5
Refinamento e Síntese de Especificações no Ambiente LindaStudio

Este capítulo apresenta o ambiente de criação de serviços oferecido junto à linguagem LindaX, chamado 'LindaStudio'. O foco deste capítulo é na apresentação das ferramentas extensoras, de refinamento e de síntese presentes no ambiente LindaStudio. Contudo, esse ambiente é composto por outras ferramentas que permitem, por exemplo, editar descrições de arquiteturas e de estilos em LindaX e verificar erros – em níveis seletivos de detalhe – nessas descrições. Essas outras ferramentas são apresentadas apenas superficialmente neste capítulo, podendo-se obter maiores informações sobre elas em (Laboratório TeleMídia, 2004).

O capítulo está estruturado como se segue. Nas Seções 5.1 e 5.2, é apresentada uma visão geral da arquitetura do ambiente LindaStudio, suas principais ferramentas e a forma como elas estão estruturadas. Na Seção 5.3 são introduzidas as ferramentas extensoras e de refinamento de descrições de arquitetura. Essa seção apresenta também um estudo de caso de aplicação dessa ferramenta no refinamento para uma ADL com suporte a descrições formais de arquitetura. A Seção 5.4 apresenta o *framework* para gerência de adaptações, sobre o qual os processos implementados pela ferramenta de síntese são construídos. A ferramenta de síntese é assunto da Seção 5.5. Essa seção apresenta também um estudo de caso de aplicação integrada do *framework* para gerência de adaptações e das ferramentas de refinamento e síntese em um *toolkit* de componentes de software especializado para redes programáveis.

5.1.
Visão geral

O ambiente LindaStudio oferece um conjunto de seis ferramentas principais de manipulação de DSLs e estilos LindaX:

1. `StyleEditor`: permite a edição de estilos LindaX a partir de notações alternativas à sintaxe-base de XML, bem como o armazenamento desses estilos em um repositório específico;
2. `ConfEditor`: permite a edição de descrições de arquitetura em qualquer DSL LindaX a partir de notações alternativas à sintaxe-base de XML, bem como o armazenamento dessas descrições em um repositório específico;
3. `Translator`: refina descrições de arquitetura em uma DSL LindaX (mantidas no repositório de descrições) para diferentes FDTs ou ADLs, ou ainda para operações de configuração em plataformas de programação específicas. Para que esses refinamentos sejam possíveis, são necessárias informações estruturais e semânticas detalhadas das quais a sintaxe das DSLs LindaX geralmente não dispõe. Conforme já mencionado, várias dessas informações relativas a uma determinada descrição de arquitetura estão embutidas em uma ferramenta extensora específica (vide abaixo), que é identificada pela referência, na descrição da arquitetura, a um estilo particular;
4. `Expander`: gera, a partir da identificação de um estilo LindaX específico (armazenado no repositório de estilos), informações estruturais e semânticas detalhadas da DSL LindaX correspondente a uma descrição de arquitetura. Essa ferramenta atua como um "*back-end*" para a ferramenta `Translator`, não sendo acessível diretamente a projetistas de sistemas;
5. `Generator`: gera, a partir de estilos LindaX, mecanismos de controle de adaptação para diferentes plataformas de programação. Esses mecanismos são construídos a partir de um *framework* para gerência de adaptações (vide Seção 5.4);
6. `Critic`: detecta erros sintáticos e semânticos em estilos e descrições de arquitetura. Essa ferramenta atua assincronamente em relação às outras ferramentas do ambiente. Mais especificamente, `Critic` é notificada a respeito de alterações, efetuadas pelas ferramentas de edição, nos repositórios de descrições e estilos. Em face dessas notificações, `Critic` revalida as descrições alteradas, eventualmente alimentando um terceiro repositório, de críticas, que é consultado sob demanda pelas ferramentas de refinamento e síntese.

A Figura 5.1 (um detalhamento da figura apresentada no Capítulo 1) ilustra o relacionamento entre as ferramentas supracitadas. Para lidar com a extensibilidade de LindaX, cada uma dessas ferramentas é composta por um *driver* principal, ao qual *plug-ins* específicos são acoplados. *Plug-ins* para a ferramenta `Expander`, por exemplo, são definidos para cada estilo específico, constituindo assim um conjunto de ferramentas extensoras. Dessa forma, um *plug-in* associado a um determinado estilo é utilizado pela ferramenta `Expander` para extrair informações estruturais e semânticas de uma descrição de arquitetura que referencia o estilo, por meio de uma propriedade `Style`. Essa mesma idéia é utilizada nas ferramentas `Translator` e `Generator`, para permitir o refinamento e a síntese de descrições em LindaX para diferentes alvos, bem como nas ferramentas `StyleEditor` e `ConfEditor`, para permitir o uso de diferentes notações alternativas (textuais ou gráficas) à sintaxe de XML. Já na ferramenta `Critic`, *plug-ins* podem ser utilizados em verificações mais especializadas que simplesmente a detecção de erros sintáticos e erros semânticos simples. Um exemplo é a verificação formal de consistência entre restrições (duas ou mais restrições, quando combinadas, levam a um conjunto vazio de configurações válidas?).[1]

Note, na Figura 5.1, que ambas as ferramentas `Translator` e `Generator` recebem como entrada não somente descrições de arquitetura e de estilo, mas também arquivos auxiliares. Esses arquivos permitem a definição de valores para propriedades parametrizadas nessas descrições. Diferentes arquivos, com valores distintos para essas propriedades, podem ser usados sobre um mesmo conjunto de descrições, viabilizando o reuso dessas descrições em relação aos alvos de refinamento e síntese das ferramentas do ambiente. O formato dos arquivos de propriedades é textual, sendo composto por linhas do tipo

```
nome_propriedade_parametrizada = valor
```

Assim, um tipo de componente `UIOModule` cuja propriedade `Behaviour` tem como valor o parâmetro `behv` de `UIOModule` pode ser parametrizado em um arquivo de propriedades por meio da linha

```
UIOModule.behv = “valor da propriedade Behaviour para UIOModule”
```

[1] Na versão atual do ambiente LindaStudio, foram implementados somente *plug-ins* de edição para a notação sem *tags* utilizada ao longo de todo o Capítulo 3. Com relação à verificação de erros semânticos, foram implementados, para a ferramenta `Critic`, somente *plug-ins* que detectam erros simples, como, por exemplo, a ausência de propriedades obrigatórias em uma descrição de arquitetura ou estilo.

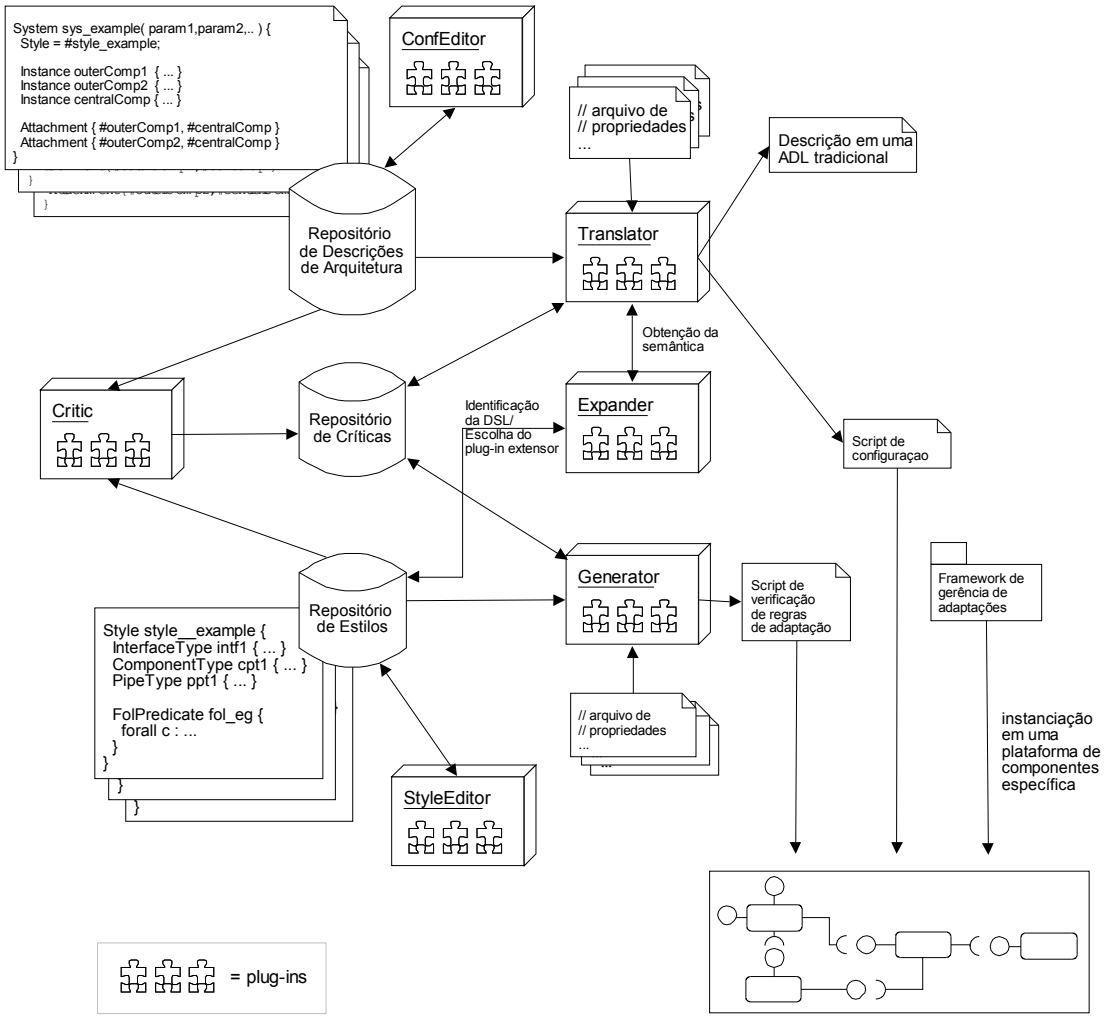

Figura 5.1. Visão geral do ambiente LindaStudio.

5.2.
Arquitetura de software do ambiente LindaStudio

O ambiente LindaStudio é integrado ao ambiente de desenvolvimento de software ArchStudio 3 (The Regents of the University of California, 2004). O ambiente ArchStudio oferece um arcabouço estrutural para a edição, integração, comparação, verificação e "execução" de descrições de arquiteturas baseadas em XML. Esse ambiente utiliza um esquema XML central, chamado `xArch`, sobre o qual todas as ferramentas do ambiente devem saber operar e a partir do qual linguagens específicas podem ser construídas.

A Figura 5.2 ilustra de que modo as ferramentas LindaStudio se inserem no arcabouço estrutural do ArchStudio. Os elementos hachurados na figura são

implementações das ferramentas LindaStudio. Para tornar possível o uso das ferramentas genéricas providas pelo ArchStudio, os esquemas-base de LindaX foram todos definidos a partir de `xArch`. Com isso, mesmo manipulações de baixo nível de descrições LindaX podem ser feitas através dessas ferramentas, sem a necessidade de edição dos documentos XML correspondentes diretamente.

Quatro ferramentas ArchStudio são largamente utilizadas no LindaStudio: `ArchEdit`, `xArchADT`, `FileManager` e `Critic`. `ArchEdit` é um editor de documentos `xArch` particularmente útil na edição e inspeção de baixo nível de especificações LindaX. `XArchADT` e `FileManager` implementam um repositório de documentos `xArch`, que é usado pelo LindaStudio ao mesmo tempo como repositório de descrições de arquitetura e de estilos LindaX. `Critic` é, na verdade, um *framework* para desenvolvimento de ferramentas de verificação de erros. Esse *framework* é composto de um repositório de críticas (`CriticADT`), uma interface gráfica de apresentação de críticas (`CriticGUI`), um "renderizador" de críticas para `CriticGUI` (`DefaultCriticArtist`)[2] e gerentes de críticas (`CriticManager` e `CriticManagerGUI`). No LindaStudio, esse *framework* foi especializado para permitir a verificação de erros sintáticos e semânticos de LindaX.

O arcabouço estrutural do ambiente ArchStudio segue o estilo C2 (Taylor et al., 1996), pelo qual um sistema é uma rede estratificada de artefatos de computação (componentes) que se comunicam por meio de trocas de mensagens através de artefatos de difusão de mensagens (conectores-barramento). Todo artefato em C2 possui interfaces 'superiores' e 'inferiores', de modo que uma interface superior de um artefato deve se ligar obrigatoriamente a uma interface inferior de outro artefato – de onde surge a noção de uma rede 'estratificada' de artefatos. Uma particularidade do estilo C2 é que componentes só podem se comunicar via conectores-barramento, porém esses conectores podem estar diretamente ligados entre si (mas, ainda assim, respeitando as regras de estratificação). Outra característica desse estilo é o princípio da visibilidade restrita: um componente em um determinado estrato do sistema só pode enviar requisições endereçadas a componentes acima dele na hierarquia. Porém, esse componente pode responder ou enviar notificações a estratos inferiores.

No ambiente ArchStudio, todas as ferramentas são implementações em Java de componentes C2 que se comunicam assincronamente por meio de conectores-

[2] Para uma apresentação mais especializada de críticas, novos renderizadores podem ser acoplados ao ArchStudio.

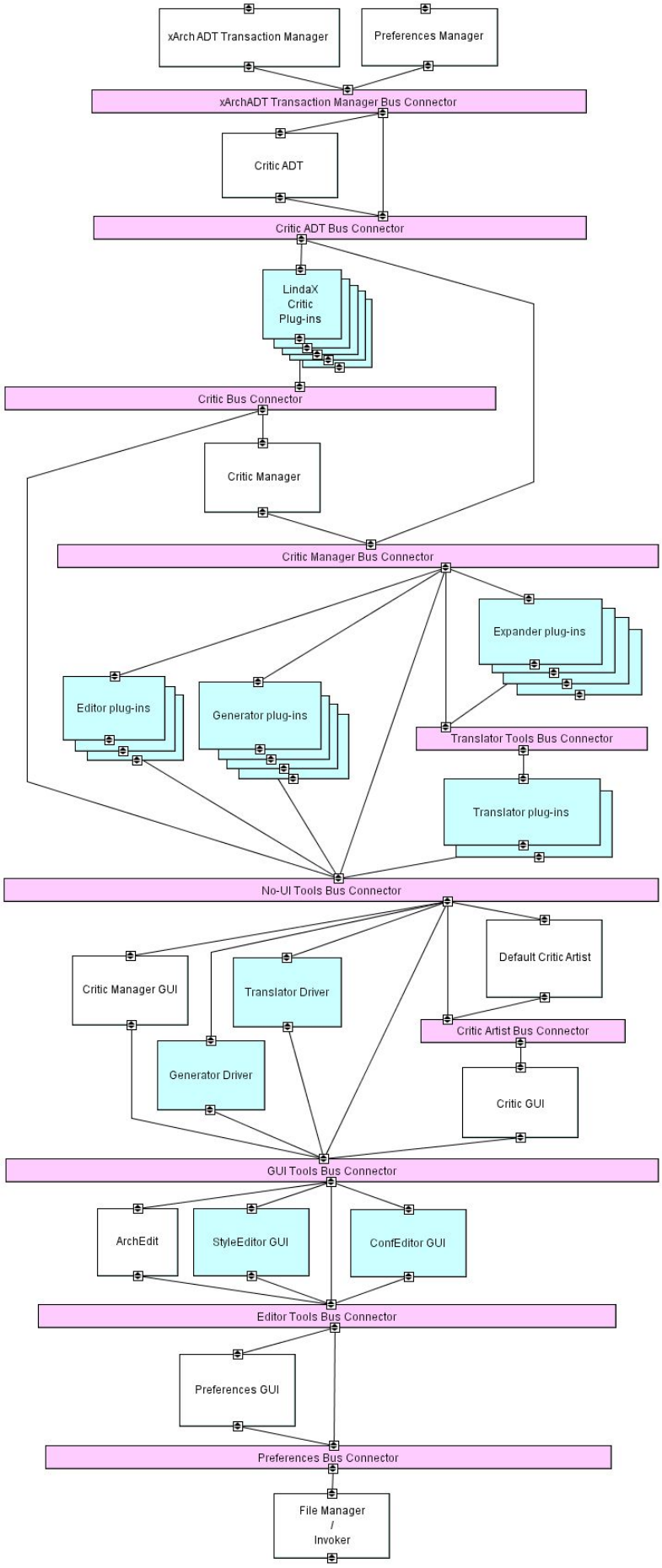

Figura 5.2. Arquitetura de software do ambiente LindaStudio.

barramento. Essa constituição facilita sobremaneira o desenvolvimento e implantação de novos *plug-ins* para as ferramentas do LindaStudio, o que é um aspecto importante dada a extensibilidade que a linguagem LindaX oferece.

Neste capítulo, somente as ferramentas `Translator`, `Generator` e `Expander` são exploradas mais a fundo. Outros detalhes de implementação do ambiente LindaStudio estão no Apêndice A. O leitor interessado nas especificidades das outras ferramentas do ambiente pode consultar também (Laboratório TeleMídia, 2004), de onde o pacote de software completo que implementa o ambiente LindaStudio pode ser obtido.

5.3. A ferramenta `Translator`

A ferramenta `Translator` é responsável por gerir os processos de refinamento de descrições de arquitetura em LindaX. Cada *plug-in* dessa ferramenta tem como alvo uma plataforma de programação, FDT ou ADL específica. De forma geral, cada um desses *plug-ins* inicia o processo de refinamento mapeando o sistema de tipos de LindaX para o sistema de tipos correspondente em uma linguagem ou modelo-alvo específico. Mapeamento similar ocorre em relação a descrições de *templates* em LindaX para tipos compostos nesse alvo.[3] A partir daí, as descrições de sistemas em uma DSL podem ser refinadas para a notação-alvo correspondente.

É importante reiterar que, nas DSLs LindaX, constituintes de configuração – em particular, instâncias, *pipes* e mapeamentos – podem estar apenas parcialmente definidos em uma descrição de arquitetura. Outras dessas definições estarão tipicamente implícitas na semântica da DSL LindaX em questão. O detalhamento dessas definições é passado à ferramenta `Translator` por um *plug-in* da ferramenta `Expander` específico para essa DSL. Note também que, no processo de refinamento para uma plataforma de programação, descrições de tarefas e provedores (esquema-base `lindaxres`, apresentado no Capítulo 3) podem ser usadas para especificar aspectos não-funcionais, em especial a distribuição dos componentes de uma configuração ou a associação dos mesmos a escopos de gerência de recursos.[4]

[3] Caso não haja suporte a tipos compostos na linguagem ou modelo-alvo, instâncias de configurações são criadas diretamente, para cada instância de *template* definida.

Conforme ilustrado na Figura 5.3, *plug-ins* da ferramenta `Translator` implementam a interface `ITranslator`, que oferece uma única operação de refinamento. *Plug-ins* da ferramenta `Expander` implementam a interface `IExpander`, que oferece operações de detalhamento de instâncias, *pipes* e mapeamentos. As operações de `IExpander` retornam estruturas de dados representando esses detalhamentos, que são processadas pelos *plug-ins* da ferramenta `Translator` mediante requisições de refinamento. Detalhes acerca do refinamento de descrições em DSLs LindaX são identificados no contexto dos estudos de caso apresentados na Seção 5.3.1 e, mais adiante, na Seção 5.5.1.

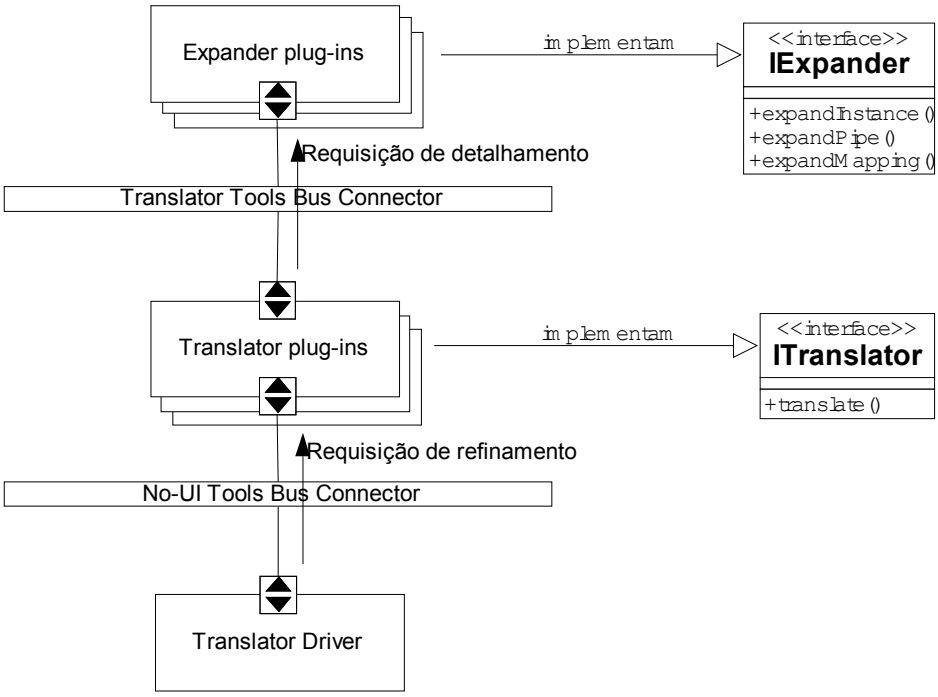

Figura 5.3. Estrutura da ferramenta `Translator`.

5.3.1. Estudo de caso

Como estudo de caso de aplicação da ferramenta `Translator`, foi utilizada a ADL Wright (Allen, 1997), uma linguagem que se baseia na descrição formal do comportamento de artefatos arquiteturais. Wright provê uma base formal para a

[4] Na versão atual do ambiente LindaStudio, não são providas ferramentas que "costuram" automaticamente especificações da visão arquitetural e de recursos (conforme nomenclatura adotada no Capítulo 2). Essa funcionalidade, mencionada novamente adiante neste capítulo, foi deixada como trabalho futuro.

descrição tanto de configurações quanto de estilos. Ela se distingue de outras ADLs pelo fato de, simultaneamente: (i) promover artefatos de comunicação a entidades de primeira ordem (conectores), (ii) permitir a descrição do comportamento de seus artefatos (tanto componentes quanto conectores) utilizando uma notação baseada em CSP (Hoare, 1985), (iii) caracterizar estilos por meio de predicados definidos sobre instâncias de configurações e (iv) permitir várias verificações estáticas que determinam a consistência e completeza de especificações.[5] O estudo de caso foca no refinamento de descrições em DSLs LindaX para especificações 'arquiteturais' em Wright. Os *plug-ins* de refinamento ligados a esse estudo de caso não tratam da especificação de comportamentos. No máximo, vislumbrou-se que esse tipo de especificação fosse permitido a partir de propriedades parametrizadas. Os valores desses parâmetros, definidos em arquivos de propriedades que alimentam a ferramenta `Translator`, podem ser utilizados pelos projetistas para descrever comportamentos utilizando a notação baseada em CSP proposta em (Allen, 1997).

5.3.1.1. A linguagem Wright

Wright é uma ADL que foca em dois aspectos principais de uma descrição de arquitetura: (i) a estrutura arquitetural e (ii) o comportamento arquitetural.

Uma estrutura em Wright é um grafo de componentes interligados por conectores. Componentes e conectores são tipados, e os tipos podem ser declarados tanto diretamente na descrição de uma arquitetura quanto separadamente, em estilos Wright. Componentes têm interfaces, que em Wright são chamadas de portas. Conectores também têm interfaces, que em Wright são denominadas papéis (*roles*). A cardinalidade dessas interfaces é estabelecida na descrição do tipo que as detém, por meio de índices associados a elas. Wright define também tipos para interfaces, que prescrevem as características de portas e papéis.

Um conjunto de componentes interligados por conectores define uma configuração Wright. Caso uma configuração seja definida a partir de estilos, a cardinalidade das instâncias dos tipos na configuração e a topologia de

[5] De fato, as características – positivas e negativas – de Wright foram umas das principais fontes de inspiração para várias decisões de projeto acerca da linguagem LindaX. Outros trabalhos que também contribuíram nessas decisões são apresentados no Capítulo 6.

acoplamentos (*attachments*) entre essas instâncias devem obedecer as restrições declaradas nos estilos.

A Figura 5.4 mostra a forma geral de especificação arquitetural baseada em estilos de um sistema cliente-servidor em Wright. No exemplo, tipos de interface são definidos para as portas e papéis de componentes e conectores. O tipo de componente `serv` na figura é parametrizado em relação ao número de componentes que podem se acoplar a instâncias desse tipo.

```
1  Style sty
2   Interface Type CliSide = [especificação dos eventos em CSP]
3   Interface Type SrvSide = [especificação dos eventos em CSP]
4
5   Component cli
6    Port p = CliSide
7    Computation C = [especificação da computação em CSP]
8
9   Component serv( nports:1.. )
10   Port p1..nports = SrvSide
11   Computation C = [especificação da computação em CSP]
12
13  Connector conn
14   Role r_in = CliSide
15   Role r_out = SrvSide
16   Glue G = [especificação da ligação em CSP]
17
18  Constraints
19   especificação das restrições em lógica de predicados de 1a. ordem
20 End Style
21
22 Configuration conf
23  Style sty
24
25  Instances
26   c1  : cli
27   c2  : cli
28   s   : serv( 2 )
29   cn1 : conn
30   cn2 : con
31
32  Attachments
33   c1.p as cn1.r_in
34   cn1.r_out as s.p1
35   c2.p as cn2.r_in
36   cn2.r_out as s.p2
37 End Configuration
```

Figura 5.4. Especificação arquitetural estática em Wright.

O modo como um componente aceita determinados eventos em certas portas e como, em resposta, produz novos eventos em outras portas caracteriza uma computação de um componente em Wright. O modo como esses eventos são trocados entre os componentes através dos papéis dos conectores caracteriza uma ligação (*glue*). O comportamento de uma arquitetura descrita em Wright é caracterizada por computações e ligações. A notação utilizada para representar comportamentos arquiteturais em Wright é uma adaptação de CSP. A partir da notação estrutural em Wright e da especificação comportamental baseada em CSP,

ferramentas Wright podem gerar especificações em CSP "puro", permitindo assim a validação formal estática da arquitetura por meio de ferramentas CSP apropriadas (Formal Systems (Europe) Limited, 2003).

Wright também permite a definição de componentes e conectores compostos, o que é feito a partir da especificação das computações de componentes (ou das ligações de conectores) como configurações. Assim, o componente ou conector composto age como uma fachada para um subsistema. Em composições, mapeamentos (*bindings*) devem ser feitos entre as portas/papéis livres dos constituintes e as portas/papéis que a composição oferece externamente ao subsistema. A Figura 5.5 ilustra um exemplo de um componente composto.

```
1  Component compoundComp
2   Port port_in = ...
3   Port port_out = ...
4   Computation
5    Configuration internalConf
6     Component C1
7      Port p1 = ...
8      Port p2 = ...
9      Computation C = ...
10
11    Component C2
12     Port p1 = ...
13     Port p2 = ...
14     Computation C = ...
15
16    Connector IntPipe
17     Role Source = ...
18     Role Sink = ...
19     Glue G = ...
20
21    Instances
22     X  : C1
23     Y  : C2
24     XY : IntPipe
25
26    Attachments
27     X.p2 as XY.Source
28     Y.p1 as XY.Sink
29   End Configuration
30
31   Bindings
32    X.p1 = port_in
33    Y.p2 = port_out
34   End Bindings
```

Figura 5.5. Especificação de componente composto em Wright.

Wright foi definida a princípio para a especificação de arquiteturas estáticas somente. Em (Allen et al., 1998), foram introduzidas novas notações em Wright para caracterizar mudanças na arquitetura durante a computação dos componentes. Essa abordagem, contudo, é limitada a sistemas com um conjunto finito de configurações possíveis e, dessa forma, não dá suporte a reconfigurações "*ad-hoc*". Por isso, essa evolução de Wright não foi explorada na tese presente.

5.3.1.2. Refinamento de DSLs LindaX para Wright

O *plug-in* da ferramenta `Translator` associado a Wright refina descrições em DSLs LindaX com base na convenção apresentada na Tabela 5.1.

Tabela 5.1. Convenção de refinamento de DSLs LindaX para descrições Wright.

DSL LindaX	Wright
Descrição de sistema	Configuração arquitetural
Tipo de componente ou *pipe* em um estilo ligado à DSL	Tipo de componente ou conector primitivo
Instância de tipo de componente	Instância de tipo de componente primitivo
Descrição de *template*	Não se aplica
Instância de *template*	Configuração arquitetural

A Figura 5.6 mostra o resultado do refinamento da descrição, na DSL LindaRouter, do sistema `FastPath`, apresentado no Capítulo 4. Outros exemplos de refinamento para Wright, a partir de descrições em uma versão anterior da DSL LindaQoS, podem ser encontrados em (Soares-Neto, 2003a).

Várias considerações acerca do processo de refinamento para Wright podem ser feitas tendo como base a Figura 5.6.

Primeiro, estilos LindaX não são refinados para estilos Wright. O interesse da tese presente no refinamento para Wright consiste, fundamentalmente, no seu suporte à descrição formal de comportamentos. Nesse sentido, o refinamento de estilos Wright traz um ganho potencialmente baixo, haja visto que as ferramentas dessa linguagem não oferecem verificações específicas de estilo, como a de adequação entre configurações e restrições de estilo ou a de consistência entre restrições. As verificações oferecidas pelas ferramentas Wright se resumem, basicamente, na adequação entre tipos e instâncias (substituição de parâmetros, validação de índices) e naquelas decorrentes da geração de CSP "puro" a partir de descrições em Wright (consistência entre comportamento de computação de componente/ligação e porta/papel, inexistência de *deadlocks*) e que, portanto, são oferecidas, efetivamente, pelas ferramentas CSP. É interessante notar que o ambiente LindaStudio dá margem à agregação de verificações (semi-)formais de estilos a Wright. Por exemplo, os *plug-ins* da ferramenta `Expander` permitem, indiretamente, a verificação de adequação entre configuração e restrições de estilo,

uma vez que a semântica do estilo é embutida nesse *plug-in*. Além disso, os *scripts* de verificação gerados pela ferramenta `Generator` (vide Seção 5.4) permitem a revalidação dessas restrições em tempo de execução. Contudo, esse aspecto não foi devidamente explorado nesta tese, uma vez que isso demandaria uma análise formal dos processos de refinamento e síntese propostos.

A segunda consideração diz respeito ao refinamento de tipos de componentes. No estilo LindaX `PCBasedForwarder`, dois tipos de componente – `UClassifier` e `UProcessor` – foram definidos. No refinamento para Wright, no entanto, três tipos foram gerados. Mais especificamente, o tipo `UClassifier` em LindaX foi refinado para dois tipos que refletem funções distintas de classificação e encaminhamento (linhas 2 a 5 e 7 a 12 na figura). Isso decorre do fato de que, nas DSLs LindaX, os tipos referenciados são considerados, para efeito de refinamento, como esboços incompletos de instâncias – conforme mencionado no Capítulo 3, descrições de portas e pontos de acesso em um tipo LindaX identificam o 'mínimo' que deve ser provido por instâncias desse tipo. Tipos específicos são construídos em uma linguagem-alvo a partir do processamento da descrição de arquitetura em LindaX pelo *plug-in* correspondente ao estilo na ferramenta `Expander`. No caso do estilo `PCBasedForwarder`, esse *plug-in* gera um tipo na linguagem-alvo para cada arranjo diferente de acoplamentos entre componentes e *pipes* definido na descrição de arquitetura.

Em terceiro lugar, as descrições em CSP dos comportamentos associados aos componentes e conectores em Wright, bem como suas portas e papéis, são definidos a partir da propriedade parametrizada `Behaviour`. Nesse caso, um arquivo de propriedades próprio para o refinamento para Wright deve conter as descrições CSP correspondentes. Note que, nas linhas 10, 41, 43 e 45 da Figura 5.6, são declaradas e utilizadas portas indexadas em Wright. Esses índices são também inferidos pelo *plug-in* correspondente ao estilo na ferramenta `Expander` a partir do arranjo de acoplamentos entre componentes e *pipes*. Contudo, é importante ressaltar que o uso de portas indexadas em Wright se reflete na especificação em CSP do comportamento do componente, complicando a definição do mesmo a partir de propriedades. Isso deve ser previsto pelo projetista que especifica a arquitetura e essas propriedades.

```
1  Configuration FastPath
2   Component CPType_UClassifier_1
3    Port p_1 = [descrição CSP da interação push out
4                com CNType_PushPipe_push_impl]
5    Computation = [descrição CSP do comportamento de classificação]
6
7   Component CPType_UClassifier_2
8    Port p_1 = [descrição CSP da interação push in
9                com CNType_PushPipe_push_impl]
10   Port p_2₃ = [descrição CSP da interação pull out
11               com CNType_PullPipe_pull_impl]
12   Computation = [descrição CSP do comportamento de encaminhamento]
13
14  Component CPType_UProcessor_1
15   Port p_1 = [descrição CSP da interação pull in
16               com CNType_PullPipe_pull_impl]
17   Computation = [descrição CSP do comportamento de escalonamento]
18
19  Connector CNType_PushPipe_1
20   Role in = [descrição CSP da interação push in]
21   Role out = [descrição CSP da interação push out]
22   Glue = [descrição CSP do protocolo de interação push]
23
24  Connector CNType_PullPipe_1
25   Role in = [descrição CSP da interação pull in]
26   Role out = [descrição CSP da interação pull out]
27   Glue = [descrição CSP do protocolo de interação pull]
28
29  Instances
30   classifier                      : CPType_UClassifier_1
31   forwarder                       : CPType_UClassifier_2
32   scheduler1,scheduler2,scheduler3 : CPType_UClassifier_1
33   _implConn_1                     : CNType_PushPipe_1
34   _implConn_2                     : CNType_PullPipe_1
35   _implConn_3                     : CNType_PullPipe_1
36   _implConn_4                     : CNType_PullPipe_1
37
38  Attachments
39   classifier.p_1 as _implConn_1.in
40   _implConn_1.out as forwarder.p_1
41   forwarder.p_2₁ as _implConn_2.in
42   _implConn_2.out as scheduler1.p_1
43   forwarder.p_2₂ as _implConn_3.in
44   _implConn_3.out as scheduler2.p_1
45   forwarder.p_2₃ as _implConn_4.in
46   _implConn_4.out as scheduler3.p_1
47 End Configuration
```

Figura 5.6. Refinamento de descrição na DSL LindaRouter para Wright.

Por fim, Wright não dá suporte ao compartilhamento de constituintes de configurações. As questões de refinamento para Wright surgidas em decorrência dessa limitação podem ser melhor compreendidas através do exemplo da Figura 5.7. Na configuração (a), assume-se que `A`, `E`, `F`, `G` e `H` são instâncias de componentes primitivos em LindaX. É clara a necessidade de planificação dessa configuração antes do refinamento para Wright, pelo fato de `A` ser compartilhado (através de `B`) entre `C` e `D`. A configuração (b) apresenta um possível resultado da planificação de (a), a partir da qual o refinamento para Wright é viável. Note, contudo, que o constituinte compartilhado `A` pode ser sucessivamente exteriorizado, por meio de mapeamentos, através de `B` e de uma das duas

composições `C` ou `D`, ou de ambas. Nesse caso, se um *pipe* fosse também estabelecido entre `G` e `A`, por exemplo, a planificação proposta em (b) poderia não ser a mais adequada, uma vez que, na configuração (a), `G` se comunica com `A` por meio de `C`, o que deveria, a princípio, ser representado em (b) por meio de um *pipe* que "atravessasse" `C`. Uma vez que, em LindaX, composições não agregam funcionalidade adicional em relação a seus constituintes, preferiu-se uma estratégia de planificação total, em que configurações intermediárias são sempre eliminadas, conforme ilustrado na configuração (c). Uma conseqüência disso é que não se refina descrições de *templates* em LindaX para tipos compostos de componentes em Wright, conforme seria de se esperar. *Templates* são vistos, no processo de refinamento para Wright, como "plantas" (*blueprints*) de configurações.

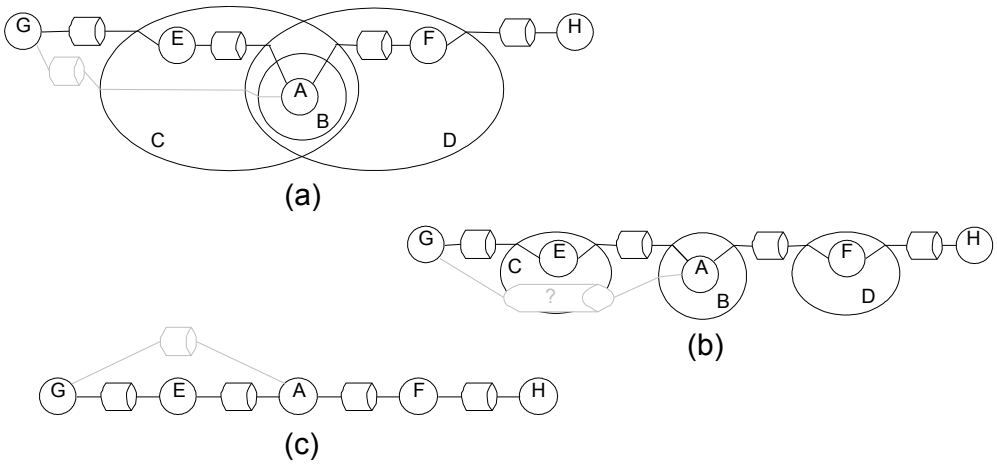

Figura 5.7. Planificação de configurações LindaX.

5.4. *Framework* para gerência de adaptações

Para tornar mais homogênea a implementação dos mecanismos de composição e de controle de adaptações em diferentes plataformas de programação e, conseqüentemente, simplificar o desenvolvimento de *plug-ins* para as ferramentas `Translator` e `Generator`, foi definido um *framework* para gerência de adaptações. Esse *framework* é genérico em relação à linguagem de especificação em uso (LindaX é apenas uma das possibilidades), à plataforma de programação subjacente e às regras de adaptação que ele impõe a composições de

seus elementos. Embora o *framework* não possua, a princípio, limitações com respeito ao uso de modelos de programação variados, nesta tese o foco são as plataformas de programação baseadas em componentes.

O *framework* para gerência de adaptações engloba três elementos principais: um 'controlador genérico de adaptações' (*Generic Adaptation Controller* – GAC), um 'controlador transacional' (*Transaction Controller* – TC) e um 'configurador' (*ConfiGurator* – CG). A Figura 5.8 ilustra uma visão geral da estrutura e do funcionamento do *framework*, envolvendo todos os seus elementos. Cada um desses elementos é apresentado em detalhe nas subseções que se seguem.

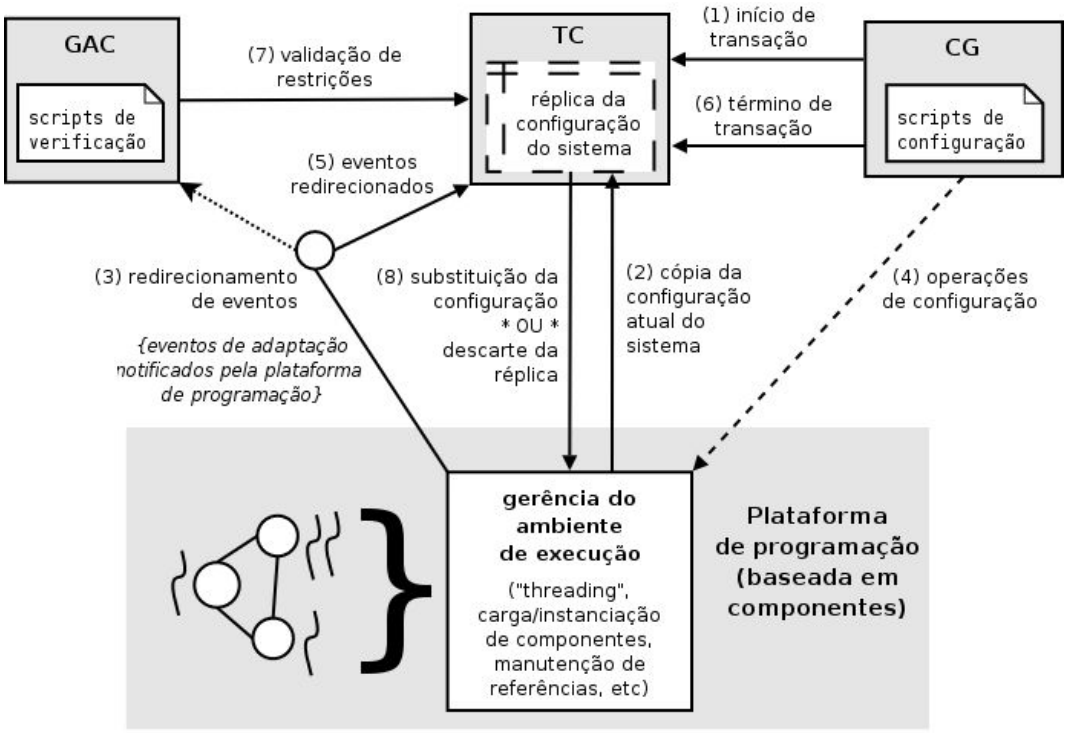

Figura 5.8. *Framework* para gerência de adaptações.

5.4.1. O controlador genérico de adaptações (GAC)

O GAC é responsável por gerir e impor regras de adaptação arbitrárias em qualquer configuração (possivelmente distribuída) de componentes. O GAC pode detectar eventos de adaptação (seta pontilhada da Figura 5.8) e verificar se esses eventos incorrem em violações em um conjunto bem definido de regras. Exemplos de eventos de adaptação em plataformas de componentes incluem: adição ou remoção de componentes em uma configuração, alocação ou liberação de

referências a interfaces de componentes (representando associações ou dissociações entre componentes) etc. A definição de regras de adaptação é específica de domínio e pode variar consideravelmente. Por isso, o GAC é configurável com relação às regras por ele conhecidas.

Regras de adaptação são representadas no *framework* para gerência de adaptações como *scripts* – tipicamente, usando uma linguagem de programação interpretada – que são gerados, a partir de estilos LindaX, por *plug-ins* da ferramenta `Generator`. O GAC pode ser configurado para interpretar esses *scripts* via requisições explícitas do programador ou automaticamente, sempre que eventos de adaptação são detectados. É interessante observar que o GAC oferece um nível adicional de reuso de código em plataformas de componentes, uma vez que a única parte sujeita a modificações nesse elemento são os *scripts* por ele geridos. Além disso, os *scripts* de verificação conferem aos mecanismos de controle de adaptação uma alta manutenibilidade, uma vez que esses *scripts* podem ser atualizados em tempo de execução, sem necessidade de recodificação ou recompilação do GAC.

5.4.2. O controlador transacional (TC)

Em diversas situações, o programador de um sistema adaptável pode fazer uso dos serviços transacionais providos pelo TC para efetivar (*commit*) ou desfazer (*roll back*) adaptações. A principal função do TC é manter a consistência de configurações quando múltiplos eventos de adaptação relacionados ocorrem – por exemplo, a instanciação de um componente seguida da associação desse componente a uma das interfaces de outro componente.

O TC provê um mecanismo de transação que trata múltiplos eventos de adaptação como um único evento atômico. Quando uma transação de adaptação é iniciada (passo `(1)` da Figura 5.8), o TC cria uma réplica da configuração atual do sistema (passo `(2)`) e desabilita – caso esteja ativo – o procedimento automático de verificação do GAC (passo `(3)`). Posteriormente, todos os eventos de adaptação são redirecionados a essa réplica (passo `(5)`). Quando a transação é encerrada (passo `(6)`), o TC requisita ao GAC a verificação da réplica modificada (passo `(7)`). Se as regras de adaptação forem satisfeitas, o TC aplica as alterações

ocorridas na réplica sobre a configuração original, efetivando a transação. Senão, a réplica modificada é descartada e a transação desfeita (passo `(8)`).[6]

5.4.3. O configurador (CG)

A configuração de uma arquitetura complexa por meio de chamadas diretas a operações oferecidas por uma determinada plataforma de componentes pode exigir do programador um esforço considerável de desenvolvimento. Além de essas operações deverem ser chamadas em uma ordem bem definida (uma associação entre componentes só pode ser criada após a instanciação dos mesmos!), o programador deverá preocupar-se com detalhes variados de implementação, como a iniciação e término de transações e a gerência de referências a componentes e associações entre eles, para citar alguns. O CG facilita a tarefa dos programadores no que se refere a essas questões. Esse elemento implementa um mecanismo de composição automática pelo qual uma configuração de sistema pode ser gerada a partir de um *script* que a descreve de modo declarativo. Em geral, o CG atua em consonância com o GAC e o TC, conforme ilustrado nos passos (`1`) e (`6`) da Figura 5.8. No entanto, o CG pode ser usado na ausência de ambos, caso em que esse elemento ignoraria os serviços de transação e requisitaria diretamente a execução de operações de configuração à plataforma de componentes subjacente (passo (`4`) da Figura 5.8).

O formato para *scripts* de configuração entendido pelo CG não é especificado no *framework*. Vislumbra-se, no entanto, três alternativas principais: (i) um formato baseado em XML (descrições em LindaX, por exemplo, seriam candidatas naturais), (ii) um formato declarativo proprietário (uma ADL ou algo similar, porém mais simples, como uma linguagem de interconexão de módulos (*Module Interconnection Language* – MIL)) ou (iii) uma estrutura de dados descrita em uma linguagem interpretada. A questão de qual alternativa vem a ser a mais apropriada é explorada na Seção 5.5.1. Independente da alternativa adotada, o ambiente LindaStudio permite, através da implementação de diferentes *plug-ins* para a ferramenta `Translator`, a geração automática de *scripts* de configuração a partir de descrições de arquitetura em uma DSL LindaX qualquer.

[6] O esquema descrito baseia-se na técnica de 'atualizações adiadas' (*deferred updates*), comumente utilizada em sistemas de gerência de bancos de dados (Silberschatz et al. 2001). O *framework* não impede, porém, que outras técnicas de manutenção de atomicidade seja usadas.

5.5. A ferramenta `Generator`

A ferramenta `Generator` é responsável por gerir os processos de síntese de mecanismos de composição e de controle de adaptações. Nesta tese, o foco dos processos de síntese são as plataformas de programação baseadas em componentes, embora a ferramenta `Generator` não possua, a princípio, limitações com respeito ao uso de outros modelos de programação.

Conforme ilustrado na Figura 5.9, *plug-ins* da ferramenta `Generator` implementam a interface `IGenerator`, que oferece uma única operação de síntese. Implementações dessa operação nos diferentes *plug-ins* têm como alvo os mecanismos de controle de adaptações. Os mecanismos de composição são alvo da ferramenta `Translator`, conforme será visto no estudo de caso da Seção 5.5.1.

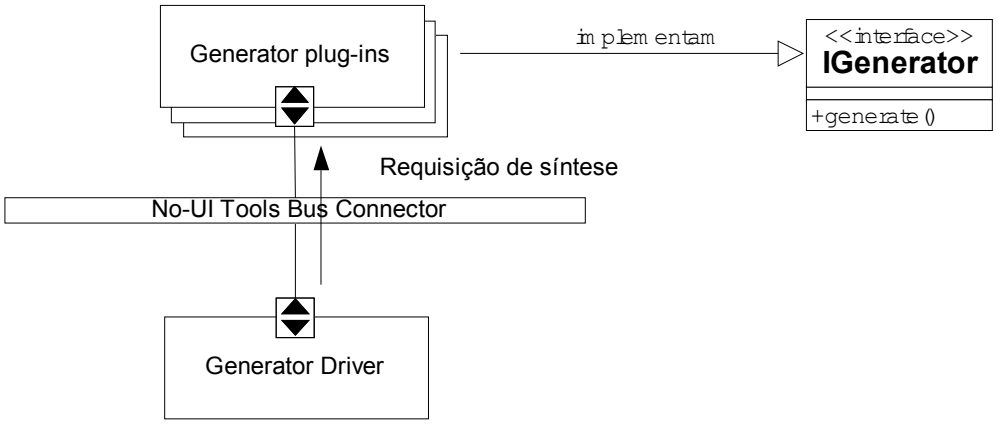

Figura 5.9. Estrutura da ferramenta `Generator`.

5.5.1. Estudo de caso

Para ilustrar a aplicação integrada do *framework* para gerência de adaptações e das ferramentas de refinamento e síntese, foi feito um estudo de caso envolvendo o *toolkit* NetKit (Coulson et al., 2003). NetKit é composto de um conjunto de componentes de software especializado na configuração de arquiteturas de rede programável. Esse *toolkit* não pressupõe nenhuma arquitetura de hardware, linguagem de programação ou sistema operacional específico, sendo perfeitamente adequado como estudo de caso. Além disso, o modelo de

componentes subjacente ao NetKit, o OpenCOM (Coulson et al., 2004), oferece facilidades de reflexão (Maes, 1987) que permitem a inspeção e adaptação de sistemas de modo genérico e organizado, separando claramente a operação de um sistema e a gerência de adaptações do mesmo. Por fim, o NetKit emprega o conceito de '*framework* de componentes' (Szyperski, 2002) para permitir a definição de composições de software que obedecem a regras de adaptação de domínio específico.

O estudo de caso foca na construção sistemática de *frameworks* de componentes a partir de descrições em LindaX. Para esse fim, o *framework* genérico para gerência de adaptações, introduzido na Seção 5.4, foi instanciado junto ao modelo de componentes do NetKit. Essa instância atua como um '*metaframework*' de componentes, de modo que um *framework* de componentes atuando sobre um domínio específico pode ser criado a partir do estabelecimento de regras no *metaframework*. Essa abordagem possibilita um alto grau de reuso de projeto e implementação entre *frameworks* de componentes NetKit distintos. Além disso, ela confere a esses *frameworks* de componentes uma alta manutenibilidade – alterações nas suas regras de adaptação podem ser feitas em tempo de execução.

5.5.1.1. O *toolkit* NetKit e a plataforma OpenCOM

O NetKit consiste em uma biblioteca de componentes especializada para redes programáveis. Essa biblioteca abrange todos os níveis presentes em uma arquitetura de rede programável, desde as funções de processamento de pacotes no plano de dados até as funções de mais alto nível de sinalização e coordenação. Por meio das facilidades de reflexão oferecidas pelo seu modelo de componentes, o OpenCOM, o NetKit permite uma alta flexibilidade na implantação, instanciação e adaptação dessas funções em ambientes de execução envolvendo diferentes arquiteturas de hardware de roteador e sistemas operacionais variados. Pelo fato da instanciação do *framework* para gerência de adaptações sobre o NetKit se apoiar diretamente nos serviços providos pelo OpenCOM, esta seção será mais dedicada à apresentação do modelo de componentes do que do NetKit em si.

O modelo OpenCOM define seis conceitos principais (vide Figura 5.10): componentes, interfaces, receptáculos, cápsulas, associações (*bindings*) locais e metamodelos.

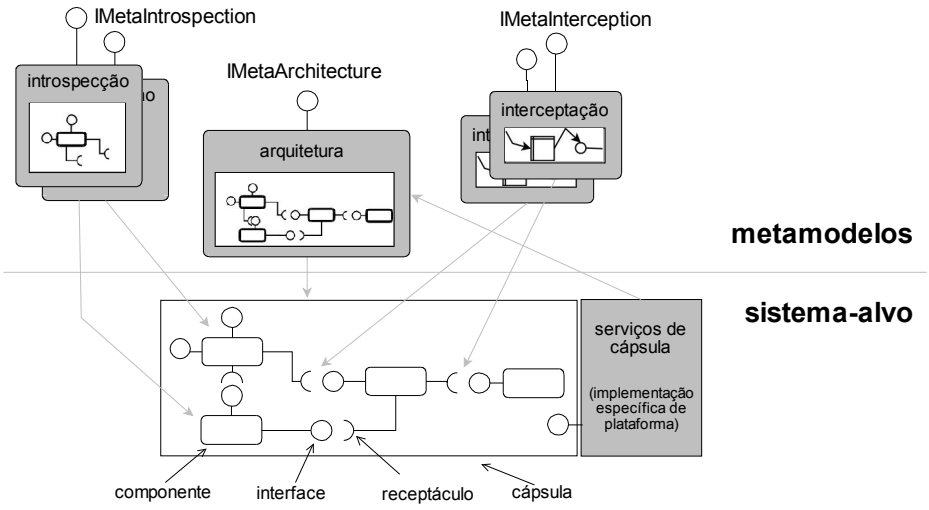

Figura 5.10. O modelo de componentes OpenCOM.

Um componente OpenCOM define um tipo para módulos de software univocamente identificado por um GUID (*General Universal IDentifier*). Componentes oferecem serviços a outros componentes por meio de interfaces e requisitam serviços a interfaces de outros componentes por meio de receptáculos. Interfaces são imutáveis e fortemente tipadas – o tipo de uma interface[7] também é identificado por um GUID – podendo admitir três modos de interação: (i) invocações de operações; (ii) fluxos contínuos; e (iii) sinais. Receptáculos tornam explícitas possíveis dependências entre componentes. Assim, quando um componente é instanciado é possível determinar a partir de seus receptáculos se outros componentes ou, mais precisamente, se interfaces de determinado tipo sendo oferecidas por outros componentes, devem estar presentes para que o componente recém-instanciado possa funcionar corretamente.

Cápsulas delineiam escopos de carga (instanciação) e de associação local (conexão entre receptáculos e interfaces) para componentes. O objetivo principal das cápsulas é uniformizar a implantação de componentes em diferentes ambientes de execução. Ao prover uma interface única de acesso a serviços de

[7] Interfaces são definidas a partir de uma 'linguagem de definição de interface' (*Interface Definition Language* – IDL) similar à CORBA (Object Management Group, 1996). Na versão atual do OpenCOM, o compilador de IDL oferecido permite somente a geração de esqueletos de componentes em C++.

carga e associação de componentes, a cápsula esconde as peculiaridades de cada ambiente. O conceito de cápsula introduz assim uma separação de papéis entre programadores. Programadores 'de implantação' fazem a ponte entre o modelo de componentes e o ambiente concreto de execução, implementando os serviços de carga e associação nesses ambientes. Programadores 'de sistemas' são isolados das peculiaridades de cada ambiente por meio desses serviços.

O nível de distribuição física de uma cápsula é arbitrário e dependente do ambiente de execução. Em casos extremos pode-se considerar até mesmo uma cápsula como abrangendo diferentes máquinas ligadas em rede. Porém, o propósito original das cápsulas é que elas sejam 'fortemente acopladas', ou seja, o conceito de 'associação local' assume conexões entre receptáculos e interfaces que sejam obrigatoriamente confiáveis, determinísticas e de baixíssimo retardo, como no caso de NPUs, de processos comunicantes em uma mesma máquina ou de redes embarcadas. O objetivo é viabilizar o uso de associações locais em comunicações que demandam alta vazão, como no encaminhamento de pacotes no plano de dados. Associações remotas ou complexas (por exemplo, uma associação multiponto) são construídas no OpenCOM sob a forma de composições (possivelmente distribuídas) de componentes (*rich bindings*). A Figura 5.11 ilustra um exemplo de associação remota no OpenCOM.[8]

Metamodelos são mecanismos de reflexão (Maes, 1987) – implementados no OpenCOM como componentes – que permitem gerir, inspecionar e adaptar metadados de uma configuração. Metamodelos mantêm uma relação causal entre metadados e configurações, de modo que mudanças nos metadados – em geral ocorridas em face de operações requisitadas nas interfaces dos metamodelos (as 'metainterfaces') – se refletem automaticamente em mudanças na configuração e vice-versa. Os metamodelos independem de um suporte nativo a reflexão nas linguagens de programação usadas na confecção de componentes. Contudo, essa independência acarreta limitações na abordagem de gerência de adaptações proposta nesta seção, quando linguagens não-reflexivas são utilizadas. Essa questão é abordada novamente no Capítulo 7.

[8] Pelo fato de *rich bindings* no OpenCOM serem implementados como componentes, eles também possuem tipos e GUIDs correspondentes. Vale mencionar que essa característica do modelo de se representar associações remotas como componentes distribuídos é explorada na implementação de uma plataforma de *middleware* com alto grau de adaptabilidade, chamada OpenORB (Blair et al., 2001).

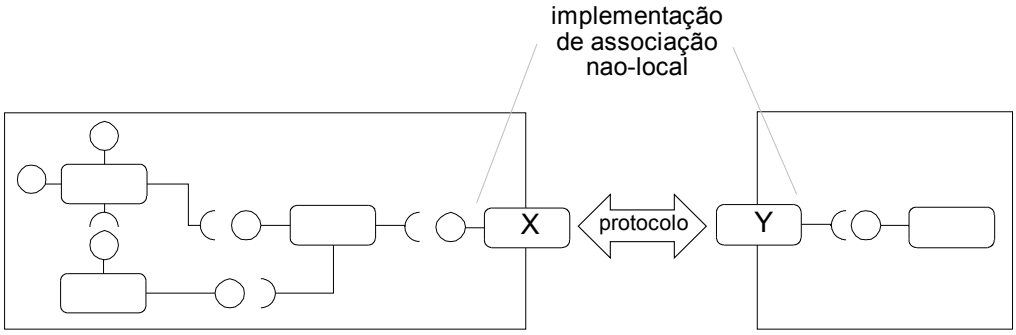

Figura 5.11. Associações "não-locais" no OpenCOM. A especificação do protocolo de comunicação utilizado entre X e Y não é englobada pelo modelo.

O OpenCOM provê três metamodelos principais (vide Figura 5.10). O metamodelo de introspecção dá suporte a inspeção e adaptação limitada (isto é, exposição e oclusão) de interfaces e receptáculos. O metamodelo de arquitetura representa a topologia de uma configuração como um grafo que pode ser inspecionado ou alterado. A invocação dos serviços de carga e associação local das cápsulas também se reflete em alterações nesse grafo. Finalmente, o metamodelo de interceptação permite a associação de 'interceptadores' a operações, fluxos ou sinais de uma interface particular. Interceptadores são pequenos módulos de software (não necessariamente componentes) que podem ser executados antes, durante ou após invocações em uma interface. Outros metamodelos complementares vêm sendo introduzidos no OpenCOM, como o metamodelo de recursos (Durán-Limón, 2001), que permite gerir a alocação de recursos (tempo de CPU, espaço de memória etc) a configurações.

5.5.1.2. *Frameworks* de componentes no NetKit

O NetKit aplica o conceito de *frameworks* de componentes (*Component Frameworks* – CFs) (Szyperski, 2002) na definição de arquiteturas de rede programável. Tradicionalmente, *frameworks* são vistos como esqueletos para a construção de software que permitem o reuso de módulos e de decisões de projeto. Porém, *frameworks* (CFs) no NetKit não são um conceito de projeto somente. Eles são representados explicitamente em tempo de execução por 'componentes controladores' que restringem o modo como os componentes de uma configuração podem se comportar e interagir. Esse controle é baseado em regras específicas de

domínio que não podem ser, em geral, descritas simplesmente pelo uso de receptáculos e metamodelos. Por exemplo, um CF de protocolos pode ser construído por meio de controladores que detêm conhecimento específico de como as entidades de protocolo de uma arquitetura em camadas podem interagir.

Como regra geral, funcionalidades de controle específicas de domínio são construídas sobre receptáculos e metamodelos. Considere novamente o exemplo do CF de protocolos. Imaginando que um programador resolva substituir um componente em uma determinada camada da pilha de protocolos, os controladores do CF podem basear-se no grafo mantido pelo metamodelo de arquitetura para manipular o arranjo topológico da pilha – por exemplo, restringindo conexões diretas entre o novo componente e componentes em camadas não-adjacentes. Esses controladores podem também utilizar o metamodelo de introspecção para verificar se o novo componente oferece uma determinada interface de passagem de pacotes, requerida naquela camada. O metamodelo de interceptação pode ser aplicado durante o processo de remoção do componente antigo e inserção do novo componente, para reduzir ao mínimo a interrupção dos fluxos de dados que estejam sendo processados pela pilha, por exemplo, através da "bufferização" dos pacotes junto a um receptáculo momentaneamente desconectado. Por fim, o metamodelo de recursos pode ser acionado para alocar recursos de CPU (*threads*) e memória (*buffers*) ao novo componente.

5.5.1.3. Aplicação do *framework* para gerência de adaptações no NetKit

Apesar de propiciarem um certo nível de reuso, CFs agregam conhecimento de domínio específico e, por isso, demandam mecanismos de controle de adaptação também específicos, que em geral não podem ser reusados. A manutenção de CFs – por exemplo, atualizações nas regras de adaptação – é um ponto ainda mais crítico no projeto desses artefatos.

Um grau maior de reuso pode ser conseguido partindo do princípio de que, apesar das regras de adaptação serem específicas de domínio, a forma como elas são verificadas é normalmente a mesma. Isto é, em face de um evento de adaptação – por exemplo, a instanciação de um componente ou uma nova associação entre componentes – o controlador responsável pela configuração em questão é acionado. A partir daí, regras específicas de domínio são aplicadas para

validar a adaptação desejada. Nesta tese, é proposta a aplicação do *framework* para gerência de adaptações, apresentado na Seção 5.4, na construção de um 'metaCF', a partir do qual CFs distintos podem ser construídos por meio da atribuição de regras de adaptação específicas.

5.5.1.4. Gerência de adaptações

A implementação do controlador genérico de adaptações (GAC) no OpenCOM demandou três tarefas principais: (i) a definição de um mecanismo para detecção automática de eventos de adaptação, (ii) a escolha de uma linguagem apropriada para os *scripts* de verificação e (iii) a provisão de um mecanismo que dê a esses *scripts* acesso a informações sobre as configurações.

O mecanismo para detecção automática de eventos de adaptação foi construído sobre o metamodelo de interceptação do OpenCOM. Interceptadores são associados às operações de adaptação fornecidas pelos outros metamodelos, permitindo assim a identificação e tratamento dos eventos associados.

Com relação aos *scripts* de verificação, um interpretador com suporte à carga dinâmica de *scripts* foi encapsulado no GAC. A linguagem de *script* escolhida para ser hospedada no GAC foi Lua (Ierusalimschy et al, 2003), por causa de sua alta flexibilidade, simplicidade e leveza – o interpretador Lua é menor que o de outras linguagens de *script* conhecidas, por exemplo Perl (Wall e Schwartz, 1991) e Tcl (Ousterhout, 1994). Ainda assim, Lua possui facilidades de descrição e manipulação eficiente de dados[9], que são particularmente úteis em configurações em lote (Seção 5.5.1.6).

Para permitir aos *scripts* Lua o acesso aos metadados mantidos pelos metamodelos, uma API Lua especial (chamada `netkit`) foi encapsulada no GAC.[10] Essa API oferece aos *scripts* as seguintes operações principais de inspeção:

[9] Além de tipos primitivos convencionais (números, cadeias de caracteres, booleanos etc), Lua define um tipo especial de *array* associativo, denominado 'tabela', que mimifica estruturas de dados variadas, tais como enumerações, conjuntos, listas e registros, para citar alguns. Qualquer menção, neste capítulo, a uma estrutura de dados em Lua implica no uso de tabelas.

[10] Uma outra API, chamada `util`, também foi definida, com uma série de operações de comparação genérica de valores de variáveis Lua (`isEqual`), manipulação de tabelas Lua como conjuntos (`PowerSet`, `SequenceSet`), entre outras. As operações dessa API acrescentam às APIs Lua básicas (em especial, `table`) a funcionalidade necessária para o mapeamento semântico correto entre os predicados em lógica de primeira ordem dos estilos LindaX e os *scripts* Lua. Detalhes sobre essa API podem ser encontrados em (Laboratório TeleMídia, 2004).

- `enumInstances(n)`: enumera os componentes, presentes em uma configuração, cujo tipo seja identificado pelo GUID `n`. Caso não seja passado parâmetro algum a essa operação, são enumerados todos os componentes da configuração. Cada elemento da enumeração é uma dupla `{guid,nome}`, onde `nome` é uma referência unívoca do componente na configuração;
- `enumPorts(ref)`: enumera pontos de interação (interfaces e receptáculos) de um componente específico referenciado por `ref`. Cada elemento da enumeração é uma tripla `{guid,ind,uses/provides}`, onde `guid` identifica o tipo do ponto de interação, `ind` é o seu índice em relação aos outros pontos de interação do componente e `uses` (ou `provides`) é uma estrutura que identifica se o ponto de interação é uma interface ou receptáculo e que dá informações gerais sobre a associação da qual o ponto de interação participa (se for o caso);
- `getType(ref)`: devolve o GUID de um componente, interface ou receptáculo específico, referenciado por `ref`;
- `enumBindings()`: enumera as associações locais entre componentes em uma configuração (*rich bindings* são considerados como componentes pela API, sendo, portanto, enumeráveis através de `enumInstances`). Cada elemento da enumeração é uma estrutura contendo a quádrupla

  ```
  {{guid_comp1,nome_comp1},{guid_recp,ind_recp,uses},
   {guid_comp2,nome_comp2},{guid_intf,ind_intf,provides}}
  ```

  que identifica os dois componentes, o receptáculo e a interface participantes da associação local;
- `getBinding(ref_C1,ref_R,ref_C2,ref_P)`: devolve uma quádrupla (como a de `enumBindings`) com informações sobre uma associação específica, identificada pelas referências aos dois componentes (`ref_C1` e `ref_C2`) e ao receptáculo (`ref_R`) e interface (`ref_P`) participantes da mesma.

Scripts de verificação são codificados como funções Lua que devolvem valores booleanos. Quando o GAC é acionado para verificar uma configuração, ele executa os *scripts* e admite a configuração somente se todos devolverem o valor "*true*". A Figura 5.12 ilustra um *script* que verifica a existência de um componente do tipo `uQoSNegGUID` ao qual todos os componentes do tipo

`uAdmCtrlGuid` na configuração devem estar conectados, por meio de interfaces do tipo `pIntraLevelGUID` (chamadas à API `netkit` estão em negrito na figura).

```
1  function evalRuleAdmCtrlsWithQoSNeg()
2    local uQoSNegGUID     = "412ece3a-0f39-45cf-86d2-fbf27f365e13"
3    local uAdmCtrlGUID    = "eb582791-4841-4bad-8722-b31b08e6145d"
4    local pIntraLevelGUID = "2b383111-61b9-47de-ab09-f7fdecb60f65"
5
6    local existsExp_1 = false
7    for _, c in ipairs( netkit.enumInstances( uQoSNegGUID ) ) do
8      local forAllExp_1 = true
9      for _, n in ipairs( netkit.enumInstances( uAdmCtrlGUID ) ) do
10       local existsExp_2 = false
11       for _, pc in ipairs( netkit.enumPorts( c ) ) do
12         local existsExp_3 = false
13         for _, pn in ipairs( netkit.enumPorts( n ) ) do
14           local boolExp =
15             table.foreachi(
16               netkit.enumBindings(),
17               function( _,e )
18                 if util.isEqual( e,netkit.getBinding( c,pc,n,pn ) )
19                   then return true end
20               end
21             )
22             and
23             util.isEqual( netkit.getType( pn ),pIntraLevelGUID )
24
25           if boolExp then
26             existsExp_3 = true; break end
27         end -- for
28
29         if existsExp_3 then
30           existsExp_2 = true; break end
31       end -- for
32
33       if not existsExp_2
34         then forAllExp_1 = false; break end
35     end -- for
36
37     if forAllExp_1
38       then existsExp_1 = true; break end
39   end -- for
40
41   return existsExp_1
42 end -- function
```

Figura 5.12. Exemplo de *script* de verificação em Lua.

É interessante notar que o exemplo da Figura 5.12 é uma descrição algorítmica do predicado em lógica de primeira ordem `AdmCtrlsWithQoSNeg` do estilo `CentralizedNQoS`, apresentado no Capítulo 4. A Figura 5.12 representa justamente parte do resultado da execução do *plug-in* de síntese específico para o OpenCOM sobre a descrição desse estilo. Os GUIDs representados na figura são obtidos a partir da propriedade parametrizada `Behaviour` de cada um dos tipos declarados no estilo `CentralizedNQoS`. Os valores dessa propriedade são descritos separadamente do estilo, em um arquivo de propriedades que alimenta a ferramenta `Generator`.

Outra observação importante acerca do exemplo da Figura 5.12 é o mapeamento de referências a *pipes*. Conforme mencionado no Capítulo 4, a inexistência da propriedade `Behaviour` na declaração de um tipo de *pipe* pode ser interpretada pelas ferramentas de refinamento e síntese como o uso do método de interação mais "simples" provido pela plataforma de programação ou linguagem de especificação alvo. No OpenCOM, esse método é a associação local (vide linhas 16 e 18 no exemplo da figura).

Uma última consideração a respeito da geração de *scripts* de verificação é que ela não ocorre a partir de restrições de estilos somente. Algumas propriedades restritivas dos tipos declarados em um estilo também podem ser usadas sob a forma de *scripts*. O principal exemplo é a propriedade `Cardinality`, que limita o número de instâncias de um tipo de componente ou *pipe*, ou então de portas e pontos de acesso desses elementos. A Tabela 5.2 relaciona trechos típicos de *scripts* de verificação em Lua às principais construções encontradas em estilos LindaX, a partir das quais esses trechos são gerados.

O componente OpenCOM que implementa o GAC oferece duas interfaces principais (vide Figura 5.13). A interface `ICFControl` permite gerenciar as restrições impostas pelo CF em tempo de execução, através de operações de inserção, remoção e substituição de *scripts* de verificação. A interface `ICFValidation` oferece uma única operação de verificação das regras definidas por esses *scripts*. Quando invocada, essa operação executa os *scripts*. Estes, por sua vez, disparam, através da API Lua descrita acima, uma série de operações de inspeção (por meio de receptáculos apropriados) nos metamodelos. Para esse fim, o GAC disponibiliza à API um conjunto de receptáculos associados às metainterfaces dos metamodelos.

Na implementação atual, o componente GAC só inspeciona os metamodelos de introspecção e arquitetura. Contudo, esses metamodelos já conferem a esse componente uma boa abrangência de eventos de adaptação reconhecidos. A IDL das interfaces providas pelo GAC é apresentada no Apêndice A.

Tabela 5.2. Mapeamento das estruturas de estilos LindaX em trechos de código em Lua.

<table>
<tr><th>Parte do estilo</th><th>Elemento-fonte</th><th>Código Lua gerado</th></tr>
<tr><td rowspan="4">Tipos de componente e pipe (nesses casos, geram-se scripts completos)</td><td>Propriedade no tipo de componente/pipe T

Cardinality=x;</td><td>1 -- tGUID é obtido a partir da
2 -- propriedade Behaviour
3 -- do tipo T.
4 function evalRule...()
5 return table.getn(
6 netkit.enumInstances(tGUID)) == x
7 end</td></tr>
<tr><td>Propriedade no tipo de componente/pipe T

Cardinality= {x..y};</td><td>1 function evalRule...()
2 local val = table.getn(
3 netkit.enumInstances(tGUID))
4
5 return val >= x and
6 val <= y
7 end</td></tr>
<tr><td>Propriedade em portas ou pontos de acesso do tipo TI no tipo de componente/ pipe TC

Cardinality=x;</td><td>1 function evalRule...()
2 local retval = true
3 for _,c in ipairs(
4 netkit.enumInstances(tcGUID)) do
5 local numP = 0
6 for _,p in ipairs(
7 netkit.enumPorts(c) do
8 if p.guid == tiGUID
9 then numP = numP + 1 end
10 end
11
12 if numP ~= x
13 then retval = false; break end
14 end
15
16 return retval
17 end</td></tr>
<tr><td rowspan="5">Predicados em lógica de primeira ordem (nesses casos, o código Lua equivalente representa um trecho de script)</td><td>Operador universal

forall e : S {
[P]
}</td><td>1 -- P é uma função hipotética que retorna
2 -- o valor booleano correspondente à
3 -- expresão [P] na lógica de 1a. ordem.
4 local forallExp = true
5 for _,e in ipairs(S) do
6 p = P(e)
7 if not p
8 then forallExp = false; break end
9 end</td></tr>
<tr><td>Operador existencial

exists e : S {
[P]
}</td><td>1 local existsExp = false
2 for _,e in ipairs(S) do
3 p = P(e)
4 if p
5 then existsExp = true; break end
6 end</td></tr>
<tr><td>Operadores e funções sobre símbolos</td><td>-- várias operações distribuídas pelas
-- APIs netkit e util.</td></tr>
<tr><td>Operadores de predicado ==, !=, >, >=, <=</td><td>-- combinações no uso das operações
-- util.isEqual e table.getn</td></tr>
<tr><td>Operador de predicado

[v in S]</td><td>1 -- val obtém o valor booleano associado
2 -- ao teste de pertinência de v no
3 -- conjunto S
4 local val = table.foreachi(
5 S,
6 function(_,e)
7 if util.isEqual(e,v)
8 then return true end
9 end
10)</td></tr>
</table>

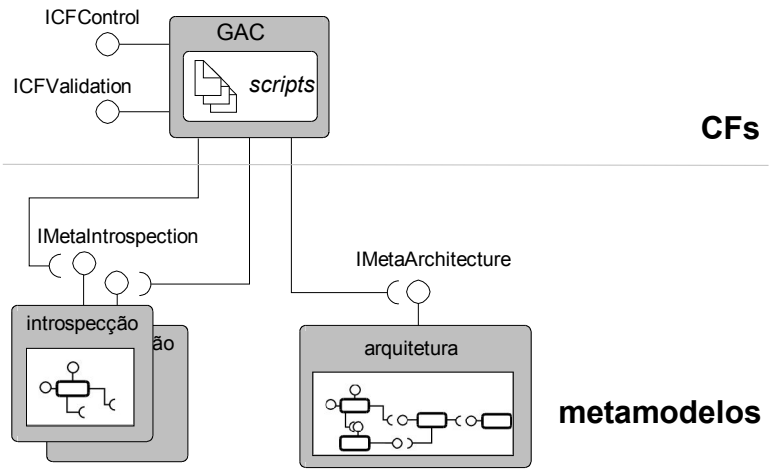

Figura 5.13. O componente GAC no OpenCOM.

5.5.1.5. Adaptações transacionais

A implementação do controlador transacional (TC) no OpenCOM envolveu dois aspectos: (i) desenvolver uma estratégia de 'procuração' (*proxying*) entre o GAC e os metamodelos do OpenCOM e (ii) definir um mecanismo pelo qual o TC conseguisse gerar réplicas dos metadados desses metamodelos. A dinâmica de funcionamento do TC, em relação a esses dois aspectos, é representada na Figura 5.14.

A estratégia de *proxying* do TC foi implementada por meio de interceptadores. Quando uma transação é iniciada, o TC requisita ao metamodelo de interceptação a inclusão de interceptadores que redirecionam os eventos de adaptação ao TC.[11] Em seguida, os receptáculos para metainterfaces providos pelo GAC são alterados de modo a serem associados a interfaces do TC que mimificam as metainterfaces originais dos metamodelos. Isso permite que o GAC efetue a verificação de regras sobre a réplica dos metadados mantida pelo TC.

Para poder gerar uma réplica dos metadados relativos a uma configuração, o TC associa seus próprios receptáculos às metainterfaces dos metamodelos. A partir daí, a réplica dos metadados é criada por meio de invocações a uma série de operações de inspeção providas por essas metainterfaces. Todo esse arranjo de interceptadores e receptáculos montado no início da transação é desfeito após o seu término.

[11] Essa operação deve ser precedida da desabilitação dos interceptadores utilizados pelo GAC, caso ele esteja verificando automaticamente as suas regras de adaptação.

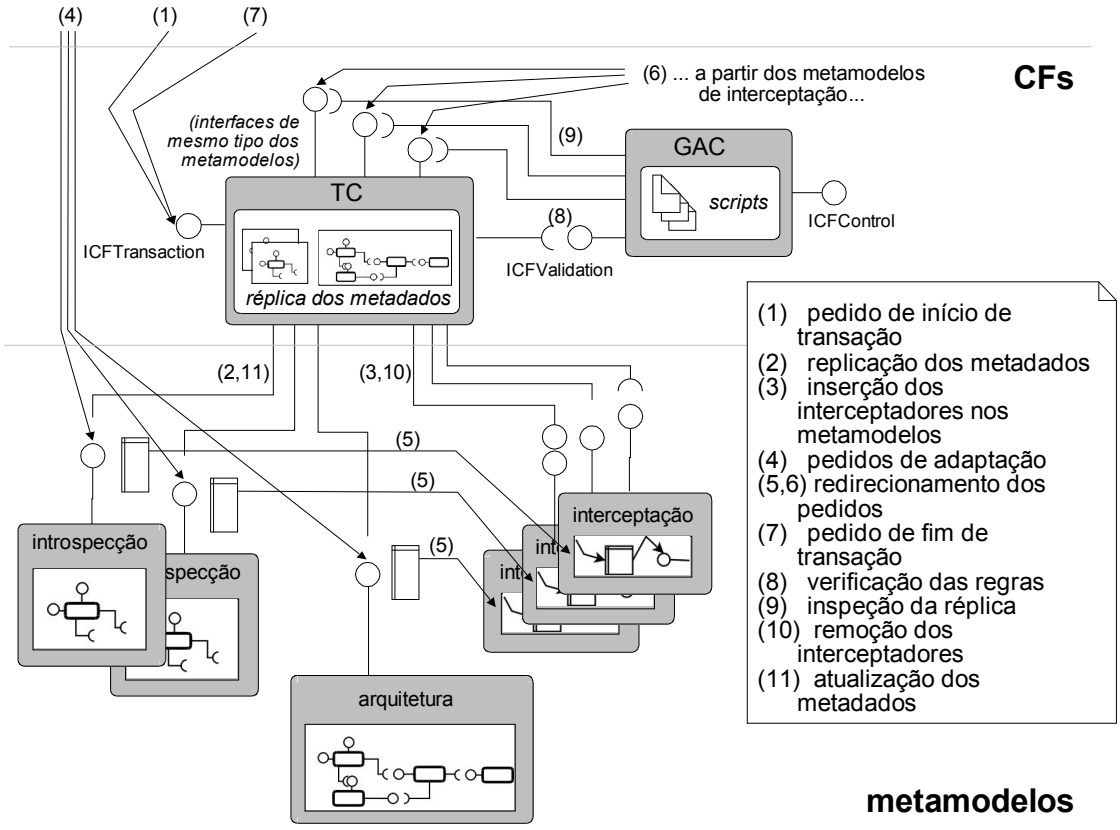

Figura 5.14. Serviço de adaptação transacional no OpenCOM.

Duas observações importantes podem ser feitas acerca dos mecanismos descritos acima.

Primeiro, o serviço transacional fornecido pelo TC é transparente para quem requisita as adaptações, o que facilita sua adoção em sistemas pré-existentes. No entanto, a definição de eventos atômicos para múltiplas adaptações associadas depende do TC ser informado sobre o início e fim das transações. Isso pode ser conseguido de duas formas. Na primeira delas, o componente OpenCOM que implementa o TC oferece, por meio de uma interface `ICFTransaction` (vide Figura 5.14), operações que permitem a um componente externo indicar explicitamente o início e fim de transação.[12] Na segunda forma, assume-se que múltiplas adaptações associadas ocorram em lote, de modo que o TC pode inferir automaticamente o início e fim de transação.

A segunda observação diz respeito à eficiência do serviço transacional fornecido pelo TC. Ela é dependente do número de eventos de adaptação ocorridos durante a transação. Em cenários complexos, uma transação pode até ser

[12] Essa foi a alternativa adotada na implementação atual do *framework* para gerência de adaptações no OpenCOM.

mais eficiente do que os eventos de adaptação sendo tratados de modo isolado (mesmo levando em conta o processo de geração da réplica dos metadados), uma vez que as regras de adaptação podem ser verificadas uma única vez (ao término da transação).

5.5.1.6. Configurações programadas

A questão principal de implementação do configurador (CG) no OpenCOM foi a escolha do formato a ser utilizado para se descrever configurações em lote. Escolheu-se novamente, para esta tese, a linguagem Lua, devido ao fato do *parser* Lua ser muito menor e mais rápido do que *parsers* para outros formatos declarativos pesquisados, como XML. Isso permitiu a implementação do CG como um componente OpenCOM leve e eficiente. Esse componente possui uma única interface `ICFConfiguration` (vide Figura 5.15), que oferece operações de definição e obtenção de descrições de configuração em Lua.

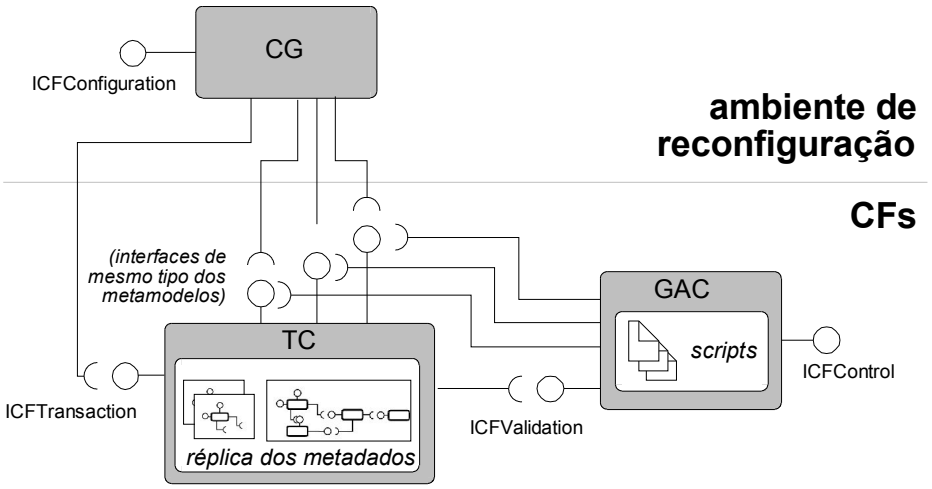

Figura 5.15. Implementação do CG no OpenCOM.

A Figura 5.16 mostra um exemplo de *script* de configuração (e a representação gráfica associada a essa configuração) entendido pelo componente OpenCOM que implementa o CG. Essa descrição é obtida a partir de uma descrição em LindaQoS relacionada ao estilo `CentralizedNQoS` apresentado no Capítulo 4. Conforme ilustrado na figura, aninhamentos de tabelas Lua são utilizados para representar descrições de configuração.

```
1  uQoSNegGUID = "412ece3a-0f39-45cf-86d2-fbf27f365e13"
2  uAdmCtrlGUID = "eb582791-4841-4bad-8722-b31b08e6145d"
3  pIntraLevelGUID = "2b383111-61b9-47de-ab09-f7fdecb60f65"
4
5  sample_configuration = {
6    outerComp1 = {
7      guid = uAdmCtrlGUID;
8      ports = {
9        [1] = {
10         guid = pIntraLevelGUID; ind = 0;
11         uses = {comp = "centralComp"; intfind = 0;};
12       };
13     };
14   };
15   outerComp2 = {
16     guid = uAdmCtrlGUID;
17     ports = {
18       [1] = {
19         guid = pIntraLevelGUID; ind = 0;
20         uses = {comp = "centralComp"; intfind = 1;};
21       };
22     };
23   };
24   centralComp = {
25     guid = uQoSNegGUID;
26     ports = {
27       [1] = {
28         guid = pIntraLevelGUID; ind = 0;
29         provides = {comp = "outerComp1"; recpind = 0;};
30       };
31       [2] = {
32         guid = pIntraLevelGUID; ind = 1;
33         provides = {comp = "outerComp2"; recpind = 0;};
34       };
35     };
36   };
37 };
```

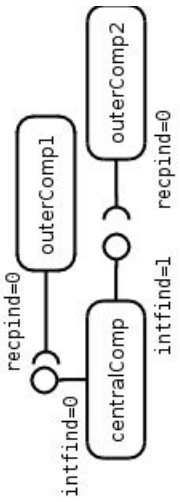

Figura 5.16. Descrição de configuração em Lua

Assim como ocorre com os *scripts* de verificação, o programador não precisa descrever configurações diretamente em Lua. A ferramenta `Translator` possui um *plug-in* específico que permite refinar descrições de arquitetura em uma DSL LindaX qualquer para *scripts* de configuração cujo formato é entendido pelo *parser* Lua hospedado no CG. Os valores de GUID representados no *script* da Figura 5.16 são obtidos a partir de propriedades cujos valores são descritos separadamente do estilo, em um arquivo de propriedades que alimenta a ferramenta de refinamento. A determinação, nesse *script*, do tipo de ponto de interação – interface ou receptáculo – de cada um dos componentes é feita a partir do atributo `Direction`, presente na descrição do tipo de componente correspondente. No formato de *scripts* entendido pelo componente CG do OpenCOM, isso é indicado pelo uso de tabelas `uses` ou `provides`, aninhadas nas tabelas descritoras de portas de componentes (linhas 11, 20, 29 e 33 no exemplo).

É importante ressaltar que, no caso de configurações distribuídas (por exemplo, em uma descrição em LindaQoS associada ao estilo

`DistributedNQoS`), descrições como a ilustrada acima precisam ser dotadas de informações adicionais acerca da localidade dos componentes. No caso do OpenCOM, essas informações são definidas por uma dupla `{endereço_IP,id_de_cápsula}`. Essas informações poderiam ser obtidas a partir da descrição de tarefas e provedores, porém essa funcionalidade não foi implementada na versão atual do ambiente LindaStudio. Assim, essas informações devem ser, atualmente, inseridas pelo projetista de modo manual na descrição de configuração em Lua (como entradas nas tabelas Lua correspondentes) antes da submissão dessa descrição ao CG. Outra questão importante, não tratada pela implementação atual do *framework* para gerência de adaptações no OpenCOM, é que só adaptações locais são controladas; isto é, configurações distribuídas são aceitas pelo CG, mas o controle de adaptações distribuídas por parte do TC e do GAC, não.

5.5.1.7. Implementação em NPUs

Os ambientes de experimentação principais do *framework* para gerência de adaptações têm sido as NPUs IXP da Intel Corporation® (2004). A Figura 5.17 apresenta a arquitetura geral de um roteador baseado na unidade IXP1200. O roteador consiste em um PC ligado à unidade IXP1200 por meio de um barramento PCI. A unidade IXP1200 contém um processador central StrongARM, um arranjo de processadores RISC – as 'micromáquinas' – e um conjunto de processadores dedicados. O StrongARM é responsável pelo tratamento de pacotes no plano de controle e por outras funções de gerência. As micromáquinas são responsáveis pelo encaminhamento de pacotes no plano de dados. Os processadores dedicados (não mostrados na figura) executam tarefas específicas como verificação de redundância cíclica, cálculo de dispersão (*hash*) etc. Por fim, uma arquitetura hierárquica de memória compartilhada (também não mostrada) interliga os processadores presentes na unidade.

Na implementação atual do OpenCOM sobre o IXP1200, uma única cápsula engloba todo o roteador, incluindo o PC hospedeiro (vide figura acima). Desse modo, o programador não precisa estar ciente das idiossincrasias da unidade, uma vez que elas são abstraídas pelo OpenCOM. Os componentes projetados para executar no StrongARM e nas micromáquinas são carregados nesses

processadores e interligados por associações "locais" de modo transparente para o programador. Detalhes do mapeamento do OpenCOM em unidades IXP encontram-se em (Coulson et al., 2003).

O GAC e o TC são sempre instanciados no PC hospedeiro, junto aos metamodelos e serviços de cápsula do OpenCOM. A princípio, o CG pode ser instanciado tanto nessa mesma cápsula quanto remotamente – por exemplo, na máquina onde o ambiente LindaStudio está sendo executado. Dois conjuntos de instâncias do GAC e do TC foram criados[13], um responsável pelo tratamento dos pedidos de carga e associação local entre componentes nas micromáquinas e outro por esse mesmo tratamento no StrongARM. A instância do GAC responsável pelas micromáquinas é dotada de *scripts* de verificação gerados a partir dos estilos de LindaRouter, enquanto a responsável pelo StrongARM detém *scripts* de verificação gerados a partir dos estilos de LindaQoS. Conforme mencionado nos Capítulos 1 e 4, embora a integração entre as duas DSLs e seus estilos correspondentes seja crucial no que se refere ao tratamento de dependências de adaptação entre diferentes aspectos, essa questão não é explorada nesta tese. Isso se reflete, na implementação sobre o roteador IXP1200, no controle disjunto de adaptações ocorridas nas micromáquinas e no StrongARM.

A forma como os pedidos de adaptação são entregues às instâncias do GAC e do TC depende do paradigma de rede programável usado no sistema. Em um ambiente de gerência de adaptação local essas instâncias são acessíveis por meio de aplicações residentes no PC hospedeiro. Nessa abordagem, adotada na implementação atual (vide Figura 5.17), essas aplicações têm também uma instância do CG nelas embutida. Já em um ambiente de rede ativa, pacotes de dados podem também carregar código de invocação de pedidos de adaptação. Em um ambiente de rede de sinalização aberta, protocolos de sinalização e metassinalização específicos são necessários para esse fim.

Experimentos envolvendo a implementação de componentes OpenCOM para as arquiteturas IntServ/ RSVP (Wroclawski, 1997) e DiffServ/ COPS (Chan et al., 2001) no roteador IXP1200 vêm sendo conduzidos como parte do ambiente de testes do *framework* para gerência de adaptações. Os componentes do plano de dados, que povoam as micromáquinas, já são oferecidos pelo NetKit. Os componentes do plano de controle, que residem no StrongARM,

[13] Para não "poluir" a Figura 5.17, somente um desses conjuntos é ilustrado. O outro conjunto é posicionado identicamente, no PC hospedeiro.

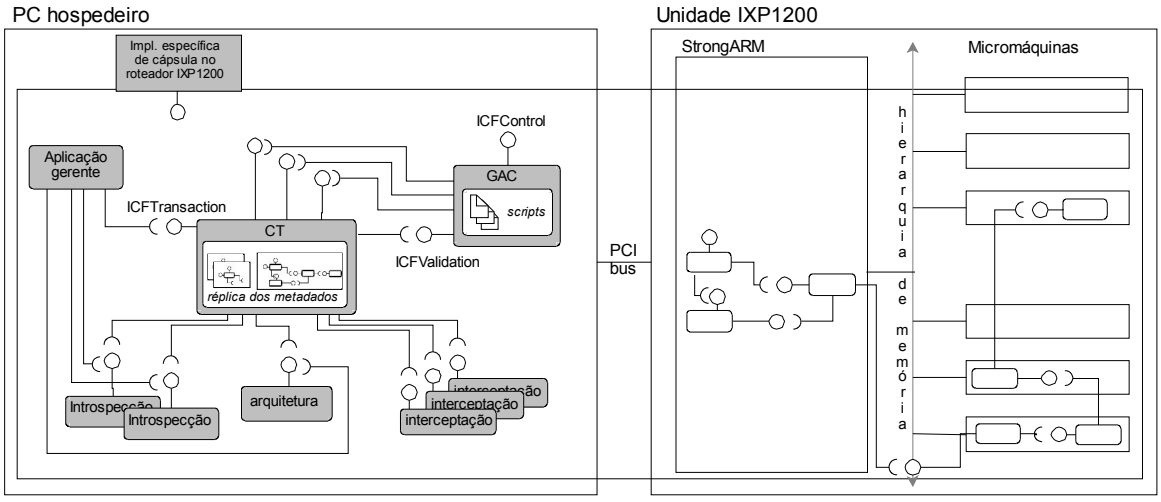

Figura 5.17. Implementação do *framework* para gerência de adaptações no IXP1200.

estão sendo desenvolvidos a partir da recodificação de módulos de software desenvolvidos por Mota (2001) com base no modelo computacional do OpenCOM.

Uma questão importante é o impacto do *framework* para gerência de adaptações no desempenho do roteador IXP1200. Isso reflete, de certa forma, o impacto potencial da abordagem proposta nesta tese no desempenho de sistemas de comunicação. Dois aspectos em particular foram levantados: a demanda de memória (*footprint*) e a sobrecarga computacional no roteador.

A Tabela 5.3 apresenta a demanda mínima de alocação de memória *(footprint)* do GAC, TC e CG. A tabela compara a demanda desses componentes com a de um componente OpenCOM "mínimo" – um componente com uma interface e um receptáculo e cuja única operação provida é uma soma aritmética de dois números inteiros – e com os serviços de cápsula do OpenCOM no PC hospedeiro do roteador IXP1200. Note, a partir da tabela, que os componentes do *framework* para gerência de adaptações incutem no roteador um uso considerável de memória, apesar disso não ser efetivamente um problema nessa implementação pelo fato dos componentes do *framework* estarem residentes no PC hospedeiro. Ademais, cerca de 70% do *footprint* do GAC e do CG se deve à hospedagem de um interpretador Lua nesses componentes. No Capítulo 7 são discutidas outras formas de se representar em tempo de execução restrições descritas em LindaX, de modo a reduzir o *footprint* desses componentes.

Tabela 5.3. *Footprint* de memória dos componentes no OpenCOM.

Componente	*Footprint* (em Kb)
Serviços de cápsula do OpenCOM	132
Componente OpenCOM "mínimo"	22
GAC	134
TC	53
CG	100

Não se tem, nesta tese, uma avaliação quantitativa da sobrecarga computacional que o *framework* para gerência de adaptações impõe ao roteador IXP1200. Contudo, as operações de configuração e de controle de adaptações ocorrem relativamente com pouca freqüência e, sobretudo, fora do *fast path*. O mesmo pode ser afirmado a respeito das operações dos metamodelos de arquitetura e de introspecção do OpenCOM. Assim, presume-se que a execução das operações mencionadas acima não incorre em perda de desempenho computacional do roteador. Já o metamodelo de interceptação demandaria atenção especial, pois um nível de indireção é necessário para permitir que interceptadores seja inseridos em uma associação local entre interfaces e receptáculos. Novamente, porém, interceptadores são usados no *framework* somente sobre as operações dos metamodelos, o que não impacta no desempenho do *fast path* do roteador.

5.6. Sumário

Neste capítulo foram apresentadas as principais ferramentas desenvolvidas dentro do ambiente LindaStudio. Duas ferramentas principais – de refinamento e síntese – mereceram destaque neste capítulo, tendo sido avaliadas em dois estudos de caso: (i) na integração de LindaX com a ADL Wright e (ii) na aplicação de LindaX na plataforma de componentes OpenCOM.

Foi apresentado também neste capítulo um *framework* para gerência de adaptações desenvolvido a fim de simplificar a implementação de processos de síntese para diferentes plataformas de componentes. O *framework* modela genericamente mecanismos de configuração segura (por meio de um serviço transacional) e de controle de adaptações de software. Ele é, em si, independente de LindaX e constitui-se em uma contribuição central desta tese. O principal

aspecto inovador do *framework* para gerência de adaptações é que os mecanismos de configuração e de controle de adaptações instanciados a partir do mesmo são manuteníveis em tempo de execução.

Por fim, este capítulo apresentou a instanciação do *framework* na plataforma de componentes OpenCOM. Nessa plataforma, o *framework* toma a forma de um 'metaCF' a partir do qual diferentes CFs podem ser construídos, aumentando assim a capacidade de reuso de projeto e implementação 'entre' CFs distintos na plataforma OpenCOM.