UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CLAUDIO NAOTO FUZITAKI

Mapeamento da linguagem Nautilus para Java

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Paulo Blauth Menezes
Orientador

Porto Alegre, setembro de 2004

CIP – CATALOGACÃO NA PUBLICACÃO

Fuzitaki, Claudio Naoto

Mapeamento da linguagem Nautilus para Java / Claudio Naoto Fuzitaki. – Porto Alegre: PPGC da UFRGS, 2004.

139 f.: il.

Dissertacão (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduacão em Computa cão, Porto Alegre, BR–RS, 2004. Orientador: Paulo Blauth Menezes.

1. Linguagem Nautilus. 2. Teoria das Categorias. 3. Automatos não sequenciais. I. Menezes, Paulo Blauth. II. Título.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Paulo Blauth Menezes, e aos meus colegas do LFC e do Instituto de Informática (Júlio, Karina, Marnes, Rodrigo, entre outros) pelo apoio e também pelas críticas que permitiram melhorar meu trabalho. Agradeço especialmente aos meus pais pela paciência. E a Deus pela oportunidade.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Este trabalho apresenta um mapeamento centrado nas construções não usuais da linguagem Nautilus, para uma linguagem convencional, no caso Java, mantendo propriedades como atomicidade que são requisitos da semântica formal da linguagem.

Nautilus é originalmente uma linguagem de especificação baseada em objetos, textual que suporta objetos concorrentes e não deterministas. Desde então a linguagem foi modificada com extensões como classes e uma notação diagramática, além de se investigar seu uso como linguagem de programação.

Suas construções incomuns (reificação, agregação, etc.) são baseados em seu domínio semântico: Automâtos Não Sequenciais. Este domínio satisfaz composição diagonal, i.e refinamentos se compõem (verticalmente) refletindo uma descrição gradual de sistemas, envolvendo múltiplos níveis de abstração, e distribui-se através de combinadores (horizontalmente), o que significa que o refinamento de um sistema composto é a composição de do refinamento de suas partes.

O trabalho inclui um mapeamento inicial de um subconjunto da linguagem (objeto base, reificação, agregação e visão), uma versão ampliada para abranger mais construções (interação e classes), e uma versão refinada mais concorrente e sugestões de modificações na linguagem.

Palavras-chave: Linguagem Nautilus, Teoria das Categorias, Automatos não sequenciais.

Mapping Nautilus Language into Java

ABSTRACT

This work presents a mapping of the uncommon constructions from the Nautilus language to a conventional language, in this case Java, preserving proprieties like atomicity that are required by its formal semantics.

Nautilus is originally a object based, textual specification language and supports concurrent objects. Since then the language was modified with extensions as classes and a diagrammatic notation, besides investigating its use as programming language.

Its uncommon constructions (refinements, aggregation, etc.) are based in its semantic domain: Nonsequential Automata. This domain satisfies the diagonal compositional requirement, i.e. refinements compose (vertically), reflecting a step-wise description of systems, involving several levels of abstraction, and distributes through combinators (horizontally), meaning that the refinement of a composite system is the composition of the refinement of its parts.

The work includes a initial mapping of a subset of the language (base object, reification, aggregation, and vision), a version that covers more constructions (interaction and classes), a refined version with more concurrency and sugestions of modifications in the language.

Keywords: Nautilus Language, Category Theory, Non Sequential Automata.

1 INTRODUÇÃO

Este trabalho têm como objetivo principal mapear as construções não usuais da linguagem Nautilus, tais como agregação, reificação, etc. (detalhados na próxima seção), para uma linguagem convencional mantendo propriedades, como atomicidade, que são requisitos da semântica formal de Nautilus.

O resto deste capítulo está organizado da seguinte forma: inicialmente é fornecido uma caracterização da linguagem Nautilus onde a necessidade do mapeamento para uma linguagem convencional é contextualizada, depois seguem-se os objetivos de forma mais detalhada e uma curta descrição do resto da dissertação.

1.1 A Linguagem Nautilus

A linguagem Nautilus, originalmente introduzida em (MENEZES; SERNADAS; COSTA, 1995), foi inspirada na linguagem GNOME (RAMOS; SERNADAS, 1995; SERNADAS; RAMOS, 1994), que por sua vez é uma simplificação e revisão da linguagem orientada a objetos OBLOG (SERNADAS et al., 1991; SERNADAS; GOUVEIA; SERNADAS, 1992; SERNADAS et al., 1992). É uma linguagem de alto nível, originalmente baseada em objetos, textual e com facilidades de abstração inspiradas no seu domínio semântico.

A linguagem Nautilus desenvolve de forma natural os mecanismos de concorrência e de sincronização, o que inibe o surgimento de um estilo de programação demasiadamente baseado em construções seqüenciais. Entre as suas principais características, destacam-se: Concorrência (MILNER, 1989; WINSKEL, 1984) e Não-Determinismo. A linguagem é naturalmente concorrente e não-determinista. Como conseqüência, os problemas normalmente existentes em linguagens de programação tradicionais onde a programação, depuração e manutenção são atividades complicadas e delicadas normalmente não ocorrem em Nautilus. Um interessante exemplo foi um teste feito com bolsistas de iniciação científica onde foi solicitado que desenvolvessem um programa em Nautilus que atingisse uma situação de deadlock ao longo de sua execução. Após alguns minutos, os alunos afirmaram que não seria possível fazer tal programa em Nautilus. Foi solicitado que pensassem no problema por mais alguns dias e, de fato, conseguiram fazer tal programa. Mas a solução encontrada foi "emular" construções de linguagens tradicionais, que induzem de forma não-natural um programa Nautilus ao *deadlock*.

As duas principais formas de reutilização de código em Nautilus são:

- Reificação (CORRADINI, 1990; MENEZES, 1997). Trata-se de um mecanismo de abstração transacional, no sentido em que uma ação de alto nível

é implementada, de forma indivisível e em um tempo finito, em composições seqüenciais ou concorrentes (ou ambas) de ações de baixo nível. A reificação é verticalmente composicional (reificações podem ser sucessivamente compostas) e pode ser dependente de estado tanto de forma explícita como implícita (MENEZES; MACHADO; COSTA, 2002). A reificação em Nautilus só foi possível devido aos resultados teóricos atingidos referentes a composicionalidade diagonal da reificação sobre a composição paralela nos Autômatos Não-Seqüenciais, os quais constituem o domínio semântico da linguagem;

- Agregação. É uma facilidade não-tradicional na maioria das linguagens de programação e é inspirada em conceitos de construção de sistemas de hardware onde as portas de entrada e saída dos objetos ("chips") são interligados por um objeto agregador ("placa e suas trilhas"), respeitando a compatibilidade de sinais. Um objeto resultante de uma agregação ("placa montada"), por sua vez, pode ser considerado como um novo componente genérico. Ou seja, como todas as construções em Nautilus, agregações são composicionais entre si e com as demais construções da linguagem. Em Nautilus um objeto pode ser especificado como um objeto simples (objeto cuja construção não depende de outros objetos) ou como objeto resultante de uma encapsulação, agregação, reificação ou composição paralela. Uma ação de um objeto em Nautilus pode ser uma composição seqüencial ou concorrente de cláusulas, executadas atomicamente.

As características aqui apresentadas conduzem a uma nova abordagem para o tratamento de problemas, pois permite ao estudante desenvolver sistemas tratando-os como uma unidade, ao invés de construí-los em pequenas etapas para após constituir um sistema complexo. Em um curso de Ciência da Computação, o aluno necessita começar a pensar em mais alto nível, mas acaba encontrando obstáculos a esta mudança de raciocínio devido à falta de prática e aos vícios das linguagens usuais. Implementando ações de objetos em mais alto nível, utilizando transações simples e não-deterministas permite ao programador alcançar ações mais detalhadas e elaboradas, cujo comportamento depende somente das condições internas do sistema atual.

A linguagem Nautilus está voltada para o ensino de noções de programação de mais alto nível do que as encontradas na maioria das linguagens atuais. Tal nível de abstração é mais bem aplicado no estudo da concorrência e/ou paralelismo que são usualmente implementados em um "baixo nível", com instruções que poderiam ficar "escondidas"do programador. Por exemplo, para implementar alguns programas em Java, os programadores precisam entender conceitos como threads e monitores. Em Nautilus não precisaria tais conceitos. Outro exemplo está no reuso de código e composição de programas, que não são fornecidos no mesmo nível que em Nautilus.

Conforme discutido em (CARNEIRO et al., 1999), Nautilus pode propiciar pessoas que se tornem mais aptas a pensar em sistemas concorrentes de alta complexidade. Esta linguagem capacita o estudante a desenvolver algoritmos complexos e detalhados permanecendo em um alto nível de abstração (ou não, caso desejado).

Diversas extensões tais como classes, operações de restrições e uma sintaxe diagramática (D'ANDREA et al., 2002) foram acrescentadas a especificação inicial da linguagem.

Portanto um foco interessante de pesquisa é usar Nautilus como uma uma pri-

meira linguagem de programação para alunos de Ciências da Computação. Para dar prosseguimento nesta linha seria importante ter um ambiente que permitisse efetivamente executar os programas Nautilus. Assim, vê-se como necessária a construção de um ambiente de execução de programas Nautilus e documentação desse processo. Dentro deste contexto, apesar do foco acadêmico, leva-se em consideração outras possibilidades visando um uso mais geral (por exemplo, melhora na usabilidade propondo formas de recursão e um uso mais intensivo de concorrência).

1.2 Objetivos e Resultados

O objetivo de alto nível é um mapeamento Nautilus-Java que possa ser usado na implementação de um ambiente de desenvolvimento de programas Nautilus, mais especificamente, um tradutor de Nautilus para Java.

Tendo este objetivo, este projeto têm como subobjetivos: selecionar um subconjunto da linguagem Nautilus, estabelecer aspectos importantes ainda não detalhados devido a linguagem ser de especificação (tais como operações de entrada/saída, tipos primitivos, etc.), verificar como mapear este subconjunto para uma linguagem base mais convencional e construir um protótipo. E de acordo com o andamento dos trabalhos, ampliar o escopo inicial do subconjunto da linguagem e propor a adição de novas construções à linguagem.

De forma mais concreta, os resultados são a publicação de artigos descrevendo a modelagem e protótipos que permitam a execução de programas Nautilus.

Relativamente a implementação de um ambiente de execução, o trabalho proposto é atuar principalmente no nível semântico mapeando as construções não usuais de Nautilus (reificação, agregação, etc.) para Java mantendo as propriedades do domínio semântico (ações atômicas, composição).

1.3 Descrição de capítulos

No capítulo 2, um subconjunto da linguagem representando o núcleo de Nautilus é selecionado, verifica-se como mapear este conjunto para uma linguagem base mais convencional (Java) e um protótipo foi construído para validar o mapeamento inicial baseado em um foco mais acadêmico.

No capítulo 3, verifica-se a possibilidade de ampliar o subconjunto da linguagem adicionando-se interação e classes, um mapeamento para Java e possíveis problemas.

No capítulo 4, o mapeamento do núcleo é refeito buscando-se um maior grau de concorrência visando um uso mais geral. Neste capítulo usou-se CSP como uma notação mais abstrata visando auxiliar a tarefa. Ao final do capítulo é esboçado um mapeamento Nautilus-CSP simplificado.

O capítulo 5 apresenta propostas e considerações sobre modificações na linguagem. Algumas podem ser usadas como base inicial para trabalhos futuros.

O capítulo 6 apresenta um sistema de checagem de protocolos para parâmetros. É uma solução elegante para os problemas de implementação decorrentes da flexibilidade do sistema de parâmetros de Nautilus.

O capítulo 7 fala de aspectos de implementação. Algumas escolhas que foram feitas, otimizações que poderiam ser feitas e limitações/problemas diversas.

Por fim, são apresentadas as conclusões finais.

Em anexo, estão três artigos (FUZITAKI et al., 2003; FUZITAKI; MENEZES;

MACHADO, 2004; FUZITAKI et al., 2004) e um tutorial. Os dois primeiros artigos foram publicados e o terceiro foi submetido. O último anexo é um tutorial da linguagem Nautilus desenvolvido para pessoas com pouca experiência em programação (apresenta a linguagem baseada em objetos Nautilus como definida em (MENEZES, 1997)).

2 MAPEAMENTO DO NÚCLEO

2.1 De Nautilus para Java

O objetivo deste documento é apresentar propostas de tradução de programas da linguagem Nautilus para programas Java (escolhida como linguagem base por causa da experiência prévia de um colaborador (BARBOSA et al., 2001)). A linguagem Nautilus é uma linguagem concorrente baseada em objetos com semântica formal bem definida com diversas construções não usuais. Para o leitor não familiarizado com Nautilus recomenda-se o tutorial em anexo.

O foco deste capitulo é basicamente em um subconjunto da linguagem, chamado de núcleo, praticamente como foi apresentada em (MENEZES, 1997) exceto interações que são apresentadas no próximo capítulo.

2.2 Tipos de Dados Básicos

Como Nautilus foi criada originalmente para ser uma linguagem de especificação, seus tipos de dados não foram muito bem definidos. Uma vez que a questão é quase ortogonal a linguagem optou por utilizar-se dos tipos primitivos utilizados em algumas especificações, tais como natural, boolean, etc., mapeando-os diretamente para tipos primitivos Java. A questão de tipos de dados compostos será tratada posteriormente.

Nautilus	Java
natural	int
integer	int
boolean	boolean
string	String
enumeração (id1, .., idn)	Não existe enum em java mas é possível emular a construção usando classes, conforme mostrado em (BLOCH, 2003)

2.3 Entrada/Saída com o usuário

Foi decidido que o subconjunto inicial da linguagem a ser implementada não utilizaria interações, que é a maneira mais usual em linguagens orientadas a objeto de trabalhar com entrada, uma vez que esta construção não é aconselhada dentro do estilo de programação de Nautilus (da mesma forma que GOTO para programação estruturada) e qualquer programa que a use pode ser escrita usando outras construções.

```
object Obj1
...
body
  slot var1:⟨type⟩
  act A1
    enb ⟨condition⟩
    ⟨body  action⟩
  ...
end Obj1
```

```
class TObj1 extends Thread {
  ⟨type⟩ var1, OLD_var1;
  synchronized void A1() {
    if (⟨condition⟩)
    try {
      ⟨body  action⟩
      OLD_var1 = var1; //commit
    } catch(Exception e) {
      var1 = OLD_var1; //rollback
      throw(e);
    }
  }
  ...
}
```

Figura 2.1: Uma ação simples Nautilus traduzida para Java

Desta forma a maneira mais próxima da orientação a objetos dentro de Nautilus para executar entradas/saídas seria ter um objeto System e todo o objeto que precisasse fazer E/S deveria ser agregado com este. Entretanto, esta idéia tem o seguinte problema: um objeto só pode ser agregado apenas uma vez. Esta restrição pode ser atenuada com o uso de classes, mas como o subconjunto de Nautilus escolhido a ser implementado ainda não as implementa, optou-se por adicionar primitivas de E/S do estilo Pascal, write e read.

2.4 Objetos e Ações Simples

Em Nautilus os objetos não são executáveis, são apenas meios de organizar as ações que são autoexecutáveis por *default*. Entretanto as ações que certamente serão executados serão apenas as que não sofrem nenhum tipo de restrição (não são componentes de nenhuma outra ação, a condição de ativação é verdadeira, etc.) detalhados na seção autoexecução.

2.4.1 Ação como transação

As ações em Nautilus são todas transações, sendo atômicas (indivisíveis) e em caso de falha para todos os efeitos deve reverter ao estado anterior. Um esquema simplificado para traduzir ações para métodos é mostrado na fig. 2.1.

Neste mapeamento todo slot de Nautilus gera dois atributos dentro da classe Java correspondente. Slots em Nautilus são como atributos de objetos em termos de orientação a objetos. Um deles mantém o estado antigo do slot para o caso de falha. A condição em `enb` é traduzida como um simples `if`. A condição é uma expressão booleana simples.

O esquema na fig. 2.1 é uma simplificação que funciona no caso em que a ação não é usada por nenhuma ação composta. Quando a ação é composta por agregação ou reificação, sua implementação necessitará referenciar as divisões internas das ações componentes, portanto a ação será traduzida em 4 métodos (a ação e suas divisões internas: body, commit e rollback). Este caso é mostrado em fig. 2.2 (onde o sufixo "_OLD" foi mudado para uma plica).

Nas próximas construções parte do código necessário para lidar com exceções e

```
object Obj2
...
body
  slot S1,S2:⟨type⟩
  act A1
    ⟨body action⟩
    // alters slots S1 and S2
  ...
end Obj2
```

```
class TObj2 extends Thread {
  ⟨type⟩ S1, S1', S2, S2';
  synchronized void A1() {
    try {
      A1_body();
      A1_commit();
    }
  catch(Exception e) {
      A1_rollback();
    }
  }
  void A1_body() {
    ⟨body action⟩
  }
  void A1_commit() {
      S1' = S1;
      S2' = S2;
  }
  void A1_rollback() {
    S1 = S1';
    S2 = S2';
  }
  ...
}
```

Figura 2.2: Ação Nautilus traduzida para Java

```
slot
  S1, S2:⟨type⟩
act A1
  cps
    val S1 << ⟨expression1⟩
    val S2 << ⟨expression2⟩
  end cps
```

```
⟨type⟩ S1, S1', S2, S2', Temp;

void A1_body() {
    Temp = ⟨expression1⟩;
    S2 = ⟨expression2⟩;
    S1 = Temp;
}
```

Figura 2.3: Tradução de concorrência intra-ação

funções auxiliares (commit, rollback) podem ser omitidas por simplicidade.

2.4.2 Concorrência intra-ação

Concorrência dentro de uma ação é expressa pela palavra chave `cps` usada para fazer atribuição concorrente, como na fig. 2.3. Pode ser traduzida de forma simples usando variáveis temporárias e execução seqüencial

Para sistemas com múltiplas CPUs é possível criar uma thread para avaliar cada expressão a direita, mas para otimizar esse comportamento seria necessário verificar o quão complexo a expressão precisa ser para compensar o processamento extra de criar uma thread.

2.4.3 Não determinismo

Em Nautilus, uma ação pode ter diversos corpos de execução alternativos (introduzidos pela palavra chave `alt`). Para traduzir tal não determinismo, inicialmente pensou-se em usar um operador de escolha que poderia ser a função `random()`

```
act A2
  alt A21
    ⟨body action A21⟩
  alt A22
    ⟨body action A22⟩
```

```
void A2_body() {
  Random rand = new Random();
  switch(rand.nextInt(2)) {
      case 0:
        ⟨body action A21⟩
        break;
      case 1:
        ⟨body action A22⟩
        break;
  }
}
```

Figura 2.4: Tradução Inicial de Não determinismo

como na fig. 2.4.

Mas se percebeu que esta não funcionaria em casos de ações que são restritas por `request` ou são usadas como um componente de uma reificação (nesses casos a semântica formal exige que se há alternativas que falham e outras que são bem sucedidas, a escolhida deve ser uma das bem sucedidas), portanto foi necessário usar um padrão onde as alternativas são testadas até que uma delas seja bem sucedida. Por exemplo, uma ação `A` com alternativas `A1`,...,`An` pode ser traduzida como mostrada em fig.2.5.

2.4.4 Ação com Parâmetros

No caso de ações com parâmetros (como em uma agregação) há mais complicações para considerar. Parâmetros impõem restrições na ordem de execução das ações. A parte inicial e final das ações se tornam pontos de sincronização onde recebem e enviam os valores dos parâmetros.

Cada ação que recebe ou retorna parâmetros é implicitamente do tipo `request`, uma vez que não pode ser executada sozinha. Uma referência a um parâmetro é feita sempre por nome, não por posição e são possíveis apenas para ações exportadas (ações que são públicas para outros objetos).

Um exemplo ilustrativo é o programa da fig.2.6. No exemplo, os objetos `Obj1` e `Obj2` têm ações com parâmetros, e um terceiro objeto `ObjAg` é uma agregação agindo como uma sincronização entre a entrada e saídas dos primeiros.

Dependendo do corpo de `A1` e `A2`, existem as seguintes possibilidades:

- Execução Seqüencial

  Como mostrada na fig. 2.7, uma das ações fornece um parâmetro de saída na primeira instrução através da palavra chave `agg`, que é a entrada da segunda ação, que deste modo pode executar e retorna um parâmetro de saída que é usado pela primeira ação.

- Execução Paralela

  Quando ambas as ações fornecem parâmetros no começo, ambos podem ser executados. Veja fig. 2.8.

- Não execução

```
act A
  alt A1
    ⟨body action A1⟩
  alt A2
    ⟨body action A2⟩
    .
    .
  alt AN
    ⟨body action AN⟩
```

```
void A_body(){
  Integer[] array = new Integer[N];
  int i = 0;
  for(i = 0; i < N; i++)
    array[i] = new Integer(i);
  //Shuffle the elements in the array
  Collections.shuffle(
    Arrays.asList(array));
  boolean success = false;
  i = 0;
  while(success == false && i < N){
    try{
      switch(array[i].intValue()){
      case 0:
        ⟨body action A1⟩
          break;
        ..
      case N−1 :
        ⟨body action AN⟩
          break;
      }
      success = true;
    }catch(Exception e){
      A_Rollback();
      i++;
      success = false;
    }
  }
}
```

Figura 2.5: Tradução de Não determinismo

```
object Obj1
export
  A1
    in i:⟨type⟩
    out o:⟨type⟩
body
  ...
  act A1
    ⟨body action A1⟩
end Obj1

object Obj2
export
  A2
    in i:⟨type⟩
    out o:⟨type⟩
body
  ...
  act A2
    ⟨body action A2⟩
end Obj2

object ObjAg
aggregation of
  Obj1
  Obj2
...
body
  act AR composed by
    A1 of Obj1
    A2 of Obj2
   match
     A1.i of Obj1
     A2.o of Obj2
   match
     A2.i of Obj2
     A1.o of Obj1
end ObjAg
```

Figura 2.6: Exemplo de Parâmetros

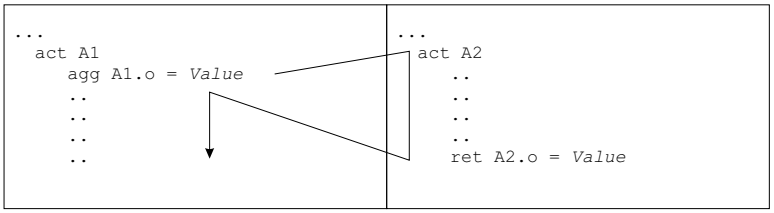

Figura 2.7: Execução Seqüencial

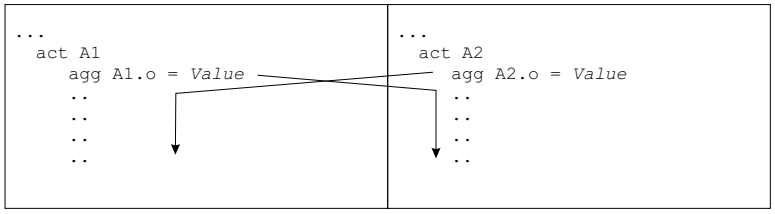

Figura 2.8: Execução Paralela

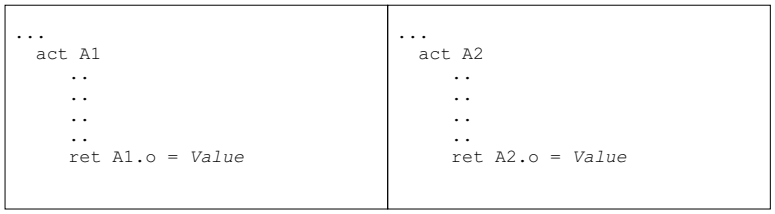

Figura 2.9: Não execução

O caso de não execução significa que as ações são mutuamente dependentes e não ocorrem (fig.2.9). A semântica é vazia, para todos os efeitos é como se fossem um `enb false`.

Com base nesses casos deverá ser feita a seguinte análise:

1. quais os parâmetros que as seções `agg` fornecem?
2. se nenhuma ação pode começar apenas com estes parâmetros então é o caso de não execução
3. senão escolhe-se qualquer ação que pode ser executada
4. se existe alguma ação componente não executada volta para 1.

A tradução de parâmetros de Nautilus pode ser feita usando "pseudo-slots". Desta forma cada parâmetro poderia ser um slot como "ActionId_Param_Id:TypeId", com a restrição que não pode ser usado fora da ação correspondente "ActionId" e que a atribuição é feita apenas nas seções `agg` e `ret`.

2.5 Auto-execução das ações

A tradução do aspecto de auto-execução das ações é particularmente complicado, ela precisa levar em conta alguns aspectos que são dependentes do código fonte, pois embora uma ação seja auto-executável por default existem diversas restrições

- a ação não pode ser do tipo `request`;
- a ação não pode ter parâmetros;
- a ação não deve ser usada por outros objetos. Este é o aspecto mais trabalhoso. É necessário fazer uma análise do código fonte para verificar se a ação é utilizada por outros objetos em operações como reificação (o que faria que a execução passasse a ser responsabilidade do objeto reificador) ou visão (a execução passa a ser responsabilidade do objeto visão).

Se a ação satisfaz estas condições, além da tradução do seu código do seu corpo, é feita uma simulação do aspecto da auto-execução da ação inserindo-se uma chamada (randômica) ao procedimento no método `run()` do objeto Java como mostrado no exemplo de não determinismo.

2.6 Reificação

A reificação é a implementação de um objeto sobre outro. Uma ação de um objeto de reificação é construído sobre as ações de um objeto base. No caso de Nautilus é uma construção que impõe uma ordem, respeitando sequencialidade, concorrência e não determinismo. Pode ser traduzido como chamadas a métodos com a restrição transacional (fig. 2.10).

2.7 Agregação

É um dos principais mecanismos de composição de Nautilus. Cada objeto pode ser visto como um sistema e a agregação estabelece alguns pontos de sincronização entre os objetos agregados. A construção também respeita o aspecto transacional (o código de tratamento de exceções é omitido pois é similar ao caso da reificação).

Um exemplo simples é mostrado na fig. 2.11.

Neste caso a agregação tira a possibilidade de uma ação de auto-executar, mas se, por exemplo `Tac` têm uma ação chamada `WriteTac2` e nenhuma referência a esta ação no objeto `TicTac`, esta ação poderia auto executar, mas seria inacessível (a agregação iria esconde-la). E o `switch` do método `run()` teria `case 1: Tac.WriteTac2()`.

Neste exemplo não há nenhuma sequencialidade imposta a como ações componentes deveriam executar, mas no caso de uma agregação usando ações com parâmetros existem as complicações mostradas em 2.4.4.

2.8 Visão

É a idéia de encapsular um objeto preexistente escondendo ações que não sejam necessários no contexto. Além disso pode-se adicionar restrições. Pode ser simulada em Java utilizando-se mecanismo similar ao usado com a reificação como mostrado na fig. 2.12.

No caso acima `ObjV` adicionou a restrição `request` para as ações `Start`, `Action1`, `Action2`. E escondeu a `Action3` mas esta não foi restrita então é a única que pode autoexecutar.

```
object ObjBase
category
  birth Start
body
  ...
  act Start
    ⟨body action Start⟩
  act Action1
    ⟨body action Action1⟩
  act Action2
    ⟨body action Action2⟩
  act Action3
    ⟨body action Action3⟩
  ...
end ObjBase

object ObjRef over ObjBase
category
  birth Start
body
  act Start
     Start
  act ActionComp1
     seq
       Action1
       Action2
     end seq
end ObjRef
```

```
class TObjBase extends Thread{
    ...
        TObjBase(){
            super();
            Start();
        }
    void Start(){...}
    void Action1(){...}
    void Action2(){...}
    void Action3(){...}
    ...
}

class TObjRef extends Thread {
    TObjBase ObjBase;
    TObjRef(){
        super();
        Start();
    }
    void Start(){
        ObjBase = new TObjBase();
    }
    void ActionComp1() {
        try {
            ActionComp1_body();
            ActionComp1_commit();
        } catch(Exception e) {
            ActionComp1_rollback();
        }
    }
    void ActionComp1_body(){
        ObjBase.Action1_body();
        ObjBase.Action2_body();
    }
    void ActionComp1_commit(){
        ObjBase.Action1_commit();
        ObjBase.Action2_commit();
    }
    void ActionComp1_rollback(){
        ObjBase.Action1_rollback();
        ObjBase.Action2_rollback();
    }
    public void run(){
        boolean the_end = false;
        Random rand = new Random();
        while (!the_end)
            switch(rand.nextInt(1)) {
            case 0:
                ActionComp1();
                break;
            }
    }
}
```

Figura 2.10: Reificação

```
object Tic
category
  birth Start
  export WriteTic
body
  ...
  act Start
    ⟨body action Start⟩
  act WriteTic
    ⟨body action WriteTic⟩
end Tic

object Tac
category
  birth Start
  export WriteTac
body
  ...
  act Start
    ⟨body action Start⟩
  act WriteTac
    ⟨body action WriteTac⟩
end Tac

object TicTac
aggregation of
  Tic
  Tac
category
  birth Start
body
  act Start composed by
     Start of Tic
     Start of Tac
  act WriteTicTac composed by
     WriteTic of Tic
     WriteTac of Tac
end TicTac
```

```
  class TTic{
    ...
    TTic(){
       Start();
    }
    void Start(){...}
    void WriteTic(){...}
     ...
  }

  class TTac{
    ...
    TTac(){
       Start();
    }
    void Start(){...}
    void WriteTac(){...}
    ...
  }

class TTicTac extends Thread {
  TTic tic;
  TTac tac;
  TTicTac() {
    super();
    Start();
  }
  void Start() {
     tic = new TTic();
     tac = new TTac();
  }
  void WriteTicTac() {
     Tic.WriteTic();
     Tac.WriteTac();
  }

  public void run() {
    boolean the_end = false;
    while (!the_end)
    switch( random(1)) {
      case 0:
    WriteTicTac();
        break;
    }
  }
}
```

Figura 2.11: Agregação

```
object ObjV view of Obj1
export
  Start Action1 Action2
category
  birth request Start
  request Action1 Action2
end ObjV

object Obj1
export
  Start Action1
  Action2 Action3
category
  birth Start
body
  act Start
  act Action1
  act Action2
  act Action3
end Obj1
```

```
class TObjV extends Thread {

  class TObj1 {
    Tobj1() {
       Start();
    }
   void Start() { }
   void Action1() { }
   void Action2() { }
   void Action3() { }
  }

  Tobj1 Obj1;

  TTObjV() {
    Super();
    Start();
  }

  void Start()
     Obj1 = new TObj1();
  }

  void Action1() {
     Obj1.Action1();
  }

  void Action2() {
     Obj1.Action2();
  }

  public void run() {
    boolean fim = false;
    while (fim)
    switch( escolhe(1)) {
      case 0:
        Obj1.Action3();
        break;
    }
  }

}
```

Figura 2.12: Visão

3 MAPEAMENTO DE EXTENSÕES

Este capítulo fala de possíveis formas de implementar algumas extensões sobre a linguagem núcleo. Interação é uma construção que estava presente na primeira versão da linguagem Nautilus (MENEZES; SERNADAS; COSTA, 2000), e Classes foram propostas por (CARNEIRO, 1999) e mais detalhado por (D'ANDREA, 2002).

3.1 Interação

Uma construção que estava presente na primeira versão da linguagem Nautilus (MENEZES; SERNADAS; COSTA, 2000) e foi omitida no mapeamento núcleo era interação.

As principais razões para essa ausência são: primeiro, a construção pode ser simulada pelos outras construções e é, de certa forma, contra o espírito da linguagem (como `goto` em programação estruturada); segundo, parecia ser a construção que mais poderia levar a complicações de implementação.

Essas críticas a interação ainda são válidas mas como interações são a construção mais encontrada em linguagens tradicionais, nesta seção veremos como seria o mapeamento de interações de Nautilus em Java.

A estrutura de composição mais próxima de interação no mapeamento original de Nautilus para Java é a agregação (ambos são baseados na sincronização dos objetos componentes na semântica formal).

O resultado de uma interação de objetos é um objeto sincronizado. Quando este objeto sincronizado é usado como componente de uma agregação ou qualquer outra operação é necessário fazer referências a todos os objetos da interação (portanto normalmente por uma questão de organização, na maioria dos exemplos a interação acaba encapsulada por uma visão).

A maior diferença entre interação e agregação é que, no primeiro, o relacionamento entre objetos é definido dentro de cada objeto enquanto no último, o relacionamento é definido de fora dos objetos componentes. Isto é, interação usa uma abordagem mais *hard-wired* do que agregação, já que faz importação explícita nos objetos componentes, e além disso como os objetos das interações podem fazer importações cruzadas é mais complicado verificar que não há recursão indireta (recursão não é permitida em Nautilus para facilitar a manutenção da atomicidade das ações).

Objetos de uma interação podem ser vistos como um único objeto com especificação distribuída. Em Nautilus uma referência ao objeto resultante de uma interação é feito através dos objetos componentes (`interaction/ end interaction`), e seguindo a mesma idéia, uma referência a uma ação composta é através de suas

ações componentes (`int/ end int`).

O esquema de tradução para ações agregadas pode ser adaptado para o caso de interação. Veja a fig. 3.1.

Se $\langle bodyO1 \rangle$ têm um `import O2`, a tradução poderia ter `TO2 O2;` para fazer possíveis referências de `O2` em `O1`.

A maior diferença da tradução de uma agregação é que a classe correspondente a uma agregação têm um nome atribuído pela construção, enquanto uma classe de uma interação tem nomes fornecidos por alguma regra do tradutor (no exemplo `Interaction_` + nome dos componentes).

A análise estática necessária se torna mais complicada quando a linguagem é estendida com interação. Por exemplo, a semântica não permite atribuições concorrentes ao mesmo slot como mostrado abaixo:

```
object O1
slot c:integer
act A
cps
  val c<<1
  val c<<2
end cps
end O1
```

este erro é obvio, mas com interação os erros se tornam mais difíceis de detectar:

```
object O1
 import O2A1 of O2
 slot c:integer
 act O1A1
   val c<< 1
 act O1A2
  cps
    val c<<2
    call O2A1 of O2
  end cps
end O1

object O2
 import O1A1 of O1
 act O2A1
   call O1A1 of O1
end O2
```

No código acima é necessário analisar todos os componentes da interação para encontrar o erro. A mesma espécie de análise é necessária para garantir que não há recursão de forma indireta.

3.2 Classes

Classes em Nautilus foram definidas em (CARNEIRO, 1999) (semântica formal) e (D'ANDREA, 2002) (onde foram esboçadas uma sintaxe textual e diagramática).

Classes são importantes para reuso de código. O melhor exemplo é o caso dos filósofos onde era necessário repetir o mesmo corpo para os diversos filósofos. Mecanismos de herança não foram definidos mas (D'ANDREA, 2002) mostra como simular herança.

```
object O1
  ⟨O1 body⟩
end O1

object O2
import A3 of O1

  slot c:integer
  act A1
  seq
    call A3 of O1
    val c << c + 1
  end seq

end O2
```

```
class Interaction_O1O2 {

class TO1 {
  ⟨O1 body⟩
}

TO1 O1;

class TO2 {

   int c,c′;

   void A1() {
     try {
       A1_body();
       A1_commit();
     } catch(E) {
       A1_rollback();
     }
   }

   void A1_body() {
      O1.A3_body();
      c = c + 1;
   }

   void A1_commit() {
      O1.A3_commit();
      c′ = c;
   }

   void A1_rollback() {
     O1.A3_rollback();
     c = c′;
  }
}

}
```

Figura 3.1: Uma interação Nautilus traduzida para Java

```
class A
category
  birth iniA
body
  act iniA
    ⟨body iniA⟩
  act metA1
    ⟨body metA1⟩
  act metA2
    ⟨body metA2⟩
end A
```

```
class A {

  ⟨actions translation⟩

  void run() {
    iniA();
    while (true)
      case (choice()) {
      0: metA1();
      1: metA2();
      }
  }
}
```

Figura 3.2: Tradução de uma Classe Simples

Nesta extensão de Nautilus, classes são declarações e os objetos são criados como instâncias das classes. Objetos criados a partir da palavra chave `object` podem ser vistos como instâncias de uma classe anônima.

Com a adição de `class`, a antiga declaração `object` se torna redundante e pode ser eliminada. Além da redundância, outro motivo para eliminação de `object` é a semântica destrutiva que, apesar de fazer sentido visualizando o programa como um grafo de um sistema de transições, não é muito intuitiva. Uma alteração que seria necessária sem `object` é definir um mecanismo que inicie a criação de instâncias das classes.

3.2.1 Operações sobre classes

Classes podem ser usadas para fornecer uma semântica não destrutiva das operações de composição. Por exemplo, atuando sobre objetos uma agregação é uma operação que consome objetos que passam a não existir e retorna um novo objeto agregado. A tradução definida para a linguagem baseada em objetos pode ser usada como base, com adaptações principalmente na parte de auto-execução.

3.2.1.1 Classe Simples

A tradução segue o mesmo molde do caso de objeto simples. Por exemplo, veja a fig. 3.2.

3.2.1.2 Reificação

Uma reificação que na versão de objetos de Nautilus consumia um objeto base e retornava um novo objeto reificado, passaria na versão de classes a ser uma função que faz uma referência a uma classe e retorna outra classe.

Seja uma reificação B (fig. 3.3) sobre a classe A (fig. 3.2), o comportamento de auto-execução da classe A não seria o esperado. Uma possível adaptação da tradução seria fazer a classe B executar sobre uma classe A' que herde as características de A exceto o método `run()` que seria sobrecarregado de acordo com as necessidades de B. Neste caso, uma reificação de B sobre A, todas as ações auto-executáveis de A' seriam anuladas.

```
class B over A
birth iniB
body
  act iniB
    iniA
  act metB
    metA1
    metA2
end B
```

```
class B {
  class A′ extends A {
    void run() {
    ⟨autoexecutable actions of A
          restricted by B⟩

    }
  }

  A′ anA′;

  void iniB() {
    anA′.iniA();
  }

  void metB() {
    ⟨translation of metB⟩
  }

  ⟨auxiliary methods : metB_body(), etc.⟩

  void run() {
    iniB();
    while (true)
      case (choice()) {
      0: metB();
      }
  }
}
```

Figura 3.3: Tradução de reificação de classes

```
object A1
  ⟨A1 body⟩
end A1

object A2
  ⟨same body as A1⟩
end A2

object B1
  ⟨B1 body⟩
end B1


object Agg1
aggregation of
  A1 A2 B1
body
  ⟨body Agg1⟩
end Agg
```

```
class A
  ⟨A body⟩
end A

class B
  ⟨B body⟩
end B1


class Agg
aggregation of
  A1:A A2:A B1:B
body
  ⟨body Agg⟩
end Agg
```

Figura 3.4: Agregação com objetos / classes

3.2.1.3 Agregação

Uma extensão natural de agregação usando classes seria receber classes, criar objetos destas, e retornar uma classe que cujas instâncias são agregações (as classes dos objetos componentes permanecem disponíveis para uso), veja fig. 3.4

Uma observação, esta definição de agregação de classes usando instâncias de classes difere da mostrada em (D'ANDREA, 2002), onde a agregação de classes usava uma referência direta a classe em vez de criar instâncias (a razão da mudança é permitir usar diversos objetos de uma mesma classe).

As mudanças que seriam necessárias em relação a agregação baseada em objetos se referem novamente a auto-execução. No caso de uma agregação tal como 3.5, o conjunto de ações auto-executáveis em E1 e E2 em C seria diferente das ações auto-executáveis de uma instância E menos as usadas por ações de B. Na tradução haveria E1 e E2 seriam instâncias de duas classes E' e E" que herdam de E, com o método `run()` sobreescrito, apenas `a2()` seria auto-executável em E' e apenas `a1()` seria auto-executável em E".

3.2.1.4 Visão

Uma visão praticamente não precisa de adaptações em relação a tradução baseada em objetos. O esquema de tradução baseado em classes funciona de forma idêntica a original, apenas a operação recebe uma classe e retorna outra classe, sendo que a primeira permanece disponível.

3.2.1.5 Interação

Uma operação de interação que faz referência a classes e retorna classes como definido pela BNF de (D'ANDREA, 2002) é possível e seria bem similar a versão baseada em objetos mostrada na seção anterior, entretanto a incapacidade de utilizar múltiplas instâncias de classes reduz sua utilidade em relação a agregação.

```
class E
body
  act ini
    ..
  act a1
    ..
  act a2
    ..
end E

class C
aggregation of
  E1:E
  E2:E
body
  act ini
    ini of E1
    ini of E2
  act a
    a1 of E1
    a2 of E2
end C
```

```
class E {
  ⟨translation of E actions⟩

  void run() {
    ini();
    while (true)
      case (choice()) {
      0: a1();
      1: a2();
      }
  }

}

class C {
  class E′ extends E {
    void run() {
    while (true)
      case (choice()) {
      0: a2();
      }
    }
  }
  class E″ extends E {
    void run() {
    while (true)
      case (choice()) {
      0: a1();
      }
    }
  }
  E′ E1;
  E″ E2;

  void ini() {
    ..
  }

  void a {
    ⟨translation of a⟩
  }

  ⟨auxiliary methods : a_body(), etc.⟩

  void run() {
    ini();
    while (true)
      case (choice()) {
      0: a();
      }
  }

}
```

Figura 3.5: Tradução de agregação de classes

Uma tentativa de estender a interação como operação baseada em classes com múltiplas instâncias junto com uma adaptação do esquema de tradução de interação baseado de objetos é mostrada a seguir, embora seu valor seja mais no sentido apresentar seus problemas. No caso de uma interação como:

```
class F
import a1 of G1:G
import a2 of G2:G
..
end F

class G
export a1 a2
..
end G
```

```
class Interact_anFG1G2 {
  G G1;
  G G2;
  class F {
  ..
  }
}

class G {
 ..
}
```

A tradução seria bem similar a original com as adaptações descritas acima para agregação. Mas a semântica da interação se torna estranha. Particularmente como lidar com o objeto resultado de instânciação de interação dessas? Algo como `i: interact G1 G2 F1:F end interact`?

E no caso de importação cruzada, o problema do esquema de tradução proposto se torna obvio:

```
class K
import a1 of L1:L
export a2
..
end K

class L
import a2 of K1:K
import a1
..
end L
```

```
class Interact_aKL1 {
  L L1;
  class K {
  ..
  }
}

class Interact_aLK1 {
  K K1;
  class L {
  ..
  }
}
```

O esquema de tradução evidentemente falha. O problema é que interação só tem sentido como comunicação entre objetos, não entre uma classe e um objeto. Portanto esta extensão de interação Nautilus de classes usando instâncias é inviável, e enquanto a versão de interação de classes usando classes não apresenta tais problemas, a perda de capacidade de manipular múltiplas objetos de uma mesma classe a tornam bem mais restrita em relação a agregação.

4 REFINANDO O MAPEAMENTO

Após feito o mapeamento Java do subconjunto de Nautilus e propor algumas extensões, pensou-se em propor algumas mudanças possíveis visando-se um nível maior de granularidade de concorrência. O mapeamento original foi conservador explorando concorrência apenas no nível de objetos e não de ações. As razões dessa escolha foram:

- Java usa o sistema de threads do sistema operacional alvo e como o tradutor a ser construído visa ensino, os sistemas operacionais alvo típicos (Linux, Windows) usam sistemas de threads que são consideravelmente pesados.
- Reduzindo o nível de concorrência, reduz-se o número de interações e é mais fácil garantir propriedades semânticas como a atomicidade das ações.

Apesar deste foco visando ensino, é sempre interessante também apontar possíveis formas de explorar mais concorrência. Entretanto percebeu-se que modificar programas concorrentes de forma razoavelmente confiável é uma tarefa complexa. Então utilizou-se CSP (HOARE, 1985) como uma ferramenta auxiliar visando otimizar o código Java do mapeamento original. Basicamente CSP foi usado como uma notação mais abstrata onde o raciocínio sobre os programas é facilitado.

Depois de ganhar alguma experiência usando CSP, o autor fez um pequeno esboço de um mapeamento de Nautilus para CSP no final do capítulo.

A notação usada para CSP é próxima do padrão de (HOARE, 1985), átomos com letras minúsculas, $\sqcap$ é escolha não determinística, $\|$ denota paralelismo, $|||$ interleaving e $\triangle$ interrupção. Os processos começam com letra maiúscula.

4.1 Refazendo o mapeamento

Considerando um objeto Nautilus simples como:

```
object o1

category
  birth a1
  death a4, a6
  request a3

body
  slot s1, s2, s3: ⟨type⟩
  act a1
    ⟨body a1⟩
```

```
  act a2
    ⟨body a2⟩
  act a3
    ⟨body a3⟩
  act a4
    ⟨body a4⟩
  act a5
    ⟨body a5⟩
  act a6
    ⟨body a6⟩
end o1
```

Um modelo CSP bem simplificado do programa Java resultante do primeiro mapeamento é
$O1 = CriaSlots;\ Construtor;\ Acoes$
$Construtor = A1$
$Acoes = \mu\, X \bullet (A2;\ X) \sqcap (A3;\ X) \sqcap A4 \sqcap (A5;\ X) \sqcap A6$
$Execucao = O1 \setminus A3$

Observando tal expressão se torna evidente a opção de implementar concorrência de ações intra-objeto como não determinismo

Uma opção obvia é substituir não determinismo por concorrência:

$$AcoesAutoexec = o1.Monitor//(\mu\, X \bullet ((A2;\ X) \;|||\; (A5;\ X)) \;|||\; A4 \;|||\; A6)$$

Neste caso como aN são métodos síncronos na semântica de monitores de Java, existe um lock implícito em qualquer aN,

$$AN = o1.lock \rightarrow AN.Body;\ o1.release$$

que deve ser sincronizado com

$$o1.Monitor = o1.lock \rightarrow o1.release \rightarrow o1.Monitor$$

Neste caso não há ganho real, as ações continuam a executar de forma alternada. Para que haja uma maior concorrência, o lock precisa ser mais granular. Por exemplo, transformar

$$A2 = o1.lock \rightarrow A2.Body;\ o1.release$$

em

$$A2 = o1.a2.Lock;\ a2.Body;\ o1.a2.Release$$

onde

$$o1.a2.Lock = o1.s1.lock \rightarrow o1.s2.lock$$

$$o1.a2.Release = o1.s1.release \rightarrow o1.s2.release$$

assumindo que a2 do objeto o1 utiliza os slots s1 e s2. E cada slot dispõe de um monitor

$$o1.s1 : Monitor$$

definido como

$$Monitor = lock \rightarrow Usa;\ release \rightarrow Monitor$$

$$Usa = (set \rightarrow Usa \sqcap get \rightarrow Usa \sqcap Skip)$$

então a definição das ações autoexecutaveis torna-se

$$AcoesAutoexec = o1.monitores // \mu X \bullet ((A2;\ X) \ |||\ (A5;\ X) \ |||\ A4 \ |||\ A6)$$

$$o1.monitores = o1.s1 \ |||\ o1.s2 \ |||\ o1.s3$$

Assim se a5 não usa nem s1 ou s2, ele pode executar de forma concorrente a a2.

É importante lembrar de ordenar o lock dos slots sempre da mesma maneira, ou uma situação de deadlock pode ocorrer, tal como:

$$o1.a2.Lock = o1.s1.lock \rightarrow o1.s2.lock$$

$$o1.a4.Lock = o1.s2.lock \rightarrow o1.s2.lock$$

Considerando agora a atomicidade das ações, uma opção é usar commit ou rollback no lugar de release. Assim

$$Monitor = lock \rightarrow Usa;\\ (commit \sqcap rollback) \rightarrow Monitor$$

e

$$A2 = o1.a2.Lock;\\ ((a2.Body;\ o1.a2.Commit) \triangle fail \rightarrow o1.a2.Rollback)$$

onde $a \triangle e \rightarrow b$ simboliza um processo que executa como a a não ser que haja um evento e (neste caso uma exceção simbolizada por *fail*), neste caso se comporta como b.

Uma forma de implementar os monitores para cada slot em Java é

```
class VarInt {
   int old_value = 0, value = 0;
   boolean free = true;
   boolean old_isNull = true, isNull  = true;

   synchronized void lock()
        throws InterruptedException
   {
     while (!free) { wait();}
     free = false;
   }

   void set(int value) {
     if (free) throw
        new Error("Trying to set without lock");
     this.value = value;
     this.isNull = false;
   }

   int get() {
     if (free) throw
        new Error("Trying to get without lock");
     if (isNull) throw
        new Error("Null value");
     return (value);
   }
```

```
   synchronized void commit() {
     old_value = value;
     old_isNull = isNull;
     free = true;
     notifyAll();
   }

   synchronized void rollback() {
     value = old_value;
     isNull = old_isNull;
     free = true;
     notifyAll();
   }
}
```

Conforme a especificação em CSP, o código cria uma região crítica onde os valores internos podem ser manipulados e as mudanças podem ser confirmadas ou canceladas (também lida com valores `Null`). Cada tipo (string, array, etc.) seria transformado em uma classe similar (VarString, VarArray, etc.).

O código de um objeto

```
object o1
 ..
body
  slot s1, s2, .., s5:integer
  act a1
    enb ⟨cond⟩
    seq
      s1 << 1
      s2 << s2+1
    end seq
  ..
end o1
```

é transformado da seguinte forma

```
public class TO1 extends Thread {
  VarInt s1 = new VarInt();
  VarInt s2 = new VarInt();
  ..
  VarInt s5 = new VarInt();
  boolean end = true;
  void a1Body(VarInt _s2) {
    s1.set(1);
    s2.set(_s2.get() + 1);
  }
  void a1() {
    try {
      // a1 Lock()
      s1.lock();
      s2.lock();
      if (!⟨cond⟩) throw
         new Exception("a1 enb false");
      a1Body(s1);
      // a1 Commit();
      s1.commit();
      s2.commit();
    } catch (Exception e) {
```

```
      // a1 Rollback()
      s1.rollback();
      s2.rollback();
    }
  }

  public class pA1 extends Thread {
    boolean autoexec;
    public pA1( boolean b,
                ThreadGroup tg,
                String s) {
      super(tg, s);
      autoexec = b;
    }
    public void run() {
      while (true) {
        if (end) break;
        a1();
        if (!autoexec) break;
      }
   }
  }
  ..

  ⟨object autoexec part⟩
}
```

Quando o contexto faz a ação não auto-executável, `pA1` não seria usado e ações compostas usariam `a1Body()` ou `pA1Body` (um `a1body()` com um thread associado, veja abaixo).

```
  public class pA1Body extends Thread {
    public pA1(ThreadGroup tg, String s) {
      super(tg, s);
    }

    private VarInt s2;

    public Exception ep = null;

    public RightVars(VarInt _s2) {
      s2 = _s2;
    }

    public void run() {
      try {
        a1Body(s2);
      } catch (Exception e) {
        ep = e;
      }
    }
  }
```

O parâmetro em `a1Body()` representa uma variável do lado direito de uma atribuição, sendo útil no caso de `cps` como explicado na seção de reificação.

Uma ação `birth` seria chamado no início do método `run()` do objeto. Esta teria um atribuição `started = true` na seção de commit. Enquanto uma ação `death` seria similar a uma ações normal com `end = true` na seção de commit.

$\langle object\ \ autoexec\ \ part \rangle\ =$

```
public void run() {
  boolean started = false;
  while (!started)
      birth();

  ThreadGroup myThreadGroup =
     new ThreadGroup("My Group of Threads");

  Thread a1 = new pA1(true,myThreadGroup,"a1");
  a1.start();
  ..
  Thread an = new pAn(true,myThreadGroup,"an");
  an.start();
}
```

4.2 Agregação

Seja por exemplo uma agregação oc de dois objetos o1 e o2, tal como

```
object o1
category
  birth o1b
body
  slot o1s1:<type>
  act o1a1
  act o1a2
  act o1a3
end o1

object o2
category
  birth o2b
body
  slot o2s1:<type>
  act o2a1
  act o2a2
  act o2a3
end o1

object oc
  aggregation o1 o2
body
  act ocac composed by
    o1a2 of o1
    o2a2 of o2
  act ocb composed by
    o1b of o1
    o2b of o2
end oc
```

A tradução em CSP é

$$\begin{aligned} O1 &= Slots//O1b;\ (O1a1 \ |||\ O1a3 \ |||\ O1a2) \\ O2 &= \text{<análogo a o1>} \end{aligned}$$

$$
\begin{aligned}
Oc &= Ocb;\ \mu X \bullet Ocac1;\ X \\
Ocac1 &= (ocac1.Lock;\ ocac1.Corpo;\ ocac1.Commit) \\
&\quad \triangle\ fail \rightarrow ocac1.Rollback \\
ocac1.Corpo &= (o1a2.Corpo \parallel o2a2.Corpo) \\
ocac1.Lock &= o1a2.Lock;\ o2a2.Lock
\end{aligned}
$$

Commit e Rollback seguem o mesmo modelo de Lock.

Se houvessem várias ações agregadas, de forma mais geral, o objeto agregado seria:

$$
\begin{aligned}
Oc &= (Ocb1 \sqcap .. \sqcap Ocbn); \\
&\quad \mu X \bullet ((Ocac1;\ X) \ |||\ ..\ |||\ (Ocacn;\ X) \ |||\ Ocd1 \ |||\ ..\ |||\ Ocdn)
\end{aligned}
$$

Em termos de código Java isto significa que ocac pode ser traduzido em

```
public void ocac_body() {
  ThreadGroup tg = new ThreadGroup("ocac");

  o1.po1a2 p1 = new o1.o1a2Body(tg,"o1a2");
  p1.start();

  o1.po2a2 p2 = new o1.o2a2Body(tg,"o1a2");
  p2.start();

  p1.join();
  p2.join();

  if ((p1.ep != null)||(p2.ep != null)) {
    if (t1.ep == null) {
      throw t2.ep;
    } else {
      throw t1.ep;
    }
}

public void ocac()  {
  try {
    // sequence of o1a2 locks and o2a2 locks
    o1.o1a2Lock();
    o2.o2a2Lock();

    ocac_body();

    // sequence of o1a2 commits and o2a2 commits
    o1.o1a2Commit();
    o2.o2a2Commit();
  } catch (Exception e) {
    // sequence of o1a2 rollbacks and o2a2 commits
    o1.o1a2Rollback();
    o2.o2a2Rollback();
  }
}
```

4.3 Reificação

Dado uma reificação como

```
object o1
category
  birth o1b
body
  act o1b
  act o1a1
  act o1a2
end o1

object oreif
category
  birth orb
body
  act orb
    o1b
  act ora
    seq
      o1a1
      o1a2
    end seq
end oreif
```

o equivalente em CSP é

$$Oreif = Orb;\ \mu X \bullet Ora;\ X$$

$$Ora = (ora.Lock;\ ora.Body;\ ora.Commit) \triangle fail \rightarrow ora.Rollback$$

$$ora.Body = o1a1.Body;\ o1a2.Body$$

Diferente da agregação onde os lock das ações componentes é disjunto, uma vez que são de objetos diferentes, na reificação pode-se ter casos como

$$o1a1.Lock = o1s1.lock \rightarrow o1s2.lock.o1s2$$

$$o1a2.Lock = o1s1.lock$$

o que geraria $ora.Lock = o1s1.lock \rightarrow o1s2.lock \rightarrow o1s1.lock$ usando o padrão de concatenação seqüencial da agregação. Portanto para evitar deadlocks, $Lock.ora$ deve ser composto pela união ordenada dos alfabetos de $Lock.o1a1$ e $Lock.o1a2$, ou $Lock.ora = ordena(\alpha(Lock.o1a1) \cup \alpha(Lock.o1a2))$ onde ordena é uma função que transforma o conjunto em uma seqüência ordenada.

O mapeamento Java se torna:

```
class To1 extends Thread{
  ...
  void o1b(){...}
  void o1a1(⟨RightSideSlots⟩){...}
  void o1a2(⟨RightSideSlots⟩){...}
  ...
}
class Toreif extends Thread {
  To1 o1;
  ...
```

```
  void ora() {
    try {
      ⟨sequence of
          Action1 locks ∪ Action2 locks⟩
      ora_body();
      ⟨sequence of
          Action1 commits ∪ Action2 commits⟩
    } catch(Exception e) {
      ⟨sequence of
          Action1 rollbacks ∪ Action2 rollbacks⟩
    }
  }
  void ora_body(){
    o1.o1a1_body(⟨RightSideSlots⟩);
    o1.o1a2_body(⟨RightSideSlots⟩);
  }
}
```

Agora explicamos o uso de $\langle RightSideSlots \rangle$.
Suponha

```
act Action1
  s1 << s2

act Action2
  s2 << s1
```

com a tradução

```
void Action1_body(VarInt _s2){
  s1.set(_s2.get());
}

void Action2_body(VarInt _s1){
  s2.set(_s1.get());
}
```

então a tradução de

```
  act ActionComp1
     seq
       Action1
       Action2
     end seq
```

seria

```
void ActionComp_Body(){
  Action1_body(ObjBase.s2);
  Action2_body(ObjBase.s1);
}
```

Mas se ActionComp1 fosse uma cláusula composta (cps)

```
  act ActionComp1
     cps
       Action1
       Action2
     end cps
```

então a tradução seria

```
void ActionComp_Body(){
  VarInt s1 = ObjBase.s1.clone();
  VarInt s2 = ObjBase.s2.clone();
  Action1_body(s2);
  Action2_body(s1);
}
```

E Action1_body, Action2_body poderiam também ser executadas em paralelo.

```
 void ActionComp_Body() {
    ThreadGroup tgAc =
       new ThreadGroup("Ac");
    VarInt s1 = ObjBase.s1.clone();
    VarInt s2 = ObjBase.s2.clone();
    ObjBase.pActionComp_Body a1 =
      new ObjBase.pActionComp_Body(tgAc,"WriteTic");
    a1.RightVars(s2);
    a1.start();
    ObjBase.pActionComp_Body a2 =
      new ObjBase.pActionComp_Body(tgAc,"WriteTac");
    a2.RightVars(s1);
    a2.start();
    a1.join();
    a2.join();

    if ((a1.ep != null) || (a2.ep != null))  {
      if ((a1.ep == null) { throw a2.ep; }
         else { throw a1.ep; }
    }
  }
```

A notação $\langle sequence\ of\ Action1\ rollbacks \cup Action2\ rollbacks \rangle$ significa que se $\langle Action1\ rollbacks \rangle$ é {s1.rollback(), s2.rollback()} e $\langle Action2\ rollbacks \rangle$ é {s2.rollback(), s3.rollback()}, então $\langle Action1\ rollbacks \cup Action2\ rollbacks \rangle$ é {s1.rollback(), s2.rollback(), s3.rollback()}. E *sequence of* ordena os elements do conjunto em uma seqüência canonica.

4.4 Parâmetros

No mapeamento apresentado em (FUZITAKI et al., 2003) sugeriu-se que a tradução utilizando-se "pseudo-slots". A sugestão era adequada para o mapeamento onde havia apenas no máximo um thread por objeto. Para este mapeamento, mudanças são necessárias para respeitar o a semântica. É necessário bloquear o consumidor até que que produtor mande o parâmetro. Em outras palavras se impõe uma seqüência:

$$Param = set?x \rightarrow get!x \rightarrow Param$$

que pode ser obtida pelo seguinte código:

```
class ParamInt {
  private boolean isNull = true;
  private int value = 0;

  synchronized int getInt() throws InterruptedException {
    while (isNull) {
      wait();
```

```
    }
    int r = Value;
    isNull = true;
    notifyAll();
    return r;
  }

  synchronized  void setInt(int i) throws InterruptedException {
    while (!isNull) {
      wait();
    }

    isNull = false;
    value = i;
    notifyAll();
  }

}
```

A idéia é que o consumidor espera até o produtor fornecer o valor. Mas há mais detalhes para ser considerados. No caso de não execução, deadlock não é uma interpretação válida da semântica pois poderia também bloquear variáveis de slots tornando as indisponíveis. Uma forma de implementar a não execução é introduzindo um thread de controle que verificasse periodicamente que as ações na agregação não estão todas bloqueadas, se estiverem o thread de controle deveria restaurar os valores e reiniciar os threads. Mas como as ações envolvidas no caso de não execução de agregação iriam entrar sempre em deadlock, no próximo capítulo foi proposto um simples verificador de protocolos para os parâmetros de Nautilus que elimina este caso patológico na análise estática.

4.5 Um mapeamento CSP simplificado

Esta seção foi uma tentativa de simplificar as construções em CSP apresentadas anteriormente. Uma vez que foram criadas baseadas no mapeamento para Java, alguns aspectos poderiam ficar mais complexos do que seriam em uma tradução Nautilus direta para CSP (basicamente devido a obrigação de lidar com variáveis compartilhadas da versão Java onde em CSP teórico normalmente se usaria um estilo mais próximo da programação funcional, isto é, as variáveis seriam passadas como parâmetros de chamadas de cauda).

4.5.1 Objeto simples

Assuma um objeto o, com construtores b_i, destrutores d_i, slots s_i, e acões normais a_i, que pode ser visualizado como:

```
object o
category
  birth b1
  death d1
  ..
  death dn
body
  slot s1:<type_s1> .. sn:<type_sn>

  act b
```

```
  act d1
  ..
  act dn

  act a1
  ..
  act an
end o
```

Para representar o estado do objeto (os slots) utiliza-se um registro:

$$OData = \{s1 :: \langle type_{s1} \rangle, .., sn :: \langle type_{sn} \rangle\}$$

Também existem dois estados especiais

$$OStart = \{start\}$$

$$OFinal = \{final\}$$

usados para marcar respectivamente inicio e final do ciclo de vida do objeto.

O construtor pode ser visto como uma função que pega um estado não inicializado representado por *start* e retorna um estado inicial:

$$\begin{aligned} &b :: OStart \rightarrow OData \\ &b = b?start \rightarrow b.corpo \triangle b.fail \rightarrow b!start \\ &b.corpo = \langle inicia\ odata \rangle \rightarrow bi!odata \end{aligned}$$

um destrutor é praticamente a operação inversa (leva um estado final)

$$\begin{aligned} &di :: OData \rightarrow OFinal \\ &di = di?odata \rightarrow di.corpo \triangle di.fail \rightarrow di!odata \\ &di.corpo = \langle altera\ odata\ para\ odata' \rangle \rightarrow di!final \end{aligned}$$

e as ações normais

$$\begin{aligned} &ai :: OData \rightarrow OData \\ &ai = ai?odata \rightarrow ai.corpo \triangle ai.fail \rightarrow ai!odata \\ &ai.corpo = \langle altera\ odata\ para\ odata' \rangle \rightarrow ai!odata' \end{aligned}$$

Observação, apesar de escrever $ai : OData \rightarrow OData$, não é necessário que o objeto tenha todos os slots de $OData$, apenas aqueles que são usados por $a1$. Isto é, a expressão deveria ser $ai : u \rightarrow u$, tal que $slotsUsedInAction(ai) \subseteq slots(u) \subseteq slots(OData)$.

No caso de um objeto sendo usado por outro objeto, após o tratamento de *fail*, é necessário acrescentar *raiseFail*. O objetivo de *raiseFail* é lançar uma evento *fail* para ser interceptado um nível acima. Por exemplo:

$$(\ b \triangle o1.a1.fail \rightarrow t1 \rightarrow o1.a1.raiseFail\ ) \triangle oa.a1.fail \rightarrow t0$$

Onde $o1.a1.raiseFail$ causa o evento $oa.a1.fail$.

Caso a ação tenha parâmetros de entrada e saída

$$\begin{aligned} ai.corpo = {} & ai.paramj?parInName \rightarrow \\ & \langle altera\ odata\ para\ odata' \rangle \rightarrow \\ & ai.paramk!parOutName \rightarrow \\ & ai!odata' \end{aligned}$$

Caso tenha parâmetros de entrada e saída, e condição de ativação

$$\begin{aligned} ai.corpo = {} & ai.paramj?parInName \rightarrow \\ \text{if} \quad & \langle enb\ condicao \rangle \text{ then} \\ & \langle altera\ odata\ para\ odata' \rangle \rightarrow \\ & ai.paramk!parOutName \rightarrow \\ & ai!odata' \\ \text{else} \quad & \\ & raiseFail \end{aligned}$$

Sendo que os eventos observáveis:

$$\begin{aligned} & ai.corpo = ai.paramj?parInName \rightarrow \\ & (ai.paramk!parOutName \rightarrow ai!odata') \sqcap raiseFail \end{aligned}$$

A semântica do objeto o seria

$$\begin{aligned} & o = b \rightarrow X \\ & X \rightarrow (a1 \rightarrow X \ |||\ ..\ |||\ an \rightarrow X \ |||\ d1 \ |||\ ..\ |||\ dn) \\ & execucao = tr(o \setminus \{request, paramactions\} \parallel escalonador \parallel init!start) \end{aligned}$$

O estilo usado assume a existência de um escalonador que pode passar o estado do objeto para qualquer ação. Um escalonador bem simples que pode ser usado é:

$$\begin{aligned} escalonador(o) = {} & b?o \sqcap (\sqcap_{i:\{1..n\}} ai?o) \sqcap (\sqcap_{i:\{1..n\}} di?o) \sqcap init?o \rightarrow \\ & \text{if } o = final \text{ then } Stop \\ & \text{elseif } o = start \text{ then } b!o \rightarrow escalonador \\ & \text{else } (\sqcap_{i:\{1..n\}} ai!o) \sqcap (\sqcap_{i:\{1..n\}} di!o) \rightarrow escalonador \end{aligned}$$

4.5.2 Objeto Agregado

Assumindo dois objetos componentes, $o1 : O1$ e $o2 : O2$, e uma agregação OA cujo estado é representado por $O1Data \times O2Data$, uma ação a é composta por:

1. receber o estado da agregação OA e enviar para os componentes $a?(od1, od2) \rightarrow (o1.ai!od1 \parallel o2.aj!od2)$

2. no caso de haver junção de parametros $(o1.ai.parOut?p \rightarrow o2.aj.parIn!p)$
3. composição de um novo estado OA' a partir do resultado dos componentes $(o1.ai?od1' \parallel o2.aj?od2') \rightarrow a!(od1', od2')$
4. seguido pelo tratamento de falhas $\triangle\, fail \rightarrow a!(od1, od2)$
5. se usado por outro objeto *raiseFail*

A execução de uma agregação pode ser caracterizada como:

$$\begin{aligned} execucao(OA) \;=\; & tr(o1 \setminus \{\text{request, param não usados por OA}\} \\ & \parallel escalonador(o1) \setminus \{\text{acoes usadas por OA}\} \\ & \parallel o2 \setminus \{\text{request, param não usados por OA}\}) \\ & \parallel escalonador(o2) \setminus \{\text{acoes usadas por OA}\} \\ & \parallel OA \setminus \{\text{request, param}\} \\ & \parallel escalonador(OA) \\ & \parallel OA.init!start) \end{aligned}$$

4.5.3 Objeto Reificado

Assumindo um objeto base $o1 : O1$, e um OR uma reificação. Uma ação a de reificação é composta da seguinte forma por:

1. no caso de seq o1.ai o1.a2 end seq

$$\begin{aligned} & (a?od \rightarrow o1.a1!od; \\ & \quad o1.a1?temp \rightarrow o1.a2!temp; \\ & \quad\quad a1.a2?od2 \rightarrow a!od2) \end{aligned}$$

2. no caso de cps o1.ai o1.a2 end cps

$$\begin{aligned} & (a?od \rightarrow split!od?(od1, od2, od3) \rightarrow \\ & \quad (o1.a1!od1?od1' \parallel o1.a2!od2?od2') \rightarrow \\ & \quad\quad join!(od1', od2', od3)?od' \rightarrow a!od') \end{aligned}$$

   onde *split* é uma função que separa od em 3 partes disjuntas ($od1$ contendo os campos usados por $o1.a1$, $od2$ contendo os campos usados por $o1.a2$, e $od3$ contém os campos não usados), *join* é a função para junta-los .

3. Seguido de $\triangle\, fail \rightarrow a!od$
4. e $\rightarrow$ *raiseFail* se usado por outro objeto

A execução de uma reificação pode ser caracterizada como:

$$\begin{aligned} execucao(OR) \;=\; & tr(o1 \\ & \parallel OR \setminus \{\text{request, param}\} \\ & \parallel escalonador(OR)) \end{aligned}$$

5 MODIFICAÇÕES NA LINGUAGEM

Este capitulo é um basicamente uma coletânea de estudos sobre possíveis mudanças na linguagem visando melhorar sua usabilidade. A seção de parâmetros serviu como um prólogo para uma solução mais satisfatória (apresentada no capitulo seguinte). A seção sobre recursão explica sua atual ausência em Nautilus e procura explorar possíveis caminhos para sua inserção. A seção sobre comandos de fluxo compara `if` não determinista e `alt`, mostrando a utilidade de sua adição.

5.1 Simplificando Parâmetros

Conforme mostrado anteriormente a semântica de Nautilus permite o uso de parâmetros cruzados. Por exemplo seja uma ação

```
a1
  out a1out
  in a1in
```

composta com

```
a2
  out a2out
  in a2in
```

tal que `a1out` → `a2in` e `a2out` → `a1in`. O uso de `agg` e `ret` permite que isso seja possível mas estabelece casos de falha que podem ser encontrados apenas observando o corpo das ações, isto é, se um programador junta duas ações observando apenas a compatibilidade das interfaces, ele pode encontrar casos onde a semântica da ação composta é vazia (não faz nada).

No caso de ações básicas pensou-se que esse problema poderia ser simplificado retirando-se a cláusula `agg`. Consideremos o caso em que o corpo das ações permite a junção bem sucedida de parâmetros cruzados (basicamente a situação onde os parâmetro de entrada da uma das ações são fornecidos pela `agg` de outras).

Uma ação

```
act a
  agg aout = ⟨exp⟩
  ⟨body a⟩
```

pode ser perfeitamente fatorada em

```
act a'
  ret aout = ⟨exp⟩
```

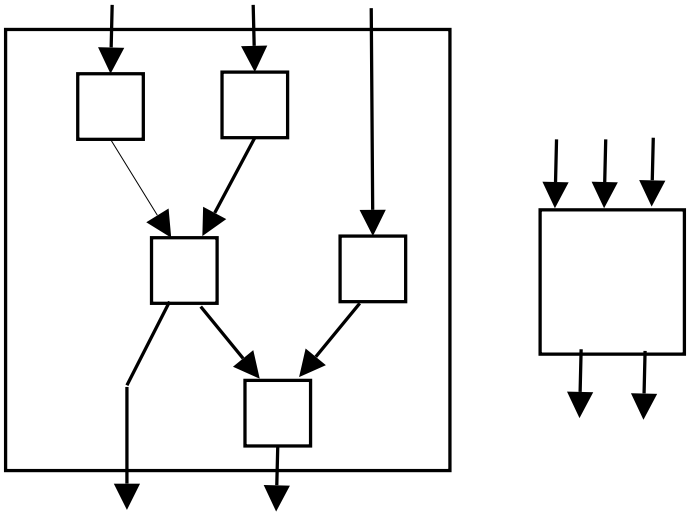

Figura 5.1: Exemplo de Parâmetros em ações 1

act a′′
⟨*body a*⟩

---

tornando o código mais simples e os casos de falha mais evidentes, desta forma qualquer composição de ações simples com parâmetros cruzados se torna um caso de falha, não é mais necessário analisar o código interno das ações componentes a procura das cláusulas `agg` e `ret`.

Entretanto a solução proposta apresenta problemas, ao contrário de `a`, o conjunto de `a′` e `a′′` não são uma transação. E se essa característica for necessária, acaba-se simulando `agg` através de reificação em `a′` e `a′′`.

Outra idéia foi tornar a interface mais rica usando-se `pre−out` e `pos−out` no lugar de `out` e renomear `agg` para `ret` já que a função é a mesma apenas em posições distintas. Mas esta solução ainda era insatisfatória para o caso de ações compostas pois a ordenação dos parâmetros derivados pode ser mais flexível do que a especificável por `pre−out` e `pos−out`.

De fato, esta flexibilidade do sistema de parâmetros derivados foi devidamente compreendida tardiamente. O protótipo seguiu uma abordagem mais simples do que a especificada pela linguagem quanto aos parâmetros derivados. A implementação foi construída sob a assunção de que parâmetros derivados de todas as entradas são definidas antes da chamada e todas as saídas estão definidas ao final (fig. 5.1).

Mas com a execução de exemplos percebeu-se que a especificação dos parâmetros derivados era mais complexa e permite o envio e a recepção de parâmetros dentro das ações compostas (fig. 5.2).

Isto é uma outra fonte de parâmetros cruzados, mas ao contrário do caso de parâmetros de ações simples cuja fatoração é simples, no caso de ações compostas a abordagem da especificação realmente adiciona mais expressividade (e reescrever programas que usam esta flexibilidade extra de parâmetros cruzados, restringindo-os de forma a ter entrada e saída bem definidas é no mínimo trabalhoso). O custo desta flexibilidade extra na linguagem original era ir contra a noção de encapsulamento de caixa preta nas ações compostas. Isto é, o programador precisa observar a estrutura das ações em vez de apenas usar as interfaces.

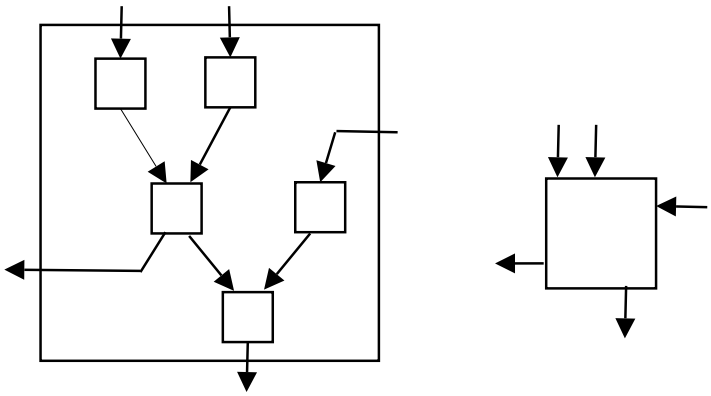

Figura 5.2: Exemplo de Parâmetros em ações 2

O tradutor protótipo inicial acabou sendo implementado assumindo-se a extensão natural da primeira simplificação proposta que seria estender a proibição de parâmetros cruzados para ações compostas. Tal restrição eliminou uma gama de programas válidos, mas podia ser defendida pelo incentivo a abstração. Mas uma solução mais satisfatória foi encontrada enriquecendo-se a interface através de um sistema de protocolos exposto no cap. 6.

5.2 Adicionando recursão

Recursão não foi implementada em Nautilus pois uma ação recursiva pode não terminar ferindo a restrição de atomicidade de ações, mas sempre se planejou adicionar alguma espécie de recursão segura que garantisse a terminação da ação.

Nesta seção são consideradas algumas alternativas para inspirar recursão em Nautilus.

5.2.1 Estado Atual

Primeiro apresentemos a situação atual de Nautilus. Atualmente Nautilus não dispõe de recursão, nem de comandos iterativos explícitos. A forma como a iteração é obtida é através da semântica autoexecutável das ações. Por exemplo, uma maneira de especificar a função fatorial em Nautilus é

```
object Factorial
export
  New in N:natural
  Done out Fat:natural
category
  birth request New
  death request Done
body
  slot N:natural
  slot Fact:natural

  act New
    val Fact << 1
    val N << New.N

  act Step
```

```
    enb N > 1
    cps
      val N << N − 1
      val Fact << Fact ∗ N
    end cps

  act Done
    enb N = 1
    ret Done.Fact = Fact
end Factorial
```

Considerando as presentes restrições de Nautilus (sem recursão ou comandos iterativos), a definição não é ruim. O problema é que não é possível transforma-la em uma ação atômica. Poderia se pensar que uma reificação como a mostrada a seguir poderia transforma-la em uma única ação,

```
object Factorial1 over Factorial
export
  Exec in N: natural
         by New.N
       out Fat: natural
         by Done.Fat
category
  birth death request Exec
body
  act Exec
  seq
    New
    Done
  end seq
end Factorial1
```

mas isso não funciona pois a ação `Step` não pode ser executada. Uma especificação correta seria algo como:

```
act Exec
seq
  New
  Step∗
  Done
end seq
```

Onde * simboliza um número indefinido de execuções. Mas é claro se isso fosse permitido perderíamos todas as garantias de terminação uma vez que, a ação `Step` poderia executada para sempre.

Como é impossível transforma-la em uma ação atômica, todos os objetos que usam o objeto `Factorial` precisariam de duas ações (uma para `New`, e outra para `Done`), e isso se mantém de forma transitiva (se outro objeto interage com um objeto que interage com `Factorial`, vai acabar precisando de duas ações). O problema lembra o antigo problema da programação funcional pura onde era necessário carregar o estado como parâmetros extras durante a execução.

Existe duas maneiras de consertar a situação: comandos iterativos ou recursão. Comandos iterativos tradicionais incluem `while` e `for`. Na maioria das linguagens de programação ambos são demasiamente expressivos para o fim pretendido pois não há garantia de terminação. Uma exceção é o `for` como implementado em Ada,

```
for P in 1..100 loop
        Put(P);         -- variavel de controle P
end loop;
```

onde a variável de controle `P` não pode ser alterado de dentro do loop. Isto assegura que o número de vezes que um loop `for` pode ser executado é fixo. Portanto é uma construção que poderia ser inclusa em Nautilus sem problemas semânticos.

A principal razão pelo qual focamos principalmente em recursão nesta seção e não comandos de iteração é que, embora ambos sejam em princípio igualmente expressivos, soluções baseadas em recursão tendem a serem mais elegantes e há mais literatura sobre propriedades de terminação.

5.2.2 Recursão com garantia de terminação dentro de ações

A linguagem Nautilus não permite recursão devido aos possíveis problemas em relação a atomicidade das ações. O objetivo foi adicionar recursão de forma restrita de tal forma que garantisse atomicidade (a computação necessariamente termina). A garantia de terminação está relacionada com o conceito de normalização forte. Isto significa que os termos de um cálculo sempre se reduzem a uma forma normal, por exemplo o cálculo lambda não tipado dispõe do mesmo poder computacional de uma máquina de Turing e portanto pode representar funções parciais e não apresenta esta propriedade, para obter a normalização forte é necessário restringi-lo com um sistema de tipos.

Uma vez que programas que não terminam são úteis, as linguagens funcionais tipadas de uso geral implementam versões enriquecidas de cálculo lambda tipado (ou sistemas formais similares) com construções como *letrec* que permitem recursão plena. Neste caso o sistema de tipos destas linguagens serve como documentação formal e elimina erros em tempo de compilação, não garante terminação.

Existem diversos formalismos com sistemas de tipos que têm a característica de normalização forte. Entre outros, o cálculo lambda tipado simples, System F (GIRARD; LAFONT; TAYLOR, 1989)(também chamado de cálculo lambda paramétrico), e a teoria de tipos de Martin-Lof (THOMPSON, 1999). O foco desta subseção será na linguagem Charity (COCKETT; FUKUSHIMA, 1992; COCKETT, 1996). A escolha é basicamente por causa das suas origens em um domínio categórico, interessante uso de combinadores e dispor de uma implementação razoavelmente bem documentada.

Charity é uma linguagem funcional/categorial que garante terminação. Ela utiliza tipos de dados indutivos e coindutivos (que podem ser utilizados para obter estruturas de dados potencialmente infinitas) usando formas restritas de recursão que se baseiam na estrutura dos tipos (combinadores *case*, *fold*, *map* no caso de tipos indutivos, *record*, *unfold* e *map* no caso de tipos coindutivos) garantindo terminação dos programas.

A abordagem parece, até certo ponto, ser adaptável para Nautilus. Os tipos coindutivos são infinitos e se implementados de forma mais tradicional (avaliação estrita) certamente levam a programas que não terminam (Charity usa avaliação procrastinada através de chamada sob demanda com interação). Como a avaliação procrastinada não funciona bem com linguagens com estado (o caso de Nautilus), os tipos coindutivos se tornam inviáveis. Os tipos indutivos parecem mais compatíveis com Nautilus e poderiam ser adicionados juntos com os respectivos combinadores *fold*, *case* e *map* (embora certamente algumas restrições precisem ser acrescentadas

como por exemplo a restrição de não alterar as variáveis sendo percorridas). Alguns tipos predefinidos como real não poderiam ser tratados por esse tipo de recursão, mas tipos predefinidos como natural que podem ser definidos indutivamente poderiam.

E restringindo-se o trabalho a tipos indutivos, poderia-se adotar a notação mais tradicional como Haskell (JONES, 2003), no lugar da notação categórica seguida por Charity. Isto é usar,

```
data Nat = Zero | Succ Nat
```

no lugar de

```
data nat -> C = zero: 1-> C
              | succ: C -> C
```

Sob o aspecto prático as principais restrições ao sistema de tipos indutivos de Charity em relação a uma linguagem funcional mais convencional como Haskell parece ser a obrigação da existência de todos construtores nas expressões de matching e a ausência de tipos mutuamente recursivos. Aparentemente são essas restrições permitem que a recursão seja baseada em *fold* (definida de forma automática sobre a estrutura do tipo) e ter semântica baseada em *Set* no lugar de *CPO* (SCHMIDT, 1986). Um exemplo de função usando *fold* é apresentado abaixo

```
def add : nat * nat -> nat
    = (x,y) => {| zero: () => y
                | succ: n => succ n
                |} x.
```

já a equivalente em Haskell é

```
foldNat::(a -> a) -> a -> Nat -> a
foldNat f z Zero = z
foldNat f z (Succ n) = f (foldNat f z n)

add x y = foldNat (Succ) x y
```

as diferenças neste caso são devidas ao fato de que em Haskell *fold* não é definido automaticamente e o uso de açúcar sintático por Charity usando operadores `{|`, `|` e `|}`.

Um exemplo usando outra estrutura de dados (lista) é

```
data list(A) -> C = nil: 1 -> C
                  | cons: A * C -> C.

def addLis:list(nat) -> nat
        = L => {| nil: () => zero
                | cons: z => add z
                |} L.
```

onde é feito a soma de uma lista.

Um fato importante é que o uso dos combinadores pode ser em boa medida escondido do usuário. Por exemplo, Haskell lida com operadores *map*, *fold* e *filter* (que como *map* é uma versão especializada de *fold*) de forma implícita em listas usando um açúcar sintático chamado de compreensão de lista. Por exemplo:

```
Input: [(x,y) | x <- [1,3,5], y <- [2,4,6], x<y]
InputComb: filter (\(x,y) -> x<y )
            (foldr (\x -> \z ->
              (map (\y -> (x,y)) [2,4,6]) ++ z) [] [1,3,5])
Output: [(1,2),(1,4),(1,6),(3,4),(3,6),(5,6)]

Input:[ 2 * x | x <- [1,2,3] ]
InputComb: map (\x -> 2 * x) [1,2,3]
Output: [2,4,6]

Input: [ x*y | x <- [1,2,3,4], y <- [3,5,7,9]]
InputComb: foldr (\x -> \z ->
            (map (\y -> x*y) [1,2,3,4]) ++ z) [] [3,5,7,9]
Output: [3,5,7,9,6,10,14,18,9,15,21,27,12,20,28,36]
```

onde Input é uma expressão usando compreensão de lista, InputComb uma tradução usando combinadores e Output é o resultado da avaliação da expressão.

Uma variação desta sintaxe foi incorporada em Python:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

Python (ROSSUM, 2002) é uma linguagem semanticamente similar a Lisp (isto é, variáveis sem tipos e permite o uso de múltiplos paradigmas), mas seu principal paradigma é a orientação a objeto. Enquanto combinadores eram uma característica controversa entre a comunidade Python, compreensão de lista se tornou um estilo universalmente aceito, portanto isto indica que pode ser uma maneira menos traumática de usar combinadores em uma linguagem imperativa.

Uma linguagem que incorpora uma variação de compreensão para outro tipo de dado além de listas é CLEAN (PLASMEIJER; EEKELEN, 2001)(uma linguagem similar a Haskell, mas que utiliza outra abordagem para lidar com estado), onde está pode ser usada com arrays estáticos destrutivos (embora este tipo de dados seja recomendado apenas em casos de performance crítica, e a notação pareça meio deslocada em relação a boa integração com listas).

Então se a notação de compreensão é muito boa para listas, e pode ser usada mesmo para arrays destrutivos, então é provável que possa ser utilizada para todo tipo de coleção. Entretanto enquanto coleções são um dos principais usos de tipo algébricos é bem dúvidoso que a notação de compreensão seja adequada para todos os tipos criados pelo usuário (por exemplo, o tipo Nat definido anteriormente).

E ainda haveria a questão de como a recursão iria interagir com a noção de estado e efeitos colaterais presentes em Nautilus (por exemplo se o passo de *fold* incluísse novos elementos na lista que esta percorrendo, isso eliminaria a garantia de terminação).

5.2.3 Recursão fora das ações - Uso de sublinguagem

O problema de como a recursão interage com os efeitos colaterais da linguagem pode ser resolvida de forma bastante simples criando-se uma sub-linguagem funcional pura. Neste caso as ações permanecem não recursivas, mas elas podem chamar funções. Funções não podem chamar ações. Uma solução similar foi usada por

(BARBOSA et al., 2001), onde uma linguagem imperativa similar a Pascal pode chamar cláusulas referencialmente transparentes de Prolog puro, mas as cláusulas não podem chamar procedimentos da linguagem imperativa.

Seguindo o objetivo original recursão com garantia de terminação, a sub-linguagem poderia ser qualquer linguagem funcional pura que garanta terminação como a já mencionada linguagem Charity, o cálculo lambda tipado simples ou System F (com o devido açúcar sintático e enriquecido com alguns tipos predefinidos).

Entretanto uma vez que fora das ações as funções interferem o mínimo possível com o fluxo das ações, talvez uma abordagem pragmática mais expressiva seja usar funções com recursão plena. Já que a recursão seria permitida apenas entre funções, isso deixa ao programador o encargo de garantir a terminação da ação, mas garante que a recursão não interfere em ações que estejam executando simultaneamente (essas funções não permitiriam mudanças em slots de objetos).

Seguindo um pouco mais no estudo desta última abordagem, resolveu-se inicialmente utilizar uma sintaxe de funções similar a linguagens imperativas de forma a não destoar muito do resto da sintaxe textual de Nautilus:

```
fun f (a1:⟨type⟩, a2:⟨type⟩):(a3:⟨type⟩, a4:⟨type⟩)
var ..
body
 ..
end fun
```

neste caso `f` é um identificador de função e `(a1:<type>, a2:<type>)`, `(a3:<type>, a4:<type>)` são tuplas.

A intenção seria permitir algo como

```
fun f (a1:integer, a2:integer):(a3:integer, a4:integer)
..

fun g (p1:integer, p2:integer):(integer)
var t:tuple
     r:integer
body
  val t << f(p1, p2)
  val r << t.a3 + t.a4
  ret r
end fun
```

Se a sub-linguagem têm recursão plena, a próxima questão é quanto a permitir efeitos colaterais restritos nas funções:

- Saída de dados

  É importante e deveria ser permitido pelo menos para debugging;

- Entrada de dados

  Menos importante, é preferível deixar exclusivo de ações;

- Mudança de estado nos parâmetros de entrada

  A defesa para essa característica seria buscar maior eficiência em casos como arrays. A abordagem exigiria adicionar outra palavra chave tal como `proc` para distinguir casos de alteração de estado senão o programador pode ter surpresas desagradáveis. Um exemplo clássico é "reverse" (que retorna uma

estrutura de dados em ordem reversa) se esta fosse implementada de forma destrutiva usando "funções" com efeitos colaterais. Outro contra-exemplo é esta tentativa de definir uma variante destrutiva de `map` (uma função que pega uma função e aplica em todos os elementos de uma coleção obtendo uma nova coleção) sem diferenciação de `proc` e `fun`:

```
fun map ((fun f (integer):(integer)), m:integer[10])
         :(m:integer[10])
  fun loop (i:integer, m:integer[10]):(integer[10])
  body
   alt
     enb (i = 10)
     val m[i] << f(m[i])
     ret m
   alt
     enb (i /= 10)
     val m[i] << f(m[i])
     ret loop (i+1) m
   end if
  end fun
body
  ret loop (1, m)
end fun
```

Assumindo que a sintaxe acima fosse válida esta "função" sobrescreveria o vetor original (em outras palavras é baseado em transformação de estado, em vez de ser baseado na avaliação de uma expressão). Além disso a distinção `proc` e `fun` seria útil por exemplo nas condições `enb` que poderia chamar funções, mas não procedimentos.

Além da questão de `proc` e `fun`, um problema da função `map` definida acima é a função auxiliar `loop`, embora ela possa ser implementada de forma eficiente (recursão de cauda pode ser implementada sem alocação de pilha), em ambientes como Java esta implementação não é simples. Outro problema de `loop` é a legibilidade, compare com a versão abaixo

```
proc map (in (fun f (integer):(integer)),
          in out m:integer[10])
var
   i:natural
body
  for i << 1 to 10 do
    val m[i] << f(m[i])
  end for
end proc
```

O que é uma boa indicação da praticidade de adicionar `for` e `while`.

Se `proc` fosse incluído, isso acarretaria uma hierarquia onde ações podem chamar funções ou procedimentos (exceto na parte de pre-condição que seria restrito a funções), procedimentos podem chamar procedimentos ou funções, e funções poderiam chamar apenas funções.

Existem dois problemas com esta extensão: Um problema pragmático é a possibilidade dos programadores optarem por deixar de lado as primitivas originais de Nautilus e só programarem usando procedimentos e funções (similar ao problema

de programadores C que migram para Java e programam de forma imperativa ignorando a maior parte das características da orientação a objeto). Um problema de natureza teórica da recursão plena é que a linguagem acaba fugindo do domínio semântico original onde as ações deviam terminar.

5.2.4 Conclusões

Vimos aqui duas possíveis formas de recursão que poderiam ser utilizadas para adicionar recursão em Nautilus. Uma baseada em combinadores que poderiam ser adaptados para serem usadas dentro de ações e outra adicionando-se uma nova construção para definição de funções.

A noção de combinadores é mais limitada, mas parece ser uma adição que pode ser integrada de forma mais uniforme dentro da linguagem. Entretanto mais trabalho teria de ser feito sobre quais restrições devem ser colocadas sobre as ações internas para que não se perca a garantia de terminação.

A definição de funções fora de ações é mais flexível. Devido a natureza mais isolada destas pensou-se mesmo a possibilidade de recursão plena e comandos iterativos, que por razões de natureza prática e teórica se revelou uma idéia pouco atrativa. Entretanto, mesmo restringido-se a recursão com garantia de terminação, as possibilidades expressivas são consideravelmente maiores (existem diversos sistemas de tipos com garantia de terminação que poderiam ser usadas como base).

As duas alternativas não são tão distintas, uma vez que a noção de combinadores precisaria de uma função como parâmetro do combinador (possivelmente uma versão disfarçada de uma abstração lambda). Em geral, a abordagem baseada em combinadores parece um pouco mais promissora e poderia ser mais desenvolvida em trabalhos futuros.

Para fins mais imediatos a incorporação de uma implementação de `for` como o de Ada já seria uma boa ajuda.

5.3 Sobre extensões ao sistema de tipos de Nautilus

O atual sistema de tipos de Nautilus é demasiadamente simples. Para tornar a linguagem mais prática será necessário a adição de tipos estruturados. A idéia de usar as classes de Nautilus, pelo menos da forma que foram originalmente propostas, como substitutos de tipos estruturados apresenta problemas, tais como falta de expressividade (não tendo definições recursivas, não é possível representar árvores de forma direta por exemplo) e dificuldade de interação com elas (a sintaxe de interações foi criada propositadamente para ser trabalhosa e fazer as pessoas evitarem seu uso).

Como se pretende adicionar recursão limitada, a escolha do sistema de tipos está também ligada com as possíveis extensões funcionais que se pretenda adotar. Por exemplo, o sistema de tipos do cálculo lambda tipado simples é minimalista e pouco expressivo (árvores não são definíveis), enquanto o sistema de tipos de System F é praticamente uma outra linguagem funcional no nível de tipos. Se a adição de recursão for por meio de uma sublinguagem, pode-se adicionar tipos estruturados de forma ortogonal as classes. Mas ao se optar por seguir uma abordagem mais integrada as classes deveriam ser modificadas de forma a possibilitar seu uso como registros como em Java e adicionar alguma construção para lidar com somas disjuntas (evitando provavelmente um dos piores erros no sistema de tipos original de Java).

5.4 Comparação de comandos de controle de fluxo - `alt` e `if`

Esta seção surgiu de uma discussão sobre a conveniência de outras instruções de controle de fluxo (como `if` ou `case`) para substituir ou pelo menos complementar a combinação obtida por `alt` e `enb`. O resultado foi esta comparação entre `if` não determinista e `alt`. A escolha de `if` não determinista como na Guarded Command Language de Dijkstra em vez da versão determinista de Pascal é basicamente por ser mais similar em estilo a Nautilus e facilitar comparações. A conclusão mais importante desta comparação é que é um sistema de controle de fluxo de execução com composição mais direta como `if` é bem mais prática do que a indireta obtida por `alt` e reificações.

Peguemos como exemplo um `if` composto como

```
if c1 –>
  s1;
  if c3 –> s3;
  [] c4 –>
    s4;
    if c5 –> s5
    [] c6 –> s6
    fi;
    s4'
  fi
[] c2 –> s2
```

Pensou-se em duas possíveis abordagens para traduzir o código acima em Nautilus: usando reificações ou variáveis auxilares. No primeiro caso, ignorando o código de declaração de objetos base e de reificação, devido ao fato de `alt` ser sintaticamente restrito a estar logo depois de ações é necessário linealizar as opções mais básicas como

```
act a1
  enb c1
  s1

act a2
  enb c2
  s2

act a3
  enb c3
  s4

act a4
  enb c4
  s4

act a4'
  s4'

act if_a56
  alt 5
    enb c5
    s5
  alt 6
```

```
    enb c6
    s6
```

e depois compor em diversas reificações

```
// reificação 1
act a4c
  seq
    a4
    if_a56
    a4'
  end seq

// reificação 2
act if_a34
  alt
    a3
  alt
    a4c

// reificação 3
act a
  alt
    seq
      a1
      if_a34
    end seq
  alt
    a2
```

Verifica-se portanto que que esta solução acarreta um trabalho considerável e o resultado final está bem longe da legibilidade do original.

A outra abordagem é usar uma variável auxiliar State que têm valor inicial zero

```
{State =0 }
if c1 –>
  s1;
  {State =1}
  if c3 –> s3;
  [] c4 –>
    {State =2}
    s4;
    {State =3}
    if c5 –> s5
    [] c6 –> s6
    fi;
    {State =4}
    s4'
  fi
[] c2 –> s2
{State =5}
```

Assim o código Nautilus seria

```
act a1
  enb c1 and (State = 0)
```

```
  s1
  val State << 1

act a2
  enb c2 and (State = 0)
  s2
  val State << 5

act a3
  enb c3 and (State = 1)
  s4
  val State << 5

act a4
  enb c4 and (State = 1)
  s4
  val State << 2

act a4'
  enb State = 4
  s4'
  val State << 5

act if_a56
  alt 5
    enb c5 and (State = 3)
    s5
    val State << 4
  alt 6
    enb c6 and (State = 3)
    s6
    val State << 4
```

É mais simples do que a abordagem anterior, alias um padrão bem similar pode ser encontrado em muitos exemplos de especificações Nautilus, mas continua mais extenso e menos legível do que o `if` original e gera código menos otimizado do a versão com as reificações.

O caminho inverso de `alt` para `if`, com uma expressão como

```
alt a1
  enb c1
  s1
alt a2
  s2
```

é praticamente direto

```
if c1 -> s1
[] True -> s2
fi
```

Um ponto a destacar, a tradução mostrada entre `if` e `alt` é uma aproximação. Existe uma diferença entre a semântica do `if` não determinista e e a semântica descrita para `alt`. A semântica de `alt` é angélica (isto é, escolhe um dos comandos cuja guarda é verdadeira E executam sem falhas) enquanto a semântica geralmente associada ao `if` é demoníaca (escolhe um dos comandos cuja guarda é verdadeira). Então a implementação de `if` é inerentemente mais simples e eficiente, mas o pro-

gramador em geral precisa lidar com mais precondições e/ou verificar se os comandos executaram sem falhas para obter exatamente o mesmo efeito que `alt`.

5.4.1 Conclusão

A construção de `if` é uma adição extremamente útil a linguagem. Não somente é mais expressiva (no sentido em que o código é mais claro e sucinto) do que as construções atuais como auxilia a obter uma implementação mais eficiente (o `if` transmite uma noção de escolha enquanto diversos `act a enb c` transmite a idéia de múltiplos processos potencialmente executando de forma paralela e portanto natural que sejam implementados desta forma).

5.5 Algumas considerações sobre a sintaxe

A sintaxe textual de Nautilus sempre foi um assunto relegado a segundo plano uma vez que é um aspecto de importância menor em comparação a semântica, e por causa dos trabalhos em direção a uma sintaxe diagramática. Entretanto uma notação textual têm algumas vantagens em relação a uma notação diagramática, e algumas mudanças sintáticas menores podem melhorar a linguagem.

- Obrigar ações a listar parâmetros de entrada e saída. A informação já existe na seção interface do objeto, mas essa redundância realmente é interessante, pois quando o programador esquece o nome do identificador do parâmetro, ele é obrigado a ir até o início do objeto procurar a informação, interrompendo o fluxo de trabalho. A falta de localidade na informação tende a ser irritante.

- Permitir o uso a convenção de posição no lugar de por nome na passagem de parâmetros. Enquanto a convenção por nome seja interessante quando há um grande número de parâmetros, uma convenção por posição provavelmente tornaria o código mais sucinto.

- Adotar a convenção `object.action` em vez de `action of object`. Puro açúcar sintático, mas é a ordem convencional das linguagens OO. Embora talvez seja interessante utilizar outro símbolo como ':' uma vez que '.' já é usado para fazer referencias a parâmetros e tornaria `foo.bar` ambíguo (ação `bar` do objeto `foo` ou parâmetro `bar` da ação `foo`?).

6 SISTEMA DE CHECAGEM DE PROTOCOLOS PARA PARÂMETROS

6.1 Introdução

O funcionamento de parâmetros de Nautilus é mais sofisticado do que de linguagens convencionais, pois estabelecem pontos de sincronização dentro das ações.

Como exposto anteriormente em 5.1, este sistema permite o estabelecimento de ações cuja semântica seja vazia (semanticamente pode ser vista como uma ação com a condição de enb false).

A correta implementação da semântica vazia de parâmetros apresenta complicações consideráveis. Em uma tradução seqüencial da ação composta, as ações componentes deveriam ser sequencializadas de forma a respeitar as restrições impostas pelos parâmetros, o caso de semântica vazia implica um caso onde isso não é possível. Isso significa que a ação falha de forma inevitável, embora sintaticamente seja uma ação perfeitamente válida.

Em uma implementação com uma thread por ação a intuição é que um thread consumidor espera até que um thread produtor forneça um valor. Mas existem mais detalhes a serem considerados. No caso da não execução, exibir um comportamento de deadlock não é uma interpretação válida da semântica pois poderia bloquear variáveis de slots, tornando-as não disponíveis. Uma maneira de implementar não execução é introduzir um thread de controle que periodicamente verificasse se todas as ações não estão bloqueadas, e se estivessem este thread controle deveria restaurar os valores e reiniciar os threads. Mas como os threads das ações envolvidas em um caso de não execução iriam sempre entrar em deadlock, uma alternativa melhor é verificar se as ações componentes podem ser executadas de forma viável antes da execução, na análise estática e isto pode ser obtido a checagem de protocolos proposto neste capitulo. Esta análise estática pode tornar proibida criar uma ação composta cuja cujo grafo de dependência de parâmetros seja inviável.

Considerando-se um paralelo com um sistema de tipos, uma ação como

```
act A1 // (in I, out O1, out O2)
  agg O1 = ⟨value⟩
  ⟨action  body⟩
  ret O2 = ⟨value⟩
```

pode ser vista como tendo um "tipo"(protocolo):

$$A1 :!O1 \rightarrow ?I \rightarrow !O2$$

Esta expressão é uma descrição do protocolo de envio e recepção dos parâmetros.

Usando convenções encontradas em álgebras de processos (CCS ou CSP), ! indica envio de parâmetros, e ? a recepção de parâmetros.

Essa informação necessária para o analisador estático também é útil para o desenvolvimento de programas. Um uso potencial seria o programador estabelecer uma expressão de um protocolo que deveria ser satisfeito pela ação (similar a programador que fornece o tipo da função antes do corpo).

6.2 Sintaxe e redução de protocolos

A sintaxe é dada pela seguinte BNF

$$P \ = ?a \to P \ \mid !a \to P \ \mid \ P \to P \ \mid \ P \| P \ \mid \ P \Box P \ \mid \ \varepsilon$$

onde $?a$ e $!a$ são parâmetros complementares (representam entrada e saída, respectivamente), $\to$ indica seqüencia, $\|$ paralelismo, $\Box$ não determinismo e ε é um tipo de elemento neutro.

As regras para redução são similares a de uma álgebra de processos como CSP ou CCS. A seqüencia

$$\frac{(a \to b) \to c}{a \to (b \to c)} \qquad \frac{\varepsilon \to a}{a} \qquad \frac{a \to \varepsilon}{a}$$

, a sincronização,

$$\frac{(!a \to P) \| (?a \to P)}{P \| Q} \qquad \frac{\varepsilon \| P}{P} \qquad \frac{P \| Q}{Q \| P} \qquad \frac{(P \| Q) \| R}{P \| (Q \| R)}$$

e substituição,

$$\frac{(c \to Q)[b/a]}{c[b/a] \to Q[b/a]} \qquad \frac{(P \| Q)[b/a]}{P[b/a] \| Q[b/a]} \qquad \frac{(P \Box Q)[b/a]}{P[b/a] \Box Q[b/a]}$$

onde $c[b/a]$ retorna b se $c = a$ senão continua como c. Quando não houver qualificador nas variáveis a e b de $[b/a]$ entenda como $[!b/!a, ?b/?a]$.

Inicialmente tentou-se formalizar a idéia de que a entrega de um valor produzido para um consumidor pode ser postergada em uma sincronização através de uma regra bem especifica:

$$\frac{(!a \to b \to P) \| (c \to Q) \qquad c \neq ?a \qquad ?a \in \alpha(Q)}{(b \to !a \to P) \| (c \to Q)}$$

onde $\alpha(Q)$ é uma expressão que retorna o alfabeto (os parâmetros presentes) do protocolo Q.

Entretanto verificou-se que uma regra mais simples:

$$\frac{(!a \to P) \| Q}{P \| Q \| (!a \to \varepsilon)}$$

estabelecendo a independência de $!a$ era suficiente.

No caso de não determinismo

$$\frac{P \Box P}{P} \qquad \frac{a \to (Q \Box R)}{(a \to Q) \Box (a \to R)} \qquad \frac{P \| (Q \Box R)}{(P \| Q) \Box (P \| R)}$$

A primeira regra para não determinismo é que se as alternativas tem o mesmo protocolo, pode ser eliminado. As outras são regras de distribuição.

6.3 Protocolo de ações primitivas

O protocolo de ações básicas pode ser obtido da seguinte forma:

---

export
 A1
  in i1:⟨*type*⟩
  in i2:⟨*type*⟩
  out o1:⟨*type*⟩
  out o2:⟨*type*⟩
..

act A1
  agg o1 = ⟨*value*⟩
  ⟨*bodyA*1⟩
  ret o2 = ⟨*value*⟩

$$\text{protocol}(A1) = !o1 \rightarrow (?i1 \| ?i2) \rightarrow !o2$$

O protocolo de um ação básica é sempre válido (não causa falhas).

6.4 Protocolo de ações compostas em agregação

Os identificadores são considerados qualificados, isto é, IdObj.IdAction.o1! no lugar de o1!.

Na exportação de parâmetros das ações compostas,

⟨*IdAction*⟩ **der** ⟨*in* | *out*⟩ a:⟨*type*⟩ **by** b

---

o parâmetro na posição a será chamado *derivado*, enquanto o parâmetro na posição b será chamado derivável ou *primitivo*.

Cada ação composta terá um conjunto de parâmetros derivados Der(*idAction*), um conjunto de deriváveis Prim(*idAction*) e um conjunto de substituição PD(*idAction*) onde cada elemento têm formato b/a.

6.4.1 Protocolos deterministas

A obtenção de um protocolo determinista para ação composta foi dividida em duas etapas: validação (onde se verifica que os protocolos das ações componentes se encaixam sem falhas) e o cálculo do protocolo resultante.

6.4.1.1 Validação

A intuição da regra de validação é simples, se há entradas e saídas que não sincronizam isso acarreta falha automática (ou seja as ações componentes entrariam em deadlock).

Uma ação composta é considerada válida se a sincronização dos protocolos das ações componentes, mais o complemento dos parâmetros deriváveis resulta em ε (soma zero).

$$\begin{aligned} \text{valid} &: IdAction \rightarrow Bool \\ \text{valid}(A) &\hat{=} \text{compProt}(A) \| \text{complPrim}(A) = \varepsilon \end{aligned}$$

A sincronização dos protocolos das ações componentes usa uma função matchs(A) que retorna uma lista de substituições, tal que cada cada `match` a b de A há um elemento b/a.

$$\begin{aligned} \text{compProt} &: \mathit{IdAction} \rightarrow \mathit{Protocol} \\ \text{compProt}(A) &\hat{=} (\|_{a \in (\text{components}(A))} \text{protocol}(a))[\text{matchs}(A)] \end{aligned}$$

$$\begin{aligned} \text{complPrim} &: \mathit{IdAction} \rightarrow \mathit{Protocol} \\ \text{complPrim}(A) &\hat{=} (\|_{p \in \text{Prim}(A)} \text{neg}(p)) \end{aligned}$$

$$\begin{aligned} \text{neg} &: \mathit{Param} \rightarrow \mathit{Param} \\ !a &\mapsto ?a \\ ?a &\mapsto !a \end{aligned}$$

Um exemplo de validação, tendo:

```
act AComp composed by
  A1 of Obj1
  A2 of Obj2
    match A1.O1 of Obj1
          A2.I2 of Obj2
    match A1.I1 of Obj1
          A2.O2 of Obj2
```

e assumindo que os protocolos de de A1 e A2 sejam

$$\begin{aligned} \text{protocol}(A1) &= !o1 \rightarrow ?i1 \\ \text{protocol}(A2) &= !o2 \rightarrow ?i2 \end{aligned}$$

o a validade de AComp pode ser calculado como

$$\begin{aligned} \text{compProt}(\mathit{Acomp}) &= A1 \| A2[i1/o2, i2/o1] \\ &= (!o1 \rightarrow ?i1) \| (!i1 \rightarrow ?o1) \\ &= (?i1 \| !o1) \| (!i1 \rightarrow ?o1) \\ &= ?i1 \| !o1 \| (!i1 \rightarrow ?o1) \\ &= !o1 \| (?i1 \| (!i1 \rightarrow ?o1)) \\ &= !o1 \| ?o1 \\ &= \varepsilon \end{aligned}$$

E portanto a ação composta Acomp é válida.

Se os protocolos de A1 e A2 fossem

$$\begin{aligned} \text{protocol}(A1) &= ?i1 \rightarrow !o1 \\ \text{protocol}(A2) &= ?i2 \rightarrow !o2 \end{aligned}$$

AComp seria calculado como

$$\begin{aligned} \text{compProt}(\mathit{Acomp}) &= A1 \| A2[i1/o2, i2/o1] \\ &= (?i1 \rightarrow !o1) \| (?o1 \rightarrow !i1) \end{aligned}$$

Que é irredutível e portanto a ação Acomp seria inválida.

6.4.1.2 Cálculo de protocolo

Com a validação da ação composta Acomp, o próximo passo é calcular o protocolo desta. Se não apresenta `der in` ou `der out` é simplesmente ε. Se apresenta, é uma expressão cujo alfabeto esta contido em Der($Acomp$). O protocolo pode ser obtido da seguinte forma:

$$\begin{aligned} \text{protocol} &: \mathit{IdAction} \rightarrow \mathit{Protocol} \\ \text{protocol}(A) &\hat{=} ((\|_{a \in \text{components}(A)} \text{protocol}(a))[\text{PD}(A)])[\varepsilon/p \in \alpha(\text{components}(A))] \end{aligned}$$

Isto significa que as ações componentes são sincronizados, os identificadores dos parâmetros primitivos são substituídos pelos parâmetros derivados e os outros são neutralizados.

Um exemplo de cálculo de protocolo é mostrado para o seguinte fragmento

```
object AT
..
export
  Atribui der in Valor: integer
    by Atribui.Valor of Tabela
..
act Atribui composed by
  Posição of Apontador
  Atribui of Tabela
    match Posição.Pos of Apontador
          Atribui.Ind of Tabela
```

o primeiro passo é a validação que será omitida, pois não apresenta diferenças significativas com os exemplos anteriores.

O protocolo das componentes:

$$\text{protocol(Apontador.Posição)} = !\text{Apontador.Posição.Pos}$$

$$\text{protocol(Tabela.Atribui)} = (?\text{Tabela.Atribui.Valor} \,||\, ?\text{Tabela.Atribui.Ind})$$

A lista de substituição baseada em parâmetros exportados:

$$\text{PD(AT.Atribui)} = [\text{AT.Atribui.Valor/Tabela.Atribui.Valor}]$$

A lista de substituição (neutralização) baseado no alfabeto das componentes:

$$\alpha(\text{Apontador.Posição}) = \{!\text{Apontador.Posição.Pos}\}$$

$$\alpha(\text{Tabela.Atribui}) = \{?\text{Tabela.Atribui.Valor}, ?\text{Tabela.Atribui.Ind}\}$$

$$\begin{aligned} [\varepsilon/\alpha(\text{components(AT.Atribui)}] \;=\; & [\varepsilon/!\text{Apontador.Posição.Pos}, \\ & \varepsilon/?\text{Tabela.Atribui.Valor}, \\ & \varepsilon/?\text{Tabela.Atribui.Ind}] \end{aligned}$$

$$
\begin{aligned}
\text{protocol(AT.Atribui)} &= (((!\text{Apontador.Posição.Pos}) \\
&\quad \|(?\text{Tabela.Atribui.Valor}\|?\text{Tabela.Atribui.Ind})) \\
&\quad [\text{AT.Atribui.Valor/Tabela.Atribui.Valor}]) \\
&\quad [\varepsilon/!\text{Apontador.Posição.Pos}, \\
&\quad \varepsilon/?\text{Tabela.Atribui.Valor}, \\
&\quad \varepsilon/?\text{Tabela.Atribui.Ind}] \\
\text{Portanto,} &= ((!\text{Apontador.Posição.Pos}) \\
&\quad \|(?\text{AT.Atribui.Valor}\|?\text{Tabela.Atribui.Ind})) \\
&\quad [\varepsilon/!\text{Apontador.Posição.Pos}, \\
&\quad \varepsilon/?\text{Tabela.Atribui.Valor}, \\
&\quad \varepsilon/?\text{Tabela.Atribui.Ind}] \\
&= (\varepsilon\|(?\text{AT.Atribui.Valor} \,||\, \varepsilon)) \\
&= ?\text{AT.Atribui.Valor}
\end{aligned}
$$

E como as ações de Nautilus não são recursivas, a expressão que expressa os protocolos é finita e tende a ser simples.

6.4.2 Protocolos não deterministas

Para considerar ações com múltiplas alternativas, algumas mudanças se tornam necessárias. As alternativas têm o mesmo alfabeto definido pela lista de parâmetros exportados, mas a ordenação pode ser diferente. Por exemplo,

```
act A
  alt A1
    agg O1 = ⟨valor⟩
    ⟨corpo⟩
    ret O2 = ⟨valor⟩
  alt A2
    ⟨corpo⟩
    ret O1 = ⟨valor⟩
    ret O2 = ⟨valor⟩
```

cujo protocolo é a expressão

$$(!o1 \rightarrow ?i \rightarrow o2)\Box(i? \rightarrow !o1 \rightarrow !o2)$$

Para facilitar a validação as regras de distribuição são usadas para manter os protocolos em uma forma normal prenex. Assim um protocolo não determinista ND pode ser interpretado como um conjunto de protocolos deterministas Di.

Assim a validação de uma ação composta $Acomp$ com duas ações componentes $A1$ e $A2$ com protocolos $A11\Box A12\Box A13$ e $A21\Box A22$ respectivamente, seria fornecido por

$$
\begin{aligned}
\text{valid}(Acomp) \;\hat{=}\; & \\
& ((A11 \,||\, A21)[\text{matchs}(Acomp)] \,||\, \text{complPrim}(Acomp) = \varepsilon) \vee \\
& ((A11 \,||\, A22)[\text{matchs}(Acomp)] \,||\, \text{complPrim}(Acomp) = \varepsilon) \vee \\
& \ldots \\
& ((A13 \,||\, A22)[\text{matchs}(Acomp)] \,||\, \text{complPrim}(Acomp) = \varepsilon)
\end{aligned}
$$

Para obter o protocolo de $Acomp$, elimina-se as combinações que falharam na validação e se aplica o procedimento para obtenção de protocolos deterministas

as combinações restantes, os protocolos resultantes são então reunidos pelo não determinismo.

6.5 Protocolos de ações compostas de reificação

Em reificações, não há matchs portanto todos os parâmetros são exportados. Assim se os protocolos das ações componentes eram válidos, o protocolo resultante será válido. No caso de `seq`, o protocolo resultante é o sequenciamento dos protocolos das ações componentes. No caso de `cps`, o protocolo resultante é a composição paralela dos protocolos (sem matchs não há sincronização interna).

7 ASPECTOS DE IMPLEMENTAÇÃO

7.1 Introdução

O mapeamento do subconjunto da linguagem descrito no capítulo 1 foi prototipado através de um programa Haskell que recebe o programa Nautilus e gera o código Java equivalente. Outros aspectos foram testados de forma manual (basicamente o autor fez o papel do tradutor).

Havia a intenção de criar um tradutor mais completo escrito em Java com um ambiente de programação junto com alunos de graduação de outra instituição, mas esta implementação acabou não sendo realizada. Este capítulo portanto descreve observações derivadas desta experiência e apresenta algumas sugestões de otimização para o caso de uma implementação mais ambiciosa.

7.2 Custo de transação

Nautilus foi idealizada como uma linguagem para ser utilizada em grandes programas concorrentes, já que foi inspirada em um formalismo (autômatos não sequenciais) que apresenta grandes propriedades de composicionalidade e que é naturalmente concorrente. Entretanto o aspecto transacional pode ser um grande problema para o seu uso no caso de sistemas distribuídos que seriam uma extensão natural para uma linguagem concorrente.

Essa característica já é um tanto cara mesmo para sistemas de memória compartilhada. Para um sistema ser transacional, ele precisa guardar o último valor válido até que todas as alterações sejam concluídas com sucesso. Isto é, no mínimo dobra a quantidade de memória necessária e se explora mais concorrência estabelece múltiplas cópias locais de estado para uso de threads.

Nautilus é construído sob a assunção que todas as computações são transações da mesma forma que CSP assume que comunicação é sempre síncrona. Em um livro sobre extensões concorrentes de ML (FLEMMING, 1997), os pesquisadores que escolheram primitivas síncronas se apoiavam em resultados de maior composicionalidade decorrente dessa escolha. Já os que escolhiam primitivas assíncronas apontavam que em sistemas distribuidos mensagens assícronas são a escolha mais natural e sincronicidade é cara (em redes o sincronismo só é alcançado através de um protocolo de mensagens assíncronas). O mesmo trade-off parece se aplicar de forma ainda mais forte ao sistema transacional de Nautilus (em banco de dados, transações distribuídas são obtidas através de um protocolo de duas fases que implicam em muitas mensagens trafegando na rede).

Uma possível forma de tentar minimizar este custo seria utilizar técnicas combi-

nadas de interpretação abstrata e análise de não interferência. Por exemplo,

```
slot a:integer

act add
  val a << a + 1
```

assumindo que o slot a só é alterado por add e dado que adição é uma operação que não causa falhas, neste caso a implementação canônica:

```
void add()
try {
  a.lock();
  add_body();
  a.commit();
} catch (e) {
  a.rollback();
}
}
```

poderia ser simplificada para

```
void add{
  a := a + 1;
}
```

Fazer esse tipo de otimização exigiria propagação de propriedades e interpretação abstrata, isto é, numa expressão $f(a, b)$ se as subexpressões a e b não causam falhas e f é total então a expressão $f(a, b)$ não causa falhas. Entretanto esse nível de análise estática provavelmente seria bem demorada.

7.3 Quebrando a abstração de atomicidade

Nesta seção são descritas situações onde por razões de implementação manter a abstração de atomicidade é inviável.

7.3.1 Entrada e Saída

Entrada e saída é um ponto onde optou-se por ignorar atomicidade de Nautilus. Ficou claro que E/S não pode ser incluído no conceito de atomicidade de ações (muitos casos onde não se pode reverter ao estado anterior, como caso de interação com o usuário, conexões em rede, etc.).

Logo no início dos trabalhos percebeu-se que adotar `read` e `write` como pertencentes a um objeto `System` como em Java seria desajeitado (devido ao fato do núcleo não incluir interações) e foi assumido taticamente que as instruções `read` e `write` são executadas de forma atômica nas ações e pretendia-se trabalhar com isso mais detalhadamente depois de mapear as outras construções de Nautilus.

Após feito o mapeamento do núcleo, e adicionando interações e classes, percebeu-se que o problema era mais difícil do que o inicialmente imaginado. A alternativa de tratar `read` e `write` como ações de um único objeto `System` era inviável, as regras de utilização de objetos não permitem que ele seja reusado (se uso ele junto com um objeto A, não poderia reutiliza-lo com um objeto B), mas acreditava-se que a adoção de classes resolveria o problema. Entretanto adotando classes poderia-se utilizar as instâncias de uma classe `System` mas ainda assim implicaria em um trabalho considerável em juntar os objetos e acarretaria certas restrições.

De qualquer forma, mesmo permitindo o uso de um único terminal, isso ainda acarreta o trabalho aborrecido das junções de objetos. Optou-se então por continuar a utilizar `read` e `write` sem objetos.

É evidente que `read` e `write` têm uma interação global com um objeto que representa o mundo externo implicitamente. Estendendo o uso de `read` e `write` com arquivos é interessante notar como isso pode ser usado de forma a passar por cima das abstrações. Por exemplo, sejam os objetos

```
object o1
category
  birth Start
body
  slot f:OutputFile
       i:integer
    // assumindo OutputFile
    // um manipulador de arquivos de saida

  act Start
    ⟨associa f com teste.txt⟩
    ⟨cria f⟩


  act Processa
    ⟨processa i⟩
    write(f,i)
    close(f)

end o1

object o2
category
  birth Start
body
  slot f:InputFile
       i:integer
    // assumindo InputFile
    //  um manipulador de arquivos de entrada

  act Start
    ⟨associa f com teste.txt⟩
    ⟨abre f⟩

  act Processa
    read(f,i)
    ⟨utiliza i⟩

end o1

object oa
  aggregation of o1 o2
```

Neste caso obtém-se comunicação entre o1 e o2 de forma indireta (o que pode ser bem útil).

Apenas uma observação: enquanto a manutenção da abstração de transação em todas as operações de E/S seja inviável, para alguns casos como a manipulação de arquivos com direito a leitura/escrita, a abstração de transação pode ser mantida (poderia se estender a abordagem utilizada com slots, adotar uma palavra-chave

como trans-file e trabalhar com arquivos de log no lugar de variáveis temporárias). A questão é se isso seria prático, isto é, se nas situações onde reversão ao estado anterior é possível e desejável como em banco de dados, essa propriedade vai coincidir com a duração de uma ação.

7.3.2 Tratamento de exceções

A implementação seguiu a idéia de que qualquer erro no meio da execução de uma ação acarreta o retorno ao valor inicial. Problemas com essa abordagem surgem com aspectos onde a noção de transação não é mantida como E/S. Para esses casos seria necessário estender a linguagem pois se as rotinas retornam valores de erro, precisa de primitivas condicionais internas (if, case) para agir de acordo com estes valores; ou se as rotinas podem levantar exceções, precisa de sistema de tratamento de exceções.

Um sistema de tratamento de exceções também se torna particularmente importante para uso de bibliotecas de outras linguagens como destacado a seguir.

7.3.3 Comunicação com outros programas

A não ser que se queira seguir um rumo puramente acadêmico, como Charity, é necessário reutilizar rotinas escritas em outras linguagens de programação. No caso do mapeamento o interesse óbvio seria reusar bibliotecas Java.

No caso de Java, o reuso de bibliotecas sem dependências de estado (como Math) não apresentam problemas de atomicidade (apesar de precisar tratar exceções).

Mas a maioria das rotinas não é tão bem comportada. Nestes casos provavelmente a solução mais razoável seria permitir ações que não respeitam atomicidade e etiqueta-las com uma palavra chave como "non-atomic". No caso de interação com Java a incorporação de um sistema de tratamento de exceções seria particularmente importante para interoperabilidade.

Outro aspecto é como seria a comunicação inversa, isto é, a chamada de código Nautilus por Java. As características de Nautilus tornam esta comunicação um pouco complicada. Programas Nautilus são entidades ativas, portanto as interações teriam que levar essa característica em conta, por exemplo poderiam usar um estilo de enviar os dados de entrada através de um procedimento e esperar que ao término o programa Nautilus fizesse uma chamada de callback.

7.4 Não determinismo

Existe um certo conflito entre os objetivos de uma linguagem de especificação e uma linguagem de programação. Enquanto existe uma ampla justificativa para manter uma especificação o mais não determinista possível garantindo mais opções, a implementação deveria ser o mais eficiente possível (o que significa levando-se as arquiteturas convencionais em retirar boa parte do não determinismo).

O atual sistema de não determinismo baseado em backtracking não seqüencial é interessante para fins didáticos, pois obriga o aluno a não depender de debugger e o força a pensar sobre o programa, similar a uma especificação em Prolog puro. Mas, como Prolog puro, isso tende a ser inerentemente ineficiente em máquinas convencionais e não é prático para uma linguagem de programação (a divisão em objetos de Nautilus tende a diminuir o espaço de buscas, mas ainda é ineficiente), em uma implementação eficiente é mais interessante ter construções de natureza

mais determinística e usar não determinismo apenas quando necessário.

O não determinismo explicito intra-ação particularmente apresenta algumas complicações devido a semântica angélica de que se existem diversas alternativas que falham e somente uma termina de forma bem sucedida, esta última deve acontecer. Esta semântica exigiu um controle de falhas que foi implementado como um vetor de controle de alternativas que falharam, simples, mas ineficiente. Embora o problema seja menos grave do que em Prolog, já que o espaço de estados é local e portanto tende a ser menor, essa característica pode se tornar pesada, uma vez a linguagem foi idealizada para ser usada de forma composicional.

Uma alternativa para tentar resolver o problema é aumentar a eficiência da implementação. Por exemplo dado um predicado $p(a, b)$ usado em um `enb`, se este é avaliado e falha, não se deveria reavalia-lo até que os valores de a ou b fossem alterados. Pode-se obter esta otimização através de um vetor de bits que indique se as variáveis foram modificadas desde a última execução do `enb` (no caso de não haver execução anterior assume-se verdadeiro). Então o pseudo-código do corpo de execução de uma ação seria:

if $(\exists\, x \in v \mid alterado(p, x))$ *then*
  if p *then* $\langle tenta\ executar\ metodo \rangle$
    else $(\forall\, x \in v \mid alterado(p, x) := False)$
else
  nothing

Onde v são slots usadas por p e $alterado(p, x)$ apresenta *True* se a variavel x foi modificada desde a última execução de p e *False* caso contrário. O predicado $alterado(p, x)$ poderia ser representado de forma bastante eficiente por uma matriz de bits.

Por exemplo seja um objeto que tenha slots a, b e c, então cada linha da matriz *alterado* teria 3 elementos $[status\_a, status\_b, status\_c]$, se o predicado p utiliza a e b, então o vetor v seria $[1, 1, 0]$. Usando esta representação, a condição $(\exists\, x \in v \mid alterado(p, x))$ se torna `(alterado[p] & v)` em Java. E $(\forall\, x \in v \mid alterado(p, x) := False)$ representado por $(alterado[p] := [0, 0, 0])$, torna-se, com um inteiro representando uma matriz de bits, `alterado[p] = 0;`.

E para toda atribuição `x << v` esta atualizaria a situação de $alterado(p, x)$ de tal forma que, para todo p que pertence ao conjunto de predicados que usam x, $alterado(p, x) := True$. Isto poderia ser implementado dentro de um método `set`, tal como

```
void x.Set(v) {
  ⟨codigo de atribuicao x = v⟩
  for p ∈ ⟨predicados Enb que usam x⟩
    alterado[p][indice_x] := 1;
}
```

Em código Java, a instrução $alterado[p][indice\_x] := 1$ torna-se `alterado[p] = alterado[p] | valor_indice_x`.

Outra alternativa é substituir ou pelo menos complementar `alt` por instruções como `if` ou `case`. Por afinidade, estas poderiam ser não determinísticas demoníacas (como a Guarded Command Language de Dijkstra), que sem a necessidade de backtracking pode ser implementada de forma bem mais eficiente. As variantes determinísticas seriam um pouco mais eficientes mas devido ao fato de tornar uma série de condições implícitas estas tendem a estimular um raciocínio mais voltado a

execução e desestimular o raciocínio baseado em lógica.

Outra razão ainda mais importante para a presença de construções como `if` é a expressividade adicional obtida como mostrado em 5.4.

7.5 Sobre o sistema de tipos de Java

Java usa um sistema de tipos dividido em por valor (chamados de primitivos, como integer) e por referência (arrays e objetos). Enquanto a semântica mais natural para Nautilus é por valor (evitando problemas com alias).

A semântica por referência representa um problema potencial para implementação de transações. A semântica original de Nautilus utiliza tipos bem simples que em geral podem ser tratadas como tipos primitivos de Java e não apresentam problemas (a exceção são arrays, permitindo apenas atribuição de elemento por elemento não há problemas, mas se permite a atribuição de um array para outro array, será necessário utilizar o procedimento `System.arrayCopy`).

Mas se o sistema de tipos de Nautilus for estendido para manipular estruturas mais complexas como records, o tradutor Nautilus-Java teria de utilizar `r1 = r2.Clone();` para cada atribuição `r1 << r2` e o tipo de `r1` e `r2` precisaria implementar a interface `Cloneable`.

7.6 Autoexecução de ações

Um ponto complicado para implementar da semântica de Nautilus é a idéia de que toda a ação é autoexecutável por default. O caso onde uma ação apenas deixa de ser autoexecutável por declaração (categoria request ou usa parâmetros) é simples, mas o caso onde uma ação deixa de ser autoexecutável por ser usada outra ação é trabalhoso. Para um implementador de um compilador deve ser o ponto menos composicional da linguagem.

É necessário consultar uma árvore de dependências feita a partir do código fonte para saber se uma ação vai ser efetivamente se autoexecuta ou não. Isto porque o código objeto de um componente pode ser levemente diferente dependendo de como ele é usado. Se um objeto é usado por uma construção, ela pode mudar o status de uma ação de executável para não executável.

Portanto se torna imprescindível ter uma estrutura de dados representando as dependências entre objetos e ações antes de efetivamente começar a tradução e essa característica tornaria a compilação separada bem trabalhosa.

Do ponto de vista pragmático de um programador, um ponto negativo da idéia de todas as ações serem autoexecutáveis é o seguinte: se um programador esta usando uma biblioteca nova e percebe algum comportamento inesperado que ele não deseja, descobrir a ação causadora disso pode ser complicado.

7.7 Relaçao expressividade/eficiência

Enquanto o mapeamento mostra que usar Nautilus como linguagem de programação é viável como ferramenta de ensino, existem aspectos técnicos que, no momento, impedem que ela seja uma linguagem de programação realmente prática.

O nível de expressividade/eficiência de Nautilus têm se mostrado ruim. Aqui considero expressividade como tendência a escrita de programas sucintos e elegan-

tes, e como eficiência, a tendência de gerar programas que usem menos recursos computacionais (CPU e memória). Por exemplo, Prolog é uma linguagem muito expressiva, mas seu sistema de execução via backtracking é muito ineficiente. Haskell é praticamente tão expressiva quanto Prolog e evita backtracking por default sendo portanto menos ineficiente. ML é um pouco menos expressiva que Haskell, mas seu sistema de execução é consideravelmente mais simples e eficiente. Pascal e C são linguagem de baixa expressividade em comparação com as anteriores, mas seus executáveis praticamente não precisam de suporte run-time auxiliar sendo portanto os mais eficientes dentre as citadas neste parágrafo.

Dentro deste contexto, comparemos Nautilus com linguagens concorrentes como Erlang, Occam2 ou CSP. Em comparação com Erlang e Occam2, Nautilus têm as primitivas mais caras, exigências de run-time são as maiores devido a característica transacional, e seus programas não parecem melhores do as das linguagens citadas. Um diferencial possível de Nautilus como linguagem de programação seria a reificação que é uma operação de restrição que pode ser útil se houvesse bibliotecas grandes as quais seriam restritas. Embora CSP tenha uma operação de restrição esta não foi incorporada nem a Erlang e Occam2, apesar das influências que elas receberam de CSP (principalmente Occam2), acredito que isso seja reflexo da filosofia de Hoare de que uma especificação deveria ser rica, mas não uma linguagem de programação (ele explicou sua posição da seguinte forma "é importante poder dizer 'não exploda' em uma especificação, mas em um programa você não pode usar um comando 'exploda' e depois nega-lo para tornar o programa mais seguro") .

Um exemplo da falta de expressividade é a não existência de recursão. Embora tecnicamente tudo que expresso de forma recursiva possa ser obtido por meio de iteração e pilhas. Um teste prático interessante para qualquer linguagem é se é simples um tradutor/interpretador/compilador para linguagem escrito na própria. A gramática de Nautilus é do tipo LL (Left to right, Leftmost derivation), e para este tipo de gramática a técnica de análise sintática mais conveniente é um analisador sintático descendente recursivo, que obviamente precisa de recursão para ser expresso de forma razoavelmente simples.

Enquanto algumas possíveis soluções tenham sido esboçadas aqui, ainda existem diversas questões a serem trabalhadas. E diversos trabalhos futuros para quem se habilitar.

8 CONCLUSÕES

O principal objetivo de obter um mapeamento das principais construções de Nautilus para Java (FUZITAKI et al., 2003) foi feito, mostrando que o mapeamento das construções de alto nível de Nautilus para Java é viável sobre as condições detalhadas no texto de trabalho.

O mapeamento básico foi testado através de um programa Haskell que trabalha utilizando a sintaxe abstrata de programas Nautilus e cria um arquivo Java. Algumas extensões como classes e interações foram mapeadas e testadas de forma manual.

Outra conclusão é que formalismos como CSP são ferramentas bastante úteis para otimização de programas concorrentes. Não apenas foi um importante auxílio para o raciocínio sobre o mapeamento, como também foi serviu como inspiração para tratar o sistema atual de parâmetros nas ações de Nautilus que se revelou complexo.

Entretanto, apesar da viabilidade como linguagem de ensino, Nautilus ainda apresenta problemas de expressividade e eficiência que limitam seu uso prático. Algumas dessas dificuldades foram discutidas e geraram propostas como a utilização de if não determinista demoníaco que auxilia tanto em expressividade como eficiência, e formas de inserir recursão na linguagem, mas ainda há muita pesquisa que precisa ser feita neste aspecto.

As principais publicações, que foram incluídas em anexos, são:

- (FUZITAKI et al., 2003) Mapping Nautilus Language into Java: Towards a Specification and Programming Environment for Distributed Systems. C. N. Fuzitaki, P. B. Menezes, J. P. Machado and S. A. Costa. Proc. of the 9th International Conference on Computer Aided Systems Theory and Technology, EUROCAST'2003. Las Palmas de Gran Canaria, 2003. Springer-Verlag. LNCS2809. pp. 243-252. (o mapeamento básico apresentado no capítulo 2);

- (FUZITAKI; MENEZES; MACHADO, 2004) A Protocol Checker for Nautilus Language. Fuzitaki, C. N., Menezes, P. B. e Machado, J. P. Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'04. Las Vegas, USA, 2004. (cujo conteúdo é uma versão preliminar do capítulo 6); e

- (FUZITAKI et al., 2004) Nautilus, a Concurrent Diagrammatic Specification and Programming Language. Claudio Fuzitaki, Paulo Blauth Menezes, Júlio Machado e Fernando D'Andrea. Journal of Supercomputing. (submetido e esperando avaliação, apresenta a versão refinada Nautilus-Java do capítulo 4).

Outras publicações como coautor foram:

- (D'ANDREA et al., 2002) Nautilus, a Diagrammatic Specification and Programming Language. Fernando D'Andrea, P. Blauth Menezes, Claudio Fuzitaki, Júlio Machado e Simone Costa. Proc. of the 14th International Conference on Parallel and Distributed Computing and Systems, PDCS'2002. Cambridge, USA. Anaheim: ACTA Press. pp. 386-391.(apresenta a notação diagramática de Nautilus); e

- (SEGEINFREDO et al., 2003) An Outline to a Diagrammatic Nautilus Environment. Segeinfredo, E. F., Gatto, R., Fuzitaki, C. N., Menezes, P. B. e Nunes, D. J. Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'03. Las Vegas, USA. v. 4. pp. 1726-1731. (descreve uma possível forma de implementar a notação diagramática).

8.1 Trabalhos Futuros

Um bom trabalho futuro é dar continuidade a proposta de adicionar recursão a linguagem integrado com um bom sistema de tipos. Sugere-se utilizar uma abordagem baseada em combinadores como a apresentada no capítulo 5.

Outro trabalho interessante seria generalizar o sistema de parâmetros de Nautilus para um sistema de troca de mensagens. Tal poderia ser realizado adicionando-se uma instrução de receive explicíto, mas a presença de recursão/interação tornaria o sistema de protocolos considerávelmente mais complexo.

Outro aspecto que poderia estudado é como fazer entrada e saída de forma mais integrada com o resto da linguagem, em vez das atuais primitivas que seguem o estilo de Pascal.

Entretanto a prioridade seria estudar maneiras de tentar minimizar os custos computacionais da linguagem. Talvez por interpretação abstrata ou estudar a idéia de trasformar Nautilus como sublinguagem transacional dentro uma linguagem não-transacional. Dentro da última idéia talvez tentar utilizar um estilo similar das construções Nautilus em Haskell. A outra forma seria criar formas de quebrar as abstrações da linguagem de forma controlada.

REFERÊNCIAS

BARBOSA, J. L. V.; DU BOIS, A.; PAVAN, A.; GEYER, C. F. R. HoloJava: translating a distributed multiparadigm language into java. In: CONFERÊNCIA LATINOAMERICANA DE INFORMÁTICA, 27., 2001, Mérida, Venezuela. **Anais...** [S.l.: s.n.], 2001.

BLOCH, J. **Substitutes for Missing C Constructs**. Disponível em: < http://developer.java.sun.com/developer/Books/shiftintojava/page1.html>. Acesso em: maio 2003.

CARNEIRO, C. **Autômato Não Seqüencial Identificado como Suporte para Classes em Nautilus**. 1999. Dissertacão (Mestrado em Ciência da Computacão) — Instituto de Informática, UFRGS, Porto Alegre.

CARNEIRO, C.; VEIT, T.; D'ANDREA, F.; MENEZES, P. B. Nautilus: its concurrent and distributed characteristics as an academic language. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 1999, Las Vegas, EUA. **Proceedings...** Athens: C.S.R.E.A., 1999. p.1919–1925.

COCKETT, R. **Charitable Thoughts**. Disponível em: < http://www.cpsc.ucalgary.ca/projects/charity/home.html >. Acesso em: maio 2003.

COCKETT, R.; FUKUSHIMA, T. **About Charity**. [S.l.]: Department of Computer Science, The University of Calgary, 1992. (Yellow Series Report, n. 92/480/18).

CORRADINI, A. **An Algebraic Semantics for Transition Systems and Logic Programming**. [S.l.]: Università di Pisa, 1990. (TD-8/90).

D'ANDREA, F. **Interface Diagramática para Linguagem Náutilus**. 2002. Projeto de Diplomacão (Bacharelado em Ciência da Computacão) — Instituto de Informática, UFRGS, Porto Alegre.

D'ANDREA, F.; MENEZES, P. B.; FUZITAKI, C.; MACHADO, J.; COSTA, S. Nautilus, a Diagrammatic Specification and Programming Language. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, PDCS, 14., 2002, Cambridge, USA. **Proceedings...** Anaheim: ACTA Press, 2002. p.386–391.

FLEMMING, N. (Ed.). **ML with Concurrency** : design analysis, implementation, and application. New York: Springer-Verlag, 1997. 255p. (Monographs in computer science).

FUZITAKI, C.; MENEZES, P. B.; MACHADO, J.; D'ANDREA, F. Nautilus, a Concurrent Diagrammatic Specification and Programming Language. **Journal of Supercomputing**, [S.l.], 2004. Submetido.

FUZITAKI, C. N.; MENEZES, P. B.; MACHADO, J. P. A Protocol Checker for Nautilus Language. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 2004, Las Vegas, USA. **Proceedings...** [S.l.: s.n.], 2004.

FUZITAKI, C. N.; MENEZES, P. B.; MACHADO, J. P.; COSTA, S. A. Mapping Nautilus Language into Java: towards a specification and programming environment for distributed systems. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED SYSTEMS THEORY AND TECHNOLOGY, EUROCAST, 9., 2003, Las Palmas de Gran Canaria. **Proceedings...** Berlin:Springer-Verlag, 2003. p.243–252. (Lecture Notes in Computer Science, n. 2809).

GIRARD, J.-Y.; LAFONT, Y.; TAYLOR, P. **Proofs and Types**. [S.l.]: Cambridge University Press, 1989. 183p.

HOARE, C. **Communicating Sequential Process**. [S.l.]: Prentice-Hall, 1985.

JONES, S. P. **Haskell 98 Language and Libraries**. Cambridge, UK: Cambridge University Press, 2003.

MENEZES, P. B. **Reificação de Objetos Concorrentes**. 1997. Tese (Doutorado em Ciência da Computacão) — Instituto Superior Técnico, Universidade Técnica de Lisboa, Lisboa.

MENEZES, P. B.; MACHADO, J. P.; COSTA, S. A. Explicit and Implicit Nondeterministic Refinement for Concurrent, Interacting Systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 2002, Las Vegas, USA. **Proceedings...** [S.l.: s.n.], 2002.

MENEZES, P. B.; SERNADAS, A.; COSTA, J. F. **Nonsequential Automata Semantics for a Concurrent Object-Based Language**. [S.l.]: Instituto Superior Técnico, Departamento de Matemática, 1995.

MENEZES, P. B.; SERNADAS, A.; COSTA, J. F. Nonsequential Automata Semantics for a Concurrent, Object-Based Language. In: FIRST US – BRAZIL JOINT WORKSHOP ON THE FORMAL FOUNDATIONS OF SOFTWARE SYSTEMS, 2000, New Orleans, EUA. **Proceedings...** Amsterdan: Elsevier, 2000. (Electronic Notes in Theoretical Computer Science, v.14).

MILNER, R. **Communication and Concurrency**. [S.l.]: Prentice-Hall, 1989.

PLASMEIJER, R.; EEKELEN, M. van. **Clean Version 2.0 Language Report**. [S.l.: s.n.], 2001.

RAMOS, J.; SERNADAS, A. **A Brief Introduction to GNOME**. Lisboa: Universidade Técnica de Lisboa, Instituto Superior Técnico, 1995. Disponível em: <http://www.cs.math.ist.utl.pt/cs/lcg/gnome.html>. Acesso em: maio 2003.

ROSSUM, G. van. **Python Tutorial**. [S.l.: s.n.], 2002.

SCHMIDT, D. A. **Denotational Semantics**: a methodology for language development. [S.l.]: Allyn and Bacon, 1986.

SEGEINFREDO, E. F.; GATTO, R.; FUZITAKI, C. N.; MENEZES, P. B.; NUNES, D. J. An Outline to a Diagrammatic Nautilus Environment. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 2003, Las Vegas, USA. **Proceedings...** [S.l.: s.n.], 2003. v.4, p.1726–1731.

SERNADAS, A.; RAMOS, J. **A Linguagem GNOME**: sintaxe, semântica e cálculo. Lisboa: Universidade Técnica de Lisboas, Instituto Superior Técnico, 1994.

SERNADAS, C.; GOUVEIA, P.; GOUVEIA, J.; RESENDE, P. The Reification Dimension in Object-Oriented Database Design. In: SPECIFICATION OF DATA BASE SYSTEMS, 1992. **Proceedings...** Berlin:Springer-Verlag, 1992. p.275–299.

SERNADAS, C.; GOUVEIA, P.; SERNADAS, A. **OBLOG**: object-oriented, logic-based conceptual modeling. Lisboa: Universidade Técnica de Lisboa, Instituto Superior Técnico, 1992.

SERNADAS, C.; RESENDE, P.; GOUVEIA, P.; SERNADAS, A. In-the-Large Object-Oriented Design of Information Systems. In: THE OBJECT-ORIENTED APPROACH IN INFORMATION SYSTEMS, 1991. **Proceedings...** [S.l.: s.n.], 1991. p.209–232.

THOMPSON, S. **Type Theory & Functional Programming**. [S.l.]: Computing Laboratory, University of Kent, 1999.

WINSKEL, G. Categories of Models for Concurrency. In: SEMINAR ON SEMANTICS OF CONCURRENCY, 1984, Pittsburgh. **Proceedings...** Berlin:Springer-Verlag:1985, 1984. p.246–267. (Lecture Notes in Computer Science, n. 197).

Os artigos em anexo foram removidos devido a possíveis problemas de copyright. Procure links para os artigos na página pessoal do autor:

`www.inf.ufrgs.br/~fuzitaki`

ANEXO A ARTIGO EUROCAST2003

Dados do artigo:

- Mapping Nautilus Language into Java: Towards a Specification and Programming Environment for Distributed Systems.

  C. N. Fuzitaki, P. B. Menezes, J. P. Machado and S. A. Costa.

  Proc. of the 9th International Conference on Computer Aided Systems Theory and Technology, EUROCAST'2003.

  Las Palmas de Gran Canaria, 2003.

  Springer-Verlag. LNCS2809. pp. 243-252

Conteúdo:

- O mapeamento básico apresentado no capítulo 2.

ANEXO B ARTIGO PDPTA2004

Dados do artigo:

- A Protocol Checker for Nautilus Language.

  Fuzitaki, C. N., Menezes, P. B. e Machado, J. P.

  International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'04.

  Las Vegas, USA, 2004

Conteúdo:

- Apresenta uma versão preliminar do capítulo 6.

ANEXO C ARTIGO JOURNAL OF SUPERCOMPUTING

Dados do artigo:

- Nautilus, a Concurrent Diagrammatic Specification and Programming Language.

  Claudio Fuzitaki, Paulo Blauth Menezes, Júlio Machado e Fernando D'Andrea.

  Journal of Supercomputing

  <Submetido>

  Conteúdo:

- Reapresenta a notação diagramática de (D'ANDREA et al., 2002).
- Apresenta o mapeamento Nautilus-Java do capítulo 4.

ANEXO D INTRODUÇÃO A LINGUAGEM NAUTILUS

O material deste anexo é baseado em um tutorial originalmente desenvolvido para pessoas com pouca experiência de programação sendo escrito em um tom mais didático. Apresenta a linguagem baseada em objetos Nautilus como definida em (MENEZES, 1997). Conceitos como objetos, ações e interação são introduzidos através de exemplos apresentados tanto de forma textual como através de uma notação gráfica. E as principais capacidades não usuais da linguagem, tais como visões, agregações e reificações, são mostradas.

Uma introdução a linguagem Nautilus

Claudio Naoto Fuzitaki

***Abstract*— O objetivo deste trabalho é fornecer uma introdução a linguagem baseada em objetos Nautilus. O tom é basicamente didático. Conceitos como objetos, ações e interação são introduzidos através de exemplos apresentados tanto de forma textual como através de uma notação gráfica. E as principais capacidades não usuais da linguagem, tais como visões, agregações e reificações, são mostradas.**

I. Introdução/Histórico

Nautilus é uma linguagem baseada em objetos [7], textual, que possui facilidades de abstração e suporta objetos distribuídos [3]. Ela foi baseada em GNOME [5], [4], consistindo de uma parte representativa desta; GNOME consiste de uma simplificação revista da linguagem orientada a objetos OBLOG [6].

A linguagem Nautilus pode ser usada para especificação e/ou programação de sistemas. A linguagem é de alto nível, fundamentada em uma abordagem de Teoria das Categorias, sendo dotada de poderosas facilidades de abstração não usuais em linguagens de programação atuais. Possui concorrência em múltiplos níveis. de fato, ações de objetos diferentes concorrem entre si, assim como ações dentro de um mesmo objeto. Mesmo as cláusulas presentes dentro de cada ação (que devem ser executados de forma atômica) podem ser concorrentes.

Duas extensões em desenvolvimento para a linguagem são classes e uma notação gráfica. Classes [1], semelhantes as utilizadas em linguagens orientadas a objeto convencionais, deverão ser implementadas de forma a manter as capacidades de composição não usuais da linguagem.

A notação gráfica [2]é especialmente útil no desenvolvimento de sistemas de grande porte, devido ao grande poder de expressão dos símbolos gráficos. A notação gráfica pode ser esclarecedora para exibir conceitos, já que não existe uma sintaxe rígida como a representação textual, portanto ela também é introduzida neste tutorial ilustrando muitos exemplos.

No momento em que este trabalho está sendo escrito ainda não existe nenhum compilador/interpretador para a linguagem. Portanto ela se presta, por enquanto, apenas a especificação. Além disso a linguagem não tem definida uma interface com o usuário (rotinas de E/S).

II. Conceito de Objeto

Um objeto pode ser visto como uma entidade que age e encapsula estados/ações. Adicionamente ele pode interagir com outros objetos através das suas ações. Os estados encapsulados de um objeto A são inacessíveis para um outro objeto B a não ser que A exporta ações que permitam acessar suas propriedades.

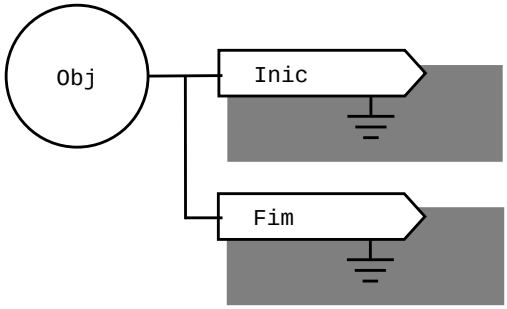

Fig. 1. Objeto Obj em notação gráfica

Por exemplo, um relógio pode ser visto como um objeto. O mostrador pode ser visto como uma ação de saída, o estado seria o horário e as ações de entrada seriam os botões para ajuste de horário, alarme, etc. Outra característica é que o objeto relógio têm ações que funcionam sozinhas sem precisar de interação externa (os mecanismos que mudam o horário).

Um ponto muito importante: um objeto precisa ter um ponto de partida. Significa que ele precisa "nascer"em algum momento antes de começar a agir. A ação de nascimento é chamada `birth`. Antes da execução desta ação para todos os efeitos o objeto não existe.

O objeto também pode ter opcionalmente um ponto de termino, quando ele "morre". A ação de morte é chamada `death`.

Como exemplo vamos definir um objeto cujas únicas ações sejam de nascimento e morte.

```
object Obj
category
  birth Inic
  death Fim
body
  act Inic
  act Fim
end
```

A forma deste programa na notação gráfica é mostrada na figura 1.

O símbolo de aterramento é usado para indicar que não existe mais comandos para serem executadas dentro da ação (em um objeto normalmente existem diversos comandos dentro de uma ação). Neste caso, onde a ação não têm nenhum comando, o símbolo de aterramento poderia ter sido omitido.

O comportamento do programa é que o objeto executaria a ação `Inic` para ser iniciado e depois, como não outras ações, executaria a ação `Fim` que o encerraria.

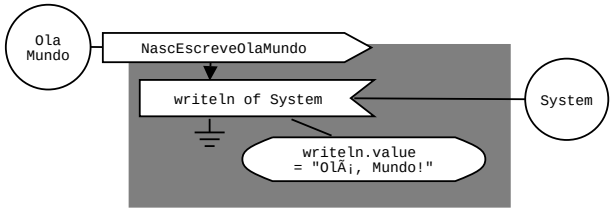

Fig. 2. Objeto OlaMundo (e sua relação com o objeto System)

III. Interação de objetos

Como dito anteriormente Nautilus não têm definido entradas e saída padrão, entretanto para facilitar o aprendizado, vamos assumir a existência de um objeto pré-definido `System` com duas ações: `writeln` e `readln`. Eles são do tipo *request*, isto é, não funcionar sozinhos, precisam ser chamados por outro objeto. Eles também possuem um parâmetro chamado `Value` que pode ser usado como saída e entrada respectivamente. Vamos assumir também que a interação com o usuário se dará por meio de uma interface do tipo terminal de texto.

Comecemos com um dos programas mais simples que existem, o "Olá mundo!".

```
spec ImprimeMsg

object OlaMundo
import
  writeln of System
category
  birth NascEscreveOlaMundo
body
  act NascEscreveOlaMundo
    call writeln of System
    arg writeln.Value = "Olá, Mundo!"
end OlaMundo

end spec.
```

Este exemplo ilustra a interação entre dois objetos [1], o objeto `OlaMundo` e o objeto `System` (fig. 2).

Basicamente este programa eventualmente será ativado através da ação `NascEscreveOlaMundo` exibindo a mensagem "Olá, Mundo!"e não fará mais nada já que sua única ação é do tipo `birth` e um objeto só pode nascer uma vez. Entretanto apesar de ficar parado o objeto não deixou de existir, para isso seria necessário a execução de uma ação do tipo `death`.

Vejamos um outro programa semelhante que separa a ação de nascimento da ação de escrita da mensagem "Olá,Mundo".

```
object OlaMundo2
import
  writeln of System
category
  birth Nasc
body
  act Nasc
    call writeln of System
    arg writeln.Value = "Nasci!"
  act EscreveOlaMundo
    call writeln of System
    arg writeln.Value = "Olá, Mundo!"
end OlaMundo2
```

Qual é o comportamento deste programa?

Em algum momento que o programa será ativado através da ação `Nasc` e a partir daí, ele poderá executar qualquer outra ação que não esteja restrita por questões de sincronização (neste caso, apenas `EscreveOlaMundo`). Portanto a saída deste programa será

```
Nasci!
Olá, Mundo!
Olá, Mundo!
Olá, Mundo!
Olá, Mundo!
Olá, Mundo!
Olá, Mundo!
Olá, Mundo!
Olá, Mundo!
⋮
```

A ação é executada infinitas vezes (em um máquina real até o usuário cancelar o processo).

IV. Não-determinismo

Agora um desafio mais interessante: Tente adivinhar qual seria a saída deste programa?

```
object XY
import
  writeln of System
category
  birth Nasc
body
  act Nasc
    call writeln of System
    arg writeln.Value = "Nasci!"
  act EscreveX
    call writeln of System
    arg writeln.Value = "X"
  act EscreveY
    call writeln of System
    arg writeln.Value = "Y"
end XY
```

A resposta é você não sabe exatamente o que vai ser impresso. A única coisa certa é `"Nasci!"`, depois é como se o programa jogasse uma moeda para decidir se vai executar `EscreveX` ou `EscreveY`. Isso é o *não-determinismo*.

Se você quisesse a saída do programa fosse

```
Nasci!
X
Y
X
Y
X
Y
⋮
```

como fazer isso?

Uma maneira é lembrar quem foi executado por último e restringir a execução deste último. Para lembrar quem foi

[1] Uma observação importante é que apesar da *interação* ser a maneira mais comum de utilizar diversos objetos em linguagens orientadas a objetos, este não é o padrão mais natural de programar com vários objetos em Nautilus (este mesmo exemplo poderia ser construído utilizando-se *agregação*).

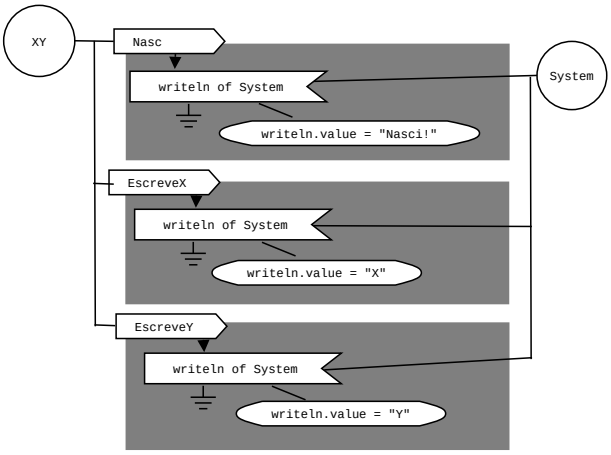

Fig. 3. Objeto XY não-determinista

executado vamos introduzir o conceito de *variável* e *pré-condição*.

```
object XY
import
  writeln of System
category
  birth Nasc
body
  slot Ultimo:boolean
  act Nasc
    seq
      val Ultimo << true
      call writeln of System
      arg writeln.Value = "Nasci!"
    end seq
  act EscreveX
    enb Ultimo = true
    seq
      val Ultimo = false
      call writeln of System
      arg writeln.Value = "X"
    end seq
  act EscreveY
    enb Ultimo = false
    seq
      val Ultimo = true
      call writeln of System
      arg writeln.Value = "Y"
    end seq
end XY
```

Uma *variável* pode ser vista como um espaço na memória do computador que contém algum valor. Em Nautilus variáveis são introduzidas usando-se a palavra reservado `slot`. No exemplo acima a variável `Ultimo` é introduzida tendo tipo `boolean`. Um *tipo* indica os valores que podem ser armazenados nas variáveis (neste caso apenas os valores lógicos `True` ou `False` são permitidos).

Uma *pré-condição* estabelece condições para execução de uma ação. Isto é, a pré-condição restringe a execução da ação para somente os casos onde a expressão lógica, que segue a palavra reservada `enb`, retorna valor verdadeiro.

A palavra chave `seq` é usada para indicar a execução sequêncial de comandos agrupados. Outra forma de juntar diversos comandos seria `cps` (compose) que é uma

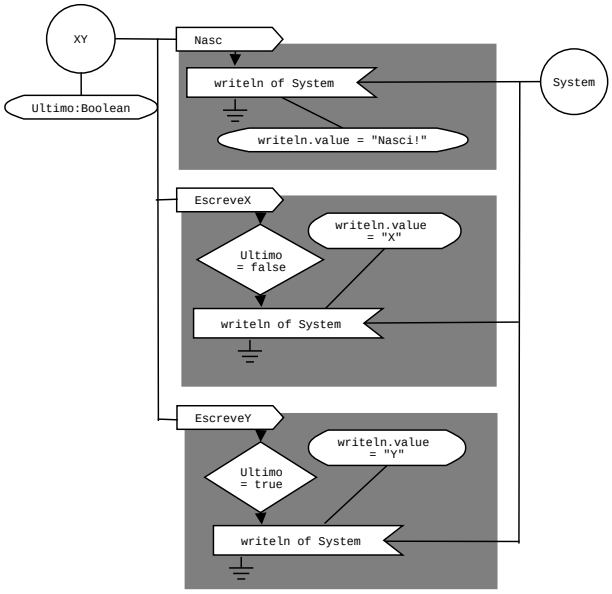

Fig. 4. Objeto XY determinista

generalização de atribuição múltipla onde as expressões do lado direto são avaliadas simultaneamente e só depois é feita a valoração.

V. Estados, Parâmetros e Condições

Neste exemplo vamos observar um programa mais usual: um cálculo de fatorial.

```
object Fatorial
export
  Inicia in N:natural
  Termina out Fat:natural
category
  birth request Inicia
  death Termina
body
  slot Fat:natural
  slot N:natural
  act Inicia
    cps
      val Fat << 1
      val N << Inicia.N
    end cps
  act Passo
    enb N > 1
    seq
      val N << N - 1
      val Fat << Fat * N
    end seq
  act Termina
    enb N <= 1
    ret Termina.Fat = Fat
end Fatorial
```

Este objeto tem uma série de características importantes que serão detalhadas a seguir.

A primeira é que têm uma ação `birth request`, isto significa que ele não começa sozinho como os anteriores, precisa ser chamado por outro objeto, e além disso recebe como entrada um `N` do tipo `natural`.

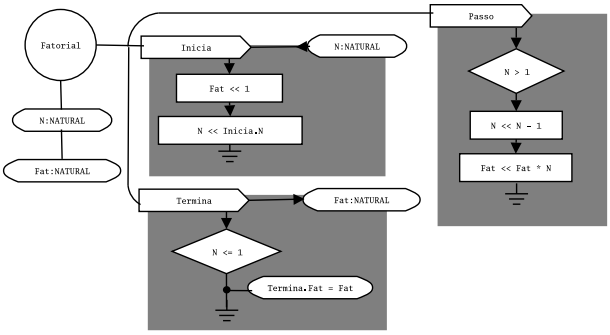

Fig. 5. Objeto Fatorial

A segunda característica neste objeto é que ele tem uma ação de finalização, indicada pela palavra-chave `death`. A partir do momento em que ela for executada o objeto deixa de existir. Adicionalmente a ação tem uma saída `Fat` do tipo `natural`.

E o ponto mais importante, o objeto tem dois estados (ou variáveis) internos chamados `Fat` e `N`. Essas variáveis são usadas para armazenar valores intermediários que são refinados até que se chegue ao estado final desejado (neste caso o valor do fatorial).

Uma definição de fatorial é

$$N! = \begin{cases} N * (N-1)! & \text{se } N > 1 \\ 1 & \text{se } N = 0 \text{ ou } N = 1 \end{cases}$$

Vejamos se o programa satisfaz esta definição.

Se iniciarmos o objeto com N = 0

```
Fat << 1
N << 0
```

Como N não atende a pré-condição de `Passo` ($N > 1$), a única ação que o objeto pode realizar é `Terminar` (cuja pré-condição $N \leq 1$ é satisfeita). Portanto

```
Termina.Fat << 1
```

e o mesmo acontece quando N é iniciado com 1.

Se iniciarmos o objeto com outro qualquer N (por exemplo 3)

```
Fat << 1
N << 3
```

como N = 3 então `Passo` é executado

```
N << 3 - 1
Fat << 1 * 3
```

agora N = 2 e Fat = 3, então `Passo` é executado novamente

```
N << 2 - 1
Fat << 3 * 2
```

finalmente, a única ação que o objeto pode realizar é `Terminar`. Portanto

```
Termina.Fat << 6
```

O funcionamento do algoritmo é o seguinte: na ação `Inicia`, `Fat` recebe o valor inicial de 1 e `N` recebe como valor inicial o parametro de entrada `Inicia.N` (que é o valor para o qual queremos calcular o fatorial).As variáveis têm seus valores alterados pela acão `Passo` enquanto N for maior que 1 (indicado pela condição `enb N > 1`. E quando a variável N atinge o valor 1, o objeto pode terminar retornando o valor do fatorial.

VI. Transação

Uma característica incomum de Nautilus é que suas ações são transacionais como em sistemas de banco de dados, isto é, uma ação deve efetuar todas as alterações preescritas pelas ações componentes ou nenhuma.

Por exemplo, seja uma ação composta:

```
act Falha
  seq
    val A << 0
    val B << 1/A
  end seq
```

onde os valores iniciais de A e B são 3 e 4.

A primeira valoração ocorre sem problemas, mas a segunda causa um erro. Na maioria das linguagem de programação a primeira instrução seria executada e a segunda causaria uma exceção, então o valor final de A e B seria 0 e um valor qualquer (talvez 4 ou nulo). No caso de Nautilus como a segunda valoração não pode ser executada, então a ação composta não pode ser executada e os valores das variáveis alteradas são restauradas para seus valores iniciais (3 e 4).

VII. Paralelismo e Concorrência

Nesta seção vamos introduzir informalmente os conceitos de paralelismo e concorrência. Para começar um exemplo de paralelismo sem concorrência:

```
object Tic
import
  writeln of System1
export
  Nasc EscreveTic
category
  birth Nasc
body
  act Nasc
    nothing
  act EscreveTic
    call writeln of System1
    arg writeln.Value = "Tic"
end Tic

object Tac
import
  writeln of System2
export
  Nasc EscreveTac
category
  birth Nasc
body
  act Nasc
    nothing
  act EscreveTac
    call writeln of System2
    arg writeln.Value = "Tac"
end Tac
```

Você deve estar se perguntando: O que são `System1` e `System2`? Eles são uma extensão do `System` que definimos antes. Pode-se pensar que cada um deles é um terminal

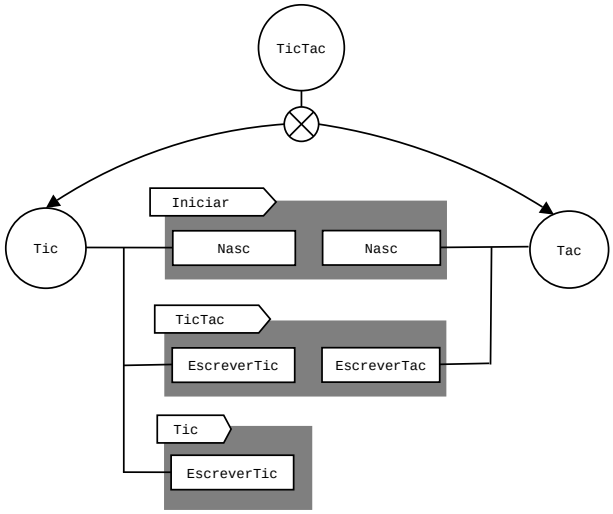

Fig. 6. Agregação de Tic e Tac

distinto. Por que isso? A razão é a seguinte se `Tic` e `Tac` interagissem com o mesmo objeto `System` isso iria impor a restrição de que `Tic` e `Tac` não podem ocorrer simultaneamente pois se os dois tentassem chamar `System.writeln` ao mesmo tempo uma das ações não iria acontecer (isto é, seria um exemplo de paralelismo com concorrência). Esta maneira de Nautilus tratar concorrência, em que se duas ou mais ações tentarem acessar o mesmo recurso, somente uma acontece, é baseada na propriedade da *atomicidade* das ações.

VIII. Agregação

No exemplo `Tic` e `Tac`, como restringir o `Tac` para que ele só ocorresse junto com `Tic` (mas que `Tic` ainda fosse independente de `Tac`)?

Pode-se utilizar uma agregação. Uma primeira tentativa seria

```
object TicTac
aggregation of
  Tic
  Tac
category
  birth Iniciar
body
  act Iniciar composed by
     Nasc of Tic
     Nasc of Tac
  act TicTac composed by
     EscreverTic of Tic
     EscreverTac of Tac
  act Tic composed by
     EscreverTic of Tic
end TicTac
```

Neste caso acrescenta-se o objeto `TicTac` a especificação. `TicTac` é um objeto *agregador* que estabelece como `Tic` e `Tac` irão se comportar em conjunto. `EscreverTic` pode ocorrer de forma independente, mas `EscreverTac` só pode acontecer junto com `EscreverTic`.

Na notação gráfica a agregação é mostrada por círculo preenchido por um X (conforme fig 6).

Entretanto, isto não funcionaria por causa da semântica destrutiva das operações sobre objetos em Nautilus. Os objetos `Tic` e `System1` são consumidos pela operação interação (usada implicitamente por `import` e `call`) produzindo um objeto `interaction Tic System1 end interaction`. Então a forma correta de executar tal operação seria:

```
object TicTac
aggregation of
  interaction
    Tic  System1
  end interaction
  interaction
    Tac System2
  end interaction
category
  birth Iniciar
body
  act Iniciar composed by
     int Nasc of Tic end int
       of interaction
         Tic  System1
       end interaction
     int Nasc of Tac end int
       of interaction
         Tac System2
       end interaction

  act TicTac composed by
    int
     EscreverTic of Tic
     writeln of System1
    end int
       of interaction
         Tic  System1
       end interaction
    int
     EscreverTac of Tac
     writeln of System2
    end int
       of interaction
         Tac System2
       end interaction

  act Tic composed by
    int EscreverTic of Tic end int
        of interaction
          Tic System1
        end interaction

end TicTac
```

O que é terrivelmente prolixo. Para que código similar a primeira tentativa fosse válido pode-se usar agregação no lugar de interação na definição de `Tic` e `Tac`, por exemplo:

```
object TicA
aggregation of
  System1
  Tic
category
  birth Nasc
body
  act Nasc composed by
    Nasc of Tic
  act EscreveTic composed by
    EscreveTic of Tic
    Writeln of System1
    match
      EscreveTic.Saida
      Writeln.Value
```

```
end TicA

object Tic
category
  birth Nasc
  EscreveTic out Saida:String
body
  act Nasc
    nothing
  act EscreveTic
    ret Saida << "Tic"
end Tic
```

assim, pode-se usar `TicA` no lugar de `interaction Tic System1 end interaction`; ou então poderia se envolver `interaction Tic System1 end interaction` usando uma visão (uma construção explicada a seguir).

IX. Visão

Visão é uma operação que encapsula um objeto. Basicamente ela torna atributos que eram acessíveis a outros objetos inacessíveis. Outra função é adicionar restrições em operações. Um exemplo é

```
object V1 view of O1
export
  A B C
category
  birth request A
  request B C
end V1

object O1
export
  A B C D E
category
  birth A
body
  ..
end O1
```

Neste exemplo na visão `V1` somente as operações `A`,`B` e `C` permanecem visíveis e além disso adiciona-se a restrição `request` nelas.

A vantagem deste recurso é facilitar a organização. Por exemplo digamos que você queira criar uma agregação e pretenda usar dois ou três ações de um objeto que tenha 50 ações (e alguns deles com nomes podem coincidir com ações de outros objetos da agregação). A melhor maneira de organizar isso é criar uma visão do objeto com somente os métodos desejados visíveis e utilizá-la no lugar do objeto.

Um uso comum é envolver o resultado de uma interação para evitar referências a partes desta da seguinte forma:

```
object TicV view of
  interaction
    Tic
    System1
  end interaction
export
  Nasc EscreveTic
body
  act Nasc as
  int
    Nasc of Tic
  end int
  act EscreveTic as
  int
    EscreveTic of Tic
    writeln of System1
  end int
end TicV
```

X. Reificação

Um conceito não encontrado normalmente em linguagens convencionais é a reificação (ou implementação de um objeto sobre outro). Se pode pensar em reificação como um mapeamento de um objeto para as computações de outro.

Veremos um exemplo baseado na implementação de uma pilha. Pilhas são uma das estruturas de dados mais conhecidas e usadas em ciências da computação. As características básicas de uma pilha são a existência de uma operação de entrada Empilha e saída Desempilha, tal que que o o último a entrar (empilhado) é o primeiro a sair (desempilhado). Um exemplo do mundo real é a pilha de pratos que se forma quando uma pessoa lava louça, empilha-se um prato em cima do outro e na hora de enxugar pega os pratos que estão mais acima.

Neste exemplo a `Pilha` é definida como uma reificação (indicada pela palavra over no texto e pelo retângulo com traços internos na fig. 7) sobre o objeto `Apontador_Tabela` que é uma agregação entre um objeto `Apontador` (usado para indicar qual o último elemento a ser inserido) e o objeto `Tabela` (onde são armazenados os dados).

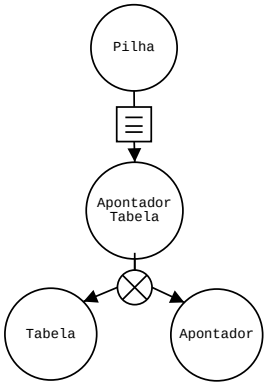

Fig. 7. Visão geral da reificação de Pilha sobre a agregação de Apontador e Tabela

```
object Pilha over Apontador_Tabela
export
  Nasc
  Empilha der in Valor: integer
    by Atribui.Valor
  Desempilha der out Valor: integer
    by Consulta.Valor
category
  birth request Nasc
  request Empilha Desempilha
body
  act Nasc
    Novo
  act Empilha
    seq
      Mais_um
      Atribui
    end seq
  act Desempilha
```

```
    seq
      Consulta
      Menos_um
    end seq
end Pilha

object Apontador_Tabela
aggregation of
  Apontador
  Tabela
export
  Novo
  Atribui der in Valor:integer
    by Atribui.Valor of Tabela
  Consulta der out Valor:integer
    by Consulta.Valor of Tabela
  Mais_um
  Menos_um
category
  birth Novo
body
  act Novo composed by
    Novo of Apontador
    Novo of Tabela
  act Atribui composed by
    Posição of Apontador
    Atribui of Tabela
      match
        Posição.Pos of Apontador
        Atribui.Ind of Tabela
  act Consulta composed by
    Posição of Apontador
    Consulta of Tabela
      match
        Posição.Pos of Apontador
        Consulta.Ind of Tabela
  act Mais_um composed by
    Mais_um of Apontador
  act Menos_um composed by
    Menos_um of Apontador
end Apontador_Tabela

object Apontador
export
  Novo
  Mais_um
  Menos_um
  Posição out Pos: natural
category
  birth request Novo
  request Posição
body
  slot Pos: natural
  act Novo
    val Pos << 0
  act Mais_um
    val Pos << Pos + 1
  act Menos_um
    val Pos << Pos - 1
  act Posição
    ret Posição.Pos = Pos
end Apontador

object Tabela
export
  Novo
  Atribui
    in Valor: integer
    in Ind: natural
  Consulta
    out Valor: integer
    in Ind: natural
  category
    birth request Novo
```

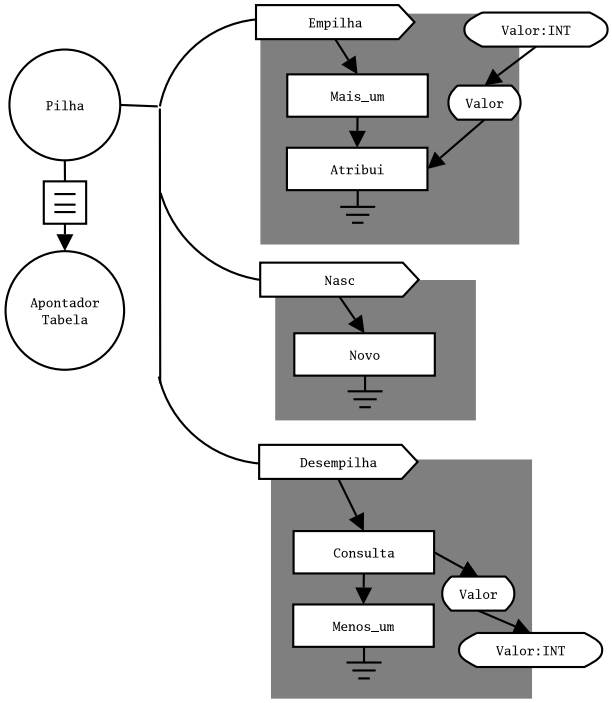

Fig. 8. Reificação de Pilha sobre Apontador_Tabela

```
    request Atribui Consulta
  body
    slot Tab: integer [1..100]
    act Novo
    act Atribui
      enb Atribui.Ind <= 100
      enb Atribui.Ind >= 1
      val Tab[Atribui.Ind] << Atribui.Valor
    act Consulta
      enb Atribui.Ind <= 100
      enb Atribui.Ind >= 1
      ret Consulta.Valor =
        Tab[Consulta.Ind]
end Tabela
```

A palavra reservada `seq` é utilizada para obrigar dois comandos seguirem uma seqüência. No exemplo a ação `Empilha` de `Pilha` é definida como uma transação atômica da seqüência de `Mais_um` e `Atribui`.

Um ponto que pode chamar a atenção são as ações `Mais_um` e `Menos_um` aparentemente livres no objeto `Apontador` (sem a restrição `request`). Isto significa que elas podem ocorrer espontaneamente? A resposta é não, porque elas são restritas no objeto `Pilha` pois elas ocorrem apenas quando o método `Empilha` é chamado.

Outra novidade são as construções `der in` e `der out` que indicam que as entradas e saídas são derivadas de outras ações.

XI. Especificação de uma Fábrica

Este é um exemplo mais elaborado que utiliza todos os novos conceitos introduzidos. A especificação de `Fábrica` é o resultado da agregação dos objetos `Produtor` e `Consumidor`. O objeto `Produtor` é implementado sobre

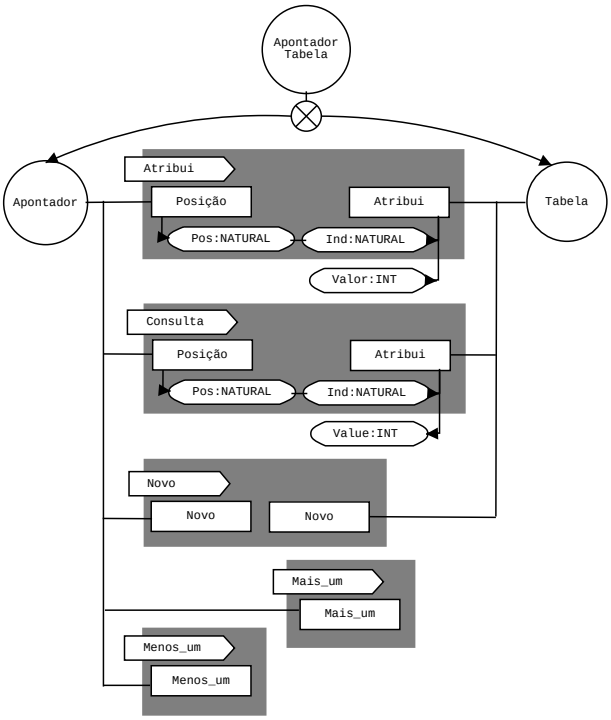

Fig. 9. Agregação Apontador_Tabela

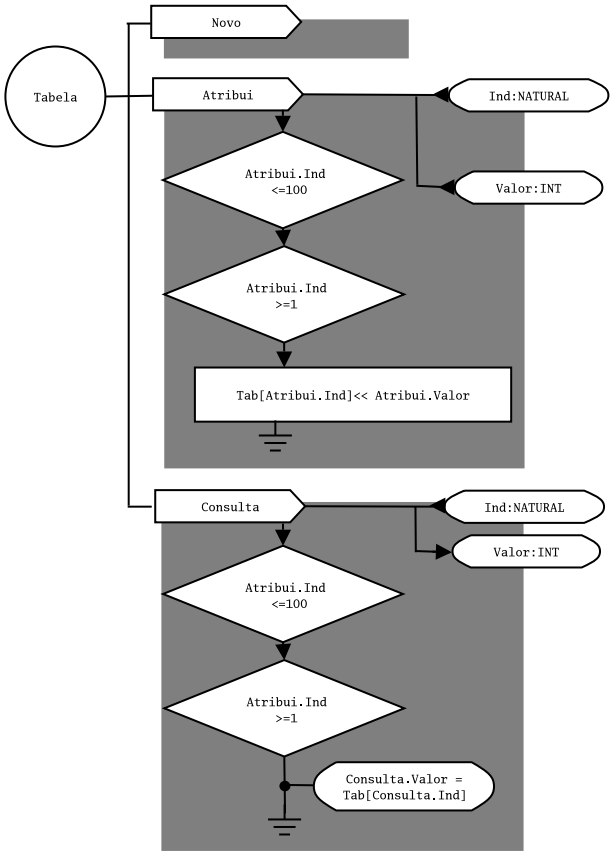

Fig. 11. Tabela

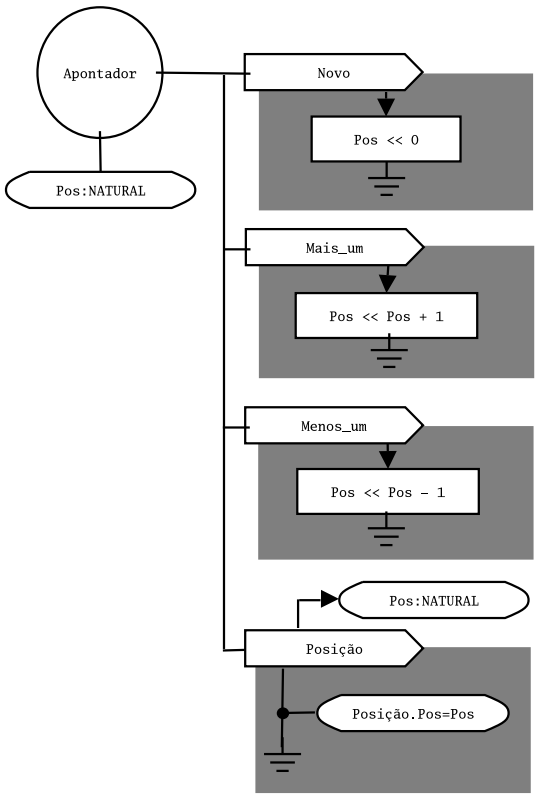

Fig. 10. Apontador

a `Linha_Produção` o qual, por sua vez, é uma visão da interação dos objetos `Produção` e `CQ` (Controle de Qualidade). O objeto `Consumidor` é implementado sobre `Linha_Consumo`.

Por motivos de espaço este exemplo não será apresentado utilizando-se a notação gráfica. Entretanto, o relacionamento entre os diversos objetos componentes é bem ilustrado pela figura 12.

```
spec Fábrica

object Fábrica_Raiz
aggregation of
  Produtor
  Consumidor
category
  birth Nasc
body
  act Nasc composed by
    Novo of Produtor
    Novo of Consumidor
  act Comunica composed by
    Envia of Produtor
    Recebe of Consumidor
      match
        Envia.Fator of Produtor
        Recebe.Fator of Consumidor
      match
        Envia.Msg of Produtor
        Recebe.Msg of Consumidor
end Fábrica_Raiz
```

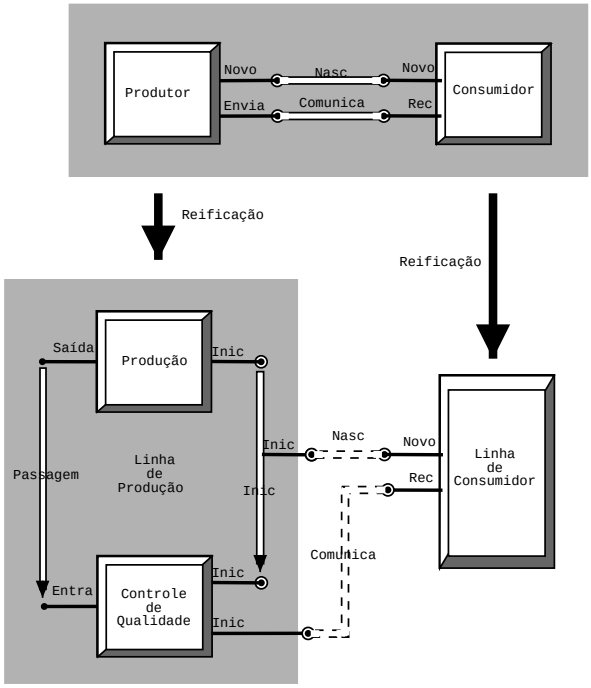

Fig. 12. Especificação de uma fábrica

```
object Produtor over Linha_Produção
export
  Novo
  Envia
    der in Fator: natural
    by Envia.Fator of Linha_Produção
    der out Msg: natural
    by Envia.Msg of Linha_Produção
category
  birth request Novo
  request Envia
body
  act Novo
    Inic
  act Prod_A
    seq
      Proc1
      Montagem
      Passagem
      ProcCQ
    end
  act Prod_A
    seq
      cps
        Proc1
        Proc2
      end cps
      Montagem
      Passagem
      ProcCQ
    end
  act Envia
    Envia
end Produtor

object Linha_Produção view of
  interaction
    Produção
    CQ
  end interaction
export
  Inic Passagem
  Proc1 Proc2
  Montagem ProcCQ
  Envia
    der in Fator:natural
      by Envia.Fator of CQ
    der out Msg: natural
      by Envia.Msg of CQ
category
  birth Inic
body
  act Inic as
    int
      Inic of Produção
      Inic of CQ
    end int
  act Passagem as
    int
      Saída of Produção
      Entrada os CQ
    end int
end Linha_Produção


objeto Produção
import
  Inic of CQ
  Entrada in CodProd: natural of CQ
export
  Inic Proc1 Proc2
  Montagem Saída
category
  birth Inic
body
  slot P1: boolean
  slot P2: boolean
  slot NumProd: natural
  act Inic
    call Entrada of CQ
    cps
      val P1 << false
      val P2 << false
      val NumProd << 0
    end cps
  act Proc1
    enb P1 = false
    val P1 << true
  act Proc2
    enb P2 = false
    val P2 << true
  act Montagem
    alt Mont_P1
      enb P1 = true and P2 = false
      val NumProd << NumProd + 1
      val P1 << false
    alt Mont_P2
      enb P2 = true and P1 = false
      val NumProd << NumProd + 1
      val P2 << false
    alt Mont_P1_P2
      enb P1 = true and P2 = false
      val NumProd << NumProd + 1
      val P1 << false
      val P2 << false
  act Saída
    enb NumProd > 0
    call Entrada of CQ
      arg Entrada.CodProd = ..
    val NumProd << NumProd - 1
end Produção

object CQ
export
```

```
  Inic ProcCQ
  Entrada
    in CodProd: natural
  Envia
    in Fator: natural
    out Msg: natural
category
  birth Inic
body
  slot NumCQ: natural
  slot NumOK: natural
  act Inic
    cps
      val NumCQ << 0
      val NumOK << 0
    end cps
  act Entrada
    val NumCQ << NumCQ + 1
  act ProcCQ
    enb NumCQ > 0
    val NumOK << NumOK + 1
    val NumCQ << NumCQ - 1
  act Envia
    enb NumOK > 0
    val NumOK << NumOK - 1
    ret Envia.Msg = Fator * ...
end CQ

object Consumidor over Linha_Consumo
export
  Novo
  Recebe
    der out Fator: natural
      by Recebe.Fator
    der in Msg: natural
      by Recebe.Msg
category
  birth request Novo
  request Recebe
body
  act Novo
    Inic
  act Recebe
    seq
      Prepara_Rec
      Recebe
    end seq
  act Consome
    Consome
end Consumidor

object Linha_Consumo
export
  Inic
  Prepara_Rec
  Recebe
    out Fator: natural
    in Msg: natural
  Consome
category
  birth Inic
body
  slot St: 1..3
  act Inic
    val St << 1
  act Prepara_Rec
    enb St = 1
    val St << 2
  act Recebe
    enb St = 2
    agg Recebe.Fator = ...
    val St << 3
  act Consome
    enb st = 3
    val St << 1
end Linha_Consumo

end spec.
```

REFERENCES

[1] C. CARNEIRO, “Autômato não seqüencial identificado como suporte para classes em náutilus,” PPGC-UFRGS: Dissertação de Mestrado, UFRGS, Porto Alegre, RS, 1999.

[2] F. P. D’Andrea and C. R. J. B. Carneiro, “Uma interface gráfica para a linguagem náutilus,” 2000.

[3] P. MENEZES, “Reificação de objetos concorrentes,” Ph.D. dissertation, Universidade Técnica de Lisboa, IST, Lisboa, PT, 1997.

[4] A. SERNADAS and J. RAMOS, “A linguagem gnome - sintaxe, semântica e cálculo,” Universidade Técnica de Lisboa, IST, Lisboa, PT, Tech. Rep., 1994.

[5] C. SERNADAS, P. CARMO, and P. PENEDO, “The gnome compiler,” Universidade Técnica de Lisboa, IST, Lisboa, PT, Tech. Rep., 1994.

[6] C. SERNADAS, P. GOUVEIA, and A. SERNADAS, “Oblog - object-oriented logic-based conceptual modeling,” Universidade Técnica de Lisboa, IST, Lisboa, PT, Tech. Rep., 1992.

[7] P. WEGNER, “Concepts and paradigms of object-oriented programming,” *OOPS MESSENGER*, vol. 1, no. 1, Aug. 1990.