ROOA (*Rigorous Object-Oriented Analysis*) [15] através de um exemplo. O método ROOA toma como ponto de partida as propriedades estáticas definidas num modelo de objectos produzido por qualquer método de análise orientado pelos objectos [5, 12, 16, 18] juntamente com as propriedades dinâmicas e funcionais dadas nos requisitos originais e produz uma especificação de requisitos orientada pelos objectos, formal e executável. As técnicas formais de descrição, tais como [1, 3, 13], são geralmente aplicadas depois da fase de análise de requisitos, mas neste artigo nós usámo-las para ajudar a determinar e compreender os requisitos do sistema. A técnica formal de descrição que escolhemos é o LOTOS (*Language Of Temporal Ordering Specification*) [1]. O LOTOS tem uma semântica matemática, precisa e formal e pode ser usado para escrever especificações num estilo orientado pelos objectos. O modelo formal resultante considera o sistema como um conjunto de objectos concorrentes onde a comunicação de mensagens é especificada por objectos que sincronizam em eventos durante a qual se pode trocar informação. A especificação resultante integra as propriedades estáticas e dinâmicas do sistema. Noutros métodos, estas propriedades são descritas por técnicas distintas, o que torna difícil garantir a coerência entre as diferentes descrições à medida que o modelo vai sendo desenvolvido.

Como a especificação obtida é executável, podemos usar a prototipagem para validar as diferentes especificações produzidas durante o processo de refinamento. Combinando o uso de técnicas formais de descrição com a prototipagem rápida durante a análise, podemos descobrir incoerências, omissões, contradições e ambiguidades mais cedo e assim corrigi-las nas fases iniciais do processo de desenvolvimento. A especificação formal de requisitos pode então ser transformada numa especificação formal de desenho. Finalmente, ao testar duas especificações executáveis com os mesmos cenários de interface podemos verificar se ambas exibem o mesmo comportamento externo do sistema.

Na Secção 2 discutimos a necessidade de especificações formais. Na Secção 3 introduzimos os métodos de análise orientados pelos objectos. Na Secção 4 descrevemos brevemente o método ROOA. Na Secção 5 apresentamos a linguagem de especificação LOTOS. Na Secção 6 mostramos de que maneira podemos usar o método ROOA para derivar uma especificação formal orientada pelos objectos, em LOTOS.

## 2 Especificações Formais e Executáveis

A vantagem principal das técnicas formais é que, tendo elas uma semântica precisa e matemática, as especificações resultantes são não ambíguas. Isto contrasta com as técnicas informais que conduzem a especificações cuja interpretação é, em parte, deixada ao critério do leitor. A imprecisão das especificações informais dá ao programador liberdade de interpretação, o que conduz a erros e omissões no código, resultando em custos de manutenção elevados, e a problemas de compreensão na validação da especificação informal com os requisitos (e na validação da implementação com a especificação). Por isso, é necessária uma aproximação formal para o desenvolvimento de especificações. Uma especificação formal de requisitos permite, pelo menos em teoria, que a implementação seja verificada "contra" a especificação, embora ainda deixe o problema de validar a especificação formal de requisitos "contra" o documento de requisitos original.

Provar que a especificação de requisitos, a especificação do desenho e a implementação descrevem exactamente o mesmo sistema está para lá do estado da arte. Uma aproximação expedita é tornar as especificações executáveis e depois fazer a validação por meio de testes de con-

formidade onde se usam uma série de cenários de interface para mostrar que as diferentes especificações e a implementação final exibem o mesmo comportamento.

Nem todos os engenheiros de *software* concordam em produzir especificações executáveis, porque este tipo de especificações contém forçosamente pormenores de implementação que estariam ausentes numa especificação não executável [9]. Todavia, se formos capazes de demonstrar que uma especificação exibe o comportamento esperado, a nossa confiança nela aumenta [8]. Podemos rebater a acusação de que isto não é mais do que testar a especificação recorrendo à avaliação simbólica. O simulador SMILE do LOTOS, por exemplo, permite o uso de variáveis não instanciadas dentro de condições e é capaz de determinar quando uma combinação de condições nunca pode ser verdadeira [6]. Assim, podemos examinar uma variedade maior de situações durante a simulação do que seria possível quando todos os valores das variáveis fossem instanciados.

# 3 Métodos de Análise Orientados pelos Objectos

O objectivo principal dos métodos de análise orientados pelos objectos é identificar os objectos e classes que constituem o sistema, compreender a estrutura e o comportamento dinâmico de cada objecto, reunir num mesmo local ("localizar") a informação que descreve cada um desses objectos e classes, e, ao mesmo tempo, mostrar como é que os objectos no sistema interactuam estaticamente e dinamicamente.

Em geral, os métodos de análise orientados pelos objectos partilham os seguintes passos:

1. Compreender os requisitos do utilizador.
2. Identificar e classificar objectos.
3. Definir objectos.
4. Identificar relações (associações) entre objectos.
5. Construir documentação.

Para compreender os requisitos do utilizador devemos ler o documento de requisitos inicial e consultar qualquer outra fonte de informação onde o problema, ou parte dele, seja descrito. Além disso, temos que entrevistar os utilizadores do sistema, é claro.

Para identificar e classificar objectos, alguns métodos propõem que se procurem substantivos, pronomes, adjectivos e advérbios, no documento de requisitos inicial, por exemplo [5, 18], enquanto outros sugerem que uma forma melhor de identificar objectos é identificar o seu comportamento no sistema [16].

Um objecto é definido em termos das suas visões estática e dinâmica. A visão estática é dada por uma lista dos seus atributos e operações. A visão dinâmica é geralmente descrita usando diagramas de transição de estado, mas desempenha um papel secundário na maior parte dos métodos. O conjunto de diagramas de transição de estado forma o modelo dinâmico do sistema.

As relações entre objectos podem ser estáticas ou dinâmicas. As relações estáticas representam-se pelo seu nome e cardinalidade. As relações dinâmicas constituem as mensagens de ligação e são representadas por setas que ligam o objecto que chama ao que é chamado. Estas relações aparecem no modelo de objectos, o qual é baseado no diagrama de entidades e associações

onde foram introduzidos melhoramentos para incorporar também agregados e herança. Alguns métodos juntam diagramas de sequência de mensagens ao modelo dinâmico para mostrar as interacções entre os objectos [18].

A documentação desempenha um papel crucial no desenvolvimento de *software*. Muitos métodos incluem explicitamente um passo para elaborar a documentação, enquanto noutros isso é apenas implícito.

Métodos mais recentes, tais como [18, 19], também incluem um modelo funcional o qual usa diagramas de fluxo de dados para descrever o significado das operações do modelo de objectos e o significado das acções do modelo dinâmico.

A maior vantagem das aproximações orientadas pelos objectos é que, como os conceitos usados são os mesmos, a passagem da análise para o desenho não é difícil. Além disso, os métodos de desenho produzem geralmente resultados muito próximos de código. Em situações em que a linguagem Ada, por exemplo, é usada como linguagem de desenho, os resultados produzidos na fase de desenho são na realidade especificações de pacotes em Ada.

# 4 O Método *Rigorous Object-Oriented Analysis*

O método ROOA (*Rigorous Object-Oriented Analysis*) envolve três tarefas principais. Na primeira, construímos o modelo de objectos. Na segunda, refinamos o modelo de objectos, normalizando-o e identificando hierarquias (de objectos). Finalmente, na terceira tarefa descrevemos um conjunto de subtarefas que devem ser levadas a cabo para construir o modelo de análise formal orientado pelos objectos.

O ROOA propõe uma aproximação paralela/recursiva, a qual nos permite reaplicar todo o método (ou apenas parte dele) aos resultados obtidos de uma iteração anterior. Além disso, podemos aplicar o método a partes diferentes do sistema (subsistemas) ao mesmo tempo (em paralelo, portanto). Em cada refinamento acrescentamos mais pormenor ao modelo formal.

### Tarefa 1: Construir o Modelo de Objectos

Antes de começarmos a produzir o modelo formal, temos que construir um modelo de objectos usando um dos métodos de análise orientados pelos objectos, por exemplo [5, 12, 16, 18]. A construção do modelo de objectos inicial pode ser considerada uma tarefa completamente separada das tarefas seguintes e pode mesmo ser realizada por uma equipa diferente. Durante a aplicação do método, é natural que o modelo de objectos venha a sofrer modificações.

A vantagem de começar com um modelo de objectos produzido por um qualquer método de análise é podermos aproveitar o trabalho que outros já desenvolveram para identificar objectos.

### Tarefa 2: Refinar o Modelo de Objectos

Aqui começamos por *normalizar* o modelo de objectos, garantindo que ele contém relações estáticas (associações), atributos, operações e mensagens de ligação e a seguir identificamos hierarquias de objectos, isto é, objectos de nível mais abstracto, de modo a tornar o sistema mais fácil de compreender e desenvolver. Esta tarefa é difícil, e por isso não podemos esperar completá-la correctamente durante a primeira iteração. Os objectos de nível abstracto mais baixo, na maioria dos casos, quase não sofrem modificações durante o desenvolvimento,

mas a estrutura de nível mais abstracto é menos estável. Nós identificamos apenas as hierarquias óbvias (isto é, agregados e herança) para começar e depois, como a aproximação é recursiva/paralela, voltamos a esta tarefa quando o nosso conhecimento acerca de cada objecto aumentar.

**Tarefa 3: Construir o Modelo Formal LOTOS**

Nesta tarefa criamos um diagrama de comunicação de objectos, especificamos objectos e classes, compomos os objectos em expressões de comportamento (em inglês, *behaviour expressions*) LOTOS, e prototipamos e refinamos a especificação.

1. Criar um diagrama de comunicação de objectos.
   Este diagrama é um grafo onde, na primeira iteração, um nó representa um objecto e cada arco que liga dois objectos representa uma porta (*gate*) de comunicação entre eles. Em iterações seguintes, o diagrama será generalizado para manipular múltiplos objectos da mesma classe. No início, alguns dos objectos podem não estar ligados por arcos ao resto do diagrama. À medida que o método é aplicado estes objectos serão ligados aos restantes e novos grupos aparecerão, refinando o diagrama.

   Podemos determinar, examinando as mensagens de ligação no modelo de objectos, se um objecto actua como cliente, servidor, ou como cliente e servidor. Se for um servidor, então todos os seus serviços (operações) para os seus diferentes clientes serão oferecidos numa só porta.

2. Especificar cada objecto e classe.
   Em geral, o comportamento de um objecto é especificado como um processo e o seu estado como um ou mais tipos abstractos de dados (*Abstract Data Types — ADTs*) que actuam como parâmetros do processo. Para cada objecto devemos começar por:

   (a) Especificar o processo identificando os eventos em que ele toma parte e a ordem por que ocorrem.

   (b) Especificar ADTs simbólicos para descrever os seus atributos.

   A herança no LOTOS é contudo uma questão mais problemática, como se pode apreciar pelo estudo teórico feito por Rudkin [17]. Há duas definições principais de herança [11]. Na herança de comportamento (*behavioural inheritance*), os objectos de uma subclasse oferecem os mesmos serviços que os objectos da superclasse e, além disso, podem sempre ser usados onde se espera um objecto da superclasse. Na herança incremental (*incremental inheritance*), uma subclasse herda a definição da sua superclasse e depois estende-a.

   Nós acreditamos que numa especificação, a herança de comportamento e a herança incremental devem ser restringidas a ser iguais. Embora o LOTOS não suporte herança directamente, não é difícil representar a herança incremental (apresentamos exemplos na Secção 6). A herança múltipla não é suportada.

3. Compor os objectos em expressões de comportamento LOTOS.
   Seguindo a estrutura do diagrama de comunicação de objectos, compomos os processos em expressões LOTOS por meio dos operadores de concorrência.

4. Prototipar a especificação.
   Usamos cenários de interface e prototipagem para verificar os serviços e as mensagens de ligação. Qualquer erro, omissão ou incoerência que encontremos levar-nos-á a voltar atrás a uma ou mais tarefas e actualizar o modelo de objectos, o diagrama de comunicação de objectos e a especificação.

5. Refinar a especificação.
   A especificação é refinada aplicando todo o processo de uma maneira recursiva/paralela. Ao longo dos sucessivos refinamentos podemos identificar novos grupos mais abstractos de objectos, definir geradores de objectos de modo a que múltiplas instâncias da mesma classe possam ser criadas dinamicamente, despromover um objecto para ser especificado apenas como um ADT, promover um objecto para ser especificado com um processo e refinar os processos e ADTs introduzindo mais pormenor.

## 4.1 Sistema Bancário Automático

O problema que escolhemos para ilustrar o nosso método é um sistema bancário automático. Apresentamos a seguir uma breve descrição do problema.

> Os clientes do banco podem levantar dinheiro das suas contas, fazer depósitos ou pedir o saldo corrente. Todas estas operações são completadas usando as caixas automáticos (*automatic teller machines - ATM*) ou ao balcão (*counter teller*). As transacções numa conta fazem-se por cheque, ordens de pagamento (*standing order*), ou então usando a ATM com um cartão. Há dois tipos de contas: à ordem (*cheque account*) e a prazo (*savings account*). Uma conta a prazo dá juros e não pode ser acedida pelas ATMs.

Aplicámos a este problema os métodos de análise orientados pelos objectos OOA [5] e OMT [18], mas apresentamos apenas o modelo de objectos final produzido por este último, usando a sua simbologia (ver Figura 1). Este modelo de objectos mostra as classes de objectos com os seus atributos e as relações entre os objectos.

# 5 Introdução ao LOTOS

O LOTOS é uma técnica formal de descrição desenvolvida pela ISO [2] para a definição de sistemas OSI (*Open Systems Interconnection*) normalizados, embora também seja adequada para desenvolver especificações para uma larga gama de sistemas, incluindo sistemas embutidos (*embedded*) [4]. Tem duas componentes principais:

- Definição de processos: esta componente descreve o comportamento dos processos e as interacções entre eles. Baseia-se no CSP [10] e no CCS [14].
- Tipos abstractos de dados: esta componente descreve os tipos de dados. Baseia-se na linguagem de tipos abstractos de dados ACT ONE [7].

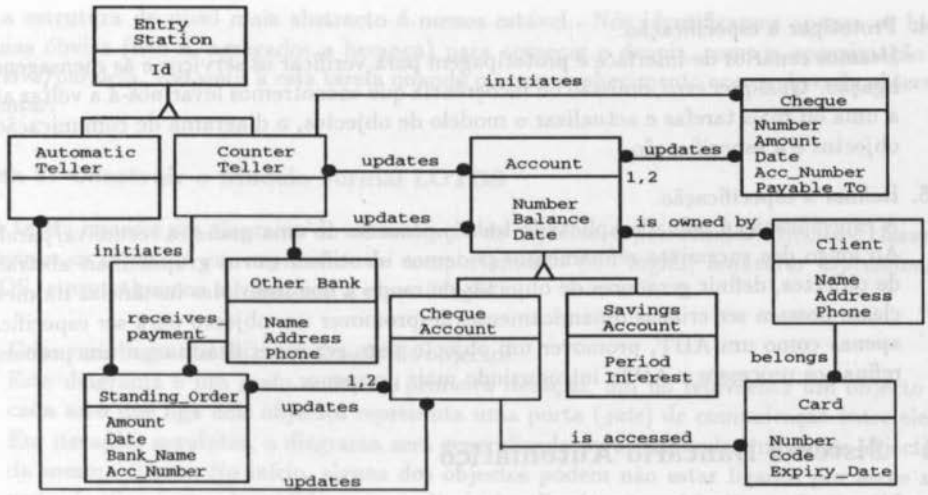


Figura 1: Modelo de objectos produzido pelo método OMT

## 5.1 Processos

Um sistema concorrente distribuído descreve-se por meio de um conjunto de processos comunicantes. Um processo é uma "caixa preta" e o seu comportamento externo observável é determinado pelas interacções que o processo tem com outros processos. Especificar um processo é definir a relação temporal entre essas interacções.

O comportamento de um processo descreve-se por meio de expressões de comportamento que consistem em acções externas observáveis, e acções internas não-observáveis. As interacções entre processos são conseguidas através de sincronização. Uma sincronização é um evento. Um evento é atómico e ocorre numa porta.

Consideremos, como exemplo, o modelo de objectos representado na Figura 1. Tal como dissemos na secção anterior, um objecto é especificado através de um processo e de um ou mais ADTs. O processo descreve o comportamento dinâmico do objecto e os ADTs descrevem o estado do objecto. Supondo que `Account` oferece os serviços `deposit` para creditar uma conta e `balance` para dar o saldo de uma conta, o processo poderia ser especificado assim:

```
process Account[g](this_account: Account): noexit :=
  ( g !deposit !Get_Account_Number(this_Account) ?m: Money;
    exit(Credit_Account(this_account, m))
  []
    g !balance !Get_Account_Number(this_account) !Get_Balance(this_account);
    exit(this_account)
  ) >> accept updated_account: Account in Account[g](updated_account)
endproc
```

O processo é definido recursivamente e usa a porta g para sincronizar com outros processos. Comunica com os outros objectos do sistema enviando mensagens, as quais são representadas como eventos com a seguinte estrutura:

*<nome porta> <nome mensagem> <identificador objecto> <parametros opcionais>*

Por exemplo, `Counter_Teller` envia uma mensagem para `Account` pedindo um depósito:

```
g !deposit !acc_number !amount;
```

uma instância de `Account` sincroniza com este evento oferecendo:

```
g !deposit !Get_Account_Number(this_Account) ?m: Money;
```

O operator "!" usa-se na forma *!v* onde *v* é uma expressão valor. O operator "?" usa-se na forma *?v: s* onde *v* é uma variável do *sort s*.

Há três tipos de sincronização, que resumimos na Tabela 1.

| Processo A | Processo B | Condiçcão | Tipo de Interacção |
|---|---|---|---|
| g ! $E_1$ | g ! $E_2$ | valor ($E_1$) = valor ($E_2$) | correspondência de valores |
| g ! $E_1$ | g ? x: s | *sort* ($E_1$) = s | passagem de valores |
| g ? y: w | g ? x: s | w = s | geração de valores |

Tabela 1: Tipos de Interacção

A correspondência de valores entre `Get_Account_Number(this_account)` e `acc_number` usa-se para garantir a sincronização entre os processos correctos. Embora um cliente tenha que conhecer o identificador do seu servidor, um servidor pode servir muitos clientes sem saber a identidade desses clientes. A passagem de valores usa-se para transmitir o valor `amount` à variável `m`. A geração de valores permite introduzir variáveis ainda não-instanciadas.

O símbolo `[]` representa o operador de escolha não-determinística e o símbolo `>>` representa o operador *enable*. A expressão `A>>B` significa que a seguir à terminação com êxito do processo `A` começa a execução do processo `B`. O operador `accept ... in` usa-se para passar valores à medida que saímos de um processo e entramos noutro.

As funções `Get_Account_Number`, `Credit_Account` e `Get_Balance` definem-se no ADT correspondente. O parâmetro `this_account` representa a informação de estado do objecto e é actualizado pela chamada recursiva.

## 5.2 Tipos Abstractos de Dados

O LOTOS usa a linguagem ACT ONE para especificar os tipos abstractos de dados. Geralmente, essas especificações são longas e complexas, embora se possam tornar mais simples recorrendo à biblioteca de ADTs pré-definidos que o LOTOS oferece.

O exemplo seguinte define um ADT que representa o estado de informação de uma `Account`:

```
type Account_Type is Account_Number_Set_Type, Money_Type, Balance_Type
  sorts Account
  opns Init_Account      : Account_Number -> Account
       Credit_Account    : Account, Money -> Account
       Get_Balance       : Account        -> Balance
       Get_Account_Number : Account       -> Account_Number
       ...
  eqns forall a: Account, n: Account_Number, m: Money, ...
    ofsort Account_Number
       Get_Account_Number(Init_Account(n))     = n;
```

```
        Get_Account_Number(Credit_Account(a,m)) = Get_Account_Number(a);
        ...
    ofsort Balance
        Get_Balance(a) = Some_Balance;
endtype
```

Depois da palavra chave **is**, aparece a lista de definições importadas. A secção **sorts** indica o nome dos *data sorts*, a secção **opns** define as operações por meio da sua assinatura e a secção **eqns** especifica, em termos de equações, os constrangimentos que as operações têm que satisfazer. Na secção **eqns forall** declaram-se as variáveis que vão ser usadas nas equações e na secção **ofsort** define-se o *sort* resultado das equações e a seguir define-se as próprias equações.

No ROOA usamos ADTs *simbólicos*. Um ADT simbólico inclui as equações necessárias para permitir simular os objectos com informação de estado e passagem de valores durante a comunicação, mas sem dar demasiado pormenor acerca de como as operações são realizadas internamente.

# 6 Exemplo de utilização do Método ROOA

Nesta secção vamos mostrar como se usa este método e para isso utilizamos o exemplo do sistema bancário automático introduzido na Secção 4.

## Tarefa 1: Construir o Modelo de Objectos

O modelo de objectos produzido pelo [18] está representado na Figura 1.

## Tarefa 2: Refinar o Modelo de Objectos

Como o modelo de objectos construído na tarefa anterior apenas inclui os atributos e as relações entre os objectos, é preciso normalizá-lo, adicionando-lhe serviços e mensagens de ligação que sejam relativamente "óbvios". Para identificar as mensagens de ligação, podemos usar cenários de interface. Estes cenários modelam as interacções do sistema com o seu ambiente externo. À medida que fazemos o seguimento (*trace*) dos "caminhos" de funcionalidade do sistema, no modelo de objectos, vamos criando mensagens de ligação. Por exemplo, o nosso sistema tem que manipular contas bancárias nas quais se podem fazer depósitos e levantamentos. Sendo assim, as operações `deposit` e `withdraw` correspondem a eventos no cenário de interface. Temos então que garantir que estes serviços são oferecidos pelo sistema, fazendo dos eventos operações que devem ser atribuídas aos objectos apropriados.

Durante a aplicação desta tarefa, apercebemo-nos que algumas das relações no modelo de objectos inicial eram na realidade mensagens de ligação. Na Figura 2 apresentamos o modelo de objectos normalizado. Os serviços representam-se na parte inferior de cada classe de objectos, as mensagens de ligação representam-se por setas, e as duas hierarquias óbvias estão marcadas a tracejado. Elas correspondem à herança definida para *tellers* e *accounts*.

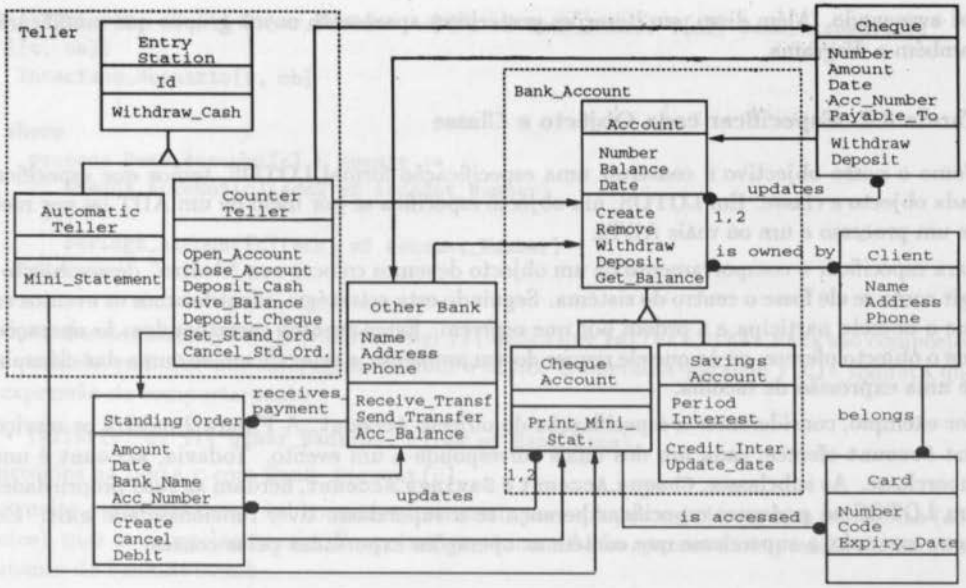


Figura 2: Modelo de objectos normalizado

## Tarefa 3: Construir o Modelo Formal LOTOS

### Tarefa 3.1: Criar o Diagrama de Comunicação de Objectos

Cada objecto no modelo de objectos é um nó no diagrama de comunicação de objectos. Neste diagrama temos que mostrar as hierarquias identificadas na tarefa anterior.

`Teller` e `Other_Bank` são os primeiros clientes no sistema. `Cheque` e `Standing_Order` desempenham simultaneamente o papel de servidores em relação a `Teller` e o de clientes de `Bank_Account`. `Bank_Account` é o servidor final no sistema e por isso apenas usa a porta `c` para comunicação (ver Figura 3).

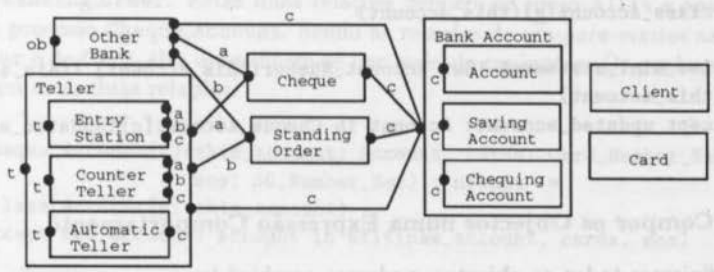


Figura 3: Diagrama inicial de comunicação de objectos

Note-se que os objectos `Card` e `Client` não estão ligados ao resto do sistema. É comum isto acontecer na primeira iteração, mas corrigir-se-á à medida que a aplicação do método

vai avançando. Além disso, em iterações posteriores aparecerão novos grupos que modificarão também o diagrama.

### Tarefa 3.2: Especificar cada Objecto e Classe

Como o nosso objectivo é construir uma especificação formal LOTOS, temos que especificar cada objecto e classe. Em LOTOS, um objecto especifica-se por meio de um ADT ou por meio de um processo e um ou mais ADTs.

Para especificar o comportamento de um objecto devemos colocar-nos "dentro" desse objecto e agir como se ele fosse o centro do sistema. Seguindo esta estratégia, identificamos os eventos em que o objecto participa e a ordem por que ocorrem. Estes eventos correspondem às operações que o objecto oferece, ou às que ele requer do seu ambiente, e especificam-se numa das cláusulas de uma expressão de escolha.

Por exemplo, consideremos a especificação do objecto `Account`. A Figura 2 mostra os serviços que `Account` oferece, cada um dos quais corresponde a um evento. Todavia, `Account` é uma superclasse. As subclasses, `Cheque_Account` e `Savings_Account`, herdam as suas propriedades. Em LOTOS só podemos especificar herança se a superclasse tiver funcionalidade **exit**. Eis, como exemplo, a superclasse que contém as operações exportadas pelas contas:

```
process Superclass_Account[c](this_account: Account): exit(Account) :=
   c !deposit !Get_Account_Number(this_Account) ?m: Money;
   exit(Credit_Account(this_account, m))
 []
   c !balance !Get_Account_Number(this_account) !Get_Balance(this_account);
   exit(this_account)
 []
   ...
endproc
```

A definição do ADT relativo a `Account` foi representada na Secção 5.2. A subclasse `Cheque_Account` define-se assim:

```
process Cheque_Account[c](this_account: Account) : noexit :=
  ( Superclass_Account[g](this_account)
  []
    c !print_mini_statement !Get_Account_Number(this_account) !this_account;
    exit(this_account)
  ) >> accept updated_account: Account in Cheque_Account[c](updated_account)
endproc
```

### Tarefa 3.3: Compor os Objectos numa Expressão Comportamento

Depois de definirmos todos os objectos, podemos combiná-los numa expressão de comportamento LOTOS. Esta expressão pode descrever o sistema todo ou apenas parte dele. É legítimo, por exemplo, ignorar `Cheque` e `Standing_Order` para começar. A expressão de comportamento principal tomária a forma:

```
((Teller[t, c] ||| Other_Bank[ob, c](bk of Bank_Name)) |[c]| Bank_Account[c])
|[t, ob]|
 Interface_Scenario[t, ob]

where
  process Bank_Account[c] : noexit :=
    Cheque_Account[c](acc2 of Account_Number)
   |||
    Savings_Account[c](acc1 of Account_Number)
  where ...
```

O operador de entrelaçamento (*interleaving*) ||| indica que `Teller` e `Other_Bank` são compostos em paralelo, mas que não interagem um com o outro. O operador paralelo |[c]| significa que a expressão de comportamento

```
Teller[t, c] ||| Other_Bank[ob, c](bk of Bank_Name)
```

sincroniza na porta `c` com `Bank_Account[c]`.

Na maior parte dos casos, tal como acontece neste exemplo, as subclasses têm instâncias (objectos), mas as superclasses não. É por isso que `Account` não aparece na expressão de comportamento de `Bank_Account`.

### Tarefa 3.4: Prototipar a Especificação

A prototipagem é usada não só para corrigir erros de sintaxe e de semântica, mas também para validar a especificação contra o modelo de objectos e contra os requisitos. Aqui usamos cenários de interface para ajudar a prototipar a especificação.

### Tarefa 3.5: Refinar a Especificação

Vejamos como tratar as relações em LOTOS. Uma relação pode ser especificada como um atributo, ou como um conjunto de atributos, num dos objectos envolvidos na relação (ou nos dois objectos, se a relação for bidireccional) [15]. Consideremos como exemplo o objecto `Cheque_Account`. Como podemos verificar pelo modelo, este objecto tem uma relação com `Card` e outra com `Standing_Order`. Estas duas relações definem-se como ADTs e aparecem como parâmetros do processo `Cheque_Account`. Sendo as relações de *um-para-muitos* na direcção de `Standing_Order` e de `Card`, elas especificam-se por meio de conjuntos. Os parâmetros `cards` e `sos` representam estas duas relações.

```
process Cheque_Account[c](this_account: Account, cards: Card_Number_Set,
                          sos: SO_Number_Set) : noexit :=
  ( Superclass_Account[g](this_account)
    >> accept new_account: Account in exit(new_account, cards, sos)
  []
    c !print_mini_statement !Get_Account_Number(this_account) !this_account;
    exit(this_account, cards, sos)
  []
    c !perhaps_deposit !Get_Account_Number(this_account) ?m: Money;
    exit(Credit_Pending(this_account, m), cards, sos)
```

```
  []
    c !full_deposit !Get_Account_Number(this_account) ?m: Money ?valid: Bool;
    ( [valid] -> exit(Add_Credit_Pending(this_account, m), cards, sos)
     []
      [not (valid)] -> exit(Sub_Credit_Pending(this_account, m), cards, sos)
    )
  ) >> accept upd_account: Account, cards: Card_Number_Set, sos: SO_Number_Set
       in Cheque_Account[c](upd_account, cards, sos)
endproc
```

Saliente-se que a nova versão de Cheque_Account aumenta as operações e o *estado de informação* que herda de Superclass_Account. Além disso, ao especificarmos relações assim, como ADTs parâmetros do processo, em vez de as especificarmos no ADT que define o estado do objecto, estamos a promover a reutilização.

Durante esta tarefa introduzimos mais informação em alguns dos processos, de modo a estudar completamente todos os outros processos no sistema. (Lembremo-nos, por exemplo, que durante a primeira iteração decidimos não considerar os objectos Cheque e Standing_Order na expressão de comportamento que representava o sistema.) Ao definirmos cada objecto com mais pormenor, em conjunto com os restantes no sistema, encontramos novas operações. Por exemplo, as operações perhaps_deposit e full_deposit foram acrescentadas para permitir manipular os cheques com mais generalidade. Neste passo, também decidimos que Card e Client deveriam ser especificados como ADTs, e não como processos.

No caso mais geral, cada classe dará origem a vários objectos. Estes objectos devem ser depois criados dinamicamente. É o que acontece, por exemplo, com Cheque_Account. Neste caso, precisamos de definir um gerador de objectos:

```
process Cheque_Accounts[c](accs: Account_Number_Set): noexit :=
  c !create !cheque ?acc_counter: Account_Number
     [(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
  ( Cheque_Account[c](Init_Account(acc_counter),
                      {} of Card_Number_Set, {} of SO_Number_Set)
  |||
    Cheque_Accounts[c](Insert(acc_counter,accs))
  )
endproc
```

O generador de objectos guarda o conjunto de identificadores já atribuídos e o predicado de selecção:

```
[(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
```

impõe que o identificador do novo objecto seja diferente de todos os já existentes. Como os dois tipos de contas (à ordem e a prazo) partilham o mesmo *sort* Account_Number, !cheque especifica o tipo de conta que queremos criar e Is_Cheque_Acc(acc_counter) garante que o identificador do novo objecto pertence ao intervalo correcto de Account_Number.

Estes refinamentos levaram às modificações introduzidas no diagrama de comunicação de objectos representado na Figura 4.

A expressão de comportamento do sistema fica agora definida assim:

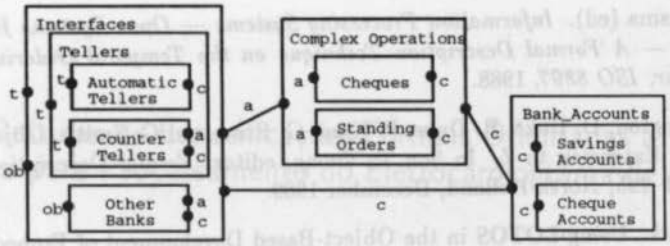


Figura 4: Diagrama de comunicação de objectos final

```
((Interfaces[t, ob, a, c] |[a]| Complex_Operations[a, c])
 |[c]| Bank_Accounts[c]
)
|[t, ob]| Interface_Scenario[t, ob]
```

# 7 Conclusões

O método ROOA (*Rigorous Object-Oriented Analysis*) integra num só modelo formal o modelo de objectos, o modelo dinâmico e o modelo funcional que os métodos informais de análise orientados pelos objectos geralmente propõem. Como a linguagem LOTOS tem uma semântica precisa e matemática, o modelo resultante é formal e não ambíguo. Além disso, sendo o LOTOS executável, o modelo também o é, podendo usar-se a prototipagem para dar *feedback* imediato aos clientes, os quais ficam em condições de apreciar se o protótipo exibe o comportamento esperado. A prototipagem de especificações formais permite identificar mais cedo omissões e incoerências que porventura existam nos requisitos originais. Esta técnica também suporta uma trajectória de desenvolvimento de *software* onde a especificação de requisitos se pode transformar numa especificação de desenho, com a prototipagem a ser usada para assegurar que as duas especificações exibem o mesmo comportamento externo do sistema.

Em LOTOS, o comportamento dinâmico de cada objecto especifica-se através de um processo e o seu estado de informação através de um ou mais ADTs. Note-se que o comportamento dinâmico de todo o sistema se obtém compondo todos os processos, por meio dos operadores de concorrência do LOTOS. Deste modo, o sistema surge como um conjunto concorrente de objectos, e isso permite-nos evitar tomar logo decisões que podem ser consideradas de desenho ou de implementação (tais como técnicas de protecção de acesso concorrente a dados partilhados). Uma grande parte da concorrência agora existente na especificação será retirada durante as fases de desenvolvimento posteriores, mas relembremos que estamos ainda na fase de análise, o que significa que o nosso objectivo principal é perceber o problema e não propor uma solução para ele desde já.

# References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[2] E. Brinksma (ed). *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique on the Temporal Ordering of Observation Behaviour, ISO 8807*, 1988.

[3] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In Son T. Vuong, editor, *Formal Description Techniques, II*, pages 281–295, North-Holland, December 1989.

[4] R.G. Clark. Using LOTOS in the Object-Based Development of Embedded Systems. In C.M.I. Rattray and R.G. Clark, editors, *Unified Computation Laboratory*, pages 307–319. Oxford University Press, 1992.

[5] P. Coad and E. Yourdon. *Object Oriented Analysis.* Yourdon Press, Prentice-Hall, 2nd edition, 1991.

[6] H. Eertink. Executing LOTOS Specifications: The SMILE Tool. In *Third LotoSphere Workshop and Seminar*, September 1992.

[7] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications*, volume 1. Springer-Verlag, 1985.

[8] N.E. Fuchs. Specifications are (preferably) Executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[9] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, November 1989.

[10] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[11] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing. Technical report, 1993.

[12] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach.* Addison-Wesley, 1992.

[13] C.B. Jones. *Software Development. A Rigorous Approach.* Prentice Hall, 1980.

[14] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[15] A.M.D. Moreira and R.G. Clark. Rigorous Object-Oriented Analysis. Technical Report TR 109, Computing Science Department, University of Stirling, Scotland, 1993.

[16] K.S. Rubin and A. Goldberg. Object Behaviour Analysis. *Communications of the ACM*, 35(9):48–62, 1992.

[17] S. Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, pages 409–423, North-Holland, 1992.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall, 1991.

[19] S. Shlaer and S.J. Mellor. *Object Lifecycles — Modeling the World in States.* Prentice-Hall, 1992.