6
Trabalhos Relacionados

O objetivo deste capítulo é apresentar um conjunto de trabalhos ligados ao tema desta tese, bem como compará-lo ao trabalho ora proposto. Durante a fase de levantamento bibliográfico, muitos outros trabalhos, não apresentados neste capítulo, foram investigados. Procurou-se, contudo, fornecer uma visão daqueles trabalhos a partir dos quais foram tomadas as principais decisões de projeto acerca da abordagem. Além disso, já existe na literatura uma série de *surveys* mais gerais que comparam vários dos trabalhos investigados, mas excluídos aqui, nas diferentes áreas abrangidas por esta tese (Medvidovic e Taylor, 2000; Dietrich e Hubaux, 2002; Möller et al., 2004; Campbell et al, 1999).

Cada seção deste capítulo é dedicada à apresentação de um trabalho específico. As Seções 6.1 e 6.2 apresentam ADLs que permitem a especificação de requisitos não-funcionais. Na Seção 6.3, é apresentado um modelo de componentes – e uma ADL associada – que dá suporte a configurações com compartilhamento de constituintes. As Seções 6.4 e 6.5 dão uma visão geral de *frameworks* de software que tratam de aspectos ligados à gerência de reconfigurações. Na Seção 6.6, outro *framework* de software, focado na autoria formal de serviços, é apresentado. A Seção 6.7 apresenta um ambiente de desenvolvimento de protocolos e serviços em redes ativas. Por fim, na Seção 6.8 é feita uma análise comparativa desses trabalhos com a abordagem ora proposta.

6.1.
Aster (Issarny e Bidan, 1996)

Aster é um ambiente de programação de sistemas distribuídos baseado em configurações. Seu objetivo é dar suporte à síntese sistemática de configurações de plataformas de *middleware* a partir de descrições arquiteturais das aplicações. Ou seja, a partir da descrição arquitetural de uma aplicação, obtém-se uma configuração de plataforma de *middleware* que seja adequada à aplicação.

Aster oferece uma MIL[1] que permite a especificação da arquitetura de software das aplicações e da plataforma de *middleware*. A linguagem Aster descreve a arquitetura de uma aplicação em termos de interconexões de componentes abstratamente definidos por meio de 'interfaces'. Interfaces declaram as assinaturas das operações clientes e servidoras dos componentes. A definição concreta de um componente é provida por meio da ligação de uma interface a uma 'implementação'. A linguagem Aster define também 'hierárquicos' – estruturas que unem conjuntos de elementos arquiteturais (interfaces, implementações ou outros hierárquicos) por meio de associações (*bindings*) entre algumas das interfaces. A Figura 6.1 mostra a forma geral de especificação arquitetural de uma aplicação em Aster.

```
1  INTERFACE nome-interface {
2    CLIENT tipo-ret nome-op-cli( params ) HANDLES exceção;
3    tipo-ret nome-op-serv(params ) RAISES exceção;
4  };
5
6  IMPLEMENTATION  nome-implementação {
7    nome-arquivo IMPLEMENTS nome-interface;
8  };
9
10 HIERARCHICAL nome-hierárquico {
11   CONSTITUENTS nome-implement-1; nome-implement-2;
12
13   BIND nome-implement-1 nome-op-cli :
14         nome-implement-2 nome-op-serv;
15
16   REQUIRES
17     nome-op-serv : requisitos-da-operação;
18 };
```

Figura 6.1. Especificação arquitetural de aplicações em Aster.

No ambiente Aster, a arquitetura de uma plataforma de *middleware* é representada com base em um elemento especial denominado 'barramento de software' (*software bus*). Um barramento pode ser visto como um componente primitivo ao qual os componentes de uma aplicação se ligam para poderem requisitar serviços a outros componentes da aplicação. Um barramento 'abstrato' representa a funcionalidade básica a ser provida por qualquer plataforma de *middleware*. A partir de um barramento abstrato, uma família de barramentos pode ser construída para cada conjunto específico de plataformas de *middleware*, linguagens de programação, requisitos de aplicações e infra-estruturas de comunicação.

A Figura 6.2 apresenta um exemplo de declaração de barramento abstrato, nesse caso o barramento abstrato padrão provido pelo ambiente Aster, chamado de

[1] Segundo definido por Issarny e Bidan (1996).

AsterBus. A cláusula PROVIDES no exemplo da figura define dois tipos de serviços de interação oferecidos pelo barramento: serviços síncronos e assíncronos de chamada remota de procedimentos. O tipo de interação a ser usado entre componentes de uma aplicação que se apóiem em uma plataforma de *middleware* derivada do AsterBus depende dos requisitos impostos pela aplicação. Tais requisitos podem ser expressos na especificação arquitetural da aplicação por meio de cláusulas REQUIRES.

```
1  ABSTRACTBUS INTERFACE AsterBus {
2    void receive ( in pid src, dest; out msg m );
3    void send ( in pid, dest; in msg m );
4
5    PROVIDES
6      PSync = Rpc {( sync:synch )( failure:at-most-once )};
7      PAsync = Rpc {( sync:async )( failure:best-effort )};
8  };
```

Figura 6.2. Especificação do barramento AsterBus.

As cláusulas REQUIRES e PROVIDES constituem ferramentas poderosas na especificação tanto de requisitos não-funcionais impostos pelas aplicações quanto de requisitos não-funcionais providos pelas plataformas de *middleware*. Esses requisitos são definidos por meio de 'árvores de propriedades'. Cada propriedade é especificada formalmente através de predicados em lógica proposicional que a relacionam às interfaces do barramento e dos componentes das aplicações. Novos conjuntos de requisitos podem ser introduzidos estendendo a árvore apropriadamente. A Figura 6.3 exemplifica os conceitos de árvore de propriedades e predicados em relação ao AsterBus.

Obviamente, deve haver plataformas de *middleware* que satisfaçam os requisitos representados na árvore de propriedades. Dependendo do tipo de plataforma de *middleware*, novos requisitos podem ser introduzidos por meio da inclusão de novos componentes "internos" à plataforma (por exemplo, serviços transacionais em CORBA (Object Management Group, 1997)) ou mesmo da alteração de componentes existentes (via reflexividade procedimental, como no OpenORB (Blair et al., 2001)).

Dado que uma aplicação pode ter à sua disposição várias configurações possíveis de plataformas de *middleware*, a escolha de qual configuração é adequada ao serviço requisitado pela aplicação é feita por uma ferramenta de comparação de propriedades, presente no ambiente Aster. Essa ferramenta implementa métodos formais de comparação baseados em lógica proposicional.

Possíveis operações de configuração necessárias para adequar uma plataforma de *middleware* aos requisitos da aplicação também podem ser inferidas e disparadas por essa ferramenta. Esse tipo de checagem é denominado por Issarny e Bidan (1996) de 'checagem vertical' pois o teste de conformidade é feito entre componente da aplicação e barramento. No entanto, Issarny e Bidan ressaltam também a importância de uma 'checagem horizontal', responsável entre outros aspectos pela comparação entre as operações requeridas e oferecidas pelos diferentes componentes de uma aplicação. O tratamento desse tipo de checagem no ambiente Aster é relacionado ao trabalho descrito por Zaremski (1997), que se baseia na especificação formal de componentes em termos de pré-condições e pós-condições.

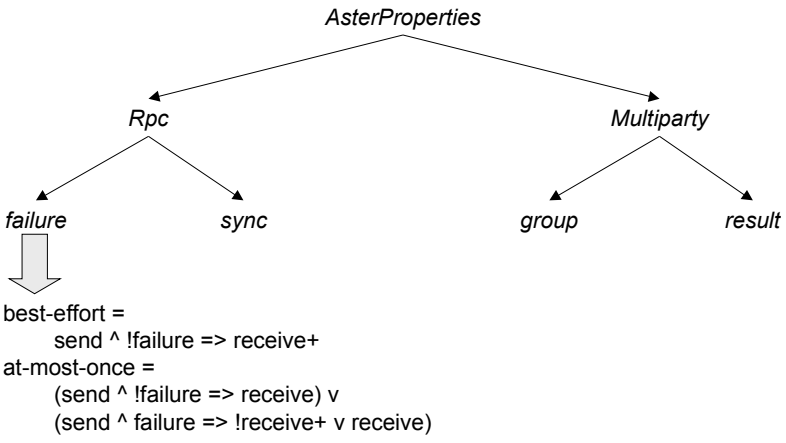

Figura 6.3. Árvore de propriedades no barramento `AsterBus`. Os predicados ilustrados são simplificações daqueles apresentados por Issarny e Bidan (1996).

Em sua concepção inicial, o ambiente Aster prevê somente a síntese de plataformas de *middleware* cujas propriedades sejam definidas em tempo de construção. Ou seja, adaptações na plataforma de *middleware* em tempo de operação não são suportadas. Blair et al. (2000) propõem extensões ao ambiente Aster que permitem esse tipo de adaptação. Nessa abordagem, um 'gerente de adaptações' seria responsável por garantir que todas as modificações na plataforma pudessem ser feitas sem corromper a integridade arquitetural da mesma. Para isso, o gerente de adaptações atuaria em fases distintas: a identificação das condições iniciais de adaptação; a avaliação do impacto da adaptação na plataforma de *middleware*; e a adaptação propriamente dita. O início de uma adaptação ocorre quando as propriedades requisitadas a uma plataforma de

middleware são revistas pelo projetista de uma aplicação (por exemplo, via uma ferramenta de autoria de serviços). Blair et al. propõem então a definição de 'propriedades fracas' de uma aplicação, que têm de ser satisfeitas por qualquer nova configuração a ser proposta. O gerente de adaptações aceitaria alterações contanto que essas propriedades mantivessem-se satisfeitas. Após a verificação das condições iniciais de uma adaptação, a nova configuração da plataforma de *middleware* poderia ser computada através do processo habitual de síntese de configuração do ambiente Aster.[2]

6.2. XelHa (Durán-Limón, 2001)

XelHa é uma ADL própria para a especificação de arquiteturas de sistemas distribuídos de tempo real. Além da definição de componentes, conectores, interfaces e configurações habitualmente encontrados em uma ADL, XelHa engloba também a definição de estruturas associadas à gerência de recursos.

Componentes e conectores em XelHa podem ser primitivos ou compostos. Configurações em XelHa são consideradas como componentes compostos de último nível, chamados de 'sistemas'. Componentes e conectores são tipados, e esses tipos podem ser parametrizados. A especificação de um tipo de componente primitivo inclui somente a cláusula `interfaces` (e, possivelmente, `tasks`, cuja função será vista adiante), que indica os nomes e tipos das interfaces oferecidas por esses elementos. Já a especificação de tipo de conector primitivo inclui, além de interfaces, o estilo de interação oferecido pelo conector (chamada de operação, sinal, ou fluxo). Além disso, conectores distinguem duas categorias de interfaces: 'dados' e 'controle'. No caso de composições, são definidas as cláusulas `components` e `connectors`, que permitem especificar os tipos dos elementos constituintes e as 'cápsulas' onde instâncias desses elementos serão criadas quando a composição for instanciada. Em XelHa uma cápsula identifica, grosso modo, um escopo de gerência de recursos. No caso de conectores distribuídos, as cápsulas envolvidas são passadas como parâmetros do tipo do conector. Com relação a componentes e conectores compostos, a cláusula `interfaces` descreve,

[2] Aqui são assumidas somente alterações decorrentes da intervenção do projetista. Há casos, porém, em que a mudança do ambiente de execução (por exemplo, a extensão de um serviço presente em uma rede fixa também para uma rede móvel) não gera mudanças nos requisitos não-funcionais, mas na configuração interna da plataforma de *middleware*. Blair et al. fazem uma avaliação acerca desse tipo de adaptação no contexto do ambiente Aster.

além do nome e tipo da interface, o elemento interno para o qual ela é mapeada. Por fim, a cláusula `composition_graph` define como os elementos de uma composição são interligados. A Figura 6.4 ilustra um exemplo de especificação – e a representação diagramática correspondente – de um sistema de transmissão de áudio em XelHa.

```
1  Def system AudioSystem:
2    components:
3      connectorServer: ConnectorServer, “capsule 2”
4      source: AudioSrc, “capsule 1”
5      sink: AudioSink, “capsule 2”
6      capsuleProxy: CapsuleProxy, “capsule 2”
7      capsuleMgr2: CapsuleManager, “capsule 2”
8      capsuleMgr1: CapsuleManager, “capsule 1”
9      userInterface: UserInterface, “capsule 2”
10   connectors:
11     operationalConnector:
12        OperationalConnector( “capsule 1”, “capsule 2”)
13   composition graph:
14     interfaces:
15       uiOUT: (userInterface, OUT)
16       connectorServIN: (connectorServer, IN)
17       connectorServCAP: (connectorServer, CAP)
18       capsuleMgr2IN: (capsuleMgr2, IN)
19       connectorServOUT: (connectorServer, OUT)
20       capsuleProxyIN: (capsuleProxy, IN)
21       capsuleProxyOUT: (capsuleProxy, OUT)
22       operationalConnectorIN: (operationalConnector,IN)
23       operationalConnectorOUT: (operationalConnector,OUT)
24       capsuleMgr1IN: (capsuleMgr1, IN)
25     edges:
26       (uiOUT, connectorServIN)
27       (connectorServCAP, capsuleMgr2IN)
28       (connectorServOUT, capsuleProxyIN)
29       (capsuleProxyOUT, operationalConnectorIN)
30       (operationalConnectorOUT, capsuleMgr1IN)
31   tasks:
32     transmitAu
33     ...
34   services:
35     AudioCommService
36     ...
```

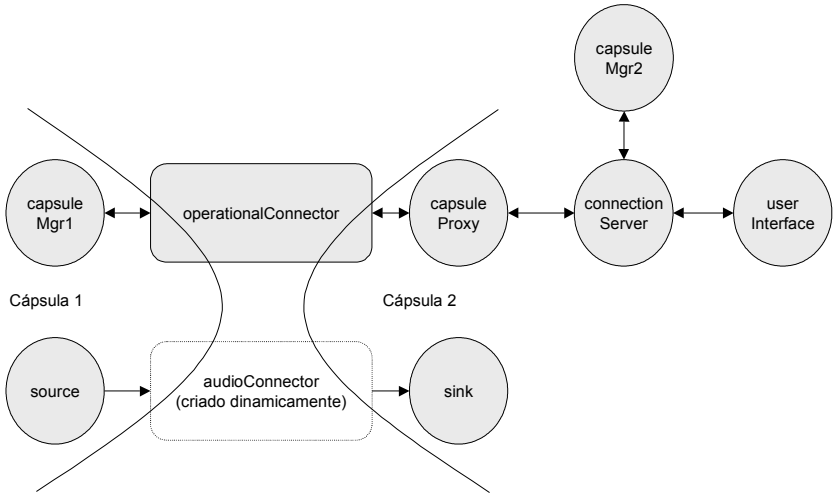

Figura 6.4. Exemplo de especificação de sistema em XelHa.

No exemplo da Figura 6.4, o servidor de conexões de áudio se localiza na cápsula 2 (linha 3) e está ligado ao componente `userInterface` (linhas 9 e 26). Esse componente permite ao usuário final requisitar uma nova conexão de áudio entre os componentes `source` e `sink`, o que disparará a operação `new_connection` no componente `connectionServer` (operação essa definida por uma cláusula `inst` presente na especificação do tipo da interface IN – linha 16). O servidor se conecta também ao componente `capsuleMgr2`, responsável por criar componentes na cápsula 2. Adicionalmente, o servidor de conexões tem acesso ao componente `capsuleMgr1`, residente na cápsula 1, por meio do componente `capsuleProxy`. Esses dois últimos componentes se comunicam via o conector `operationalConnector` (linhas 11 e 12).

Durán-Limón (2001) permite a especificação de mecanismos associados à gerência de recursos em XelHa por meio de estruturas especiais denominadas 'tarefas' e 'serviços'. Cada serviço associa um conjunto de tarefas a um determinado nível de QoS. Tarefas são especificadas conforme o exemplo ilustrado na Figura 6.5.

Cada tarefa pode estar associada, por meio da cláusula `switching_points`, a um ou mais 'pontos de chaveamento', representados pela tripla `{componente,interface,operação}` (linha 3 do exemplo). Tarefas compostas também podem ser especificadas por meio da cláusula `includes`. A cláusula `capsule` identifica a cápsula onde a tarefa será executada. Por sua vez, a cláusula `qos_specifications` permite associar parâmetros de QoS a tarefas. Esses parâmetros são definidos em termos de pontos de chaveamento iniciais e finais.

XelHa define também elementos especiais associados a tarefas chamados 'componentes de gerência', que se responsabilizam pelas funções de orquestração de recursos. A especificação desses componentes é feita na cláusula `qos_management_structure`. Dois componentes de gerência principais são definidos em XelHa: 'monitores' e 'ativadores de estratégias'. A clásula `collector` define as interfaces coletoras de eventos que permitem que determinados componentes do sistema sejam inspecionados por monitores. Monitores verificam a ocorrência de violações na QoS contratada e selecionam uma 'estratégia de orquestração' a ser adotada em face dessas violações. O ativador de estratégia responsabiliza-se então pela aplicação da estratégia selecionada. Monitores são especificados formalmente por meio de autômatos temporizados, identificados na clásula `timed_automaton`. Os ativadores de

estratégia, identificados na cláusula `strategy_activator`, são especificados fora do ambiente XelHa. Finalmente, as ligações entre os componentes de gerência são descritas na cláusula `qos_management_graph`. Note que, no exemplo da figura, o ativador de estratégia possui uma interface ligada a uma interface de controle do conector, por onde a orquestração será efetivada.

```
1  Def task transmitAu.marshall:
2    switching points:
3      srcStub:CTRL:start
4    capsule: “capsule 1”
5    qos specifications:
6      delay(srcStub:IN:read, streamConn:IN:put) = 5
7      throughput(srcStub:OUT:put) = 64
8
9  Def task transmitAu.unmarshall:
10   switching points: ...
11   capsule: “capsule 2”
12   qos specifications: ...
13
14 Def task transmitAu includes transmitAu.marshall,
15                              transmitAu.unmarshall:
16   importance: 5
17   qos specifications:
18     delay(streamConn:IN:put, streamConn:OUT:put) = 10
19     packetloss(streamConn:IN:put, streamConn:OUT:put)= 5
20     delay(srcStub:IN:read, sinkStub:OUT:write) = 20
21     jitter(srcStub:IN:read, sinkStub:OUT:write) = 1
22   qos management structure:
23       interfaces:
24         automatonIN: (automaton, IN)
25         automatonOUT: (automaton, OUT)
26         activatorIN: (activator, IN)
27         activatorOUT: (activator, OUT)
28         streamConnCTRL: (streamConn, CTRL)
29       edges:
30         (COLLECT, automatonIN)
31         (automatonOUT, activatorIN)
32         (activatorOUT, streamConnCTRL)
```

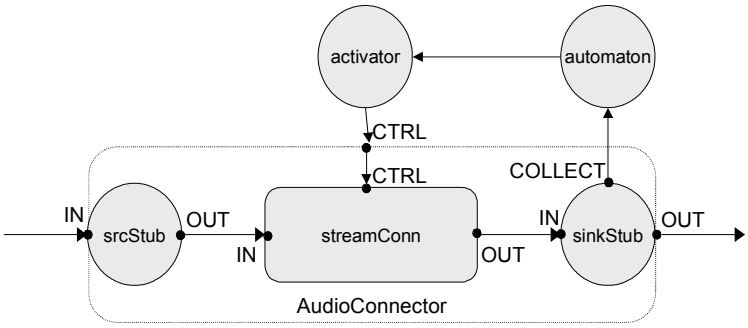

Figura 6.5. Exemplo de especificação de tarefas em XelHa.

Serviços são especificados como o exemplo da Figura 6.6. A especificação de um tipo de serviço pode conter um conjunto de serviços relacionados que se diferenciam pelo nível de QoS oferecido. Esse nível de QoS está relacionado à visão de QoS dos usuários finais de um sistema. As tarefas envolvidas na provisão de um serviço são especificadas por meio da cláusula `tasks`, que as associam a

um número máximo de instâncias de tarefas que podem ser criadas. Conforme ilustrado na Figura 6.4, em uma especificação de sistema as cláusulas `tasks` e `services` permitem identificar, respectivamente, as tarefas executáveis e os serviços oferecidos pelo sistema. Em XelHa, o uso dessas cláusulas é limitado a especificações de sistemas somente.

```
1  Def service type AudioCommService:
2    Def service AudioComm1:
3      user qos: “low”
4      tasks:
5        transmitAu.marshall, 20
6        transmitAu.unmarshall, 20
7    Def service AudioComm2:
8      user qos: “high”
9      tasks:
10       transmitHighAu.marshall, 10
11       transmitHighAu.unmarshall, 10
```

Figura 6.6. Exemplo de especificação de serviço em XelHa.

O processo de análise de uma especificação em XelHa é ilustrado na Figura 6.7, sendo dividido em duas fases. Na primeira fase, os aspectos funcionais descritos pela linguagem são traduzidos em classes de objetos em Python, e os aspectos não-funcionais são traduzidos em RCDL (*Resource Configuration Description Language*), uma linguagem de mais baixo nível utilizada para a especificação de mecanismos de gerência de recursos. RCDL inclui detalhes de implementação do sistema e requisitos específicos da plataforma de *middleware* a ser utilizada. RCDL é, na realidade, um conjunto de sublinguagens que tratam de aspectos não-funcionais distintos, como os tipos de serviços a serem oferecidos pelo sistema, os recursos a serem associados a cada serviço específico, os mecanismos de orquestração envolvidos etc. Na segunda fase do processo de análise, as descrições em RCDL são traduzidas em módulos do sistema real (implementados em Python) responsáveis pela gerência de recursos. A princípio, conforme descrito por Durán-Límon (2001), essa fase prevê somente a tradução das descrições em RCDL para o modelo de componentes oferecido pela plataforma de *middleware* OpenORB.

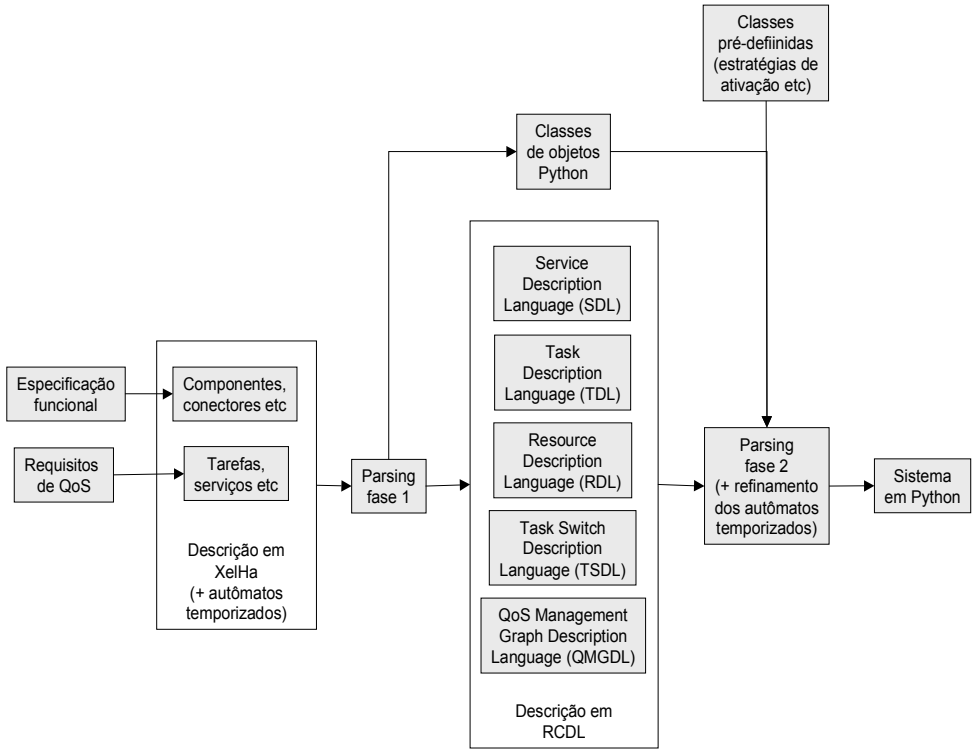

Figura 6.7. Processo de análise de especificações em XelHa.

6.3. Fractal (Bruneton et al., 2002; ObjectWeb, 2004)

Fractal é um projeto que define, dentre outros aspectos, um modelo de componentes extensível e independente de linguagem de programação e uma ADL que permite a descrição de configurações de componentes nesse modelo. Além das abstrações típicas de modelos de componentes, como interfaces e associações, Fractal define também a noção de 'conteúdo' e 'controlador' na especificação da estrutura de um componente. O conteúdo de um componente Fractal são outros componentes, que estão sujeitos ao controle estabelecido pelo controlador do componente que os engloba. O modelo é recursivo, permitindo o aninhamento de conteúdos e componentes em níveis arbitrários – a recursão termina em um componente sem conteúdo, ou, mais especificamente, em um componente cuja única informação mantida pelo modelo são suas interfaces. A Figura 6.8 ilustra a notação utilizada por Bruneton et al. (2002) para representar configurações de componentes Fractal.

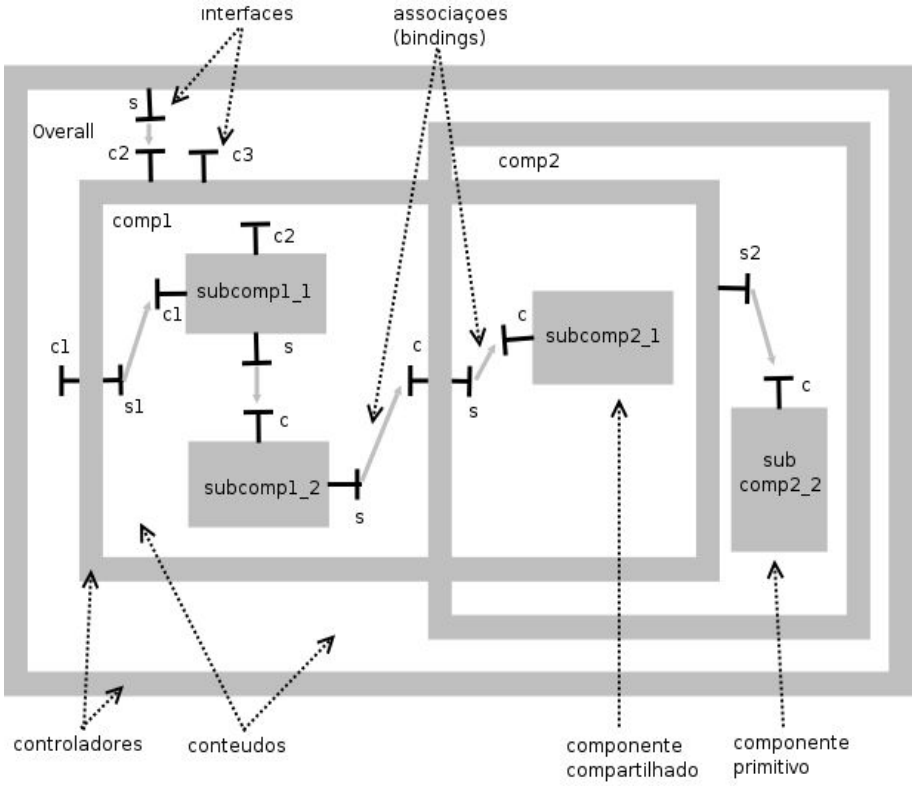

Figura 6.8. Configuração de componentes Fractal.

A principal característica do modelo Fractal é que componentes distintos podem compartilhar um mesmo constituinte em seus conteúdos. Um componente compartilhado por dois ou mais componentes distintos está sujeito ao controle de seus respectivos controladores. O controle resultante de uma configuração desse tipo (concorrência, transação etc) é determinado por um outro componente que engloba todos os componentes participantes nesse compartilhamento. A visibilidade das interfaces dos constituintes de um componente também é estabelecida pelo controlador do componente. Note, portanto, que controladores são abstrações poderosas no que se refere à gerência de aspectos não-funcionais.

A ADL Fractal é uma linguagem baseada em DTDs (*Document Type Definitions*) XML que permite a descrição de configurações de componentes Fractal. Ela é composta de módulos, cada um definindo uma sintaxe abstrata para um determinado elemento arquitetural do modelo – interfaces, associações, atributos e relações de conteúdo. A Figura 6.9 apresenta um exemplo de especificação na ADL Fractal do componente composto ilustrado na Figura 6.8. Note, nas linhas 20 e 27 a 30, que um constituinte compartilhado é declarado normalmente em uma das composições aos quais ele pertence, enquanto nas

outras composições ele é simplesmente referenciado por meio do atributo XML `definition`.

```
1  <component name="Overall">
2    <interface name="s" role="server" signature="..."/>
3    <component name="comp1">
4      <interface name="c1" role="client" signature="..."/>
5      <interface name="c2" role="client" signature="..."/>
6      <interface name="c3" role="client" signature="..."/>
7      <interface name="s1" role="server" signature="..."/>
8      <interface name="s2" role="server" signature="..."/>
9      <component name="subcomp1_1">
10       <interface name="c1" role="client" signature="..."/>
11       <interface name="c2" role="client" signature="..."/>
12       <interface name="s" role="server" signature="..."/>
13       <content class="..."/>
14     </component>
15     <component name="subcomp1_2">
16       <interface name="c" role="client" signature="..."/>
17       <interface name="s" role="server" signature="..."/>
18       <content class="..."/>
19     </component>
20     <component name="subcomp1_3" definition="comp2/subcomp2_1"/>
21     <binding client="this.s1" server="subcomp1_1.c1"/>
22     <binding client="subcomp1_1.s" server="subcomp1_2.c"/>
23   </component>
24   <component name="comp2">
25     <interface name="c" role="client" signature="..."/>
26     <interface name="s" role="server" signature="..."/>
27     <component name="subcomp2_1">
28       <interface name="c1" role="client" signature="..."/>
29       <content class="..."/>
30     </component>
31     <component name="subcomp2_2">
32       <interface name="c" role="client" signature="..."/>
33       <content class="..."/>
34     </component>
35     <binding client="this.s" server="subcomp2_1.c"/>
36   </component>
37   <binding client="this.s1" server="comp1.c2"/>
38   <binding client="comp1/subcomp1_2.s" server="comp2.c"/>
39   <binding client="comp1.s2" server="comp2/subcomp2_2.c"/>
40 </component>
```

Figura 6.9. Exemplo de especificação na ADL Fractal. Os atributos `signature` e `content_class` são preenchidos de acordo com a linguagem de programação em uso.

A ADL Fractal ainda não foi completamente especificada. Em particular, somente aspectos relativos a descrições arquiteturais foram definidos na linguagem. Várias implementações do projeto Fractal têm sido desenvolvidas (ObjectWeb, 2004), visando a aplicação do modelo de componentes e da ADL propostos no projeto em diferentes contextos. Menção especial pode ser feita ao *framework* de software Think (Fassino et al., 2002). Think baseia-se no modelo Fractal para permitir a implementação de núcleos de sistemas operacionais. Componentes Think são módulos codificados em C cuja principal característica é a representação de interfaces de componentes segundo diferentes níveis de otimização e flexibilidade. Essa abordagem muito se assemelha à idéia de

implementação de associações locais específicas por cápsula da plataforma OpenCOM. Como estudo de caso, uma implementação de Think em uma arquitetura de rede ativa é apresentada em (Fassino et al., 2002).

6.4. FORMAware (Moreira, 2003)

FORMAware é um *framework* especializado no desenvolvimento de entidades controladoras de adaptações a partir de descrições semi-formais de estilos arquiteturais. A principal característica dessas entidades é a sua capacidade de gerenciar transações de reconfiguração em plataformas de componentes. Essas entidades permitem enfileirar um certo número de operações de reconfiguração, de modo que essas operações possam ser anuladas antes de sua efetivação, caso elas desrespeitem as regras estabelecidas por um estilo específico.

Moreira (2003) define uma ontologia em FORMAware para modelos de componentes com suporte à reflexividade 'estrutural' – isto é, somente a topologia de ligações entre componentes e a estrutura desses componentes em termos de suas interfaces são consideradas como metadados nesse *framework*. A partir dessa ontologia, Moreira define primitivas de acesso a esses metadados, que podem ser usadas na construção de estilos arquiteturais. Mais especificamente, na codificação de uma entidade controladora de adaptações, essas primitivas são invocadas em uma ordem e com um conjunto de parâmetros tais que representam um estilo como um algoritmo.

Um protótipo do *framework* FORMAware foi implementado em Java. Nessa implementação, é oferecida uma ferramenta encapsuladora (`WrapperGenerator` – vide Figura 6.10) que, a partir de um conjunto de objetos Java e de um arquivo de metadados de configuração, gera componentes que encapsulam esses objetos e uma configuração de aplicação com esses componentes. Essa configuração pode então ser utilizada em conjunto com uma entidade controladora de adaptações representando um determinado estilo, de modo que adaptações nessa configuração possam ser feitas de modo seguro e sujeitas a regras de validação específicas.

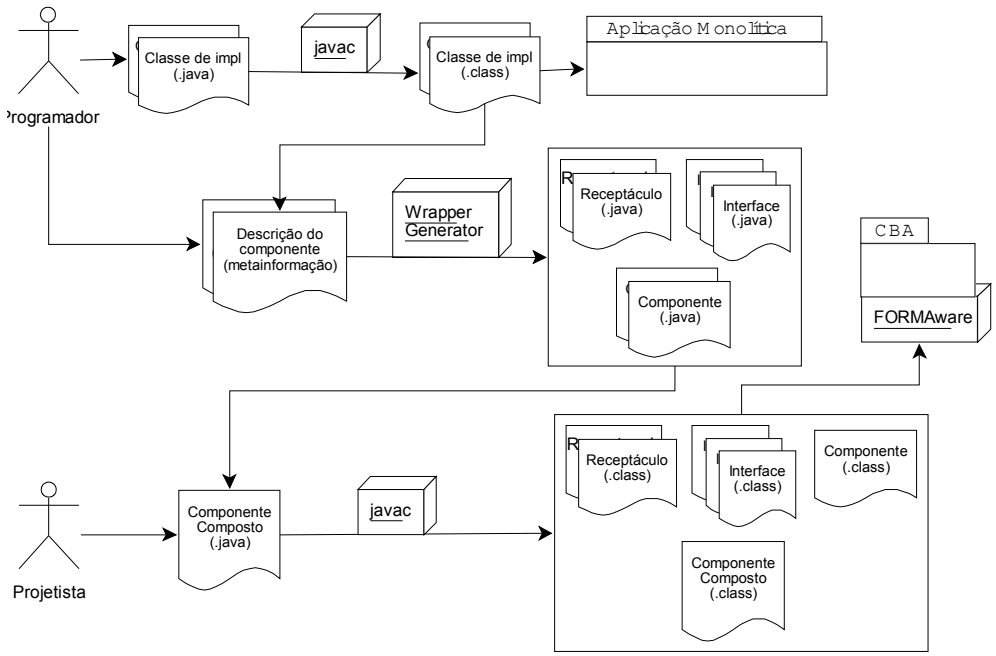

Figura 6.10. A ferramenta `WrapperGenerator` de FORMAware.

6.5. MOTEL (Logean, 1999; Dietrich, 1999)

Logean (1999) propõe um *framework* de monitorização de propriedades, em tempo de execução, de aplicações desenvolvidas sobre qualquer plataforma de *middleware* orientada a objetos. Uma instância desse *framework*, chamada 'MOTEL' (*MOnitoring and TEsting tooL*), é implementada sobre CORBA.

A Figura 6.11 apresenta uma visão global desse *framework*. A especialização do *framework* para uma dada aplicação é iniciada por uma ferramenta que extrai automaticamente um modelo comportamental baseado em eventos de interação entre objetos da aplicação a partir das descrições em IDL das interfaces associadas a esses objetos. Filtros interceptadores de interações podem ser gerados automaticamente a partir dos eventos definidos nesse modelo. Além disso, propriedades de interesse podem ser descritas (passo `(1)` da figura) por meio de uma lógica temporal definida sobre esses eventos (Dietrich, 1999). Essas propriedades são passadas a uma ferramenta que traduz as propriedades (passo `(2)`) em máquinas de estados. Essas máquinas de estados são implantadas em um verificador de propriedades (passo `(3)`) que recebe pedidos de verificação a partir de um gerente de observações (passo `(5)`). Esse gerente recebe notificações de eventos a partir dos filtros interceptadores de interações (passo `(4)`).

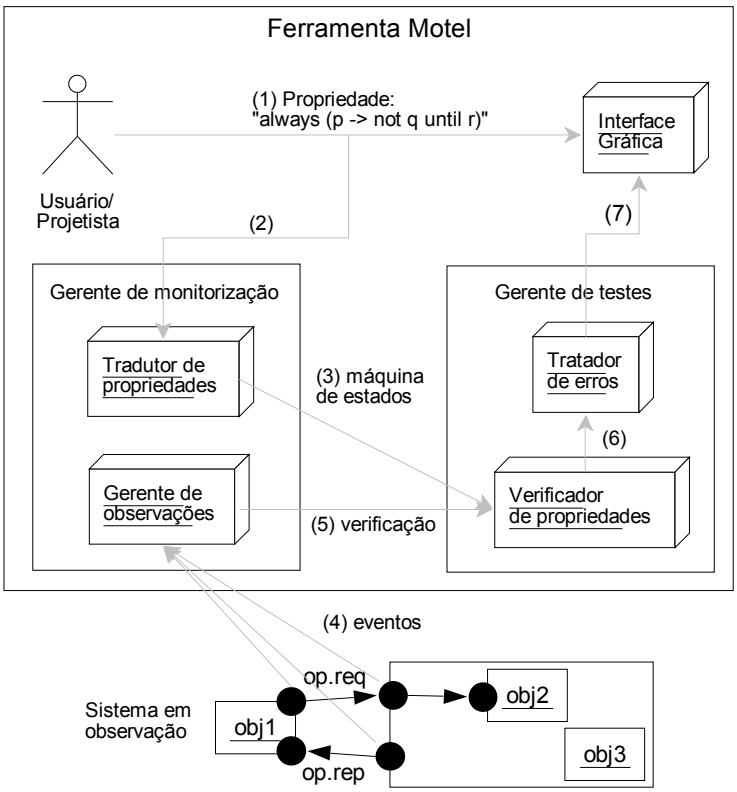

Figura 6.11. *Framework* de monitorização de propriedades da ferramenta MOTEL.

6.6. Arquitetura TOSCA (Kolberg et al., 1999)

A arquitetura TOSCA (*TINA Open Service Creation Architecture*) descreve uma metodologia e um conjunto de ferramentas para a autoria de serviços de telecomunicações sobre a plataforma de *middleware* TINA (Berndt et al., 1994). A principal característica dessa arquitetura é a combinação do uso de *frameworks* com especificações em SDL (Int'l Telecommunications Union, 1992b).

O processo de criação de um serviço em TOSCA começa com a autoria de um serviço por meio de 'paradigmas'. Paradigmas são maneiras de se representar serviços em um nível de abstração tal que permita a um projetista "não-técnico" – um consultor de negócios, por exemplo – criar serviços facilmente. Em geral esses paradigmas envolvem ferramentas gráficas de autoria onde aspectos técnicos não são visíveis ao usuário final. A partir desses paradigmas, '*frameworks* de serviço' pré-definidos por projetistas "técnicos" podem ser especializados. Um *framework* de serviço é representado inicialmente pela especificação, em uma IDL própria de TINA, da interface dos objetos que compõem um serviço, especificação essa

associada a mapas de caso de uso (*Use Case Maps* – UCMs) e descrições textuais semi-formais do comportamento dos objetos descritos. Partes das interfaces descritas nessa IDL não são associadas a comportamento algum, constituindo-se desse modo como *hot spots* do *framework* de serviço. A partir do *framework* de serviço, é gerado um modelo em SDL que o descreve formalmente. O mapeamento dos *hot spots* do *framework* de serviço em SDL é feito por meio do uso de blocos SDL com comportamento nulo (blocos `NULL`).

Com o modelo SDL, verificações formais podem ser feitas acerca do *framework* de serviço associado – por exemplo, a detecção de interações indesejadas entre serviços. O modelo SDL pode ser usado também para gerar casos de teste de serviços antes mesmo que especializações do *framework* de serviço, ocorridas durante a autoria de serviços (por meio de paradigmas), sejam utilizadas em um sistema real. Essas especializações são realizadas e podem ser validadas formalmente por meio, respectivamente, do preenchimento dos *hot spots* do *framework* de serviço e da definição de comportamentos nos blocos `NULL` do modelo SDL associado.

6.7. NetScript (Da Silva et al., 1996)

NetScript é um ambiente de desenvolvimento de protocolos e serviços para redes ativas. O ambiente NetScript pode ser dividido em duas partes: uma plataforma de *middleware* baseada em 'agentes' – elementos computacionais que, ao serem implantados e compostos em um nó de uma rede ativa (chamado aqui simplesmente de 'nó ativo'), permitem o oferecimento de novos protocolos e serviços na rede – e uma linguagem de *script* de alto nível a partir da qual esses agentes podem implantados e compostos sobre a plataforma de *middleware*.

A linguagem NetScript é baseada em um modelo de execução orientado a fluxos de dados. Nesse modelo, uma computação em um nó ativo consiste em: (i) uma coleção de agentes – chamados por Da Silva et al. (1998) de 'caixas' (*boxes*) – que processam fluxos de dados e (ii) recursos alocados no nó necessários a esse processamento. NetScript define 'caixas primitivas', que consistem basicamente em módulos implementados em linguagens de programação convencionais como C e Java. Exemplos típicos de caixas primitivas incluem *drivers* de dispositivos, implementações de protocolos básicos (como IP) etc. Composições de protocolos

e serviços em NetScript são obtidas pela conexão de caixas por meio de 'portas' de entrada e saída. Tais composições podem ser encapsuladas, recursivamente, em 'caixas compostas'.

O que torna a linguagem NetScript adequada à composição de protocolos e serviços em redes ativas é seu suporte a adaptações: caixas podem ser adicionadas, removidas, conectadas e desconectadas em tempo de execução. Para isso, NetScript oferece uma construção, chamada '*template* de caixa', que descreve as assinaturas das portas de entrada e saída de uma caixa bem como a sua estrutura interna. Uma ou mais 'instâncias' de um *template* de caixa podem ser criadas em um mesmo nó ativo, com exceção dos *templates* de caixa que refletem protocolos e serviços tipicamente globais – como IP, DNS etc. – e que, por isso, são normalmente declarados como 'compartilhados', para admitirem no máximo uma instância deles em cada nó ativo.

No ambiente NetScript, a implantação de novos protocolos e serviços em uma rede ativa começa com a definição de um *template* de caixa por um projetista. Esse *template* é posteriomente enviado aos nós ativos por meio dos serviços oferecidos pela plataforma de *middleware* do NetScript. Quando o nó ativo recebe um *template* de caixa, ele cria uma instância do mesmo e a conecta a outras caixas no nó com base nas assinaturas das suas portas. Se o *template* identifica uma caixa composta, a plataforma de *middleware* se responsabiliza por carregar e instanciar cada uma das caixas internas e conectá-las entre si. Se o *template* de uma das caixas internas não existir localmente no nó ativo, a plataforma de *middleware* pode buscar e instalar esse *template* a partir de um 'localizador uniforme de recursos' (*Uniform Resource Locator* – URL) especificado na declaração do *template* da caixa composta.

A Figura 6.12 ilustra o esqueleto de um *template* de caixa para um "novo" protocolo hipotético chamado RTMP (*Real-Time Multicast Protocol*), empilhado sobre o protocolo IP. Esse *template* de caixa é composto de três caixas: uma para o IP, uma para a função de controle de pertinência de grupo e de manutenção de árvores de *multicast* (`RTMPControl`), e uma para a função de (de)multiplexação de PDUs do RTMP para o IP (`IP_RTMPMuxDemux`). A cláusula `connect` define a estrutura interna do *template* de caixa do RTMP

```
1  box template RTMP {
2    inport groupMgtRequests( in Msg req )
3    outport groupMgtReplies ( out Msg rep )
4    box import
5      "http://cs.columbia.edu/ns/boxlb/RTMPControl.nbt" RTMPControl;
6    box import
7      "http://cs.columbia.edu/ns/boxlb/IP_RTMPMux.nbt"  IP_RTMPMux;
8    box import
9      "http://ietf.org/ns/boxlib/IP.nbt"                IP;
10
11   connect {
12     IP.rcvUp -> IP_RTMPMux.rcvDown,
13     IP_RTMPMux.sndDown -> IP.sndUP,
14     IP_RTMPMux.rcvUp -> RTMPControl.rcvDown,
15     RTMPControl.sndDown -> IP_RTMPMux.sndUp
16   }
17 }
```

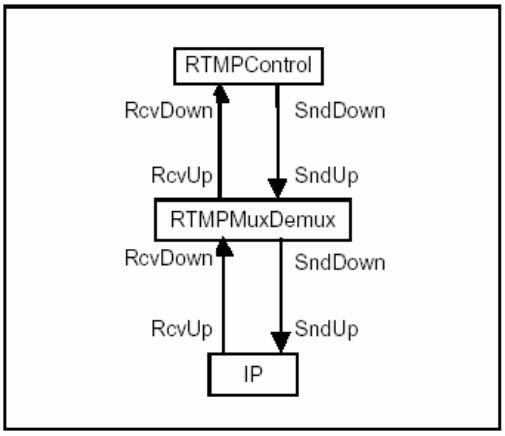

Figura 6.12. Composição de protocolos em NetScript.

6.8. Análise comparativa

Neste capítulo foram apresentados vários trabalhos ligados ao tema desta tese. Em geral, esses trabalhos advogam soluções baseadas em princípios de métodos formais e engenharia de software em diferentes contextos da área de sistemas distribuídos. Essas soluções encontram-se distribuídas ao longo de um espectro que abrange desde especificações de alto nível de serviços até ambientes de configuração e programação especializados.

Dietrich e Hubaux (2002) apresentam um estudo detalhado da aplicação de formalismos na especificação de serviços e levantam a necessidade de integração gradual de linguagens formais no desenvolvimento de sistemas distribuídos em geral, e de telecomunicações em particular. A ferramenta MOTEL, desenvolvida em parte por esses mesmos autores, e a arquitetura TOSCA oferecem abordagens que seguem essa linha. Em contraste com a tese presente, ambos os trabalhos mencionados privilegiam os aspectos comportamentais de um sistema. Graças à

extensibilidade da linguagem LindaX, vislumbra-se que não seja muito complicado combinar outros formalismos à estrutura dessa linguagem (vide considerações no Capítulo 7). Além disso, o refinamento de descrições em LindaX para Wright já dá margem a uma série de verificações de consistência e completeza, embora as simplificações assumidas nesse processo restrinjam o escopo de verificações possíveis.

Abordagens baseadas em ADLs enfocam aspectos estruturais. Porém, poucas ADLs, como as apresentadas neste capítulo, lidam simultaneamente com a integração de formalismos, a síntese automática de sistemas e aspectos de gerência de recursos. Em particular, a estratégia em XelHa de se definir escopos de gerência de recursos de modo relativamente separado de descrições arquiteturais serviu como inspiração para a definição da visão de recursos do modelo sobre o qual a linguagem LindaX se apóia. Porém, ao contrário da abordagem proposta neste trabalho, tanto Aster quanto XelHa não dão suporte a estilos arquiteturais, a configurações com compartilhamento de constituintes ou a atualizações de regras de adaptação em tempo de execução.

Em relação ao suporte a estilos arquiteturais, não se pode deixar de discorrer sobre o *framework* FORMAware. Ao propor a representação explícita de estilos arquiteturais em modelos computacionais, esse *framework* permite que um sistema tenha consciência de sua própria arquitetura interna e das regras de adaptação aos quais está sujeito. Em particular, a concepção do serviço transacional provido pelo *framework* para gerência de adaptações proposto nesta tese baseou-se no serviço equivalente provido pelo FORMAware. Contudo, em contraste com a abordagem proposta nesta tese, estilos FORMAware são especificados e codificados de modo entremeado aos componentes do sistema, resultando em estilos difíceis de serem formalmente especificados dentro do *framework*, além de serem pouco manuteníveis.

Em relação ao compartilhamento de constituintes, vale mencionar também o suporte dado pela ADL Fractal. Contudo, o fato dessa linguagem ainda não ter sido completamente definida impediu uma análise mais qualificada da mesma acerca, por exemplo, de seu suporte a adaptações.

Na linha de ambientes de geração e programação de aplicações em redes programáveis, merece destaque a abordagem do ambiente NetScript. Apesar de altamente especializada para redes ativas, a linguagem de descrição de protocolos e serviços provida por esse ambiente não lida com adaptações dinâmicas;

adaptações são obtidas através da modificação e resubmissão de descrições (caixas e *templates* de caixa). Isso contrasta com o suporte ao planejamento de adaptações bem controladas provido como parte da abordagem proposta nesta tese. Ainda nessa linha, dois outros trabalhos, embora não descritos neste capítulo, merecem uma atenção especial. Batory e Geraci (1997) propõem uma ferramenta que permite a síntese de código a partir de descrições em uma DSL. A abordagem adotada nesta tese é menos "granular", na medida em que uma biblioteca de componentes pré-existente (o NetKit) é usada e os únicos trechos de código que são realmente sintetizados são os *scripts* de configuração e de verificação em Lua. Reid et al. (2000) propõem uma linguagem de configuração que permite a síntese de sistemas a partir de módulos pré-existentes codificados em C. A abordagem proposta no presente trabalho é mais genérica na medida em que não pressupõe linguagem de programação alguma.