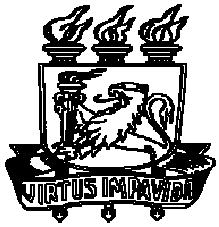


Universidade Federal de Pernambuco
Centro de Informática
Pós-Graduação em Ciência da Computação

Cidcley Teixeira de Souza

Arquitetura de Software e Estilos Arquiteturais Distribuídos

Especificação, validação, análise e implementação

TESE DE DOUTORADO

Recife, Setembro de 2003

Universidade Federal de Pernambuco
Centro de Informática
Pós-Graduação em Ciência da Computação

Arquitetura de Software e Estilos Arquiteturais Distribuídos

Especificação, validação, análise e implementação

Cidcley Teixeira de Souza

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Paulo R. F. Cunha

Recife, Setembro de 2003

Trabalho intelectual é uma palavra errada. Não é "trabalho", é prazer, dissipação, nossa melhor recompensa.

Mark Twain

Agradecimentos

Ao longo desses anos na UFPe pude conhecer diversas pessoas e situações que, de alguma forma, influenciaram meu trabalho. Principalmente não estando em minha cidade de origem e deixando minha família a distância, pude vivenciar diversos tipos de sentimentos que por vezes vieram a contribuir e algumas outras vezes atrapalharam o andamento desse trabalho. Mas agora é o momento para agradecer a todos que tiveram participação direta ou indireta na elaboração dessa tese.

Inicialmente gostaria de agradecer a meu orientador. O Prof. Paulo Cunha, com seu estilo perfeccionista e com uma visão periférica extremamente aguçada, permitiu que eu parasse várias vezes e repensasse minhas idéias. Contudo, uma das suas melhores características foi a liberdade que ele me ofereceu ao longo desses anos. Liberdade por vezes vigiada, mas com toda a prudência de quem realmente estava sabendo o que faz.

Também agradeço ao colega Luís Carlos, que me acolheu em sua casa por todo esse tempo. Eu também tive méritos em conviver com seu "jeitão esquisito", mas ao longo dos anos eu observei que você é realmente uma pessoa muito bacana.

Agradeço aos professores que participaram das minhas bancas de proposta e de tese e avaliaram esse trabalho. De fato, mesmo tendo sido um trabalho literalmente grande, ainda assim recebi comentários minunciosos e extremamente pertinentes.

Agradeço aos meus professores na UFPe, Prof. Carlos Ferraz, Prof. Alexandre Vasconcelos, Prof. Djamel Sadok, Profa. Ana Lúcia, entre outros, pelo belo exemplo de dedicação demonstrado dentro e fora da sala de aula.

O Prof. Augusto Sampaio, coordenador da pós-graduação nos meus últimos anos na UFPe, também deve ser lembrado. Sua competência e zelo pela pós-graduação, materializadas nas mudanças introduzidas em sua gestão, são, no mínimo, dignos de respeito.

Por fim, mas não menos importante, gostaria de agradecer à minha família. A meus pais por todo o esforço para que eu chegasse até aqui. A meu irmão que revisou muitos dos meus artigos em inglês. E, um agradecimento muito especial à minha esposa Fátima. Realmente não tenho palavras para agradecer sua dedicação, confiança e, principalmente, seu apoio. Tenho certeza que eu não teria conseguido sem você.

Resumo

Ao longo da última década, a demanda por técnicas e tecnologias para o desenvolvimento de aplicações distribuídas tem chamado a atenção tanto da indústria de software como da comunidade científica. Nesse sentido, pelo fato dessas aplicações serem complexas por natureza, a estrutura do software como um todo - ou sua arquitetura - tem se apresentado como o problema central no projeto dessas aplicações.

Contudo, mesmo com todos os avanços realizados nessa área, ainda assim as técnicas atuais se mostram ineficazes para viabilizar a produção de software distribuído de qualidade tendo como foco principal o projeto arquitetural.

Pelo lado da indústria de software, vemos a utilização ostensiva de infraestrutura de middleware, onde podemos destacar a arquitetura CORBA, que permite a construção de aplicações distribuídas complexas utilizando diversos serviços padronizados. Entretanto, aspectos arquiteturais não são considerados no desenvolvimento de aplicações CORBA, o que é agravado pelo fato de CORBA não ser plenamente conhecido da grande maioria dos desenvolvedores de software.

Já pelo lado da comunidade de Engenharia de Software, observamos ao longo da última década a proliferação de Linguagens de Descrição de Arquitetura (*Architecture Description Languages* - ADLs) que dão suporte à especificação de arquiteturas de software complexas. Contudo, várias são as limitações dessas ADLs: frequentemente possuem sintaxes complexas; normalmente não são utilizadas para modelar problemas cotidianos; não implementam na sua maioria conceitos de estilos arquiteturais; quase nunca apresentam suporte para a concretização das especificações, entre outros fatores.

Para dificultar ainda mais, várias pesquisas recentes demonstram que as ADLs disponíveis atualmente não fornecem as características desejáveis para a especificação de aplicações que devam ser implementadas sobre infraestruturas de middleware.

Nessa tese, mostraremos que é possível construir aplicações distribuídas complexas baseadas em infraestruturas de middleware utilizando a noção de arquitetura de software e estilos arquiteturais, partindo de especificações arquiteturais produzidas a partir de conceitos e ferramentas que fazem parte da vida cotidiana dos desenvolvedores de software, tendo esses apenas que ter noções básicas de arquitetura de software e estilos arquiteturais.

Para realizar essa tarefa, apresentamos o *framework* DraX (*DistRibuted Architecture based on XML*), que é um conjunto de linguagens, esquemas, e programas que possibilitam a realização de especificação, validação, análise e implementação de arquiteturas de software e estilos arquiteturais distribuídos utilizando tecnologias baseadas em XML, UML e Java e que permitem a geração de aplicações que podem ser executadas sobre infraestruturas de middleware como CORBA e RMI. Além disso, fornece um ambiente automatizado para o suporte ao ensino de arquitetura de software, indo das descrições estruturais de arquiteturas e estilos até à realização de análises em modelos formais construídos em álgebra de processos.

Abstract

For sometime the demand for development techniques and technologies of distributed applications has received a lot of attention by both the software industry and the scientific community. In this sense, since such applications are by nature complex, the software structure as a whole - or its architecture - can be considered the main problem in the project of such applications.

However, even when we consider the advances in this field, the current techniques have been shown to be inefficient to enable the development of quality distributed software when the main focus is the architectural design.

On the software industry side, one can notice the massive use of middleware infrastructures, where the CORBA architecture stands out, which enables the development of complex distributed applications with the use of several standard services. However, architectural aspects are not considered in CORBA development, which is worsen by the fact that the CORBA API is not friendly to the majority of software developers.

On the Software Engineering community point of view we can see the proliferation, in the last decade, of Architecture Description Languages (ADLs) that support the specification of complex software architectures. However, there is a number of limitations on these ADLs: they usually have a complex syntax; normally they are not used to model sample problems; the concepts of architectural styles are not present in most ADLs; there are few tools to provide support to realize the specifications, among other factors.

To make things even more difficult, recent researches have shown that the available ADLs do not have the ideal characteristics to the specification of applications that must be implemented under middleware infrastructures.

In this thesis, we show that it is possible to build complex middleware based distributed applications by adopting the notion of software architecture and architectural styles, starting from architectural specifications developed from concepts and tools that are widely known by software developers, who need only to have basic knowledge about software architecture and architectural styles.

In order to archive this, we present framework DraX (*DistRibuted Architecture based on XML*), a set of specifications, schemas, scripts and programs that enable the realization of the specification, validation, analysis and implementation of distributed software architectures and architectural styles by using technologies based on XML, UML and Java, which allow the generation of applications that can be executed over middleware systems such as CORBA and RMI. In addition, it provides an automated environment that supports the teaching of software architecture, from the structural description of architectures and styles to the realization of analyses in formal models built in processes algebra.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Lista de Abreviaturas

ADL	Architecture Description Language
CORBA	Commom Object Request Broker Architecture
CCM	CORBA Component Model
CSP	Communicating Sequencial Processes
CSS	Cascading Stylesheets
DCOM	Distributed Component Object Model
DDP	Diagrama de Descrição de Protocolo
DOM	Document Object Model
DTD	Document Type Definition
HTML	HyperText Markup Language
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
MDA	Model Driven Architecture
MSMQ	Microsoft Message Queuing
MWB	Mobility WorkBench
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

Capítulo 1

Introdução

1.1 Motivação

A indústria de software já há muito tempo tem se defrontado com grandes desafios, dentre os quais podem ser destacados o aumento do tamanho e complexidade dos sistemas, além da constante busca pela redução de esforços, custos, tempo de desenvolvimento e manutenção do software. Neste contexto, o reuso de software [59] é reconhecido como uma necessidade, principalmente nas fases iniciais do projeto, onde esses benefícios podem ser alcançados de forma mais expressiva.

Diante deste cenário, a arquitetura de software [90, 102, 9] tem se mostrado uma disciplina realmente útil por desempenhar um papel importante no processo de desenvolvimento do software, separando os elementos de computação (componentes) dos mecanismos de comunicação (conectores). Esta separação permite e facilita a promoção do reuso em níveis abstratos, que é um dos principais benefícios fornecidos pelo desenvolvimento arquitetural. Entretanto, para que esses benefícios da arquitetura de software sejam realmente alcançados, ela deve ser tratada explicitamente servindo como base para análise, projeto e implementação.

1.1.1 Desenvolvimento Baseado em Arquitetura

No processo de desenvolvimento baseado em arquitetura, a arquitetura de software desempenha um papel importante durante a conversão dos requisitos para a implementação do sistema [45, 78], efetuando a ligação entre esses dois níveis de abstração. Neste cenário, as pesquisas atualmente têm se preocupado basicamente em tratar as etapas requisitos-arquitetura [33, 57] e arquitetura-implementação [8, 9].

Como mencionado anteriormente, se o reuso for aplicado em níveis bem abstratos, maiores são os benefícios obtidos. Neste contexto, a arquitetura de software tem seu papel reconhecido, pois ela promove o reuso de elementos de projeto. Entretanto, para que uma organização alcance este nível de reuso, torna-se necessário que ela repense seus processos de engenharia de sistemas, a fim de permitir aos desenvolvedores a identificação das oportunidades de reuso em fases iniciais do processo de desenvolvimento. Por esta razão, a arquitetura de software deve ser focada, projetada e realizada explicitamente, para que seus benefícios possam de fato ser alcançados.

Esses aspectos não são tratados no desenvolvimento tradicional, onde normalmente a estrutura do software não é explicitamente manipulada. Além disso, a adoção de estilos arquiteturais [38] também contribui para a elaboração de especificações reusáveis em contextos específicos.

1.1.1.1 Estilos Arquiteturais

Um estilo arquitetural [41, 102] define um vocabulário de elementos de projeto e um conjunto de restrições sobre como esses elementos podem ser combinados. Ele permite o reuso de organizações arquiteturais estabelecidas para resolver um determinado problema recorrente. Existem numerosos estilos arquiteturais, como por exemplo: cliente-servidor, *pipeline*, entre outros.

Estilos arquiteturais são importantes na etapa de especificação da arquitetura, mas para que a arquitetura seja a base do projeto ela deve ser comunicada claramente para todos os participantes do processo de desenvolvimento. Nesse contexto, destacamos as ADLs (*Architecture Description Languages*) para a realização dessa tarefa.

1.1.1.2 Linguagens de Descrição de Arquitetura

Linguagens de Descrição de Arquitetura (ADLs) são notações semi-formais usadas na representação e análise de arquiteturas [40]. Com ênfase na estrutura de alto nível da aplicação em vez de se deter a detalhes de implementação [80]. ADLs têm se tornado recentemente uma área de pesquisa intensa na comunidade de arquitetura de software. Diversas ADLs têm sido propostas para modelagem de arquiteturas, tanto para domínio específico como para propósito geral. As ADLs mais comuns são: ACME, Aesop, C2, Darwin, MetaH, Rapide, SADL, UniCon, Wright, CL.

1.1.2 Infraestruturas de Middleware

Em termos pragmáticos, atualmente o desenvolvimento de aplicações distribuídas se dá pela adoção de infraestruturas de middleware que forneçam serviços que permitam o desenvolvimento de aplicações complexas sem a preocupação com os fatores de distribuição em si. Arquiteturas como DCOM [83], RMI [60], CORBA [51], MSMQ [84], entre outras, têm fornecido mecanismos realmente poderosos para a criação de aplicações distribuídas complexas. É válido ressaltar que CORBA é um padrão amplamente aceito atualmente no que tange a disponibilização de serviços para aplicações distribuídas, como transações, eventos, persistência, entre outros. Contudo, mesmo fornecendo uma API independente de linguagem para acessar seus serviços, a utilização de CORBA na implementação de aplicações que fazem uso desses serviços não é uma tarefa trivial, exigindo do desenvolvedor um forte embasamento conceitual inicial para que as tarefas de implementação venham a ser de fato produtivas [97]. Esse problema é exacerbado quando diversos serviços têm que ser utilizados simultaneamente, fato que é relativamente normal para a demanda das aplicações atuais. Dessa forma, a maioria das aplicações desenvolvidas em CORBA usam essa arquitetura apenas como um mecanismo de comunicação [95]. De fato, a manipulação das interações entre objetos em aplicações complexas e a implementação dessas interações utilizando os serviços oferecidos por CORBA é realmente uma tarefa que desestimula até mesmo o melhor dos programadores [97]. Esses fatos também se aplicam a outros infraestruturas de middleware disponíveis atualmente.

1.1.3 Implementação de Arquiteturas de Software Sobre Middleware

Tendo observado sobre a importância da adoção do desenvolvimento arquitetural para a construção de aplicações distribuídas e tendo como fato concreto à utilização de infraestruturas de middleware como principal forma de implementação dessas aplicações, dois questionamentos básicos surgiram como motivação para o desenvolvimento dessa tese: os desenvolvedores de

software distribuído estão realmente dispostos a adotar os conceitos e ferramentas propostas pelos pesquisadores em arquitetura de software na consecução de seus projetos ? As ADLs atuais são realmente capazes de produzir especificações de fácil implementação sobre tecnologias de middleware ?

Pudemos observar durante a pesquisa bibliográfica para essa tese que a indústria de software se mostra realmente interessada em adotar as idéias de arquitetura de software, mas não de utilizar as ferramentas atualmente propostas pela academia. Pudemos concluir isso através de diversos esforços que vêm sendo desenvolvidos para tentar capturar as idéias das ADLs nas ferramentas que fazem parte do cotidiano dos desenvolvedores de software. Tecnologias como UML [15] e XML [16] vêm sendo exploradas como elo entre os conceitos de arquitetura de software e a prática de desenvolvimento. Contudo, essas ferramentas ainda não atingiram realmente o potencial de uma ADL [29], pois com elas não podemos especificar arquiteturas complexas, validar formalmente ou verificar a consistência de estilos arquiteturais.

Com relação à segunda pergunta, que versa sobre a aplicabilidade das ADLs atuais como suporte à implementação de aplicações baseadas em infraestruturas de middleware, pudemos concluir que isso realmente não ocorre. Baseamos nossas conclusões verificando as características das mais importantes ADLs. Facilmente podemos observar que a maioria delas foi desenvolvida para dar suporte apenas à descrição de arquiteturas e verificação formal das mesmas. E poucas tiveram o interesse em fornecer mecanismos onde as especificações pudessem ser concretizadas. Só para exemplificar, a ADL Wright [7] fornece mecanismos de descrição formal de sistemas, Darwin [72] descreve arquiteturas distribuídas mas não fornece mecanismos de especificação de estilos, UniCon [101] fornece mecanismos de implementação apenas para alguns estilos pré-definidos. Contudo, a importância de se tratar aspectos de concretização de especificações arquiteturais pode ser comprovada em trabalhos como [27]. Também encontramos em [36] a argumentação mais forte com relação à motivação dessa tese. Nesse artigo, é deixado claro a necessidade de se produzir ADLs que permitam a utilização de conceitos arquiteturais no desenvolvimento de aplicações distribuídas. Além de serem mostradas as limitações das abordagens existentes para este fim.

Um outro problema que observamos, e estamos utilizando como meta nessa tese, é relativo ao potencial das ADLs atuais para a especificação de estilos arquiteturais. Pudemos concluir ao longo desse trabalho que as ADLs atuais não fornecem mecanismos específicos para a descrição de estilos arquiteturais. De fato, o que se trata como especificação de estilos arquiteturais na maioria das ADLs que dão suporte a esse conceito, é a criação de especificações arquiteturais genéricas que podem ser utilizadas para gerar arquiteturas específicas [41, 29], como é o caso de Wright [7] e ACME [44]. Uma ADL que mais se aproxima dessa idéia é Aesop [41], que é um *framework* que permite o desenvolvimento de arquiteturas de software com estilos específicos. Contudo, os estilos de Aesop são definidos na própria linguagem e o desenvolvedor não tem a flexibilidade de combinar estilos ou criar novos estilos. Um outro fato que corrobora com a idéia de se ter uma linguagem específica para a definição de estilos, principalmente para dar suporte à aplicações distribuídas, é a necessidade de uma melhor separação de interesses [56]. Queremos dizer com isso que, ao se ter uma linguagem específica para a definição de estilos arquiteturais podemos permitir que esses sejam desenvolvidos de forma independente de uma arquitetura específica e que as arquiteturas apenas devem referenciar qual especificação de estilo seguir para se beneficiar desse conceito. Nesse caso, podemos ter equipes que se preocupem especificamente em criar estilos específicos ou refinar estilos já existentes.

Uma outra restrição encontrada nas ADLs atualmente é a que diz respeito à geração de código a partir das especificações arquiteturais. Poucas são as que tratam do assunto, uma

exceção é UniCon [101], que apresenta implementações concretas para diversos conectores e gera especificações utilizando arquivos externos de implementação.

Contudo, mesmo os poucos exemplos de ADLs que fornecem algum tipo de mecanismo que permita a implementação de aplicações baseadas nas descrições de arquitetura e de estilos, essas possuem restrições em virtude da especificação de conectores dessas linguagens [29]. Mesmo tendo sido bastante trabalhada a importância da adoção desses elementos [79], a utilização de conectores nas ADLs foi extremamente influenciada pelos mecanismos clássicos de comunicação, como *sockets*, *RPCs* e *pipes*. O que pode ser observado é que os ORBs não são comumente tratados como formas de concretização de conectores para implementação de aplicações distribuídas [29]. De fato, quando a idéia de um *bus* de dados é utilizada na construção de conectores em ADLs, como em C2 [75], esses são utilizados como um outro componente na comunicação, gerando mais um nível de indireção e induzindo a modificação do estilo de comunicação das ADLs que usam esses conectores.

Em [29] foi realizado um estudo aprofundado sobre a indução de estilos na utilização de infraestruturas de middleware. Nesse estudo foi concluído que as ADLs atuais carecem de características que permitam a implementação de suas especificações e que as infraestraturas de middleware ainda não são compatíveis com as especificações de arquiteturas desenvolvidas por ADLs.

Podemos resumir essa discussão inicial em quatro pontos principais:

1. Não é uma tarefa simples implementar aplicações distribuídas complexas de forma a utilizarmos as potencialidades de infraestruturas de middleware como CORBA ou RMI;

2. A utilização de projetos arquiteturais [81] traz diversos benefícios ao desenvolvimento dessas aplicações distribuídas;

3. As ADLs atuais, além de não fornecerem mecanismos que permitam a implementação de suas especificações utilizando middleware, possuem características que não são atrativas para os desenvolvedores de aplicações distribuídas que utilizam esse tipo de software; e

4. Tanto o mercado como a academia estão realmente interessados em utilizar as facilidades das arquiteturas de software e estilos arquiteturais no desenvolvimento de aplicações e, particularmente as aplicações distribuídas podem se beneficiar bastante com essa abordagem.

Nessa tese iremos mostrar que podemos especificar, validar e analisar projetos arquiteturais complexos tanto de forma sintática e estrutural como comportamental, de modo a produzir especificações de fácil implementação sobre infraestruturas de middleware utilizando apenas tecnologias e ferramentas que fazem parte do cotidiano dos desenvolvedores.

1.2 Nosso Tratamento

Antes de apresentar nossa abordagem para demonstrar a tese sugerida, temos que, inicialmente, definir os objetivos a serem considerados. Dessa forma, definimos três objetivos básicos que devem ser alcançados como resultado dessa tese:

1. Contribuir para o estado da arte em arquitetura de software com a intenção de facilitar a utilização dos conceitos de arquitetura de software e estilos arquiteturais através da

apresentação de novas abordagens para essa disciplina e pela criação de ferramentas de fácil utilização;

2. Desenvolver linguagens, *scripts* e outros mecanismos que venham a ser aplicados no ensino dos conceitos de arquitetura de software e estilos arquiteturais; e

3. Mostrar a aplicabilidade dos conceitos de arquitetura de software e estilos arquiteturais no desenvolvimento de aplicações distribuídas sobre infraestruturas de middleware de modo a, tanto tornar mais simples a criação dessas aplicações utilizando a enorme gama de recursos dessas tecnologias, como também se beneficiar do potencial formal das ADLs.

Para atingir esses objetivos, propomos nessa tese um conjunto de ferramentas baseadas em tecnologias de larga aceitação no mercado e na academia para a especificação, verificação, análise e implementação de arquiteturas de software e de estilos arquiteturais distribuídos. Mostraremos que essas ferramentas podem realmente capturar os conceitos clássicos de arquiteturas de software e estilos arquiteturais e podem fornecer meios para que as especificações criadas possam gerar implementações baseadas em infraestruturas de middleware onde todo o potencial dessas plataformas possam ser utilizados. Alem disso, essas ferramentas poderão ser aplicadas como mecanismos de ensino dos conceitos abstratos de arquiteturas de software e de estilos arquiteturais.

Desse modo, apresentaremos nessa tese um *framework* denominao DraX (*DistRibuted Architecture based on XML*), que conta com linguagens para a descrição de arquiteturas de software e estilos arquiteturais, tanto sintática como comportamentalmente, além de um conjunto de *scritps* e esquemas de validação estrutural e comportamental de arquiteturas e de *scritps* de geração de código sobre infraestruturas de middleware baseado nas informações arquiteturais e de estilos. Nesse *framework*, todas as informações sintáticas são representadas em XML e os fatores relativos ao comportamento são descritos utilizando uma variação dos diagramas de estados UML, que podem ser representados utilizando XMI (*XML Metadata Interchange*) [52].

Na verificação formal, utilizamos XML Schemas[25] para verificar a consistência das especificações dos componentes, das arquiteturas e dos estilos e esquemas utilizando a linguagem Schematron [61] para verificar consistências tanto na descrição dos componentes, arquiteturas e estilos como também com relação a aderência[1] das especificações aos estilos arquiteturais. Para verificar as consistências comportamentais entre os componentes que formam uma arquitetura, produzimos *scripts* de mapeamento da representação XMI dos diagramas UML para π-cálculo, sendo que essas especificações geradas são verificadas utilizando o MWB (*Mobility WorkBench*) [120]. Além disso, desenvolvemos uma nova álgebra de processos derivada de π-cálculo que possui uma sintaxe mais apropriada para representar as estruturas de interconexões arquiteturais com a qual podemos realizar uma verificação mais completa de especificações de arquitetura.

Na geração de códigos utilizamos XSLT [18] para a criação de *templates* de classe em linguagem Java a partir das informações das especificações arquiteturais e de estilos. Nesses *templates* estão embutidas as informações necessárias para que os códigos gerados possam ser executados sobre um middleware respeitando as especificações de arquitetura e estilos.

[1]Por aderência definimos a validade de uma especificação com relação ao estilo que essa deve seguir.

1.3 Estrutura da Tese

De forma a contemplar todos os aspectos de desenvolvimento com as ferramentas de DraX podemos dividir logicamente essa tese em quatro partes, sendo que cada parte é formada por um ou mais capítulos, cada um tratando de um aspecto específico do desenvolvimento com DraX. Na Figura 1.1 apresentamos a estrutura de capítulos da tese e sua relação com as etapas de desenvolvimento com DraX. Como podemos observar, temos as fases de Especificação, onde as arquiteturas e os estilos arquiteturais são descritos. A fase de Validação e Análise, onde essas especificações são devidamente avaliadas e a fase de Implementação, onde os códigos são gerados e a implementação da lógica da aplicação é inserida.

No Capítulo 2, apresentamos algumas das tecnologias utilizadas em DraX. Nele tratamos dos conceitos de arquitetura de software e estilos arquiteturais, middleware, XML e π-cálculo. No Capítulo 3 discutimos alguns trabalhos relacionados. No Capítulo 4 descrevemos o *framework* DraX, sendo que cada uma de suas ferramentas são descritas no contexto de uma metodologia de utilização dessas ferramentas no desenvolvimento de aplicações com DraX. O objetivo principal desse capítulo é fornecer uma visão superficial de DraX e de sua aplicabilidade, sendo que nos capítulos posteriores essas etapas são detalhadas.

A etapa de Especificação de DraX é apresentada no Capítulo 5. Na Seção 5.2 descrevemos os requisitos das linguagens de DraX através de uma linguagem de padrões. Na Seção 5.3 apresentamos a linguagem ArchML, que é a ADL de DraX. Já na Seção 5.4 apresentamos a linguagem Xtyle, que é a linguagem de descrição de estilos de DraX. Nessas seções são apresentadas tanto a sintaxe dessas linguagens como a forma de realizar a descrição de comportamento de arquiteturas e estilos arquiteturais.

Já a etapa de Verificação e Análise é apresentada no Capítulo 6. Sendo que na Seção 6.2 é tratado o problema de validação sintática, onde tanto as arquiteturas como os estilos são verificados quanto a sua sintaxe e estrutura como também quanto a seus relacionamentos. Na Seção 6.3 é realizada a verificação comportamental. Nela as arquiteturas são avaliadas quanto à consistência comportamental de componentes que estão conectados formando uma arquitetura, como quanto à aderência comportamental de uma arquitetura em relação a seu estilo.

A fase de implementação é tratada no Capítulo 7. Nele alguns *scripts* de geração de código Java para ser executado sobre um middleware são apresentados. Sendo que em seguida, no Capítulo 8 um estudo de caso é apresentado de forma a exemplificar e validar a aplicação de todos as ferramentas de DraX.

Por fim, no Capítulo 9, apresentamos as conclusões desse trabalho. Apresentamos também um resumo das contribuições e um conjunto de trabalhos futuros que devem ser realizados a partir do desdobramento das idéias dessa tese.

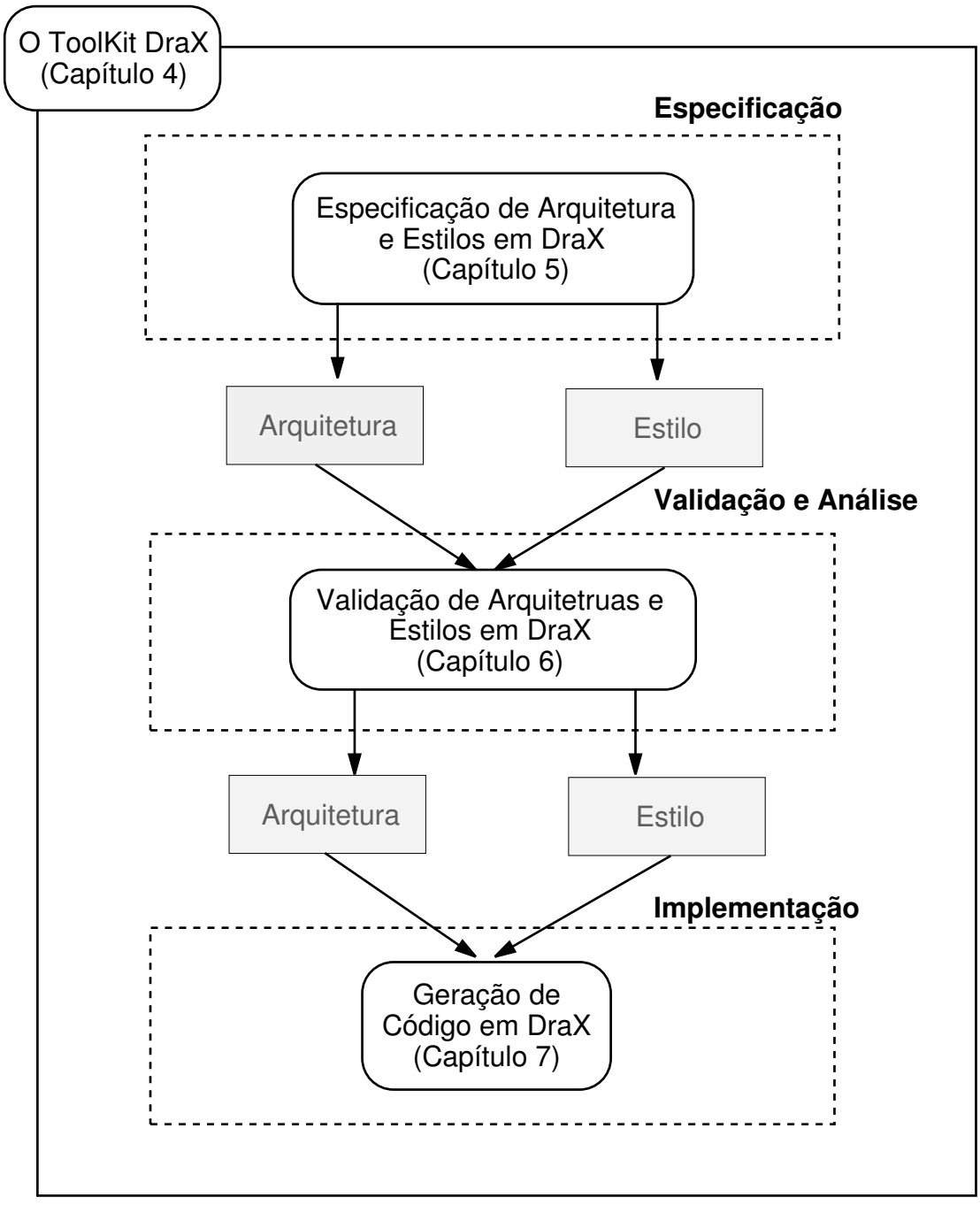


Figura 1.1: Relação entre os capítulos da tese e o ToolKit DraX.

Capítulo 2

Conceitos Básicos

2.1 Introdução

Nesse capítulo alguns dos conceitos utilizados ao longo dessa tese são apresentados de forma mais detalhada. A intenção principal aqui é fornecer um embasamento conceitual básico para quem não estiver totalmente inteirado das tecnologias abordadas ao longo do trabalho.

Inicialmente os conceitos de arquitetura de software e estilos arquiteturais são apresentados. Em seguida, apresentamos um texto introdutório sobre middlewares, onde as arquiteturas CORBA e RMI são descritas. Uma introdução sobre XML é apresentada em seguida. Por fim, apresentamos uma seção sobre o π-cálculo, pois essa álgebra é utilizada com base para as verificações comportamentais e das análises de arquitetura de DraX, sendo que esse cálculo serviu de base para propormos o cálculo $R\pi$ (veja a Seção 6.3) que é uma álgebra de processos com características mais apropriadas para a especificação formal de estruturas de interconexão de arquiteturas, e particularmente em DraX, é utilizada para fornecer a semântica formal da ADL que propomos.

2.2 Arquitetura de Software e Estilos Arquiteturais

Diversos autores apresentam suas próprias definições de arquitetura de software [90, 102, 9]. Essas definições possuem uma certa diferença em seus detalhes, mas todas tem em comum o consenso em relação aos elementos utilizados nas descrições arquiteturais - estrutura, componentes e conectores.

Arquitetura de software [102] é a descrição mais abstrata do software cuja definição ocorre nas fases iniciais do desenvolvimento. A arquitetura de software é definida através de três abstrações básicas: componentes, conectores e configuração. Componentes modelam os elementos que realizam computação ou armazenam informações, como por exemplo clientes e servidores, e comunicam-se com o ambiente externo através de uma interface (conjunto de portas). Conectores modelam as interações entre os componentes e as regras que governam estas interações, como por exemplo *Remote Procedure Call*. A configuração descreve como componentes e conectores são interligados.

A arquitetura de software desempenha um papel importante no desenvolvimento de sistemas de software, permitindo um melhor entendimento do sistema através da separação dos elementos de computação (componentes) e de interação (conectores). Ela fornece diversos benefícios, tais como, maior rapidez na construção de sistemas através do reuso de componentes

Categoria	**Subcategoria**
Componentes Independentes	Processos Comunicantes Sistema de Eventos
Fluxo de Dados	*Batch* Sequencial *Pipes-and-Filters*
Sistemas Centrado em Dados	Repositórios *Blackboards*
Máquinas-Virtuais	Interpretador Sistemas baseados em Regras

Tabela 2.1: Exemplos de Estilos Arquiteturais.

de alta escala, o que contribui para a redução de custo, tempo e esforço de desenvolvimento e manutenção do software.

2.2.1 Estilos Arquiteturais

Um estilo arquitetural [41, 102] define um vocabulário de elementos de projeto (tipos de componentes e conectores) e um conjunto de restrições sobre como esses elementos podem ser combinados. Ele permite o reuso de organizações arquiteturais estabelecidas para resolver um determinado problema recorrente.

Existem numerosos estilos arquiteturais, alguns são genéricos e outros específicos. Neste ponto, destacamos o conceito de arquitetura de software de domínio específico ou arquitetura de referência [102, 9] que prescrevem configurações (frequentemente parametrizadas) de componentes e interações para aplicações de áreas específicas, como por exemplo, a organização de um compilador.

Estilos não são arquiteturas, mas são de grande utilidade durante o projeto da arquitetura de um sistema específico. A Tabela 2.1 [102] mostra um catálogo de estilos arquiteturais mais comuns.

Estilos arquiteturais são importantes na etapa de criação da arquitetura, mas para que a arquitetura seja a base do projeto ela deve ser comunicada claramente para todos os que tenham interesse no projeto, como clientes, usuários finais, gerentes, analistas, projetistas e programadores. Para realizar essa tarefa podemos utilizar uma ADL (*Architecture Description Language*).

2.2.2 Linguagens de Descrição de Arquitetura

Linguagens de Descrição de Arquitetura (ADLs - *Architecture Description Languages*) são linguagens semi-formais usadas na representação de uma arquitetura. Com ênfase na estrutura de alto nível da aplicação em vez de se deter a detalhes de implementação [76]. ADLs têm se tornado recentemente uma área de pesquisa intensa na comunidade de arquitetura de software. Diversas ADLs têm sido propostas para modelagem de arquiteturas, tanto para domínio específico como para propósito geral. As ADLs mais comuns são: ACME, Aesop, C2, Darwin, MetaH, Rapide, SADL, UniCon, Weaves e Wright. Em [79] pode ser encontrado um *framework* que classifica e compara essas ADLs.

Um outro caminho seguido no campo de linguagens para representar arquiteturas de software é a utilização de uma notação de propósito geral [42]. Em particular, a *Unified Modelling Language* (UML) [15], devido esta linguagem fornecer uma série de benefícios, tais como: padrão largamente aceito e difundido, familiaridade entre desenvolvedores, mapeamento próximo à implementação e suporte oferecido por ferramentas disponíveis. O uso de UML para representar arquitetura tem sido investigado em diversas pesquisas [77, 34, 98, 67, 70, 48, 93].

UML fornece três mecanismos importantes: *stereotypes*, *tagged values* e *constraints* que podem ser usados para adicionar novos construtores de modo a tratar novos aspectos de desenvolvimento, sem violar a sintaxe e semântica da UML padrão. Desta forma, não comprometem um dos seus benefícios, que é o suporte oferecido por ferramentas disponíveis. Contudo, o que se tem concluído até agora é que ADLs e UML podem ser complementares ou alternativas em um processo de desenvolvimento.

2.3 Infraestruturas de Middleware

Nessa seção abordaremos duas infraestruturas de middleware que atualmente são utilizados em diversos projetos de aplicações distribuídas. Inicialmente, apresentamos RMI (*Remote Method Invocation*), que é uma arquitetura disponível como um pacote de software na linguagem Java. Em seguida a arquitetura CORBA é apresentada. CORBA é um padrão largamente aceito e que possui implementações de diversos serviços importantes para o desenvolvimento de aplicações distribuídas.

2.3.1 *Remote Method Invocation* (RMI)

O sistema RMI consiste em três camadas: a camada de *stub/skeleton*, a camada de referência remota, e a camada de transporte. O limite de cada camada é definido por uma interface e protocolo específicos; então, cada camada é independente da próxima e pode ser substituída por uma implementação alternativa sem afetar as outras camada no sistema. Por exemplo, o implementação de transporte atual é baseada em TCP (usando sockets Java), mas poderia ser substituído por um transporte baseado em UDP.

Para realizar a transmissão transparente de objetos de um espaço de endereçamento para outro, a técnica de serialização de objetos (especificamente projetada para a linguagem Java) é usada. Outra técnica, chamada de carregamento dinâmico de *stubs*, é usada para fornecer *stubs* do lado cliente que implementam o mesmo conjunto de interfaces remotas de um objeto remoto. Esta técnica, usada quando um stub do tipo exato já não está disponível para o cliente, permitindo um cliente usar os operadores embutidos da linguagem Java para *casting* e checagem de tipos.

2.3.1.1 Arquitetura

Como apresentado anteriormente, o sistema de RMI consiste em três camadas :

- A camada de *stub/skeleton* : *stubs* de cliente (procurações) e esqueletos do servidor;
- A camada de referência remota : comportamento de referências remotas (como invocação para um único objeto ou para um objeto replicado);
- A camada de transporte : montar conexão e administração de objetos localizados remotamente.

Uma invocação de método remoto de um cliente para um objeto de servidor remoto viaja abaixo pelas camadas do sistema RMI até a camada de transporte do lado cliente, então caminha para cima pela camada de transporte do lado do servidor.

Um cliente que invoca um método em um objeto do servidor remoto, de fato faz uso de um *stub* ou procuração para o objeto remoto como um canal para este objeto. Este *stub* é uma implementação das interfaces remotas do objeto remoto e envia os pedidos de invocação para aquele objeto servidor através da camada de referência remota. Os *stubs* são gerados usando o compilador *rmic*. A camada de referência remota é responsável por realizar a semântica da invocação. Por exemplo, a camada de referência remota é responsável por determinar se o servidor é um único objeto ou é um objeto replicado que requer comunicações com localizações múltiplas. Cada implementação de objeto remoto escolhe sua própria semântica de referência remota - se o servidor é um único objeto ou é um objeto replicado que requer comunicações com suas réplicas.

A camada de transporte é responsável para criação e administração de conexão, além de manter e despachar pedidos para objetos remotos.

A Camada de Stub/Skeleton

A camada de *stub/skeleton* é a interface entre a camada de aplicação e o resto do sistema RMI. Esta camada não trata particularmente de qualquer transporte, mas transmite dados à camada de referência remota pela abstração de *streams*. Os *streams* empregam um mecanismo chamado serialização que habilita objetos Java a serem transmitidos entre espaços de endereçamento. Objetos transmitidos usando o sistema de serialização de objetos são passados através de cópia ao espaço de endereço remoto, a menos que eles sejam objetos remotos, onde são passados através de referência.

Um *stub* para um objeto remoto é a procuração do lado cliente para o objeto remoto. Tal *stub* implementa todas as interfaces que são fornecidas pela implementação de objeto remoto. Um *stub* cliente é responsável por:

- Iniciar as chamadas para objetos remotos (chamando a camada de referência remota);
- Empacotamento de argumentos para um *stream;*
- Informar à camada de referência remota que a chamada deve ser realizada;
- Desempacotar o valor de retorno ou exceção do *stream; e*
- Informar à camada de referência remota que a chamada foi completada.

Um squeleton para um objeto remoto é uma entidade do lado servidor que contém um método que despacha chamadas para a implementação de objeto remoto atual. O esqueleto é responsável por:

- Empacotamento de argumentos para um *stream;*
- Realizar a chamada da implementação de objeto remoto; e
- Empacotar o valor de retorno da chamada ou uma exceção.

A Camada de Referência Remota

A camada de referência remota se trata da interface de transporte de baixo nível. Esta camada também é responsável por realizar um protocolo de referência remoto específico que é independente dos *stubs* cliente e esqueletos do servidor.

Cada implementação de objeto remoto escolhe sua própria subdivisão de classe de referência remota que opera em seu lado. Vários protocolos de invocação podem ser utilizados nessa camada. Exemplos são:

- Invocação *Unicast* ponto-a-ponto.
- Invocação para grupos de objeto replicados.

A camada de referência remota tem dois componentes cooperando: o lado cliente e os componentes do lado servidor. O componente do lado cliente contém informação específica para o servidor remoto (ou servidores, se a referência remota é de um objeto replicado) e comunica pelo transporte ao componente do lado servidor. Durante cada invocação de método, o cliente e os componentes do lado servidor executam a semântica de referência remota específica. Por exemplo, se um objeto remoto é parte de um objeto replicado, o componente do lado cliente pode remeter a invocação para cada réplica em lugar um único objeto remoto.

De uma maneira correspondente, o componente do lado servidor implementa a semântica de referência remota específica para entregar uma invocação de método remoto ao esqueleto. Por exemplo, este componente assegura a entrega multicast atômica se comunicando com outros servidores em um grupo de réplica.

A camada de referência remota transmite dados à camada de transporte pelo abstração de uma conexão orientada à fluxo. O transporte cuida dos detalhes de implementação de conexões. Embora conexões apresentem uma interface baseada em fluxo, um transporte sem conexão pode ser implementado sob essa abstração.

A Camada de Transporte

Em geral, a camada de transporte do sistema RMI é responsável por:

- Montar conexões em espaços de endereços remotos.
- Administrar conexões.
- Monitorar conexões.
- "Escutar" chamadas que entram.
- Manter uma tabela de objetos remotos que residem no espaço de endereçamento.
- Montar uma conexão para uma chamada que entra.

A abstração de transporte administra canais. Cada canal é uma conexão virtual entre dois espaços de endereçamento. Dentro de um transporte, existe apenas um canal para um par de espaços de endereçamento (o espaço de endereçamento local e um espaço de endereçamento remoto). Determinado um *endpoint* para um espaço de endereçamento remoto, um transporte monta um canal para aquele espaço de endereçamento. A abstração de transporte também é responsável por aceitar chamadas em conexões que entram para o espaço de endereçamento, montando um objeto de conexão para a chamada, e despachando para as camadas mais altas do sistema.

2.3.1.2 Carregamento Dinâmico de Classes

Em RPC, o código de *stub* do lado cliente deve ser gerado e deve ser unido ao cliente antes da chamada de procedimento remoto ser realizada. Este código pode ser ou unido estaticamente ao cliente ou pode ligado em tempo de execução de forma dinâmica através de bibliotecas ou em cima de um sistema de arquivo de rede. No caso de ligação estática ou dinâmica, o código específico para direcionar um RPC deve estar disponível na máquina de cliente em forma compilada.

RMI generaliza esta técnica e usa um mecanismo chamado carregamento dinâmico de classe para carregar em tempo de execução as classes necessárias para direcionar as invocações de métodos em um objeto remoto.

Estas classes são:

- As classes de objetos remotos e suas interfaces.
- As classes *stub* e esqueleto que servem como procurações para objetos remotos.
- Outras classes usaradas diretamente em uma aplicação baseada em RMI.

Além de carregadores classe, o carregamento dinâmico de classe emprega dois outros mecanismos: o sistema de serialização de objeto para transmitir classes pela rede, e o gerente de segurança, para conferir as classes que estão sendo carregadas.

2.3.1.3 Segurança

Em Java, quando um carregador de classe carrega classes do CLASSPATH local, essas classes são consideradas segurar e não são restringidas por um gerente de segurança. Porém, quando o `RMIClassLoader` tenta carregar classes da rede, deve haver um gerente de segurança ou uma exceção é lançada.

O gerente de segurança deve ser inicializado na primeira ação de um programa Java de forma que esse possa regular as ações subseqüentes. O gerente de segurança assegura que aquelas classes carregadas aderem à segurança que Java padrão garante, por exemplo que classes são carregadas de "fontes confiáveis".

Applets sempre estão sujeito às restrições impostas pela classe `AppletSecurity`. Este gerente de segurança assegura que classes só são carregadas do host do applet ou dos host designados no codebase.

Aplicações ou têm que definir seu próprio gerente de segurança ou têm que usar o `RMISecurityManager` restritivo. Se nenhum gerente de segurança for indicado, uma aplicação não pode carregar classes da rede.

2.3.2 A Arquitetura CORBA

A CORBA (*Common Object Request Broker Architecture*) [53] é uma arquitetura padrão para o desenvolvimento de sistemas distribuídos e foi inicialmente implementada em 1991 pelo *Object Management Group* (OMG), um consórcio de mais de 700 companhias das mais diferentes áreas interessadas em prover uma estrutura comum para o desenvolvimento independente de aplicações, usando técnicas de orientação a objeto em redes de computadores heterogêneas. Ao invés de aplicações, o OMG produz especificações que tornam a computação orientada a objeto possível. Este modelo baseado em objetos permite que métodos de objetos

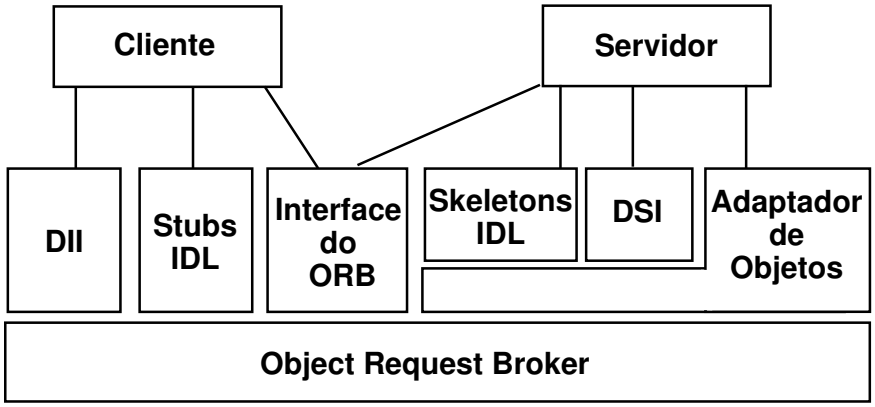


Figura 2.1: Estrutura de CORBA 2.0.

sejam ativados remotamente, através de um elemento intermediário chamado *Object Request Broker* (ORB), situado entre o objeto propriamente dito e o sistema operacional.

Introduzido como parte do CORBA 2.0 em dezembro de 1994, o *Internet Inter-ORB Protocol* (IIOP) é a solução para interoperabilidade produzida pelo OMG. Antes do IIOP, a especificação CORBA definia somente interação entre objetos distribuídos criados pelo mesmo fornecedor. Os objetos tinham que ser desenvolvidos por uma implementação específica. Usando-se IIOP, a segunda especificação CORBA torna-se a solução definitiva para a interoperabilidade entre objetos que não estão presos a uma plataforma ou padrão específico. A CORBA encontra-se na versão 3.0.1, revisada em Novembro de 2002, e propõe novas características como: passagem de objeto por valor, interoperabilidade entre CORBA e DCOM e a adoção do Portable Object Adapter (POA) como adaptador de objetos.

A especificação da interface de objetos CORBA é escrita em uma linguagem neutra chamada Linguagem de Definição de Interfaces (*Interface Definition Language* - IDL). Tal interface é independente de linguagem de programação e sistema operacional adotados. A interface escrita em IDL é puramente declarativa e, portanto, sem implementação. A Figura 2.1 mostra a arquitetura que surgiu na especificação CORBA 2.0 e ainda se mantém.

Repositório de Interface Repositório de Interface (*Interface Repository* - IR) CORBA é um banco de dados das definições dos objetos. Essas definições podem ser acessadas, escritas e destruídas em tempo de execução. O IR é formado por um conjunto de objetos CORBA cujas operações podem ser invocadas como qualquer outro objeto CORBA. Dessa forma, aplicações podem vasculhar a inteira definição de uma interface OMG IDL usando as operações disponívies nos objetos do IR. Há duas formas para acessar as informações do IR:

- Iniciando em um escopo de alto nível do IR e interagindo sobre as definições dos módulos. Quando o módulo desejado é encontrado, ele pode ser aberto e a iteração pode ser realizada sobre suas definições internas;
- Obter uma referência para o objeto `InterfaceDef` a partir da operação `get_interface` definida na interface `CORBA::Object`. Como todo objeto CORBA suporta esta operação, uma vez que qualquer objeto CORBA é derivado de `CORBA::Object`, a definição da interface pode ser obtida sem dificuldade.

Repositório de Implementação O Repositório de Implementação fornece um banco de dados sobre as informações de objetos que o servidor suporta, os objetos que são instanciados e seus identificadores (IDs). Também serve como um local de armazanamento de informações adicionais associadas com a implementação de ORBs. Além disso, o método `get_implementation` pode ser invocado sobre qualquer referência de objeto para se obter uma representação do Repositório de Implementação, que descreve a implementação do objeto onde o método foi invocado.

Stubs e Skeletons Os compiladores IDL geram *stubs* do lado cliente e *skeletons* do lado servidor. O primeiro é um mecanismo que cria e lança requisições em nome do cliente, enquando que o segundo é responsável pela entrega dessas requisições para a implementação do objeto CORBA. Assim, o *stub* deve ser gerado na mesma linguagem de programação do objeto cliente. O mesmo é válido para o *skeleton* em relação ao objeto servidor.

Invocação Dinâmica Além da invocação estática, CORBA suporta duas interfaces de invocação dinâmica: Interface de Invocação Dinâmica (*Dynamic Invocation Interface* - DII), que suporta invocação de requisições dinâmicas dos clientes; e a Interface de Skeleton Dinâmico (*Dynamic Skeleton Interface* - DSI), que fornece um *skeleton* dinâmico para servidores. A DII e a DSI podem ser vistas como *stubs* e *skeletons* genéricos, respectivamente.

Interface de Invocação Dinâmica A DII permite que aplicações clientes usem objetos servidores sem, previamente, conhecer a interface desses objetos em tempo de compilação. O cliente pode obter uma referência de um objeto CORBA e fazer invocações sobre o mesmo construindo as requisições dinamicamente.

A DII não possui um desempenho tão bom quanto a invocação estática, mas oferece algumas vantagens como:

- Clientes não estão restritos a usarem somente a interface definida quando o cliente foi compilado;
- Clientes não precisam ser recompilados para acessar novas operações exportadas pela implementação do objeto.

A DII faz uso do repositório de interfaces para validar e recuperar a assinatura do método sobre o qual uma requisição é feita.

Interface de Skeleton Dinâmico Análoga à DII, a DSI permite que objetos servidores sejam escritos sem possuir skeletons para objetos sendo implementados. Do ponto de vista de uma aplicação cliente, um objeto implementado com DSI comporta-se da mesma forma que qualquer outro objeto CORBA. Os clientes não precisam realizar qualquer tratamento especial para manipular implementações de objetos que usam DSI.

A implementação de objetos com DSI requer mais atividade de programação do que o uso de skeletons estáticos, porém permite que um objeto possa oferecer múltiplas interfaces. A

ISD faz uso do repositório de implementações para recuperar a implementação da operação que está sendo requisitada pelo cliente.

Diferente da DII, que foi partr da especificação inicial da CORBA, a DSI foi introduzida em CORBA 2.0. Seu principal propósito era dar suporta à implementação de gateways entre ORBs que utilizavam diferentes protocolos de comunicação. Entretanto, a DSI tem outras aplicações além da interoperabilidade. Essas aplicações incluem ferramentas de software interativas baseadas em interpretadores e depuradores distribuídos.

Adaptadores de Objetos (*Object Adapter* - OA) A implementação de um objeto CORBA acessa os serviços fornecidos pelo ORB através do adaptador de objetos. Este possui algumas responsabilidades que tornam o ORB tão simples quanto possível, uma vez que o ORB não precisa mais se preocupar com as seguintes tarefas realizadas pelo adaptador de objetos:

- Registro de objeto: o OA fornece operações que permitem que entidades de uma linguagem de programação sejam registradas como implementação para objetos CORBA;
- Geração de referência de objeto: o OA gera referência de objetos para objetos CORBA;
- Ativação de processo servidor: o OA inicia processos servidores, se necessário, onde objetos podem ser ativados;
- Ativação de objeto: o OA ativa objetos, se necessário, quando requisições chegam a eles; e
- Chamada de objetos: o OA entrega requisições para objetos registrados.

O OA registra os objetos que ele suporta e suas instâncias junto ao Repositório de Implementação.

Adaptador de Objetos Básico (*Basic Object Adapter* - BOA) A especificação CORBA 2.0 trouxe o BOA como seu principal adaptador de objetos, apesar de suportar mais de um OA, e pretendia que este fosse suficiente para grande parte das implementações. Porém, como os adaptadores de objetos tendem a ser bastante específicos da linguagem devido a sua proximidade com os objetos da linguagem de programação e a especificação do BOA era bastante vaga em alguns aspectos, o resultado foi um grande problema de portabilidade entre implementações de BOAs, pois cada fabricante preenchia as lacunas da especificação com soluções proprietárias.

Adaptador de Objetos Portável (*Portable Object Adapter* - POA) "POA é o BOA feito corretamente" [89]. Ele faz tudo que o BOA faz e introduz alguns conceitos novos que tornam o adaptador de objetos mais robusto. Faz parte da última versão da especificação CORBA. O POA suporta objetos transientes e persistentes. Os primeiros ficam ativos somente dentro do processo que os criou. Os outros permanecem ativos além do processo que os criou, ou seja, se o processo criador foi eliminado, os objetos ativos nada sofrem. O POA permite que um gerente de serventes seja ativado para cada implementação do objeto, onde cada servente

é uma implementação de uma interface em execução. Assim, o POA invoca operações sobre os gerentes de serventes para criar, ativar e desativar serventes. O POA mantém um mapa de gerentes de serventes ativos e também um mapa de identificadores de objetos ativos (serventes) chamado Mapa de Objetos Ativos. Um objeto ativo é identificado por sua referência de objeto que, por sua vez, encapsula um identificador de objeto (`Object Id`) .

2.3.2.1 Serviços CORBA

O OMG, através da OMA (*Object Management Architecture*), define os serviços CORBA, que estendem as funcionalidades do ORB e dão suporte para a criação de grandes aplicações distribuídas. A implementação desses serviços não é definida pelo OMG, somente as interfaces pelas quais os serviços são acessados. A implementação fica a cargo de desenvolvedores de software.

Os serviços CORBA podem ser combinados de tal forma que as funcionalidades de cada um possam ser somadas a fim de fornecer o framework mínimo para atender as necessidades das aplicações distribuídas. Dentre os diversos serviços especificados pelo OMG e que já possuem implementação podemos destacar:

- Serviço de Nomes: permite que objetos CORBA sejam registrados e localizados através de um nome determinado pela aplicação;
- Serviço de Trading: permite que objetos CORBA sejam localizados através dos serviços (interfaces) que são ofertados;
- Serviço de Eventos: fornece um mecanismo através do qual objetos CORBA podem enviar e receber eventos;
- Serviço de Persistência: fornece um conjunto de interfaces para gerenciar a persistência de objetos CORBA;

Nesse trabalho utilizamos os serviços de Nomes e Eventos dentro do *framework* DraX. Contudo, a extensão desse *framework* para a utilização de outros serviços CORBA pode ser realizada facilmente (apresentamos os passos para realizar essa tarefa mais adiante nessa tese).

2.4 A Linguagem XML

XML é a sigla de *eXtensible Markup Language* [16], ou seja, linguagem de marcação extensível, uma linguagem de marcação como HTML (*HyperText Markup Language*). Apesar de a primeira vista parecer semelhante a HTML, há diferenças importantes. HTML serve para apresentação de dados enquanto XML foi concebida para descrição de dados, em HTML os marcadores são fixos e pré-definidos, o que não acontece com XML. Com XML não é somente o conteúdo dos dados que é armazenado, mas também a relação estrutural existente dentro dos dados. Como HTML, XML é derivada da mãe de todas as linguagens de marcação, SGML (*Standard Generalized Markup Language*). SGML é um padrão internacional para definir a estrutura e conteúdo de documentos eletrônicos.

SGML surgiu há trinta anos na tentativa de definir uma linguagem de marcação para representação de informação textual. Um documento SGML, assim como um documento XML, é definido como uma coleção de elementos identificados por marcadores (ou tags) que descrevem sua ação dentro de um dado contexto. Linguagens de marcação são ideais para

organizar informação que é estruturada, sujeita a freqüentes mudanças e portável. SGML é excessivamente geral e complexa para uso na Web ou em aplicações.

Sendo uma linguagem de marcação usada para descrever dados, qualquer um que receba e envie dados, armazenados ou transmitidos de um lugar a outro, é um usuário potencial das capacidades de XML. XML permite que se descreva e entregue dados para qualquer aplicação em um modelo padrão e consistente.

As aplicações mais conhecidas de XML são as relacionadas à Internet, mas há muitas outras aplicações em que XML é útil, como por exemplo em substituição ou complemento a bases de dados tradicionais ou para transferência de informações entre agentes de negócios. A família de tecnologias XML é formada por especificações relacionadas, algumas delas em estágio inicial de desenvolvimento. As mais importantes são:

- XML 1.0 [16] é a especificação base sobre a qual a família XML está construída. Ela descreve a sintaxe que os documentos XML devem seguir, as regras que os validadores XML devem utilizar para ler e escrever documentos XML;

- Como a estrutura e os nomes dos elementos do documento são criados pelo autor, DTD (*Document Type Definition*) [16] e Schema [25] provêem meios de criar modelos para tipos de documentos (também chamados de vocabulários). A partir daí pode-se verificar se o documento obedece ao modelo e produzir documentos compatíveis. Fica então garantida a validade e integridade dos dados do documento XML;

- Namespaces [19] fornecem um significado para distinguir um vocabulário XML de outro, o que permite criar documentos mais ricos combinando múltiplos vocabulários em um único tipo de documento;

- XPath [20] descreve uma linguagem de consulta para endereçar (acessar) partes de um documento XML. Isto permite que aplicações procurem uma parte específica do documento ao invés de ter sempre que tratar um grande bloco de informação;

- Em alguns casos é preciso visualizar o conteúdo do documento XML. *Cascading Stylesheets* (CSS) podem ser utilizados para definir a apresentação do documento, ou ainda *eXtensible Stylesheet Language* (XSL) [22]. XSL é constituído de XSLT (*eXtensible Stylesheet Language Transformation*) [18] que pode transformar o documento de um tipo em outro, e ao mesmo tempo adicionar elementos de estilo tais como HTML, gráficos, ou arquivos de audiovisual; e objetos de formatação que tratam o formato do documento para exibição e impressão;

- XLink [23] e XPointer [24] são especificações para definir ligação de documentos XML (ou partes de documentos XML) entre si ou a recursos não-XML, similar aos hyperlinks HTML, porém bem mais poderosos. XML tem mecanismos para ligações entre múltiplos recursos e ligações entre recursos apenas de leitura. XPointer descreve como endereçar um recurso e XLink descreve como associar dois ou mais recursos; e

- Para prover um modo de aplicações tradicionais acessarem dados de documentos XML há um modelo de objeto de documento (DOM - *Document Object Model*) [21] que fornece um conjunto de objetos para representar o documento XML como uma árvore, e métodos e propriedades que podem ser usadas para manipular esses objetos. Uma alternativa para programar a interface com documentos XML é usar o) [1] que induz o validador a gerar um conjunto de eventos à medida que percorre o documento.

2.4.1 Características

XML é um mecanismo poderoso para troca de dados. É a solução ideal para transferência de dados estruturados de servidor para cliente, de servidor para servidor ou de uma aplicação para outra. As características que fazem de XML essa ferramenta ideal para troca de dados, inclusive ou principalmente pela internet, são [4]:

- **XML é informação textual independente de plataforma**. Sendo texto é fácil desenvolver validadores e todas as outras tecnologias em plataformas diferentes. Provê interoperabilidade optando pela abordagem do mínimo denominador comum. Recebendo e enviando informação na forma de texto, programas executando em plataformas díspares podem se comunicar. Para aproveitar essa característica pode-se escrever programas que funcionem em cima de sistemas legados realizando a comunicação com o mundo externo através de documentos XML.

- **XML é um padrão aberto**. Ter uma organização, o World Wide Web Consortium (W3C), como mantenedora do padrão XML, assegura que ninguém poderá causar problemas de interoperabilidade entre sistemas que usam o padrão. Manter todos os dados em XML e usar XML como protocolo de comunicação maximiza o tempo de vida do investimento em produtos e soluções.

- **XML é totalmente extensível**. XML aceita qualquer número de marcadores desde que eles respeitam as regras de sintaxe de XML. É permitido aos desenvolvedores customizar seus dados como eles acharem adequado. XML é uma linguagem genérica e pode ser usada para a criação de versões especializadas que suportem um conjunto fixo de marcadores e atributos projetados para um propósito particular. De fato, podemos considerar XML como uma meta-linguagem para definição de linguagens de marcação.

- **XML é independente de linguagem**. XML não requer um padrão binário de codificação ou um formato de armazenamento. Isso estimula a interoperabilidade entre sistemas heterogêneos e é bom para a compatibilidade futura.

- **DOM e SAX são interfaces abertas, independentes de linguagem**. Definindo um conjunto de interfaces de programação independentes de linguagem para permitir acesso e mudança de documentos XML, o W3C tornou fácil a programação para lidar com XML.

- **XML é capacitada para Web**. XML é derivado de SGML assim como HTML. Assim, em essência, a infraestrutura existente hoje para tratar conteúdo HTML pode ser usada para trabalhar com XML. É uma grande vantagem poder dispor da estrutura de software e de rede em uso atualmente para entrega de conteúdo XML. Mesmo que o cliente não suporte XML nativamente, com folhas de estilo, pode-se converter XML em HTML para apresentação.

- **XML suporta estrutura compartilhável (usando DTD ou Schema)**. A estrutura de XML pode ser especificada em DTD (*Document Type Definition*) ou em DS (*Data Schema*), isto fornece um meio de trocar documentos XML em conformidade com um modelo. Usando DTD e Schema pode-se ter certeza que dois ou mais documentos XML são do mesmo tipo.

- **XML permite interoperabilidade**. Todas as vantagens citadas trabalham juntas para tornar a interoperabilidade possível. Esse é um dos mais importantes requisitos de XML, capacitar sistemas díspares a compartilharem informação facilmente. Por adotar a abordagem do mínimo denominador comum, sendo capacitado para Web, independente de protocolo, independente de rede, independente de plataforma e extensível, XML torna possível a comunicação entre novos e velhos sistemas. Codificar informação em texto com marcadores é melhor que usar formatos binários proprietários e dependentes de plataforma.

- **XML é auto-descritiva**. Um pequeno conjunto de elementos XML é usado para definir qualquer novo conjunto de elementos (e sua estrutura hierárquica) que estão contidos em um documento XML.

2.5 O π-cálculo

O π-cálculo [86] é um cálculo para os processos móveis, isto é, a estrutura de comunicação pode modificar dinamicamente durante a execução de um processo. A mobilidade é obtida permitindo que os nomes de portas de comunicação sejam passados como valores nas interações entre processos, modificando desse modo o escopo da porta. Como exemplo seja um sistema com três processos P, Q e R apresentado na Figura 2.2, onde P compartilha uma porta de comunicação de nome x com Q (isto é, ambos conhecem a mesmo nome de porta x), mas R não conhece esse nome (Figura a esquerda). Q pode enviar x para R através da porta de nome y, e depois disso P, Q e R podem se comunicar diretamente através de x (Figura do meio). Se Q libera o nome x, somente P e Q o conhecem e o "movimento" de x é completado (Figura da direita).

Essa passagem de nomes pode ser vista como passagem de ponteiros para objetos ou processos. No exemplo apresentado podemos ver x como um ponteiro para o processo P. O nome x pode ser passado sobre R, que por sua vez pode ativar P usando x. Nesse sentido nós podemos dizer que P "moveu-se" de Q para R.

O π-cálculo postula um conjunto de nomes, variado entre u, v,..., z, usado para representar portas de comunicação, valores, variáveis, etc. Uma interação ocorre entre dois processos, um remetente executando uma saída e um recebedor executando uma entrada. O efeito dessa interação é local para o recebedor: as variáveis de entrada são instanciadas com os objetos emitidos pelo remetente. Por exemplo:

$$\overline{u}vw.P \mid u(xy).Q \xrightarrow{\tau} P \mid Q(v/x, w/y)$$

O prefixo de saída $\overline{u}wv.P$ envia dois objetos v e w ao longo de u antes de continuar com o processo P, e o prefixo de entrada $u(xy).Q$, o qual liga x e y, ao contrário, recebe dois objetos. A interação entre esses processos causa uma transição com o nome τ, representando uma

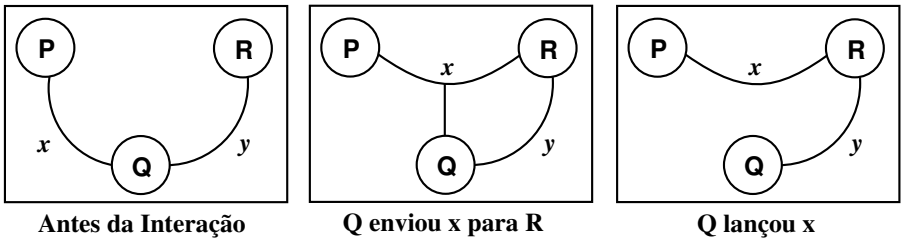


Figura 2.2: Interação em π-cálculo.

atividade interna, e a substituição dos objetos recebidos pelos objetos ligados em Q. Como a ligação é local para Q, a substituição também é local para Q.

Um outro tipo de ligação ocorre no operador de restrição: em $(\nu x)P$ o nome x em P é local para P (ou "novo nome em P"). Isso significa que x é diferente de qualquer outro nome ocorrendo fora de P e de outros nomes ocorrendo em P e, além disso, que caso P como seu ambiente evoluam durante a execução, esse nome permanecerá distinto. Apresentaremos agora a sintaxe formal e semântica de π-cálculo. Escrevemos $\tilde{x}$ para indicar uma seqüência finita (possivelmente vazia) $x_1 \ldots x_n$ de nomes, e $|\tilde{x}|$ para o tamanho da seqüência $\tilde{x}$.

Definição 2.1 *Os prefixos, representados por b, e os agentes, representados por P, Q, ..., são definidos como sendo:*

Ações:

$$\begin{array}{rcll} \alpha & ::= & a(x) & \textit{(Entrada)} \\ & | & \overline{a}y & \textit{(Saída)} \end{array}$$

Agentes:

$$\begin{array}{rcll} P & ::= & \emptyset & \textit{(Inação)} \\ & | & \alpha.P & \textit{(Prefixação)} \\ & | & Q + R & \textit{(Soma)} \\ & | & Q \mid R & \textit{(Composição)} \\ & | & (\nu x)\, Q & \textit{(Restrição)} \end{array}$$

1. 0 é o agente vazio, que não pode realizar nenhuma ação.
2. $\alpha.P$ significa que o processo realiza uma das ações α e então continua como P.
3. $Q + R$ representa um agente que age ou como Q ou como R.
4. $Q \mid R$ representa a combinação de comportamento de Q e R executando em paralelo.
5. $(\nu x)\, Q$ introduz um novo nome x como escopo Q (ligando todas a ocorrências livres de x em Q.

Inspirada pela Máquina Abstrata Química de Berry e Boudol [12], a semântica de π-cálculo é apresentada baseada em uma congruência estrutural e uma relação de transição. Isso nos permite separar a estrutura física dos agentes de sua interação. A congruência estrutural iguala todos os agentes não distinguíveis por nenhuma razão semântica.

PREF	$\frac{-}{\alpha.P \xrightarrow{\alpha} P}$	(Prefixing)
SUM	$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$	(Summation)
PAR	$\frac{P \xrightarrow{\alpha} P', \mathtt{fn}(\alpha) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$	(Composition)
COMM	$\frac{P \xrightarrow{a(x)} P', Q \xrightarrow{\overline{b}y} Q'}{P \mid Q \xrightarrow{\tau} P'\{y/x\} \mid Q'}$	(Communication)

Tabela 2.2: Leis de ação para o π-cálculo.

Definição 2.2 *A congruência estrutural, $\equiv$, entre agentes é a menor congruência que satisfaz as leis do monóide abeliano para Soma e Composição:*

$$P + Q \equiv Q + P, \quad P + (Q + R) \equiv (P + Q) + R, \quad P + 0 \equiv P$$
$$P \mid Q \equiv Q \mid P, \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R, \quad P \mid 0 \equiv P$$

as leis de escopo

$$(\nu x)0 \equiv 0, \quad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P, \quad (\nu x)(P + Q) \equiv (\nu x)P + (\nu x)Q$$

Definição 2.3 *A família de transições $P \xrightarrow{\alpha} Q$ é a menor família que satisfaz as leis na Tabela 2.2. Nessa definição agentes estruturalmente equivalentes são considerados iguais, isto é, se $P \equiv P'$, $Q \equiv Q'$ e então $P \xrightarrow{\alpha} Q$ então $P' \xrightarrow{\alpha} Q'$.*

Definição 2.4 *As transições fracas $P \stackrel{\alpha}{\Rightarrow} Q$ são definidas como:*

- *Se $\alpha = \tau$, então $P \stackrel{\alpha}{\Rightarrow} Q$ significa $P \Rightarrow Q$;*
- *Se $\alpha \cong \tau$, então $P \stackrel{\alpha}{\Rightarrow} Q$ significa $P \Rightarrow \xrightarrow{\alpha} \Rightarrow Q$;*

Onde $P \Rightarrow Q$ se, somente se $P(\xrightarrow{\tau})^* Q$.

2.6 Conclusão

Nesse seção observamos de forma resumida os principais conceitos relativos às tecnologias utilizadas nesse trabalho. Particularmente no que tange as informações sobre infraestruturas de middleware, outros trabalhos também são relavantes, como por exemplo DCOM [83], e MSMQ [84]. Contudo, essas informações podem ser encontradas de forma detalhada em [103].

Com relação aos serviços CORBA, que utilizamos extensivamente nos estudos de caso, também resolvemos não estender demasiadamente a explanação para não ter que entrar em detalhes de implementação tão prematuramente. Contudo, um extenso trabalho onde os principais serviços CORBA são explicados de forma conceitual e algumas implementações ilustram à utilização desses serviços pode ser encontrado em [112].

Capítulo 3

Trabalhos Relacionados

3.1 Introdução

Nesse capítulo apresentamos os principais trabalhos que estão relacionados com essa tese. Inicialmente uma breve introdução sobre a origem dos trabalhos em arquitetura de software é apresentada, sendo que as principais Linguagens de Descrição de Arquiteturas são mostradas de modo a que possamos traçar um paralelo entre essas linguagens e o *framework* DraX. Em seguida apresentamos o Modelo de Componentes CORBA (CCM), que é um modelo conceitual proposto pelo OMG e que possui algumas semelhanças com as ferramentas de DraX. Mais adiante apresentamos a MDA (*Model-Driven Architecture*), que também é um esforço do OMG para tratar de aspectos similares aos tratados nessa tese.

3.2 Arquitetura de Software e Estilos Arquiteturais

Uma base conceitual para arquitetura de software foi proposta originalmente por dois grupos, Shaw e Garlan [46, 102], e Perry e Wolf [90]. Nestes trabalhos iniciais são descritos os temas principais da área e os blocos de construção conceitual, como componentes, conectores, configurações e estilos. Estes trabalhos introduzem os conceitos através de exemplos informais, e não fornecem mecanismos ou notações específicas para trata-los. Ao invés, eles apenas identificam um campo emergente e sugere realização de pesquisas nessa área.

Shaw e Garlan [46, 100, 40] descrevem arquitetura de software como um passo necessário na elevação do nível de abstração no qual o software é concebido e desenvolvido. Eles explicam que, da mesma maneira que o desenvolvimento informal do conhecimento sobre tipos de dados nos anos 60 conduziu à sua codificação como tipos de dados abstratos nos anos 70, os desenvolvedores estão começando a ter uma noção informal de integração modular e projeto de arquitetura, e isto vai conduzir à sua codificação como arquitetura de software. Eles descrevem uma coleção de estilos arquiteturais, como *pipe-and-filters* e *blackboard* para mostrar que este tipo de abstração é encontrado informalmente na prática.

Shaw e Garlan utilizam um modelo de arquitetura baseado em duas abstrações: o componente, uma unidade independente de computação, e o conector, uma interação entre componentes. Shaw e Garlan sugerem que estilos de arquitetura de componentes e conectores, repetindo padrões de computação e interação, junto com regras de como estes são usados em configurações específicas.

Perry e Wolf [90] descrevem uma visão geral de arquitetura de software como um mediador entre requisitos e projeto. Eles adotam uma visão abstrata de arquitetura através de três idéias

básicas: Elementos, Forma, Razão e justificam essa idéia através de uma visão do processo arquitetural como a aplicação sucessiva de limites para um projeto. Além disso, eles também introduzem a idéia de estilo como restrição em uma classe de arquitetura; porém não fazem uma distinção clara entre instâncias e estilos. Para eles uma configuração de arquitetura consiste também de uma coleção de restrições, e assim a divisão entre uma configuração e um estilo torna-se não muito clara.

O modelo de estilos apresentado em DraX é baseado em uma linguagem específica para tratar estilos e portanto permite o tratamento mais refinado dessa idéia. Dessa forma, conseguimos distinguir uma arquitetura de um estilo, mesmo sabendo da necessidade de se utilizar restrições tanto sintático/topológicas como comportamentais para a descrição desses elementos.

3.2.1 Linguagens de Descrição de Arquitetura

Recentemente têm sido propostas várias Linguagens de Descrição de Arquitetura (ADLs). Estas linguagens fornecem notações para descrever a estrutura de sistemas de software em termos de configurações hierárquicas de componentes interagindo, além de uma base comum explícita para descrever configurações arquiteturais. Assim elas podem reusar estruturas em novos sistemas. Porém, como veremos adiante, as ADLs atuais possuem três deficiências principais [7]: Falta de suporte para estilos, falta de uma base para análise arquitetural, e habilidade insuficiente para descrever interações arquiteturais (conectores).

3.2.1.1 Darwin

Darwin [72] é uma linguagem de descrição de arquitetura desenvolvida por Magee e Kramer. Essa linguagem descreve tipos de componentes através de interfaces que consistem de uma coleção de serviços providos (i.e. declarados pelo componente) ou exigidos (i.e. esperado pelo componente). Configurações são desenvolvidas através de declarações de instanciação de componentes e ligações entre serviços exigidos e providos. Darwin suporta a descrição de arquiteturas dinamicamente reconfiguráveis através de dois construtores - instanciação tardia (*lazy instantiation*) e construções dinâmicas explícitas. Utilizando instanciação tardia, uma configuração logicamente infinita pode ser descrita. Nessa configuração, componentes só são instanciados quando os serviços que eles provêem são utilizados por outros componentes. As estruturas dinâmicas explícitas são fornecidas através do uso de construtores de configuração imperativos. As declarações de configuração se tornam programas que são executados em tempo-de-execução no lugar de uma declaração estática de estrutura.

Darwin provê uma semântica para seus aspectos estruturais através de π-cálculo [35]. Cada serviço provido é modelado como um nome de canal, enquanto cada declaração de ligação é um processo que transmite o nome daquele canal para um componente que requer o serviço. Magee e Kramer usaram este modelo de elaboração estrutural para analisar algoritmos distribuídos usados na implementação de configurações de Darwin.

Em uma implementação gerada por Darwin, cada componente primitivo (não-hierárquico) é implementado em alguma linguagem de programação, e um código específico de ligação com a plataforma é gerado para cada tipo de serviço. O algoritmo de elaboração age, essencialmente, como um servidor de nomes que provê um contrato de localização de serviços para qualquer componente executando.

Apesar da presença de um modelo π-cálculo para as descrições estruturais de Darwin, essa linguagem não fornece uma base adequada para análise do comportamento de uma arquitetura

Código 3.1 *Pipeline* em Darwin.

```
component pipeline(int n) f {
   provide input;
   require output;

   array F[n]:filter;
   forall k:0 ..n-1 {
      inst F[k];
      bind F[k].output - output;
      when k<n-1 bind
         F[k].next - F[k+1].prev;
   }
   bind
      input - F[0].prev;
      F[n-1].next - output;
}
```

[7]. Isto porque o modelo não provê qualquer meio de descrever as propriedades de um componente ou de seus serviços. Sendo que as implementações de componentes são caixas pretas não interpretadas.

O suporte de Darwin para estilos arquiteturais é limitado à descrição de configurações parametrizadas. Por exemplo, a descrição Darwin mostrada no Código 3.1 indica que um *pipeline* é uma sucessão linear de filtros onde a produção de cada filtro é ligada com a entrada do próximo filtro.

Darwin também está limitado por sua falta de mecanismos explícitos para introduzir novos tipos de serviços. Darwin assume que a coleção de tipos de serviço é provida pela plataforma na qual a implementação será desenvolvida, e confia na existência de nomes de tipo de serviço que são usados, sem interpretação, para verificação de compatibilidade. Assim, Darwin fornece uma checagem de consistência em configurações relativamente restritas.

Na nossa abordagem com DraX, utilizamos diversos níveis de verificação. Inicialmente verificamos a consistência da arquitetura com relação à estrutura topológica. Em seguida a verificação de comportamento permite observar se o comportamento de componentes que estão conectados são realmente compatíveis. Além disso, podemos verificar a estrutura comportamental da arquitetura com um todo.

3.2.1.2 UniCon

UniCon é uma Linguagem de Descrição de Arquitetura desenvolvida por Shaw et al. [101] como uma ferramenta para construir configurações executáveis baseadas em tipos de componentes e implementações de conectores específicos que suportam tipos particulares de conexão. UniCon é semelhante a Darwin no fato de provê ferramentas para construir configurações executáveis baseado em implementação caixa-preta e por ter uma coleção de tipos fixos de interação, mas difere em seu modelo de conectores.

Enquanto Darwin provê interações por declarações de provide/require assimétricas de conectores implícitos, UniCon suporta conectores explícitos, simétricos e assimétricos. Ou seja, uma configuração arquitetural contém declarações de conector que logicamente definem uma interação. Cada conector tem uma coleção de papéis que definem que participantes são

esperados em uma determinada interação. Interfaces de componentes, em lugar de serviços providos e requeridos, são definidos por players que têm um tipo (indicando a natureza da interação esperada) e um conjunto de propriedades (fornecendo detalhes da interação do componente com essa interface). No passo de configuração, players de componentes são associados com papéis em conectores.

Enquanto o modelo de UniCon de conectores explícitos parece ser promissor para estilos arquiteturais, criando um lugar onde poderiam ser definidas interações novas e regras de composição elaboradas, UniCon provê mecanismos limitados atualmente para definir novos tipos de conectores. Tipos novos só podem ser adicionados se implementados manualmente e adicionados à coleção de tipos embutidos. Além disso, UniCon não provê nenhum meio de descrever ou delinear famílias de sistemas. Assim, a análise de arquiteturas em UniCon é limitado às ferramentas providas com os tipos de conector e não há nenhum modo para descrever estilos arquiteturais.

No modelo de DraX fornecemos mecanismos explícitos de criação de estilos arquiteturais. Seguindo a idéia de UniCon, utilizamos interfaces nos componentes, sendo que cada interface deve possuir um papel específico. Assim, na descrição de estilos em DraX, criamos vocabulários que permitem a verificação da compatibilidade entre componentes com relação a um determinado estilo. Diferente de UniCon, em Drax oferecemos contrutores que possibilitam termos mais de uma interface em um mesmo componente, permitindo que esse participe de diversa interações obedecendo a estilos diferentes.

3.2.1.3 ACME

ACME [44] é uma Linguagem de Descrição de Arquitetura desenvolvida como um esforço articulado de vários grupos de pesquisa em arquitetura de software. Essa linguagem tem a intenção de servir como uma linguagem de intercâmbio para descrições arquiteturais. ACME define um vocabulário básico e não interpretado de componentes, conectores, portas, papéis, ligações, e configurações. Estes elementos podem ter especificações associadas através de listas de propriedades. Propriedades em ACME podem ser especificadas em qualquer outra ADL. A semântica das propriedades e, até mesmo da especificação arquitetural com um todo, é fornecida por estas ADLs auxiliares. A meta é que uma especificação escrita em uma ADL, como UniCon, poderia compartilhar uma estrutura arquitetural comum com uma especificação em outra ADL, como Darwin, e assim o arquiteto poderia tirar vantagem na utilização de ferramentas analíticas de outras ADLs.

Pelo fato de adotarmos XML como base para a criação das linguagens de DraX, obtemos também o mesmo potencial de ACME como linguagem de intercâmbio de informações, sendo possível transformar uma especificação ArchML em uma especificação em outra ADL através de, por exemplo, *scripts* XSLT.

3.2.2 Descrição Arquitetural em XML

A idéia de se ter ADLs que sejam facilmente manipuladas sintaticamente estimulou a criação de diversos outros esforços na adoção de XML para essa tarefa, como por exemplo as linguagens Xarch [28], xADL [29] e xACME [96]. Contudo, essas linguagens são utilizadas praticamente para documentação de arquiteturas tendo pouco ou nenhum suporte a análise de estruturas arquiteturais, realizando apenas validação na sintaxe da arquitetura.

Xarch, desenvolvida na Universidade da Califórnia, oferece uma forma de representar conceitos arquiteturais através de XML. XADL, atualmente na versão 2.0, é uma aplicação

de Xarch desenvolvida em conjunto pela Universidade da Califórnia e pela Universidade de Carnegie Mellon. xADL é um conjunto de XML Schemas para fornecer, dentre outras características, a utilização de versões e a criação de famílias de arquiteturas. Contudo essas linguagens não possuem ferramentas que venham a auxiliar na avaliação das arquiteturas especificadas com elas.

xACME é uma extensão de xArch para a linguagem ACME, onde os conceitos de propriedades e restrições são adicionados. Quanto à disponibilização de ferramentas, xACME se apresenta como uma representação interoperável utilizada pelo AcmeStudio a partir da sua versão 6.0, contudo, mesmo sendo interoperável, as especificações xACME não são suportadas por nenhuma outra ferramenta de análise arquitetural disponível atualmente.

Um outro fator negativo de todos esses trabalhos é a forma monolítica como essas linguagens apresentam os conceitos de arquitetura. De fato, toda a estrutura, desde os tipos de componentes e conectores como também as instâncias e conexões, são modeladas em uma mesma especificação. Isso dificulta a visualização da especificação além de não fornecer suporte ao desenvolvimento distribuído.

Além de tudo, as ferramentas disponíveis para essa arquiteturas praticamente inexistem, sendo deixada para os desenvolvedores a tarefa de produzir suas ferramentas sob a justificativa da facilidade de manipulação de estruturas XML. Contudo, pudemos observar ao longo dessa tese, através das ferramentas de DraX, que de fato essas ferramentas podem ser desenvolvidas, mas para que possamos utilizar todo o potencial de uma ADL a partir de uma especificação XML devemos lançar mão de dezenas de scripts e esquemas que, sem o auxilio de quem realmente entenda mais profundamente dos conceitos de arquiteturas de software, o trabalho de desenvolvimento dessas ferramentas não é tão simples assim.

ArchML, a ADL de DraX, foi publicada pela primeira vez em 2001 [105] e possui uma gramática muito mais simples que as outras ADLs que usam XML. Esse fato se deve ao suporte fornecido pelos scripts e esquemas de DraX para a validação de arquiteturas especificadas em ArchML. Por meio desses scripts e esquemas, podemos avaliar a especificação sem ter que utilizar construtores como IDs e IDREFs de XML que tornam a especificação bem mais complexa de se entender.

Um outro diferencial de ArchML é a sua conexão com especificações de estilos. Nenhuma das outras ADLs baseadas em XML possui essa característica. Utilizamos a idéia de múltiplas interfaces e da criação de papéis para cada interface de forma a poder capturar a idéia de estilos arquiteturais. Além disso, ArchML se propõe a construção de arquiteturas de forma distribuída.

3.2.3 Descrição Arquitetural em UML

A adoção de UML como forma de se especificar arquiteturas de software tem sido observada atualmente. Contudo, a intenção principal de UML com relação ao projeto de software se dá pela elaboração de projetos baseada em notações que fornecem diferentes visões dos sistemas de software.

Diversos trabalhos [68, 118, 99, 94] têm avaliado UML como uma possível linguagem de descrição de arquitetura. Esses artigos têm concluído que a notação padrão de UML pode ser bem aplicada para a descrição de aspectos específicos das arquiteturas mas falham na modelagem de outros [68].

Outras abordagens fornecem extensões de UML para modelar melhor os conceitos de arquitetura de software. Em [118] por exemplo, é proposta uma mudança no meta-modelo de UML de forma a poder, quando for necessário, realizar alterações nas especificações UML de

Código 3.2 Exemplo em ArchJava.

```
public component class Parser{
public port in {
   provides void setInfo(Token symbol, SysTabEntry e);
   requires Token nextToken() throws ScanException;
}
public port out {
   provides SymTabEntry getInfo(Token t);
   requires void compile(AST ast);
public void parse() {
   Token tok = in.nextToken();
   AST ast = parseFile(tok);
   out.compile(ast);
   ...
}
```

forma a gerar representações XMI passíveis de tratamento, contudo a linguagem resultante é incompatível com as ferramentas UML disponíveis.

Outros trabalhos [68] propõem a extensão de UML com estereótipos e restrições, que são compatíveis com a maioria das ferramentas UML disponíveis. Contudo, os próprios autores concluem que mesmo assim não foram capazes de acomodar as características requeridas por uma ADL.

Em DraX também utilizamos UML para representar o comportamento de componentes. Utilizamos DDP (Diagramas de Descrição de Protocolos) que é uma representação de máquina de estado UML usando algumas regras específicas. Esses diagramas devem ser utilizados para produzir especificações XMI que são tratadas posteriormente na verificação de aspectos comportamentais das arquiteturas. Um fato importante é que os DDPs são completamente compatíveis com todas as ferramentas UML que dão suporte a diagramas de estados. Além disso, não é apenas UML que é utilizado para descrever a arquitetura, de fato DraX possui um conjunto de ferramentas que, em conjunto, fornecem as características necessárias para especificar, validar, analisar e implementar arquiteturas de softwares e estilos arquiteturais distribuídos.

3.2.4 Descrição Arquitetural em Java

Todas as linguagens apresentadas anteriormente são utilizadas apenas para a especificação de arquiteturas. Nenhuma delas chega até ao desenvolvimento do código da aplicação em si. Uma abordagem oposta considera a implementação da arquitetura utilizando os conceitos arquiteturais sem realizar a especificação a priori, mas dentro do próprio código de implementação. Essa abordagem, adotada na linguagem ArchJava [6], permite a implementação de aplicações Java utilizando diretamente os conceitos de arquitetura de software, como portas e conexões. O Código 3.2 apresenta um trecho de código simples de uma aplicação ArchJava.

Mesmo facilitando a implementação de arquiteturas de software em Java, ArchJava possui a limitação de ser demasiadamente concreta, ou seja, os conceitos devem ser expressos em Java, que é uma linguagem de programação, o que torna difícil a visualização abstrata da arquitetura.

Em DraX, fornecemos scripts de geração de código a partir das especificações ArchML e Xtyle. Nessa tese, geramos códigos Java/CORBA, contudo, os scripts podem ser facilmente modificados para gerar qualquer outra linguagem.

3.3 O Modelo de Componente CORBA

O Modelo de Componentes CORBA (CCM) [50] é uma especificação que padroniza o processo de projeto, desenvolvimento, empacotamento, instalação, configuração e execução de componentes CORBA. O CCM, além de resolver as limitações do modelo de objeto CORBA, melhora a produtividade de software e permite a reutilização de código de uma maneira eficiente.

3.3.1 Estrutura do CCM

A especificação CCM está estruturada em cinco modelos:

- Modelo Abstrato - extensões da IDL CORBA oferecem aos desenvolvedores meios para definição de interfaces usadas e oferecidas pelos componentes;
- Modelo de Programação - uso da CIDL (Linguagem de Definição para Implementação de Componentes) para definir como a parte funcional (a lógica do negócio que é programada) e a não funcional (gerada automaticamente) irão interagir entre si;
- Modelo de Execução - descreve o ambiente de execução das instâncias dos componentes;
- Modelo de Empacotamento - especifica como os tipos de componentes e implementações devem ser empacotados;
- Modelo de Instalação - define um processo que permita que alguém instale e configure uma aplicação em vários locais de execução de uma forma simples e automatizada;

3.3.2 O Modelo Abstrato

Com este modelo o projetista tem uma visão geral dos componentes, suas propriedades, que operações ele oferecerá e que operações usará de outros componentes. Facilita a análise de como os componentes se relacionam com outros componentes e sua interação com o lado cliente de uma aplicação e vice versa.

Componente é o tipo básico deste modelo. Para se definir tipos de componentes é usado a nova IDL 3.0 que é formada de extensões à IDL 2.0 do modelo CORBA e tem como propósito o projeto do componente. Para realizar a implementação do mesmo deverá ocorrer um mapeamento para a IDL 2.0 que será então compilado e gerado stubs no lado cliente e os skeletons no lado servidor.

Existem dois níveis de componentes: básico e estendido. Componentes básicos oferecem, essencialmente, um simples mecanismo para "componentizar" objetos CORBA comuns sem adicionar muitas funcionalidades. Componentes estendidos, contrariamente, proporcionam um conjunto mais rico de funcionalidades do que o existente no modelo CORBA.

Para que haja interação entre componentes ou entre componentes e outros elementos CORBA a especificação definiu mecanismos chamados de portas. São definidos os seguintes tipos de portas.

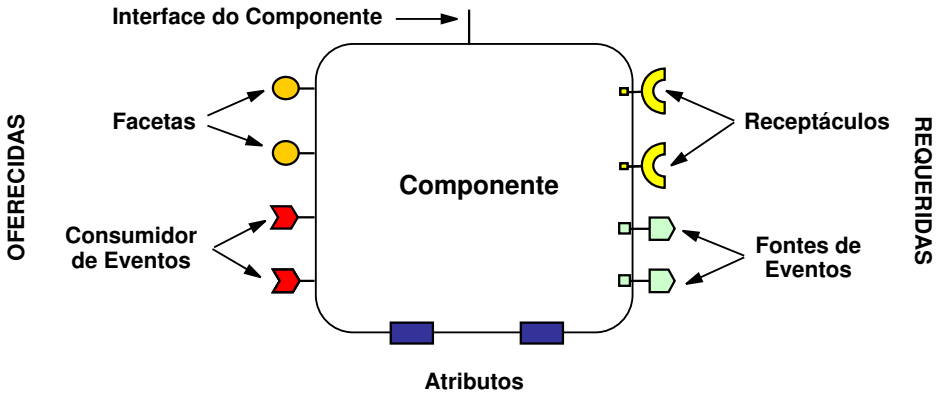


Figura 3.1: Um Componente CORBA.

- Facetas : é por esta interface onde os componentes oferecem suas operações. As implementações destas operações são encapsuladas pelo componente e não estão acessíveis aos clientes.
- Receptáculos : Usando estas interfaces um componente pode invocar operações oferecidas por outros componentes.
- Consumidor de evento : É o ponto de conexão por onde um componente recebe notificações de um determinado tipo de evento, originadas de fontes arbitrárias.
- Fonte de evento : É um ponto de conexão usado pelo componente para emitir eventos de um determinado tipo.
- Atributos : tem como função principal permitir a configuração do componente.

Componentes básicos só oferecem atributos, enquanto os estendidos oferecem todos os tipos de portas. Uma determinada instância de um componente pode ser identificada por uma referência de objeto chamada de interface equivalente do componente. Esta interface permite que clientes naveguem pelas facetas do componente e se conectem em suas portas. A Figura 3.1 ilustra o conceito de componente.

3.3.3 O Modelo de Programação

O Framework de Implementação de Componentes (CIF) define o modelo de programação para a construção de implementações de componentes. O CCM inclui uma linguagem declarativa chamada de Linguagem de Definição para Implementação de Componentes (CIDL) para descrever implementações de componentes e seus estados abstratos. Os compiladores CIDL usam as descrições CIDL para gerar as implementações de skeletons que automatizam muito dos comportamentos básicos dos componentes, tais como : navegação, ativação, gerenciamento de estados, gerenciamento de ciclo de vida etc (parte não funcional). Os desenvolvedores escreverão então a lógica do negócio (parte funcional) que será embutida nos skeletons para completar a implementação dos mesmos. É objetivo do CIF descrever como a parte funcional e a não funcional deverão interagir entre si.

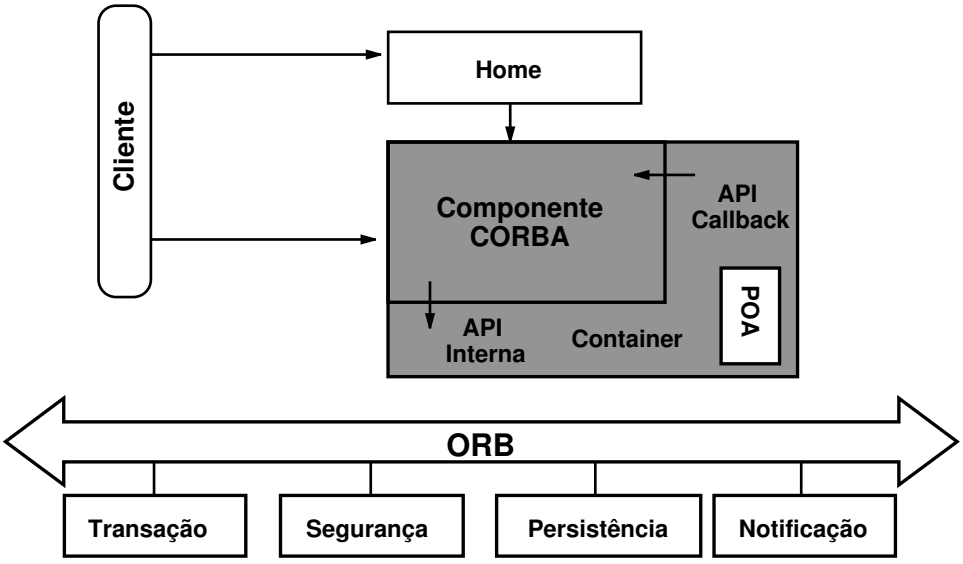


Figura 3.2: Arquitetura de um *Container* CCM.

3.3.4 O Modelo de Execução

O `Container` é o ambiente de execução para uma implementação de um componente CORBA. Este ambiente é implementado por um servidor de aplicação que proporciona um ambiente robusto projetado para suportar um grande número de usuários simultâneos. O Container oferece interfaces para que os componentes acessem os serviços CORBA (transação, segurança, eventos e persistência), sendo o único contato exterior que um componente pode ter, sendo responsável por realizar o acesso ao ORB e ao adaptador de objetos quando necessário.

Componentes e *containers* interagem via interfaces que são definidas em IDL. Interfaces oferecidas pelo componente e usadas pelo *container* são chamadas de interfaces *callback* e as interfaces oferecidas pelo container e usadas pelo componente são chamadas de interfaces internas. A Figura 3.2 ilustra a arquitetura geral deste modelo.

Um servidor de *container* é um processo que hospeda um número arbitrário de *containers* e componentes. Um gerente é responsável pela criação e destruição de *containers*. Cada tipo de *container* inclui um adaptador de objetos POA especializado. A criação de *containers* e POAs, configuração de políticas e o uso dos serviços CORBA, além de outras atividades, são definidas baseadas em arquivos contendo descrições de instalação (*deployment descriptors*). Um POA é usado para criar referências para serem exportadas para os clientes, além de ser utilizado para gerenciar a ativação de instâncias quando as requisições forem recebidas. A criação de um *container* implica, geralmente, na criação de um POA que este container usará. Esta criação é decomposta em quatro passos:

1. Criação das políticas do POA definidas pelo tipo de implementação do container;
2. Criação do POA usando as políticas;
3. Atribuição de um ServantManager para o POA e,
4. Ativação do POA.

3.3.5 O Modelo de Empacotamento

Depois que o desenvolvedor termina de implementar seu componente é preciso prepará-lo para instalação. A primeira tarefa a ser feita é empacotar o componente seguindo o modelo definido na especificação. Um pacote nada mais é que um arquivo que agrupa um tipo de componente, uma ou várias implementações deste tipo e um conjunto de descritores. Estes descritores são escritos na linguagem OSD (*Open Software Descriptor*). Um pacote de software é feito de um descritor e um conjunto de arquivos. Estes vários elementos estão agrupados num arquivo compactado (zip). Este tipo de pacote é usado para instalar qualquer pacote de software.

No contexto CCM um pacote inclui implementações de componentes. Um descritor define o conteúdo do pacote. Ele dá informações gerais sobre peças de software e também detalhes da implementação. Este descritor também enumera as dependências relacionadas com o ambiente da implementação do componente.

O descritor de um pacote de componente especifica as características do componente definidas nas fases de projeto e desenvolvimento. Ele é gerado parcialmente pelo compilador IDL3 e modificado parcialmente por uma ferramenta de empacotamento. Este descritor inclui as várias características de um componente.

3.3.6 O Modelo de Instalação

Este modelo é outra grande contribuição do modelo CCM. Com ele fica mais fácil executar a complicada tarefa de distribuição e instanciação de componentes que formam uma aplicação distribuída.

Este processo deve ser realizado por uma ferramenta pertencente à uma plataforma CCM na qual a aplicação foi implementada. Ela distribuirá e instalará os componentes nos hosts escolhidos. É um cliente de objetos que estão nos locais escolhidos para as instalações dos componentes.

Os cinco passos básicos para realizar a tarefa de distribuição, instalação e configuração de uma aplicação são:

1. Definir e escolher os locais de execução.
2. Instalar as implementações onde for preciso.
3. Instanciar servidores e containers.
4. Instanciar componentes.
5. Conectar e configurar componentes.

Como podemos observar, os conceitos fornecidos pelo Modelo de Componentes CORBA se aproximam bastante da nossa abordagem com DraX no que diz respeito à geração de aplicações CORBA. Contudo, DraX permite a especificação arquitetural baseada na idéia de estilos arquiteturais e só a partir dessas especificações é que os códigos Java/CORBA são gerados, inclusive poderíamos ter optado por gerar Java/CCM em DraX, mas o ferramental disponível atualmente para executar implementações CCM deixa bastante a desejar. É válido ressaltar que DraX permite a geração de aplicações em diversas outras infraestruturas de middleware.

Na verdade, por ser CCM um modelo extremamente complexo, ainda não existe uma implementação completa. Com relação a essas implementações podemos destacar o OpenCCM [3] como uma ferramenta inicial que implementa algumas das características básicas de CCM.

3.4 MDA

O OMG introduziu a iniciativa MDA (*Model-Driven Architecture*) [49] como uma abordagem para interoperabilidade e especificação de sistemas baseada no uso de modelos formais. Nesta arquitetura, modelos independentes de plataformas são inicialmente expressos em uma linguagem de modelagem independente de plataforma, tal como a UML. O modelo independente de plataforma é então traduzido para um modelo específico de plataforma, mapeando-se o primeiro para alguma plataforma ou linguagem de implementação por meio de regras formais.

No estudo da MDA estão envolvidos os seguintes padrões OMG:

- UML (*Unified Modelling Language*);
- MOF (*Meta Object Facility*);
- XMI (*XML MetaData Interchange*);
- CWM (*Common Warehouse Metamode*l)

Estes padrões definem a estrutura central da MDA e têm contribuído bastante para o estado da arte em modelagem de sistemas.

MDA é um processo OMG e está sendo apontada como o maior passo evolucionário na forma como este grupo define os padrões de interoperabilidade. Por muito tempo, a interoperabilidade baseou-se largamente nos padrões e serviços CORBA, o qual propõe que os sistemas de software heterogêneos interoperem em nível de interfaces de componentes padrão. Na MDA, por sua vez, é proposto o uso de modelos de sistemas formais como solução para o problema da interoperabilidade.

O ponto mais importante nesta abordagem é a independência da especificação do sistema em relação a uma plataforma ou tecnologia de implementação específica. A definição do sistema existe independente de qualquer modelo de implementação, possuindo mapeamentos formais para infra-estruturas de plataformas, tais como Java, XML, SOAP, etc.

Os padrões centrais da MDA propõem-se a formar a base de construção de esquemas para autoria, publicação e mapeamento de modelos dentro da arquitetura. Atualmente, existe um interesse da indústria de software pela realização dos padrões centrais da MDA na plataforma Java, ou seja, o desenvolvimento de padrões de mapeamento de modelos independentes de plataforma para modelos dependentes, no qual o modelo dependente é a plataforma Java. Esta é uma estratégia de implementação sensível, pois o desenvolvimento e integração são facilitados através de serviços de plataforma e modelos de programação comuns (interfaces e APIs), fornecidas como parte da plataforma Java.

A evolução da MDA divide-se em uma visão a curto prazo e uma a longo prazo. A primeira consiste no alcance da interoperabilidade baseada nas traduções dos modelos independentes de plataforma para os modelos dependentes de plataforma, e no compartilhamento de metadados. As tecnologias de suporte estão largamente especificadas e implementações estão sendo construídas por diversas organizações mundiais. A visão de longo prazo, baseia-se no amplo crescimento da distribuição de AOMs, como uma evolução da MDA, e ainda está sendo conceitualizada.

A visão de curto prazo propõe um ambiente na qual a interoperabilidade eficiente e quase sem retalhos entre as diversas aplicações, ferramentas e bancos de dados, seja alcançada através do intercâmbio de modelos compartilhados. Componentes participam desse

ambiente alavancando serviços padrões oferecidos pelas implementações dos padrões MDA, que os permite expor e trocar seus metadados como instâncias de modelos bem definidos. Esses serviços de plataforma têm definições padronizadas que são expressas por modelos de programação padronizados, os quais são automaticamente gerados a partir de modelos independentes de plataforma.

Os metadados são críticos em todos os aspectos da interoperabilidade em ambientes heterogêneos. Na realidade, metadado é o meio principal para atingir a interoperabilidade. A interoperabilidade é largamente facilitada por APIs padrões, mas atualmente vêm sendo necessário que as definições da semântica e capacidades dos sistemas sejam expressas como metadados compartilhados. Um sistema baseado em MDA deve ser capaz de armazenar, gerenciar e publicar metadados a nível de sistema e aplicação, incluindo descrições do próprio ambiente. Aplicações, ferramentas, bancos de dados e outros componentes registram-se no ambiente e descobrem descrições de metadados pertencentes ao mesmo. Similarmente, um componente ou produto colocado dentro do ambiente também pode publicar seus próprios metadados para o resto do ambiente.

Um sistema baseado em MDA não exige que representações internas dos metadados dentro das aplicações sejam modificadas para corresponder às definições compartilhadas. Os metadados compartilhados consistem de definições externalizadas que são trocadas entre os componentes comunicantes. Estas definições são prontamente entendidas pelos componentes que concordam no metamodelo, descrevendo os metadados.

Definições externas são altamente genéricas, mas também possuem completude semântica quanto ao domínio do problema ao qual os componentes são aplicados, e são, desta forma, entendidas por um grande número de participantes. Os metadados altamente específicos de produto e que não contemplam o modelo genérico são manipulados através do uso de mecanismos de extensão que são pré-definidos como parte dos modelos genéricos. Isto implica no uso de mecanismos de extensão UML tais como *tagged values*, estereótipos e restrições.

Para garantir que os metadados compartilhados sejam prontamente compreendidos por todos os componentes participantes, um sistema baseado em MDA requer que tais componentes tenham padronizadas as seguintes características:

- Uma linguagem formal, quanto à sintaxe e à semântica, para representação dos metadados;
- Um formato de intercâmbio para troca e publicação de metadados;
- Um modelo de programação para acesso e descoberta de metadados. Isto inclui capacidades de programação genéricas para executar metadados de natureza desconhecida;
- Mecanismos para extensão;
- Um serviço de metadados opcional, no qual os metadados publicados residem.

Além dos metadados compartilhados, um outro bloco de construção para interoperabilidade de sistemas é a padronização de serviços comuns em nível de aplicação e de sistema. Uma API padrão define um modelo de programação padrão dos serviços que ela representa. Esta forma de padronização simplifica os clientes e facilita a integração de novos componentes dentro do ambiente baseado em MDA. Clientes usam serviços comuns que consomem menos memória do sistema e são menos complexos, pois só necessitam expor sua interface, independente de como

são realmente implementados em uma plataforma particular. Por outro lado, os fornecedores de serviços implementam APIs padrões que estão prontamente disponíveis para uso por um grande número de clientes.

O último bloco de construção da visão de curto prazo MDA é a especificação da plataforma. Esta representa a completa definição de metadados para interoperabilidade e intercâmbio de estratégias, serviços comuns e APIs padrões que toda instância de um sistema baseado em MDA seja capaz de suportar. Cada instância inclui um descritor que especifica aquelas características realmente suportadas por uma distribuição particular do ambiente baseado em MDA. Ferramentas de software para especificação e integração de componentes geram o descritor, e as ferramentas para configuração, instalação e inicialização são dirigidas por este descritor.

A visão de longo prazo para arquiteturas de sistemas baseadas em MDA inclui software capaz de automatizar descoberta de propriedades de seu ambiente e adaptação para este ambiente por vários meios, incluindo modificações dinâmicas de seu próprio comportamento. Esta é uma visão ambiciosa que constrói-se significativamente em experiências e idéias obtidas a partir da implementação da visão de curto prazo.

A funcionalidade do sistema gradualmente se tornará mais orientada pelo conhecimento e a capacidade de descobrir automaticamente propriedades comuns de domínios semelhantes, tomando decisões inteligentes baseadas naquelas descobertas e em deduções resultantes do armazenamento e esboço efetuados. Em geral, este conhecimento é suportado por um avançado e altamente evoluído conceito de metadados unipresente, no qual a habilidade de usar este conhecimento em tempo de execução é fornecida através de Modelos de Objetos Adaptativos (AOM).

Ainda, segundo POOLE [91], a habilidade dos desenvolvedores de produzir tais sistemas será tão grande quanto for o resultado das experiências extensivas destes com o uso de metamodelos e ontologias no comportamento de sistemas e na tomada de decisões. O autor afirma que os desenvolvedores aprenderão como construir sistemas nos quais uma quantidade considerável de conhecimento do domínio é colocada dentro de elevados níveis de abstração, nos quais os sistemas estarão preparados para extrair e usar as informações eficientemente.

DraX compartilha alguns dos objetivos principais de MDA que é permitir se produzir software de uma forma mais abstrata e que as implementações possam ser geradas a partir dessas abstrações. Contudo, MDA é bastante complexa e não apresenta as características requisitadas para a adoção de estilos arquiteturais apresentadas em DraX.

3.5 Outros Trabalhos

Diversas outros trabalhos pontuais podem ser considerados relacionados à nossa abordagem. A Programação Orientada a Aspectos (*Aspect Oriented Programming* - AOP) [65] é uma delas. A AOP foi desenvolvida com a idéia de propos linguages de programação com baseadas nos conceitos de separação de interesses. Assim, aspectos diferentes de programação são considerados separadamente e então compiladores atuam na criação de um sistema único com os aspectos programados. Em DraX, também tratamos aspectos de separação de interesse no momento que separamos a especificação da arquitetura da separação do estilos. Mesmo havendo essa separação, que é uma contribuição original desse trabalho, ao tratamento da arquitetura é realizada de forma sistêmica considerando todos os aspectos especificados.

Observamos também a adoção da idéias de arquitetura na criação de aplicações CORBA, apresentado em [74]. Nesse trabalho, foi utilizada a linguagem Darwin como uma opção

para a especificação de aplicações que foram implementadas em CORBA. Na verdade essa especificações tiveram que ser traduzidas manualmente para CORBA, sem ser realizado nenhum tratamento de estilos ou verificação de nenhum tipo. Em DraX, a infraestrutura de middleware utilizada não afeta na descrição da arquitetura. Assim, de forma geral, uma arquitetura pode ser descrita e sua implementação é gerada posteriormente sem a necessidade de criar restrições nos códigos nem mapeamentos manuais de especificações de interfaces.

3.6 Conclusão

Os trabalhos relacionados apresentados nessa seção são, de fato, trabalhos pontuais que de alguma forma possuem características em comum com nossas propostas. Não apresentamos nenhum trabalho que tenha os mesmos objetivos gerais simplesmente por que esses não existem. Dessa forma, resolvemos apresentar aspectos mais pontuais de forma a pudermos traçar paralelos entre as contribuições dessa tese e outros trabalhos realizados em áreas relacionadas.

Capítulo 4

O *framework* DraX

4.1 Introdução

Como apresentamos no Capítulo 1, as ADLs disponíveis atualmente não fornecem as características necessárias para que possam ser adotadas sem dificuldades no desenvolvimento de projetos de softwares distribuídos. De fato, a grande maioria das ADLs não se adequam à realidade da maioria dos desenvolvedores, tanto com relação ao perfil de conhecimento desses desenvolvedores como às ferramentas que esses já estão acostumados a utilizar corriqueiramente em seus projetos de software.

Além disso, mesmo que venhamos a supor que uma equipe de desenvolvimento venha a tentar adotar alguma das ADLs existentes para a produção de projetos arquiteturais para aplicações distribuídas, outros problemas podem surgir, como por exemplo, as restrições dessas ADLs para a descrição de arquiteturas baseadas em estilos arquiteturais e a própria definição de novos estilos, a conexão das especificações arquiteturais com as ferramentas de implementação das aplicações distribuídas, entre outros.

Em [30] é demonstrado que as ADLs disponíveis atualmente não fornecem mecanismos satisfatórios para produzir especificações arquiteturais que possam ser facilmente implementadas sobre infraestruturas de middleware, visto que esses infraestruturas induzem diversas características que não são capturadas pelas ADLs. Ao mesmo tempo, temos que os desenvolvedores de software atualmente estão utilizando extensivamente infraestruturas de middleware como CORBA e RMI para facilitar a construção de aplicações distribuídas [31].

Baseado nessas idéias, apresentamos nesse capítulo o *framework* DraX. DraX (*DistRibuted Architecture based on XML*) é um conjunto de linguagens, esquemas e *scritps* que podem ser utilizados em conjunto para permitir a descrição, validação, análise e implementação de arquiteturas de software e de estilos arquiteturais distribuídos sobre uma infraestrutura de middleware. Nossa principal intenção com DraX é mostrar que é possível utilizar as ferramentas que realmente fazem parte do dia-a-dia dos desenvolvedores para produzir projetos arquiteturais distribuídos com as mesmas características dos produzidos por ADLs tradicionais. Além disso, *scripts* de DraX podem ser utilizados para gerar códigos que formam *templates* de implementação das especificações que são produzidos com base na idéia de implementação da arquitetura sobre alguma infraestrutura de middlware.

4.2 Requisitos para o *framework* DraX

Além das óbvias características necessárias para a elaboração de projetos arquiteturais disponíveis em quase todas as ADLs (para uma descrição detalhada sobre essas características consultar [80]), resolvemos incluir em DraX um conjunto de características que pudessem tanto facilitar a utilização dos conceitos de arquitetura de software e de estilos arquiteturais por profissionais da indústria de software que não estão familiarizados com as notações das ADLs, como também fornecer mecanismos para que as especificações possam ser facilmente implementadas em alguma arquitetura de middlaware sem que seja necessário o domínio dessa arquitetura pelos desenvolvedores.

Dessa forma, diversos requisitos devem ser atendidos pelas linguagens, *scripts* e esquemas de DraX para conseguir atingir os objetivos que traçamos. Dentre esses requisitos podemos destacar os seguintes:

1. **Descrição da Arquitetura do Software Distribuído de forma Distribuída**:
   Esse item pode parecer um pouco estranho, mas de fato se observarmos na literatura técnica sobre arquiteturas de software distribuído [66, 62, 72, 63], podemos observar que esse conceito se aplica à descrição arquitetural de um software distribuído. Nessa tese, propomos a idéia de construir essa descrição arquitetural também de forma distribuída [104], no sentido de que os componentes que formam a arquitetura podem ser criados de forma distribuída sendo referenciados no momento da definição da arquitetura. A ADL que propomos em DraX, que denominamos ArchML [105, 110, 104], possui a característica de permitir a descrição separada, e possivelmente distribuída, dos componentes que formam a arquitetura.

2. **Descrição de Estilos Arquiteturais**:
   Convencionalmente a utilização de estilos arquiteturais nas ADLs se dá pela utilização de descrições arquiteturais parametrizadas [30], como em Rapide [69], Darwin [72] e Wright [7]. Diferente dessa abordagem, em DraX, para conseguirmos uma melhor separação de interesses [56] na especificação de estilos e arquiteturas, desenvolvemos uma linguagem específica para a descrição de estilos arquiteturais, que denominamos Xtyle [114, 105]. Com Xtyle podemos construir novas especificações de estilos, refinar especificações existentes e até mesmo criar especificações baseadas em informações sintáticas de outros estilos, herdando algumas de suas características.

3. **Descrição de Comportamentos**:
   Uma das vantagens de se utilizar uma ADL é a possibilidade de se realizar algum tipo de análise na arquitetura antes de sua implementação. As ADLs que permitem uma maior gama de análises são as que utilizam algum tipo de formalismo matemático na representação dos componentes das arquiteturas ou na sua própria definição semântica. Como exemplos desse tipo de ADL podemos citar Wright [7], que usa CSP (*Communicating Sequencial Processes*) [55] para permitir a descrição da semântica formal de componentes e Darwin [72], que usa π-cálculo para analisar sistemas distribuídos baseados em troca de mensagens. Contudo, o uso desses formalismos de fato distancia as ADLs da imensa maioria dos desenvolvedores de software e quase que as confina ao mundo acadêmico. Em DraX, resolvemos capturar a idéia de descrição de comportamento, tanto de componentes como de estilos, propondo uma variação de diagramas de estados UML e produzindo *scripts* que geram especificações em uma álgebra de processos, o cálculo $\mathcal{R}\pi$ [111, 117, 106, 108, 109], que criamos para fornecer

semântica às linguagens de DraX. Dessa forma, possibilitamos a realização de análises formais sem ter que se escrever complexas expressões algébricas.

4. **Utilizar um ORB para representar Conectores**:
   A intenção de DraX é produzir projetos de arquiteturas distribuídas passíveis de implementação sobre algum middleware[1]. Dessa forma, resolvemos não utilizar definição explícita de conectores nas linguagens de DraX, visto que implicitamente as especificações serão implementadas tendo um ORB como representação desses conectores. Desse modo, podemos dizer que ArchML é uma ADL que utiliza a idéia de configuração *in-line* [81]. Como a intenção de DraX é produzir software distribuído, e para isso devemos lançar mão de algum middleware, a adoção de um ORB para representar os conectores é uma solução adequada, mesmo restringindo as especificações às funcionalidades do ORB subjacente. Contudo, se por exemplo, utilizarmos um ORB CORBA, ainda assim temos uma gama razoável de estilos de comunicação, que podem ser síncrono, assíncrono, baseado em mensaens, baseado em eventos, que por sua vez, pode seguir o modelo *push* ou *pull*, entre outros. Para fortalecer ainda mais esse conceito, são apresentados em [36] argumentos que explicam que para a maioria das aplicações distribuídas baseadas em middleware é completamente desnecessário a descrição explícita desses concetores. Contudo, deve-se ficar claro as limitações de comunicação relativas ao middleware subjacente.

5. **Utilizar apenas Software Gratuito**:
   Resolvemos utilizar como tecnologias para suportar as ferramentas de DraX apenas softwares gratuitos. Com isso queremos mostrar que não é necessário nem mesmo se ter custos extras com software para que tenhamos todas as vantagens da adoção da idéia de projeto arquitetural e de implementação de aplicações distribuídas utilizando middlwares. É importante ressaltar nesse ponto que resolvemos adotar diversos softwares em DraX, que serão utilizados para escrever os códigos, executar *scripts* etc, apenas por motivos didáticos, pois poderíamos ter produzido uma ferramenta que tornasse tudo isso transparente (esse é um trabalho futuro). Contudo, limitaríamos o poder de DraX como ferramenta de ensino de arquitetura de software.

Na seção a seguir os componentes do *framework* DraX são apresentados. Esses componentes fornecem uma arquitetura que considera todos os requisitos citados aqui. Nessas seções cada um dos componentes de DraX são apresentados de forma detalhada.

4.3 Componentes do *framework* DraX

De forma a contemplar os requisitos propostos na seção anterior, o *framework* DraX é formado por um conjunto de linguagens e ferramentas que dão suporte a todo o processo de desenvolvimento baseado em arquitetura, que vai desde a especificação da arquitetura e do estilo arquitetural, passando pela fase de verificação de especificações até a chegar à geração de código.

Para facilitar a apresentação do ferramental disponibilizado por DraX subdividimos o *framework* em três partes, a saber:

[1] Ao longo dessa tese utilizamos o termo ORB (*Object Request Broker*) para identificar a implementação do modelo de comunicação dos middlwares.

- **Linguagens de Especificação**: Reúne as linguagens disponibilizadas por DraX para a especificação tanto de arquiteturas de software distribuídas como de estilos arquiteturais;
- **Esquemas e *Scripts* de Validação e Verificação de Consistência**: Reúne um conjunto de esquemas e *scripts* para a validação sintática e estrutural de especificações de componentes e arquiteturas e estilos arquiteturais, como também de conformidade de arquiteturas com relação à estilos. Também estão disponíveis *scripts* específicos para a verificação de propriedades comportamentais em arquiteturas e em estilos arquiteturais; e
- ***Scripts* de Geração de Código**: Esses *scripts* permitem a geração de código Java baseados no middleware escolhido para a comunicação a partir das informações das especificações de arquiteturas e estilos arquiteturais.

A seguir cada uma das partes de DraX é melhor detalhada.

4.3.1 Linguagens de Especificação

Definimos duas linguagens em DraX, a primeira, denominada ArchML [105, 107, 115], tem a função de descrever componentes e arquiteturas de aplicações distribuídas; a segunda, denominada Xtyle [110, 108] para descrever estilos arquiteturais. Essas linguagens são aplicações de XML e possuem características específicas apresentadas a seguir.

4.3.1.1 Linguagem de Descrição de Componentes e Arquiteturas

Tradicionalmente as ADLs se preocupam com aspectos de projeto de arquitetura e não tratam a representação de arquiteturas em execução [29]. Em DraX, resolvemos utilizar os dois aspectos para representar as arquiteturas [2]. Desse modo, tanto é possível descrever os componentes em DraX, como a arquitetura gerada por instâncias desses componentes. Para permitir essa característica, usamos duas linguagens, uma para representar apenas os componentes e a outra para representar a arquitetura distribuída a partir das instâncias desses componentes. Para facilitar futuras referências resolvemos denominar o conjunto dessas linguagens de ArchML, que é apresentada em detalhes na seção 5.3.

A descrição tanto de um componente como de uma arquitetura ou estilo utilizando uma linguagem baseada em XML requer a definição a priori dos elementos XML que serão utilizados para representar essas informações e da forma como esses estão relacionados. Para definir a sintaxe de ArchML e de Xtyle desenvolvemos uma linguagem de padrões [115, 110] a partir de um estudo das características das ADLs atuais. Essa linguagem de padrões foi necessária para facilitar e justificar as escolhas que fizemos tanto para quais informações colocar em ArchML como também com relação à forma que essas informações estão organizadas. Essa linguagem de padrões é apresentada na Seção 5.2.

Para a descrição de componentes em ArchML podemos utilizar os seguintes tipos de informações:

1. **Informações gerais**: Basicamente informações gerenciais sobre o componente, como por exemplo o nome da pessoa que realizou a especificação, a data em que ela foi realizada, a data da última modificação, entre outras. Essas informações são utilizadas para documentar o componente.

[2] Temos a intenção de, em um trabalho futuro, tratar fatores de reconfigurabilidade em ArchML, dessa forma resolvemos representar a arquitetura em execução.

Código 4.1 Exemplo de Componente em ArchML.

```
<?xml version="1.0"?>
<component name="LowerCase">
<document>
   ...
</document>
<propertySet>
   ...
</propertySet>
<interfaces>
   <interface role="Filter»
      ...
   </interface>
</interfaces>
<behavior href="LowerCase.xmi"/>
</component>
```

2. **Propriedades**: É possível se definir um conjunto de propriedades que um componente pode ter. Essas propriedades definem fatores relacionados ao modo de funcionamento das instâncias desse componente. Por exemplo, existe um conjunto de propriedades pré-definidas em ArchML que são relativas a CORBA, como a política de threads do POA, entre outras. Os valores dessas propriedades podem ser manipulados pelo desenvolvedor.

3. **Interfaces**: Um componente deve implementar uma ou mais interfaces. Essas interfaces apresentam um conjunto de portas, onde informações sobre tipos de dados e sentido de comunicação são representadas. A adoção do mecanismo de múltiplas interfaces na descrição de componentes em ArchML permite agrupar portas que determinem comportamentos mais complexos de um componente. Em [14] é apresentado uma discussão sobre o carater restritivo das ADLs que utilizam a definição de um papel (*role*) para um componente relativo a apenas um tipo definido em um estilo. Nesse artigo, que apresenta um exemplo em UniCon, fica demonstrado as vantagens de um componente possuir mais de um *role*. Contudo, mesmo com essa discusão as ADLs atuais ainda não trataram esse problema.

   Em DraX, utilizamos o mecanismo de múltiplas interfaces para a criação de arquiteturas complexas que possuam diversos tipos de interação e possam obedecer a estilos arquiteturais mistos. Essa característica poderá ser melhor entendida quando for apresentada a noção de estilos arquiteturais em DraX, na Seção 5.4.

4. **Comportamento**: o comportamento do componente deve ser descrito através de uma representação XMI de um Diagrama de Descrição de Protocolos, que é uma adaptação que propomos dos diagramas de estados UML para representar a forma com que os dados fluem pelas portas de um componente.

Para mostrar a estrutura geral de uma descrição ArchML, o Código 4.1 apresenta uma especificação simples de um componente. Como comentado anteriormente, ArchML será detalhada na Seção 5.3.

Código 4.2 Exemplo de Arquitetura em ArchML.

```
<?xml version="1.0"?>
<system name="Capitalize"
<document>
   ...
<document>
<style href="Pipeline.sty"/>
<types>
   ...
</types>
<instances>
   ...
</instances>
<links>
   ...
</links>
</system>
```

A descrição de arquiteturas em ArchML é realizada pela definição de instâncias a partir dos componentes descritos a priori e pela definição da estrutura de conexão entre as portas das interfaces dessas instâncias. Para realizar essa tarefa em ArchML devemos descrever a seguintes informações:

1. **Informações Gerais**: As mesmas utilizadas para descrever componentes;
2. **Estilo Utilizado**: A arquitetura desenvolvida deve seguir regras específicas de algum estilo arquitetural, assim, deve ser definido qual estilo está sendo utilizado. Se nenhum estilo for fornecido, será utilizado o valor padrão que indica o modelo Cliente-Servidor.
3. **Contexto**: Os tipos de componentes que participam da arquitetura são definidos como contexto da arquitetura. Esses componentes devem estar especificados;
4. **Instâncias**: Usando referências aos tipos do contexto, são declaradas instâncias de componentes com nomes específicos;
5. **Configuração**: A estrutura de interconexão entre as portas das instâncias é descrita uma a uma.

Para exemplificar a utilização de ArchML na descrição de arquiteturas, o Código 4.2 mostra um trecho de especificação de arquitetura descrita nessa linguagem.

4.3.1.2 Linguagem de Descrição de Estilos Arquiteturais

Uma característica importante e original do *framework* DraX é a possibilidade de se descrever estilos arquiteturais e utilizar essas descrições na construção de arquiteturas distribuídas. Para realizar essa tarefa, desenvolvemos a linguagem Xtyle, que serve especificamente para a descrição de estilos arquiteturais no *framework* DraX. Essa linguagem permite definir as seguintes características dos estilos:

1. **Informações Gerais**: Basicamente são as mesmas informações utilizadas na descrição de componentes em ArchML;

2. **Estilo Base**: Seguindo a idéia de composição e refinamento de estilos [3] em Xtyle podemos definir um estilo derivado a partir de outros estilos;

3. **Tipos**: Os tipos de componentes que um determinado estilo possui devem ser definidos quando da descrição de um estilo. Isso permite a definição de um vocabulário para esses estilos. Cada tipo deve possuir um conjunto de portas com um conjunto de propriedades para cada porta, que define informações como por exemplo, o modo de comunicação ou o número máximo de conexões permitidas. Cada tipo também possui a definição de um comportamento esperado para os componentes de cada tipo com relação a utilização das suas portas;

4. **Topologia**: A relação de comunicação entre os componentes é descrita nessa seção, assim como o fluxo de controle e de dados e as restrições topológicas do estilo.

O Código 4.3 apresenta um trecho da especificação de um estilo usando Xtyle. Xtyle é apresentada em detalhes na Seção 5.4.

4.3.2 Esquemas e *Scripts* de Validação

As linguagens ArchML e Xtyle permitem a descrição de complexas arquiteturas e de estilos arquiteturais distribuídos. Entretanto, uma tarefa importante é validar essas especificações tanto sintaticamente e estruturalmente como comportamentalmente [30]. Em DraX, definimos um conjunto de esquemas e *scripts* para realizar essa tarefa. Esses esquemas e *scripts* de validação estão organizados em duas categorias: esquemas e *scripts* de validação sintática e estrutural e esquemas e *scripts* de validação comportamental, sendo que existem esquemas e *scripts* específicos para tarefas específicas. Os principais tipos de esquemas e *scripts* de DraX são:

- **Esquemas de Validação Sintática**: Esses esquemas foram descritos em W3C Schemas [25] e permitem a verificação sintática das especificações dos componentes das arquiteturas e dos estilos, verificando se as especificações estão sintaticamente corretas com relação à gramática definida;

- ***Scripts* de Verificação de Consistência Estrutural de Arquiteturas**: Esses *scripts* foram definidos em Schematron [61] e permitem a verificação da compatibilidade sintática entre componentes distribuídos que formam uma arquitetura. Fatores como tipos de portas, sentido de comunicação, tipos de dados retornados, entre outros, podem ser validados com esses *scripts*;

- ***Scripts* de Verificação Comportamental**: Esses *scripts* foram definidos em XSLT [18] e permitem a geração de especificações π-cálculo a partir da definição de comportamento de componentes e estilos. Na verdade, esses *scripts* recebem arquivos XMI e retornam a representação π-cálculo correspondente. Essas especificações devem ser utilizadas para verificar informações comportamentais nas arquiteturas tal como a presença de impasses [4].

[3] Por composição de estilos definimos a tarefa de se criar novos estilos a partir de outros estilos pré-definidos através da herança de especificações, sendo que o refinamento se dá pela modificação de especificações herdadas.

[4] Do inglês *deadlock*.

Código 4.3 Exemplo de Estilo Arquitetural em Xtyle.

```
<?xml version="1.0"?>
<xtyle name="DataFlowNetwork"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.cin.ufpe.br/~cts/ArchML
        ...\Xtyle.xsd">
<document>
   ...
<document>
<types>
   <type name="Filter">
      <ports>
         <in mode="async"/>
         <out mode="async"/>
      </ports>
      <behavior href="Filter.xmi"/>
   </type>
      <type name="Source">
         <ports>
         <out mode="async"/>
      </ports>
      <behavior href="Source.xmi"/>
   </type>
   <type name="Sink">
      <ports>
         <in mode="async"/>
      </ports>
      <behavior href="Source.xmi"/>
   </type>
   ...
</types>
   <topology>
   <link name="SourceFilter" start="Source" end="Filter"
         controlType="push"/>
   ...
   </topology>
</xtyle>
```

Os esquemas de validação sintática de componentes, arquiteturas e estilos serão apresentados juntamente com as respectivas linguagens de especificação, pois esses esquemas definem a gramática para ArchML e para Xtyle. Os *scripts* de validação estrutural de arquiteturas e estilos são apresentados na Seção 6.2. Já os *scripts* de validação comportamental são apresentados na Seção 6.3, juntamente com uma álgebra de processos que propomos para facilitar a verificação de arquiteturas em DraX.

4.3.3 *Scripts* de Geração de Código

O *framework* DraX, além de dar suporte à especificação e validação de arquiteturas e estilos distribuídos, também fornece um conjunto de *scripts* para a geração de templates de implementação baseados em Java contendo o código relativo a um middleware específico. Esses templates são gerados de acordo com as características definidas na arquitetura e no estilo, permitindo a criação de aplicações distribuídas complexas sobre esse middleware. Esses códigos seguem as especificações de um determinado estilo arquitetural que foram previamente testadas com relação a algumas propriedades.

Os templates gerados por DraX, através de *scripts* XSLT [18], aliviam o trabalho tedioso de se manipular a API do middleware subjacente e dos serviços que ele disponibiliza. Assim, se por exemplo estivermos utilizando CORBA como middleware, DraX pode gerar templates com informações necessárias para utilizar serviços como persistência, eventos, nomes, entre outros, que devem ser definidos na especificação da arquitetura ou baseado em propriedades de estilos que essas arquiteturas utilizam.

Esses *scripts* de geração de código de DraX são apresentados no Capítulo 7, onde são descritos a forma com que foram desenvolvidos permitindo que novos serviços sejam incorporados facilmente a DraX, ou que o desenvolvedor possa adaptar os *scripts* já existentes às suas necessidades específicas ou a outras infraestrutura de middleware.

Até aqui fornecemos uma visão geral das ferramentas fornecidas no *framework* DraX para a construção de aplicações distribuídas usando a idéia de arquiteturas e estilos arquiteturais. A seguir apresentamos uma metodologia de desenvolvimento que pode ser utilizada para guiar a especificação dessas arquiteturas distribuídas utilizando o ferramental de DraX. Além disso, essa metodologia ajudará a entender melhor todas as linguagens, esquemas e *scripts* presentes em DraX e as interações entre eles.

4.4 Metodologia de Desenvolvimento

Como apresentamos nas seções anteriores, DraX possui um conjunto de linguagens, esquemas e *scripts* que permitem realizar todas as fases do desenvolvimento arquitetural, desde a especificação das arquiteturas e estilos, passando pela validação sintática, estrutural e comportamental até a geração de templates de implementação. Entretanto, como o ferramental de DraX é relativamente extenso, resolvemos criar uma metodologia que possa guiar os passos de desenvolvimento de arquiteturas com o *framework*.

Inicialmente devemos comentar sobre a utilização de estilos arquiteturais nas arquiteturas desenvolvidas por DraX. Como veremos na especificação de ArchML, não é obrigado se definir um estilo para uma arquitetura. Pelo menos explicitamente, pois, de fato, se não for definido um estilo, será considerada a utilização de um estilo padrão Cliente-Servidor (estilos arquiteturais em DraX são discutidos na Seção 5.4).

Desse modo, podemos visualizar duas etapas de desenvolvimento distintas e paralelos que interagem entre si. Na primeira etapa a arquitetura da aplicação é descrita; na segunda etapa o estilo arquitetural é descrito, caso esse ainda não tenha sido especificado. O resultado do desenvolvimento de um estilo é sua especificação em Xtyle devidamente validada. Essa especificação poderá ser referenciada na etapa de descrição de uma arquitetura ArchML. Onde a tarefa de validação dessa arquitetura, entre outras coisas, verificará a aderência da mesma ao estilo referenciado.

É bom observar que a etapa de desenvolvimento de estilos não é comumente usada, pois DraX possui uma base de estilos pré-definidos que podem ser referenciados para criar arquiteturas, entretanto, é um diferencial de DraX com relação a outras ADLs e/ou ambientes de descrição de arquiteturas, o fato de o desenvolvedor contar com uma linguagem específica para definir seus próprios estilos arquiteturais. Além disso, esses novos estilos tanto podem ser definidos a partir de refinamentos dos estilos pré-existentes em DraX, como podem ser totalmente definidos. Essa contribuição de DraX para a área de estilos arquiteturais é apresentada na seção 5.4.

De qualquer modo, a metodologia geral que propomos para a utilização de DraX para especificação de arquiteturas distribuídas é composta pelas seguintes etapas:

1. **Especificação**: Nessa fase são especificadas as arquiteturas e os estilos arquiteturais utilizando as linguagens ArchML e Xtyle respectivamente;

2. **Validação**: A verificação sintática, estrutural e comportamental das arquiteturas e estilos é realizada nessa fase. Sendo que são utilizados os esquemas e *scripts* de validação e verificação de DraX;

3. **Geração de Código**: Os templates de implementação são gerados a partir das especificações arquiteturais previamente validadas e das informações dos estilos que essas arquiteturas utilizam.

A Figura 4.1 apresenta o diagrama geral de atividades a serem realizadas para se descrever uma arquitetura no DraX. Como pode ser observado, existe um paralelismo entre as etapas de descrição de arquiteturas e de descrição de estilos, como foi explicado anteriormente.

É importante ressaltar nesse ponto que não estamos apresentando um processo de desenvolvimento de software distribuído baseado em arquitetura. Mesmo sabendo que apresentamos uma metodologia de aplicação das ferramentas de DraX para a produção de software distribuído baseado em arquitetura, não nos preocupamos em formalizar os passos e interações relativas a essas ferramentas. Dessa forma, por exemplo, aspectos como refinamento de especificações arquiteturais [39, 33, 34] não são considerados nesse trabalho, mesmo sabendo que essa etapa pode ser realizada por ocasião da construção das especificações ou até mesmo diretamente nos códigos gerados a partir das ferramentas de DraX. [5]

Segue agora um detalhamento sobre cada uma das atividades propostas na Figura 4.1.

4.4.1 Especificar Arquitetura

Na descrição da arquitetura da aplicação em DraX se dá pela utilização da linguagem ArchML. Essa linguagem, que será detalhada na Seção 5.3, permite se descrever tanto os componentes como arquiteturas utilizando instâncias desses componentes. Uma característica de DraX é

[5] A integração de DraX a um processo formal de desenvolvimento de software é um trabalho futuro.

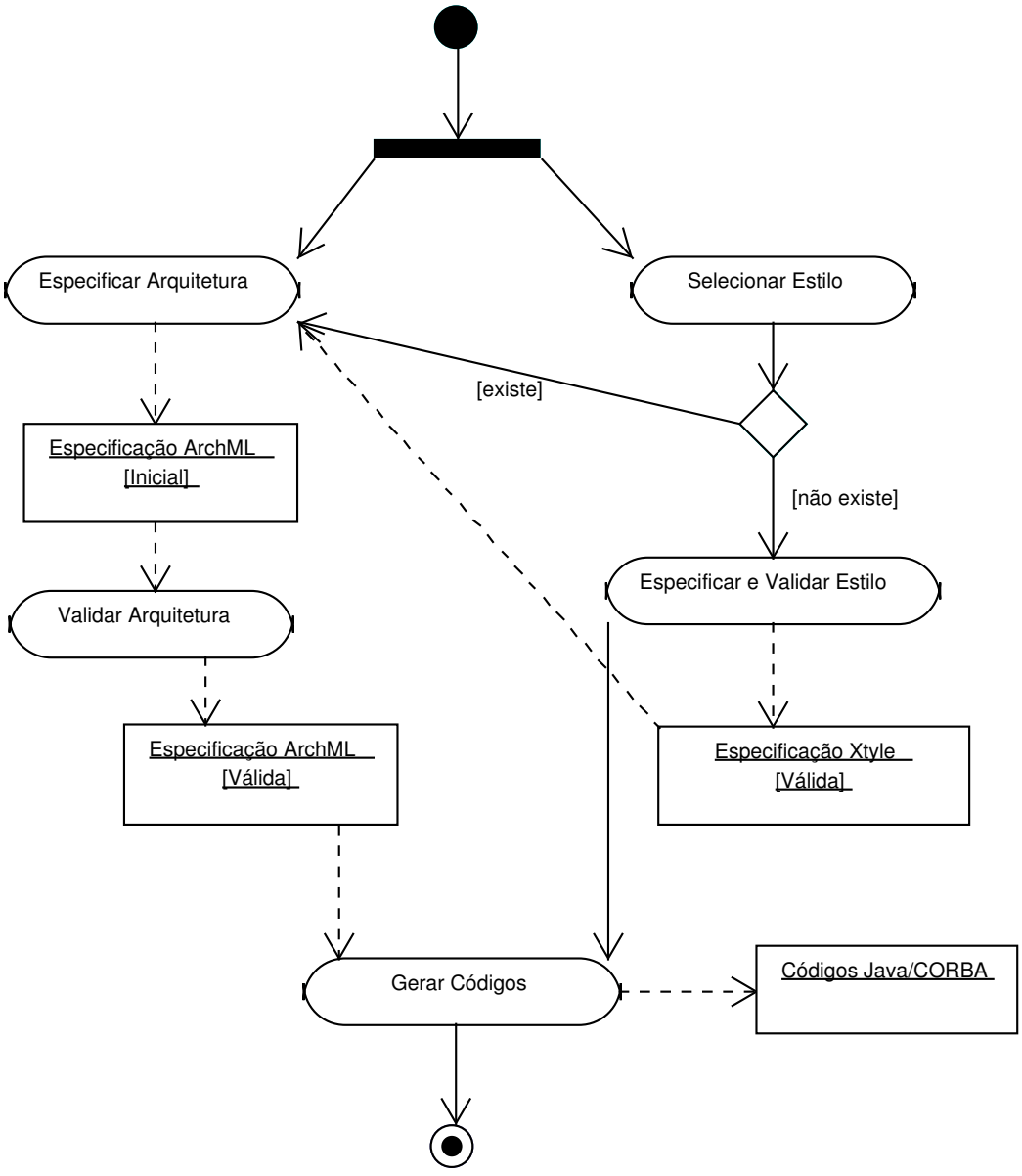


Figura 4.1: Metodologia de Desenvolvimento com DraX.

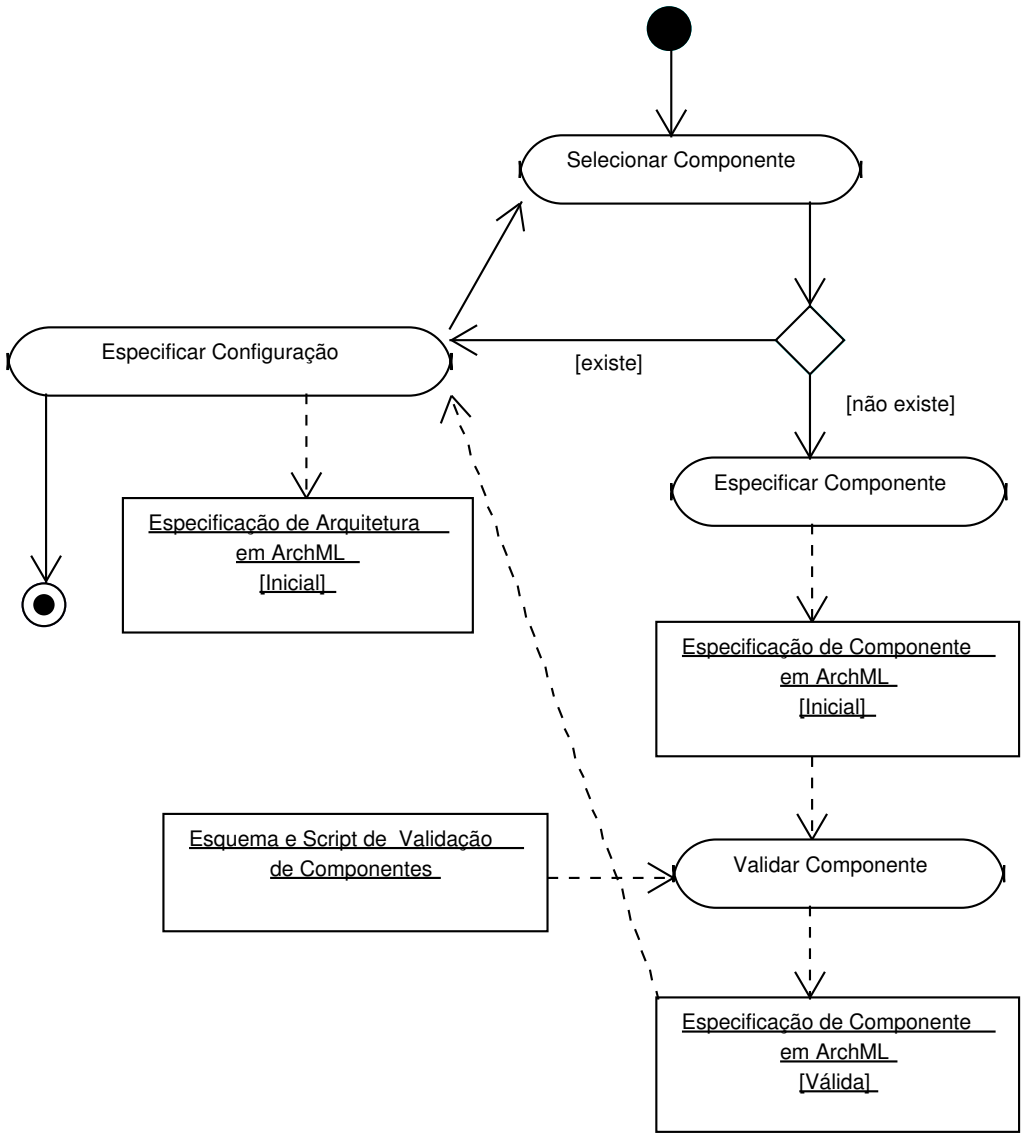


Figura 4.2: Especificar Arquitetura.

possibilitar a criação de arquiteturas distribuídas cujas descrições dos componentes também possam ser distribuídas. Além disso, ArchML permite se referenciar um estilo na especificação de uma arquitetura.

A Figura 4.2 apresenta o detalhamento do diagrama de atividades Especificar Arquitetura. Podemos observar que, inicialmente há a seleção dos componentes que formarão a configuração concomitantemente com a especificação da configuração (topologia) da arquitetura.

Essa etapa de seleção[6] reforça a idéia de reuso de componentes previamente desenvolvidos por terceiros. Esses componentes são referenciados na especificação ArchML, o que quer dizer que eles podem estar distribuídos.

Caso o componente não exista, ele é especificado, tanto sintaticamente como comportamentalmente. Além disso, a especificação desses componentes deve ser devidamente validada antes de ser inserida na arquitetura.

[6]O termo *seleção de componentes* é utilizado de forma simplista nesse trabalho para indicar a etapa de identificação de componentes desenvolvidos por outros integrantes de uma mesma equipe distribuída.

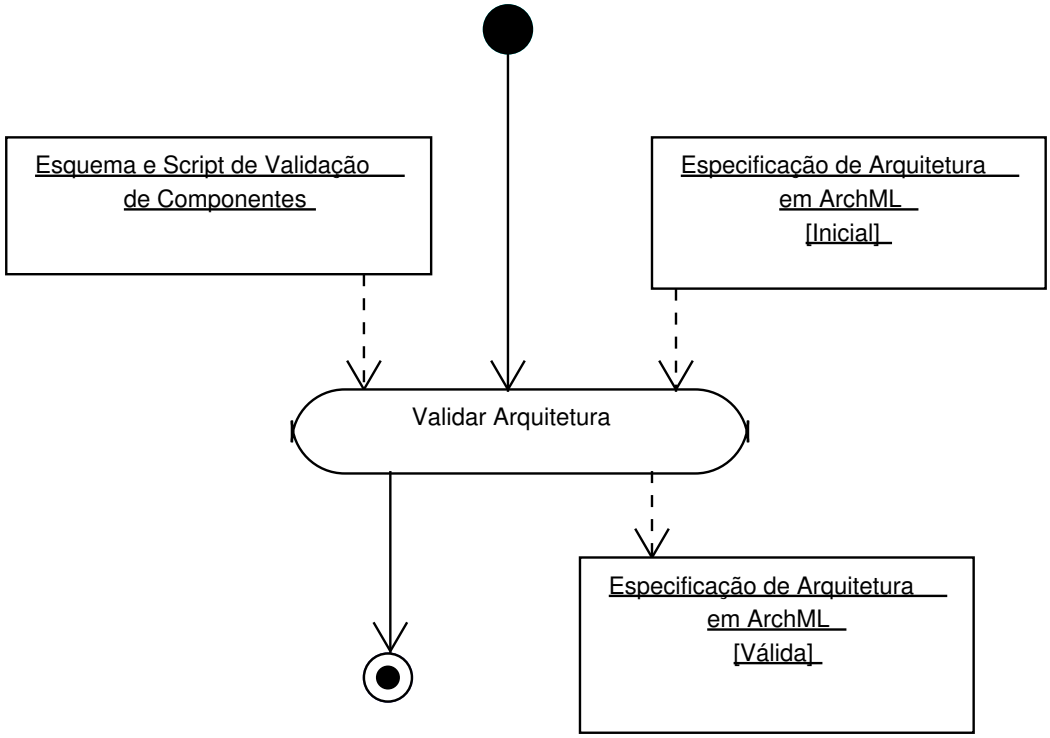


Figura 4.3: Validar Arquitetura.

Para se especificar completamente a arquitetura utilizando a linguagem ArchML, devemos definir o comportamento de cada componente. Essa definição é realizada atravé de Diagramas de Descrição de Protocolos (DDP). Esses diagramas, que desenvolvemos a partir dos diagramas de estados UML, permitem se descrever o comportamento observável de um componente com relação às suas portas (a Seção 5.3 apresenta uma descrição detalhada desses diagramas). Para se descrever esses diagramas utilizamos qualquer ferramenta de projeto UML e utilizaremos as mesmas notações dos diagramas de estados. Em seguida, devemos gerar uma especificação XMI a partir desse diagrama, pois o que devemos referenciar na descrição do comportamento do componente em ArchML é o nome do arquivo XMI no qual temos a descrição do comportamento do componente.

4.4.2 Validar Arquitetura

Depois de ter selecionado todos os componentes para a criação da configuração, será gerado uma especificação ArchML do sistema completo. Essa especificação será validada também através de XML Schemas e de Schematrons que verificarão, entre outras coisas, a compatibilidade entre as portas das instâncias dos componentes, os tipos de dados, etc.

A Figura 4.3 mostra o diagrama de atividades para esse processo. Além disso, a arquitetura é validada com relação a aderência ao estilo especificado, ou seja, é verificado se todos os componentes da arquitetura são dos tipos especificados para o estilo e se todas as regras topológicas são obedecidas. O resultado desse processo é a especificação ArchML devidamente validada.

Uma outra faceta da validação de arquitetura diz respeito a validação comportamental. Esse tipo de validação é realizado através da verificação da compatibilidade comportamental

entre os componentes conectados em uma arquitetura. Para realizar essa verificação de forma automatizada, desenvolvemos um *script* Java utilizando bibliotecas DOM [21] e XPath que gera automaticamente a partir de uma especificação XMI de um Diagrama de Descrição de Protocolos uma especificação em π-cálculo. Essas especificações podem ser testadas utilizando ferramentas como o *Mobility WorkBench* [120], que executa verificações em especificações π.

Além de se verificar a compatibilidade entre componentes, devemos garantir algumas propriedades comportamentais da arquitetura como um todo. Assim, devemos construir uma especificação semântica para a linguagem ArhcML de forma a podermos verificar essas propriedades formalmente. Desse modo, propomos uma álgebra de processos denominada $\mathcal{R}\pi$-cálculo[108, 106, 107, 109] (na Seção 6.3, o cálculo $\mathcal{R}\pi$ é apresentado com detalhes). Através de especificações $\mathcal{R}\pi$ podemos verificar facilmente propriedades como impasses em especificações ArchML.

4.4.3 Especificar e Validar Estilo

Na Figura 4.1, a atividade de Especificar e Validar estilos aparece de forma integrada ao processo de desenvolvimento da arquitetura. Na verdade essa tarefa é realizada em paralelo, possivelmente por projetistas especializados, e tem como finalidade fornecer uma descrição do estilo que deverá ser utilizado no desenvolvimento da arquitetura. Esse estilo fornecerá um vocabulário de tipos e um conjunto de restrições de comunicação e topológicas. Essas informações também serão consideradas na geração dos templates de implementação. A Figura 4.4, apresenta um diagrama de atividades detalhado para a atividade de Especificar e Validar Estilos.

Pode-se perceber a partir da Figura 4.4, que os estilos em DraX, especificados pela linguagem Xtyle, podem ser derivados, ou seja, estilos podem ser gerados a partir de restrições em estilos já pré-existentes na biblioteca de estilos de DraX, ou podem ser definidos pelo usuário. Os estilos definidos pelo usuário são criados totalmente pelo desenvolvedor. Tanto o vocabulário como as restrições topológicas ficam a cargo do desenvolvedor.

A etapa seguinte, que é a validação do estilo, deve ser realizada por esquemas apropriados para cada estilo. Esses esquemas de validação de DraX são definidos utilizando Schematron [61]. Para os estilos derivados são utilizados os esquemas de validação dos estilos primitivos. Entretanto, os novos estilos definidos não possuem esquemas próprios para validação. Contudo construímos um *script* XSLT que realiza a geração de Schematros validadores para esses novos estilos. Assim, o desenvolvedor do estilo não precisa se preocupar em realizar essa tarefa manualmente[7]. O resultado dessa atividade é a especificação de um novo estilo devidamente validada e a especificação de um esquema de validação de arquiteturas que pode ser utilizado para verificar se uma arquitetura está de acordo com o estilo.

Da mesma forma que fizemos para descrever comportamento em ArchML devemos usar os diagramas DDP para especificação o comportamento dos tipos de componentes de um estilo. Contudo, a notação para esses diagramas é um pouco diferentes das usadas para descrever componentes em ArchML (apresentamos essa nova notação na Seção 5.4). Entretanto, a metodologia de se gerar uma especificação XMI é seguida da mesma forma em Xtyle.

[7]Exemplos mais detalhados da utilização dessa característica de DraX pode ser encontrado em [113]

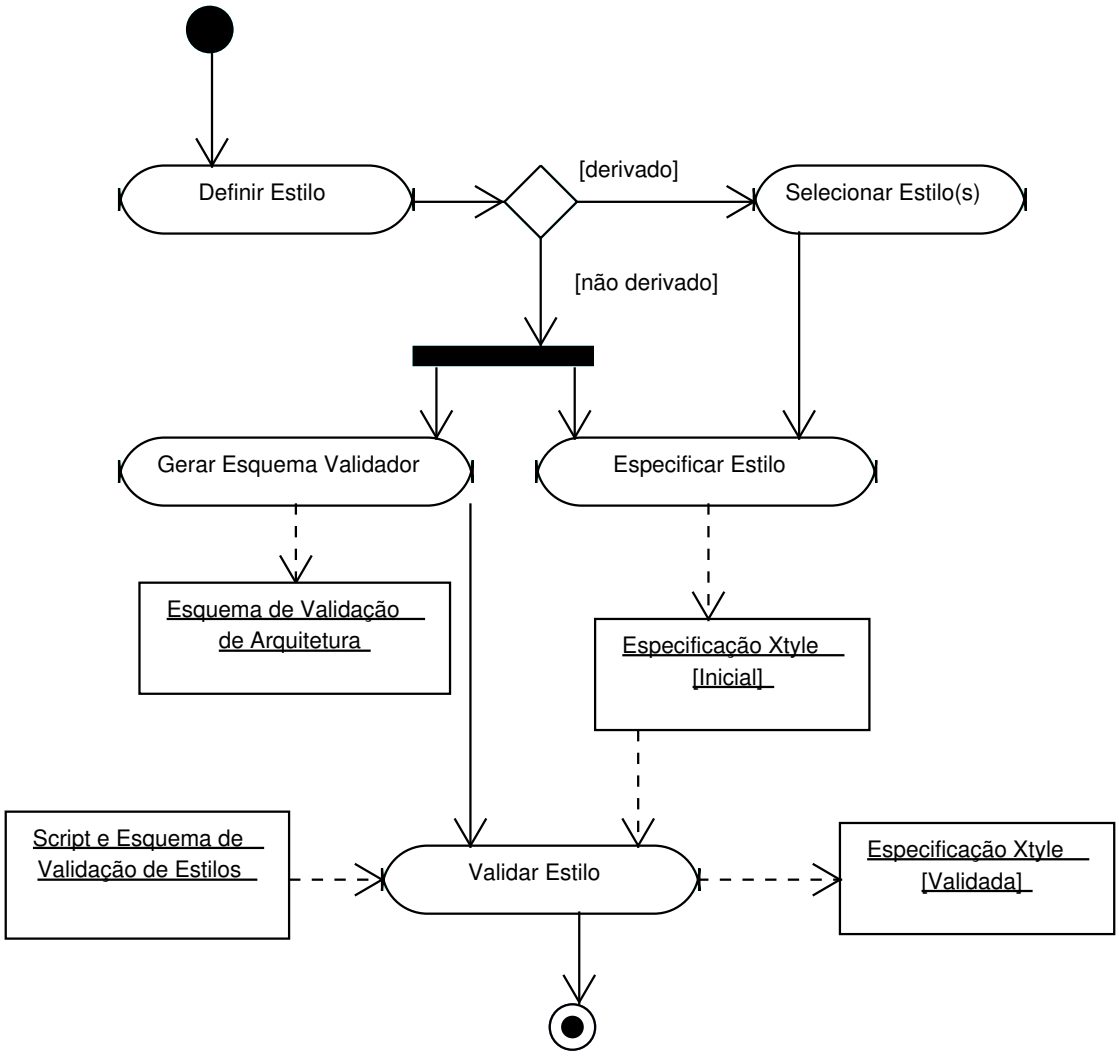


Figura 4.4: Especificar e Validar Estilo.

4.4.4 Gerar Templates

A última parte do processo de construção arquitetural com DraX é a geração de templates de implementação Java com código relativo a um middleware. Esses templates consideram as informações da arquitetura e dos estilos que essas seguem. Assim, se for definida que uma arquitetura, por exemplo, deve realizar comunicação assíncrona e que os objetos serão *multithread*[8], e se tivermos utilizando CORBA como middleware, os templates gerados utilizarão o serviço de notificação de CORBA e o POA terá definida uma política *multithread* para essa aplicação.

A Figura 4.5 apresenta um diagrama de atividades mais detalhado para essa fase. Podemos observar que são gerados templates de implementação para os componentes da arquitetura. Nesses templates, tanto os códigos dos serviços do middleware como também as interconexões entre os objetos são fornecidos de forma automatizada. Assim, o desenvolvedor fica encarregado de apenas implementar a lógica da aplicação, sendo que, se estivermos utilizando CORBA por exemplo, as etapas de inicialização do serviço de nomes, adaptador de objetos, definição de políticas do POA, registro e instanciação de serviços CORBA, já foram resolvidos pelos *scripts* de DraX.

4.5 Conclusão

Nesse capítulo a estrutura do *framework* DraX foi apresentada. Em DraX todas as etapas relativas à construção de arquiteturas e estilos arquiteturais distribuídos são consideradas. Essas etapas vão desde a especificação, passando pela validação sintática, estrutural e comportamental até chegar à geração de templates de implementação.

Seguindo os objetivos iniciais traçados para essa tese, utilizamos em DraX apenas tecnologias que fazem parte do trabalho regular das empresas de desenvolvimento. Assim, as arquiteturas e os estilos são descritos, respectivamente, utilizando as linguagens ArchML e Xtyle, que são aplicações de XML. A Validação dessas arquiteturas e estilos é relizada através de XML Schemas e Schematrons, que também são aplicações de XML. A descrição de comportamentos é realizada através de adaptações dos diagramas de estados UML e incorporadas a ArchML e Xtyle através de XMI, que é uma aplicação de XML. A validação sintática de arquiteturas e estilos e realizada através de Schematrons e a verificação comportamental é realizada através de $\mathcal{R}\pi$ entretanto, as especificações $\mathcal{R}\pi$ são geradas por *scripts* XSLT a partir das especificações XMI. Os templates de implementação são gerados através de *scripts* XSLT e não é necessário se conhecer os detalhes do middleware subjacente.

Uma outra vantagem de DraX está na modelagem de estilos arquiteturais. Xtyle é uma linguagem de fácil manipulação e que permite criar estilos arquiteturais complexos. Uma outra contribuição de Xtyle para a área de estilos arquiteturais é a possibilidade de realizar (de fato) a especificação de estilos através de refinamentos de outros estilos pré-existentes. Além de minimizar o trabalho de especificação, o desenvolvedor conta com todo o potencial das ferramentas de validação também já existentes em DraX.

Com relação à introdução de estilos arquiteturais na construção de arquiteturas distribuídas, DraX trás a contribuição para a área através da possibilidade de criação de arquiteturas que incorporem diversos estilos. Essa abordagem, levantada por [14], foi tratada por [7], porém, sua abordagem baseia-se em descrições formais e de difícil implementação. Em

[8]Utilizamos em DraX o mesmo mecanismo de anotações de ACME, que é utilizado para armazenar informações específicas de outras linguagens.

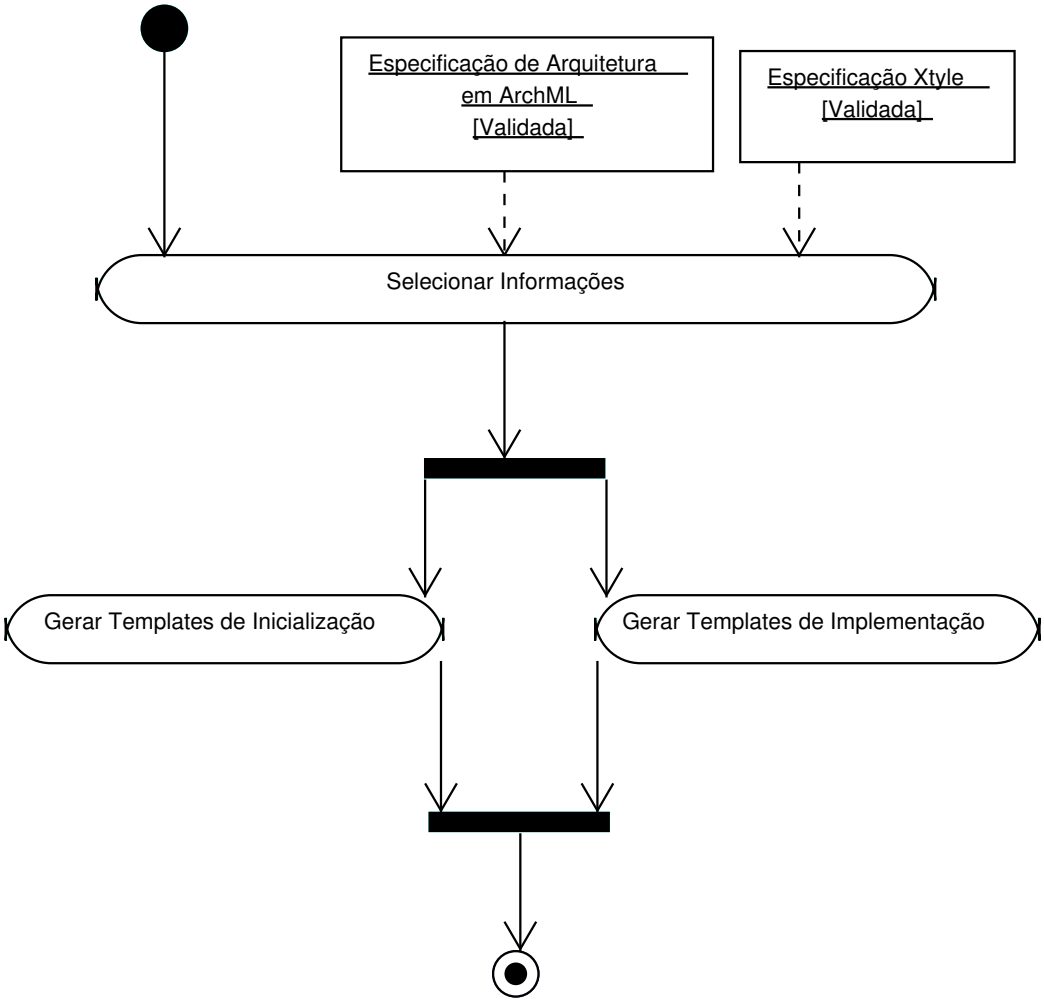


Figura 4.5: Gerar Códigos.

DraX, concretizamos essa idéia sem perder as vantagens das descrições formais pregadas em [7].

Nos capítulos seguintes todas as partes de DraX serão detalhadas. Inicialmente, no Capítulo 5, serão apresentadas as linguagens de DraX. Na Seção 5.2 é apresentada uma linguagem de padrões que explica todas as decisões que utilizamos para criar as estruturas sintáticas das linguagens ArchML e Xtyle. Em seguida essas linguagens são descritas com detalhes e exemplos nas Seções 5.3 e 5.4, respectivamente. Continuando, no Capítulo 6 a etapa de validação é apresentada. Na Seção 6.2 serão apresentados os *scrips* de validação sintática de DraX. Em seguida, na Seção 6.3, será apresentado o processo de validação comportamental de DraX, onde um novo cálculo, o $\mathcal{R}\pi$ é apresentado. Os *scripts* de geração de templates de implementação são tratados no Capítulo 7 e para validar o trabalho e mostrar a versatilidade de DraX é apresentado no Capítulo 8 dois estudos de caso clássicos, onde mostramos a aplicação de infraestruturas de middleware diferentes e a utilização de estilos definidos pelo próprio usuário.

Capítulo 5

Especificação de Arquiteturas e Estilos

5.1 Introdução

A fase inicial do desenvolvimento com DraX é realizada através da especificação das arquiteturas e estilos arquiteturais. Cada arquitetura é especificada através da referência a componentes que são especificados externamente. Nesse caso instâncias de cada tipo de componente são definidas para formar uma arquitetura. Em DraX, a linguagem ArchML foi definida para realizar essa tarefa.

Além da definição da arquitetura em si, essa pode seguir um estilo arquitetural. A obediência a um estilo em DraX é definida através da aderência dos componentes a um vocabulário específico definido no estilo. Nesse caso, cada componente da arquitetura deve ser de um (ou mais) tipos que são definidos no estilo (ou estilos) que a arquitetura segue. Esses estilos, por sua vez, tanto podem já existir previamente, como podem ser criados através da linguagem Xtyle.

Essas linguagens, embora sejam simples, passaram por diversos refinamentos até chegar a suas estruturas atuais. Todo esse processo de decisão sobre o que utilizar em cada linguagem para que as descrições das arquitetura e estilos fossem simplificadas é apresentado logo a seguir na Seção 5.2 desse capítulo, onde desenvolvemos uma linguagem de padrões para facilitar a definição da estrutura sintática de ArchML e de Xtyle. Em seguida descrevemos em detalhes as linguagens ArchML (Seção 5.3) e Xtyle (Seção 5.4).

5.2 Requisitos das Linguagens de DraX

Como apresentado anteriormente, sabemos que para formalizar a representação de projetos arquiteturais foram criadas as ADLs *(Architecture Description Languages)*. Essas linguagens fornecem um *framework* sintático e um conjunto de ferramentas que possibilitam a especificação de projetos arquiteturais e a realização de diversos tipos de análises nessas arquiteturas. Exemplos de ADLs incluem Darwin [72], UniCon [101], Meta-H [13], Wright [7], Acme [44] entre outras.

Entretanto, a proliferação de ADLs tem dificultado a decisão de qual linguagem adotar para realizar um determinado projeto arquitetural. Principalmente se considerado que cada ADL possui características específicas e ferramentas que realizam tarefas também específicas. Além disso, como apresentado em [36], há atualmente uma necessidade de se construir ADLs com características que facilitem a manipulação de arquiteturas distribuídas.

Essa especificidade das ADLs acaba tornando necessário a adoção de mais de uma dessas linguagens para a realização de um mesmo projeto arquitetural, de modo a se conseguir atingir todos os objetivos do projeto. Contudo, fatores como tempo e custos necessários para se dominar essas linguagens acabam sendo, em alguns casos, proibitivos no desenvolvimento de alguns projetos arquiteturais com ADLs.

Nesse caso, pode ser razoável se optar pela construção de uma nova ADL pela própria equipe de desenvolvimento, onde as característica dessa linguagem possam ser definidas de acordo com o perfil da equipe que vai utilizá-la. Com uma ADL definida pela própria equipe, os projetos arquiteturais serão desenvolvidos mais rapidamente, visto que não haveria a necessidade da adoção de outras ADLs, e até mesmo o tempo de adaptação da equipe à nova linguagem seria extremamente reduzido.

Para facilitar o processo de desenvolvimento de uma nova ADL, nessa seção apresentamos uma linguagem de padrões que captura o processo decisório relativo ao projeto da sintaxe de ADLs. Esses padrões auxiliam a definir aspectos importantes relativos à forma com que a sintaxe de uma ADL deve ser projetada, de modo a ser de fácil manuseio e flexível no que diz respeito à definição de informações sobre arquiteturas de software. Essa linguagem de padrões é utilizada nesse trabalho para definir estruturas sintáticas das linguagens de DraX.

5.2.1 Estrutura da Linguagem de Padrões

Os problemas tratados nessa linguagem de padrões e as soluções propostas para estes problemas são resumidos na Tabela 5.2. A Figura 5.1 apresenta as dependências entre os padrões propostos nesse capítulo. Nessa figura, estão apresentadas em linhas cheias as dependências diretas entre os padrões, que indicam a necessidade de se ter concluído o padrão para se iniciar o seguinte. Por sua vez, a linha tracejada define dependências que não possuem ordem de precedência ou padrões que devem ser desenvolvidos em paralelo. A ordem com que esses padrões devem ser aplicados é a mesma com que eles são descritos ao longo do capítulo.

O padrão *Projetar ADL*, apresentado na Seção 5.2.2, é o padrão principal da linguagem. Ele é quem justifica a criação de uma nova ADL e identifica os padrões a serem utilizados para o projeto sintático dessa ADL. O padrão *Elementos Arquiteturais Básicos*, apresentado na Seção 5.2.3, identifica quais os tipos de elementos arquiteturais devem ser descrito por uma ADL. A estruturação de uma especificação arquitetural a partir dos elementos que constituem sua arquitetura é definida no padrão *Organização das Especificações*, tratado na Seção 5.2.4. Na Seção 5.2.5 é apresentado o padrão *Informações dos Elementos Arquiteturais*, que descreve como se decidir a forma com que uma ADL deve especificar as informações para cada elemento arquitetural. O padrão *Estrutura Hierárquica de Informações* apresentado na Seção 5.2.6 define a forma de como se organizar a apresentação das informações sobre os elementos arquiteturais.

5.2.2 Projetar ADL

Contexto

Você está trabalhando em um sistema de software complexo no qual um grande número de requisitos devem ser considerados. Além disso, é necessário a realização de diversos tipos de simulações e análises nesse sistema. Para suavizar a passagem da fase de elicitação e análise dos requisitos para a fase de implementação deve ser especificada uma arquitetura para o software. Contudo, para se conseguir uma representação arquitetural que permita a realização das análises requeridas é necessário a utilização de uma ADL. Entretanto, é

Padrão	**Problema**	**Solução**
Projetar ADL	Como minimizar esforço e custo na especificação de arquiteturas de software quando há a necessidade de se utilizar diversas ADLs ?	Desenvolva uma nova ADL que contemple as principais necessidades identificadas pela equipe de desenvolvimento para a representação de seus projetos arquiteturais.
Elementos Arquiteturais Básicos	Que tipos de elementos arquiteturais devem ser representados pela ADL de forma a se criar especificações arquiteturais fáceis de ser construídas e mantidas ?	Defina tipos de elementos básicos para representar elementos arquiteturais, onde cada tipo de elemento deverá ter uma funcionalidade específica na arquitetura.
Organização das Especificações	Como organizar a especificação da arquitetura de um software a partir das especificações de seus elementos arquiteturais constituintes ?	Especifique em apenas um arquivo tanto os elementos arquiteturais como a arquitetura do software em si.
Informações dos Elementos Arquiteturais	Quais informações deverão estar disponíveis na ADL para a especificação de cada elemento arquitetural ?	Desenvolva a ADL de forma a deixar o projetista livre para decidir quais informações representar em cada elemento arquitetural. Entretanto, identifique algumas informações que devem ser obrigatórias para cada elemento e torne todas as demais opcionais.
Estrutura Hierárquica de Informações	Como a ADL deve representar as informações dos elementos arquiteturais de forma a facilitar o entendimento dessas informações ?	Crie um conjunto de contextos para cada informação e, dentro de cada contexto, defina de forma hierárquica as informações pertinentes ao contexto.

Tabela 5.2: Problemas e Soluções dos Padrões de Projeto Sintático de ADLs.

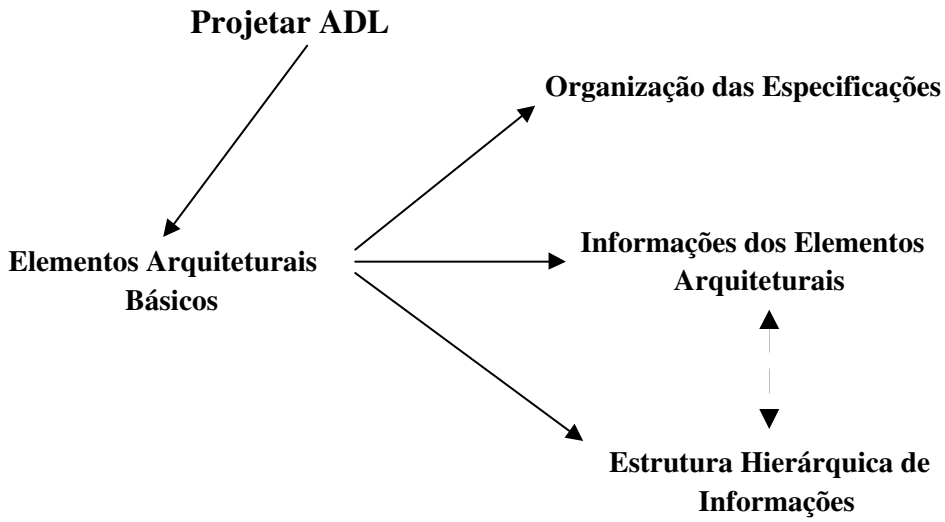


Figura 5.1: Dependência entre os Padrões.

bastante difícil encontrar uma ADL que sozinha possua todas as características necessárias para representar essa arquitetura e forneça todas as ferramentas necessárias para a realização das análises requeridas. Nesse sentido, surge a necessidade de se utilizar mais de uma ADL para especificar a arquitetura desse sistema.

Problema
Como minimizar esforço e custo na especificação de arquiteturas de software quando há a necessidade de se utilizar diversas ADLs ?

Forças

- Quanto mais complexo é o projeto, mais difíceis são as análises a serem realizadas, o que exige a utilização de ADLs que possuam ferramentas específicas.
- Ao se utilizar ADLs diferentes para realizar tarefas específicas em um projeto arquitetural surge a necessidade de treinar a equipe de desenvolvimento para entender a sintaxe e as ferramentas de cada nova ADL a ser utilizada, o que aumenta o tempo de desenvolvimento do projeto.
- Os custos de um projeto aumentam significativamente com o treinamento da equipe e com a necessidade de adaptação da arquitetura para cada nova ADL utilizada em um determinado projeto arquitetural.
- A equipe de desenvolvimento pode optar em adotar uma ADL específica que mais se aproxime das características do projeto e da formação dessa equipe. Entretanto, isso limita a expressividade dos projetos às características da ADL escolhida.

Solução
Desenvolva uma nova ADL que contemple as principais necessidades identificadas pela equipe

de desenvolvimento para a representação de seus projetos arquiteturais. Essas características devem ser identificadas e refinadas a partir do acúmulo de experiência dessa equipe, adquirida com a especificação de arquiteturas de software utilizando outras ADLs existentes.

Para se criar uma nova ADL, diversos aspectos devem ser considerados. Dentre eles, o poder de representação das informações requeridas pela equipe de desenvolvimento para a especificação dos elementos arquiteturais é um dos mais importantes. A forma com que a ADL fragmenta a representação de arquiteturas complexas em elementos arquiteturais mais simples tem influência direta na manutenibilidade dessa arquitetura.

Contexto Resultante
A aplicação desse padrão resulta na decisão de se construir uma nova ADL que melhor se adapte às necessidades específicas dos projetos arquiteturais de uma determinada equipe de desenvolvimento.

Racionalização
Embora seja caro produzir uma nova ADL, o custo de treinamento da equipe para entender determinadas ADLs, ou adaptar uma arquitetura para uma ADL específica, pode ser muito mais alto. Principalmente se for considerado que cada projeto tem características diferentes sendo que, normalmente, há a necessidade de se utilizar uma ADL que se adapte melhor a cada um dos projetos.

Ao se desenvolver uma nova ADL, a equipe terá a facilidade de adaptá-la às suas necessidades sempre que for necessário, sem necessidade de treinamento para entender a ADL. Com o passar do tempo, e com a acúmulo de experiência da equipe, a ADL estará estável o suficiente para suprir toda a demanda de análise arquitetural necessária sem gasto extra. Além disso, as ferramentas poderão ser desenvolvidas pela própria equipe, de modo a se conseguir realmente os resultados desejados.

Consequências
Algumas das vantagens do padrão Projetar ADL são as seguintes:

- **Uniformidade na representação arquitetural.** Com apenas uma ADL todos os projetos arquiteturais da equipe terão apenas uma linguagem de representação, o que facilita a compreensão dos projetos e cria um vocabulário comum entre os membros da equipe de desenvolvimento.
- **Agilização do projeto arquitetural.** Com a adaptação da equipe de desenvolvimento à nova ADL, todos os novos projetos que se seguem serão mais rapidamente realizados.
- **Tempo menor de aprendizagem.** Se for necessário a mudança de membros da equipe, o tempo de treinamento para a incorporação desses novos membros será bastante reduzido.
- **Adaptabilidade.** Por ser um produto da equipe de desenvolvimento, a ADL poderá ser adaptada facilmente para novas necessidades que surgirem.

Algumas das desvantagens do padrão Projetar ADL são as seguintes:

- **Custo inicial alto.** A implementação de uma nova ADL vai requerer esforço, tempo e, portanto, um orçamento extra para essa tarefa. Entretanto, futuras atualizações tendem a ter custos bem reduzidos.

- **Implementação de ferramentas.** Além da ADL em si, as ferramentas que permitirão à realização de análises nas arquiteturas, também devem ser implementadas.

Padrões Relacionados
Para que a implementação de uma nova ADL seja viabilizada, vários aspectos devem ser considerados a priori. Esses aspectos dizem respeito principalmente às características sintáticas da ADL, onde a primeira decisão a ser tomada está relacionada com a forma com que os elementos arquiteturais são identificados pala ADL. O padrão *Elementos Arquiteturais Básicos* (seção 5.2.3) trata desse problema.

5.2.3 Elementos Arquiteturais Básicos

Contexto
De acordo com o padrão *Projetar ADL*, você decidiu criar uma ADL para especificar a arquitetura de um software complexo. Essa tarefa é iniciada pela definição dos tipos de elementos arquiteturais que a ADL deve possuir.

Problema
Que tipos de elementos arquiteturais devem ser representados pela ADL de forma a se criar especificações arquiteturais fáceis de ser construídas e mantidas ?

Forças

- A definição dos tipos de elementos arquiteturais a serem especificados para uma arquitetura depende muito da experiência da equipe de desenvolvimento.
- Os tipos de elementos arquiteturais a serem definidos para a ADL influencia diretamente a gerenciabilidade e a clareza das especificação realizadas por essa linguagem. Sendo que um aumento exagerado no número de tipos de elementos especificados resulta em uma arquitetura de difícil gerenciamento. Por outro lado, a construção de uma especificação sem a definição clara dos tipos de elementos arquiteturais representados pode tornar a arquitetura ilegível.
- O potencial de reuso de elementos de especificações arquiteturais é definido pela forma com que esses elementos são descritos pela ADL. Sendo que quanto mais clara e precisa for a classificação dos tipos de elementos arquiteturais melhor será o reusabilidade dessas especificações.

Solução
Defina tipos de elementos básicos para representar elementos arquiteturais, onde cada tipo de elemento deverá ter uma funcionalidade específica na arquitetura, de forma a tornar clara a especificação como um todo. Desse modo os seguintes tipos de elementos arquiteturais podem ser identificados:

- Interfaces - Devem descrever as funcionalidades requeridas e fornecidas pelos elementos arquiteturais.

- Componentes - Devem descrever as partes da arquitetura que realizam algum tipo de computação ou elementos de armazenamento.
- Conectores - Devem descrever as estruturas de comunicação entre os componentes.
- Configurações Arquiteturais - Devem descrever a forma com que componentes e conectores são organizados para formar uma arquitetura.

Contexto Resultante
A aplicação desse padrão identifica os tipos de elementos necessários à ADL para a construção de arquiteturas de software. Dessa forma, cada elemento a ser representado em uma especificação de arquitetura está relacionado a um tipo específico.

Racionalização
A definição de tipos de elementos arquiteturais em uma especificação facilita sobremaneira o entendimento da mesma. Nesse padrão, também foi adotada a abordagem de se especificar separadamente as interfaces dos componentes e dos conectores. Essas interfaces descrevem o que cada elemento fornece e utiliza de outros elementos. Com essa abordagem, existe a possibilidade de reuso de especificações de interfaces e de uma melhor checagem de tipos entre elementos arquiteturais distintos, além de tornar mais intuitiva a tarefa de particionar a arquitetura.

Usos Conhecidos
A ADL Darwin utiliza os conceitos de componentes e arquiteturas, mas não representa conectores explicitamente, podendo esses serem simulados a partir de especificações de componentes. Nessa linguagem as interfaces dos elementos são descritas dentro dos próprios elementos arquiteturais.

UniCon representa os elementos arquiteturais através de componentes, conectores e configurações. Entretanto, os conectores de UniCon são predefinidos como parte da própria ADL.

Wright também utiliza os conceitos de componentes, conectores e configurações. Entretanto, nessa ADL os conectores podem ser definidos pelo usuário e sua semântica é especificada através de CSP.

Variação
A abordagem clássica utilizada na definição de tipos de elementos arquiteturais em ADLs sugere a utilização dos seguintes elementos apenas: Componentes, Conectores e Configurações Arquiteturais. Nessa abordagem, as interfaces tanto dos componentes como dos conectores, são definidas internamente a esses elementos, não podendo ser reutilizadas em outras arquiteturas.

Conseqüências
Algumas das vantagens do padrão Elementos Arquiteturais Básicos são as seguintes:

- **Reusabilidade.** A separação da especificação de interfaces dos elementos arquiteturais permite a reusabilidade dessas interfaces em outros projetos.

- **Previsibilidade de comportamento.** A definição de tipos para cada elemento arquitetural, permite se inferir que comportamento um determinado elemento vai ter dentro de projeto arquitetural.

Uma desvantagem do padrão Elementos Arquiteturais Básicos é a seguinte:

- **Configurações mais complexas.** Além da definição das interações entre componentes e conectores, os próprios componentes e conectores devem indicar quais interfaces devem utilizar, isso pode tornar as arquiteturas um pouco mais complexas.

Padrões Relacionados
Um aspecto importante a ser considerado na especificação da sintaxe de uma ADL é a forma com que os elementos arquiteturais especificados são organizados para produzir a especificação de uma arquitetura. Esse problema é tratado no padrão *Organização das Especificações*, apresentado na seção 5.2.4.

Um outro aspecto também importante é a representação das informações sobre os elementos que compõem as arquiteturas. Essas informações devem ser de fácil compreensão e gerenciamento, e deve ser possível a inclusão de informações não previstas sobre determinados elementos arquiteturais sem ter que reescrever a ADL. Para se determinar os tipos de informações necessárias para representar elementos arquiteturais pela ADL utilize o padrão *Informações dos Elementos Arquiteturais*, apresentado na seção 5.2.5.

5.2.4 Organização das Especificações

Contexto
Foi definida a necessidade de se criar uma nova ADL, apresentada no padrão *Projetar ADL*. Também foram definidos os tipos de elementos arquiteturais básicos que essa ADL pode representar, relacionados no padrão *Elementos Arquiteturais Básicos*. Esses elementos arquiteturais devem ser utilizados para se criar a representação de arquiteturas de software.

Problema
Como organizar a especificação da arquitetura de um software a partir das especificações de seus elementos arquiteturais constituintes ?

Forças

- A organização da especificação da arquitetura de um software realizada em um único arquivo junto com a especificação de seus elementos arquiteturais constituintes, garante uma manipulação mais fácil dessa especificação, entretanto a reusabilidade dos elementos arquiteturais fica comprometida.
- Dependendo do tamanho da equipe que está produzindo a especificação arquitetural pode ser necessários desmembrar a especificação em partes menores. Sendo que quanto maior a equipe, maior pode ser a fragmentação necessária.
- A complexidade da especificação arquitetural e a estabilidade dos requisitos que a gerou, pode determinar a forma com que os elementos que compõem a arquitetura

serão organizados. Sendo que quanto mais complexa for a estrutura do software mais necessário se faz a centralização da especificação.

Solução
Especifique em apenas um arquivo tanto os elementos arquiteturais como a arquitetura do software em si. Inicialmente realize a especificação de todos os elementos arquiteturais presentes na arquitetura. Em seguida especifique a arquitetura através das conexões entre esses elementos.

Contexto Resultante
Tanto os elementos arquiteturais como a configuração desses elementos formando a arquitetura são especificados em apenas um documento, cabendo à equipe a tarefa de organizar o trabalho de seus participantes de forma independente em cima de uma mesma fonte de informação.

Racionalização
Especificar tanto os elementos arquiteturais como a arquitetura do software em um mesmo arquivo texto facilita a compreensão da especificação e a visualização de inconsistências, o que pode agilizar a tarefa de especificação.

Usos Conhecidos
A organização de especificações arquiteturais em um arquivo único é um consenso entre as ADLs. A ADL UniCon, por exemplo, utiliza essa estrutura para organizar suas especificações. O Código 5.1, apresenta uma especificação arquitetural escrita em UniCon. Como pode ser observado, tanto os elementos arquiteturais como a configuração desses elementos são definidos em apenas um documento.

Variação
Elementos Arquiteturais em Arquivos Separados
Uma variação para esse padrão é a especificação separada dos elementos arquiteturias. Isso permite a construção de especificações arquiteturais de forma distribuída, além de possibilitar o reuso dessas especificações. Entretanto, a tarefa de gerenciamento da construção da arquitetura torna-se bem mais complexa.

Consequências
Algumas das vantagens do padrão Organização das Especificações são as seguintes:

- **Fácil compreensão.** Tendo centralizado a especificação de todos os elementos em um arquivo único, fica mais fácil se observar e manipular as características da arquitetura.
- **Facilidade de Evolução.** Qualquer modificação que seja necessária, pode ser melhor planejada se observando as características dos elementos arquiteturais individualmente.

Algumas das desvantagens do padrão Organização das Especificações são as seguintes:

Código 5.1 Organização de Especificação Arquitetural em UniCon.

```
COMPONENT Reverser {
   INTERFACE IS
      TYPE Filter
      PLAYER input IS StreamIn
         SIGNATURE (“line”)
         PORTBINDING (stdin)
      END input
      PLAYER output IS StreamOut
         SIGNATURE (“line”)
         PORTBINDING (stdout)
      END output
      PLAYER error IS StreamOut
         SIGNATURE (“line”)
         PORTBINDING (stderr)
      END error
   END INTERFACE

   IMPLEMENTATION IS
      ...
      USES stack INTERFACE Stack
      ...
      CONNECT reverse.iob TO datause.user
      ...
      ESTABLISH C-proc-call WITH
         reverse.stack init AS caller
         stack.stack init AS definer
      END C-proc-call
      ...
      BIND output TO ABSTRACTION
         MAPSTO (reverse.fprintf)
      END output
   END IMPLEMENTATION
END Reverser
```

- **Baixo reuso.** O reuso de elementos arquiteturais fica comprometido com a especificação dos mesmos dentro de uma especificação arquitetural única. Sendo que para que um elemento seja reusado, este deve ser copiado para dentro da nova especificação.
- **Falta de Independência.** A construção da arquitetura deverá ser realizada por apenas uma pessoa por vez, visto que só existe um único arquivo onde todos os elementos arquiteturais são especificados. Isso compromete a evolução dos elementos de forma independente da arquitetura, cabendo a quem for modificar algum elemento, realizar essa mudança dentro do código da arquitetura.

Padrões Relacionados
Pode-se utilizar o padrão *Software Architecture* [82] para guiar a definição de instâncias de elementos arquiteturais para um determinado projeto.

5.2.5 Informações dos Elementos Arquiteturais

Contexto
Você tem que especificar os elementos arquiteturais em uma arquitetura de software. Cada elemento possui, além de um tipo específico (identificado no padrão *Elementos Arquiteturais Básicos*), diversos requisitos diferentes a serem representados, sendo que cada requisito pode necessitar de níveis de detalhamento diferentes.

Problema
Quais informações deverão estar disponíveis na ADL para a especificação de cada elemento arquitetural ?

Forças

- Especificar todos os requisitos dos elementos arquiteturais pode tornar a especificação complexa demais e desnecessariamente detalhada.
- Especificar poucos requisitos dos elementos arquiteturais pode tornar a especificação pouco expressiva.
- A quantidade de requisitos a serem representados é controlada pelas necessidades de cada projeto. Sendo que quanto mais complexas forem as análises requeridas por uma determinada arquitetura, maior será a quantidade de informações necessárias para realizar essas análises.

Solução
Desenvolva a ADL de forma a deixar o projetista livre para decidir quais informações representar em cada elemento arquitetural. Entretanto, identifique algumas informações que devem ser obrigatórias para cada elemento e torne todas as demais opcionais.

As informações obrigatórias são as necessárias para identificar as características básicas de cada elemento arquitetural. Já as informações opcionais são as que têm sua utilização definida de acordo com as necessidades de cada projeto.

Para organizar a representação dessas informações de forma a permitir um melhor entendimento das especificações, o padrão *Estrutura Hierárquica de Informações* deve ser utilizado.

Contexto Resultante
Os tipos de informações necessárias para capturar os requisitos dos elementos arquiteturais estão definidos, podendo esses serem representadas pela ADL. Entretanto, além de se definir quais informações são relevantes para se especificar um determinado elemento arquitetural, essas informações devem estar organizadas para que sejam mais facilmente entendidas e manipuladas.

Exemplo
A equipe de desenvolvimento deseja especificar um componente para uma arquitetura de software. Os requisitos levantados para esse componente indicam que ele possui as seguintes características:

- Representa um fornecedor de serviços (servidor);
- Aceita um número limitado de conexões simultâneas;

Baseado nesses requisitos podem ser observados dois tipos diferentes de informações a serem representadas pela ADL para se especificar esse componente. O primeiro tipo de informação diz respeito a identificação do componente em si, como por exemplo a utilização de um identificador único para representar o componente. Esse tipo de informação é considerada uma informação básica para o componente, e deve ser obrigatória para quaisquer outros componentes especificados pela ADL.

Um outro tipo de informação diz respeito às características do componente que podem variar de acordo com as necessidades de cada projeto. Por exemplo, o número máximo de conexões aceito pelo componente pode ser uma restrição referente às limitações do sistema de comunicação onde esse componente está inserido, sendo que essa informação pode não ser relevante se a aplicação especificada estiver totalmente centralizada em uma máquina apenas. Desse modo, a ADL deve representar essas informações de forma opcional, ou seja, devem haver elementos sintáticos que possam ser utilizados para representar essa informação sem se ter a obrigação de representá-las.

Usos Conhecidos
Todas as ADLs possuem mecanismos para a identificação de elementos arquiteturais. Seja através de um nome, ou por uma representação simbólica. Contudo, a representação de outros tipos de informações contextuais sobre os elementos não é uma característica comum a essas linguagens.

Entretanto, existem algumas exceções. A linguagem Acme, por exemplo, permite a representação de propriedades através de anotações dentro da especificação do elemento arquitetural. O Código 5.2, apresenta a especificação de algumas propriedades de um componente em Acme. Uma peculiaridade de Acme é que essas propriedades são tratadas apenas como anotações, e são manipuladas apenas por ferramentas específicas.

Conseqüências
Uma das vantagens do padrão Informações dos Elementos Arquiteturais é a seguinte:

Código 5.2 Propriedades em Acme.

```
Component server = {
   Port receive-request;
   Properties { idempotence :  boolean = true;
                max-concurrent-clients :  integer = 1 }}
```

- **Flexibilidade.** A quantidade de informações a ser utilizada na descrição de cada elemento é definida por quem está fazendo a especificação, de acordo com as necessidades do projeto.

Uma desvantagem do padrão Informações dos Elementos Arquiteturais é a seguinte:

- **ADL mais complexa.** A implementação da ADL para dar suporte a essa característica é um pouco mais difícil, visto que não se pode definir a priori quais elementos devem ser inseridos para a descrição de cada elemento arquitetural.

Padrões Relacionados
Tendo definido a forma com que a ADL deve permitir a especificação das informações dos elementos arquiteturais, o passo seguinte é se definir como a ADL realizará a organização dessas informações de forma a se conseguir um melhor entendimento e manipulação das mesmas. Para esse fim o padrão *Estrutura Hierárquica de Informações* deve ser utilizado.

5.2.6 Estrutura Hierárquica de Informações

Contexto
Você está especificando um elemento arquitetural para um software complexo. Esse elemento possui uma grande quantidade de informações para ser representada. De acordo com os requisitos do projeto em que esse elemento está inserido, há a necessidade de se representar todas as informações levantadas (o padrão *Informações dos Elementos Arquiteturais* mostra como a ADL deve representar essas informações). A especificação desse componente deve ser utilizada pelos outros participantes no processo de desenvolvimento do sistema.

Problema
Como a ADL deve representar as informações dos elementos arquiteturais de forma a facilitar o entendimento dessas informações ?

Forças

- Um elemento arquitetural sem uma estrutura bem definida para representar suas informações é difícil de ser entendido e, conseqüentemente, difícil de ser mantido.
- A mistura de informações dentro da especificação de um elemento arquitetural dificulta bastante a realização de análises sobre o elemento. Desse modo, quanto mais fácil for reconhecer as informações dentro da especificação dos elementos, mais fácil será sua manipulação.

- A criação de hierarquias para representar as informações dos elementos arquiteturais pode facilitar o entendimento da relação entre as informações. Entretando, se a profundidade dessa hierarquia for muito grande as informações não ficarão claras.

Solução
Crie um conjunto de contextos para cada informação e, dentro de cada contexto, defina de forma hierárquica as informações pertinentes ao contexto. Cada contexto deve servir como um delimitador de informações de um determinado tipo. Essas informações, por sua vez, devem ser organizadas de forma hierárquica dentro desses contextos.

Contexto Resultante
Com a aplicação desse padrão as informações necessárias para representar os elementos arquiteturais são organizadas contextualmente. Sendo que cada contexto representa um delimitador para tipos de informações diferentes.

Racionalização
Com a definição de contextos para organizar a representação de informações sobre elementos arquiteturais se diminui a profundidade da hierarquia de um determinado tipo de informação. Com isso, conseguimos todas as vantagens da representação de informações de forma hierárquica minimizando os riscos de tornar a especificação confusa.

Além disso, os contextos permitem se definir conjuntos de informações específicas, o que ajuda bastante a manutenção e a realização de análises sobre a arquitetura.

Exemplo
Suponha que se queira representar as seguintes informações sobre um componente de uma arquitetura de software.

1. Esse componente foi implementado por Cidcley;
2. Esse componente foi implementado no dia X;
3. O componente foi implementado em Java;
4. O arquivo que implementa o componente chama-se `Comp.class`
5. O PATH onde o arquivo deve ser instalado deve ser `/temp/Componentes`
6. Esse componente utiliza a Interface `Itf1`;

Para que todas essas informações possam ser mais facilmente representadas na especificação do componente de software através da ADL, elas devem ser separadas por tipos (como apresentado no padrão *Informações dos Elementos Arquiteturais*. Nessa caso, podem ser identificados três tipos distintos de informações: o primeiro tipo (itens 1 e 2) diz respeito ao gerenciamento do componente. O segundo tipo (itens 3, 4 e 5) são propriedades sobre a implementação do mesmo. O terceiro tipo (item 6) é uma informação sobre as interfaces utilizadas pelo componente.

Desse modo, podemos observar a existência de diferentes contextos relacionados às informações do componente. Assim, poderíamos definir os contextos `Gerencial`, `Propriedades` e `Interfaces`, onde cada tipo de informação, de acordo com esse exemplo,

Código 5.3 Contextos em MetaH.

```
periodic process implementation P1.SIMPLE is attributes
   self'SourceTime := 100 μs;
   self'Period := 1 sec;
   self'SourceFile := "p1.a";
end P1.SIMPLE;
periodic process implementation P2.SIMPLE is attributes
   self'SourceTime := 50 μs;
   self'Period := 1 sec;
   self'SourceFile := "p1.a";
end P2.SIMPLE;
```

poderia ser respectivamente especificada.

Usos Conhecidos
A ADL MetaH possui construtores que permitem a definição de contextos para certas propriedades da aplicação. Essas propriedades são fornecidas através de atributos. Entretanto, a definição de informações são limitadas a escalonabilidade, confiabilidade e segurança. O Código 5.3 apresenta a criação de contextos em MetaH para representar atributos de processos.

Conseqüências
Algumas das vantagens do padrão Estrutura Hierárquica de Informações são as seguintes:

- **Legibilidade.** Os contextos criados permitem uma melhor visualização das informações representadas.
- **Manutenibilidade.** A adoção de um esquema hierárquico facilita a representação de dependências sobre informações e possibilita uma melhor manutenção das especificações.

Uma desvantagem do padrão Estrutura Hierárquica de Informações é a seguinte:

- **Classificação de informações.** Algumas vezes pode ser bem difícil se definir onde se colocar uma determinada informação.

5.2.7 Resumo da Linguagem de Padrões

Os padrões apresentados nesse capítulo apóiam a tomada de decisões sobre o projeto sintático de ADLs. Com eles, os projetistas de ADLs podem construir linguagens mais fáceis de serem manipuladas e, principalmente, sintaticamente fáceis de serem utilizadas. Iniciando com o padrão *Projetar ADL*, que define a necessidade de se projetar uma nova ADL para a realização de tarefas de projetos arquiteturais específicos de uma equipe de desenvolvimento, todos os padrões tratam de como se definir os elementos sintáticos necessários para a representação de projetos arquiteturais.

Nesse contexto, o padrão *Elementos Arquiteturais Básicos* é apresentado para auxiliar a definição dos tipos de elementos arquiteturais a serem especificados por uma ADL de forma

a se conseguir uma melhor reusabilidade desses elementos. Já o padrão *Organização das Especificações* trata da forma com que a ADL deve organizar os elementos arquiteturais de uma arquitetura para gerar especificações de fácil manutenção e evolução. Além disso, o problema da determinação dos tipos de informações que devem ser representadas para cada elemento especificado por uma ADL é tratado pelo padrão *Informações dos Elementos Arquiteturais*, que objetiva a criação de especificações flexíveis, no sentido de que novas informações possam ser facilmente introduzidas sem ter que modificar a ADL. Por fim, é definido o padrão *Estrutura Hierárquica de Informações* que especifica a forma pela qual as informações dos elementos arquiteturais devem ser organizadas pela ADL, de modo a aumentar tanto a legibilidade das especificações como facilitar a manutenção das mesmas.

5.3 A Linguagem ArchML

A primeira etapa no desenvolvimento de uma arquitetura de software é a descrição dessa arquitetura. Em DraX, como apresentado no capítulo 4 desenvolvemos a Linguagem ArchML exclusivamente para realizar essa tarefa. Nessa seção apresentamos ArchML. Sua sintaxe e a forma de como usá-la para descrever tanto componentes de uma arquitetura de software como a configuração desses componentes para criar um sistema complexo são mostradas através de exemplos simples onde utilizamos uma gramática informal. Também apresentamos os DDPs (Diagramas de Descrição de Protocolos) que é a forma de realizar a descrição de comportamento de componentes ArchML.

5.3.1 Objetivos de ArchML

Como o principal objetivo desse trabalho é tornar acessível o mundo da arquitetura de software através de ferramentas baseadas em tecnologias que façam parte do convívio diário da maioria dos desenvolvedores de software e também fornecer um ambiente propício ao ensino dos conceitos de desenvolvimento arquitetural, temos que, antes de tudo, utilizar uma ADL que de fato satisfaça esse requisito.

Obviamente que poderíamos utilizar algumas das tantas ADLs disponíveis atualmente, contudo, como já relatamos anteriormente, a sintaxe dessas ADLs é realmente pouco atraente para a maioria dos desenvolvedores e as ferramentas que as manipulam não são suficientes para realizar todas as etapas do desenvolvimento.

Desse modo resolvemos produzir uma nova ADL que possuísse uma sintaxe simples e que utilizasse uma linguagem que fosse bastante familiar e fácil de trabalhar e que fornecesse mecanismos para que a criação de ferramentas que manipulassem as especificações, pudessem ser facilmente produzidas.

Desse modo optamos por utilizar a linguagem XML para criar a ADL de DraX, que denominamos ArchML. Além de ser uma linguagem de fácil manipulação, com pouco construtores, contamos com a extensibilidade de XML para que ArchML possa ser ajustada às mais diversas necessidades de desenvolvimento. Além disso, como vai puder ser notado na próxima seção com a apresentação da sintaxe de ArchML, resolvemos não utilizar construtores XML que pudessem tornar as especificações menos legíveis e, conseqüentemente, menos atraentes para o desenvolvedor.

5.3.2 A Sintaxe de ArchML

A sintaxe de ArchML é baseada em XML, sendo descrita formalmente através XML Schema. Entretanto, saindo um pouco da convenção das aplicações XML, resolvemos não utilizar identificadores dos tipos ID e IDREF, pois, entre outros motivos, esses tipos de identificadores são restritivos com relação aos nomes que esses possam vir a receber, visto que esses nomes devem seguir o padrão rígido para a definição de identificadores de XML. Optamos por deixar esse tipo de verificação para etapas posteriores do desenvolvimento para que pudéssemos ficar livres para nomear os tipos de componentes. Essa mesma opção se estende a descrição de sistemas a partir das especificações de componentes. Maiores informações com relação às restrições impostas pelo uso dos tipos ID e IDREF podem ser encontradas em [119].

Também saindo dos padrões normais de notação, utilizamos nesse capítulo uma notação informal para descrever a gramática XML de ArchML, sendo que a notação formal, representada por XML Schemas, é apresentada em detalhes na Seção 6.2. A baixo segue a notação utilizada:

- A sintaxe se aproxima da descrição de documentos, com a ressalva de que os valores de atributos indicam tipos de dados ao invés de valores;
- Para indicar a cardinalidade de elementos e atributos foram usados alguns elementos utilizados com freqüência em DTDs, da seguinte forma: "?" (0 ou 1), "*" (0 ou mais), "+" (1 ou mais);
- Nomes de elementos terminados com "...", indicam que elementos ou atributos estão sendo omitidos por serem irrelevantes ao contexto; e
- Nomes em negrito, indicam que está sendo apresentado um novo elemento/atributo/valor ou que esse tem importância particular para o exemplo.

Para referenciar tipos de dados em ArchML optamos por usar os tipos definidos na especificação de tipos de dados de XML Schemas, que é definida na URL http://www.w3.org/2000/10/XMLSchema. Com isso, herdamos todos os tipos definidos nesse espaço de nomes. Para facilitar as referencias ao longo da tese, usaremos o prefixo `xsd` para identificar os tipos de dados de XML Schema.

Ao longo da apresentação das linguagens de DraX, usamos dois tipos de referencias para elementos. O primeiro tipo, `nmtoken` [16] se refere a um conjunto de caracteres que identificam um elemento/atributo. O segundo tipo, `qname` se refere a elementos/atributos definidos a priori em algum espaço de nomes.

5.3.3 A Estrutura de ArchML

A descrição de arquiteturas distribuídas se dá em duas etapas. Na primeira os componentes da arquitetura são descritos, logo em seguida a arquitetura como um todo é definida. A descrição de componentes pode ser realizada de forma distribuída, sendo que a localização dos arquivos onde essa descrição foi realizada deve estar disponível no momento da descrição da arquitetura.

Código 5.4 Estrutura Geral de um Componente em ArchML

```
<?xml version="1.0"?>
<component name="nmtoken">
   <document .../> ?
   <propertySet .../> ?
   <interfaces .../>
   <behavior .../>
</component>
```

5.3.3.1 Componentes

Na definição de componentes em ArchML podemos definir quatro blocos de informações diferentes. Inicialmente as informações documentais, que permitem a descrição de informações sobre o próprio documento. Depois temos as informações sobre propriedades dos componentes. Nesse momento, fatores relativos a execução do componente devem ser descrito, como por exemplo, qual a política de threads será utilizada, etc. Em seguida, as informações relativas às interfaces e portas de comunicação dos componentes devem ser definidos. Por fim, a informação comportamental, onde são descritos os aspectos de comportamento observável do componente com relação às suas portas definidas nas interfaces.

Por ser uma aplicação de XML, ArchML especifica componentes através de marcações específicas. O Código 5.4 apresenta a estrutura geral de um documento de descrição de um componente em ArchML.

No Código 5.4, podemos observar os elementos básicos para a definição de documentação, propriedades, interfaces e de comportamento. O atributo `name` é um atributo obrigatório e define um nome para o componente. Em ArchML, um componente funciona como um tipo que deve ser utilizado na descrição de instâncias específicas no momento da criação de arquiteturas.

Também deve ser observado a possibilidade de um componente possuir mais de uma interface. Essa faceta de ArchML, permite, como veremos em capítulos posteriores, uma maior versatilidade na utilização de um componente. Possibilitando a utilização desses componentes nos mais diversos tipos de interações.

Segue um detalhamento sobre cada elemento de ArchML.

O Elemento `document`

Esse elemento é utilizado para se descrever textualmente a funcionalidade do componente. Utilizado para fins documentais, esse elemento não é processado em nenhuma etapa do desenvolvimento de arquiteturas com ArchML e serve, apenas, para controle por parte da equipe de desenvolvimento. O elemento `document` possui a seguinte estrutura geral (Código 5.5):

Código 5.5 O elemento `document`.

```
<document> ?
   <version num="qname"/> ?
   <author name="nmtoken" email="nmtoken"/> *
   <lastUpdate value="xsd:date"/> ?
   <comments/> ?
</document>
```

Se utilizado, esse elemento pode apresentar os sub-elementos `version`, que identifica a versão do componente; `author`, que identifica o(s) autor(es) do componente com seus respectivos emails; `lastUpdate`, que indica a data da última atualização da especificação (tipo fixo é xsd:date) e `comments`, que deve apresentar uma descrição sucinta do componente.

O Elemento `propertySet`

Cada componente em ArchML irá gerar uma instância na descrição de sistemas. Essas instâncias executarão em cima de um middleware. A execução de um objeto pode ser ajustada utilizando as propriedades tanto da linguagem onde esse está implementado, como das funcionalidades do ORB e dos serviços que esse oferece. Assim um objeto pode ser executado em um ORB CORBA definindo-se um conjunto de propriedades do POA, por exemplo. Ou através das propriedades de serviços, como eventos, persistência e nomes.

O elemento `propertySet` permite definir valores para as propriedades que as instâncias do tipo de componente terão quando estiverem em execução. A descrição geral do elemento é apresentada no Código 5.6.

Código 5.6 O elemento `propertySet`.

```
<propertySet> ?
   <property context="nmtoken" propertyType="nmtoken"
             propertyValue="nmtoken"/>+
</propertySet>
```

Podemos observar a presença de um conjunto de sub-elementos `property`, cada um identificando uma propriedade diferente. O atributo `context` especifica sobre que serviço ou recurso está sendo definida uma propriedade, por exemplo, um valor possível para esse elemento poderia ser *POA*, que identificaria que a propriedade que está sendo definida se refere ao POA de um ORB CORBA. O atributo `propertyType` identifica a que propriedade relativa ao contexto estamos nos referindo. Assim, esse elemento pode assumir, por exemplo, o valor *ThreadPolicy*, que, no contexto do POA, identifica a política de threads que o POA deve ter para executar instâncias desse componente. Por fim, o atributo `value` que define o valor da propriedade.

Como pode se intuir há uma relação forte entre os atributos de uma propriedade e esses também possuem uma relação com algum serviço ou propriedade do ORB. Para identificar essas relações definimos um conjunto de valores que esses atributos podem assumir e seus

respectivos contextos e propriedades. Além disso, definimos que valores serão tratados como *default* caso não haja definição explícita para uma propriedade. A Tabela 5.3 apresenta os valores de algumas propriedades e os contextos definidos previamente em ArchML para infraestruturas de middleware CORBA, RMI e DCOM.

Contexto	**Tipo**	**Valores**	**Valor Padrão**
POA	ThreadPolicy	SingleThread MultiThread	SingleThread
NamingService	NamingContext	DU(String)	RootContext
EventService	ChannelName	DU(String)	DefaultChannel
Security	SecurityManager	DU(String)	RMISecurityManager
Identifier	UUID CLSID	DU(String)	-

Tabela 5.3: Propriedades de Componentes ArchML.

Na Tabela 5.3, **Contexto** define os valores assumidos pelo atributo `context`. Na coluna **Tipo** estão definidos os valores possíveis para `propertyType` relativa à `context`. Na coluna **Valores** estão os valores do atributo `value` e identificam os valores que os tipos podem assumir com relação a `context` e `propertyType`. Por fim, na coluna **Valor Padrão** está o valor assumido caso esse não seja definido explicitamente.

Podemos observar na Tabela 5.3 que existem algumas linhas referentes à coluna **Valores** que possuem valor definido como DU[1]. Esses valores são geralmente *strings* ou números que podem ser definidos livremente pelo usuário, como por exemplo, a identificação do contexto de nomes onde as instâncias de um determinado componente vão ser criadas.

É bom ressaltar que novas propriedades podem ser definidas pelo usuário, entretanto, para que essas possam ser de fato consideradas no momento da geração dos templates de implementação, os *scripts* de geração desses templates deverão ser modificados de forma a incorporar essas novas propriedades.

O Elemento `interfaces`

Seguindo a forma clássica de descrever componentes utilizando ADLs, também em ArchML as interfaces são encarregadas da descrição das portas que representam os serviços prestados pelo componente (portas de entrada) e dos serviços que esse venha a utilizar (portas de saída). A estrutura geral do elemento `interfaces` está representada no Código 5.7.

[1]Definido pelo Usuário

Código 5.7 O elemento `interfaces`.

```
<interfaces>
   <interface role="qname"?> +
      <port name="nmtoken"direction="in | out"returnType="qname"/> +
         <param name="nmtoken" type="qname"/> *
      </port>
   </interface>
</interfaces>
```

As portas dos componentes são definidos através de interfaces em ArchML, sendo que diversas interfaces podem ser definidas, cada uma com um conjunto de portas. A característica peculiar de ArchML com relação à outras ADLs atuais, que é a possibilidade de definir diversas interfaces em um mesmo componentes, possibilita que um componente participe de diversos tipos de interações diferentes. Essa característica será melhor apresentada quando a especificação de estilos arquiteturais em DraX for apresentada na Seção 5.4.

Cada elemento `interface` é referenciado pelo atributo `role`. Esse atributo define as diversas características de execução como por exemplo a forma de comunicação, e está estritamente ligada a definição de estilos arquiteturais. Como pode ser observado na gramática, esse atributo não é obrigatório, entretanto se ele não for definido receberá um valor fixo. Como já mencionado, todas essas características relativas a estilos arquiteturais serão apresentadas na seção 5.4.

Cada um dos elementos `interface` possui um conjunto de elementos filhos `port` que servem para definir cada uma das portas. Os elementos `port` possuem os atributos `name`, que identificam o nome pela qual a porta é referenciada; `direction`, que identifica o sentido de comunicação da porta, e tem como únicos valores possíveis *in* ou *out*, para identificar, respectivamente, portas de entrada e portas de saída; `returnType` que é um atributo não obrigatório e serve para definir o tipo de dado relativo ao valor retornado pela porta se esta for de entrada ou o valor esperado com a invocação, caso essa porta seja de saída.

Cada uma das portas pode possuir um conjunto de parâmetros de dados utilizados na comunicação. Assim, o elemento filho `param` pode ser utilizado para definir esses parâmetros. Cada elemento `param` possui os atributos `name`, que identifica o nome do parâmetro e `type`, que identifica o tipo (XSD) do parâmetro.

O Elemento `behavior`

O *framework* DraX tem como um de seus objetivos, permitir a especificação e validação de arquiteturas e de estilos arquiteturais distribuídas. Com a na fase de validação, apenas os dados sintáticos não são suficientes [30] é necessário a utilização de características comportamentais [108, 107]. Assim é possível em ArchML descrever o comportamento dos componentes. Essa descrição diz respeito ao comportamento observável com relação às portas definidas nas interfaces do componente [107].

A abordagem atual tomada para a descrição de comportamentos em ADLs é de se lançar mão de formalismos como CSP [7]. Entretanto, esses formalismos dificultam a adoção dessas ADLs para especificação da maioria dos softwares e para projetos de curto e médios prazos. Dessa forma, seguindo os objetivos de DraX em adotar apenas ferramentas presentes no cotidiano de desenvolvimento atual, resolvemos utilizar UML para realizar essa tarefa de

especificação de comportamentos. Usando a notação de diagramas de estados UML podemos capturar o comportamento observável com relação às portas de um componente [32].

Uma tendência atual na utilização de UML para especificação de projetos de software é fornecer uma representação dos diagramas que seja facilmente interoperável entre ferramentas diversas. Assim, XMI (*XML Metadata Language*) [52] entra em cena, sendo que complexos e pesados diagramas UML podem agora ser mapeados em pequenos arquivos textos no formato XMI. A transformação de um diagrama UML em uma especificação XMI é realizada atualmente pelas maioria das ferramentas de projeto baseadas em UML. Dessa forma optamos por utilizar em ArchML as informações XMI relativas à descrição comportamental dos componentes através de diagrama de estados. Sendo que o elemento `behavior` da descrição de um componente serve para identificar o arquivo XMI correspondente à essas descrição. Esse elemento tem a estrutura representada pelo Código 5.8.

Código 5.8 O elemento `behavior`.

```
<behavior xlink:href="uri"/>
```

A criação de diagramas UML para a representação de comportamentos de componentes é apresentada em detalhes mais adiante nesse capítulo na Seção 5.3.4.

Uma observação importante é que, na fase de realização de análises nas arquiteturas e estilos, as informações XMI relativas ao comportamento de componentes serão mapeadas em especificações formais em π-cálculo. Sendo que, de fato, essas especificações é que serão utilizadas na realização de análises. Contudo, o desenvolvedor não entrará em contato com essas especificações, visto que DraX possui scripts de geração de especificações π-cálculo a partir de definições XMI. Todas essas características relativas à realização de análises formais em arquiteturas e estilos arquiteturais em DraX serão apresentados em detalhes na Seção 6.3.

5.3.3.2 Sistemas

A partir da definição de componentes podemos definir estruturas arquiteturais complexas através da instanciação desses tipos e da especificação das conexões entre as portas dessas instâncias. Um Sistema, nome dado a essa estrutura em ArchML, é a descrição estrutural de uma arquitetura distribuída. Eventualmente referenciaremos por arquitetura os sistemas ArchML, visto que esses termos são sinônimos.

O Código 5.9 apresenta o formato geral de uma especificação de sistemas em ArchML. Podemos observar a presença de três elementos, além do elemento principal, que possui um atributo name que define o nome do sistema.

Código 5.9 Estrutura Geral de um Sistema em ArchML.

```
<?xml version="1.0"?>
<system name="nktoken">
   <document .../> ?
   <style .../> ?
   <types .../>
   <instances .../>
   <links .../>
</system>
```

O Elemento `document`

Esse elemento tem a mesma estrutura do elemento `document` de `component` e está representado no Código 5.5. A função desse na descrição de um sistema ArchML é descrever informações sobre o próprio sistema.

O Elemento `style`

A função do elemento `style` (Código 5.10) é descrever o estilo que o sistema que está sendo descrito está seguindo. A descrição completa desse elemento está no Código 5.10. Podemos observar a presença de um único elemento filho (`styleRef`) que identifica, através de um XLink [23], qual a especificação do estilo está sendo utilizada. Esse elemento também será melhor detalhado quando o conceito de estilos arquiteturais for introduzido em DraX na Seção 5.4.

Código 5.10 O elemento `style`.

```
<style>
   <styleRef xlink:href="uri"/>
</style>
```

O Elemento `types`

Esse elemento serve para referenciar os tipos utilizados na criação do sistema. O Código 5.11 apresenta a definição completa de `types`. Esse elemento é formado por um conjunto de elementos filho `type`. Como a intenção principal de ArchML é a criação de especificações arquiteturais de forma distribuída, cada um desses elementos filhos representa a referência de um tipo definido externamente e referenciado através de um XLink. Esses elementos filhos possuem os atributos `name`, que identifica o nome com que esse tipo vai ser tratado na especificação da arquitetura, e um XLink para a especificação real.

Código 5.11 O elemento `types`.

```
<types>
   <type name="nmtoken"
      xlink:href="uri">
   </type> +
</types>
```

O Elemento `instances`

O elemento seguinte na especificação de sistemas em ArchML é o elemento `instances`. Através desse elemento são identificadas as instâncias que formam a arquitetura. O Código 5.12 apresenta a descrição completa desse elemento. O elemento `instances` é formado por diversos elementos filhos `instance`, que identificam instâncias específicas. Cada um desses elementos possui dois atributos. O primeiro é `name`, que identifica o nome pelo qual a instância será tratada na arquitetura. O segundo é `typeRef`. Esse atributo é uma referência para o nome do tipo do componente a partir do qual a instância em questão é definida. Esse nome tem que ter sido definido dentro da própria especificação do sistema como um elemento `types/type`.

Código 5.12 O elemento `instances`.

```
<instances>
   <instance name="nmtoken" typeRef="qname">
   </instance> +
</instances>
```

O Elemento `links`

Tendo sido definido os tipos e instâncias que formam a arquitetura, basta agora descrever a topologia de conexões das portas dessas instâncias. O elemento `links` tem essa função. O Código 5.13 apresenta a especificação completa desse elemento. Como pode ser observado esse elemento é formado por um conjunto de elementos filhos `link`, cada um representando uma conexão específica.

Código 5.13 O elemento `links`.

```
<links>
   <link name="conn1"> +
      <point instRef="qname" portRef="qname">
      <point instRef="qname" portRef="qname">
   </link>
</links>
```

Cada elemento `link` possui um atributo `name`, que identifica o nome do link. Cada `link`, por sua vez, possui dois elementos filhos `point`. Cada elemento `point` identifica uma

extremidade do link. Esses elementos possuem dois atributos. O primeiro, `instRef`, é uma referência para a instância do componente no qual o link possui uma extremidade. O segundo é `portRef`, que é uma referência ao nome da porta da instância em questão que participa do link.

5.3.4 Comportamento de Componentes

A definição do comportamento de componentes em ArchML permite realizar verificações mais refinadas nas arquiteturas [106, 107], principalmente no que diz respeito a utilização de estilos arquiteturais. Como foi apresentado anteriormente nesse capítulo, o elemento `behavior` é utilizado para realizar a anexação de uma especificação de comportamento em uma especificação de componentes em ArchML.

Como levantamos no início dessa tese, para realizar algum tipo de análise em arquiteturas, a abordagem clássica é definir ADLs que tenham uma base formal como Wright [7] que é baseada em CSP. Como salientamos na motivação desse trabalho, esses formalismos não fazem parte da vida da maioria dos desenvolvedores de software. Nosso objetivo com DraX é permitir com que tenhamos as mesmas vantagens das ADLs formais utilizando técnicas informais ou semi-formais através de ferramentas de domínio público. Assim, introduzimos a idéia de também realizar a descrição comportamental das arquiteturas e escolhemos para isso os diagramas de estados UML.

UML é uma linguagem de modelagem de projetos orientados a objetos bastante difundia da industria de software atualmente. Os diagramas de estados UML permitem se modelar sistemas complexo de transições e fatores como sincronização e paralelismo. Entretanto, em DraX, necessitamos apenas do poderio dos diagramas de estados para modelar o protocolo de comunicação de um componente com relação às suas portas. Assim, baseado nos diagramas UML propomos um novo tipo de diagrama que captura mais facilmente a noção de protocolos que precisamos. Os Diagramas de Descrição de Protocolos (DDP). Os DDPs são uma especialização dos diagramas de estados UML onde introduzimos algumas regras sintáticas na descrição das transições, um estado novo denominado `compute`, que identifica uma computação interna não observável de um componente e utilizamos a idéia de estereótipos de UML para modelar comunicações síncronas e assíncronas.

As modificações que introduzimos nos diagramas de estados não desvirtuam a idéia básica desses diagramas, sendo que os DDPs podem, e devem, ser modelados usando ferramentas UML. Apenas algumas novas regras devem ser seguidas de forma a simplificar o processo de descrição do protocolo, visto que a utilização dos diagramas de estados UML diretamente podem gerar especificações muito complexas e difíceis de serem verificadas e principalmente, comparadas com outras especificações (necessitaremos disso na verificação comportamental de aderência de um componente a um determinado estilo arquitetural).

Além disso, os diagramas de estados UML são alvo de diversos tipos de pesquisa relativa à análise comportamental [54, 85] onde as representações desses diagramas são mapeadas em especificações algébricas como CSP/FDR [47] de modo a permitir a realização de análises formais. Dessa forma, usaremos as mesmas regras de mapeamento dos DDPs para álgebras de processos.

ArchML utiliza DDPs como forma de descrever comportamento de componentes. Entretanto, como gráficos UML não são facilmente transmitidos pela rede (grandes especificações ocupam um espaço razoável com relação à especificação ArchML) nem são fáceis de serem automaticamente mapeados em especificações algébricas, como vimos anteriormente. Desse modo, resolvemos utilizar a representação XMI correspondente para os diagramas de

estados que representam o comportamento dos componentes. Essa especificação pode ser gerada pelas ferramentas que trabalham com UML atualmente, visto que XMI é o formato de intercâmbio padrão do OMG para UML e podem ser aplicadas sem problemas aos DDPs.

Utilizando uma especificação XMI tanto ganhamos na facilidade de interoperabilidade das especificações como na possibilidade de utilizar ferramentas como XSLT para gerar especificações algébricas automaticamente. Esse processo será detalhado quando falarmos de verificação e análise comportamental de arquiteturas e estilos em DraX na seção 6.3.

Construção dos Diagramas de Descrição de Protocolos

As regras de construção de especificações π-cálculo a partir de diagramas UML foram propostas em [32]. Nesse trabalho é indicado a necessidade da utilização de diagramas de seqüência e diagrama de estados para que se possa realizar uma especificação π completa, sendo que os diagramas de estado fornecem informações de estados e transições, que são mapeadas em agentes e comunicações em canais, respectivamente. Entretanto, a seqüência de comunicações não é fornecida pelos diagramas de estado, sendo que os diagramas de seqüência podem resolver o problema.

Contudo, em DraX, a realização da especificação de um componente ArchML é totalmente independente da especificação de outros componentes, sendo que é impossível a priori se construir um diagrama de seqüência para o sistema como um todo. Dessa forma, propomos os Diagramas de Descrição de Protocolos (DDPs) que impõem algumas regras e restrições na descrição dos diagramas de estados de forma a conseguir também a descrição do sequenciamento de ações, além de permitir a construção mais fácil de complexos protocolos de componentes.

Os DDPs utilizam a idéia de modelar apenas o comportamento observável em relação às portas definidas no componente. Essa abordagem foi escolhida pelo fato de necessitarmos que essa especificação seja transformada em uma especificação algébrica pelas ferramentas de DraX, e as álgebras de processos usam esse enfoque para descrever processos comunicantes. Cada DDP é formado pelos seguintes itens:

1. Regras de escrita de transições:
   - **a.** Nomes das portas de entrada do componente;
   - **b.** Nomes das portas de saída do componente;
   - **c.** O nome `compute`, que indica uma transição para um estado de computação interno e não observável do componente; e
   - **d.** Números para indicar a seqüência de transições. Ex.: 1:send, 2:getFile.
2. Estereótipos para definir o modo de comunicação das portas:
   - **a.** `Synchronous`: define uma comunicação síncrona; e
   - **b.** `Asynchronous`: define uma comunicação assíncrona.

Além desses itens, devemos seguir um conjunto de regras para descrever o protocolo de um componente ArchML. A seguir, essas regras são apresentadas e explicadas. Ao longo das explicações utilizaremos um exemplo de um componente denominado Buffer. Esse componente possui uma porta de entrada (**readData**) e uma porta de saída (**sendData**) e seu comportamento é o de um buffer de duas posições.

Regras de Construção de Diagramas de Descrição de Protocolos:

1. O estado inicial da descrição do protocolo de um determinado componente deve ter o mesmo nome do componente o qual estamos modelando o protocolo

   Ex.: A Figura 5.2 mostra a configuração inicial da descrição de um componente denominado Buffer.

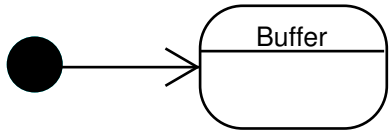


Figura 5.2: Exemplo de DDP.

2. Qualquer transição do componente que represente comportamento observável, mas que não utilize as suas portas deve ser representado por um estado denominado `compute` que deve ser disparado pela transição também de mesmo nome.

   Ex.: Se o componente Buffer, no momento que iniciar sua execução, realizar uma ação de ler um arquivo do disco com as configurações do buffer sem que nenhum tipo de invocação em suas portas seja realizada, sua descrição de protocolo teria a seguinte configuração:

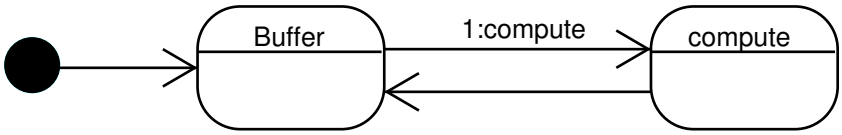


Figura 5.3: DDP com ação interna.

3. Os eventos a serem modelados são apenas o envio ou recebimento de dados por alguma porta do componente.

   Ex.: A representação do comportamento de Buffer com relação a sua porta readData pode ser o seguinte:

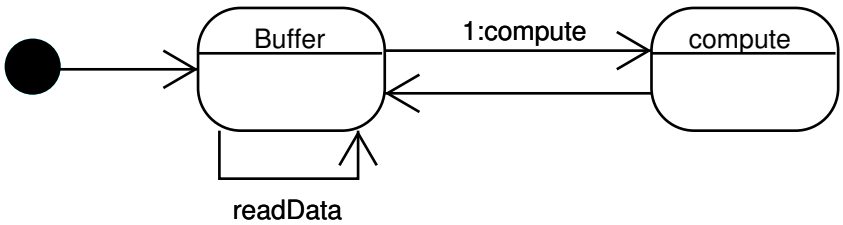


Figura 5.4: Relação DDP x porta do componente.

4. Para incorporar informações de sequenciamento de ações nos DDP, se um componente possui mais de uma transição de saída que são disparadas obedecendo uma cronologia,

essas transições devem ser numeradas de acordo com essa ordem cronológica de acontecimentos usando números inteiros > 0. Se essas escolhas forem não determinísticas (escolhas decorrente do estado interno do componente), elas não necessitam ser numeradas.

Ex.: Em Buffer, uma seqüência de comunicação pode ser representada nas transições compute e readData, ambas saindo do componente Buffer. Já as transições que saem de Buffer1, não são numeradas por que são não determinísticas:

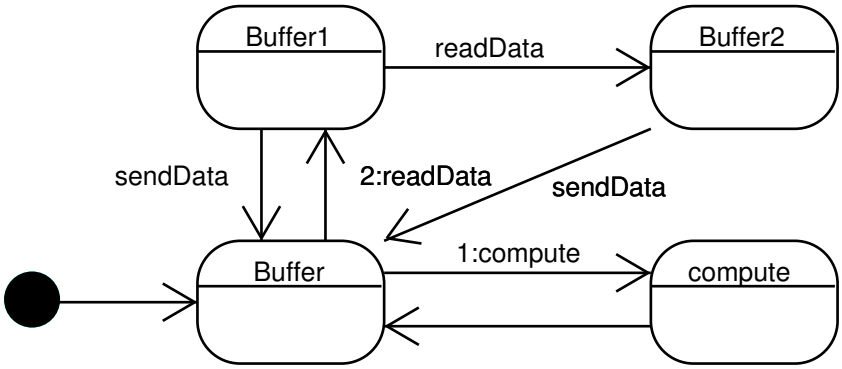


Figura 5.5: Sequenciamento em DDPs.

5. O modo de comunicação dos componentes com relação às suas portas deve ser definido utilizando estereótipos UML. Assim, as comunicações síncronas devem ser indicadas com o estereótipo `<<synchronous>>` e as comunicações assíncronas devem ser indicadas com o estereótipo `<<asynchronous>>`. O valor *default*, caso não explicitado, é `<<synchronous>>`.

   Ex.: Supondo agora que o componente Buffer envia seus dados pela porta **sendData** utilizando uma comunicação assíncrona, teremos o seguinte diagrama:

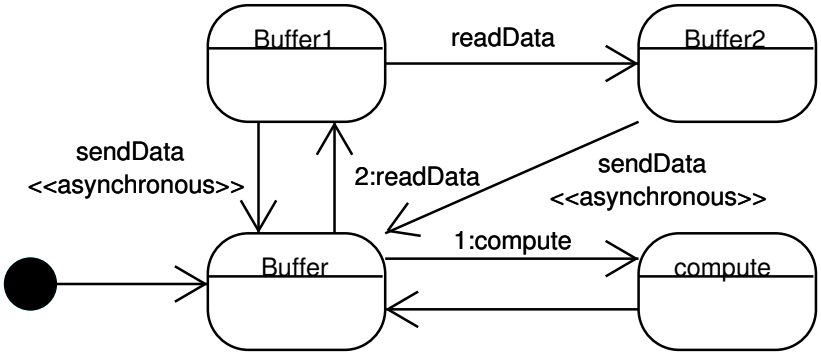


Figura 5.6: Estereótipos em DDPs.

Para construir um arquivo XMI a partir de uma especificação de um DDP devemos utilizar uma ferramenta UML comum que consiga realizar a tarefa de salvar as informações em formato XMI. Além disso, devemos atentar para as seguintes regras para a criação de um projeto completo de um protocolo:

1. Criar uma classe UML (no diagrama de classes) com o mesmo nome do componente, onde os métodos dessa classe devem ter os mesmos nomes das portas declaradas na especificação ArchML, sendo que as portas de entrada, são representadas por métodos públicos na classe e as portas de saída devem ser representadas por métodos privados. Apenas o nome da classe e o nome dos métodos precisam ser definidos.

2. Criar um diagrama de estados para essa classe utilizando as características dos Diagramas de Descrição de Protocolos descritas anteriormente.

3. Gerar o arquivo XMI a partir de somente essa especificação. Ou seja, deve ser criado um projeto novo para cada componente.

É importante salientar que todas essas regras servem tanto para facilitar a descrição do protocolo (por exemplo, uma comunicação síncrona exigiria um estado de espera do envio do pedido, que não precisamos modelar no DDP pois usamos estereótipos para as transições e incorporamos informações semânticas a elas), como na geração das especificações algébricas correspondentes, como veremos na Seção 6.3.

5.4 A Linguagem Xtyle

Como apresentado no Capítulo 2, o termo Estilos Arquiteturais de Software se refere a "um conjunto de regras de projeto que identifica os tipos de componentes e conectores que devem ser utilizados para compor um sistema ou subsistema, juntamente com um conjunto de restrições globais e locais na forma como essa composição deve ser realizada" [100].

Garlan em [43], apresenta de forma mais detalhada um conjunto de aspectos de estilos arquiteturais que devem ser utilizados por qualquer modelo que trabalhe com esse conceito. Ele apresenta os seguintes aspectos:

- **Vocabulário**
  Deve ser adotado um conjunto de valores a serem definidos para os componentes de uma arquitetura de forma a permitir a verificação de compatibilidade com relação a um estilo, que também deve ser identificado, também por convenção, na descrição do sistema como um todo. Esses valores definem papéis que os componentes devem assumir com relação ao estilo requerido.

  Dessa forma, supondo que um sistema deva seguir o estilo arquitetural definido como Pipeline, os componentes que formam esse estilos devem possuir interfaces definidas com tipos como Filter, Source ou Sink. Assim, esses componentes assumem os papéis devidos com relação ao estilo.

- **Definição de Restrições Arquiteturais**
  Além de definir as convenções sintáticas que permitem a verificação da compatibilidade dos tipos das interfaces dos componentes utilizados com o estilo arquitetural do sistema como um todo, também devemos criar um conjunto de regras topológicas de forma a garantir que esses componentes estejam respeitando as restrições definidas pelo estilo.

> Assim, um sistema que segue o estilo Pipeline, por exemplo, deve possuir um componente do tipo Source que esteja conectado com um componente do tipo Filter que, por sua vez deve ser conectado com um componente do tipo Sink. Além disso, a comunicação deve ser realizada sem ciclos e de forma unidirecional.

A idéia de estilos arquiteturais nas ADLs atuais ou é muito informal ou é demasiadamente formal. Em UniCon [101], por exemplo, existe um conjunto de restrições sintáticas que devem ser utilizados para a criação de arquiteturas que sigam determinados estilos. Contudo, essas restrições são baseadas em um conjunto de tipos de papéis (*roles*) que componentes podem assumir e esses tipos são fixos, tornando a linguagem restrita aos tipos disponíveis. Já em Wright [7], o desenvolvedor pode produzir seus próprios estilos mas para isso deve lançar mão de especificações algébricas baseadas em CSP o que não é um atrativo para os desenvolvedores.

Dessa forma, resolvemos desenvolver uma linguagem específica para a especificação de estilos arquiteturais em DraX. Através dessa linguagem, que denominamos Xtyle, podemos descrever tanto informações estruturais como comportamentais.

Nessa seção apresentamos Xtyle e mostramos a aplicação dessa linguagem na descrição dos principais estilos arquiteturais de DraX. As informações relativas à definição de restrições arquiteturais na criação de estilos serão apresentadas em detalhes na Seção 6.2, que trata da verificação sintática de especificação. Optamos por essa disposição a fim de tanto facilitar o entendimento de descrições de arquiteturas baseadas em estilos arquiteturais, como para concentrar em um só capítulo todas as informações relativas à verificação de consistência e validação tanto de arquiteturas como de estilos arquiteturais.

5.4.1 Estilos Arquiteturais em DraX

Optamos por usar a idéia de classificar os estilos de DraX tendo como parâmetro a existência de um vocabulário e de regras de configuração pré-definidos. Dessa forma, também pudemos desenvolver *scripts* de validação e verificação em DraX para tratar esses estilos. Baseado nessa idéia, definimos três tipos de estilos: Estilos Básicos, Estilos Derivados e Estilos Definidos-Pelo-Usuário.

Os estilos básicos fornecem um conjunto de componentes descritos através de um vocabulário e comportamentos pré-definidos para cada componente participante desse estilo. Esses estilos, encontram um conjunto de *scripts* definidos em DraX para a validação de arquiteturas baseadas neles. Isso é possível pelo fato de esses serem estilos clássicos e de características relativamente bem definida.

Os estilos derivados são os que herdam as definições (vocabulários e comportamentos) de outros estilos definidos a priori. Esse tipo de estilo possui uma sub-classificação definida por: Derivados Simples e Derivados Compostos. Os estilos Derivados Simples herdam as características de apenas um estilo específico, alterando algumas de suas características. Já os estilos Derivados Compostos são os que herdam de mais de um estilo. Esse estilos também podem usar o ferramental disponível em DraX para realizar verificação em arquiteturas.

Por fim, temos os estilos Definidos-Pelo-Usuário. Esses estilos são os que os usuários podem criar a seu critério, ou seja, não existe suporte a priori para validação desse estilos em DraX, entretanto, como sabemos da importância de permitir o usuário criar estilos próprios, fato que não é contemplado pela quase totalidade das ADLs atuais [40], fornecemos em DraX, *scripts* genéricos geradores de *scripts* específicos para a validação desse tipo de estilo. Assim, o usuário fica livre para especificar seus estilos e as ferramentas de DraX criam *scripts* de

validação para esses novos estilos. Essa é uma das principais contribuições de Xtyle para o campo de estilos arquiteturais.

A descrição detalhada de todos os estilos atualmente pré-definidos em DraX pode ser encontrada em [113], a seguir, na Tabela 5.4, apresentamos um resumo dos estilos Básicos de DraX e na Tabela 5.5 apresentados os estilos Derivados. Nela podemos observar as informações sobre os estilos relativas a vocabulários, as restrições topológicas e sobre informações sobre os comportamentos dos componentes, sendo essas informações são definidas nas seguintes colunas da Tabela 5.4:

- **Estilo**: O nome do estilo é definido.
- **Tipos Componentes**: Aplicado aos estilos Básicos de DraX, nessa coluna os componentes que formam o estilo são apresentados.
- **Origem da Derivação:** Aplicado apenas aos estilos derivado de DraX, nessa coluna é apresentado o nome do estilo do qual ocorre a derivação. Caso o estilo seja resultado de uma Derivação Composta, os nomes de todos os estilos são apresentados.
- **Restrições Componentes**: As restrições relativas aos componentes (quantas portas pode ter, quantas conexões cada porta deve suportar, etc) são apresentadas nesse momento.
- **Restrições Comunicação/Topológicas**: As restrições relativas à comunicação (quais são os tipos de componentes que podem se comunicar diretamente) e topológicas (qual as topologias permitidas) são apresentadas nessa coluna.
- **Fluxo Dados/Controle**: Nesse item são apresentadas informações sobre o fluxo de dados (em que sentido(s) os dados caminham) e sobre o fluxo de controle (o dado é requisitado, os dados são enviados sem necessidade de pedidos, etc.).

Estilo	**Tipos Componentes**	**Restrições Componentes**	**Restrições Comunicação/ Topológicas**	**Fluxo Dados/Controle**
Rede de Fluxo de Dados (*Dataflow-Network*)	- Source - Filter - Sink	- Source: Uma porta de entrada e uma ou mais portas de saída; - Filter: Uma ou mais portas de entrada ou saída; - Sink: Uma ou mais porta de entrada e uma de saída.	- Conexões permitidas: Source com Filter e Filter com Sink; - Permitido ciclos.	- Modo assíncrono; - Unidirecional; - Dados no Sentido de Source para Filter e de Filter para Sink; - Controle push.
Sistema de Eventos (*Events*)	- Supplier - Consumer	- Supplier: Pelo menos uma porta de saída; - Consumer: Pelo menos uma porta de entrada	Consumidores só devem receber notificações que tenham interesse.	- Modo assíncrono; - Unidirecional; - Dados no sentido de Producer para Consumer; - Controle push.
Cliente-Servidor (*Client-Server*)	- Client - Server	- Client: Uma ou mais portas de saída - Server: Uma ou mais portas de entrada	- Server pode receber pedidos de diversos Clients; - Clients pode realizar pedidos a vários Servers.	- Modo síncrono; - Bidirecional; - Dados do Servidor para Cliente; - Controle do Cliente para Servidor.
Dados Compartilhados (*Shared-Data*)	- Module - Shared-Data	- Module: Pelo menos uma porta de saída. - Shared-Data: Apenas uma porta de entrada	- Topologia em estrela.	- Modo síncrono; - Unidirecional; - Dados nos dois sentidos; - Controle de Module para Shared-Data.

Tabela 5.4: Estilos Básicos de DraX.

Estilo	**Origem da Derivação**	**Restrições Componentes**	**Restrições Comunicação/ Topológicas**	**Fluxo Dados/Controle**
Pipeline	Rede de Fluxo de Dados (*Dataflow-Network*)	- Source: exatamente uma porta de saída. - Sink: exatamente uma porta de entrada. - Filter: devem ter exatamente uma porta de entrada e uma de saída	- Topologia linear - Acíclica	- Unidirecional - Comunicação síncrona
Repositório Ativo (*Active-Blackboard*)	Dados Compartilhados (*Shared-Data*)	Mesmas de Dados Compartilhados	Mesmas de Dados Compartilhados	- Bidirecional - Comunicação pode ser iniciada tanto por *Module* como por *Shared-Data*
PullEvent	Sistema de Eventos (*Events*)	- Supplier: Pelo menos uma porta de entrada. - Consumer: Pelo menos uma porta de saída.	Mesmas de Sistema de Eventos	- Segue o modelo *Pull*
JustoCunha	Cliente-Servidor (*Client-Server*)	Inclusão dos Componentes S-R, R-S e GServer com várias portas de entrada e saída.	- S-R: Manda dados depois recebe. - R-S: recebe dados depois manda. - GServer: Funciona como cliente depois de receber pedido.	- Assíncrono - Unidirecional

Tabela 5.5: Estilos Derivados de DraX.

Estilo	**Origem da Derivação**	**Restrições Componentes**	**Restrições Comunicação/ Topológicas**	**Fluxo Dados/Controle**
Pipeline Event	Pipeline + Sistema de Eventos	- Source = Producer - Sink = Consumer - Filter = Producer e Consumer	Mesmas de Pipeline	- Assíncrona - Push
Rede de Fluxo de Dados Compartilhados (*Shared-Dataflow*)	Rede de Fluxo de Dados + Dados Compartilhados	- Source = Module - Sink = Module - Filter = Module	Mesmas de Rede de Fluxo de Dados	- Bidirecional

Tabela 5.5: Estilos Derivados de DraX.

Código 5.14 Usando Estilos em ArchML.

```
<?xml version="1.0"?>
<system name="Main">
   <style href="Pipeline.sty"/>
   ...
</system>
```

5.4.1.1 Aplicando Estilos Arquiteturais em ArchML

Para se definir que uma arquitetura em ArhchML segue um determinado estilo arquitetural devemos introduzir nos documentos que descrevem essas arquiteturas informações relativas ao estilo que se quer produzir. Dessa forma, tanto os documentos que descrevem os componentes como os sistemas devem possuir informações relativas aos estilos adotados pela arquitetura.

Sistemas

Para que possamos realizar verificações com relação a aderência de uma arquitetura descrita em ArchML a um estilo arquitetural, devemos identificar qual estilo essa arquitetura utiliza. A associação de um sistema a um determinado estilo é realizada pelo elemento style. Esse elemento deve ser descrito antes da definição dos tipos de componentes utilizados na arquitetura. O Código 5.14 mostra a associação de um estilo Pipeline em um arquitetura.

O elemento `style` possui um único atributo `href` no qual o nome do arquivo onde a descrição do estilo está definida.

Componentes

Como apresentado na Seção 5.3, cada componente ArchML deve possuir uma ou mais interfaces, onde são descritas portas pelas quais a comunicação acontece. Como as conexões são realizadas entre portas, para identificar um determinado papel que um componente deve tomar na interação deve ser definido um valor para o atributo role na descrição da interface do componente que identifique um tipo de componente em um determinado estilo. O Código 5.15 apresenta um componente denominado CodeGen que possui duas interfaces com papéis (*roles*) diferentes.

Quando esse componente é utilizado nas descrição de uma arquitetura, os `roles` dos componentes cujas portas estão conectadas devem ser condizentes com o estilo arquitetural adotado pela arquitetura. Ou seja, se a arquitetura adotou um estilo Pipeline, todos os componentes que formam essa arquitetura devem possuir interfaces com `roles` Source, Filter ou Sink e as restrições de conexão devem ser avaliadas com relação as imposições do estilo. Esse processo de avaliação de aderência de uma arquitetura a um estilo será apresentado na Seção 6.2.

5.4.2 A Sintaxe de Xtyle

Até agora a descrição de estilos arquiteturais está restrita à informalidade da prosa. Contudo, podemos observar que na descrição das arquiteturas em ArchML definimos uma referência a um arquivo externo onde o estilo está definido para que se possam ser realizadas as verificações

Código 5.15 Componente usando Vários Estilos.

```
<?xml version="1.0"?>
<component name="CodeGen">
   <interfaces>
      <interface role="Filter">
         <port name="inport" direction="in">
            <param name="data" type="xsd:string"/>
         </port>
      </interface>
      <interface role="Producer">
         <port name="outport" direction="out">
            <param name="data" type="xsd:string"/>
         </port>
      </interface>
   </interfaces>
   <behavior href="CodeGen.xmi"/>
</component>
```

e validações devidas com relação a aderência de uma arquitetura a um estilo. Portanto, esse estilo deve ser definido nesse arquivo.

Uma característica marcante de DraX é possuir uma linguagem específica para descrever estilos arquiteturais. A linguagem Xtyle, permite descrever tanto o vocabulário permitido pelo estilo, como as restrições de comunicação e topológicas. Essa é uma das contribuições orginais dessa tese para a área de estilos arquiteturais. Atualmente as abordagens utilizadas na descrição de estilos se polarizam em descrições informais e sujeitas a todas as imprecisões dos textos escritos [46] às descrições formais e difíceis de serem de fato implementadas. Contudo as duas visões são importantes [7]. No informalismo somos capazes de entender o funcionamento de um estilo, no formalismo podemos avaliar uma arquitetura que segue o estilo e examinar diversas propriedades que seriam impossíveis, ou no mínimo, muito difíceis de avaliar sem o uso de modelos matemáticos.

Baseado nestes pontos, resolvemos que Xtyle permitiria tanto a informalidade na descrição de vocabulários e regras de comunicação como o formalismo algébrico para a verificação de arquiteturas que usam um estilo. Assim, seguindo a mesma estrutura de ArchML, em Xtyle, usamos XML para descreve as estruturas sintáticas de um estilo e as restrições topologias e de comunicação. Essas informações sintáticas serão utilizadas para realizar verificações e validações em arquiteturas que utilizem o estilo. Essas verificações serão apresentadas na Seção 6.2.

Também, usando a idéia de máquinas de estados, descrevemos o comportamento geral de cada componente que forma a arquitetura e utilizaremos a representação XMI dessas máquinas de estados na descrição dos componentes do estilo. Essas representações XMI servirão de base para a geração de especificações em álgebras de processos de forma a realizar a verificação formal desses estilos (essa etapa será apresentada na Seção 6.3).

Da mesma forma que fizemos com ArchML, nesse seção apresentaremos informalmente a linguagem Xtyle, sendo que sua representação formal é apresentada na Seção 6.2. Utilizaremos, portanto, a mesma notação que usamos em ArchML para descrever os elemento de Xtyle.

Código 5.16 Estrutura Geral de uma Descrição Xtyle.

```
<?xml version="1.0"?>
<xtyle name="nmtoken">
   <document/>
   <uses/>?
   <types/>
   <topology/>
</xtyle>
```

Código 5.17 O Elemento `document`.

```
<document> ?
   <version num="qname"/> ?
   <author name="nmtoken" email="nmtoken"/> *
   <lastUpdate value="xsd:date"/> ?
   <comments/> ?
</document>
```

A estrutura geral de uma especificação Xtyle é a apresentada no Código 5.16. Podemos observar os elementos básicos de uma especificação de estilos se referem a: documentação, tipos e topologia.

O elemento `xtyle`, que é a raiz da especificação, possui um atributo `name`, que serve para identificar o estilo que está sendo descrito. O elemento `document`, que também é usado em ArchML, serve para definir informações sobre a realização da especificação. O elemento `types`, permite a identificação do tipos de componentes permitidos para o estilo e o elemento `topology`, permite se definir aspectos topológicos.

5.4.2.1 O Elemento `document`

Esse elemento é utilizado para se descrever textualmente a funcionalidade do estilo. Utilizado para fins documentais, esse elemento não é processado em nenhuma etapa e serve, apenas, para controle por parte da equipe de desenvolvimento. O elemento `document` possui a estrutura geral apresentada no Código 5.17:

Se utilizado, esse elemento pode apresentar os elementos filhos `version`, que identifica a versão do componente; `author`, que identifica o(s) autor(es) do componente com seus respectivos `emails`; `lastUpdate`, que indica a data da última atualização da especificação (tipo fixo é xsd:date) e `comments`, que deve apresentar uma descrição sucinta do componente.

5.4.2.2 O Elemento `uses`

Quando descrevermos um estilo derivado, temos que definir de qual(is) estilo(s) ele deriva. O elemento `uses` serve para esse fim. A sintaxe desse elemento é apresentada no Código 5.18.

O elementos `uses` é formado por vários elementos do tipo `style`, que identifica um estilo dos quais estamos herdado a especificação. Cada elemento `style`, possui um `name`, que identifica

Código 5.18 O Elemento `uses`.

```
<uses>
   <style name="nmtoken" href="nmtoken"/> +
</uses>
```

Código 5.19 O Elemento `types`.

```
<types>
   <type name="nmtoken" from="nmtoken"?> +
   <alias name="nmtoken" from="nmtoken"/> *
      <ports>
         <in  minOccurs="xsd:nonNegativeInteger | *"?
              maxOccurs="xsd:nonNegativeInteger | *"?
              minConns="xsd:nonNegativeInteger | *"?
              maxConns="xsd:nonNegativeInteger | *"?
              mode="sync | async "?/> ?
         <in  minOccurs="xsd:nonNegativeInteger | *"?
              maxOccurs="xsd:nonNegativeInteger | *"?
              minConns="xsd:nonNegativeInteger | *"?
              maxConns="xsd:nonNegativeInteger | *"?
              mode="sync | async "?/> ?
      </ports>
      <behavior href="Filter.xmi"/>
   </type>
</types>
```

o nome que será usado na especificação para referenciar o estilo herdado e o atributo `href` pelo qual podemos definir qual arquivo o estilo que estamos usando está descrito.

5.4.2.3 O Elemento `types`

O vocabulário do estilo é definido nesse elemento. Além das informações sobre possíveis restrições nos componentes que utilizam esses tipos. O Código 5.19 apresenta a estrutura geral de especificação desse elemento.

O elemento `types` é formado por um conjunto de elementos `type`. Cada `type` identifica um tipo diferente de componente permitido no estilo. O atributo `name` de `type`, identifica o nome que o tipo recebe. Esses nomes seguem a descrição realizada nas Tabelas 5.4 e 5.5. Podemos observa um atributo `from`. Esse atributo é usado na descrição de estilos derivados para indicar de qual estilo estamos herdando o tipo, visto que um estilo pode herdar de diversos outros. Além disso, o elemento filho `alias` permite se definir as correlações entre tipos em um estilo derivado. Esse elemento será melhor entendido quando apresentarmos a aplicação de Xtyle na descrição dos estilos de DraX.

Cada elemento `type`, também possui como elementos filhos, os elementos `in` e `out`. O elemento `in`, define que o tipo deve possuir portas de entrada, sendo que os atributos desse elemento indicam as restrições relativas a essas portas. O mesmo ocorre para o elemento `out`, que descrever portas de saída.

Código 5.20 O Elemento `topology`.

```
<topology>
   <link name="nmtoken" from="nmtoken"?
         start="nmtoken" end="nktoken"
         controlType="push|pull"?  /> +
</topology>
```

Os elementos `in` e `out` possuem diversos atributos opcionais. O atributo `minOccurs`, define a quantidade mínima de portas de entrada/saída os componentes que seguem esse tipo devem possuir, sendo que o valor default é 1 e pode também ser definido o valor "*" para um número ilimitado de portas. O mesmo ocorre para o atributo `maxOccurs`, que define o número máximo de portas de entrada/saída que o componente que segue esse tipo deve possuir. O atributo `minConns` e `maxConns` identificam, respectivamente, o número mínimo é máximo de conexões que uma porta de entrada/saída deve permitir. Já o atributo `mode`, define o modo de comunicação das portas, podendo assumir os valores *sync* ou *async* para os modos síncrono e assíncrono, respectivamente.

5.4.2.4 O Elemento `topology`

O elemento `topology`, define informações sobre as interações entre os tipos definidos para o estilo. A descrição geral desse elemento é apresentada no Código 5.20.

O elemento `topology` é formado por um conjunto de elementos do tipo `link`. Cada elemento `link`, representa uma conexão permitida entre os tipos definidos no estilo. O atributo `name`, define um nome para a conexão e o atributo `from`, que é opcional, é utilizado na descrição de estilos derivados e apresenta o nome do estilo que herdamos essas descrição de conexão. Os atributos `start` e `end` de `link`, definem os nomes dos tipos que podem ser conectados. O atributo opcional `controlType` identifica o tipo de comunicação entre esses componentes, tendo os valores possíveis *push* ou *pull*, tendo *push* como default. Esses valores identificam quem inicia a comunicação entre os componentes.

5.4.3 Comportamento de Tipos de Componentes

Da mesma forma que fizemos para ArchML, para cada tipo de componente de uma especificação Xtyle, usamos um arquivo XMI para descrever o padrão de comunicação desse componente com relação às suas portas (comportamento observável). Esse arquivo XMI é gerado a partir de uma especificação de um Diagrama de Descrição de Protocolo (DDP), que foi apresentado na seção 5.3.

Dessa forma, supondo o trecho de especificação de um estilo representado no Código 5.21,podemos observar que o arquivo *SimpleFilter.xmi* é utilizado para descrever o comportamento observável do tipo de componente `SimpleFilter` com relação às suas portas. Esse componente pode possuir apenas uma porta de entrada e diversas portas de saída, sendo que nas portas de saída a comunicação é assíncrona. Também podemos observar uma diferença com relação a ArchML na descrição de comportamentos. Na descrição de componentes em ArchML, a priori sabemos todas as portas que esse componente possui, sendo que a seqüência de comunicação pode ser facilmente modelada através de um DDP.

Código 5.21 Trecho de Especificação de Estilo.

```
<type name="SimpleFilter">
   <ports>
      <in maxOccurs="1"/>
      <out maxOccurs="*"mode="async"/>
   </ports>
   <behavior href="SimpleFilter.xmi"/>
</type>
```

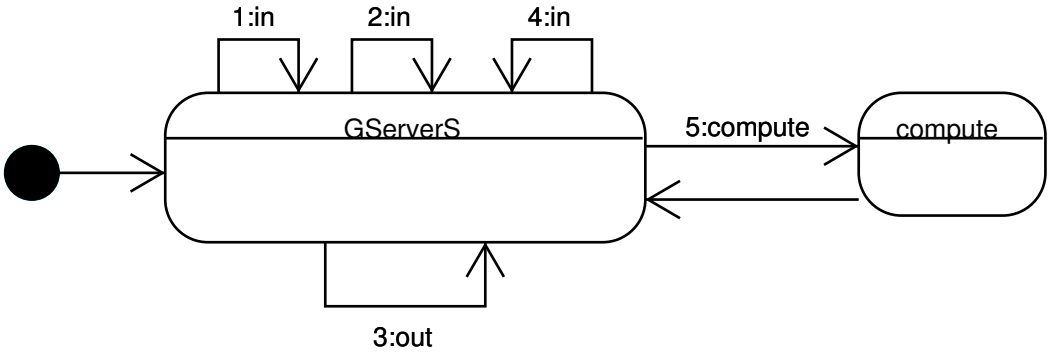


Figura 5.7: do Tipo GServerS.

Em Xtyle, a descrição de um tipo de componente é mais geral que a descrição de um componente ArchML, visto que muitas vezes nos tipos de componentes de Xtyle não sabemos de fato quantas portas temos que descrever (`maxOccurs="*"`). Contudo, quando descrevemos o comportamento de um tipo em um estilo, estamos interessados apenas nas seqüências de entradas e saídas, independente dos nomes das portas (usamos os elementos `in` e `out` para descrever apenas os tipos de portas). Dessa forma, os Diagramas de Descrição de Protocolos devem ser capazes de descrever esse tipo de protocolo genérico de Xtyle. Assim, propomos algumas alterações (generalizações) dos DDPs:

1. Os únicos nomes de eventos possíveis são `in`, `out` e `compute`, sendo que as mesmas regras utilizadas para descrição de protocolos de componentes ArchML devem também ser usadas para descrever esses eventos, assim se o número de portas de um tipo de componentes Xtyle for conhecido, deve-se usar a nomenclatura `1:in`, `2:in`, `3:out`, `4:in` para modelar a seqüência de comunicação.

   Ex.: Na Figura 5.7 apresentamos um DDP de um tipo chamado GserverS. Nele temos 3 portas de entrada e 1 porta de saída funcionando na seqüência indicada pelos números dentro dos colchetes realizando uma comunicação síncrona tanto para as portas de entrada como para as portas de saída.

2. Se em algum dos tipos (`in` ou `out`) o número de portas não for conhecido (`maxOccurs="*"`), devemos usar a seguinte nomenclatura para indicar um número

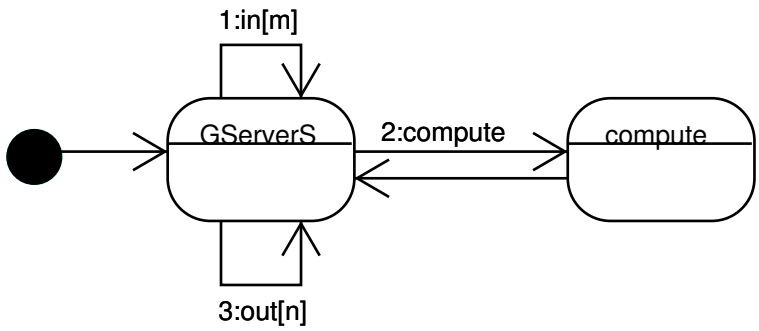


Figura 5.8: Comunicação Intercalada em DDP.

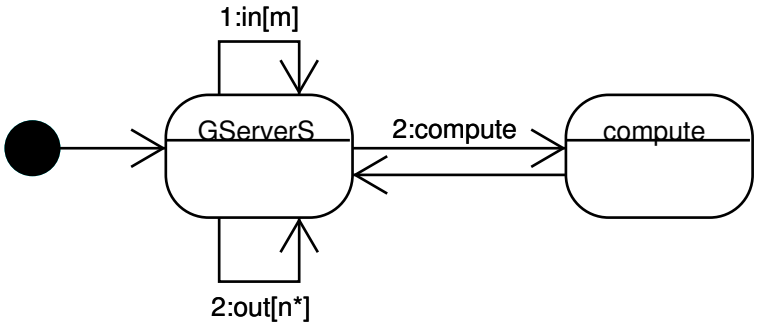


Figura 5.9: Comunicação Contínua em DDP.

limitado de vezes que uma comunicação pode ocorrer in[m], que significa que poderá ocorrer recebimentos de dados em m portas in e out[n], que significa que poderá ocorrer envios de dados em n portas out. Nesse caso, isso significa que a comunicação ocorrerá uma vez por um tipo de porta outra vez pelo outro tipo, de forma intercalada. Para indicar em qual porta a comunicação deve começar, usaremos um número antes do nome da porta. Assim, `1:in[m] 2:out[n]`, significa que haverá uma comunicação iniciando em `in` e depois uma em `out`, e segue nessa seqüência até realizar comunicações nas m portas `in` e nas n portas `out`.

Ex.: No exemplo da Figura 5.8, podemos observar que o componente GserverS realiza comunicação síncrona nas portas `in` e `out`, sendo que, pela seqüência e pela representação das transições temos uma comunicação iniciada em uma porta `in`, seguida de uma transição interna e depois de uma comunicação em uma porta `out`. Essa seqüência continua em todas as portas `in` e `out` de forma intercalada.

3. Devemos usar * depois da letra que determina o número de portas para indicar que uma comunicação deve acontecer em todas as portas que estão sendo definidas. Também usamos os números para identifica quem iniciará a comunicação.

   Ex.: Na Figura 5.9 apresentamos um exemplo em que haverá uma comunicação iniciada em `in`, seguida de uma transição interna e depois comunicações em todas as n portas `out`. Em seguida, haverá uma comunicação na porta `in` seguinte e o processo se repete para as m portas `in`.

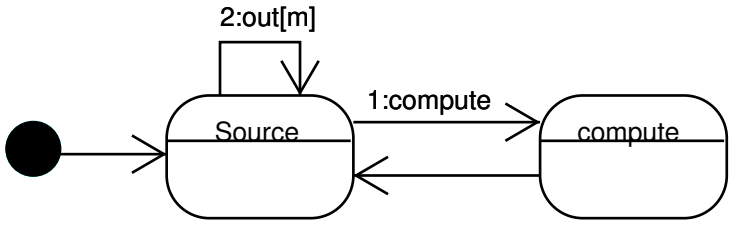


Figura 5.10: DDP de Source.

Essa nomenclatura para os eventos nos permite escrever DDPs de forma mais geral, com ela capturamos apenas as informações de comunicação dos tipos de componentes dos estilos e abstraímos os estados que não fazem parte desse processo além de trabalharmos com os aspectos de não conhecimento de quantidade de determinado tipo de porta.

As mesmas regras para a criação dos arquivos XMI relacionados aos protocolos, apresentadas na Seção 5.3, devem também ser seguidas em Xtyle.

5.4.4 Descrição de Estilos em Xtyle

Nessa seção exemplificaremos a descrição de um estilo suportado por DraX usando Xtyle. Em [113] pode ser encontrada a descrição completa de todos os demais estilos suportados por DraX. Essas especificações foram baseadas nas informações resumidas nas Tabelas 5.4 e 5.5. Para facilitar a apresentação das especificações Xtyle dos estilos de DraX utilizaremos dois itens distintos:

- **Especificação Xtyle**: Nesse item a especificação sintática em Xtyle de cada estilo é apresentada; e
- **Diagramas de Estado**: O comportamento de cada tipo de componente utilizados na especificação Xtyle é mostrado nesse item. Para facilitar o entendimento, apenas os diagramas de estados serão apresentados.

Rede de Fluxo de Dados

A especificação Xtyle de Rede de Fluxo de Dados é apresentada no Código 5.22. O DDP de `Source` está representado na Figura 5.10. O DDP de `Filter` está representado na Figura 5.11 e o DDP de `Sink` está representado na Figura 5.12.

5.5 Conclusão

Nesse capítulo apresentamos as linguagens de DraX. Inicialmente, na Seção 5.2 apresentamos uma linguagem de padrões que nos ajudou a definir como seria a estrutura sintática das linguagens de DraX. Para que os projetos arquiteturais pudessem ser desenvolvidos em DraX, implementamos ArchML, apresentada na Seção 5.3, como a ADL desse *framework*. Na verdade poderíamos lançar mão de outras ADLs disponíveis atualmente, como ACME ou Wright, contudo, buscamos com DraX conseguir uma forma de produzir especificações arquiteturais

Código 5.22 Especificação Xtyle de Rede de Fluxo de Dados.

```
<?xml version="1.0"?>
<xtyle name="DataFlowNetwork">
   <document>
      <version num="1.0"/>
      <author name="Cidcley T. de Souza"/>
      <lastUpdate date="2002-09-22"/>
      <comments>
         Estilo DataFlow Network
      </comments>
   </document>
   <types>
      <type name="Filter">
         <ports>
            <in mode="async"/>
            <out mode="async"/>
         </ports>
         <behavior href="Filter.xmi"/>
      </type>
      <type name="Source">
         <ports>
            <out mode="async"/>
         </ports>
         <behavior href="Source.xmi"/>
      </type>
      <type name="Sink">
         <ports>
            <in mode="async"/>
         </ports>
         <behavior href="Sink.xmi"/>
      </type>
   <topology>
      <link name="SourceFilter" start="Source" end="Filter" type="push"/>
      <link name="FilterSink" start="Filter" end="Sink" type="push"/>
      <link name="FilterFilter" start="Filter" end="Filter" type="push"/>
   </topology>
</xtyle>
```

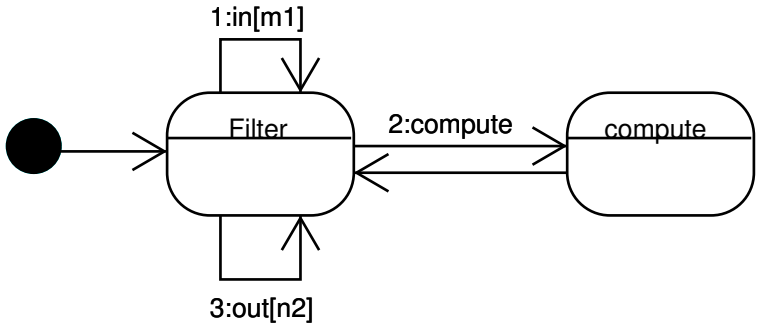


Figura 5.11: DDP de Filter.

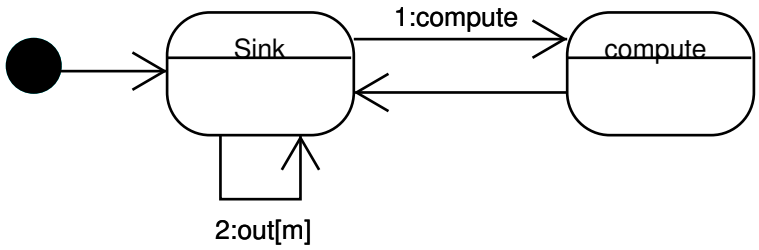


Figura 5.12: DDP de Sink.

de forma mais amigável e utilizando tecnologias e ferramentas que de fato façam parte da rotina de trabalho da indústria de software. E essas ADLs não possuem nem mesmo uma sintaxe que seja facilmente assimilada. Nesse contexto resolvemos criar uma nova linguagem que possuísse uma sintaxe de fácil manipulação e que tivessem ferramentas que pudessem manipula-la. Nesse sentido resolvemos utilizar XML para construir ArchML. Além disso, como apresentado em [36], há atualmente uma necessidade de se construir ADLs com características que facilitem a manipulação de arquiteturas distribuídas.

Uma outra abordagem que poderíamos seguir seria utilizar uma ADL baseada em XML disponível atualmente para realizar essa tarefa, como xADL [28] ou xACME [96]. Contudo essas linguagens apresentam uma estrutura muito confusa no sentido que essas estão mais para permitir a descrição de especificações arquiteturais como uma forma de documentação do que a criação de especificações passíveis de geração de implementação. Em ArchML optamos pela simplicidade, que é conseguida pela utilização de uma estrutura hierárquica simples e pela ausência de construtores XML (presentes nas outras ADLs baseadas em XML) que de alguma forma venha tornar a especificação menos clara. Assim, não utilizamos IDs nem IDREFs, sendo que deixamos as validações de sintaxe internamente à especificação ArchML ocorrem em uma fase posterior, onde essas conexões são devidamente avaliadas. Dessa forma conseguimos uma linguagem mais "enxuta" e de fácil aprendizagem e, principalmente, de fácil manipulação via ferramentas XML.

Uma outra peculiaridade de ArchML é a descrição em separado dos componentes. Temos uma gramática que permite a criação de componentes e um outra que utiliza referência externas a componentes para a criação de arquiteturas baseadas em instâncias desses componentes referenciados. Dessa forma, permitimos a construção de arquiteturas distribuídas onde a própria especificação pode ser distribuída. As demais ADLs que trabalham com XML realizam a descrição tanto dos componentes como das instâncias e da arquitetura em si em um mesmo arquivo, o que além de tornar a especificação ilegível, dificulta a criação de especificações com componentes especificados por equipes remotas.

A linguagem Xtyle, apresentada em detalhes na Seção 5.4, tem a intenção de fornecer um mecanismo de fácil manipulação para a descrição de estilos arquiteturais em DraX. A contribuição principal desse linguagem para o estado-da-arte em arquitetura de software está relacionada ao fato de que Xtyle serve exclusivamente para a descrição de estilos, o que a diferencia das ADLs que definem estilo a partir de restrições nas especificações de arquiteturas genéricas.

Em Xtyle, além da definição do vocabulário de um estilo, ou seja, dos nomes dos papéis que os componentes devem assumir em uma arquitetura de modo a seguir um determinado estilo, também definimos outras restrições importantes, como às relativas ao número de portas de

entrada e saída, o tipo de comunicação realizada nessa porta (síncrona ou assíncrona). Além disso, também podemos definir restrições topológicas, que dizem respeito às interações entre os tipos de um estilo, refletindo informações relativas ao fluxo de dados entre os participantes do estilo e o fluxo de controle, ou seja, como a comunicação é controlado entre esses participantes.

Uma outra vantagem de Xtyle é a possibilidade de definirmos o comportamento dos tipos através de DDPs (Diagramas de Descrição de Protocolos). Esses DDPs foram adaptados da linguagem ArchML (apresentada na seção 5.3) para melhor representar o protocolo de comunicação de um tipo na especificação de um estilo.

Nesse Capítulo também mostramos a especificação de alguns estilos já definidos em DraX e que podem ser utilizados diretamente sem a necessidade de especificação. Para melhor organizar os estilos em DraX, criamos uma taxonomia baseada na disponibilidade de ferramentas em DraX para a realização de validação do estilo. Assim definimos os estilos Básicos, Derivados e Definidos-Pelo-Usuário.

Os estilos Básicos são os estilos que já possuem todas as ferramentas de manipulação disponíveis em DraX (por ferramentas de manipulação podemos entender *scripts* de validação sintática e comportamental para o estilo). Os estilos Derivados são os que são formados se herdando definições de estilos de DraX e realizando adaptações nessas especificações. Um fato importante a ressaltar é que esses estilos podem ser derivados de um ou mais estilos. Os estilo Definidos-Pelo-Usuário são os estilos que podem ser criados livremente pelo desenvolvedor. É importante observar que DraX fornece ferramentas de geração de *scripts* validadores também para esses estilos, cabendo ao desenvolvedor apenas a tarefa de especifica-lo usando Xtyle.

Baseado em todas essas idéias, concluímos que Xtyle fornece um ambiente flexível para a manipulação de especificações de estilos arquiteturais. Além de fornecer um ambiente que facilita o ensino de estilos arquiteturais através de sua linguagem simples e expressiva.

Capítulo 6

Validação de Arquiteturas e Estilos

6.1 Introdução

Apenas especificar uma arquitetura utilizando as linguagens ArchML e Xtyle de DraX não garante que será gerada uma implementação que realmente funcione. Antes de gerar os templates de implementação as arquiteturas e estilos devem ser verificados tanto sintaticamente como comportamentalmente. Nesse capítulo trataremos do problema de validação das arquiteturas e estilos especificados em DraX.

Para iniciar nossa discussão é necessário definir o que estamos conceituando de validação de consistências sintáticas em DraX. Nos capítulos anteriores vimos a sintaxe para a especificação de componentes e arquiteturas através da linguagem ArchML e para a especificação de estilos arquiteturais através da linguagem Xtyle. Naquele momento, utilizamos uma gramática informal para apresentar os construtores sintáticos dessas linguagens. Contudo, apenas o fato de existir uma gramática não garante que venhamos a escrever especificações corretas, muito menos se essas forem informais. Desse modo, trataremos por validação sintática o processo de se avaliar a correção de uma especificação com relação à representação formal de sua gramática. Para tanto nesse capítulo apresentaremos as gramáticas formais para ArchML e Xtyle.

Um outro fator muito importante, principalmente por estarmos realizando a especificação de arquiteturas de forma distribuída, é garantir que as especificações sejam consistentes umas com as outras. Isto é, quando especificamos um sistema em ArchML devemos verificar se as portas dos componentes que conectamos possuem sentidos compatíveis, ou seja, se estamos conectando uma porta de entrada com uma porta de saída, entre outras informações. Além disso, diversos outros fatores devem ser considerados para que tenhamos uma especificação distribuída realmente consistente. Para realizar essa tarefa em DraX, utilizaremos uma linguagem de esquema baseada em XML bastante aceita atualmente, que são os Schematrons [61] para realizar a tarefa de verificação de consistências entre especificações.

A etapa de validação sintática em DraX foi apresentada na Seção 6.2. Essa etapa é formada tanto para validação e verificação de arquiteturas como de estilos arquiteturais. Além da verificação de conformidade sintática de uma arquitetura com relação a seu estilo. Ao longo dessa seção trataremos de todos esses aspectos.

Já a verificação de consistências comportamentais em arquiteturas serve para dois propósitos básicos: a verificação da compatibilidade entre componentes que formam uma arquitetura em um nível semântico e a verificação de propriedades na arquitetura como um todo.

Como apresentado nas Seções 5.3 e 5.4, a especificação de comportamentos de componentes de arquiteturas e de tipos em estilos ocorre pela incorporação de um arquivo XMI gerado por por uma ferramenta UML a partir da descrição usando DDPs do comportamento observável do componente com relação às suas portas.

A verificação de comportamento é realizada automaticamente em DraX, visto que desenvolvemos diversos *scripts* de geração de especificações π-cálculo que podemos utilizar para realizar as análises utilizando uma ferramenta como o MWB [120]. Assim, o desenvolvedor não precisa dominar as estruturas algébricas de π-cálculo basta apenas executar o *script* de geração de especificações e utilizar o resultado gerado no MWB para realizar as análises necessárias.

Além disso, podemos também verificar se os componentes de uma arquitetura têm comportamentos compatíveis com os seus respectivos tipos relativos a um estilo que a arquitetura esteja utilizando. Usamos o mesmo raciocínio anterior para realizar essas verificações.

Na Seção 6.3 os aspectos de validação comportamental de DraX são tratados em detalhes.

6.2 Validação Sintática em DraX

Como apresentado na introdução, a validação sintática de DraX diz respeito a avaliação da descrição de uma arquitetura ou componente com relação a uma gramática formal. A formalização de gramáticas para XML, seguindo as normas do W3C, é realizada através de DTDs (*Document Type Definition*) [16] e W3C XML Schemas [25], ou comumente referenciado como XSD. De fato, em um processo natural, as gramáticas baseadas em DTDs estão sendo substituídas por XSDs, visto que essa último possuem diversas vantagens em relação às DTDs [25]. Dentre muitas, podemos destacar a representação das gramáticas usando a própria sintaxe de XML e o que é mais importante para o nosso trabalho, é que XSDs possuem uma grande variedade de tipos de dados para representar elementos e atributos, o que nos permite a adoção dos próprios tipos definidos na especificação de XSD [26] para representar os tipos de dados em DraX. A seguir apresentaremos as gramáticas formais de DraX para a validação de componentes, arquiteturas e estilos arquiteturais. Para sermos mais precisos, explicaremos as porções dessa gramática ao longo da seção.

Na definição dessas gramáticas optamos por utilizar a técnica de inicialmente descrever atributos e elementos simples e em seguida apresentar elementos complexos. Com isso, ganhamos no reuso da especificação de atributos e na melhor apresentação visual e, conseqüentemente, um incremento na manutenibilidade dessas especificações.

6.2.1 Gramática para Validação de Componentes

O nome do esquema utilizado para a validação de componentes em DraX é `Component.xsd`. Como explicado anteriormente, iniciaremos a especificação pelos atributos e elementos simples usados na especificação de componentes por ArchML. Assim os seguintes atributos são definidos:

```
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="num" type="positiveDecimal"/>
<xsd:attribute name="propertyType" type="xsd:string"/>
<xsd:attribute name="propertyValue" type="xsd:string"/>
<xsd:attribute name="date" type="xsd:date"/>
<xsd:attribute name="returnType" type="xsd:QName"/>
<xsd:attribute name="role" type="xsd:string"/>
<xsd:attribute name="href" type="xsd:uri"/>
<xsd:attribute name="type" type="xsd:QName"/>
<xsd:attribute name="direction" type="portDirection"/>
```

O atributo `name`, que é do tipo **xsd:string**, onde `xsd` é o espaço-de-nomes (namespace) que identifica os elementos de XSD, é largamente usado nos elementos de ArchML para identifica-los. Assim como os atributos `href`, que é do tipo **xsd:uri**. Essas definições serão reusadas algumas vezes. Os atributos `num` e `date` são utilizados na descrição do elemento document. O atributo `date` é **xsd:date**, Já o atributo `num` é usado no elemento `version` e define o número da versão. No caso, decidimos construir um novo tipo para esse atributo visto que os tipos numéricos definidos em XSD não definem elementos decimais apenas positivos. Desse modo definimos o tipo `positiveDecimal`, cuja descrição é a seguinte:

```
<xsd:simpleType name="positiveDecimal">
   <xsd:restriction base="xsd:decimal">
      <xsd:minInclusive value="0.0"/>
      <xsd:maxInclusive value="99.9999"/>
   </xsd:restriction>
</xsd:simpleType>
```

Os atributos `propertyType` e `propertyValue` são usados na descrição do elemento `propertySet`. Eles são do tipo **xsd:string** mas possuem diversas restrições que não são facilmente capturadas em XSD. Essas restrições dizem respeitos à relação entre os valores assumidos por `propertyType` que restringem os valores de `propertyValue`. Essas restrições serão capturadas por Schematrons e serão apresentadas mais adiante nesse capítulo.

O atributo `role` é usado para identificar os papéis assumidos pelos componentes, sendo que os valores possíveis são verificados na especificação dos estilos. Assim, esses valores dizem respeito à verificação sintática dos componentes com relação à arquitetura. Apresentaremos essa faceta mais adiante.

O atributo `type`, que também é utilizado diversas vezes na descrição de um componente, representa os tipos que esses podem assumir. Nesse caso, optamos por usar os tipos definidos em XSD em ArchML. Assim, definimos o tipo **xsd:QName**, que exige a representação de nomes qualificados para esse elemento. Da mesma forma funciona o atributo `returnType`, que define um tipo de retorno para uma porta.

O atributo `email`, que é usado para se definir o endereço de correio eletrônico do autor da especificação, usa o tipo `emailType` que é um novo tipo definido na especificação da seguinte forma:

```
<xsd:simpleType name="emailType">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value=".*@.*"/>
   </xsd:restriction>
</xsd:simpleType>
```

O tipo `emailType`, permite se definir um padrão para a validação de email através de uma expressão regular.

Por fim, a atributo `direction`, usado no elemento `port`, identifica a direção de comunicação de uma porta. Como os valores possíveis para esse elemento são `in` e `out`, resolvemos optar por criar um tipo simples que represente essa característica. Dessa forma, o tipo de `direction` foi definido com `portDirection`, cuja especificação é a seguinte:

```
<xsd:simpleType name="portDirection">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="in"/>
      <xsd:enumeration value="out"/>
   </xsd:restriction>
</xsd:simpleType>
```

Usando a descrição desses atributos podemos agora apresentar os elementos complexos da gramática de ArchML para a especificação de componentes. Os primeiros tipos complexos que apresentaremos são os que formam o elemento `document`, ou seja: `version`, `author`, `lastUpdade` e `comments`, cujas especificações são as seguintes:

```
<xsd:element name="version">
<xsd:complexType>
   <xsd:attribute ref="num" use="required"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="author">
<xsd:complexType>
   <xsd:attribute ref="name" use="required"/>
   <xsd:attribute ref="email" use="emailType"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="lastUpdate">
<xsd:complexType>
   <xsd:attribute ref="date" use="required"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="comments" type="xsd:string"/>
```

As especificações são simples, sendo que a única especificação diferente é a do elemento `comments`, que é um elemento que representa algum comentário dentro do elemento `document` e que não possui atributos e cujo conteúdo é apenas do tipo **xsd:string**.

De posse da especificação dos elementos filhos de `document`, agora podemos especifica-lo. Segue sua representação em XSD:

```
<xsd:element name="document">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="version"/>
      <xsd:element ref="author"/>
      <xsd:element ref="lastUpdate"/>
      <xsd:element ref="comments"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Pode ser observado na especificação que os elementos filhos de `document` são todos obrigatórios e devem aparecer exatamente uma vez.

Agora, para especificar o elemento `propertySet`, devemos inicialmente definir o elemento `property`. A especificação de `property` é a seguinte:

```
<xsd:element name="property">
<xsd:complexType>
   <xsd:attribute ref="context" use="required"/>
   <xsd:attribute ref="propertyType" use="required"/>
   <xsd:attribute ref="propertyValue" use="required"/>
</xsd:complexType>
</xsd:element>
```

O elemento `property` é formado por um conjunto de atributos obrigatórios que identificam o contexto em que a propriedade de aplica (`context`), o tipo de propriedade que está sendo definida (`propertyType`) e o valor para a propriedade (`propertyValue`).

Nesse momento já é possível se definir o elemento `propertySet`. Esse elemento serve como um contexto para armazenar um conjunto de propriedades de um componente, ou seja, ele é formado por um conjunto de elementos do tipo `property`. Segue a especificação de `propertySet`:

```
<xsd:element name="propertySet">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="property" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

O próximo elemento a ser especificado é o `port`. Esse elemento define as portas de um componente, sendo que cada porta possui um conjunto de parâmetros. Dessa forma, iniciaremos apresentando a especificação do elemento `param`, que define esses parâmetros de `port`. Segue a especificação de `param`:

```
<xsd:element name="param">
<xsd:complexType>
   <xsd:attribute ref="name" use="required"/>
   <xsd:attribute ref="type" use="required"/>
</xsd:complexType>
</xsd:element>
```

Cada elemento `param` é formado por um conjunto de atributos que definem o nome do parâmetro (`name`) e o tipo do parâmetro (`type`). Tendo a especificação de `param`, podemos agora especificar `port`. Segue a especificação:

```
<xsd:element name="port">
<xsd:complexType>
   <xsd:sequence>
   <xsd:element ref="param"/>
   </xsd:sequence>
   <xsd:attribute ref="name" use="required"/>
   <xsd:attribute ref="direction" use="required"/>
   <xsd:attribute ref="returnType" use="optional"/>
</xsd:complexType>
</xsd:element>
```

Além de um conjunto de elementos filho `param`, o elemento `port` possui os atributos `name`, `direction` e `returnType`, que identificam, respectivamente, o nome da porta, a direção de comunicação e um tipo opcional de retorno. Um conjunto de portas define o elemento `interface`. Segue a especificação de `interface`:

```
<xsd:element name="interface">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="port" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute ref="role" use="optional" default="Object"/>
</xsd:complexType>
</xsd:element>
```

Cada elemento `interface`, além de um conjunto de elementos filho do tipo `port`, possui um atributo `role`, que define o papel do componente com relação as portas definidas na interface. Esse valor está ligado a definição de um estilo. Como o uso de estilo não é obrigatório em ArhcML, e caso esse não seja especificado será assumido o estilo Cliente-Servidor, o valor default para `role` é *Object*, que é o único tipo de papel possível de ser assumido nesse tipo de estilo.

A especificação do elemento `interfaces`, que é composto por um conjunto de elementos `interface`, é apresentada a seguir:

```
<xsd:element name="interfaces">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="interface" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Para que possamos descrever o elemento `component` falta apenas definir seu elemento filho `behavior`. Esse elemento, cuja especificação se segue, permite a definição de uma especificação XMI do DDP que representa o comportamento do componente.

```
<xsd:element name="behavior">
<xsd:complexType>
   <xsd:attribute ref="href" use="required"/>
</xsd:complexType>
</xsd:element>
```

Por fim, segue a especificação do elemento `component`, cujos únicos elementos filhos obrigatórios são `interfaces` e `behavior`.

```
<xsd:element name="component">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="document" minOccurs="0"/>
      <xsd:element ref="propertySet" minOccurs="0"/>
      <xsd:element ref="interfaces"/>
      <xsd:element ref="behavior"/>
   </xsd:sequence>
      <xsd:attribute ref="name" use="required"/>
</xsd:complexType>
</xsd:element>
```

6.2.2 Gramática para Validação de Arquiteturas

Da mesma forma que apresentamos a gramática para a descrição de componentes faremos também para a gramática de especificação de arquiteturas de ArhcML. O nome do arquivo na qual essa gramática foi definida é System.xsd. Também seguiremos a mesma metodologia para a descrição da gramática para arquiteturas, ou seja, descreveremos primeiro os atributos, tipos e elementos simples e em seguida compremos os elementos complexos a partir das definições já realizadas. Assim, temos os seguintes atributos para a descrição de arquiteturas em ArchML:

```
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="num" type="positiveDecimal"/>
<xsd:attribute name="href" type="xsd:anyURI"/>
<xsd:attribute name="date" type="xsd:date"/>
<xsd:attribute name="email" type="emailType"/>
<xsd:attribute name="typeRef" type="xsd:string"/>
<xsd:attribute name="instRef" type="xsd:string"/>
<xsd:attribute name="portRef" type="xsd:string"/>
```

Basicamente temos muitos do atributos utilizados na descrição de componentes, como por exemplo, `name`, `num` e `date`, que pertencem ao elemento `document`, que também está presente em `component`. Além disso, temos o atributo, `href`, que também é uma referência **URI** a um arquivo externo. Os atributos, `typeRef`, `instRef` e `portRef`, respectivamente, funcionam como referências a algum tipos de componente, a uma instância de componente e a uma porta de um componente.

Com relação aos elementos de `system`, inicialmente temos o elemento `document`, e todos os seus elementos filhos cuja descrição e explicação detalhada foi realizada na seção anterior:

O elemento `style`, que é o seguinte na estrutura de `system`, permite a definição de um estilo arquitetural para a arquitetura. Essa definição é realizada através da referência a um arquivo externo especificado em Xtyle. Assim, temos a seguinte especificação XSD:

```
<xsd:element name="style">
<xsd:complexType>
   <xsd:attribute ref="href" use="required"/>
</xsd:complexType>
</xsd:element>
```

O próximo elemento é o `types`. Esse elemento é formado por um conjunto de elemento `type`, sendo que cada `type` possui um conjunto de atributos `name` e `href` que representam, respectivamente, o nome de um componente e a referência ao arquivo onde esse está especificado em ArchML. Segue a especificação:

```
<xsd:element name="type">
<xsd:complexType>
   <xsd:attribute ref="name" use="required"/>
   <xsd:attribute ref="href" use="required"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="types">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="type" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Da mesma forma é realizada a especificação de instâncias de componentes que formam a arquitetura. Essa tarefa, realizada pelo elemento `instances`, é conseguida pela definição de um conjunto de elementos `instance`, cada um dos quais representando uma instâncias de um determinado tipo de componente previamente declarado nos elementos `types`. Assim, cada elemento `instance` possui dois atributos, um `name`, que representa o nome da instância e `typeRef`, que possui o nome do tipo do qual aquela instância se refere. A especificação completa é a seguinte:

```
<xsd:element name="instance">
<xsd:complexType>
   <xsd:attribute ref="name"/>
   <xsd:attribute ref="typeRef"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="instances">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="instance" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

O último elemento de `system` é `links`. Esse elemento descreve os links entre as portas das instâncias declaradas através de elementos `link`. Cada `link`, possui um atributo `name`, que fornece uma identificação para o `link`, e dois elementos filhos `point`, que representam os pontos de conexão do link. Cada elemento `point`, por sua vez, possui dois atributos, o `instRef`, que representa a referencia de uma instância de componentes descrita previamente e `portRef`, que descreve o nome da porta dessa instância que vai servir de ponto de conexão daquele link. A especificação completa é a seguinte:

```
<xsd:element name="point">
<xsd:complexType>
   <xsd:attribute ref="instRef"/>
   <xsd:attribute ref="portRef"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="link">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="point" minOccurs="2" maxOccurs="2"/>
   </xsd:sequence>
   <xsd:attribute ref="name"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="links">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="link" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Um detalhe importante é que cada elemento `link` tem que possuir exatamente dois elementos filhos `point`, indicando as duas extremidades de uma conexão.

6.2.3 Gramática para Validação de Estilos

Seguindo o mesmo método apresentado para ArchML, nessa seção definimos a gramática formal para Xtyle. O arquivo Xtyle.xsd é usado para definir essa gramática e a definição inicia pelos atributos, como se segue:

```
<xsd:attribute name="name"type="xsd:string"/>
<xsd:attribute name="num"type="positiveDecimal"/>
<xsd:attribute name="date"type="xsd:date"/>
<xsd:attribute name="minOccurs"type="Occurs"use="optional"default="1"/>
<xsd:attribute name="maxOccurs"type="Occurs"use="optional" default="1"/>
<xsd:attribute name="maxConns"type="Occurs"use="optional"default="*"/>
<xsd:attribute name="minConns"type="Occurs"use="optional"default="*"/>
<xsd:attribute name="mode" use="optional"default="sync">
   <xsd:simpleType>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="sync"/>
         <xsd:enumeration value="async"/>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="href"type="xsd:anyURI"/>
<xsd:attribute name="start"type="xsd:string"/>
<xsd:attribute name="end"type="xsd:string"/>
<xsd:attribute name="email"type="emailType"/>
<xsd:attribute name="controlType"use="optional"default="push">
   <xsd:simpleType>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="push"/>
         <xsd:enumeration value="pull"/>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:attribute>

<xsd:attribute name="from"type="xsd:string"/>
```

Vários dos atributos apresentados aqui são os mesmos usados em ArchML e já foram devidamente explicados anteriormente, como `name`, `num`, `date`, `href`. Entretanto, Xtyle introduz vários outros atributos, como por exemplo `minOccurs` e `maxOccurs`, esses atributos são utilizados para se definir, respectivamente, o número mínimo e máximo de um determinado tipo de porta em um tipo de componente de um estilo. Esses atributos são do tipo `Occurs`, que tem a seguinte definição:

```
<xsd:simpleType name="Occurs">
   <xsd:union memberTypes="xsd:nonNegativeInteger»
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:enumeration value="*"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:union>
</xsd:simpleType>
```

Como os valores possíveis para `minOccurs` e `maxOccurs` são inteiros positivos ou então o caractere `"*"` para indicar um número indeterminado de portas, o tipo `Occurs` é definido com uma união para representar esses valores.

Os atributos `maxConns` e `minConns`, são utilizados para definir, respectivamente, o número máximo e mínimo de conexões permitidas em uma determinada porta, e te como tipo, também `Occurs`.

O atributo `controlType`, utilizado para definir o modo do fluxo de controle na comunicação entre dois componentes numa arquitetura, pode assumir os valores *push*, *pull* ou *any*, significando, respectivamente, que quem estiver interessado num dado deve realizar a invocação, que quem estiver com a posse do dado é quem deve passar para quem quer utilizá-lo, ou os dois modos podem ocorrer.

Um outro atributo notável é o `mode`, que define o modo de comunicação, podendo assumir os valores *sync* e *async*, para especificar, respectivamente, uma comunicação síncrona ou assíncrona.

Além de document, já apresentado anteriormente, os elementos complexos da gramática de Xtyle são os seguintes:

```
<xsd:element name="style">
   <xsd:complexType>
      <xsd:attribute ref="name" use="required"/>
      <xsd:attribute ref="href" use="required"/>
   </xsd:complexType>
</xsd:element>
<xsd:element name="uses">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="style" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

O elemento `style` deve ser usado para definir estilos derivados, sendo que no elemento style devem ser utilizados os atributos nome e uma referência para identificar o estilo usado. Ainda para a definição de estilos derivados, como podemos ter uma derivação composta, ou seja, um estilo derivados de dois ou mais estilos, o elemento use, serve como elemento pai para a definição de diversos elementos `style`, cada um especificando um estilo diferente.

O próximo elemento a ser definido é o `types`. Esse elemento permite a definição de tipos de componentes que podem ser usados na descrição de arquiteturas que seguem o estilo, ou seja, esse elemento, além de outras coisas, define a gramática de tipos permitida para o estilo. O elemento `types` é formado por elementos do tipo `type`, sendo que cada elemento `type`, representa um tipo diferente Os seguintes elementos são usados para definir type:

```
<xsd:element name="alias">
   <xsd:complexType>
      <xsd:attribute ref="name" use="required"/>
      <xsd:attribute ref="from" use="required"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="in">
   <xsd:complexType>
      <xsd:attribute ref="minOccurs"/>
      <xsd:attribute ref="maxOccurs"/>
      <xsd:attribute ref="maxConns"/>
      <xsd:attribute ref="minConns"/>
      <xsd:attribute ref="mode"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="out">
   <xsd:complexType>
      <xsd:attribute ref="minOccurs"/>
      <xsd:attribute ref="maxOccurs"/>
      <xsd:attribute ref="maxConns"/>
      <xsd:attribute ref="minConns"/>
      <xsd:attribute ref="mode"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="ports">
<xsd:complexType>
   <xsd:sequence>
      <xsd:element ref="in" minOccurs="0"/>
      <xsd:element ref="out" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="behavior">
   <xsd:complexType>
      <xsd:attribute ref="href"/>
   </xsd:complexType>
</xsd:element>
```

Os elementos `type` e `types` ficam estruturados dessa forma:

```
<xsd:element name="type">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="alias" minOccurs="0" maxOccurs="unbounded"/>
         <xsd:element ref="ports"/>
         <xsd:element ref="behavior"/>
      </xsd:sequence>
      <xsd:attribute ref="name"/>
      <xsd:attribute ref="from" use="optional"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="types">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="type" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Em Xtyle, podemos definir estilos derivados compostos, neste caso dois estilos ou mais são utilizados para se definir um novo estilo, sendo que os tipos de componentes das definições herdadas podem agir como se fossem mais de um tipo, ou seja, um elemento poderá agir como o tipo `Filter` do estilo `Pipeline`, ou como `Module` do estilo `Dados Compartilhados`. Assim, o primeiro elemento a ser definido é `alias`, que permite relacionar os tipos de componentes que poderão agir como outros tipos de componentes.

Em seguida as possíveis portas do tipo de componente são definidas. Essas portas tanto podem ser de entrada (*in*) como de saída (*out*). Cada tipo de porta possui um conjunto de atributos que permite realizar restrições na definição de um tipo de componente de um determinado estilo. Esses atributos são `maxOccurs`, `minOccurs`, `maxConns`, `minConns` e `mode`.

Só então o elemento `ports` é definido. Esse elemento permite a descrição das portas de um tipo de componente, sendo esse pode apresentar portas *in* ou portas *out*.

Seguindo a mesma estrutura de ArchML o comportamento de um componente deve ser descrito através de um arquivo XMI gerado a partir de uma máquina de estados UML.

O elemento `type` permite é o elemento pai da descrição de um tipo em Xtyle. Assim, usamos os atributos `name`, para definir o nome do tipo e o atributo opcional `from`, que define de onde esse tipo foi herdado em caso de uma derivação. Como elementos filho, temos `alias`, que define as relações do tipo com outros tipos em estilos derivados; `ports`, que define as portas para o tipo e `behavior` que descreve o comportamento do tipo.

Por fim podemos especificar o elemento `types` é o elemento pai da definição dos tipos de elementos de Xtyle. Assim esse elemento possui um conjunto de elementos filhos do tipo `type`.

A descrição da topologia e de restrições de fluxo e controle em estilos arquiteturas descritas em Xtyle é realizada pelo elemento `topology`. Os seguintes elementos são usados para descreve `topology`:

```
<xsd:element name="link">
   <xsd:complexType>
      <xsd:attribute ref="name"/>
      <xsd:attribute ref="from" use="optional"/>
      <xsd:attribute ref="start"/>
      <xsd:attribute ref="end"/>
      <xsd:attribute ref="controlType"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name="topology">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="link" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

O elemento `link` permite a descrição de um tipo de conexão no estilo, cujo nome e definido pelo atributo `name`. Caso esse seja um estilo derivado o atributo `from` pode ser usado para identificar a origem da descrição da conexão, sendo então necessário apenas se descrever os atributos que são diferentes da descrição original, usando a idéia de sobrecarga.

Os atributos `start` e `end` indicam o fluxo de dados na conexão, sendo que os valores desses atributos são nomes de tipos descritos no estilo. O atributo `controlType`, por sua vez, descreve o fluxo de controle entre os componentes, ou seja, explicita quem deve começar a comunicação, tendo com valores possíveis, como explicado anteriormento, *push*, *pull* e *any*.

O elemento `topology` é o pai da descrição da topologia do estilo. Assim, ele é formado por diversos elementos do tipo `link`.

O elemento `xtyle` é o elemento raiz da descrição de um estilo. Sua gramática é a seguinte:

```
<xsd:element name="xtyle"/>
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="document" minOccurs="0"/>
         <xsd:element ref="uses" minOccurs="0"/>
         <xsd:element ref="types"/>
         <xsd:element ref="topology"/>
      </xsd:sequence>
      <xsd:attribute ref="name"/>
   </xsd:complexType>
</xsd:element>
```

Podemos perceber que um estilo em Xtyle é formado por um elemento `document` opcional, um elemento `uses`, também opcional, um elemento `types` e um elemento `topology`.

6.2.4 Restrições Sintáticas em Componentes e Arquiteturas

Com a utilização de gramáticas XSD, podemos controlar a forma com que os componentes são descritos. Entretanto, essas gramáticas não realizam as validações que considerem a relação entre elementos ou atributos em um mesmo documento. As únicas informações com relação

a esse tipo de validação são as identificadas pelos atributos do tipo ID e IDREF. Entretanto, como resolvemos não adotar esse tipo de atributos em ArchML. Contudo, na definição de componentes não necessitamos deles, mas necessitamos de outro tipo de validação que garanta a correta especificação dos elementos do tipo `property`.

Os elementos `property`, como apresentados anteriormente, servem para definir propriedades dos componentes e possuem os atributos, `context`, `propertyType` e `propertyValue`. Os valores assumidos por esses atributos estão intimamente relacionados (ver Tabela 5.3 na seção 5.3 e, para garantir a correção da especificação de um componente, devem ser validados.

Para resolver esse problema utilizamos regras em Schematron [61], que é uma linguagem de esquema baseada em XML que usa XPath para realizar testes estruturais em documentos XML. Através de Schematron definimos uma regra para complementar a validação realizada por XSD em componentes descritos em ArchM. Dessa forma, a seguinte regra garante a validade dos elementos `property` na definição de um componente:

```
<rule context="propertySet/property">
   <assert test="( (@context='POA' and @propertyType='ThreadPolicy'
      and @propertyValue='SingleThread') or
       (@context='POA'
      and @propertyType='ThreadPolicy'
      and @propertyValue='MultiThread') ) or
       (@context='NamingService'
      and @propertyType='NamingContext'
      and (@propertyValue='RootContext' or
       string-length(@propertyValue)!=0))
    ">Propriedade Invalida.
   </assert>
</rule>
```

Essa regra, usada para cada elemento `property` de `propertySet`, valida componentes que possuam valores determinados para os atributos `context`, `propertyType` e `propertyValue`, como apresentado na especificação Schematron. Podemos observar, por exemplo, que é possível se definir uma propriedade válida do tipo:

```
<property context="POA" propertyType="ThreadPolicy"
          propertyValue="SingleThread"/>
```

Também, por ser definida usando o mesmo estilo de especificação de XML, essa regra pode ser facilmente ampliada caso seja necessário a introdução de novos tipos de propriedades para componentes. Além disso, é muito fácil se realizar qualquer manutenção nessa especificação.

Um outro aspecto que deve ser validado nos componente está relacionado aos nomes dos `roles` das interfaces e aos nomes dos parâmetros das portas. Cada componente pode possuir mais de uma interface, como apresentado na gramática de ArchML, sendo que cada interface possui um atributo role que identifica o papel assumido pelo componente. Assim, devemos garantir que só exista uma interface com o mesmo `role` na especificação de um componente. Usando a seguinte regra Schematron podemos garantir essa propriedade:

```
<pattern name="Validacao de Roles">
   <rule context="interfaces/interface">
      <assert test="count(//interface[@role=current()/@role])=1
      ">Interfaces com Roles Iguais.
      </assert>
   </rule>
</pattern>
```

Com essa regra, fixamos um elemento `interface` e contamos quantos elementos do tipo `interface` existem dentro do elemento `interfaces` cujo atributo `role` é igual ao atributo `role` do elemento fixado (current). O resultado deve ser 1, que é exatamente o próprio elemento fixado.

No que diz respeito a validação dos nomes dos parâmetros, temos o seguinte esquema:

```
<pattern name="Validacao de Parametros">
   <rule context="port/param">
      <assert test="not(following-sibling::param/@name=@name)
      "> Nome de Parametro Ja Usado.
      </assert>
   </rule>
</pattern>
```

Como pode ser observado que mesmo a operação de validação sendo semelhante, tivemos que usar uma outra estrutura de validação. Isso se deu pelo fato que o elemento `param`, cujo atributo `name` deve ser único, pode ocorrer diversas vezes em portas diferentes. O que estamos querendo é que dentro de uma mesma definição de porta, esses elementos sejam diferentes. Assim, a regra verifica, para cada elemento `param`, se os irmãos desse elemento (nesse momento restringimos a busca a uma única porta por vez) não possui um atributo com o mesmo valor do elemento fixo.

Também devemos realizar a validação dos nomes das portas dos componentes. Como um componente pode ter mais de uma interface, os nomes das portas, mesmo em interfaces diferentes, devem ser diferentes. Assim temos o seguinte esquema de validação para essa propriedade:

```
<pattern name="Validacao de Nomes de Portas">
   <rule context="port">
      <assert test="not(following::port/@name=@name)
      "> Nome de Parametro Ja Usado.
      </assert>
   </rule>
</pattern>
```

Já a validação da descrição de arquiteturas em ArchML é um pouco mais complexa que a validação de componentes. Cada arquitetura possui um conjunto de tipos definidos a partir de referências a especificação de componentes externos e, possivelmente, distribuídos. Esses tipos são referenciados dentro da própria especificação no momento da definição de instâncias de componentes. Da mesma forma, cada instância também possui um nome que é referenciado na arquitetura no momento da descrição das conexões.

Como já observado, em ArchML optamos por não usar atributos do tipo ID e IDREF, devemos garantir as propriedades de integridade referencial e de unicidade nas definições de nomes de tipos e de nomes de instâncias. Para realizarmos essa tarefa também utilizamos regras Schematron. Assim, o esquema que desenvolvemos realiza a validação das seguintes características das arquiteturas:

1. Unicidade de nomes de tipos e de nomes de instâncias: definimos duas regras em Schematron, uma para garantir a existência de nomes únicos para os tipos de componentes e outra para realizar a mesma tarefa nos nomes das instâncias.

2. Referência de Tipos: criamos uma regra que testa se, na especificação de instâncias, os tipos de componentes referenciados realmente existem na especificação.

3. Referência de Instâncias: da mesma forma, observamos por meio de uma regra se as instâncias referenciadas na especificação dos links foram de fato especificadas anteriormente.

Dessa forma, as primeiras regras aplicadas para a validação de uma arquitetura através de Schematron são as que garantem a unicidade dos nomes nas descrições dos tipos e das instâncias. Segue as especificações:

```
<pattern name="Tipos Diferentes">
   <rule context="types/type">
      <assert test="count(//type[@name=current()/@name])=1">
      Tipo "<value-of select="current()/@name"/>"ja Declarado.
      </assert>
   </rule>
</pattern>
<pattern name="Instancias Diferentes">
   <rule context="instances/instance">
      <assert test="count(//instance[@name=current()/@name])=1">
      Instancia "<value-of select="current()/@name"/>"Ja Declarada.
      </assert>
   </rule>
</pattern>
```

Nessas regras, podemos observar que testamos cada tipo ou instâncias e, através de uma expressão XPath, verificamos se os atributos `name` são declarados apenas uma vez. Isso é realizado pela fixação de um elemento (current()) e na contagem de todos os outros elementos que possuem um atributo cujo valor seja igual ao do elemento fixado. Obviamente isso deve retornar 1, que é o próprio elemento. Entretanto, com realizados o teste dentro de um elemento assert o teste é falso se o valor da expressão no atributo `test` não ocorrer.

Num segundo momento, validamos as referências aos nomes de componentes realizadas na especificação de instâncias através do atributo `typeRef`. A seguinte regra deve ser aplicada para realizar essa validação:

```
<pattern name="Referencia de Tipos">
   <rule context="instance">
      <assert test="@typeRef=//type/@name">
      Tipo "<value-of select="@typeRef"/>"Nao Declarado.</assert>
   </rule>
</pattern>
```

Essa regra testa, para cada elemento `instance`, se o existe um elemento `type` cujo atributo `name` seja igual ao atributo `typeRef` atual. O mesmo teste fazemos para a verificação dos nomes das instâncias na declaração de `links`, a seguinte regra deve ser aplicada:

```
<pattern name="Referencia de Instancias">
   <rule context="point">
      <assert test="@instRef=//instance/@name">
      Instancia "<value-of select="@instRef"/>"Nao Declarada.</assert>
   </rule>
</pattern>
```

Do mesmo modo, para cada elemento `point`, verificamos se existe um atributo `name` em algum elemento `instance` cujo valor seja igual ao atributo `instRef` de `point`.

6.2.5 Restrições Sintáticas em Estilos

Da mesma forma que apenas a gramática não consegue validar uma arquitetura ou um componente em ArchML, também em Xtyle temos que lançar mão de Schematron para validar as relações entre elementos na descrição de um estilo. Contudo, diferentes das arquiteturas, que possuem um grande número de informações para ser validada com relação às relações entre os elementos internos à descrição, em Xtyle a única relação interna que temos que validar são as que garantem que os nomes usados nos atributos `start` e `end` na definição dos elementos `link`, foram realmente declarados com tipos na arquitetura.

Assim, o seguinte esquema consegue validar essa propriedade:

```
<pattern name="Tipos nos Links">
   <rule context="link">
      <assert test="//type[@name=current()/@start] and
      //type[@name=current()/@end]"> Tipo Nao Declarado.</assert>
   </rule>
</pattern>
```

Nessa regra, observamos se os nomes dos atributos `start` e `end` do elemento `link`, possuem o mesmo valor de algum atributo `name` de algum elemento `type`.

6.2.6 Validação de Consistência Sintática em Arquiteturas

As arquiteturas descritas em ArchML possuem referências a especificação de componentes externos e, possivelmente, distribuídos. Assim, além de garantir a validade da especificação da arquitetura com relação às regras gramaticais de XSD, devemos garantir que os elementos externamente referenciados são compatíveis. Assim, tratamos por verificação de consistência sintática a tarefa de se verificar as relações entre componentes que fazem parte de uma arquitetura tendo como base a especificação ArchML dessa arquitetura.

As atividades realizadas na verificação sintática de arquiteturas dizem respeito tanto a arquitetura em si, ou seja, a verificação dos tipos de dados das portas, dos sentidos de comunicação, etc, como também com relação aos estilos que uma arquitetura segue. Como na conformidade de estilos temos que considerar tanto aspectos de nomenclatura como também de

restrições topológicas, e essas verificações são diferentes para cada tipo de estilo arquitetural, dessa forma, separamos uma seção a parte nesse capítulo unicamente para tratar os aspectos de verificação de conformidade de estilos arquiteturais. Nessa seção trataremos puramente da verificação de consistências entre os componentes das arquiteturas.

Para a realização das verificações de consistências também utilizamos regras Schematon, essas regras estão relacionadas às seguintes informações de arquiteturas:

1. **Referência de Portas**: Nesse tipo de verificação para cada porta definida no elemento `point`, são observados se essa porta foi realmente definida na descrição do tipo de componente correspondente à instância.

2. **Sentido das Portas**: Para cada par de elementos `point` definido, são observados se são compatíveis os tipos das portas com relação ao fluxo de informações entre os componentes através daquela porta, isto é, em um `point` que possui uma porta de entrada, deve estar conectado com um outro `point` que define uma porta saída.

3. **Parâmetros das Portas**: Além de garantir a compatibilidade com relação ao fluxo de informações entre as portas, devemos garantir que os parâmetros que elas trocam também sejam compatíveis. Nesse sentido esse tipo de verificação observa se para cada porta conectada possuem o mesmo número de parâmetros e se os tipos de dados trocados são iguais.

O primeiro tipo de verificação, que diz respeito à referência de nomes de portas, é realizado por uma regra em Schematron que verifica se na especificação do componente, cuja instância é referenciada pelo atributo `instRef` no elemento `point`, existe uma porta com o mesmo nome. A especificação completa da regra é a seguinte:

```
<pattern name="Referencia de Portas">
   <rule context="point">
      <assert test="document(//type[@name=//instance[@name=current()
            /@instRef]/@typeRef]/@href)//port[@name=current()/@portRef]">
      Referencia de Porta Invalida.</assert>
   </rule>
</pattern>
```

Na especificação da regra, foi utilizado o comando XPath `document` para inspecionar um arquivo externo à arquitetura. Nessa regra, aplicada em cada elemento `point`, é observado, se o valor do atributo `portRef` do elemento atual (point) é igual ao valor do atributo `name` do elemento `port` que está sendo referenciado pelo atributo `instRef` de `point` e cujo arquivo que o especifica está representado no atributo `href` do elemento `type`, cujo atributo `name` é igual a `instRef`.

Embora pareça complicado, essa regra é simplesmente a aplicação de expressões XPath para resolver os nomes das referências até chegar ao atributo `href` do elemento `type`, que possui o URI da especificação do componente. Assim, como pode ser observado na regra, iniciamos observando o valor do atributo `instRef` do elemento atual (`point`).

```
current()/@instRef
```

Aplicamos esse valor em uma expressão XPath no elemento `instance` para que localize a instância cujo atributo `name` é igual a `instRef`, e extraímos o valor de `typeRef` dessa instância.

```
//instance[@name=current()/@instRef]/@typeRef]
```

O resultado dessa expressão é o nome do tipo, que da mesma forma usamos para localizar um tipo de elemento em `type` cujo atributo name seja igual a esse valor, e desse elemento `type`, extraímos o valor do atributo `href`, que é a referência que procuramos.

```
//type[@name=//instance[@name=current()/@instRef]/@typeRef]/@href)
```

De posse desse endereço podemos aplicar a função `document` e já teremos a referência do arquivo destino. Daí tentamos localizar com a expressão XPath uma porta que tenha o atributo name igual a `portRef` do elemento `point` atual. Se esse valor for encontrado é por que a referência é válida.

```
document(//type[@name=//instance[@name=current()/@instRef]/@typeRef]/@href)
        //port[@name=current()/@portRef]
```

Essa expressão é usada nas outras verificações, visto que sempre necessitaremos percorrer as referências dentro da arquitetura para localizar o arquivo onde a especificação de um determinado componente foi realizada e daí possamos realizar as verificações devidas.

A verificação seguinte é a realizada com relação ao sentido das portas. Essa verificação é apenas um incremento da expressão anterior onde, ao invés de determinarmos a existência da porta, fato que já garantimos pela verificação anterior, verificamos em cada elemento `link` se os valores de `direction` para a referência de porta do primeiro e do segundo elementos filhos `point` são compatíveis, ou seja, se o valor de `direction` da porta referenciada no primeiro elemento filho de um `link` for "out", o valor de direction da porta referenciada no segundo elemento filho desse mesmo `link` deve ser "in" ou vice-versa. Segue abaixo o esquema completo.

```
<pattern name="Sentido das Portas">
   <rule context="//link">
      <assert test="(document(//type[@name=//instance[@name=current()
         /point[1]/@instRef]/@typeRef]/@href)//port/@direction='in'
         and document(//type[@name=//instance[@name=current()/point[2]
         /@instRef]/@typeRef]/@href)//port/@direction='out')
         or (document(//type[@name=//instance[@name=current()/point[1]
         /@instRef]/@typeRef]/@href)//port/@direction='out'
         and document(//type[@name=//instance[@name=current()/point[2]
         /@instRef]/@typeRef]/@href)//port/@direction='in')">
         Sentidos de Portas Invalidos.
      </assert>
   </rule>
</pattern>
```

Esse esquema é diferenciado do anterior apenas pelo fato de que estamos partindo dos elementos `link` para realizar as buscas já que temos que considerar os pares de elementos `point`.

As próximas verificações a serem realizadas são sobre os parâmetros das portas conectadas. Assim, tanto o número de parâmetros das portas que estão conectadas na arquitetura como os tipos de dados devem ser condizentes.

Inicialmente verificaremos se é o mesmo o número de parâmetros nas duas portas que formam cada conexão (`link`) da arquitetura. O esquema para realizar essa validação é o seguinte:

```
<pattern name="Numero de Parametros">
   <rule context="//link">
      <assert test="count(document(//type[@name=//instance[@name=current()
            /point[1]/@instRef]/@typeRef]/@href)//port[@name=current()
            /point[1]/@portRef]//*)=count(document(//type[@name=
            //instance[@name=current()/point[2]/@instRef]/@typeRef]/@href)
            //port[@name=current()/point[2]/@portRef]//*)">
         Numero de Parametros Diferentes.</assert>
      </rule>
</pattern>
```

Embora um pouco grande, esse script é bem simples. A maior parte é referente a busca do nome do arquivo externo onde cada porta está sendo declarada. Essa parte foi explicada com detalhes anteriormente. O que há de diferente nesse script é que para cada link, testamos se o número de elementos filhos de cada porta é igual. Isso é o suficiente por que pela gramática XSD, os elementos port só podem ter elementos `param` como elementos filhos.

6.2.7 Validação de Consistência Sintática em Estilos Derivados

Estando as especificações Xtyle devidamente validadas com relação a seu conteúdo interno, devemos agora garantir que as relações entre estilos na descrição de estilos derivados são realmente válidas. Nessa seção apresentamos um conjunto de regras definidos em Schematron para verificar essas relações na descrição de estilos derivados.

As validações que devemos fazer dizem respeito aos tipos de componentes e links que herdamos na descrição de estilos derivados, sejam simples ou compostos. Desse modo a primeira regra que definimos foi para verificar se, caso forem declarados atributos `from` nos elementos `type`, os valores dos atributos `name` desse mesmo elemento devem estar também definidos na especificação do estilo do qual estamos derivando o estilo atual. A regra para essa verificação é a seguinte:

```
<pattern name="Referencia de Tipos em Types">
   <rule context="//type">
      <report test="@from and not(document(//style[@name=current()
                  /@from]/@href)//type[@name=current()/@name])">
      Nome de Tipo Invalido.</report>
   </rule>
</pattern>
```

Para cada elemento `type` de xtyle, essa regra inicialmente verifica se o atributo `from` está definido e em seguida, a partir do atributo `href` do elemento `style`, cujo atributo `name` é igual ao atributo `name` do elemento `type` atual, que retorna o nome do arquivo onde o estilo de onde estamos derivando o componente, verifica se o atributo `name` de algum elemento `type` desse estilo é igual ao atributo `name` do elemento `type` do estilo atual.

A regra seguinte faz a mesma verificação só que para os elementos `alias`, caso esses sejam definidos. Como cada `alias` é uma referência de um nome de tipo de componente definido da mesma forma que o próprio elemento `type`, então basta realizar uma pequena alteração no esquema anterior para realizar a verificação. Assim, temos o seguinte resultado:

```
<pattern name="Referencia de Aliases em Types">
   <rule context="//alias">
      <report test="@from and not(document(//style[@name=current()/@from]
               /@href)//type[@name=current()/@name])">
      Nome de Alias Invalido.</report>
   </rule>
</pattern>
```

Como última regra de verificação de consistência na definição de um estilo temos que observar se os elementos do tipo `link`, que também podem ser herdados, estão definidos. Também usamos a mesma abordagem, fazendo apenas uma pequena alteração no esquema e tendo como resultado o seguinte:

```
<pattern name="Referencia de Links em Topology">
   <rule context="//link">
      <report test="@from and not(document(//style[@name=current()
            /@from]/@href)//link[@name=current()/@name])">
      Nome de Link Invalido.</report>
   </rule>
</pattern>
```

Essas três regras garantem que uma especificação Xtyle seja valida com relação a outros estilos que essas venham a utilizar.

6.2.8 Conformidade de Arquiteturas com Relação a Estilos

Até o momento realizamos a validação sintática de uma especificação Xtyle e verificamos se ela é consistente com relação a outros estilos que possa utilizar. Entretanto, a real aplicação dos estilos arquiteturais é realizado pelas arquiteturas.

Um estilo é definido para fornecer tanto regras sintáticas como restrições topológicas e de comunicação, além das relações comportamentais. Uma arquitetura descrita usando ArchML deve seguir um determinado estilo arquitetural e a verificação da aderência dessa especificação arquitetural com as características definidas no estilo damos o nome de conformidade arquitetural a um estilo.

A conformidade arquitetural abrange tanto regras sintáticas como semânticas e comportamentais. Nessa seção abordaremos a verificação de conformidade arquitetural a nível sintático, sendo que as informações semânticas e comportamentais são tratadas na Seção 6.3.

Várias são as verificações que devem ser realizadas para garantir que, sintaticamente, uma arquitetura está de acordo com um determinado estilo. Alguns tipos de verificações são independentes do tipo de estilo que está sendo verificado. Já outras verificações estão intrinsecamente ligadas a um determinado estilo arquitetural.

As verificações independentes de estilos são as que realizam a observância de uma arquitetura ao vocabulário de um determinado estilo e as restrições tanto de componentes como topológicas impostas por esse estilo. Assim, de forma geral, podemos definir os seguintes tipos básicos de verificação de conformidade:

1. **Nome do Estilo**: Cada arquitetura deve referenciar um estilo (o default e Cliente-Servidor) e o nome desse deve ser verificado com relação ao esquema de validação que estamos empregando para garantir que as regras que estamos aplicando são de fato para o estilo que a arquitetura está acompanhando;

2. **Papéis dos Componentes**: Cada estilo possui um conjunto de tipos de componentes que devem ser utilizados para definir os papéis dos componentes em uma arquitetura. O atributo role de um componente é usado exatamente para realizar essa ligação de um componente com seu papel em um estilo. Esse conjunto de tipos de componentes é comumente denominado vocabulário de um estilo [41, 38];

3. **Restrições nos Componentes**: Com base nos papeis que um componente pode assumir podemos realizar um conjunto de verificações sobre a estrutura desse com relação ao seu tipo definido no estilo, como a quantidade de Portas de entrada, ou saída; e

4. **Conexões Permitidas e Fluxo de Dados**: Além das restrições na estrutura dos componentes um estilo também impõe um conjunto de regras de conexão entre os tipos. Assim, dependendo dos papeis dos componentes esses podem ou não conversar entre si. Mais ainda, além da definição das possibilidades de comunicação entre os tipos, o fluxo dos dados também é definido pelo estilo.

Para exemplificar, a seguir apresentamos as regras de verificação de conformidade para o estilo Rede de Fluxo de Dados. A especificação dos demais estilos disponibilizados por DraX pode ser encontrada em [113].

Rede de Fluxo de Dados

- **Regra 1:** O nome do estilo tem que ser DataflowNetwork (Código 6.1).

Código 6.1 Validação do Nome do Estilo.

```
<rule context="//style">
   <assert test="document(@href)//xtyle/@name='DataFlowNetwork'">
      Estilo Invalido
   </assert>
</rule>
```

Essa regra define que, quando estivermos verificando a conformidade de um estilo que supostamente segue o estilo Rede de Fluxo de Dados, deveremos encontrar no elemento `style` uma referência a descrição do estilo cujo nome é *DataflowNetwork*.

- **Regra 2:** Os tipos de componentes permitidos são Filter, Sink e Source (Código 6.2).

Código 6.2 Validação de Tipos de Componentes.

```
<rule context="type">
   <assert test="document(@href)//interface[@role='Filter'] or
         document(@href)//interface[@role='Source'] or
         document(@href)//interface[@role='Sink']
   ">Estilo Invalido
   </assert>
</rule>
```

Podemos observar que estamos verificando se nos tipos de componentes definidos para a arquitetura existe alguma interface cujo **role** seja igual a um dos tipos de componentes permitidos para estilo.

- **Regra 3:** Os componentes Source, Filter e Sink, devem ter um número determinado de portas (Código 6.3).

Código 6.3 Validação do Número de Portas.

```
<rule context="type">
   <assert test="( count(document(@href)//interface[@role='Filter']
          //port[@direction='in'])>=1 and
         count(document(@href)//interface[@role='Filter']
          //port[@direction='out'])>=1 ) or
         ( count(document(@href)//interface[@role='Source']
          //port[@direction='in'])=0 and
         count(document(@href)//interface[@role='Source']
          //port[@direction='out'])>=1 ) or
         ( count(document(@href)//interface[@role='Sink']
          //port[@direction='in'])>=1 and
         count(document(@href)//interface[@role='Sink']
          //port[@direction='out'])=0 )
         ">Estilo Invalido
   </assert>
</rule>
```

Podemos observar que estamos verificando se o número de portas para tipo de cada tipo (in,out) em cada tipo de componente seguindo as regras do estilo.

- **Regra 4:** Somente pode haver conexões entre componentes com papeis Source-Filter, Filter-Filter ou Filter-Sink, e sempre nesse sentido (Código 6.4).

Essa regra verifica para cada elemento `link` da arquitetura qual o papel do tipo de componente cuja porta estamos referenciando no primeiro elemento `point` do link se o papel do tipo de componente cuja porta estamos referenciado no segundo elemento `point`, seguem a seqüência exigida na descrição do estilo.

Código 6.4 Validação de Conexões.

```
<rule context="link">
   <assert test="( document(//type[@name=//instance[@name=current()
      /point[1]/@instRef]/@typeRef]/@href)//port[@name=current()
      /point[1]/@portRef]/../@role='Source' and document(//type[@name=
      //instance[@name=current()/point[2]/@instRef]/@typeRef]/@href)
      //port[@name=current()/point[2]/@portRef]/../@role='Filter') or
      ( document(//type[@name=//instance[@name=current()/point[1]/
      @instRef]/@typeRef]/@href)//port[@name=current()/point[1]/@portRef]
      /../@role='Filter' and document(//type[@name=//instance[@name=
      current()/point[2]/@instRef]/@typeRef]/@href)//port[@name=current()
      /point[2]/@portRef]/../@role='Filter') or ( document(//type
      [@name=//instance[@name=current()/point[1]/@instRef]
      /@typeRef]/@href)//port[@name=current()/point[1]/@portRef]
      /../@role='Filter' and document(//type[@name=//instance
      [@name=current()/point[2]/@instRef]/@typeRef]/@href)//port
      [@name=current()/point[2]/@portRef]/../@role='Sink')
   ">Estilo Invalido
   </assert>
</rule>
```

- **Regra 5:** Obrigatoriedade da arquitetura possuir componentes com todos os papéis (Código 6.5).

Código 6.5 Validação de Papéis dos Componentes.

```
<rule context="types">
   <assert test="count(document(//type/@href)//interface
            [@role='Filter'])>=1
            and count(document(//type/@href)//interface
            [@role='Source'])=1
            and count(document(//type/@href)//interface
            [@role='Sink'])=1
   ">Estilo Invalido
   </assert>
</rule>
```

Essa regra é uma regra específica para o estilo Rede de Fluxo de Dados, ou seja, nem todos os outros estilos devem segui-la, nela verificamos se os componentes usados na arquitetura realmente possuem todos os papéis definidos pelo estilo, pois uma comunicação nesse estilo só ocorre se houver os componentes que possuam os três papéis. Assim, para o elemento `types` contamos quantos elementos filhos possuem uma interface com os papéis impostos pelo estilo. Obrigatoriamente temos que ter pelo menos um de cada papel.

6.2.8.1 Conformidade com Estilos Definidos-Pelo-Usuário

As regras de verificação de conformidade de estilos básicos e derivados de Xtyle conseguem verificar um conjunto de características em uma arquitetura que são exigidas por um estilo arquitetural determinado. Contudo, usando Xtyle, o desenvolvedor pode criar seus próprios

estilos arquiteturais e esses, da mesma forma, devem ser passíveis de verificação com relação às arquiteturas que o seguem.

Entretanto, cada novo estilo desenvolve seu próprio vocabulário e suas próprias restrições topológicas, sendo impossível (ou pelo menos muito difícil) se criar esquemas para todas as possibilidades fornecidas por Xtyle. Contudo, com uma observação mais comparativa com relação às regras de cada um dos estilos básicos e derivados apresentados em [113], pudemos concluir que algumas delas se repetem em todos os estilos. Essas regras gerais dizem respeito ao vocabulário do estilo e às restrições sintáticas impostas na descrição desses estilos por Xtyle. Assim, se não podemos escrever esquemas de verificação de conformidade arquitetural individuais para cada possibilidade fornecida por Xtyle, podemos escrever um script que gere esses esquemas individuais baseado na descrição Xtyle. Ficando para o desenvolvedor apenas a tarefa de escrever, se for necessário, mais algumas regras específicas do estilo no esquema gerado.

Olhando com cuidado, observamos que as regras de 1 a 4 apresentadas anteriormente são constantes em todos os estilos e que essas podem ser facilmente extraídas diretamente das especificações Xtyle. A seguir, apresentamos um script (denominado `GenerateScrematronValidateXtyle.xsl`) que usa XSLT [18] para gerar os esquemas Schematron de verificação de conformidade baseada em uma especificação Xtyle para um estilo definido pelo usuário.

- **Geração da Regra 1**: Verificação do Nome do Estilo (Código 6.6).

Código 6.6 Geração da Validação de Nome.

```
<xsl:template match="xtyle">
   <sch:pattern name="Verificacao do Nome do Estilo">
      <sch:rule context="//style">
         <sch:assert test="document(@href)//xtyle/@name='@name'
         ">Estilo Invalido</sch:assert>
      </sch:rule>
   </sch:pattern>
   <xsl:apply-templates select="types"/>
   <xsl:apply-templates select="topology"/>
</xsl:template>
```

Essa primeira parte do *script*, gera a regras do esquema Schematron para a verificação do nome do estilo na especificação de arquitetura. Além disso, ela invoca os templates para as outras porções do arquivo Xtyle.

- **Geração da Regra 2**: Verificação de Tipos de Componentes Permitidos (Código 6.7).

Código 6.7 Geração de Validação de Tipos.

```
<sch:pattern name="Verificacao de Tipos de Componentes Permitidos">
   <sch:rule context="type">
      <sch:assert>
         <xsl:attribute name="test">
            <xsl:for-each select="type»document(@href)//
             interface[@role='<xsl:value-of select="@name"/>']
              <xsl:if test="not(position()=last())"> or
              </xsl:if>
            </xsl:for-each>
         </xsl:attribute>
         Estilo Invalido
      </sch:assert>
   </sch:rule>
</sch:pattern>
```

Podemos observar que as regras que dizem respeito ao elemento types de uma descrição Xtyle estão todas contidas em um mesmo template XSL, sendo que essa primeira regra desse template, permite a geração de uma regra Schematron de verificação dos tipos de componentes de uma arquitetura com relação ao estilo.

- **Geração da Regra 3**: Verificação do Número de Portas dos Tipos de Componentes (Código 6.8).

Código 6.8 Geração de Validação de Número de Portas.

```
<sch:pattern name="Numero de Portas">
   <sch:rule context="type">
      <sch:assert>
         <xsl:attribute name="test">
            <xsl:for-each select="type">(
            count(document(@href)//interface[@role=
               '<xsl:value-of select="@name"/>']//port[@direction='in'])
               <xsl:variable name="minin">
                  <xsl:value-of select="current()/ports/in/@minOccurs"/>
               </xsl:variable>
               <xsl:variable name="maxin">
                  <xsl:value-of select="current()/ports/in/@maxOccurs"/>
               </xsl:variable>
               <xsl:variable name="numin">
                  <xsl:value-of select="count(current()/ports/in)"/>
               </xsl:variable>
            <xsl:choose>
               <xsl:when test = "$minin='*' and $maxin='*' and
                  number($numin)!=0">
               <xsl:text»=1</xsl:text>
               </xsl:when>
               ...
            </xsl:choose>
            and


             count(document(@href)//interface[@role='<xsl:value-of
             select="@name"/>']//port[@direction='out'])
             <xsl:variable name="minout">
                <xsl:value-of select="current()/ports/out/@minOccurs"/>
             </xsl:variable>
             <xsl:variable name="maxout">
                <xsl:value-of select="current()/ports/out/@maxOccurs"/>
             </xsl:variable>
             <xsl:variable name="numout">
                <xsl:value-of select="count(current()/ports/out)"/>
             </xsl:variable>
          <xsl:choose>
             <xsl:when test = "$minout='*' and $maxout='*' and
             number($numout)!=0">
          <xsl:text»=1</xsl:text>
          </xsl:when>
          ...
       </xsl:choose>
          <xsl:if test="not(position()=last())">
          ) or
          </xsl:if>
          <xsl:if test="position()=last()">
          )
          </xsl:if>
       </xsl:for-each>
    </xsl:attribute>
    Estilo Invalido
    </sch:assert>
    </sch:rule>
    </sch:pattern>
 </xsl:template>
```

Essa parte do *script* de geração de esquema de validação de conformidade está abreviada para tentar tornar mais clara o entendimento do mesmo Ele é a última parte do template XSLT iniciado na regra anterior e permite a geração de uma regra de verificação do número de portas dos componentes de uma arquitetura e o padrão imposto pelo estilo.

- **Geração da Regra 4**: Verificação dos Tipos de Componentes Passíveis de Conexão (Código 6.9).

Código 6.9 Geração de Validação de Conexões.

```
<xsl:template match="topology">
   <sch:pattern name="Tipos de Conexao">
   <sch:rule context="link">
      <sch:assert> <xsl:attribute name="test">
         <xsl:for-each select="link">( document(//type[@name=
          //instance[@name=current()/point[1]/@instRef]/@typeRef]
          /@href)//port[@name=current()/point[1]/@portRef]/../@role=
          '<xsl:value-of select="@start"/>' and document(//type[@name=
          //instance[@name=current()/point[2]/@instRef]/@typeRef]
          /@href)//port[@name=current()/point[2]/@portRef]/../@role=
          '<xsl:value-of select="@end"/>'
          <xsl:if test="not(position()=last())">
            <xsl:text> ) or </xsl:text>
          </xsl:if>
          <xsl:if test="position()=last()">
            <xsl:text> ) </xsl:text>
          </xsl:if>
         </xsl:for-each>
      </xsl:attribute>
   Tipo de Ligacao Invalido
   </sch:assert>
   </sch:rule>
   </sch:pattern>
</xsl:template>
```

Essa parte do *script* atua sobre o elemento topology da descrição Xtyle. Ele gera uma regra Schematron para verificar as possíveis conexões entre tipos de componentes permitidos pelo estilo.

Esse *script* XSLT permite a geração automática de um esquema Schematron a partir de uma especificação de estilo feita pelo desenvolvedor. Esse esquema contém um conjunto de regras para a verificação de conformidade de uma arquitetura com relação ao novo estilo descrito. Com esse *script*, tornamos possível a verificação de conformidade sem a necessidade de se escrever manualmente um esquema Schematron para isso.

6.3 Validação Comportamental em DraX

A verificação de consistências comportamentais em arquiteturas serve para dois propósitos básicos: a verificação da compatibilidade entre componentes que formam uma arquitetura em um nível semântico e a verificação de propriedades na arquitetura como um todo.

Como apresentado nas Seções 5.3 e 5.4, a especificação de comportamentos de componentes de arquiteturas e de tipos em estilos ocorre pela incorporação de um arquivo XMI gerado por por uma ferramenta UML a partir da descrição usando DDPs do comportamento observável do componente com relação às suas portas.

A verificação de comportamento é realizada automaticamente em DraX visto que desenvolvemos diversos *scripts* de geração de especificações π-cálculo que podemos utilizar para realizar as análises utilizando uma ferramenta como o MWB [120]. Assim, o desenvolvedor não precisa dominar as estruturas algébricas de π-cálculo basta apenas executar o *script* de geração de especificação e utilizar o resultado gerado no MWB.

Além disso, podemos também verificar se os componentes de uma arquitetura têm comportamentos compatíveis com os seus respectivos tipos relativos a um estilo que a arquitetura esteja utilizando. Usamos o mesmo raciocínio anterior para realizar essas verificações.

A etapa de verificação de consistências comportamentais de DraX pode ser desmembrada nas seguintes fases:

1. **Verificação de Compatibilidade Comportamental entre Componentes**
   Nessa fase é realizada uma verificação se os componentes que estão conectados em uma arquitetura são de fato comportamentalmente compatíveis.

2. **Verificação de Conformidade de Estilos**
   Como cada tipo de um estilo também possui uma especificação comportamental correspondente, e cada componente de uma arquitetura deve possuir pelo menos um tipo relativo a um estilo, nessa etapa realizamos uma verificação se o comportamento dos componentes são compatíveis com seus respectivos tipos estilísticos.

3. **Análise de Propriedades Arquiteturais**
   A arquitetura como um todo é verificada nessa etapa. Para realizar essa análise, utilizamos o cálculo $\mathcal{R}\pi$, que propomos especificamente para facilitar a descrição da semântica de especificações de arquitetura em ArchML. Com esse cálculo, podemos facilmente verificar propriedades como impasses em arquiteturas ArchML.

Ao longo dessa seção cada uma dessas etapas é detalhada.

6.3.1 Validação de Compatibilidade Comportamental entre Componentes

A verificação de compatibilidade entre componentes que formam uma arquitetura é realizada através de especificações formais em π-cálculo. Como um dos objetivos principais de DraX é permitir que os desenvolvedores possam realizar a construção e verificação de especificações distribuídas sem ter que lançar mão de conceitos ou tecnologias que não façam parte do seu dia-a-dia, seria contraditório deixar que os desenvolvedores criassem essas especificações algébricas manualmente. Para suplantar esse problema, desenvolvemos um programa Java baseado em uma biblioteca XPath, denominada Jaxen [2], de modo que fosse possível se criar automaticamente especificações π-cálculo a partir das descrições XMI dos comportamentos dos componentes.

Denominamos esse programa Java de *script* XMI2PI em virtude de sua funcionalidade de gerar especificações π-cálculo rastreia a estrutura do arquivo XMI e cria especificações π-cálculo para cada agente encontrado.

6.3.1.1 O *Script* XMI2PI

As regras de geração dos diagramas DDPs em π-cálculo desenvolvidas em XMI2PI são encontradas em [5], onde é apresentado um programa perl que gera especificações CCS a partir de especificações de diagramas de estado UML descritos em XMI. Também em [32] é apresentado um conjunto de regras de geração de especificações π-cálculo a partir de diagramas UML. Em XMI2PI tivemos que realizar algumas adaptações para capturar as características dos DDPs, contudo, preservamos todas as regras de transformação provadas em [5] e [32]. Nesse *script* utilizamos construtores intermediários para calcular nomes livres e conectados (*free* e *bound names*) além de definir canais privados entre processos de um mesmo agente.

De um modo geral, na implementação do *script* XMI2PI utilizamos um conjunto de vetores para armazenar informações sobre o sistema. Assim podemos destacar os seguintes vetores como sendo principais:

- **Source**: armazena os agentes origem das transições;
- **Target**: armazena os agentes destino das transições;
- **Channel**: armazena os nomes das transições em si;
- **ChannelType**: armazena os tipos das transições para posterior criação dos agentes; e
- **ChannelMode**: armazena os modos de comunicação de cada agente baseada em estereótipos UML.

Para facilitar a manipulação das informações no arquivo XMI optamos por utilizar uma biblioteca que permitisse a manipulação das informações através de XPath. Dentre diversas bibliotecas avaliadas, resolvemos utilizar a biblioteca Jaxen [2]. Com Jaxen podemos interagir com o arquivo XMI através de poucas linhas de código utilizando expressões XPath.

Tratando agora da implementação do script XMI2PI, além da facilidade trazida por Jaxen, resolvemos criar dois métodos Java para extrair valores do arquivo XMI, o primeiro, denominado `searchName`, apresentado em parte a baixo permite se passar uma expressão XPath que deve retornar o valor de um atributo em forma de uma `string`.

```
private String searchName(String xpath){
   String resultAux = null;
      XPath expression = new org.jaxen.dom.DOMXPath(xpath);
      expression.addNamespace("UML","//org.omg/UML/1.3");
      navigator = expression.getNavigator();
      List result = expression.selectNodes(doc);

      Iterator iterator = result.iterator();

      while (iterator.hasNext()) {
         Node res = (Node)iterator.next();
         String value = StringFunction.evaluate(res, navigator);
         resultAux=value;
      }

      ...

      return(resultAux);
}
```

O segundo método, denominado `searchElement`, cujo trecho de código está apresentado a seguir, retorna um conjunto de elementos resultante de uma pesquisa XPath em um vetor.

```
private Vector searchElements(String xpath){

   Vector resultAux = new Vector(1);

      XPath expression = new org.jaxen.dom.DOMXPath(xpath);
      expression.addNamespace("UML","//org.omg/UML/1.3");
      navigator = expression.getNavigator();
      List result = expression.selectNodes(doc);

      Iterator iterator = result.iterator();

      while (iterator.hasNext()) {
         Node res = (Node) iterator.next();
         String value = StringFunction.evaluate(res, navigator);
         resultAux.addElement(value);
      }
      return(resultAux);
}
```

Para caminhar no arquivo XMI e gerar a especificação π-cálculo resolvemos seguir a seguinte estratégia: Inicialmente procuramos todos as transições e recolhemos o valor do atributo `xmi.id` em um vetor auxiliar.

```
Vector results = searchElements("//UML:Transition/@xmi.id");
```

Em seguida vasculhamos o vetor resultando e procuramos os atributos, `source` e `target` de cada transição, ignorando as transições cujo valor seja `FinalState`.

```
for(int i=0; i<Temp.size(); i++){

   String id = Temp.elementAt(i).toString();
   String tName = searchName("//UML:Transition
      [@xmi.id='"+id+"']/@name");

   String tSourceRef = searchName("//UML:Transition
      [@xmi.id='"+id+"']/@source");
   String tSourceName = searchName("//UML:SimpleState
      [@xmi.id='"+tSourceRef+"']/@name");

   String tTargetRef = searchName("//UML:Transition
      [@xmi.id='"+id+"']/@target");

   String tTargetName = searchName("//UML:SimpleState
      [@xmi.id='"+tTargetRef+"']/@name");
   if(tTargetName==null)
      tTargetName = new String("FinalState");

   // Calculo do Tipo de Canal
   String name = null;
   if(tName.indexOf(":")!=-1)
      name = tName.substring(tName.indexOf(":")+1);
   else
      name = tName;
```

Como temos como regra na construção dos DDP (vide Seção 5.3) que cada porta é na verdade um método do objeto, sendo que métodos privados são portas de saída e métodos públicos são portas de entrada, verificamos se para nome de transição existe um elemento XMI `Operation` cujo atributo `name` seja igual ao nome da transição e daí verificamos qual o valor do atributo `visibility`. Se esse atributo for `private` essa será uma porta de saída, senão será uma porta de entrada.

```
String tNameType = null;
   String opType = searchName("//UML:Operation
      [@name='"+name+"']/@visibility");

   if(opType == null || opType.equals("private"))
      tNameType = new String("out");
   else
      tNameType = new String("in");
```

Também para cada transição, verificamos se existem algum estereótipo associado. Se não houver podemos concluir que a comunicação é síncrona (default) se houver um estereótipo capturamos o identificador (`@stereotype`) e verificamos no elemento `Stereotype` cujo identificador (`xmi.id`) seja igual ao valor capturado. Se esse valor for `asynchronous` a comunicação é assíncrona senão a comunicação é síncrona.

```
String tNameMode = null;
   String stereotypeRef = searchName("//UML:Transition
      [@xmi.id='"+id+"']/@stereotype");

   if(stereotypeRef == null)
      tNameMode = new String("sync");
   else{
      String mode = searchName("//UML:Stereotype
         [@xmi.id='"+stereotypeRef+"']/@name");

   if(mode.equals("asynchronous") )
      tNameMode = new String("async");
   else
      tNameMode = new String("sync");
   }
```

Em seguida preenchemos o valor dos vetores com as informações colhidas e seguimos para a próxima transição.

```
            Source.addElement(tSourceName);
            Target.addElement(tTargetName);
            Channel.addElement(tName);
            ChannelType.addElement(tNameType);
            ChannelMode.addElement(tNameMode);

         }
```

A partir daí, realizamos um conjunto de operações intermediárias com vetores para permitir a geração corretas das especificações π-cálculo sendo que em seguida invocamos o método `generatePI`, onde a especificação π-cálculo é de fato gerada.

Logo no início do método `generatePI`, realizamos algumas adaptações nos vetores, como por exemplo trocar os nomes `compute`, que representa uma transição interna pelo similar em π-cálculo que é "t". Uma ressalva com relação à forma de geração da especificação, é que utilizamos a sintaxe textual [120] para π-cálculo utilizada pelo MWB.

Para exemplificar a geração de uma especificação, seja o diagrama DDP da Figura 6.1, que representa o exemplo de um buffer de duas posições.

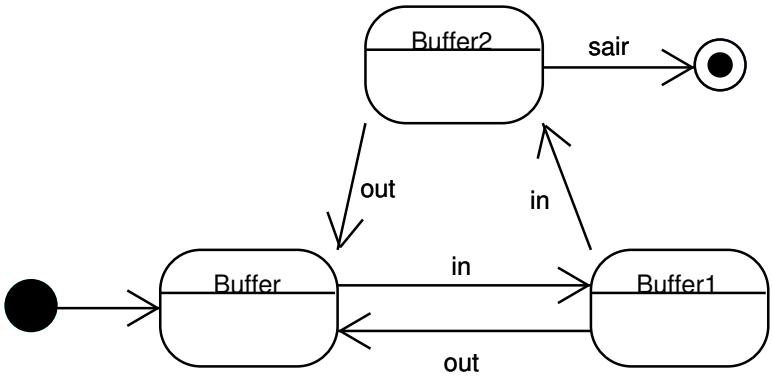


Figura 6.1: DDP de Buffer.

A partir dessa especificação geramos o código XMI correspondente e aplicamos o *script* XMI2PI, que irá gerar a seguinte especificação π-cálculo:

$$
\begin{aligned}
&agent\ Buffer(in, out, sair) = in(data1).Buffer1(in, out, sair)\\
&agent\ Buffer1(in, out, sair) = (\hat{}data3)(in(data2).\\
&\quad Buffer2(in, out, sair) +' out < data3 > .Buffer(in, out, sair))\\
&agent\ Buffer2(in, out, sair) = (\hat{}data4, data5)'out < data4 > .\\
&\quad Buffer(in, out, sair) +' sair < data5 > .0
\end{aligned}
$$

6.3.1.2 Compatibilidade de Comportamentos

A idéia de compatibilidade está relacionada a possibilidade de interação entre componentes que estão conectados em uma arquitetura. Assim, dizemos que dois componentes são compatíveis se eles são passíveis de interação. Para se verificar formalmente essa propriedade dois mecanismos são propostos na literatura. O primeiro, proposto em [17] é voltado explicitamente para π-cálculo e descreve compatibilidade entre componentes através de uma definição formal complexa, onde processos compatíveis devem ser capazes de engajar em transições complementares e sua composição em paralelo deve ser possível.

A segunda abordagem, definida em [10] utiliza a noção de bissimulação fraca para verificar a compatibilidade de comportamentos entre processos, onde dois processos são compatíveis se suas representações baseadas na exclusão das interações entre canais não conectados (operação *hiding* de CCS) e a modificação dos nomes dos canais para que possa ser garantida a sincronização (operação de *relabeling* de CCS) gere especificações passíveis de funcionarem em execução paralela.

Na escolha da abordagem que deveria ser utilizada por DraX, a adoção do primeiro mecanismo deixaria essa etapa do desenvolvimento sem nenhum atrativo para a comunidade de desenvolvedores, visto que uma abordagem formal e ainda não automatizada deve ser utilizada, e isso vai diretamente contra aos objetivos dessa tese. Já o segundo mecanismo, mesmo sendo facilmente adaptado para π-cálculo utiliza a noção de *hiding* e *relabeling* que não são operações diretamente implementadas em π-cálculo.

Como queríamos garantir que os desenvolvedores não tivessem que manipular diretamente expressões algébricas, decidimos construir um *script* que manipula as especificações geradas pelo *script* XMI2PI e simula as tarefas de *relabeling* e *hiding* permitindo a verificação das especificações resultantes diretamente no MWB. Denominamos esse *script* de `CompCheck.java`

Para exemplificar, se utilizarmos a especificação gerada pelo XMI2PI (retirando a operação de saída θ) para os componentes da aplicação apresentada na Figura 6.2, teríamos as especificações seguintes:

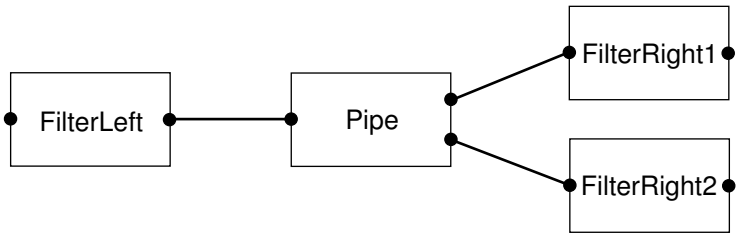


Figura 6.2: Aplicação Exemplo.

$$agent\ Filter(accept\_item, serve\_item) = accept\_item(data1).Filter1(accept\_item, serve\_item)$$

$$agent\ Filter1(accept\_item, serve\_item) = (\hat{}data3)(accept\_item(data2).\ Filter2(accept\_item, serve\_item) +' serve\_item < data3 > .\ Filter(accept\_item, serve\_item))$$

$$agent\ Filter2(accept\_item, serve\_item) = (\hat{}data4)'serve\_item < data4 > .\ Filter(accept\_item, serve\_item)$$

$$agent\ Pipe(accept\_item, forward\_item1, forward\_item2) = (accept\_item(data1).('forward\_item1 < data1 > .\ Pipe(accept\_item, forward\_item1, forward\_item2)+ 'forward\_item2 < data1 > .\ Pipe(accept\_item, forward\_item1, forward\_item2)))$$

$$agent\ F0(accept\_item, serve\_item) = Filter(accept\_item, serve\_item)$$

$$agent\ F1(accept\_item, serve\_item) = Filter1(accept\_item, serve\_item)$$

$$agent\ F2(accept\_item, serve\_item) = Filter2(accept\_item, serve\_item)$$

$$agent\ P(accept\_item, forward\_item1, forward\_item2) = Pipe(accept\_item, forward\_item1, forward\_item2)$$

$$agent\ System() = (\hat{}accept\_item, serve\_item, forward\_item1, forward\_item2)\Big(F0(accept\_item, serve\_item) \mid F1(accept\_item, serve\_item) \mid F2(accept\_item, serve\_item) \mid P(accept\_item, forward\_item1, forward\_item2)\Big)$$

Para verificarmos a compatibilidade comportamental entre a instância P e $F1$, por exemplo, devemos verificar se há impasses na composição em paralelo da especificação dessas instâncias, observando-se apenas as conexões entre elas e renomeando os canais que estão conectados para um mesmo nome (para que haja sincronização).

Assim, devemos utilizar o *script* `CompCheck` na especificação original que gerará a especificação contendo especificações parciais de cada componente conectado dois a dois tendo apenas os canais dos componentes que estão se comunicando de forma que possamos verificar a compatibilidade. A seguir apresentamos a especificação gerada para os componentes $Pipe$ e $Filter$.

$$agent\ Filter(accept\_item) = accept\_item \\ (data1).Filter1(accept\_item)$$

$$agent\ Filter1(accept\_item) = (\hat{}data3)(accept\_item \\ (data2).Filter2(accept\_item))$$

$$agent\ Filter2(accept\_item) = (\hat{}data4)Filter(accept\_item)$$

$$agent\ Pipe(forward\_item1) = (\hat{}data1) \\ 'forward\_item1 < data1 > .Pipe(forward\_item1)$$

A verificação de compatibilidade se verifica através da validação da comunicação de um sistema formado pela composição em paralelo dos componentes a serem avaliados. Assim, temos a seguinte estrutura gerada:

$$agent\ System() = (\hat{}a)(Filter(a) \mid Pipe(a))$$

Executando o MWB para verificar se há impasses na especificação, como mostrado a baixo, verificamos que as especificações são compatíveis.

```
MWB> deadlocks System()
No deadlocks found.
MWB>
```

Essa estratégia, embora resolva o problema de verificação de compatibilidade, não é uma estratégia fácil de ser aplicada manualmente, caso queiramos realizar algum outro tipo de verificação formal, visto que a renomeação de canais não é muito simples de ser realizada em grandes sistemas com uma estrutura complexa de interconexões. Sendo que esse problema de fato é gerado pela restrição das álgebras de processos em realizar sincronização apenas entre canais com mesmos nomes e sentidos contrários. Para contornar esse problema, propomos mais adiante nessa seção, uma álgebra de processos, denominada cálculo $\mathcal{R}\pi$, que é baseada em π-cálculo e onde desenvolvemos um operador que permite a construção de especificações (re)configuráveis, cuja semântica formal permite a sincronização entre canais com nomes diferentes. $\mathcal{R}\pi$ fornece um mecanismo mais adaptado para formalização da estrutura geral de interconexão de arquiteturas de software e, desse modo, é utilizada para dar semântica a ArchML.

6.3.2 Validação de Conformidade de Estilos

Tendo realizado a verificação de compatibilidade comportamental de todos os componente da arquitetura, o próximo passo é verificar se o comportamento de cada componente é realmente compatível com os tipos definidos em ArchML com relação ao estilo que a arquitetura segue. Denominamos esse tipo de validação de verificação de conformidade de estilos.

Usando as ferramentas de verificação sintática e estrutural em um passo anterior no processo de desenvolvimento com DraX (definidas na Seção 6.2), podemos garantir as arquiteturas já estão em conformidade estrutural com os estilo.

Para verificar se um componente possui um comportamento realmente compatível com o tipo definido em um estilo arquitetural devemos comparar se a representação algébrica do componente é fracamente bissimilar a expressão algébrica do tipo seu tipo correspondente [10, 11].

A geração da representação algébrica do comportamento do componente é realizada pelo *script* XMI2PI. Para que esse mesmo *script* pudesse gerar também as especificações algébricas dos tipos dos estilos, fizemos algumas alterações no mesmo. Essas alterações foram realizadas de forma a adaptar o *script* para gerar as informações algébricas intermediárias requeridas nos estilos. Essas informações, por sua vez, dizem respeito a representação generalizada do comportamento dos tipos nos estilos (vide Seção 5.4).

A representação gerada por XMI2PI para o tipo GServer [114], que possui diversas portas de entrada e diversas portas de saída, é a seguinte:

$$
\begin{aligned}
&agent\ GServer(in[1..m], out[1..n]) = \\
&\qquad in[m](l).t.\,'out[n] < l > .GServer(in[1..m], out[1..n]) \mid \\
&\qquad in[m](l).t.\,'out[n] < l > .GServer(in[1..m], out[1..n])
\end{aligned}
$$

Como apresentado na Seção 5.4, essas representação informa sobre o comportamento do componente com relação às suas portas de entrada e saída, desse modo são utilizados apenas os nomes `in` e `out` genericamente para representar as portas do tipo, sendo que informações sobre o sequenciamento de execução também está representado. Contudo, na representação comportamental de um componente, as portas possuem nomes bem definidos e um número determinado. Para que seja possível verificar bisimilaridades entre as representações algébricas geradas por XMI2PI, devemos adaptar as representações algébricas resultados de XMI2PI. De um lado devemos uniformizar os nomes das portas da representação dos componentes, transformando as portas de entrada em `in` e as de saída em `out`. Por outro lado devemos expandir a representação das representações dos tipos no estilo com o mesmo número de portas de entrada e saída dos componentes que estamos querendo checar a bisimilaridade. Para realizar essa tarefa construímos um novo *scripts* que trabalha sobre as representações algébricas resultantes de XMI2PI para adaptá-las para a realização dos teste de bisimulação. Denominamos esse *script* de `AdaptXtyle.java`

Para ver seu funcionamento, supondo que queiramos verificar se o componente cujo comportamento está definido na Figura 6.1 com o comportamento do tipo do estilo GServer apresentado anteriormente. Devemos aplicar o *script* em ambas as especificações e teremos os seguintes resultados:

$$
\begin{aligned}
&agent\ Buffer(in, out) = in(data1).Buffer1(in, out) \\
&\\
&agent\ Buffer1(in, out) = (\hat{}data3)(in(data2).Buffer2(in, out) + \\
&\qquad 'out < data3 > .Buffer(in, out)) \\
&\\
&agent\ Buffer2(in, out) = (\hat{}data4) \\
&\qquad 'out < data4 > .Buffer(in, out)
\end{aligned}
$$

$$
\begin{aligned}
&agent\ GServer(in, out) = \\
&\qquad in(data1).t.'out < l > .GServer(in, out) \mid \\
&\qquad in(data1).t.'out < l > .GServer(in, out)
\end{aligned}
$$

Depois da aplicação do *script* devemos simplesmente submeter as especificações resultantes ao MWB para testar se são bisimilares. De fato, o resultado obtido com o MWB permite concluirmos que o comportamento do componente está condizente com o comportamento do tipo definido no estilo.

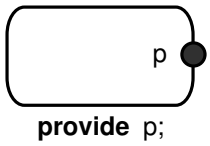
6.3.3 O Cálculo $\mathcal{R}\pi$ e a Semântica Formal de ArchML

6.3.3.1 Formalização de Aplicações Reconfiguráveis em π-cálculo

O propósito de modelar um sistema em π-cálculo é fornecer uma semântica precisa à linguagem. Nessa seção utilizaremos a linguagem Darwin para mostrar como utilizar π-cálculo com essa finalidade. Nesse sentido, será mostrado que Darwin especifica corretamente um conjunto de instâncias de componentes primitivos em uma configuração requerida em tempo de execução.

Conexões

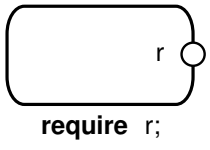
As conexões em Darwin são modeladas através de construtores relativos a requisição, fornecimento e ligação de serviços. A declaração de um serviço fornecido, **provide** p, em Darwin é modelada em π-cálculo como o agente $Prov(p, s)$ que é acessado pelo nome p e é gerencia o serviço s, como mostrado a seguir:



$$Prov(p, s) \stackrel{def}{=} !p(x).\bar{x}s$$

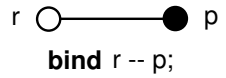
O serviço s é simplesmente o nome ou referência para um serviço que deve ser implementado pelo componente. Darwin não trata de como o serviço s é implementado, ele trata apenas da localização s do serviço que é requisitado por outros componentes. Assim, o agente $Prov$ recebe a localização de x na qual o serviço é requisitado e envia s para aquela localização. Visto que pode haver mais de um cliente para cada serviço, o agente $Prov$ é definido como um processo replicado (!) que irá enviar repetidamente a referência do serviço cada vez que a localização for requerida.

A declaração de um serviço requerido, **require** r, é modelada pelo agente $Req(r, l)$ que é acessado pelo nome r e que gerencia a localização l na qual o serviço é requerido. Novamente, Darwin não trata de como um componente cliente trata um serviço, ele deve apenas garantir que a referência para o serviço é colocada em algum local no componente cliente. O agente Req recebe o nome de acesso do agente $Prov$ e envia a localização l para esse agente. Uma requisição em Darwin deve ser utilizada por um único serviço e assim, o agente Req envia a localização l precisamente uma vez, como mostrado a baixo:



$$Req(r, l) \stackrel{def}{=} r(y).\overline{y}l$$

O construtor de ligação em Darwin é modelado pelo agente *Bind* que simplemente envia o nome de acesso do agente *Prov* para o agente *Req*.



$$Bind \stackrel{def}{=} \overline{r}p$$

Componentes

Agente ou processos em π-cálculo não podem ser diretamente nomeados, ao invés disso, agentes são acessados através de nomes de canais. Embora Darwin utilize nomes para instâncias de componentes, esses nomes são utilizados apenas para qualificar os nomes dos serviços que os componentes requerem ou fornecem. Isso é ilustrado pela tradução da configuração Darwin em π-cálculo apresentada na Figura 6.3.

Cada componente primitivo é representado por um agente que é uma composição dos agentes *Prov* e *Req* que gerenciam os serviços fornecidos e requeridos dos agentes os quais estão definindo o comportamento. O componente "Server" da Figura 6.3 é representado pelo seguinte agente π-cálculo:

$$Server(p) \stackrel{def}{=} (vs)(Prov(p, s) \mid Server'(s))$$

no qual $Server'$ representa o comportamento implementado pelo usuário do componente *Server* que fornece o serviço s. Similarmente, o agente *Client* é representado pelo seguinte agente:

$$Client(r) \stackrel{def}{=} (vl)(Req(r, l) \mid Client'(l))$$

Deve ser observado que o escopo do serviço s é local ao agente *Server* e, similarmente, o nome no qual o serviço é requerido l é também local ao agente *Client*. Como pode ser visto a seguir, a operação de ligação `bind` estende o escopo desses nomes.

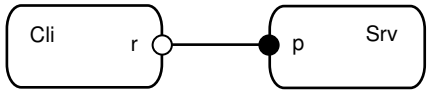
Configurações

A configuração do componente `System` da Figura 6.3 é representada em π-cálculo pela composição em paralelo dos agentes *Client*, *Server* e *Bind*:

$$System \stackrel{def}{=} (vr_A, p_B)(Client(r_A) \mid Server(p_B) \mid Bind(r_A, p_B))$$

As instâncias A e B na Figura 6.3 são usadas apenas para qualificar e assim, renomear o nome requerido r e o nome fornecido p dos componentes *Client* e *Server*. A expressão anterior é a tradução precisa da configuração Darwin da Figura 6.3. Para demonstrar que o modelo está correto, deve ser mostrado que a instância cliente A irá obter a referência do serviço fornecida pelo servidor B quando a configuração é elaborada. Substituindo as definições de *Client* e *Server* e executando as comunicações entre os processos, teremos:

$$(v\, r_A, p_B)((vl)Req(r_A, l) \mid (vs)Prov(p_B, s) \mid Bind(r_A, p_B))$$




```
component Server {
   provide p;
}

component Client {
   require r;
}

component System {
   inst
    A:Client;
    B:Server;
   bind
    A.r - B.p
}
```

Figura 6.3: Sistema Cliente-Servidor.

$$\rightarrow (vp_B)((vl)\overline{p_B}l \mid (vs)Prov(p_B, s))$$
$$\rightarrow (vl, s, p_B)(\bar{l}s \mid Prov(p_B, s))$$

Podemos observar que a expressão descrevendo o sistema se reduz a uma expressão que envia o serviço s para a localização requerida l em paralelo com o agente $Prov$ a com os agentes $Server'$ e $Client'$. Antes do cliente utilizar o serviço ele deve realizar uma ação de entrada. Uma possível definição para $Client'$ pode ser:

$$Client'(l) \stackrel{def}{=} l(x).Client''$$

O sistema: $(vl, s, p_B)(\bar{l}s \mid Prov(p_B, s) \mid Client'(l) \mid Server'(s))$
se reduz a: $\rightarrow (vs, p_B)Prov(p_B, s) \mid Client''(s/x) \mid Server'(s)$

Que é o resultado desejado de uma instância de um componente servidor executando em paralelo com um componente cliente, na qual todo ocorrência do nome local x foi trocada pelo nome s, que é a referência do serviço requerido.

Discussão

A representação de sistemas reconfiguráveis em π-cálculo ilustrada anteriormente através da linguagem Darwin, apresenta algumas restrições relativas a legibilidade, independência de contexto e reconfigurabilidade.

- **Legibilidade**
  O termo legibilidade é usado aqui não apenas para tratar de uma boa representação da

especificação, mas também com relação à materialização da estrutura da configuração na sua especificação formal. Como é notório, um dos mais importantes objetivos da formalização de sistemas é a possibilidade de realizar análises nas configurações antes dessas serem implementadas. Contudo, se for observado apenas a especificação π-cálculo relativa a uma configuração, não conseguimos nenhuma informação estrutural a respeito da configuração. Só é possível observar uma porção de agentes sem nenhuma informação estrutural. Esse problema é causado pela falta de operadores sintáticos em π-cálculo para representar a estrutura de configuração na especificação. Por exemplo, não se pode representa um componente em π-cálculo ao invés disso pode-se apenas representar um conjunto de portas através de agentes. Em outras palavras, em Darwin, é necessário se utilizar três agentes para representar cada conexão de porta.

- **Independência de Contexto**
  Em arquiteturas de softwares distribuídas os componentes devem ser implementados e testados independentemente do resto do sistema que eles fazem parte. Esses componentes devem ser passíveis de serem reutilizados em diferentes contextos sem que haja a necessidade de se realizar modificações em suas operações internas. Nesse sentido podemos observar que a especificação π-cálculo de agentes que representam componentes Darwin não podem compartilhar dessa mesma característica. Ou seja, a especificação de um componente em π-cálculo não pode ser reusada da mesma forma que o próprio componente. Isso se deve ao fato de que para se realizara comunicação entre dois processos π-cálculo esses devem sincronizar as ações em canais com nomes iguais e sentidos diferentes. Em outras palavras, eles devem compartilhar nomes de canais entre cada porta conectada. Essa característica de π-cálculo limita a representação de agentes independentes de contexto.

- **Reconfigurabilidade**
  A mais importante operação em sistemas baseados em conexões arquiteturais é a possibilidade de realizar a modificação da estrutura da aplicação apenas se manipulando as conexões entre os componentes. Em nível de configuração basta apenas modificar as conexões entre as portas, contudo essa mesma característica não pode ser imitada em nível de especificação formal. Quando modificamos as conexões entre portas em uma configuração, devemos modificar a especificação interna de todos os agentes π-cálculo relacionados com essa conexão. Isso se dá pelo fato de π-cálculo não ser equipado com operadores sintáticos que permita a manipulação dos padrões de sincronização entre canais.

6.3.3.2 O Cálculo $\mathcal{R}\pi$

De forma a dar suporte a reconfigurabilidade em especificação de componentes utilizando π-cálculo criamos uma álgebra de processos denominada $\mathcal{R}\pi$. Utilizando $\mathcal{R}\pi$, tratamos às restrições impostas por π-cálculo em permitir sincronização de ações entre processos através dos nomes das portas, que devem ser iguais e em sentidos opostos para que uma comunicação venha a ocorrer.

Essa imposição de π-cálculo cria restrições quanto a reusabilidade de especificações de componentes em especificações de outras aplicações. Essas restrições podem ser tratadas através de mecanismos de definição de agentes em π-cálculo [87], entretanto acabamos por trabalhar na estrutura de definição dos agentes e não apenas nas relações entre os canais, que é o que deveria ser feito para termos uma especificação reconfigurável [72].

Para exemplificar essa restrição observemos a especificação a seguir.

$$a(x).P \mid \overline{a}y.Q \xrightarrow{\tau} P\{y/x\} \mid Q$$

A comunicação entre os canais a e $\overline{a}$ só é possível pelo fato de que esses canais representam uma ação e uma co-ação em um mesmo nome, isso é, a sincronização dos dois depende de uma relação de igualdade entre os nomes dos canais.

Se agora supusermos que os nomes dos canais são a e b, teremos o seguinte sistema:

$$a(x).P \mid \overline{b}y.Q$$

Na verdade, esse sistema está em impasse, visto que não há nenhuma transição possível de ser realizada se considerarmos a semântica de π-cálculo. Pois não poderá haver sincronização entre os canais dos processos P e Q.

Em $\mathcal{R}\pi$ propomos o relaxamento dessa disciplina de sincronização de modo a permitir a comunicação de processos em diferentes canais, ou seja, nós não impomos a disciplina de sincronização de $\mathcal{R}\pi$ baseada em uma relação de igualdade entre nome, ao invés disso usamos uma relação de equivalência entre os nomes, denominamos essa relação de *correlação*, na qual dois canais correlatos são considerados iguais. Por exemplo:

$$a(x).P \mid \overline{b}y.Q \xrightarrow{[a \diamond b]} P\{y/x\} \mid Q$$

A transição anterior define que a e b são correlatos (representado pelo símbolo $\diamond$), ou seja, eles podem sincronizar suas ações. Dessa forma, o comportamento do sistema é idêntico ao de π-cálculo, com a vantagem que podemos criar processos completamente independentes de contexto, no qual a disciplina de sincronização é explicitamente definida.

Uma restrição clara da nossa abordagem é a vulnerabilidade na colisão de nomes, na qual podemos produzir um sistema utilizando processos com canais com o mesmos nomes, de forma a não sermos capazes de identificar qual nome estamos referenciado em uma correlação. Para exemplificar, consideremos a seguinte especificação, onde o processo R é reusado de outra especificação.

$$[a \diamond b](a(x).P \mid \overline{b}y.Q \mid a(w).c(t).R)$$

Nesse sistema existe um conflito com relação à interação que deve ocorrer. Considerando a correlação $a \diamond b$, podemos ter comunicação entre P e Q, através dos canais a e b, ou entre R e Q, através dos canais a e b. Para lidar com essa característica, formulamos a idéia de modularização de especificações de forma a sermos capazes de separar porções de especificações reusáveis. Nesse sentido, inserimos a noção de componentes em $\mathcal{R}\pi$. Através de componentes, que são as menores porções de especificações reusáveis em $\mathcal{R}\pi$, podemos criar especificações independentes de contexto, que podem facilmente ser relacionadas através de correlações.Na especificação a seguir temos a representação em forma de componentes da especificação anterior:

$$\begin{array}{l} C_1[a(x).P] \mid C_2[\overline{b}y.Q] \mid C_3[a(w).c(t).R] \xrightarrow{[C_1.a \diamond C_2.b]} \\ C_1[P\{y/x\}] \mid C_2[Q] \mid C_3[a(w).c(t).R] \end{array}$$

Agora podemos identificar os componentes C_1, C_2, e C_3. Cada componente encapsula processos cujos nomes de canais são inteiramente independentes de contexto. A correlação $[C_1.a \diamond C_2.b]$ relaciona os canais a do componente C_1 e o canal b do componente C_2.

Pelo relaxamento da disciplina de sincronização através de correlações nos nomes de canais e inserindo o conceito de componentes nas especificações formais, o cálculo $\mathcal{R}\pi$ se apresenta como uma solução mais adequada para a modelagem de arquiteturas de software baseadas em componentes reusáveis. Apresentamos a seguir a sintaxe e a semântica de $\mathcal{R}\pi$.

Sintaxe

Como uma extensão de π-cálculo, a sintaxe de $\mathcal{R}\pi$ é definida a partir de π-cálculo através da adição de novos conceitos. Tais novos conceitos estão relacionados à noção de reconfigurabilidade, que é o principal objetivo de $\mathcal{R}\pi$. Inicialmente, o conceito de componente deve ser definido. Um componente é um container independente de contexto para processos π. Cada componente pode ser composto por um ou mais processos π e representam a menor unidade de reuso e de reconfigurabilidade em $\mathcal{R}\pi$.

Como mostrado anteriormente nos exemplos, a comunicação entre componentes em $\mathcal{R}\pi$ é controlada pela sincronização de processos de diferentes componentes em canais relacionados, no qual essa relação, que denominamos de correlação, leva em conta os nomes dos componentes e os nomes dos canais.

Como em π, $\mathcal{R}\pi$ possui um conjunto infinito $\mathcal{N}$ de *nomes* representado por $a, b, c, \ldots, x, y, z$. Seja τ um elemento distinto tal que $\mathcal{N} \cap \{\tau\} = \emptyset$. Esses nomes representam as portas de comunicação.

Seja $\mathcal{C}$ o conjunto de todos os nomes de componentes, representados por C_1, C_2, C_3. E seja $\mathcal{P}$ um conjunto tal que $\mathcal{P} = \mathcal{C} \times \mathcal{N}$, cujos elementos são representados por p_1, p_2, p_3. Por conveniência, escrevemos $C_1.a$ para representar o elemento $p = (C_1, a)$.

Também definimos ψ como um conjunto de todas as relações de equivalência sobre $\mathcal{P}$. Escrevemos $[p_1 \diamond p_2]$ para representar a menor relação de equivalência relacionando p_1 e p_2, onde $p_1, p_2 \in \mathcal{P}$. Também por conveniência, usamos $\mathbf{1}$ para descrever a relação reflexiva. Como consequência, a relação $[p_1 \diamond p_1]$ é a mesma que $[p_2 \diamond p_2]$.

Definição 1 ***(Ações)***
As ações de $\mathcal{R}\pi$ são de entrada e de saída, representadas por γ e a correlação representada por ψ, sendo definidas pela seguinte gramática:

$$\begin{array}{rcll} \gamma & ::= & C_1.a(x) & (\text{Entrada}) \\ & / & C_2.\bar{a}y & (\text{Saída}) \\ \psi & ::= & [p_1 \diamond p_2] & (\text{Correlação}) \end{array}$$

Seja $\mathcal{A}$ o conjunto de todas as ações. Usamos α como uma meta variável para representar essas ações.

1. $a(x)$ é chamada de *prefixo de entrada.* Um nome a é tratado como uma porta de entrada de um processo que o contém.

2. $\overline{a}y$ é chamado de *prefixo de saída.* Uma nome a é tratado como uma porta de saída de um processo que o contenha.

3. $[p_1 \diamond p_2]$ é chamada *correlação.* Um elemento p_1 está correlacionado a um outro elemento p_2, em outras palavras, supondo $p_1 = C_1.a$ e $p_2 = C_2.b$, $[p_1 \diamond p_2]$ é uma relação representada pelo seguinte conjunto: $\{(C_1.a, C_2.b), (C_2.b, C_1.a)\} \cup 1$, onde 1 representa a relação de identidade.

Definição 2 ***(Agentes)***
Os agentes, representados por $(P, Q, R, \ldots)$, são definidos por:

$$\begin{array}{rcll} P & ::= & \emptyset & \textit{(Inação)} \\ & | & \alpha.P & \textit{(Prefixo)} \\ & | & Q + R & \textit{(Soma)} \\ & | & Q \mid R & \textit{(Composição)} \\ & | & (\upsilon x)Q & \textit{(Restrição)} \\ & | & A(x) & \textit{(Identificação)} \end{array}$$

Os agentes (ou processos) são encapsulados encapsulated em componentes e têm o mesmo comportamento do agentes definidos em π-cálculo.

Definição 3 ***(Componentes)***
Os componentes, representados por C, C_1, C_2, são definidos por

$$\begin{array}{rcll} C & ::= & \emptyset & (Componente\ Vazio) \\ & | & \psi.C_1[P] & (Componente\ Prefixado) \\ & | & C_1 \mid C_2 & (Composicão\ de\ Componentes) \end{array}$$

Componentes são *containers* para agentes π e a comunicação entre componentes é controlada através de correlações e todas as comunicações que ocorrem dentro de um componente, isto é, entre os processos que formam um componente, são reguladas pelas regras usuais de π-cálculo incluindo congruência estruturas e regras de redução.

Nós definimos nomes livre (fn) e nomes ligados (bn) em componentes da seguinte forma:

$$\begin{array}{lcl} fn(C[\emptyset]) & \xrightarrow{\text{def}} & \emptyset \\ fn(C[P]) & \xrightarrow{\text{def}} & fn(P) \\ fn(C[P \mid Q]) & \xrightarrow{\text{def}} & fn(P) \cup fn(Q) \\ fn(C[P + Q]) & \xrightarrow{\text{def}} & fn(P) \cup fn(Q) \end{array}$$

Definição 4 ***(Substituição)***
Uma substituição é uma mapeamento σ de $\mathcal{N}$ para $\mathcal{N}$. Suponha $\sigma(x_i) = y_i$, para todo $1 \leq i \leq n$ e $\sigma(x) = x$ para todos os outros nomes x, escrevemos $\{y_1/x_1, \ldots, y_n/x_n\}$. $C_1[P]\sigma$ para denotar a expressão obtida de $C_1[P]$ pela substituição simultânea $\sigma(x)$ de cada ocorrência de x em P por cada x, com mudança de nomes ligados quando necessário.

A congruência estrutural iguala agentes que não são distinguíveis por nenhuma razão semântica.

Definição 5 ***(Congruência Estrutural)***
A congruência estrutural, representada por $\equiv$*, é a menor relação de congruência satisfazendo as seguinte leis:*

1. $C_1[a(x).\overline{b}x] \equiv C_2[a(y).\overline{b}y]$
   $C_1[a(x).\overline{b}x] \equiv C_2[c(x).\overline{d}x]\ if\ [C_1.a \diamond C_2.c, C_1.b \diamond C_2.d]$
2. $C_1 \mid 0 \equiv C_1$
   $C_1 \mid C_2 \equiv C_2 \mid C_1$
   $C_1 \mid (C_2 \mid C_3) \equiv (C_1 \mid C_2) \mid C_3$
3. $C_1[P] \equiv C_2[Q] \Leftrightarrow P \equiv Q$
4. $C_1[P \mid Q] \equiv C_1[P] \mid C_1[Q]$
5. $C_1[P + Q] \equiv C_1[P] + C_1[Q]$
6. $C[0] \equiv 0$

Apresentamos somente as regras relacionadas a componentes, todas as leis relacionadas a processos que estão encapsulados dentre dos componentes são derivadas diretamente de π-cálculo. Nessas regras definimos os componentes que satisfazem as leis do monóide abeliano para a soma e para a composição. Além disso, na regra 1, definimos uma noção estendida de conversão-alfa para componentes. Nessa regra, podemos observar que componentes produzidos a partir de agentes variantes por conversão-alfa, isto é, pela escolha de nomes ligados, são considerados os mesmos, e componentes cujos whose nomes livres estão correlacionados e cujos nomes ligados são iguais também são considerados os mesmos.

Semântica Operacional

A forma padrão de definir a semântica operacional para uma álgebra de processos é através de um LTS (*Labelled Transition System*). A semântica operacional para $\mathcal{R}\pi$ é definida a seguir.

STR	$\dfrac{C_1[P'] \equiv C_1[P], C_1[P] \xrightarrow{C_1.\alpha} C_1[Q], C_1[Q] \equiv C_1[Q']}{C_1[P'] \xrightarrow{C_1.\alpha} C_1[Q']}$	(*Estrutura*)
PREF	$\dfrac{-}{C_1[\alpha.P] \xrightarrow{C_1.\alpha} C_1[P]}$	(*Prefixo*)
SUM	$\dfrac{C_1[P] \xrightarrow{C_1.\alpha} C_1[P']}{C_1[P] + C_2[Q] \xrightarrow{C_1.\alpha} C_1[P']}$	(*Soma*)
PAR	$\dfrac{C_1[P] \xrightarrow{C_1.\alpha} C_1[P'], \mathtt{fn}(\alpha) \cap \mathtt{fn}\ (Q) = \emptyset}{C_1[P] \mid C_2[Q] \xrightarrow{\alpha} C_1[P'] \mid C_2[Q]}$	(*Composição*)
COMM	$\dfrac{C_1[P] \xrightarrow{C_1.a(x)} C_1[P'],\ C_2[Q] \xrightarrow{C_2.\overline{b}y} C_2[Q'], C_1.a \diamond C_2.b}{C_1[P] \mid C_2[Q] \xrightarrow{C_1.a \diamond C_2.b} C_1[P'\{y/x\}] \mid C_2[Q']}$	(*Comunicação*)

Tabela 6.1: Semântica Operacional de $\mathcal{R}\pi$.

Código 6.10 Especificação π-cálculo do GPD.

$$
\begin{aligned}
RECEPTIONIST(next, talk) &\stackrel{def}{=} talk(n, ill, reply).\\
&\quad next(answer).\overline{answer}(n, ill, reply).\\
&\quad RECEPTIONIST(next, talk)\\
\\
DOCTOR(next) &\stackrel{def}{=} (\upsilon\, answer)\\
&\quad \overline{next}(answer).answer(n, ill, reply).\\
&\quad \overline{reply}(diagnose(ill)).\\
&\quad DOCTOR(next)\\
\\
PATIENT(n, talk) &\stackrel{def}{=} \upsilon\, reply)\\
&\quad \overline{talk}(n, rand(), reply).\\
&\quad reply(prescription).0\\
\\
SYSTEM &\stackrel{def}{=} RECEPTIONIST(next, talk) \mid\\
&\quad DOCTOR(next) \mid PATIENT(n, talk)
\end{aligned}
$$

Definição 6 A família de transições $\xrightarrow{\alpha}$ ($\subset \mathcal{C} \times \mathcal{C}$), para $\alpha \in \mathcal{A}$, é a menor família que satisfaz as leis da Tabela 6.1.

A semântica de $\mathcal{R}\pi$ define a comunicação entre componentes. Todas as interações são relacionadas a componentes, na qual as ações de sincronização são definidas por meio de correlações, na qual pares de elementos indicam quais nomes são equivalentes. A sincronização de ações em processos que forma esses componentes são reguladas pela semântica operacional usual de π-cálculo.

Basicamente, as regras apresentadas na Tabela 6.1 têm a mesma estrutura das correspondentes em π só que aplicadas à componentes. Nessas regras nós apresentamos o sistema de transições baseado em nomes compostos, representados pelo nome do componente mais o nome do canal além de uma correlação entre esses nomes compostos.

A regra mais expressiva de $\mathcal{R}\pi$ é a regra [COMM], que representa a sincronização de componentes em diferentes nomes de canais relacionados por uma correlação. Com essa regra podemos construir componentes independentes de contexto e sistemas reconfiguráveis, no qual a disciplina de sincronização dos componentes é modificada na codificação do sistema.

Exemplo

Para ilustrar a aplicação de $\mathcal{R}\pi$ na especificação de sistemas reconfiguráveis, usaremos a aplicação *Group Practice of Doctor* (GPD), adaptada de [73]. Essa aplicação tem três tipos de processos: *receptionist*, *patient* and *doctor*. A semântica pretendida é a seguinte. O paciente (*patient*) chega ao consultório e fala seu nome e sua doença à recepcionista (*receptionist*). Num outro lado, os médicos (*doctors*) que estão livres para receber pacientes informam à recepcionista de sua disponibilidade. A recepcionista informa as médico disponível o nome e a doença do paciente. Ao paciente é dada uma receita pelo médico. Depois disso, o paciente deixa o consultório e o médico fica novamente livre para receber outro paciente. O Código 6.10 apresenta a modelagem π-cálculo dessa aplicação [35].

Para verificar formalmente as propriedades desses processos temos que compor um agente com as definições dos processos utilizando o operador de paralelismo de π-cálculo. O sistema resultante é o seguinte:

$$RECEPTIONIST(next, talk) \mid DOCTOR(next) \\ \mid PATIENT(n, talk)$$

$$talk(n, ill, reply).next(answer).\overline{answer}(n, ill, reply) \mid \\ (\upsilon\ answer)\,\overline{next}(answer).answer(n, ill, reply).\overline{reply}(diagnose(ill)) \mid \\ (\upsilon\ reply)\,\overline{talk}(n, rand(), reply).reply(prescription).0$$

Podemos observar que a especificação dos componentes em π-cálculo possui uma memória global do sistema, isso é, seus canais são construídos baseado nos nomes dos canais de outros processos que forma o sistema. Sendo que a modificação no nome dos canais não pode ser manipulada sem a redefinição de agentes. Dessa forma, não podemos construir especificações formais independentes de contexto, visto que π-cálculo assim como outras álgebras de processos, seguem a disciplina restritiva de sincronização baseada na igualdade entre nomes de canais. Usando $\mathcal{R}\pi$ para relaxar essa propriedade nós podemos resolver esse problema e construir especificações formais de componentes realmente independentes de contexto. Para exemplificar, suponha que tenhamos obtido a seguinte especificação do componente $PATIENT$ de um outro sistema.

$$(\upsilon\ response)\,\overline{ask}(n, rand(), response).response(receipt).0$$

Seguindo a idéia de reconfigurabilidade, se o componente $PATIENT$ foi construído como um componente independente de contexto, isto é, nós não devemos nos preocupar com o ambiente externo para construir a especificação. Nesse caso, os nomes dos canais são independentes de qualquer contexto. Na especificação da arquitetura, para reconfigurar o sistema anterior utilizando esse componente, temos apenas que conectar as portas apropriadas. Contudo, esse fato não pode ser imitado na especificação formal do sistema, isso é, usando π-cálculo nós não podemos reconfigurar o sistema. Contudo, usando $\mathcal{R}\pi$ nós podemos construir a seguinte especificação:

$$[C_1.talk \diamond C_3.ask, C_1.next \diamond C_2.next, C_1.answer \diamond C_2.answer, \\ C_2.reply \diamond C_3.response] \\ (\ C_1[talk(n, ill, reply).next(answer).\overline{answer}(n, ill, reply)]\ \mid \\ C_2[(\upsilon\ answer)\,\overline{next}(answer).answer(n, ill, reply). \\ \overline{reply}(diagnose(ill))]\ \mid \\ C_3[(\upsilon\ response)\,\overline{ask}(n, rand(), response).response(receipt).0]\ )$$

Na especificação $\mathcal{R}\pi$, todas as (re)configurações são realizadas através das correlações. Nesse sentido, os nomes dos canais são conectados através da especificação tornando as especificações dos componentes independentes de contexto, o que torna $\mathcal{R}\pi$ um cálculo que captura a noção de reconfigurabilidade também em nível de especificação formal. Além disso, podemos perceber claramente a estrutura da aplicação materializada na especificação formal, isso é, nós podemos identificar facilmente os componentes e suas conexões, o que não é tão fácil de realizar em outras álgebras de processos.

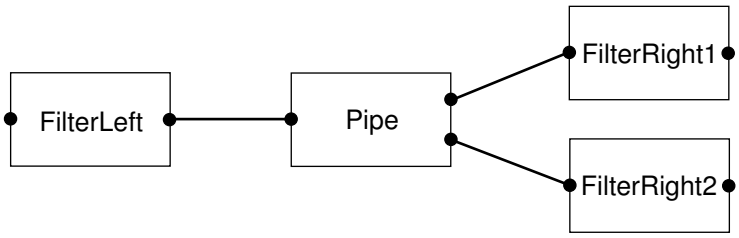


Figura 6.4: Diagrama do Sistema Exemplo.

A Semântica de ArchML

A semântica de um sistema especificado em ArchML é obtida pela composição em paralelo da semântica das instâncias dos componentes que formam o sistema. Para exemplificar, seja arquitetura, apresentada na Figura 6.4, que representa um sistema simples do tipo *Pipe-Filter*, formado por três componentes do tipo *Filter*: `FilterLeft`, `FilterRight1`, e `FilterRight2` e um componente do tipo *Pipe*: `Pipe`. Nesse exemplo, os filtros possuem um buffer de duas posições, sendo que um stream de caracteres é lido através de `FilterLeft`, que envia esses dados para `Pipe`, que por sua vez os repassa para um dos filtros seguintes de acordo com a disponibilidade em seus buffers.

Nesse exemplo, a semântica do sistema `PipeFilter` é definida em termos da semântica de suas instâncias `FilterLeft`, `FilterRightUp` e `FilterRightDown`.

$$\begin{aligned}[\![PipeFilter]\!] = {} & [\![FilterLeft]\!] \mid \\ & [\![FilterRightUp]\!] \mid \\ & [\![FilterRightDown]\!] \mid \\ & [\![P]\!]\end{aligned}$$

A semântica de cada componente por sua vez, é definida através da composição em paralelo dos valores dos atributos `value` de cada elemento na descrição ArchML do componente. Para isso, definimos a função $v(CompX)$ para retornar a descrição do comportamento de um componente.

$$\begin{aligned}[\![Filter]\!] &= v(Filter) \\ [\![Pipe]\!] &= v(Pipe)\end{aligned}$$

Do mesmo modo, a semântica das instâncias de um componente é definida pela semântica de seu tipo correspondente.

$$\begin{aligned}[\![FilterLeft]\!] &= [\![Filter]\!] \\ [\![FilterRightUp]\!] &= [\![Filter]\!] \\ [\![FilterRightDown]\!] &= [\![Filter]\!] \\ [\![P]\!] &= [\![Pipe]\!]\end{aligned}$$

Usando a definição estendida de cada agente, podemos observar que é necessário realizar mudança nos nomes das portas conectadas em cada componente de forma a permitir a

interação dos agentes em nível de especificação formal. Como exemplo, `FilterLeft` e `Pipe` devem interagir via as portas `Input` and `Output`, que possuem nomes diferentes. Visto que a semântica de transição do operador de paralelismo de π-cálculo só permite a sincronização em ações em canais com o mesmo nome, devemos renomear as portas conectadas em cada componente utilizando um nome comum.

6.4 Conclusão

Nesse capítulo tratamos da fase de validação e verificação sintática e comportamental de arquiteturas e estilos de DraX. Como validação sintática apresentamos as gramáticas formais de ArchML e de Xtyle. Essas gramáticas são aplicações de XML Schema e permitem se garantir a validade sintática das especificações de arquiteturas e de estilos arquiteturais em DraX.

Contudo, pela forma simplificada que adotamos na construção de ArchML e Xtyle, onde optamos por não usar estruturas de XML que viessem a tornar as especificações menos legíveis, tivemos que produzir outros níveis de validação para garantir a correção das especificações. Dessa forma, lançamos mão de um conjunto de esquemas baseados em Schematron para realizar essas avaliações. Além de verificar aspectos de integridade referências nos nomes utilizados para identificar componentes, instâncias, portas entre outros, utilizamos esquemas Schematron para realizar a tarefa de verificação sintática de correção de especificações com relação à estrutura de conexões entre instâncias de componentes, entre os tipos das portas, entre os parâmetros trocados na comunicação, entre outros fatores.

Além disso, utilizamos esquemas Schematron para verificar se uma especificação ArchML, que seguem um estilo definido em Xtyle, de fato concorda com as regras sintáticas e topológicas impostas pelo estilo. Assim, são verificados os `roles` das interfaces dos componentes com relação aos tipos definidos no estilo, o fluxo de dados e controle imposto pelo estilo, entre outros testes.

Para terminar, pelo fato de Xtyle permitir a descrição de estilos Definidos-Pelo-Usuário, no qual não existe esquemas Schematron desenvolvidos a priori para realizar a validação tanto do estilo como das arquiteturas que o utilizam, criamos um *script* XSLT que pode gerar as regras Schematron para validar o estilo. Dessa forma, tornamos a fase de verificação e validação sintática de DraX realmente flexível e compatível com as idéias principais dessa tese de tornar realmente fácil o trabalho de desenvolvimento baseado em arquitetura e de fornecer um ambiente didático para o aprendizado de arquitetura de software.

Com relação à realização de validação comportamental de DraX, nesse capítulo desenvolvimos um conjunto de *scripts* de geração de especificações algébricas em π-cálculo a partir das especificações de comportamento realizado pelos DDPs de DraX.

Nesse capítulo apresentamos a forma de como realizar as verificações comportamentais em DraX a partir dessas especificações. É importante ressaltar que em momento algum manipulamos diretamente as especificações π-cálculo geradas, a penas as utilizamos para realizar as verificações tanto de arquiteturas como de estilos.

Ao longo do processo de desenvolvimento de DraX com relação à realização de análises formais, verificamos algumas restrições de π-cálculo para permitir a especificações das arquiteturas. Nesse contexto, resolvemos propor uma nova álgebra de processos, o cálculo $\mathcal{R}\pi$. Esse cálculo é utilizado nessa tese para fornecer a semântica formal de ArchML. Contudo, como não produzimos uma ferramenta de manipulação de especificações $\mathcal{R}\pi$, nesse trabalho não utilizamos esse cálculo de forma automática, sendo que utilizamos apenas π-cálculo, visto

que já existe uma ferramenta para esse fim (MWB) . Contudo, é uma trabalho futuro a implementação de uma ferramenta para a manipulação de especificações $\mathcal{R}\pi$. Uma outra restrição que encontramos no momento também diz respeito ao MWB, que trabalha apenas com especificações síncronas. De fato, não existe uma implementação do MWB que trabalhe com π-cálculo assíncrono.

Capítulo 7

Geração de Código

7.1 Introdução

Nesse capítulo tratamos a última etapa de desenvolvimento de aplicações distribuídas utilizando os conceitos de arquitetura de software. Aqui apresentaremos um conjunto *scripts* de geração de templates de código com todas as informações de execução sobre um middleware baseado nas informações arquiteturais produzidas pelas ferramentas de DraX.

Inicialmente devemos observar que é possível utilizar a idéia de templates de implementação pelo fato de que podemos identificar em todas as aplicações distribuídas baseadas em algum middleware, como CORBA ou RMI, uma estrutura fixa sempre seguida. Códigos relativos, por exemplo, a inicialização do ORB, inicialização e configuração do POA, invocação de métodos em objetos remotos, cast de tipos RMI, entre outros, estão sempre presentes em aplicações distribuídas. Além disso, também podemos encontrar porções fixas de código relativas aos serviços disponibilizados pelos middleware. Assim, se utilizarmos CORBA, por exemplo, temos a recuperação da referência do servidor de nomes, a criação de um nome ou contexto, a resolução de um nome, a criação de um canal de eventos, são exemplo típicos de códigos que possuem a mesma estrutura e que estão repetidos em todas as aplicações que utilizam os serviços de nomes e eventos de CORBA. Como também, se utilizarmos RMI, temos a manipulação do espaço de nomes, a inicialização do sistema de segurança RMI, entre outros, também tarefas constantes nas implementações Na verdade o que podemos observar é que a organização desses códigos é que pode ser diferenciada, enquanto que seus conteúdos basicamente são os mesmos.

Utilizando as informações da arquitetura e dos estilos que essas utilizam, além das propriedades específicas dos tipos de componentes que formam essas arquiteturas, podemos facilmente gerar templates Java com códigos relativos ao middlware escolhido de forma automática. O principal objetivo dessa abordagem é livrar o desenvolvedor de ter que conhecer a fundo o middlware que está sendo utilizado para implementar suas aplicações descritas usando as linguagens de DraX. Dessa forma, o desenvolvedor descreve a arquitetura e os componentes em ArchML e o estilo em Xtyle. Depois dessas especificações serem devidamente validadas e verificadas, um conjunto de *scripts* geram templates Java para essa arquitetura. Nesse capítulo apresentamos a forma pela qual esses templates são gerados em DraX.

Para facilitar e direcionar as especificações desse capítulo, daremos ênfase aqui a geração de códigos para infraestruturas de middleware orientados a objetos. Assim, utilizaremos para exemplificar os códigos desse capítulo, os padõres Java/CORBA, por serem esses padrões largamente aceitos para o desenvolvimento de aplicações distribuídas. Contudo, na Seção

7.7, discutiremos a criação/adaptação de *scripts* para a geração de templates em outras infraestruturas de middleware orientados a objetos.

7.2 Mecanismo de Geração de Templates

De forma a permitir a geração de códigos com todas as informações relativas a arquitetura e aos serviços do middleware que esses devam utilizar para implementar as características dos estilos arquiteturais, um conjunto de *scripts* XSL geram, a partir da especificação das arquiteturas, dos componentes e do estilo, um conjunto de templates que devem ser utilizados como base para implementar à lógica da aplicação. Esses templates tiram do desenvolvedor a tarefa árdua de ter que entender os serviços disponibilizados pelo middleware e toda a sintaxe da plataforma, ficando esse exclusivamente com a implementação da parte funcional da aplicação.

Nossa abordagem inicial para a geração de templates de implementação foi realizar a geração de cada template usando apenas um *script*. Essa abordagem foi abandonada quando observamos a complexidade desse *script* e na clara mistura de informações de implementação, tais como, códigos de serviços, códigos de invocação, etc. Dessa forma, optamos por utilizar um conjunto de *scripts* com finalidades específicas, onde cada um dos *scripts* gera um template com alguns elementos XML que devem ser tratados por outros *scripts* específicos. Dessa forma, obtivemos *scripts* mais simples, e portanto mais fáceis de manter e adaptar a outras infraestruturas de middleware, além de separar de forma clara as tarefas específicas de implementação.

Para a geração completa de um template de implementação em Java com códigos de comunicação CORBA, propomos os seguintes *scripts*:

1. ***Script* de Geração de IDL**
   Esse *script* é utilizado para gerar as interfaces em CORBA IDL dos componentes que forma a arquitetura.

2. ***Script* de Geração de Código de Esqueleto de Classe**
   Para cada tipo de componente da arquitetura, esse *script* gera um arquivo XML que é um esqueleto Java com um conjunto de elementos XML marcando a localização no código onde os códigos de serviços e de invocações de objetos externos devem ser inseridos.

3. ***Script* de Geração de Códigos de Mecanismos de Invocação CORBA**
   Esse *script* utiliza o resultado do script anterior e substitui os elementos relativos à invocação, pelos códigos Java/CORBA devidos. Esses códigos respeitam às características do estilo que arquitetura utiliza, onde podem ser geradas comunicações síncronas ou assíncronas.

4. ***Script* de Geração de Códigos de Serviços CORBA**
   Os códigos dos serviços CORBA, como nomes e eventos, são inseridos através desse *script*. Também consideramos o POA como um serviço e os códigos relativos ao POA também são tratados nesse *script*.

O fluxo de aplicação dos *scripts* de geração de templates a ser adotado para a criação de uma aplicação deve ser o seguinte:

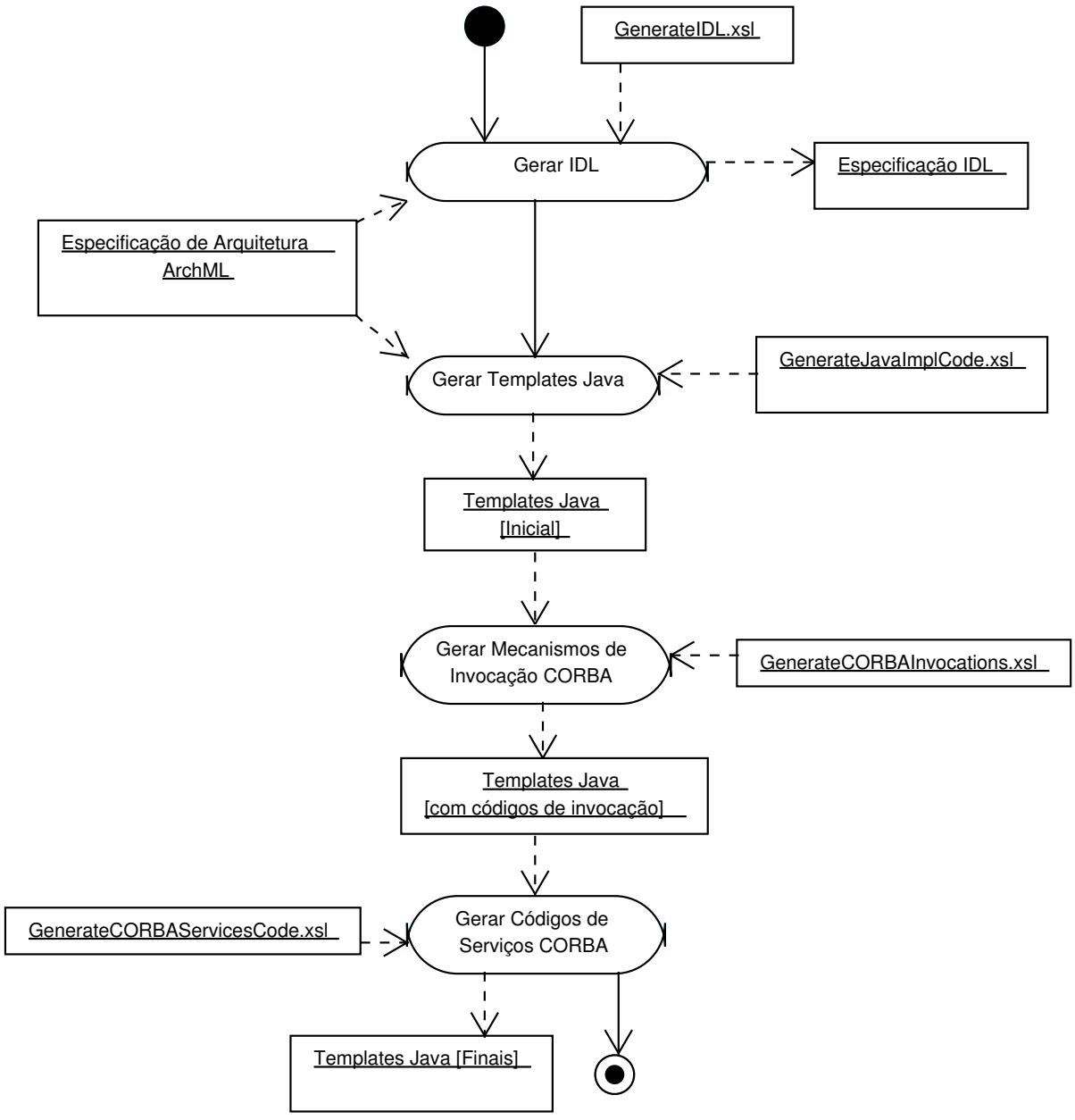


Figura 7.1: Atividades com *os scripts* de Geração de Código.

7.3 Geração de IDL

A geração de interfaces IDL para os sistemas é realizado por um *script* XSL (`GenerateIDL.xsl`). Esse *script* utiliza a especificação da arquitetura em ArchML. Para cada tipo encontrado na especificação ArchML é gerada uma interface diferente. Sendo que o conjunto dessas interfaces é colocada dentro de um `module`, cujo nome é o mesmo do atributo `name` da especificação arquitetural em ArchML.

A seguir apresentamos o trecho de código que gera o módulo IDL.

```
<xsl:template match="system">
   module <xsl:value-of select="@name"/>{
      <xsl:apply-templates select="types"/>
   };
</xsl:template>
```

O rastreamento dos tipos para a geração das interfaces é realizado pelo trecho de código a seguir:

```
<xsl:template match="types">
 <xsl:for-each select="type">
  <xsl:if test="not(count(document(@href)//port[@direction='in'])=0)">
  interface <xsl:value-of select="document(@href)//component/@name"/>Itf{
   <xsl:for-each select="document(@href)//port[@direction='in']»
    <xsl:call-template name="XSD2IDL">
     <xsl:with-param name="xsdtype"select="substring-after(@return,':')"/>
    </xsl:call-template>
   <xsl:value-of select="string('')"/>
   <xsl:value-of select="@name"/>(
   <xsl:for-each select="param">
   in
    <xsl:call-template name="XSD2IDL">
     <xsl:with-param name="xsdtype"select="substring-after(@type,':')"/>
    </xsl:call-template>
   <xsl:value-of select="string(' ')"/>
   <xsl:value-of select="@name"/>
  </xsl:for-each>
   );
  </xsl:for-each>
};
  </xsl:if>
 </xsl:for-each>
</xsl:template>
```

É interessante observar que somente as portas de entrada da arquitetura produzem métodos em IDL, sendo que para os parâmetros dessas portas/métodos há uma transformação dos tipos XSD (que representam os tipos ArchML) e os tipos IDL. Essa transformação é realizada por um template XSL que é invocada para cada parâmetro encontrado em ArhcML. A seguir apresentamos um trecho de código desse template.

XSD	IDL
xsd:boolean	boolean
xsd:byte	octet
xsd:string	string
xsd:short	Short
xsd:long	Long
xsd:float	Float
xsd:double	double

Tabela 7.1: Relação XSD x IDL.

```
<xsl:template name="XSD2IDL">
   <xsl:param name="xsdtype"/>
   <xsl:choose>
      <xsl:when test = "$xsdtype='boolean'">
         <xsl:text>boolean</xsl:text>
      </xsl:when>
      <xsl:when test = "$xsdtype='byte'">
         <xsl:text>octet</xsl:text>
      </xsl:when>
      <xsl:when test = "$xsdtype='string'">
         <xsl:text>string</xsl:text>
      </xsl:when>
      ...
   </xsl:choose>
</xsl:template>
```

A Tabela 7.1 apresenta as conversões que adotamos entre XSD e IDL. De fato convertemos apenas os tipos IDL que podem ser transformados a partir do tipo `Any` [51], visto que o *script* de geração de mecanismo de invocação de DraX, que será apresentado mais adiante nesse capítulo, utiliza somente invocação dinâmica de métodos, e, dessa forma, faz conversões entre tipos usando o tipo `Any` como padrão.

Para exemplificar seja o seguinte trecho de arquitetura em ArchML:

```
<?xml version="1.0"?>
<system name="Capitalize"
   <types>
      <type name="splitT"href="...Split.xml"/>
      <type name="upperT"href="...UpperCase.xml"/>
      <type name="lowerT"href="...LowerCase.xml"/>
      <type name="mergeT"href="...Merge.xml"/>
   </types>

   <instances>
      <instance name="splitInst"typeRef="splitT"/>
      <instance name="upperInst"typeRef="upperT"/>
      <instance name="lowerInst"typeRef="lowerT"/>
      <instance name="mergeInst"typeRef="mergeT"/>
   </instances>

   <links>
      <link name="conn1»
         <point instRef="splitInst"portRef="splitUp"/>
         <point instRef="upperInst"portRef="upperLeft"/>
      </link>
      <link name="conn2»
         <point instRef="splitInst"portRef="splitDown"/>
         <point instRef="lowerInst"portRef="lowerLeft"/>
      </link>
      <link name="conn3»
         <point instRef="upperInst"portRef="upperRight"/>
         <point instRef="mergeInst"portRef="mergeUp"/>
      </link>
      <link name="conn4»
         <point instRef="lowerInst"portRef="lowerRight"/>
         <point instRef="mergeInst"portRef="mergeDown"/>
      </link>
   </links>
</system>
```

O resultado da aplicação do *script* `GenerateIDL.xsl` sobre essa especificação é o seguinte:

```
module Capitalize{

   interface UpperCaseItf {
      void upperLeft (in string dataUpperLeft);
   };

   interface LowerCaseItf {
      void lowerLeft (in string dataLowerLeft);
   };

   interface MergeItf {
      void mergeUp (in string dataMergeUp);
      void mergeDown (in string dataMergeDown);
   };

};
```

O *script* `GenerateIDL` vasculha as especificações ArchML de cada componente e verifica os tipos de portas, sendo que apenas as portas de entrada são traduzidas em métodos IDL.

7.4 Geração de Códigos de Esqueleto de Classe

Como falamos anteriormente, para cada tipo de componente existente em uma arquitetura um esqueleto de classe Java é gerado. Esse esqueleto é um arquivo XML que conterá porções de código Java relativas à API do middleware. Além disso, serão utilizados um conjunto de elementos XML para indicar a localização dentro do código do template a ser gerado, onde devem ser inseridos códigos de serviços específicos. Esses elemento, devem fornecer informações suficientes para facilmente serem convertidos em códigos Java/CORBA.

Um aspecto importante na implementação de aplicações com CORBA, advém do fato de que os códigos de implementação podem variar de acordo com a função do objeto na arquitetura. Dessa forma, resolvemos gerar templates específicos para cada um dos tipos de funções de um objeto CORBA. Assim, esses templates são definidos para os seguintes tipos de objetos:

- **Objetos Apenas Clientes (ClientOnly)**
  Esses são objetos que não possuem métodos disponíveis para invocação através de CORBA. Ou seja, não possuem operações IDL definidas. Assim, esses objetos apenas utilizam os serviços de outros objetos.

- **Objetos Apenas Servidores (ServerOnly)**
  Os objetos Servidores apenas fornecem serviços a outros objetos, não utilizando nenhum serviço externo especificado em IDL.

- **Objetos Cliente-Servidores (ClientServer)**
  Os objetos Cliente-Servidores tanto implementam serviços disponíveis em IDL como utilizam serviços implementados por outros objetos.

Como explicado na introdução desse capítulo, um fator importante na implementação de aplicações CORBA é a forma com que estruturamos as classes. Duas as abordagens utilizadas. A primeira, utilizada em [89], é bastante difundida, onde temos, para casa interface IDL, um objeto que implementa essa interface e um outro objeto encarregado de instanciar o primeiro e realizar todas as operações de inicialização e controle do ORB. A segunda abordagem, utilizada em [OpenORB], usa uma mesma classe para realizar as duas tarefas.

Embora a primeira abordagem separe melhor os códigos da aplicação dos códigos relativos à CORBA, essa abordagem usa uma classe a mais para cada classe que implementa uma interface IDL, sendo que, em aplicações com muitas classes, temos uma grande confusão no momento da instanciação da aplicação. A segunda abordagem, por sua vez, peca por misturar os códigos CORBA com os códigos relativos à lógica da aplicação. Entretanto, essa abordagem usa exatamente uma classe para cada interface, o que facilita a estruturação da aplicação.

Como o objetivo de DraX e resguardar a implementação da parte de CORBA dentro dos códigos das aplicações, visto que esses códigos serão gerados automaticamente por *scripts*, resolvemos utilizar a segunda abordagem para organizar as classes das aplicações. Assim utilizaremos, basicamente a seguinte estrutura:

```
public class <nomeClasse>{                          (I)
   public static void main(String[] args ){         (II)
      new <nomeClasse>(args);
   }
   <nomeClasse>(String[] args){                      (III)
   }
   <método1>                                         (IV)
   <método2>
   ...
}
```

Podemos identificar as seguintes partes nesse código básico. Representado por **I**, temos a definição da classe em si. Em **II** vemos o método main, onde serão implementados/gerados os códigos relativos a instanciação do objeto no ORB CORBA e na resolução de serviços para esse objeto. Em **III** vemos o método construtor da classe. Nesse método serão implementados/gerados os códigos referentes a inicialização do objeto. Em **IV** temos os métodos da classe.

Como falamos no início dessa seção, o *script* de geração de templates gera um arquivo XML que possui alguns elementos XML relativos a serviços CORBA que devem ser tratados por outros *scripts* específicos. Esses elementos XML são inseridos nos arquivos gerados se levando em conta o tipo de objeto que vai ser gerado. Assim, por exemplo, um objeto Apenas Cliente, não precisa ser instanciado pelo POA, pois ele nunca será invocado via CORBA. Assim, o código básico apresentado anteriormente, deverá conter linhas de códigos que devem ser diferentes dependendo do tipo de objeto a ser gerado.

De forma geral o arquivo XML gerado pelo *script* de geração de templates de códigos de objetos tem o seguinte formato:

```
<?xml version="1.0"?>
<javaCODES style="nmtoken»
<javaCODE> *
   <service context="nmtoken" operation="nmtoken"
            name="name"?    value="nmtoken"?  />*
   <invocation portName="nmtoken"/>*
</javaCODE>
<links>
   <link> *
      <point1 comp="nmtoken " port="nmtoken»
         <param name="nmtoken" type="nmtoken"/> *
      </point1>
      <point2 comp="RemoveVowels.aml" port="getPhrase»
         <param name="nmtoken" type="nmtoken"/> *
      </point2>
   </link>
</links>
</javaCODES>
```

Esse arquivo, gerado pelo *script* de geração de templates de implementação, denominado (`GenerateJavaImplCode.xsl`), deverá conter partes de códigos Java relativas a cada objeto pertencente a arquitetura como texto dos elementos. Assim, dentro de cada elemento javaCODE, deve ser gerado o código do objeto Java relativo a um componente descrito

na especificação de arquitetura em ArchML. Usando o exemplo apresentado na Figura X. (UpperMerge), podemos verificar na Figura Y o formato geral gerado por esse *script* para o componente a arquitetura citada.

```
<?xml version="1.0"?>
<javaCODES style="Pipeline.sty">
<javaCODE>
   Objeto Split
</javaCODE>
<javaCODE>
   Objeto LowerCase
</javaCODE>
<javaCODE>
   Objeto UpperCase
</javaCODE>
<javaCODE>
   Objeto Merge
</javaCODE>
<links>
   ...
</links>
</javaCODES>
```

O elemento `service` é utilizado no arquivo XML gerado para registrar informações sobre serviços CORBA. Por questão de simplicidade, tratamos por serviço tanto os serviços CORBA (*CORBA services*) com o POA (*Portable Object Adapter*). As informações dos atributos desse elemento são extraídas do próprio sistema como também dos elementos `Property`, que definem as propriedades dos componentes ArchML.

O elemento `service` é formado pelos atributos `context`, que indica sob qual serviço estamos realizando a configuração; `operation`, que indica uma operação específica relativa ao serviço; `name`, pela qual definimos nomes que passamos como parâmetro para algum serviço e `value` que é o valor do parâmetro passado.

O código abaixo, por exemplo, serve para indicar a criação de um contexto de nomes no serviço de nomes CORBA.

```
<xsl:element name='service'>
   <xsl:attribute name='context'>
      <xsl:text>NamingService</xsl:text>
   </xsl:attribute>
   <xsl:attribute name='operation'>
      <xsl:text>createContext</xsl:text>
   </xsl:attribute>
   <xsl:attribute name='name'>
      <xsl:value-of select="document(@href)//component/@name"/>
   </xsl:attribute>
   <xsl:attribute name='value'>
      <xsl:value-of select="document(@href)//component/propertySet
            //property[@context='NamingService'
            and @propertyType='NamingContext']
            /@propertyValue"/>.<xsl:value-of select="
            //instance[@typeRef=$typeRef]/@name"/>.
   </xsl:attribute>
</xsl:element>
```

Podemos observar que utilizamos uma operação denominada `createContext` e passamos o nome do objeto para que esse contexto será criado e o nome do contexto em si. Outras operações são apresentadas mais adiante quando falarmos da geração de código para serviços.

7.4.1 *Script* de Geração de Templates de Códigos

De uma forma geral, o *script* de geração de templates Java/CORBA, denominado de `GenerateJavaImplCode.xsl`, é um arquivo XSL que lê uma especificação de arquitetura ArchML e gera um arquivo XML com todos os códigos Java dos objetos da arquitetura e com elementos XML que representem serviços CORBA específicos que serão tratados por outros *scripts* posteriormente. A estrutura do *script* é a seguinte :

```
<xsl:template match='/'>
   <xsl:element name='JavaCODES'>
      <xsl:attribute name='style'>
         <xsl:value-of select='//style/@href'/>
      </xsl:attribute>
         <xsl:apply-templates select='system'/>
      </xsl:element>
</xsl:template>

<xsl:template match='system'>
   <xsl:apply-templates select='instances'/>
</xsl:template>
```

Essa parte do *script* gera a estrutura básica do template de código, onde o elemento `javaCODES`, que possui o atributo `style`, que representa o estilo que a arquitetura segue. A partir daí é aplicado um template para o elemento `system`. Esse template simplesmente ativa o template para o elemento `instances` de ArchML.

```
<xsl:template match='instances'>
   <xsl:for-each select='instance'>
      <xsl:variable name='numIN'>
         <xsl:value-of select='count(document(../../
            types/type[@name=current()/@typeRef]/@href)
            //port[@direction="in"])'/>
      </xsl:variable>
      <xsl:variable name='numOUT'>
         <xsl:value-of select='count(document(../../
            types/type[@name=current()/@typeRef]/@href)
            //port[@direction="out"])'/>
      </xsl:variable>
```

Como há diferença na geração de código dependendo do tipo de objeto (relativo aos tipos de portas do mesmo), essa parte do *script* conta o número de portas de entrada e saída de cada tipo que a arquitetura possui e armazena nas variáveis `numIN` e `numOUT`.

Essas informações são usadas dentro do comando `xsl:choose` para lançar um template específico para cada tipo de objeto a ser gerado. Assim, para cada tipo, um elemento `javaCODE` e gerado e o template que possa tratar o tipo é invocado passando os parâmetros o nome e a referência do tipo. Assim, os templates `justClient`, `JustServer` e `ClientServer` são invocados para cada tipo, passando-se os parâmetros devidos.

```
<xsl:choose>
   <xsl:when test='$numIN=0'>
      <xsl:element name='javaCODE'>
         <xsl:call-template name='justClient'>
            ...
         </xsl:call-template>
      </xsl:element>
   </xsl:when>
   <xsl:when test='$numOUT=0'>
      <xsl:element name='javaCODE'>
         <xsl:call-template name='justServer'>
            ...
         </xsl:call-template>
      </xsl:element>
   </xsl:when>
   <xsl:otherwise>
      <xsl:element name='javaCODE'>
         <xsl:call-template name='ClientServer'>
            ...
         </xsl:call-template>
      </xsl:element>
   </xsl:otherwise>
</xsl:choose>
```

A seguir as partes do *script* de geração de códigos referentes a cada tipo de objetos são apresentados detalhadamente.

7.4.1.1 Apenas Cliente

O formato geral do template de código Java/CORBA a ser gerado pelo *script* para componentes do tipo Apenas Cliente é o seguinte:

XML Schema DataType	Java Type
xsd:Boolean	boolean
xsd:byte	byte
xsd:string	java.lang.String
xsd:short	short
xsd:int	int
xsd:long	long
xsd:float	float
xsd:double	double

Tabela 7.2: Relação Tipos XSD x Tipos Java.

```
public class <nomeClasse>{
   org.omg.CORBA.ORB orb = null;
   org.omg.CosNaming.NamingContext ncRef = null;
   static org.omg.CORBA.Object                              (I)
   <nomeTipoComponenteInvocado>Ref = null;
   public static void main(String[] args){
      new <nomeClasse>(args)
   }
   <nomeClasse>(String[] args){
      orb = org.omg.CORBA.ORB.init(args,null);
      <service context='NamingService'                      (IIa)
            operation='initNamingService'/>
      <service context='NamingService'                      (IIb)
            operation='createName'
            name=''
            value=''>
      <service context='NamingService'                      (IIc)
            operation='resolveName'
            name=''
            value=''>
   }
   private <tipoJavaRetornoPortadeSaída>                    (III)
         <nomePortadeSaída>(<tipoJavaParamPortadeSaída>
         <nomeParamPortadeSaída>){
         <invocation portName=«nomeParamPortadeSaída>"/>    (IV)
   }
}
```

O comando **I** deve ser repetido para cada objeto externo invocado pelo componente, sendo que da mesma forma devem ser repetidos os comandos (**IIb** e **IIc**). Já o comando **III** deve ser utilizado para cada porta (no caso somente de saída) do componente. O comando **IV** diz respeito a conexão com a porta de entrada de um componente externo.

Dentro do *script* que gera os códigos dos métodos, devemos transformar as informações de tipos de portas de ArchML, que são formatadas via XSD em tipos Java. Dessa forma, desenvolvemos um template específicos para realizar essa tarefa. Essa transformação é baseada na associação de valores XSD e Java apresentada na Tabela 7.2:

Os códigos relativos aos serviços, que serão manipulados por um outro *scripts*, serão explicados mais adiante.

7.4.1.2 Apenas Servidor

O código de um objeto que seja apenas servidor deve conter algumas informações relativas às funcionalidades do ORB para tratar da instanciação e criação de identificadores para esses objetos. O POA, componente do ORB responsável por essa tarefa deve ser utilizado. Um outro fator importante é que , como o objeto é apenas servidor, não deve haver a criação de referências para objetos externos, vistos que esses não são invocados (via CORBA) por esses objetos.

De forma geral, um template de um objeto com características de apenas Servidor tem o seguinte formato:

```
public class extends <nomeSistema>.<nomeClasse>ItfPOA{          (I)
   public static void main(String[] args){
      org.omg.CORBA.ORB orb = null;
      org.omg.CosNaming.NamingContext ncRef = null;
      org.omg.PortableServer.POA rootpoa = null;
      org.omg.CORBA.Object <nomeClasse>Ref = null;              (II)
      orb = org.omg.CORBA.ORB.init(args,null);
      <service context="POA"operation="initPOA"/>               (III)
      <service context="POA"operation="activatePOA"/>           (IV)
      <nomeClasse> <nomeClasse>Obj = new <nomeClasse>();        (V)
      <service context="POA"                                    (VI)
            operation="generateID"
            value=/>
      <service context="NamingService"                          (VII)
            operation="initNamingService"/>
      <service context="NamingService"                          (VIII)
            operation="createContext"
            name=
            value=/>
      <service context="NamingService"                          (IX)
            operation="bindName"
            name=''
            value=''
            contextName= '' />
      System.out.println("<nomeClasse> ready and waiting...");
      orb.run();
   }
   <nomeClasse>(){
      System.out.println(«nomeClasse> Working.");
   }
   public<tipoJavaRetornoPortadeSaída> <nomePortadeSaída>       (X)
      (<tipoJavaParamPortadeSaída> <nomeParamPortadeSaída>){ }
   }
```

O comando **I** conecta o componente com os códigos de comunicação POA gerados a partir da IDL. O comando **II** é gerado apenas uma vez e, no contexto do código Java/CORBA, serve para ser utilizado como referência do próprio objeto para o registro no serviço de nomes. Os comandos **III** e **IV** são, respectivamente, a inicialização e ativação do POA. O comando **V** é a criação da instância do próprio objeto (*servant*). O comando **VI** gera um ID a partir do *servant* via métodos do POA. Os comandos **VII**, **VIII**, e **IX** dizem respeito, respectivamente a inicialização do serviço de nomes, criação de um contexto de nomes e registro do objeto no contexto criado. O comando **X** deve ser repetido para todas as portas do componente.

7.4.1.3 Cliente-Servidor

O código de objetos do tipo Cliente-Servidor é uma fusão dos códigos anteriores, visto que o objeto atua, ao mesmo tempo, como cliente e como servidor. A seguir apresentamos o formato geral do template gerado pelo *script* para esse tipo de componente.

```
public class extends <nomeSistema>.<nomeClasse>ItfPOA{                    (I)
   static org.omg.CORBA.ORB orb = null;
   static org.omg.CosNaming.NamingContext ncRef = null;
   static org.omg.PortableServer.POA rootpoa = null;
   org.omg.CORBA.Object <nomeTipoComponenteInvocado>Ref = null;            (II)
   public static void main(String[] args){
      orb = org.omg.CORBA.ORB.init(args,null);
      <service context="POA"operation="initPOA"/>                          (III)
      <service context="POA"operation="activatePOA"/>                      (IV)
      <nomeClasse> <nomeClasse>Obj = new <nomeClasse>();                   (V)
      <service context="POA"operation="generateID"                         (VI)
            value= />
      <service context="NamingService"                                     (VII)
            operation="initNamingService"/>
      <service context="NamingService"operation="createContext"            (VIII)
            name=
            value=/>
      <service context="NamingService"operation="bindName"                 (IX)
            name=''
            value=''
            contextName=''/>
      System.out.println(«nomeClasse> ready and waiting...");
      orb.run();
   }
   <nomeClasse>(){
      <service context="NamingService"
            operation="initNamingService"/>
      <service context="NamingService"operation="createContext"
            name=''
            value=''/>
      <service context="NamingService"operation="resolveName"
            value=/>
      }
      public<tipoJavaRetornoPortadeSaída><nomePortadeSaída>                (IV)
         (<tipoJavaParamPortadeSaída><nomeParamPortadeSaída>){ }
      private <tipoJavaRetornoPortadeSaída><nomePortadeSaída>              (V)
         (<tipoJavaParamPortadeSaída><nomeParamPortadeSaída>){
            <invocation portName=«nomeParamPortadeSaída>"/>                (VI)
      }
   }
```

Como pode ser observado o template de código gerado para esse tipo de componente contempla todos os aspectos dos templates anteriores gerados para objetos apenas clientes e apenas servidores, apenas diferenciando a localização de alguns poucos comandos.

7.5 Geração de Códigos de Mecanismos de Invocação CORBA

Usando as marcações `invocation`, e `links` gerados pelo *script* anterior a partir da descrição ArchML, e observando o modo de comunicação adotado pelo estilo que arquitetura segue (descrito pelo atributo `style` do elemento `JavaCODES`), os mecanismos de invocação CORBA são gerados através de outro *script* XSL (`GenerateCORBAInvocation.xsl`). Basicamente temos as possibilidades de invocação síncrona e invocação assíncrona em nível arquitetural, resolvemos mapear esses tipos para os tipos de invocação CORBA *send oneway* e *synchronous*. De fato, resolvemos utilizar a DII para realizar as chamadas ao invés de utilizar os stubs, visto que isso exigiria a presença de stubs no lado cliente, o que dificultaria um pouco mais a execução da aplicação.

A geração dos códigos relativos à invocação CORBA é conseguido pela substituição dos conteúdos do elemento `invocation` pelo devido código CORBA. Esse elemento tem a seguinte sintaxe:

```
<invocation portName="splitUp"/>
```

Onde o valor do atributo `portName` é o nome da porta para qual devemos criar o código de invocação, sendo que para cada porta devemos observar os dados registrados no elemento `links`, registrado no final do arquivo XML gerado pelo *script* anterior. Esses valores referem-se aos dados necessários para a criação das portas, e, assim, das invocações.

O trecho do template XSL abaixo apresenta é utilizado para realizar a construção inicial da invocação.

```
<xsl:template match="//invocation">
   <xsl:variable name='port' select='@portName'>
   </xsl:variable>
   <xsl:for-each select="//link/point1[@port=$port]">
      org.omg.CORBA.Request request = <xsl:value-of select='
         document(../point2/@comp)/component/@name'/>
         Ref._request(«xsl:value-of select='../point2/@port'/>");
      <xsl:for-each select='current()/param'>
         request.add_in_arg().insert_<xsl:value-of
            select='@type'/>(<xsl:value-of select='@name'/>);
   </xsl:for-each>
```

Além dos dados de invocação, também são necessários informações dos estilos, visto que o funcionamento do modo de invocação depende da semântica de invocação definida no estilo. Assim, num segundo momento, olhamos na definição das portas do estilo e verificamos se, para o tipo de porta atual, o modo de comunicação da porta definida no arquivo Xtyle é síncrono ou assíncrono. O trecho do template XSL abaixo realiza essa tarefa:

```
<xsl:variable name='role'>
   <xsl:value-of select="document(@comp)//
         port[@name=$port]/../@role"/>
</xsl:variable>
<xsl:variable name='mode'>
   <xsl:if test='document(@comp)//
      port[@name=$port]/@direction="out"'>
      <xsl:value-of select='document(/JavaCODES/@style)//
         type[@name=$role]//ports/out/@mode'/>
   </xsl:if>
   <xsl:if test='document(@comp)//port[@name=
      $port]/@direction="in"'>
      <xsl:value-of select='document
         (/JavaCODES/@style)//type[@name=
         $role]//ports/in/@mode'/>
   </xsl:if>
</xsl:variable>
<xsl:choose>
   <xsl:when test='$mode="async"'>
      request.send_oneway();
   </xsl:when>
   <xsl:otherwise>
      <xsl:if test='document(@comp)//
         port[@name=$port]/returnType'>
      <xsl:call-template name='XSD2IDL'>
         <xsl:with-param name='xsdtype' select='document
            (@comp)//port[@name=$port]/returnType'/>
      </xsl:call-template>
   </xsl:if>
      request.invoke();
   </xsl:otherwise>
</xsl:choose>
```

Podemos observar que utilizamos um outro template externo para a transformação de tipos XSD em tipos IDL, necessário para formatar os dados de invocação. Esse template é o mesmo apresentado no início desse capítulo.

7.6 Geração de Códigos de Serviços CORBA

Depois de gerado o código das invocações na etapa anterior, devemos executar um novo *script* para tratar os códigos de serviços. Por código de serviços estamos nos referindo além dos serviços CORBA (*CORBA Services*), também aos comandos do POA.Usando o código já com as invocações resolvidas, aplicamos esse *script* para gerar códigos relativos às propriedades definidas nos componentes. Assim, cada elemento `service` é avaliado dentro do código XML gerado pelo *script* `GenerateJavaImplCode.xsl` e o código Java/CORBA correspondente é gerado através do *script* (`GenerateCORBAServicesCODE.xsl`).

Esse *script* verifica o context de cada elemento service e invoca um template XSL específico para tratar os serviço, passando os parâmetros necessários para tal. O trecho de código a seguir mostra essa tarefa:

```
<xsl:template match="//service">
<xsl:choose>
   <xsl:when test="@context='NamingService'">
      <xsl:call-template name='NamingServiceCODE'>
         <xsl:with-param name='operation' select='@operation'/>
         <xsl:with-param name='name' select='@name'/>
         <xsl:with-param name='value' select='@value'/>
         <xsl:with-param name='contextName' select='@contextName'/>
      </xsl:call-template>
   </xsl:when>
   <xsl:when test="@context='POA'">
      <xsl:call-template name='POACODE'>
         <xsl:with-param name='operation' select='@operation'/>
         <xsl:with-param name='name' select='@name'/>
         <xsl:with-param name='value' select='@value'/>
      </xsl:call-template>
   </xsl:when>
</xsl:choose>
</xsl:template>
```

A seguir apresentamos separadamente a geração de código para cada tipo de serviço.

7.6.1 Geração de Código para o POA

A geração de código para o POA deve ser realizada de forma a fornecer basicamente mecanismos de inicialização, ativação e geração de identificadores para servants. Dessa forma os seguintes elementos XML podem ser encontrados no arquivo XML a ser tratado:

```
<service context="POA" operation="initPOA"/>
```

Esse comando trata da sinalização da necessidade da geração de código para a inicialização do POA. Essa inicialização é realiza pela resolução desse serviço via métodos da interface do ORB. Um trecho do código gerado para esse comando é:

```
rootpoa = org.omg.PortableServer.POAHelper.narrow(
     orb.resolve_initial_references("RootPOA"));
```

O comando seguinte apresentado a seguir, define a operação `activatePOA`:

```
<service context="POA" operation="activatePOA"/>
```

Esse comando indica a geração de código para ativar a instância do POA. Assim o código gerado para ele é o seguinte:

```
rootpoa.the_POAManager().activate();
```

O último comando relativo ao POA é o que define a geração de identificadores para servants. Para que o código Java/CORBA possa ser gerado corretamente é necessário passar como parâmetro para o template que realiza a geração desse código, o nome do *servant* para qual o ID deve ser gerado pelo POA. O atributo `value` é usado para este fim, no código abaixo utilizamos o valor `LowerCase`.

```
<service context="POA" operation="generateID"value=/>
```

O código gerado para esse comando deve ter a seguinte estrutura:

```
LowerCaseRef = rootpoa.servant_to_reference(LowerCaseObj);
```

7.6.2 Geração de Código para o Serviço de Nomes CORBA

Da mesma forma que fizemos para o POA também um conjunto de elementos service são gerados pelo *script* `GenerateJavaImplCode.xsl` indicando os locais e as operações a serem gerados para o serviço de nomes.

A primeira operação encontrada é a

```
<service context="NamingService" operation="initNamingService"/>
```

que sinaliza a geração do código para a inicialização do serviço de nomes a partir da invocação de métodos da interface do ORB. O código Java/CORBA gerado para esse comando é o seguinte:

```
org.omg.CORBA.Object obj =
      orb.resolve_initial_references("NameService");
ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
```

O comando seguinte é utilizado para a criação de contextos de nomes dentro do servidor de nomes CORBA. Ele tem a seguinte sintaxe:

```
<service context="NamingService"operation="createContext"
        name="Merge"value="Operation."/>
```

O elemento para essa operação utiliza dois atributos que identificam, respectivamente, o nome do objeto para o qual estamos criando o nome (`name`) e o nome do contexto a ser criado. Um observação importante é que o nome do contexto pode conter mais de um nó da árvore. Por exemplo, podemos construir o contexto `Strings.Operacaoes.`, onde temos os contextos `Strings` e `Operacoes`. O código gerado, considerando o exemplo anterior, é o seguinte:

```
org.omg.CosNaming.NameComponent[] name1 = {
        new org.omg.CosNaming.NameComponent("Operation",)};

org.omg.CosNaming.NamingContext cntx1 =
        ncRef.bind_new_context(name1);
```

Os últimos dois comando utilizados pelo serviço de nomes dizem respeito a invocação de objetos e o registro de objetos no servidor de nomes CORBA. O primeiro, cuja operação é definida como o valor `createName`, permite a criação de um nome para ser pesquisado no servidor de nomes. A sintaxe completa é a seguinte:

```
<service context="NamingService" operation="createName"
           name="RemoveVowels"
           value="Strings.StringOperations."/>
```

O atributo name indica o nome da variável que receberá a referência do nome. O atributo value possui o nome do contexto que devemos procurar para localizar o nome. O código final gerado é o seguinte:

```
org.omg.CosNaming.NameComponent[] RemoveVowelsName = {
new org.omg.CosNaming.NameComponent("Strings",),
new org.omg.CosNaming.NameComponent("StringOperations",)
```

Observe que esse código está incompleto. De fato, após uma chamada de `createName` temos sempre uma chamada de um outro serviço, cuja operação e definida com o valor `resolveName`. Essa operação necessita de um parâmetro extra que indica o nome do objeto para o qual estamos resolvendo o nome. O código abaixo apresenta um exemplo, onde buscamos o objeto que possui o nome `removevowels` registrado, sendo a referência para esse objeto será armazenada na variável `RemoveVowels`.

```
<service context="NamingService" operation="resolveName"
          name="RemoveVowels" value="removevowels"/>
```

O código Java/CORBA gerado a partir dessa especificação é o seguinte:

```
,new org.omg.CosNaming.NameComponent("removevowels",)};
        RemoveVowelsRef = ncRef.resolve(RemoveVowelsName);
```

Como podemos observar a especificação anterior é completada por essa.

O último comando, utilizado para o registro de servidores no serviço de nomes, tem o atributo operation com valor `bindName`, e utiliza os parâmetros name, que indica o nome da variável que possui a referência do objeto que estamos querendo disponibilizar no serviço de nomes; o atributo value que indica o nome pelo qual esse objeto será disponibilizado e o atributo `contextName`, que indica o nome do contexto a baixo do qual o nome será disponibilizado. O código a seguir é utilizado para indicar a necessidade de gerar um código para o objeto `ShowFile`, que será ligado com o nome `showfile` a baixo do contexto `String.StringOperations.`

```
<service context="NamingService" operation="bindName"
        name="ShowFile" value="showfile"
        contextName="Strings.StringOperations."/>
```

O código gerado para o exemplo anterior é o seguinte:

```
org.omg.CosNaming.NameComponent[] name1 = {
      new org.omg.CosNaming.NameComponent("Strings",)};
org.omg.CosNaming.NamingContext cntx1 = ncRef.bind_new_context(name1);
org.omg.CosNaming.NameComponent[] name2 ={
      new org.omg.CosNaming.NameComponent("StringOperations",)};
org.omg.CosNaming.NamingContext endcntx = ncRef.bind_new_context(name2);
org.omg.CosNaming.NameComponent[] ShowFileName = {
      new org.omg.CosNaming.NameComponent("showfile",)};
endcntx.rebind(ShowFileName,ShowFileRef);
}catch ( org.omg.CosNaming.NamingContextPackage.AlreadyBound nf){
   try{
      org.omg.CosNaming.NameComponent[] ShowFileName = {
         new org.omg.CosNaming.NameComponent("Strings",),
         new org.omg.CosNaming.NameComponent("StringOperations",),
         new org.omg.CosNaming.NameComponent("showfile",)};
         ncRef.rebind(ShowFileName,ShowFileRef);
   }catch(Exception e){}
}
```

Observe que geramos cada elemento do contexto separadamente, sendo que o último nível de contexto conterá o nome `endcntx`. Caso haja uma exceção indicando que o contexto já existe, executaremos, dentro da cláusula de tratamento de exceção apenas a criação do nome e a ligação da referência do objeto a baixo do contexto root `ncRef`.

7.7 Geração de Código para Outras infraestruturas de Middleware

A construção de *scripts* para a geração de códigos para outras infraestruturas de infraestruturas de middleware orientados a objetos deve ser realizada por desenvolvedores que detenham bastante conhecimento do middleware em questão. De fato, esse conhecimento é necessário para que os códigos recorrentes sejam identificados e colocados nos templates. Contudo, de forma geral, para exemplificar como realizar essa tarefa devemos observar os seguintes pontos:

1. **Quais as interfaces a serem geradas e como definir o nomes das classes e objetos**

2. **Quais os métodos de comunicação oferecido pelo middleware**

   Um *script* específico deve ser criado para manipular os mecanismoa de comunicação disponibilizados pelo middleware. Dessa forma, de acordo com o estilo seguido pela arquitetura, devemos capturar as formas de comunicação e gerar os códigos devidos.

3. **Quais os serviços oferecido pelo middleware**

   Normalmente infraestruturas de middleware oferecem um serviço de nomes pelo menos. Esse serviço deve ser manipulado e os nomes dos objetos devem ser capturados como instâncias da arquitetura. Sendo que hierárquias de nomes devem ser explicitadas como propriedades dos componentes. Que devem ser avaliadas, assumindo, no caso de omissão, algum valor padrão. Outros serviços também podem ser fornecidos, devendo também serem tratados por algum *script*.

4. **Como configurar o ambiente de execução para as aplicações**

   Para que um objeto possa executar, algumas vezes é necessário de manipular alguma informação do middleware, como alguma política de segurança, alguma referência, entre outros. Esses fatores dever ser tratados sem que o desenvolvedor da arquitetura venha a ser consultado.

Para melhor entende esses passos, vamos exemplificá-los através do desenvolvimento de alguns códigos para a criação *scripts* de geração de templates RMI [60].

Como observado anteriormente, o desenvolvedor dos *scripts* de geração de código para um middleware específico deve estar familiarizado com o desenvolvimento de aplicações para esses middlwares. Inicialmente, como identificamos nos ponto número 1, devemos decidir como geraremos os nomes dos objetos, referência, instâncias e interfaces das aplicações. Para isso devemos considerar os nomes identificados na descrição ArchML. Dessa forma, podemos utilizar, por exemplo, as seguintes convenções:

1. Para os componentes que tivemos portas de entrada, devemos gerar uma interface Java com o nome do componente acompanhado com os caracteres `itf`.

   Ex.: seja o componente de nome `Split`, que possua uma porta de entrada com o nome `splitIN`, que recebe como parâmetro uma string de nome var. Poderíamos gerar uma interface Java com nome `SplitItf`, contendo um método público `splitIN`. Observe o código a baixo.

```
public interface SplitItf extends java.rmi.Remote {
   void SplitIN(String var) throws java.rmi.RemoteException;
}
```

2. Da mesma forma que fizemos anteriormente para CORBA, se o componentes só tem portas de entrada ele é um servidor, se tiver portas de entrada e saída ele é cliente-servidor, e se tiver apenas portas de saída ele é um cliente. Desse modo, poderíamos gerar templates específicos para cada caso, pois a estrutura geral de cada um é bastante similar. Uma decisão que teríamos que realizar seria relativo ao nome das referências a objetos externos. Dessa forma poderíamos adotar uma abordagem de utilizar sempre o nome da instância remota como o nome da variável de referência ao objeto externo.

   Ex.: seja um componente com uma porta de saída (referência a um objeto externo) para uma instância denominada `Remove`, poderíamos utilizar o nome `Remove`, como sendo o nome da variável que receberia a referência do objeto referenciado pela porta.

3. Tendo portas de entrada, também devemos gerar uma nova classe que implemente o serviço oferecido pelo objeto. Essa classe deve extender a classe `UnicastRemoteObject` e implementar a interface do serviço. Podemos convencionar essa classe com o mesmo nome do componente terminando com `Impl`. Nessa classe deverá ter um construtor padrão e os métodos identificados na interface.

   Ex.: no exemplo, do item 1, devemos gerar uma classe denominada `SplitImpl` que extenda `UnicastRemoteObject` e implemente `SplitItf`. Observe o código a baixo.

```
public class SplitImpl extends java.rmi.server.UnicastRemoteObject
implements SplitItf {
    public SplitImpl() throws java.rmi.RemoteException{
      super();
      System.out.println("Split Working.");
    }

    public void splitIn(String var){

    }
 }
```

Logo em seguida, no ponto 2, devemos observar como o middleware realiza comunicação, ou seja, que formas de comunicação são disponibilizadas aos desenvolvedores. Na sua forma padrão, RMI fornece comunicação síncrona apenas, onde os objetos devem ser invocados e o resultado da computação deve ser esperado. Podemos observar em códigos de aplicações RMI que é bastante trivial e transparente a forma de invocação de RMI, bastando apenas realizar a chamada do método do objeto remoto. Assim podemos realizar a geração direta do mecanismos de invocação bastante realizar a invocação do método requisitado.

No ponto número 3, onde tratamos dos serviços oferecidos pelo middleware, devemos considerar as informações anotadas nas descrições dos componentes para realizar a geração correta e customizada dos serviços oferecidos. Assim, em RMI, podemos identificar o Serviço de Nomes e a customização do ambiente de segurança. Dessa forma, podemos, por exemplo, identificar as seguintes informações para a geração de código para esses serviços em RMI:

- O nome que o objeto deverá receber no espaço de nomes

  Esse nome deve ser identificado como uma propriedade do componente descrito em ArchML. Ex.: suponha que o objeto `Split` deverá ser registrado como `splitserver`. Observe o código abaixo.

```
SplitImpl Splitobj = new SplitImpl();
Naming.rebind("splitserver", Splitobj);
```

- Definição das políticas de segurança da máquina virtual Java

  Essas informações de segurança tanto podem ser definida diretamente como propriedades do componente como podem ser definidas a través da identificação de um arquivo de descrição de políticas.

A construção de *scripts* de geração de templates de classes com códigos de infraestruturas de middleware específicos deve seguir uma padronização que deve ser observada pelo desenvolvedor. Dessa forma, inicialmente devemos avaliar quais serviços estão disponíveis no middleware e como os códigos desses estão estruturados para que possamos gerar os códigos de forma correta e genérica o suficiente para que esses possam funcionar em qualquer aplicação. Nessa seção, para não nos estendermos demais, apresentamos apenas algumas "dicas" de como realizar a geração de código para RMI. Uma explanação mais detalhada sobre essa geração pode ser encontrada em [113].

7.8 Conclusão

A etapa de geração de código apresentada nesse capítulo, permite a criação de complexas aplicações distribuídas baseadas das arquiteturas devidamente descritas e validadas pelas ferramentas de DraX apresentadas nas etapas anteriores. Aqui utilizamos *scripts* XSL para a partir de descrições ArchML e Xtyle, construir templates de programação Java onde os códigos do middlware estão devidamente acomodados cabendo ao desenvolvedor a tarefa apenas de fornecera implementação para os métodos dos objetos.

Ao longo desse capítulo, apresentamos mais detalhadamente a geração de código para middleware CORBA. Contudo, como apresentado na Seção 7.7, a adaptação/criação de *scripts* outras infraestruturas de middleware não é uma tarefa muito difícil.

Uma característica importante que utilizamos durante a essa etapa de geração de código, diz respeito a utilização de um único arquivo onde todos os códigos gerados são armazenados, cabendo ao desenvolvedor extrair os códigos de cada um dos componentes e salvá-los em arquivos específicos. Na verdade também poderíamos automatizar essa tarefa utilizando APIs XSL como a oferecida por XALAN [92]. Contudo, seria mais uma etapa a ser trabalhada no desenvolvimento de DraX e como o objetivo da tese e demonstrar a aplicabilidade direta de tecnologias e ferramentas atuais no tratamento de arquiteturas e estilos distribuídos e de fornecer um ambiente didático resolvemos utilizar diretamente um interpretador XSL e construir apenas os templates para realizar as tarefas de geração de código.

Capítulo 8

Estudos de Caso

8.1 Introdução

Para mostrar a aplicação de todas as ferramentas e linguagens de DraX e para validar sua aplicabilidade como um *framework* para o ensino de arquitetura de software, apresentaremos nesse capítulo dois estudos de caso que abrangem diversas características importantes, tanto relativas à aplicação de DraX como quanto às novas contribuições conceituais que cercam as linguagens e ferramentas desse *framework* no que diz respeito a Arquiteturas de Software, Estilos Arquiteturais, Modelos Formais e Sistemas Distribuídos.

Para validar as características de DraX escolhemos dois estudos de caso, sendo que cada um deles possui algum tipo de característica específica que permite a observação dos conceitos propostos nessa tese de forma direta e simplificada. O primeiro estudo de caso é uma versão do clássico sistema denominado *Remove Vowels*, que foi proposto em [14] e que consiste de um sistema de remoção de vogais em uma cadeia de caracteres. Embora seja um sistema bastante simples, podemos observar mais facilmente a utilização das idéias de DraX, através da aplicação das ferramentas propostas. Esse sistema utiliza um estilo arquitetural derivado `Pipeline`. Nesse estudo de caso, por definição do estilo, utilizamos uma comunicação síncrona e serão gerados códigos RMI para suportar essa comunicação.

O segundo estudo de caso na verdade é uma variação do primeiro, apresentamos a construção de um estilo arquitetural novo, que é uma `Pipeline Assíncrono`, e o aplicamos à descrição do sistema realizado no primeiro estudo de caso. Daí, verificamos a nova geração de código, agora em CORBA, com os novos requisitos de comunicação. Com ele queremos mostrar a praticidade em tanto de construir um estilo através de derivação de um já pré-existente, como mostrar a facilidade de se modificar a estrutura de comunicação de uma arquitetura através apenas da redefinição do estilo que essa está utilizando, validando a idéia de termos uma linguagem específica para a descrição de estilos.

Nesses estudos de caso utilizamos a metodologia proposta para Drax, apresentada no Capítulo 4 e refinada nos demais capítulos. Dessa forma, para facilitar tanto a apresentação como a exploração desses estudos de caso dividimos a apresentação em diversas subseções, sendo que cada subseção contempla um aspecto do desenvolvimento com DraX. Segue relacionada cada uma das subseções utilizadas:

a) Descrição da Aplicação
A aplicação é apresentada de maneira informal. A idéia desse item a fornecer uma visão simples e funcional da aplicação.

b) Especificação dos Componentes

Os componentes que formarão a aplicação são definidos e especificados nesse item através da linguagem ArchML. Tanto a estrutura sintática de ArchML como a descrição do DDP e subseqüente geração de especificação XMI relacionada, devem ser realizados nesse momento.

c) Especificação da Arquitetura/Estilo

Utilizando os componentes especificados no item anterior construiremos a especificação da arquitetura como um todo. ArchML também é utilizada para este fim.

Como apresentado na Seção 5.3 nesse momento devemos definir qual estilo a arquitetura deve seguir. Utilizando a abordagem proposta nessa tese de permitir tanto estilos já pré-definidos como de se descrever um estilo novo utilizando a linguagem Xtyle, nesse item também pode ser necessário realizar a tarefa de especificar um novo estilo para a aplicação (caso este não exista). Assim, devemos lançar mão de Xtyle para realizar essa tarefa.

Também nessa etapa devemos utilizar uma ferramenta de projeto UML para descrever os DDPs de cada componente da arquitetura e dos tipos dos estilos, além de gerar as especificações XMI correspondentes.

d) Validação e Verificação de Consistências Sintáticas na Arquitetura/Estilo

Tendo realizado a especificação ArchML da arquitetura e, se necessário, a especificação Xtyle de um novo estilo, devemos validar sintática e estruturalmente essas especificações. Essa tarefa é realizada pelos esquemas de validação XSD e pelos esquemas Schematron. Com relação a um novo estilo que eventualmente tenha sido produzido, devemos, antes de tudo, gerar o *script* de validação para arquiteturas e em seguida aplica-lo à especificação ArchML.

e) Verificação de Consistência Comportamental na Arquitetura

Do mesmo modo que fizemos no item anterior devemos também verificar as consistências comportamentais da arquitetura. Para isso utilizaremos os *scripts* de geração de especificações formais a partir da descrição XMI de cada componente e aplicaremos os outros *scripts* de adaptação das especificações para que possamos utilizar o MWB para realizar as verificações comportamentais de cada par de conexões.

f) Verificação de Aderência Comportamental ao Estilo

Sendo o estilo comportamentalmente consistente, devemos verificar se o comportamento de cada componente condiz com o comportamento esperado para o tipo arquitetural equivalente definido no estilo que a arquitetura está seguindo. Da mesma forma que fizemos no item anterior, devemos gerar especificações formais tanto dos comportamentos tipos do estilo como dos comportamentos dos componentes e realizar as verificações com o MWB.

g) Semântica $\mathcal{R}\pi$ e Análise

Depois de terem sido sintaticamente, estruturalmente e comportamentalmente validados tanto a arquitetura como o estilo, devemos agora verificar se a aplicação como um todo se comporta de forma satisfatória, ou seja, se os componentes integrados e em execução não entrarão em impasses. Nesse item, a descrição π-cálculo do sistema como um todo é gerada e analisada formalmente pelo MWB assim como a descrição $\mathcal{R}\pi$ pode ser gerada e verificada.

h) Geração de Templates

Nesse item estaremos com especificações completamente validadas e verificadas, sendo possível partir para a implementação. Nesse momento utilizaremos os *scripts* de DraX para realizar a geração de código no middleware escolhido para cada um dos componentes da arquitetura.

i) Geração do Código dos Serviços e de Invocações

Utilizando os códigos gerados no item anterior, aplicamos os *scripts* de geração de serviços do middleware para substituir as marcações de serviços por sua implementação no middleware escolhido. Além disso, utilizamos o *script* de geração de mecanismos de invocações para substituir as marcações relativas a invocações pelos códigos do middleware correspondente.

j) Implementação da Lógica da Aplicação

Nesse item os códigos dos métodos e a estrutura de invocação local para os objetos da aplicação são implementados manualmente a partir dos códigos gerados nos items anteriores.

k) Execução

A execução da aplicação, depois de todas as etapas concluídas, é realizada nesse item.

De forma a simplificar as descrições dos códigos algumas partes das especificações e implementações foram suprimidas, sendo que os códigos completos desses estudos de caso podem ser encontrados no Apêndice A.

8.2 Ambiente de Experimentação

Mesmo tendo repetido isso diversas vezes ao longo dessa tese, resolvemos, mais uma vez, explicar que um dos objetivos principais desse trabalho e mostrar que é possível se produzir aplicações distribuídas utilizando os conceitos de arquitetura de software através de tecnologias e ferramentas que de fato fazem parte do dia-a-dia dos desenvolvedores de software, além de fornecer uma ambiente simplificado para o ensino de arquitetura de software.

Sendo assim, utilizamos diversas ferramentas ao longo desse trabalho. Ferramentas essas baseadas em também diversas tecnologias. Sendo que optamos por utilizar apenas software gratuito. Esse fato reforça ainda mais a idéia de que nem mesmo precisaríamos incorporar gastos extras para usufruir as vantagens do desenvolvimento arquitetural e das aplicações distribuídas baseadas em middlewares. A Tabela 8.1 apresenta todas as ferramentas que utilizamos para validar as idéias de DraX:

Utilizamos o *Schematron Validator* para a realização de várias tarefas relativas à validação e verificação sintática. Essa ferramenta é usada para avaliar tanto esquemas XSD como Schematrons das arquiteturas e dos estilos arquiteturais. Além disso, os *scripts* XSLT foram aplicados utilizando essa mesma ferramenta.

Jaxen é uma API XPath para Java que facilita bastante o desenvolvimento de *scripts* de manipulação de arquivos XMI. Essa ferramenta foi usada nos *scripts* de geração de especificações π-cálculo a partir dos arquivos XMI com o comportamento dos componentes.

O MWB (*Mobility WorkBench*) é uma implementação de ferramenta de manipulação de especificações π-cálculo. Essa ferramenta é implementada sob a linguagem **Standard ML of**

Ferramenta	Atuação	Site
Schematron Validator	Validação Schematron e XSD; Geração XSLT	www.topology.com
Jaxen	API Java para XPath	sourceforge.net/projects/jaxen
MWB (*Mobility WorkBench*)	Ferramenta π-cálculo	www.docs.uu.se/~victor/
J2sdk 1.4.1	Linguagem de Programação e Middleware RMI	java.sun.com/j2se/1.4
OpenORB	ORB CORBA	sourceforge.net/projects/openorb
Poseidon UML CE 1.5	Construção de DDPs	www.gentleware.com/
umltool	Geração XMI 1.1	umltool.d-a-t.com

Tabela 8.1: Ferramentas Usadas nos Estudos de Caso.

New Jersey em sua versão 0.93. Como atualmente essa versão do compilador SML não mais está disponível, tivemos que realizar diversas alterações nos códigos do MWB para adaptá-lo à versão SML 110.

O OpenORB é uma implementação CORBA de código aberto que possui um conjunto considerável de serviços disponíveis.

O Poseidon UML é uma ferramenta de criação de modelos UML. Como pode ser observado na Tabela 8.1, utilizamos duas ferramentas UML o Poseidon e o umltool. Tivemos que optar por essas duas ferramentas pelo fato de que o umltool, que gera especificações XMI na versão 1.1, que foi a que escolhemos para manipular através dos *scripts* de DraX, não permite a elaboração de diagramas como ações que apontem para o próprio estado de origem (auto-ações). Assim, utilizamos o Poseidon UML, que permite construir esse tipo de diagrama, porém gera especificações XMI 1.0. E então importamos as especificações XMI 1.0 dentro do umltool e geramos especificações XMI 1.1.

8.3 Estudo de Caso 1: *RemoveVowels* Assíncrono

8.3.1 Descrição da Aplicação

Esse estudo de caso, embora seja bastante simples, nos permitirá ter uma visão mais planificada tanto dos conceitos como das ferramentas de DraX. Apresentado em [14], essa aplicação é formada por três componentes, a saber: o primeiro, denominado `CatFile`, é uma fonte de dados em forma de uma cadeia de caracteres. Esse componente emite essa cadeia para o segundo componente, denominado `RemoveVowels`, que extrai as vogais da cadeia recebida e gera uma nova cadeia, envianda para o terceiro componente, denominado `ShowFile`, que simplesmente apresenta os dados recebidos.

Resolvemos utilizar o estilo derivado `Pipeline` para a especificação da arquitetura. A adoção desse estilo foi sugerida em [14]. De uma forma geral podemos apresentar a aplicação diagramaticamente através da Figura 8.1.

Figura 8.1: Estrutura Geral do Estudo de Caso 1.

8.3.2 Especificação dos Componentes

O primeiro componente da aplicação é `CatFile`, cuja especificação ArchML (`CatFile.aml`) é apresentada a seguir. Esse componente possui apenas uma porta de saída (`output`) onde é passada uma `string`. Também podemos observar a utilização do papel "`Source`" do estilo `Pipeline` para a interface desse componente.

```
<?xml version="1.0"?>
<component name="CatFile"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="Component.xsd»

   <document>
      <version num="1.0"/>
      <author name="Cidcley T. de Souza"
              email="cts@cin.ufpe.br"/>
      <lastUpdate date="2002-09-25"/>
      <comments>
         Gera uma String e a envia pela porta de saida.
      </comments>
   </document>

   <propertySet>
      <property context="NamingService"
                propertyType="NamingContext"
                propertyValue="Strings.StringOperations"/>
   </propertySet>

   <interfaces>
      <interface role="Source">
         <port name="output" direction="out">
            <param name="phrase" type="xsd:string"/>
         </port>
      </interface>
   </interfaces>
   <behavior href="CatFile.xmi"/>
</component>
```

O comportamento de `CatFile`, representado pelo DDP da Figura 8.2, realiza uma computação interna seguida de uma comunicação assíncrona pela porta `output`. Por questão de simplicidade resolvemos não apresentar a especificação XMI correspondente.

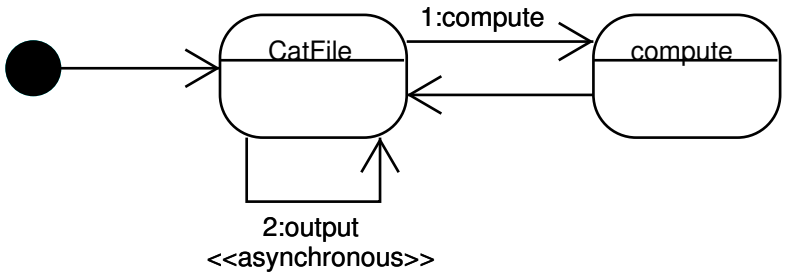


Figura 8.2: Componente `CatFile`.

O componente seguinte é `RemoveVowels`, cuja especificação ArchML (`RemoveVowels.aml`) é apresentada a seguir. Esse componente, como pode ser visto na especificação, é do tipo "`Filter`" e possui uma porta de entrada (`getPhrase`) e uma porta de saída (`outPhrase`).

```
<component name="RemoveVowels"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="Component.xsd»
   <document>
      <version num="1.0"/>
      <author name="Cidcley T. de Souza"
              email="cts@cin.ufpe.br"/>
      <lastUpdate date="2002-09-25"/>
      <comments>
         Extrai as vogais de um texto recebido pela porta de entrada e
         repassa o resultado pela porta de saida.
      </comments>
   </document>
   <propertySet>
      <property context="POA"
                propertyType="ThreadPolicy"
                propertyValue="MultiThread"/>

      <property context="NamingService"
                propertyType="NamingContext"
                propertyValue="RootContext"/>
   </propertySet>
   <interfaces>
      <interface role="Filter">
         <port name="getPhrase" direction="in»
            <param name="phrase" type="xsd:string"/>
         </port>
         <port name="outPhrase" direction="out»
            <param name="phrase" type="xsd:string"/>
         </port>
      </interface>
   </interfaces>
   <behavior href="RemoveVowels.xmi"/>
</component>
```

O comportamento de `RemoveVowels`, apresentado na Figura 8.3, mostra que esse componente inicialmente recebe um pedido na porta `getPhrase`, sendo que em seguida realiza

uma computação interna com os dados recebidos. Ao termino desse processamento, o resultado é enviado pela porta `outPhrase` e o componente retorna ao estado inicial.

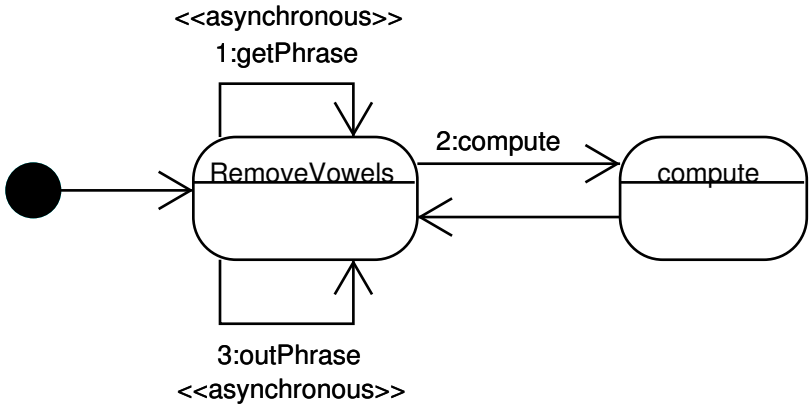


Figura 8.3: Componente `RemoveVowels`.

A especificação ArchML do componente seguinte, `ShowFile` (`ShowFile.aml`), é apresentada a seguir. Esse componente (que é do tipo "`Sink`") recebe dados através da porta de entrada (`showPhrase`), formata-os e os apresenta na tela.

```
<?xml version="1.0"?>
<component name="ShowFile"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="Component.xsd»
   <document>
      <version num="1.0"/>
      <author name="Cidcley T. de Souza"
              email="cts@cin.ufpe.br"/>
      <lastUpdate date="2002-09-25"/>
      <comments>
         Mostra o que for recebido na porta de entrada.
      </comments>
   </document>
   <propertySet>
      <property context="POA"
                propertyType="ThreadPolicy"
                propertyValue="MultiThread"/>
      <property context="NamingService"
                propertyType="NamingContext"
                propertyValue="RootContext"/>
   </propertySet>
   <interfaces>
      <interface role="Sink">
         <port name="showPhrase" direction="in»
            <param name="phrase" type="xsd:string"/>
         </port>
      </interface>
   </interfaces>
   <behavior href="ShowFile.xmi"/>
</component>
```

O comportamento de `ShowFile` é apresentado na Figura 8.4.

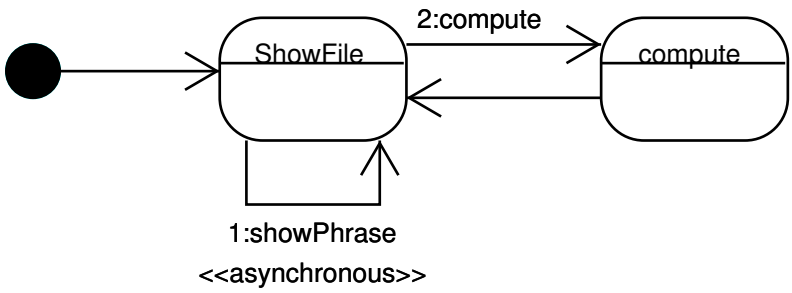


Figura 8.4: Componente `ShowFile`.

8.3.3 Especificação da Arquitetura/Estilo

A arquitetura do sistema, também descrita em ArchML (`SystemRemoveVowels.aml`), é apresentada a seguir.

```
<?xml version="1.0"?>
<system name="SystemRemoveVowels"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="System.xsd»
   <document>
      <version num="1.0"/>
      <author name="Cidcley T. de Souza"
              email="cts@cin.ufpe.br"/>
      <lastUpdate date="2002-09-26"/>
      <comments>
         Sistema baseado em Pipeline que envia String e retira
         vogais mostrando os resultados.
      </comments>
   </document>
   <style href="Pipeline.xty"/>
   <types>
      <type name="CompCatFile" href="CatFile.aml"/>
      <type name="CompRemVow" href="RemoveVowels.aml"/>
      <type name="CompShowFile" href="ShowFile.aml"/>
   </types>
   <instances>
      <instance name="catfile" typeRef="CompCatFile"/>
      <instance name="removevowels" typeRef="CompRemVow"/>
      <instance name="showfile" typeRef="CompShowFile"/>
   </instances>
   <links>
      <link name="conn1">
         <point instRef="catfile" portRef="output"/>
         <point instRef="removevowels" portRef="getPhrase"/>
      </link>
      <link name="conn2">
         <point instRef="removevowels" portRef="outPhrase"/>
         <point instRef="showfile" portRef="showPhrase"/>
      </link>
   </links>
</system>
```

Como estamos utilizando um estilo pré-definido na descrição dessa arquitetura (`Pipeline`) não é necessáruio se descrever o estilo, bastando apenas referênciá-lo da descrição da arquitetura. No segundo estudo de caso apresentaremos a aplicação dessa abordagem.

8.3.4 Validação de Consistências Sintáticas na Arquitetura/Estilo

Quanto a validação dos componentes da arquitetura, cada um deve ser submetido à validação contra a gramática formal XSD de componentes ArchML (`Component.xsd`) como também ao esquema Schematron de verificação estrutural de componentes (`CompValidateSyntax.sch`).

O mesmo deve ser feito para a especificação da arquitetura, que deve ser validada contra sua gramática formal XSD (`System.xsd`), como também com relação ao Schematron de verificação estrutural de arquiteturas (`SysValidateSyntax.sch`). No caso das arquiteturas uma verificação extra deve ser realiza com relação às relações sintático-topológicas entre os componentes que a formam. Essa tarefa é realizada pelo esquema Schematron (`ArchCheck.sch`).

Utilizando o *Schematron Validator* para aplicar os devidos esquemas para cada componente e depois para a arquitetura como um todo verificamos que não encontramos erros nas especificações.

Particularmente para esse estudo de caso estamos utilizando um estilo derivado já definido anteriormente, assim, não estamos apresentando a etapa de validação sintática e estrutural do mesmo.

8.3.5 Validação de Consistência Comportamental na Arquitetura

Estando as especificações ArchML devidamente verificadas e validadas com relação à estrutura sintática e topológica, devemos agora utilizar os *scripts* de verificação de consistências comportamentais de DraX para avaliar a compatibilidade dos componentes da arquitetura. Assim, inicialmente devemos aplicar o *script* Java (`XMI2PI.java`) em cada arquivo XMI obtidos a partir dos DDPs, para gerar as especificações π-cálculo de cada componente.

As especificações geradas para cada componente da arquitetura são as seguintes:

$$
\begin{aligned}
&agent\ CatFile(output) = (\hat{}data1)\, t.\,'output < data1 > .CatFile(output)\\
&agent\ RemoveVowels(getPhrase, outPhrase) = (\hat{}data2)\, getPhrase(data1).t\\
&\qquad\qquad .\,'outPhrase < data2 > .RemoveVowels(getPhrase, outPhrase)\\
&agent\ ShowFile(getPhrase) = getPhrase(data1).t.ShowFile(getPhrase)
\end{aligned}
$$

Em seguida, executando o programa (`CompPI.java`) para realizar a associação entre a especificação de cada tipo definido no estilo com a especificação do componente. Essa associação permite a expansão de nomes de portas dos tipos (não é o caso nessa especificação) e a renomeação dos nomes de portas, realizamos a verificação dos componentes dois a dois, considerando apenas as portas de conexão de cada par. Assim, verificamos se o componente `CatFile` é compatível com o componente `RemoveVowels`. Como apresentado na seção 6.3, esse script modifica as especificações de forma a considerar apenas as portas conectadas em cada par de componentes e gera um agente com a composição em paralelo dos componentes tendo o nome dos canais igualados de modo a permitir a sincronização entre os componentes.

Se executarmos essas especificações π-cálculo no MWB, podemos observar que o sistema não entra em impasse. Da mesma forma fazemos para os componentes `RemoveVowels` e `ShowFile`, que tem suas especificações alteradas da seguinte forma:

$$
\begin{aligned}
&agent\ RemoveVowels(outPhrase) = (\hat{}data2)\ t.\,'outPhrase < data2 >\\
&\qquad .RemoveVowels(outPhrase)\\
&agent\ ShowFile(showPhrase) = showPhrase(data1).t.ShowFile(showPhrase)\\
&agent\ System() = (\hat{}channel1)(RemoveVowels(channel1) \mid ShowFile(channel1))
\end{aligned}
$$

Nesse caso também não são encontrados impasses na verificação usando o MWB. Assim podemos dizer que os comportamentos dos componentes com relação às suas conexões são compatíveis.

8.3.6 Validação de Aderência Comportamental ao Estilo

Além dos componentes da arquitetura compatíveis dois a dois, devemos verificar se cada componente possui um comportamento semelhante ao papel (`role`) definido no estilo que a

arquitetura segue. Desse modo, devemos utilizar tanto a descrição algébrica do componente como a do tipo, e verificar se eles são fracamente bisimilares. Utilizaremos o script de DraX para gerar a especificação algébrica dos tipos `Source`, `Filter` e `Sink` do estilo `Pipeline`. O código a seguir mostra o resultado da aplicação do *script*.

$$
\begin{array}{l}
agent\ Source(out1) = (\hat{}data1)\ t.\ 'out1 < data1 > .Source(out1) \\
agent\ Filter(in1, out1) = (\hat{}data2)\ in1(data1).t.\ 'out1 < data2 > .Filter(in1, out1) \\
agent\ Sink(in1) = in1(data1).t.Sink(in1)
\end{array}
$$

Mesmo estando preparado para a realização da verificação formal, em alguns casos, temos que expandir a especificação das portas (quando utilizamos a representação [m]) para refletir o número de portas do componente com o qual estamos verificando a aderência de estilo. Além disso, devemos renomear as portas dos componentes para in(entrada) e out(saída) para que a verificação seja realizada corretamente pelo MWB. Assim, a nova representação passível de ser verificada com relação a bisimulação fraca para os componentes da arquitetura é a seguinte:

$$
\begin{array}{l}
agent\ CatFile(out1) = (\hat{}data1)\ t.\ 'out1 < data1 > .CatFile(out1) \\
agent\ RemoveVowels(in, out) = (\hat{}data2)\ in(data1).t.'\ out < data2 > . \\
\qquad\qquad\qquad RemoveVowels(in, out) \\
agent\ ShowFile(in) = in(data1).t.ShowFile(in)
\end{array}
$$

Agora basta aplicar o MWB para verificar se a especificação do componente é fracamente bisimilar com a especificação do tipo (`role`) com relação ao estilo. Podemos observar que a bisimilaridade é facilmente alcançada por todos os pares, assim, podemos concluir da aderência comportamental da arquitetura ao estilo `Pipeline`.

8.3.7 Semântica $\mathcal{R}\pi$ e Análise

A especificação $\mathcal{R}\pi$ para os componentes da arquitetura é apresentada a seguir.

$$
\begin{array}{l}
[C_1.out1 \diamond C_2.in, C_2.out \diamond C_3.in] \\
C_1[t.\ 'out1 < data1 > .CatFile(out1)] \\
\mid C_2[in(data1).t.'\ out < data2 >] \\
\mid C_3[in(data1).t.ShowFile(in)]
\end{array}
$$

Podemos observar a facilidade de visualizar a estrutura da aplicação através dessa álgebra. Contudo, como já mencionamos na Seção 6.3, ainda não implementamos ferramenta para a manipulação desse cálculo, assim, resolvemos criar um script de geração de especificações π-cálculo a partir de especificações $\mathcal{R}\pi$ e daí podemos utilizar o MWB para realizar as análises formais. Dessa forma, a especificação arquitetural geral em π-cálculo seria a seguinte:

$$
\begin{array}{l}
agentSys() = (\hat{}channel1, channel2) \\
\qquad (CatFile(channel1) \mid \\
\qquad RemoveVowels(channel1, channel2) \mid \\
\qquad ShowFile(channel2))
\end{array}
$$

Utilizando o MWB podemos verificar facilmente que a arquitetura não apresenta impasses.

8.3.8 Geração de Templates

Como apresentado no Capítulo 7 a geração de templates de implementação é realizada a partir de especificações devidamente validadas e verificadas sintática, estrutural e formalmente. Essa etapa apresenta diversas fases e um conjunto de *scripts* XSL devem ser utilizados. Contudo, por questões de clareza e concisão, nesse capítulo apresentaremos apenas o resultado final da aplicação de todos os *scripts* de DraX para a geração de código.

Escolhemos para esse estudo de caso gerar códigos para CORBA. Assim, inicialmente o arquivo IDL para o sistema como um todo é gerado. Esse arquivo está apresentado no código a seguir.

```
module SystemRemoveVowels{

   interface RemoveVowelsItf {
      void getPhrase (in string phrase);
   };

   interface ShowFileItf {
      void showPhrase (in string phrase);
   };

};
```

Como também ressaltamos no Capítulo 7, utilizamos uma representação XML na geração dos códigos, sendo que diversos códigos são gerados como elementos XML que serão tratados mais adiante por *scripts* XSL específicos, como é o caso dos códigos de serviços CORBA e os códigos de invocação dinâmica. Dessa forma, a estrutura geral onde os códigos de cada componente gerados pelos *scripts* de DraX para essa aplicação estarão contidos é o seguinte:

```
<?xml version="1.0"?>
<javaCODES style="Pipeline.xty">
<javaCODE>
   Template CatFile
</javaCODE>
<javaCODE>
   Template RemoveVowels
</javaCODE>
<javaCODE>
   Template ShowFile
</javaCODE>
<links>
<link>
   <point1 comp="CatFile.aml" port="output">
      <param name="phrase" type="String"/>
   </point1>
   <point2 comp="RemoveVowels.aml" port="getPhrase">
      <param name="phrase" type="String"/>
   </point2>
</link>
<link>
   <point1 comp="RemoveVowels.aml" port="outPhrase">
      <param name="phrase" type="String"/>
   </point1>
   <point2 comp="ShowFile.aml" port="showPhrase">
      <param name="phrase" type="String"/>
   </point2>
</link>
</links>
</javaCODES>
```

No código abaixo temos o template gerado para o componente `CatFile`, que está contido dentro da estrutura apresentada no código anterior.

```
public class CatFile{
   org.omg.CORBA.ORB orb = null;
   org.omg.CosNaming.NamingContext ncRef = null;
   static org.omg.CORBA.Object RemoveVowelsRef = null;
   public static void main(String[] args){
      new CatFile(args);
   }
   CatFile(String[] args){
      orb = org.omg.CORBA.ORB.init(args,null);
      <service context="NamingService" operation="initNamingService"/>
      <service context="NamingService" operation="createName"
               name="RemoveVowels" value="Strings.StringOperations."/>
      <service context="NamingService" operation="resolveName"
               name="RemoveVowels" value="removevowels"/>
   }
   private void output(String phrase){
      <invocation portName="output"/>
   }
}
```

Agora, apresentamos os códigos do template gerado para o componente `RemoveVowels`.

```
public class RemoveVowels extends
            SystemRemoveVowels.RemoveVowelsItfPOA{
   static org.omg.CORBA.ORB orb = null;
   static org.omg.CosNaming.NamingContext ncRef = null;
   static org.omg.PortableServer.POA rootpoa = null;
   static org.omg.CORBA.Object RemoveVowelsRef = null;
   static org.omg.CORBA.Object ShowFileRef = null;
   public static void main(String[] args){
      orb = org.omg.CORBA.ORB.init(args,null);
      <service context="POA" operation="initPOA"/>
      <service context="POA" operation="activatePOA"/>

            RemoveVowels RemoveVowelsObj = new RemoveVowels();

      <service context="POA" operation="generateID"
               value="RemoveVowels"/>
      <service context="NamingService" operation="initNamingService"/>
      <service context="NamingService" operation="createContext"
               name="RemoveVowels" value="Strings.StringOperations."/>
      <service context="NamingService" operation="bindName"
               name="RemoveVowels" value="removevowels"
               contextName="Strings.StringOperations."/>
      System.out.println("RemoveVowels ready and waiting...");
      orb.run();
   }
   RemoveVowels(){
      <service context="NamingService" operation="initNamingService"/>
      <service context="NamingService" operation="createName"
               name="ShowFile" value="Strings.StringOperations."/>
      <service context="NamingService" operation="resolveName"
               name="ShowFile" value="showfile"/>
   }
   public void getPhrase(String phrase){
      <invocation portName="getPhrase"/>
   }
   private void outPhrase(String phrase){
      <invocation portName="outPhrase"/>
   }
}
```

Por fim, temos a seguir o template gerado para o componente `ShowFile`.

```
public class ShowFile extends
            SystemRemoveVowels.ShowFileItfPOA{
   public static void main(String[] args){
      org.omg.CORBA.ORB orb = null;
      org.omg.CosNaming.NamingContext ncRef = null;
      org.omg.PortableServer.POA rootpoa = null;
      org.omg.CORBA.Object ShowFileRef = null;
      orb = org.omg.CORBA.ORB.init(args,null);
      <service context="POA" operation="initPOA"/>
      <service context="POA" operation="activatePOA"/>

         ShowFile ShowFileObj = new ShowFile();

      <service context="POA" operation="generateID"
               value="ShowFile"/>
      <service context="NamingService" operation="initNamingService"/>
      <service context="NamingService" operation="createContext"
               name="ShowFile"value="Strings.StringOperations."/>
      <service context="NamingService" operation="bindName"
               name="ShowFile"value="showfile"
               contextName="Strings.StringOperations."/>

      System.out.println("ShowFile ready and waiting...");
      orb.run();
   }
   ShowFile(){
      System.out.println("ShowFile Working.");
   }
   public void showPhrase(String phrase){
   }
}
```

8.3.9 Geração do Código dos Serviços e de Invocações

Utilizando os códigos anteriores, e aplicando os *scripts* de geração de códigos de Serviços CORBA (`GenerateCORBAServicesCode.xsl`) e de geração de códigos de invocação (`GenerateCORBAInvocations.xsl`), temos códigos gerados para a aplicação que substituem as marcações XML. Para facilitar a visualização suprimimos várias partes do código que não dizem respeito a essa etapa do desenvolvimento, sendo que esses códigos podem ser encontrados no Apêndice A

A seguir temos os códigos dos serviços e de invocação para o componente `CatFile`.

```
public class CatFile{
   ...

   public static void main(String[] args){
      ...
   }
   CatFile(String[] args){

      try{
         org.omg.CORBA.Object obj = orb
                        .resolve_initial_references("NameService");
         ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
      } catch ( org.omg.CORBA.ORBPackage.InvalidName in) {
         System.out.println("InvalidName");
         return ;
      }

   try{
      org.omg.CosNaming.NameComponent[] RemoveVowelsName = {
         new org.omg.CosNaming.NameComponent("Strings",),
         new org.omg.CosNaming.NameComponent("StringOperations",),
         new org.omg.CosNaming.NameComponent("removevowels",)};
      RemoveVowelsRef = ncRef.resolve(RemoveVowelsName);}
   } catch ( java.lang.Exception e ) {
      e.printStackTrace();
      return ;
   }
}

   private void output(String phrase){
      org.omg.CORBA.Request request = RemoveVowelsRef.
               _request("getPhrase");
      request.add_in_arg().insert_string(phrase);
      request.invoke();
   }

}
```

O componente `RemoveVowels`, por ter um código mais complexo, quebramos a especificação em três partes apresentadas a seguir.

```
public class RemoveVowels extends
        SystemRemoveVowels.RemoveVowelsItfPOA{
...
   public static void main(String[] args){
      try{
      rootpoa = org.omg.PortableServer.POAHelper.narrow (
         orb.resolve_initial_references("RootPOA"));
      } catch ( org.omg.CORBA.ORBPackage.InvalidName ex ){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
      }
      try{
         rootpoa.the_POAManager().activate();
      }catch ( org.omg.PortableServer.POAManagerPackage.
                  AdapterInactive ex){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
      }

      RemoveVowels RemoveVowelsObj = new RemoveVowels();

      try{
         RemoveVowelsRef = rootpoa.servant_to_reference(RemoveVowelsObj);
      }catch ( org.omg.PortableServer.POAPackage.ServantNotActive ex ) {
         System.out.println( "Servant Not Active!");
         System.exit( 1 );
      }catch ( org.omg.PortableServer.POAPackage.WrongPolicy wp){
         System.out.println( "Wrong Policy!");
         System.exit( 1 );
      }
      try{
         org.omg.CORBA.Object obj = orb.
               resolve_initial_references("NameService");
         ncRef = omg.CosNaming.NamingContextHelper.narrow(obj);
      }catch ( org.omg.CORBA.ORBPackage.InvalidName in ){
         System.out.println( "InvalidName");
         return;
      }
```

```
try{
   org.omg.CosNaming.NameComponent[] name1 = {new
         org.omg.CosNaming.NameComponent("Strings",)};
   org.omg.CosNaming.NamingContext cntx1 = ncRef.bind_new_context(name1);
   org.omg.CosNaming.NameComponent[] name2 = {new
         org.omg.CosNaming.NameComponent("StringOperations",)};

   org.omg.CosNaming.NamingContext endcntx = ncRef.
               bind_new_context(name2);
   org.omg.CosNaming.NameComponent[] RemoveVowelsName = {new
         org.omg.CosNaming.NameComponent"removevowels",};
   endcntx.rebind(RemoveVowelsName, RemoveVowelsRef);
} catch ( org.omg.CosNaming.NamingContextPackage.AlreadyBound nf){
try{
   org.omg.CosNaming.NameComponent[] RemoveVowelsName = {new
         org.omg.CosNaming.NameComponent("Strings",),
   new org.omg.CosNaming.NameComponent("StringOperations",),
         new org.omg.CosNaming.NameComponent("removevowels",)};

   ncRef.rebind(RemoveVowelsName,RemoveVowelsRef);
}catch(Exception e){}
}catch ( java.lang.Exception e ){
   e.printStackTrace();
   return;
}
   ...
}
```

```
RemoveVowels(){
   try{
      org.omg.CORBA.Object obj = orb.
               resolve_initial_references("NameService");
      ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
   }catch ( org.omg.CORBA.ORBPackage.InvalidName in ){
      System.out.println( "InvalidName");
      return;
   }
   try{
      org.omg.CosNaming.NameComponent[] ShowFileName = {new
         org.omg.CosNaming.NameComponent("Strings",),
      new org.omg.CosNaming.NameComponent("StringOperations",),
      new org.omg.CosNaming.NameComponent("showfile",)};

      ShowFileRef = ncRef.resolve(ShowFileName);
   }catch ( java.lang.Exception e ){
      e.printStackTrace();
      return;
   }
}

public void getPhrase(String phrase){
}

private void outPhrase(String phrase){
   org.omg.CORBA.Request request = ShowFileRef._request("showPhrase");
   request.add_in_arg().insert_string(phrase);
   request.invoke();
}
}
```

Por fim o componente `ShowFile`, cujo código gerado também foi dividido em duas partes, é o seguinte:

```
public class ShowFile extends
        SystemRemoveVowels.ShowFileItfPOA{
   public static void main(String[] args){
      org.omg.CORBA.ORB orb = null;
      org.omg.CosNaming.NamingContext ncRef = null;
      org.omg.PortableServer.POA rootpoa = null;
      org.omg.CORBA.Object ShowFileRef = null;
      orb = org.omg.CORBA.ORB.init(args,null);
      try{
         rootpoa =org.omg.PortableServer.POAHelper.narrow(
         orb.resolve_initial_references("RootPOA"));
      }catch ( org.omg.CORBA.ORBPackage.InvalidName ex){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
   }
      try{
         rootpoa.the_POAManager().activate();
      }catch ( org.omg.PortableServer.POAManagerPackage.
                  AdapterInactive ex){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
      }

      ShowFile ShowFileObj = new ShowFile();

      try{
         ShowFileRef = rootpoa.servant_to_reference(ShowFileObj);
      }catch ( org.omg.PortableServer.POAPackage.ServantNotActive ex){
         System.out.println( "Servant Not Active!");
         System.exit( 1 );
      }
      catch ( org.omg.PortableServer.POAPackage.WrongPolicy wp){
         System.out.println( "Wrong Policy!");
         System.exit( 1 );
      }
      try{
         org.omg.CORBA.Object obj = orb.
               resolve_initial_references("NameService");
         ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
      }catch ( org.omg.CORBA.ORBPackage.InvalidName in){
         System.out.println( "InvalidName");
         return ;
      }
```

```
      try{
         org.omg.CosNaming.NameComponent[] name1 = {
            new org.omg.CosNaming.NameComponent("Strings",)};
         org.omg.CosNaming.NamingContext cntx1 = ncRef.
               bind_new_context(name1);
         org.omg.CosNaming.NameComponent[] name2 = {
            new org.omg.CosNaming.NameComponent("StringOperations",)};
         org.omg.CosNaming.NamingContext endcntx = ncRef.
               bind_new_context(name2);
         org.omg.CosNaming.NameComponent[] ShowFileName = {
            new org.omg.CosNaming.NameComponent("showfile",)};
            endcntx.rebind(ShowFileName,ShowFileRef);
      }catch ( org.omg.CosNaming.NamingContextPackage.AlreadyBound nf){
      try{
         org.omg.CosNaming.NameComponent[] ShowFileName = {
            new org.omg.CosNaming.NameComponent("Strings",),
            new org.omg.CosNaming.NameComponent("StringOperations",),
            new org.omg.CosNaming.NameComponent("showfile",)};
      ncRef.rebind(ShowFileName,ShowFileRef);
      }catch(Exception e){}
   }catch ( java.lang.Exception e){
      e.printStackTrace();
      return ;
   }
   System.out.println("ShowFile ready and waiting...");
   orb.run();
}
ShowFile(){
   System.out.println("ShowFile Working.");
}

public void showPhrase(String phrase){
}

}
```

8.3.10 Implementação da Lógica da Aplicação

Para finalizar a implementação dos componentes devemos implementar a lógica da aplicação. A seguir mostramos as porções de códigos com a implementação realizadas manualmente pelos desenvolvedores (códigos em itálico). Mostramos apenas um esboço da classe e a implementação da lógica da aplicação, sendo que os códigos completos podem ser vistos no Apêndice A.

O componente `CatFile` é apresentado no código abaixo.

```
public class CatFile{
   ...
   public static void main(String[] args){
      ...
   }
   ...
   CatFile(String[] args){
      ...
      while(true){
         System.out.print("Entre Mensagem:");
         byte[] buffer = new byte[30];
         try{
            System.in.read(buffer);
         }catch(Exception e){System.out.println("Error");}
         String line = new String(buffer);
         this.output(line);
       }
   }
   private void output(String phrase){
      ...
   }
}
```

O componente seguinte é o RemoveVowels, apresentado a baixo. Nele, além da inclusão do código no método `getPhrase`, implementamos um novo método (`Remove`), que de fato realiza a remoção das vogais das frases recebidas.

```
public class RemoveVowels extends
      SystemRemoveVowels.RemoveVowelsItfPOA{
   ...
   public static void main(String[] args){
      ...
   }
   RemoveVowels(){
      ...
   }
   public void getPhrase(String phrase){
      String StringFinal = this.Remove(phrase);
      this.outPhrase(StringFinal);
   }
   private void outPhrase(String phrase){
      ...
   }
   private String Remove(String data){
      char[] line = new char[100];
      int cont=0;
      for(int i=0; i<data.length(); i++){
         char str = data.charAt(i);
         if (str=='a' || str=='A' || str=='e' || str=='E' ||
            str=='i' || str=='I' || str=='o' || str=='O' ||
            str=='u' || str=='U'){
                line[cont]=data.charAt(i);
               cont++;
         }
      }
      return(new String(line));
   }
}
```

Por fim apresentamos no código a baixo o componente `ShowFile`.

```
public class ShowFile extends
            SystemRemoveVowels.ShowFileItfPOA{
   public static void main(String[] args){
      ...
   }
   ShowFile(){
      ...
   }
   public void showPhrase(String phrase){
      System.out.println(phrase);
   }
}
```

8.3.11 Execução

Antes de executar a aplicação, o desenvolvedor deve criar os arquivos `.java` de cada classe e incluir o código gerado por DraX. Em seguida, a execução da aplicação se da pela instanciação dos objetos. A ordem a ser adotada é:

1. Objeto ShowFile;
2. Objeto RemoveVowels;
3. Objeto CatFile.

8.4 Estudo de Caso 2: *RemoveVowels* Síncrono

8.4.1 Descrição da Aplicação

Para a eleboração desse segundo estudo de caso vamos utilizar o mesmo problema anterior, contudo vamos realizar a descrição de uma arquitetura com comunicação síncrona entre os componentes. Para esse fim iremos apenas definir um novo estilo arquitetural que seja uma variante de `Pipeline`, que denominaremos de `PipelineSincrono`. Esse novo estilo é definido pela herança do estilo `Pipeline` se introduzido um estilo de comunicação síncrono entre os componentes.

Utilizando as mesmas descrições arquiteturais do estudo de caso 1, iremos trocar o estilo que a arquitetura segue e regerar os códigos. Na geração de código utilizaremos os *scripts* que geram templates em um novo middleware, no caso RMI, apresentado em [113]. Para preservar espaço, não iremos reapresentar os códigos das especificações ArchML, apenas mostraremos o que foi alterado com relação ao estudo de caso anterior. Nesse estudo de caso utilizaremos os passos que apresentamos no estudo de caso 1 que são relevantes para esse aplicação, pois sabemos que alguns deles não devem ser realizadas, visto que estamos apenas readaptando as especificações.

Nesse estudo de caso temos dois objetivos principais: o primeiro é demonstrar a vantagem de se ter uma linguagem exclusiva para a definição de estilos arquiteturais. Nesse sentido e geração de código em outro middleware.

8.4.2 Especificação da Arquitetura/Estilo

Nesse estudo de caso iremos utilizar um novo estilo arquitetural que devemos especificar utilizando Xtyle. Esse novo estilo, que denominaremos de `PipelineSincrono`, é uma derivação simples do estilo `Pipeline`, onde introduziremos uma forma de comunicação síncrona entre os componentes. Segue a especificação Xtyle desse novo estilo.

```
<?xml version="1.0"?>
<xtyle name="PipelineSincrono">
   <document>
      <version num="1.0"/>
      <author name="Cidcley T. de Souza"/>
      <lastUpdate date="2003-07-18"/>
      <comments>
         Estilo Derivado de Pipeline
      </comments>
   </document>
      <uses>
         <style name="Pipeline"href="Pipeline.xty"/>
      </uses>
   <types>
      <type name="Filter"from="Pipeline">
         <ports>
            <in mode="sync"/>
            <out mode="sync"/>
         </ports>
      </type>
      <type name="Source"from="Pipeline">
         <ports>
            <in mode="sync"/>
         </ports>
      </type>
      <type name="Sink"from="Pipeline">
         <ports>
            <out mode="sync"/>
         </ports>
      </type>
   </types>
   <topology>
      <link name="SourceFilter"from="Pipeline"/>
      <link name="SinkFilter"from="Pipeline"/>
      <link name="FilterFilter"from="Pipeline"/>
   </topology>
</xtyle>
```

Para que possamos utilizar esse estilo na arquitetura, devemos referenciá-lo dentro da especificação ArchML. Como aqui só estamos modificando a forma de comunicação, todo a estrutura sintática se mantém preservada, sendo que a verificação de consistências sintáticas e comportamentais tanto do estilo como da arquitetura também matêm preservadas. A seguir apresentamos o código que devemos modificar na descrição da arquitetura apresentada no estudo de caso 1 para referenciar o novo estilo.

```
<style href="PipelineSincrono.sty"/>
```

8.4.3 Geração de Templates

A geração de templates é realizada através de *scripts* específicos para o middleware que escolhemos. Para esse estudo de caso utilizamos *scripts* de geração de código RMI. A seguir apresentamos trechos dos códigos gerados de cada componente da aplicação. É importante

ressaltar que não modificamos a lógica da aplicação com relação ao estudo de caso 1, sendo que apenas *copiamos-e-colamos* as especificações do estudo de caso 1.

Inicialmente geramos os códigos para o componente `CatFile`. Esse componente é um cliente simples, portanto é gerada apenas uma classe denominada `CatFile.java` (segue código a seguir).

```
// CatFile.java

import java.io.*;
import java.rmi.*;
public class CatFile{

 RemoveVowelsItf Remove = null;

 public static void main(String[] args){
  new CatFile(args);
 }
 CatFile(String[] args){
  System.setSecurityManager(new RMISecurityManager());
   try {
    Remove = (RemoveVowelsItf)Naming.
       lookup("rmi://localhost/removevowels");
    System.out.println("Objeto Inverte Localizado.");
    //**
    while(true){
     System.out.print("Entre Mensagem:  ");
     byte[] buffer = new byte[30];
     try{
      System.in.read(buffer);
     }catch(Exception e){System.out.println("Error");}
     String line = new String(buffer);
     this.output(line);
    }
    //**
   }catch ( RemoteException re ) {
    System.out.println( "A Remote Exception");
    System.out.println(  + re );
   }
  }
  private void output(String phrase){
   try{
    Remove.getPhrase(phrase);
   }catch ( RemoteException re ) {
    System.out.println( "A Remote Exception");
    System.out.println(  + re );
   }
  }
}
```

Já o componente `RemoveVowels` é tanto cliente como servidor, dessa forma, temos, além da própria classe RemoveVowels.java a geração das classes `RemoveVowelsItf.java`, que é a interface Java/RMI do serviço oferecido pelo componente e a classe `RemoveVowelsImpl.java` que é a implementação do serviço oferecido pela interface. (códigos a seguir)

```
// RemoveVowelsItf.java

public interface RemoveVowelsItf extends java.rmi.Remote {
   void getPhrase(String phrase) throws java.rmi.RemoteException;
}
```

```
// RemoveVowelsImpl.java

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class RemoveVowelsImpl extends
   UnicastRemoteObject implements RemoveVowelsItf {
 ShowFileItf showfile = null;

 public RemoveVowelsImpl() throws RemoteException{
  super();
 try{
  System.out.println("RemoveVowels Working.");
  showfile = (ShowFileItf)Naming.
     lookup("rmi://localhost/showfile");
 } catch(Exception e){System.err.println("Erro");}
 }

 public void getPhrase(String phrase){
  String StringFinal = this.Remove(phrase);
  this.outPhrase(StringFinal);
 }
 private void outPhrase(String phrase){
  try{
   showfile.showPhrase(phrase);
  } catch(Exception e){System.err.println("Erro");}
 }

 private String Remove(String data){
  char[] line = new char[100];
  int cont=0;
  for(int i=0; i<data.length(); i++){
   char str = data.charAt(i);
   if (str=='a' || str=='A' || str=='e' || str=='E' ||
        str=='i' || str=='I' || str=='o' || str=='O' ||
        str=='u' || str=='U'){
        line[cont]=data.charAt(i);
        cont++;
   }
  }
  return(new String(line));
 }
}
```

```
// RemoveVowels.java

import java.io.*;
import java.rmi.*;
public class RemoveVowels{

 public static void main(String[] args){
  try {
   RemoveVowelsImpl obj = new RemoveVowelsImpl();

   Naming.rebind("rmi://localhost/removevowels", obj);
   System.out.println("RemoveVowels ready and waiting ...");

  }catch ( RemoteException re ) {
   System.out.println( "A Remote Exception");
   System.out.println(  + re );
  }
 }
}
```

Por último, temos o componente `ShowFile` que é um servidor simples. Desse modo, temos a geração, além da classe `ShowFile.java`, das classes `ShowFileItf.java`, com a implementação da interface desse componente e `ShowFileImpl.java`, com a implementação do serviço que ele oferece. (seguem códigos)

```
// ShowFileItf.java

public interface ShowFileItf extends java.rmi.Remote {
   void showPhrase(String phrase) throws java.rmi.RemoteException;
}
```

```
// ShowFileImpl.java

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShowFileImpl extends UnicastRemoteObject
implements ShowFileItf{

   public ShowFileImpl() throws RemoteException{
    super();
    System.out.println("ShowFile Working.");
   }

   public void showPhrase(String phrase){
    System.out.println(phrase);
   }
}
```

```
// ShowFile.java

import java.rmi.*;
import java.rmi.server.*;
public class ShowFile{
 public static void main(String[] args){
  try{
   ShowFileImpl obj = new ShowFileImpl();
   Naming.rebind("rmi://localhost/showfile", obj);
   System.out.println("ShowFile ready and waiting ...");
  }catch ( RemoteException re ) {
   System.out.println( "A Remote Exception");
   System.out.println(  + re );
  }
 }
}
```

8.5 Conclusão

Os estudos de caso apresentados nesse capítulo, embora sejam simples, nos permitem avaliar a aplicabilidade das ferramentas de DraX e seu impacto tanto no que diz respeito ao grau de dificuldade da implementação final gerada como na aplicação didática desse *framework*.

Podemos observar que nos estudos de caso trabalhamos no nível de especificação de arquitetura, sendo que os códigos são gerados de acordo com as especificações utilizando as linguagens ArchML e Xtyle. Onde o trabalho de especificação se dá pela descrição de elementos XML.

A fase de validação sintática/estrutural se dá pela aplicação de esquemas Schematron e devemos retornar às especificações ArchML ou Xtyle caso esses scripts venham a encontrar algum erro. Daí refinamos a especificação até que essas sejam devidamente validadas.

Já a validação comportamental se torna a parte mais complexa no contexto do desenvolvimento arquitetural com DraX. Contudo, automatizamos todos os passos, sendo que o desenvolvedor não terá que manipular as especificações algébricas diretamente. Contudo, é bom observar, que a fase de verificação formal quase nunca é utilizada convencionalmente pela indústria de software, e com o ferramental de DraX podemos extrair os benefícios dessa etapa sem muito esforço.

A geração de código CORBA ou RMI é realizada automaticamente através de *scripts* XSL. Esses *scripts* trabalham com representações intermediárias dos códigos até chegar ao resultado final, que é um único arquivo com o código Java/CORBA ou Java/RMI de todos os objetos devidamente gerados. Por fim, os códigos de cada classe devem ser utilizados para criar arquivos `.java`, podendo então serem compilados e executados.

No segundo estudo de caso mostramos como pode ser vantajoso o fato de se ter uma linguagem específica para a descrição de estilos arquiteturais. Nesse estudo de caso, produzimos um novo estilo arquitetural baseado em um estilo já existente em DraX. Daí, partindo da mesma especificação arquitetural do estudo de caso 1, apenas trocamos a referência ao estilo dentro da especificação da arquitetura para que esse possa utilizar o nosso novo estilo. Então, os novos códigos, agora baseados em RMI, são gerados utilizando a idéia de comunicação síncrona. Observe que não tivemos que realizar nenhuma modificação nas especificações arquiteturais, apenas modificar o estilo utilizado pela arquitetura. Também podemos observar nesse estudo de caso a versatilidade de DraX em gerar códigos para

infraestruturas de middleware diferentes, pois, baseado na mesma especificação arquitetural, bastando apenas ajustar as propriedades dos componentes, geramos códigos para um middleware RMI.

No início dessa tese, realizamos considerações sobre a facilidade do desenvolvimento arquitetural na criação de aplicações distribuídas complexas. Contudo, nesse capítulo de estudos de caso, não colocamos nenhuma aplicação que pudesse mostrar a aplicabilidade de DraX na realização dessa tarefa. Na verdade, em [113], pode ser encontrado um estudo de caso complexo modelado com ArchML e Xtyle. Resolvemos deixar nesse capítulo apenas os estudos de caso que, de forma simplificada, pudessem melhor explicar as principais contribuições científicas desse trabalho. Ressaltando as características de originalidade dessa tese. De fato, a aplicação de DraX na construção de aplicações distribuídas complexas pode ser, de forma generalista, observado nesses estudos de caso. Pudemos observar que depois de realizarmos a descrição ArchML e Xtyle utilizada na aplicação, devemos nos preocupar apenas com a implementação relativa à logica da aplicação. Nesses estudos de caso, que têm uma estrutura arquitetural simplificada por ter um conjunto bem pequeno de objetos, já á observado que não trabalhamos com o desenvolvimento de códigos relativos a infraestrutura de middleware subjacente. Se pensarmos em uma aplicação mais complexa, com dez objetos, por exemplo, teremos uma especificação arquitetural também complexa e uma geração de código ainda mais difícil, que é realizada automaticamente pelas ferramentas de Drax. Assim, podemos concluir que também na criação de aplicações complexas a abordagem de DraX trás grandes facilidades para os desenvolvedores.

Capítulo 9

Conclusão e Trabalhos Futuros

9.1 Conclusões

Ao longo desta tese foram realizadas incursões em diversas sub-áreas da Engenharia de Software e de Sistemas Distribuídos. Vários conceitos desenvolvidos nessas sub-áreas foram agregados para formar o *framework* DraX. De fato, DraX é um agregado de tecnologias que tanto de forma pragmática como de forma didática pode ser aplicado na construção de sistemas distribuídos baseado nas idéias de arquitetura de software e estilos arquiteturais, além de fornecer um ambiente flexível para o auxílio ao ensino dessa disciplina.

Na verdade, para que esse conjunto de soluções tecnológicas fosse concebido, tivemos que realizar diversas avaliações sobre como integrar essas tecnologias de forma linear (no sentido de alocar a ferramenta certa para cada etapa) sem ter que desconsiderar os aspectos conceituais envolvidos e preservando o conteúdo didático de cada etapa.

A seguir apresentamos as principais decisões envolvidas com todas as etapas de desenvolvimento cobertas pelo *framework* DraX.

9.1.1 As Idéias de DraX

Desde sua fase inicial de desenvolvimento, baseado nas idéias descritas em [104], DraX foi imaginado como uma ferramenta de descrição de arquiteturas de software distribuídas sob o prisma de desenvolvimento baseado em grupos de trabalhos distribuídos. De fato, DraX permite a construção de descrições arquiteturais utilizando componentes descritos externamente à arquitetura e possivelmente distribuídos.

Com a elaboração da primeira versão de ArchML [105], observamos que diversos outros aspectos relativos ao desenvolvimento arquitetural poderiam ser incluídos no contexto dessa linguagem. Contudo, resolvemos avançar no sentido de deixar ArchML "enxuta" e permitir a utilização das outras nuances do desenvolvimento arquitetural. Assim pensamos em incluir aspectos de desenvolvimento de estilos arquiteturais. Nesse sentido propomos a linguagem Xtyle [107]. Através de Xtyle, contribuímos com a área de desenvolvimento de estilos arquiteturais no que diz respeito a produção de novos estilos através de refinamentos e herança de descrições arquiteturais de outros estilos pré-existentes além da forma didática de especificar informações apenas sobre estilos.

Tendo sido descritos arquitetura e os estilos arquiteturais, resolvemos dar mais um passo adiante e realizar as etapas de validação das descrições arquiteturais. Nesse sentido, baseado nas decisões de utilizar XML como base de ArchML e Xtyle, encontramos nas gramáticas e nos esquemas XML uma maneira de formalizar as descrições e validar as especificações

tanto de estilos como de arquiteturas. Utilizamos as linguagens XML Schema e Schematron para realizar essa etapa do desenvolvimento. Mesmo tendo sido uma tarefa bastante árdua inicialmente, observamos que a criação de *scripts* de validação seguia uma lógica bem definida. A partir daí, construímos *scripts* XSLT que, tendo como base uma descrição de estilos, gera *scripts* de validação automaticamente. De fato, essa etapa do desenvolvimento com DraX está completamente automatizada, bastando apenas aplicar o *scripts* específico e aguardar que a validação sintática e estrutural seja realizada.

Com a etapa de validação sintática/estrutural garantimos que a arquitetura está válida tanto com relação à relação entre os componentes que a forma quanto com relação ao estilo que essa segue. Contudo, ainda não podíamos garantir o funcionamento de uma possível implementação decorrente dessa especificação. Para que pudéssemos inferir sobre o comportamento da arquitetura resolvemos incluir uma etapa de validação comportamental. Nessa etapa, baseado nos objetivos de DraX de fornecer um ambiente de fácil manipulação, utilizamos DDPs (Diagramas de Descrição de Protocolos), que são Diagramas de Estados com algumas regras de concepção pré-estabalecidos, para descrever o comportamento observável dos componentes com relação às suas portas. Essa idéia foi baseada em diversos outros trabalhos [72, 10, 58, 37, 71] que utilizaram a mesma abordagem para a verificação de arquiteturas de software utilizando álgebras de processos, com o diferencial que nessa tese, desenvolvemos a uma nova álgebra de processos, o cálculo $\mathcal{R}\pi$ [108, 106] que melhor se adapta à descrição de sistemas baseados em arquiteturas.

A etapa seguinte na construção da arquitetura, seria a realização da implementação. Como todas as descrições foram desenvolvidas em XML, a definição de *scripts* XSLT para a geração de códigos seria de certa forma bastante facilitada. Para essa etapa, resolvemos criar *scripts* para a geração de templates Java contendo código para execução sobre CORBA e RMI. Nesse sentido, desenvolvemos diversos *scripts* para a geração de códigos Java/CORBA e Java/RMI baseados nas informações de arquiteturas e estilos.

Como pode ser observado a origem de DraX como um *framework* se deu de forma natural. Na verdade acabamos por produzir um conjunto de etapas relacionadas que permitem a concepção e implementação de arquiteturas de software distribuído sobre middleware orientados a objetos. É válido observar que não utilizamos a palavra "processo de desenvolvimento" ao longo da tese. Isso se deve ao fato de que não tivemos a intenção de produzir um processo formal de desenvolvimento, mas apenas um conjunto de etapas que permita a descrição a implementação de aplicações distribuídas com o auxílio de descrições arquiteturais.

Nessa seção demos apenas uma visão geral sem mais detalhes de todas as ferramentas e etapas envolvidas no desenvolvimento utilizando DraX. A seguir apresentamos com detalhes uma discussão sobre cada uma dessas feramentas, considerando suas vantagens e limitações.

9.1.2 As Linguagens de DraX

Em DraX resolvemos adotar duas gramáticas diferentes, sendo uma utilizada para a especificação de arquiteturas e outra para a especificação de estilos. Ao longo da tese consideramos essas duas gramáticas de forma separada, dando origem a duas linguagens: ArchML e Xtyle. De fato, essa abordagem é diferente da utilizada na literatura de arquitetura de software, onde estilos são definidos a partir de restrições em arquiteturas. Contudo, tendo essa definição realizada separadamente ganhamos na possibilidade de termos mecanismos como herança de especificações de estilos além de uma real separação de interesses, onde podemos nos preocupar apenas em questões arquiteturais em Xtyle. No estudo de caso 2 8.4, pudemos

observar melhor a vantagem dessa abordagem, onde apenas modificamos a referência externa ao estilo em ArchML para termos a geração de código com características diferentes.

A primeira dificuldade que tivemos para a elaboração de ArchML e Xtyle foi a definição do que essas linguagens deveriam ter e de como essas informações deveriam ser organizadas. Para facilitar essa tarefa, verificamos diversas outras ADLs e criamos uma linguagem de padrões [115, 110] para o projeto sintático ADLs, apresentada na Seção 5.2, e baseamos a criação de ArchML e Xtyle nessa linguagem de padrões.

De posse das decisões tomadas na linguagem de padrões, incialmente propomos ArchML. Essa linguagem tem a característica de fornecer um mecanismo simples e eficiente de descrição de arquiteturas de software. Um diferencial de ArchML é que os componentes da arquiteturas são referenciados a partir de descrições externas. O que permite que esses componentes sejam descritos de forma distribuída. Um outro fator importante de ArchML é que resolvemos não utilizar conectores explícitos (adotamos a técnica de arquitetura com configuração *in-line* [9]) pelo fato de termos decidido adotar algum middleware para dar suporte à implementação das arquiteturas [30, 31].

A linguagem Xtyle, desenvolvida em seguida, permite a descrição de estilos arquiteturais. Dentre as diversas características novas de Xtyle, podemos citar como a mais importante a possibilidade de criação de novos estilos a partir da herança de um ou mais estilos pré-definidos. Esses estilos podem ser criados através da restrição e extensão de outros estilos de forma rápida e simples. O entendimento do processo de descrição de estilos, normalmente realizado do modo formal em outras ADL, é bastante acentuado, sendo que Xtyle pode ser aplicado no ensino de estilos arquiteturais em disciplinas de graduação em Engenharia de Software.

9.1.3 Ferramentas de Validação

A etapa de validação de DraX se dá pela aplicação de gramáticas e esquemas de verificação tanto da sintaxe das especificações arquiteturais e de estilos, como pela validação das conexões entre os componentes de uma arquitetura e a gramática de estilos utilizada na descrição da mesma.

Ao longo da tese apresentamos diversos esquemas que desenvolvemos para a validação de arquiteturas e de estilos arquiteturais suportados diretamente por DraX. Apresentamos também *scripts* XSLT para a geração automática de esquemas baseado na descrição de um novo estilo. Assim, o desenvolvedor não precisa desenvolver um esquema de validação para um novo estilo, bastando apenas gerá-lo automaticamente e aplicá-lo na validação de novas arquiteturas que utilizem o novo estilo.

A verificação comportamental em DraX foi completamente baseada em diversos outros trabalhos de validação de arquitetura utilizando álgebras de processos CCS, CSP e também π-cálculo. Particularmente resolvemos abrandar a tarefa formal de descrever especificações algébricas para representar o comportamento observável de um componentes através da aplicação de Diagramas de Estados UML. Esses diagramas de estados podem ser construídos em qualquer ferramenta de desenho UML. Na verdade, como esses diagramas devem seguir certas regras que facilitem a geração de especificações algébricas a partir deles, resolvemos impor restrições na criação desses diagramas e os denominamos de DDPs (Diagramas de Descrição de Protocolos).

As descrições de comportamento tanto de componentes em arquiteturas como em estilos, devem ser incluídas nas especificações ArchML e Xtyle. Essas descrições são incluídas no formato XMI, permitindo a manipulação das especificações a partir de programas que trabalhem com XML. Na verdade, inicialmente tentamos utilizar *scripts* XSLT para

gerar especificações algébricas baseadas nas descrições XMI, contudo, até onde conseguimos trabalhar, esses *scripts* ficaram extremamente complexos e possivelmente não conseguiríamos alcançar todas as metas de geração. Dessa forma, resolvemos desenvolver um programa Java (XMI2PI) para gerar especificações π-cálculo a partir de XMI.

É válido ressaltar que realizamos a geração de especificações π-cálculo simplesmente por que a ferramenta que tínhamos disponível, o MWB (*Mobility WorkBench)*[120], na qual tivemos que realizar diversas adaptações, trabalha com essa álgebra de processos. Contudo, como semântica formal de ArchML desenvolvemos o cálculo $\mathcal{R}\pi$. Esse cálculo, mesmo sem ter uma ferramenta de manipulação direta (esse é um trabalho futuro), pode ser facilmente derivado (implementamos um programa java para isso) de especificações π-cálculo.

A análise formal das especificação geradas segue a estrutura definida em [71, 37, 10, 58, 64], onde as especificações de cada componente conectado são colocadas em uma mesma especificação e testadas quanto a comunicabilidade e a geração de situações de impasses.

Devemos também ressaltar que a versão de π-cálculo utilizada no MWB é síncrona, portanto, atualmente em DraX, ainda não podemos realizar verificações em arquiteturas que sigam estilos que utilizem comunicação assíncrona. A implementação de uma ferramenta que trabalhe com $\mathcal{R}\pi$ em uma versão assíncrona, que não é uma tarefa trivial, deixamos como futuros desdobramentos desse trabalho.

9.1.4 Geração de Código

A geração de código é a última etapa de desenvolvimento com o *framework* DraX. Nessa etapa os códigos relativos à descrição arquitetural já devidamente validada tanto sintática/estruturalmente como comportamentalmente são gerados. É válido ressaltar que até esse momento a descrição tanto da arquitetura como dos estilos é genérica. Isso é, podemos gerar código para qualquer middleware que quisermos. Particularmente, a utilização do elemento `propertySet` na descrição dos componentes da arquitetura, permite a inclusão de anotações sobre características peculiares à plataforma onde o código será gerado. Contudo, esse elemento pode assumir diversos valores, deixando o desenvolvedor livre para elaborar seus *scripts* de geração de código para a plataforma e para a linguagem que mais lhe for conveniente. Uma observação importante é que esse mecanismo foi inspirado pelas anotações da linguagem ACME, que permitem a inclusão de aspectos específicos da algumas ADLs na especificação de arquiteturas.

Para esse trabalho resolvemos detalhar a geração de códigos para infraestruturas de middleware orientados a objetos. Para isso escolhemos utilizar a linguagem Java sobre middleware CORBA. E, em seguida, apresentamos algumas etapas para o desenvolvimento de *scripts* para outras infraestruturas de middleware, na qual utilizamos como exemplo a arquitetura RMI. Escolhemos Java pela portabilidade da linguagem e pelo grande ferramental disponível atualmente para essa plataforma. A adoção de CORBA como middleware principal se deu pelo fato de que essa especificação já está extremamente madura e funcional e, além de ser um padrão bastante aceito na academia, possui implementações em código livre em Java. Além disso, já havíamos realizado alguns trabalhos anteriores relativos à utilização de CORBA como ambiente de suporte a aplicações reconfiguráveis [103, 116]. A arquitetura RMI foi utilizada para mostrar a forma geral de definição de *scripts* para outras infraestruturas de middleware em virtude dessa arquitetura ser utilizada em diversas soluções para computação distribuída em Java.

A geração desses códigos é realizada através de um conjunto de *scripts* XSLT que manipulam as especificações de arquitetura e de estilos e fornecem códigos que são refinados

em uma sequência específica. Na verdade, o processo de geração de código é alcançado através de sub-etapas em que representações intermediárias baseadas em XML são geradas até que tenhamos os códigos finais.

Um contratempo que pode ser percebido nessa abordagem é que e temos que executar muitos esquemas e *scripts* até ter o código final gerado. Contudo, devemos observar que esse fato é proposital, de modo que o desenvolvedor possa ter a real noção de cada etapa de desenvolvimento e para que se possa deixar explícitas essas fases para fins didáticos. Contudo, uma ressalva que temos a fazer, é que todas os *scripts* e esquemas já estão prontos e são facilmente aplicados, pois as tecnologias utilizadas são amplamente conhecidas na indústria de software. Obviamente que uma ferramenta que de fato automatize todas as etapas não é difícil de ser construída, contudo essa tarefa não faz parte do objetivo dessa tese.

9.2 Sumário de Contribuições

As contribuições principais dessa tese estão no fato de termos apresentado uma abordagem baseada em tecnologias e ferramentas que, atualmente são de domínio público, para o desenvolvimento de arquiteturas de softwares e estilos arquiteturais distribuídos sobre infraestruturas de middleware, onde as propriedades induzidas por esses são capturadas e manipuladas automaticamente.

Uma outra contribuição bastante relevante envolve a utilização de DraX como ferramenta no ensino de arquitetura de software. Pudemos observar, com a aplicação de DraX em situações reais, que diversos conceitos completamente abstratos podem ser capturados e explicitados pelas ferramentas de DraX. Tanto no que diz respeito à construção de arquiteturas e de estilos arquiteturais pelas linguagens ArchML e Xtyle, como pelo entendimento fornecidos pelos scripts e esquemas de DraX de etapas mais complexas do desenvolvimento arquitetural relativas à validação das arquiteturas e a realização de análises comportamentais.

Além dessas contribuições maiores, ao longo do tempo em que esse trabalho foi realizado, diversas outras contribuições foram surgindo tanto em Engenharia de Software como em Sistemas Distribuídos, como também na intersecção dessas duas áreas de pesquisa. Em cada uma dessas áreas, diversos campos de pesquisa foram visitados e contribuições pontuais foram desenvolvidas.

9.2.1 Contribuições em Engenharia de Software

Em Engenharia de Software as contribuições dessa tese se desenvolveram nas seguintes sub-áreas:

9.2.1.1 Em Padrões de Software

- Desenvolvemos uma linguagem de padrões de projeto para a descrição sintática de Linguagens de Descrição de Arquitetura. Essa linguagem foi utilizada em DraX como base para a criação das linguagens ArchML e Xtyle.

9.2.1.2 Em Arquitetura de Software

- A Linguagem ArchML, que é uma ADL baseada em XML sendo que sua estrutura sintática é bem mais simples que as ADLs baseadas em XML atualmente disponíveis, visto que não utiliza estruturas como IDs e IDREFs. Além disso, em ArchML temos a

separação da especificação dos componentes da especificação da arquitetura, permitindo a construção de especificações arquiteturais de forma distribuída.

Ainda em ArchML também podemos citar a possibilidade de termos um mesmo componente possuindo diversos papéis e sendo capaz de ser usado em uma arquitetura que utilize um estilo arquitetural misto, ou seja, um componente que seja compatível com mais de um estilo simultaneamente.

- A Linguagem Xtyle, que é uma linguagem para a especificação de estilos arquiteturais. Essa é uma contribuição bastante original e significativa dessa tese na área de arquitetura de software, visto que não há atualmente uma linguagem que forneça mecanismos abstratos para a especificação exclusivamente de estilos arquiteturais e que contemple a idéia de definição de tipos permitidos, das ligações permitidas, das restrições topológicas e da definição de fluxo de controle e de fluxo de dados para os estilo.

  Além disso, realizamos a definição de uma nova taxonomia de estilos que considera a idéia de estilos derivados. Até então a literatura só apresentava a idéia de refinamentos de estilos [39], contudo foi mostrado em [14] a necessidade de se ter estilos compostos. Tratamos esse problema em Xtyle e permitimos a criação de estilos derivados, onde um novo estilo pode ser definido através da composição de outros estilos pré-existentes através da herança de especificações e da definição de restrições sobre as especificações herdadas.

- No ensino de arquitetura de software DraX foi utilizado em uma disciplina de graduação em Engenharia de Software no curso de Sistemas de Informação da Faculdade 7 de Setembro, para o ensino de arquitetura de software e estilos arquiteturais. Realmente pudemos observar que os conceitos formais dessa área, normalmente apresentados em um curso introdutório de forma apenas descritiva, visto que a utilização de ADLs não seria inicialmente viável, puderam ser facilmente entendidos. Os alunos em pouco tempo estavam construindo especificações de estilos arquiteturais e de arquiteturas de software seguindo esses estilos de forma rápida e sem ter que lançar mão de rebuscadas especificações. De fato, foi uma grata surpresa a aderência de DraX na área de educação em engenharia de software, e, com certeza, devemos explorar mais essa faceta desse *framework* em forma de um ambiente de ensino de arquiteturas de software.

9.2.1.3 Em Métodos Formais

- Criação de um cálculo para sistemas baseado em arquiteturas. Apresentamos nessa tese o cálculo $\mathcal{R}\pi$, desenvolvido a partir do π-cálculo e que possui a característica de criar especificações de agentes de forma independentes da aplicação, sendo que através do operador de correlação, podemos definir a estrutura de conexão de complexas arquiteturas sem a necessidade de reescrever os agentes, seguindo a mesma idéia das linguagens de configuração de sistemas distribuídos [66]. Essa também é uma contribuição relevante e original desse trabalho.

9.2.2 Contribuições em Sistemas Distribuídos

9.2.2.1 Em Objetos Distribuídos

- Implementação de aplicações distribuídas complexas sem a necessidade do domínio das arquiteturas como CORBA ou RMI. Com DraX podemos produzir aplicações

distribuídas com complexas interações dificilmente implementadas por quem não possuir uma excelente base conceitual e prática em infraestruturas de middleware orientados a objetos como CORBA e RMI.

- Adoção de estilos na especificação de arquiteturas baseadas em infraestruturas de middleware. Em [95] é mostrado que atualmente a utilização de infraestruturas de middleware como CORBA é realizada apenas como uma ferramenta e não como uma tecnologia, ou seja, não especificamos aplicações pensando no middleware e sim o utilizamos como uma API de programação distribuída. Nesse trabalho de fato usamos as infraestruturas de middleware como uma tecnologia, sendo que aplicamos as idéias de projeto de arquitetura, verificação arquitetural e análise formal em aplicações que serão implementadas em infraestruturas de middleware como CORBA e RMI sem se ater a esse fato.

9.3 Trabalhos Futuros

Como pode ser observado nas contribuições dessa tese, esse trabalho se desdobra em diversas outras sub-áreas tanto da Engenharia de Software como em Sistemas Distribuídos. De fato esse trabalho foi apenas o início de diversos outros trabalhos que podem ser derivados de cada uma das etapas cobertas pelo *framework* DraX. Além disso, o próprio *framework* ainda pode/deve ser melhorado e novas características devem ser incluídas. De forma simplificada, podemos destacar os seguintes trabalhos futuros principais:

- **Implementar um ambiente visual para DraX**
  Ao longo dessa tese utilizamos diversas ferramentas para mostrar as funcionalidades de DraX. De fato a intenção principal era provar que existe suporte para desenvolvimento de aplicações distribuídas baseadas em descrições arquiteturais e de estilos. Contudo, para melhor aplicar DraX, um ambiente visual que abranja todas as etapas de desenvolvimento seria extremamente útil. Esse ambiente deve dar suporte tanto à especificação de arquiteturas e estilos como à criação de *scripts* e esquemas de validação. Além da construção de diagramas DDPs e geração de especificações XMI.

- **Melhorar a base formal de DraX**
  O cálculo $\mathcal{R}\pi$, utilizado nesse trabalho para fornecer a semântica formal de ArchML, deve ser melhor tratado no sentido de se especificar uma teoria de bisimulação que permita o refinamento de arquiteturas a partir da comparação de funcionalidades semânticas. Além disso, a criação de uma axiomatização para $\mathcal{R}\pi$ deve ser providenciada de forma a simplificamos a tarefa de manipulação de especificações nesse cálculo. Uma outra tarefa a ser desenvolvida nesse área seria a criação de uma ferramenta de tratamento de especificações $\mathcal{R}\pi$, nos moldes do *Mobility WorkBench* e que forneça mecanismos de comunicação assíncrona de forma a podermos capturar todas as características de DraX.

- **Produzir um processo de desenvolvimento**
  Como ressaltamos no início desse capítulo, não nos preocupamos em definir um processo formal para a utilização das ferramentas de DraX. Na verdade definimos um conjunto de etapas onde cada tipo de ferramenta pode ser aplicada. A formalização de um processo de software que forneça características como refinamentos deve ser providenciado. De fato, como pudemos observar nos trabalhos relacionados, o framework Arcade

(*ARChitecture-based Analisys and DEsign*)[88], o qual integra a arquitetura de software no RUP. Um trabalho que pode ser realizado nesse contexto seria a aplicação do *framework* DraX na criação dos artefatos das etapas do Arcade.

- **Desenvolver ferramenta para o ensino de arquitetura de software**
  Como ressaltamos nas contribuições dessa tese, observamos a aplicabilidade real de DraX como ferramenta de ensino de arquitetura de software. Sendo realmente proeminente os seus benefícios, particularmente no ensino de estilos arquiteturais. Contudo, esse processo de ensino foi baseado apenas nos *scripts* de DraX. De fato, seria de grande valia o desenvolvimento de uma ferramenta baseada nesses *scripts* e que de alguma forma funcione como base de um sistema tutor para o ensino de arquitetura de software.

A partir desses trabalhos futuros podemos concluir que existem muitos mais esforços a serem realizados e que o *framework* DraX abre um enorme leque de possibilidades para o desenvolvimento de novos experimentos tanto na área de Arquitetura de Software como na área de Sistemas Distribuídos. Contudo, embora nessa tese tenhamos dado um passo adiante no tratamento arquitetural de aplicações distribuídas baseadas em middleware, como afirmado em trabalhos como [36], essa área de pesquisa é bastante promissora e está em fase embrionária e ainda carente de resultados práticos.

Referências Bibliográficas

[1] About SAX. Disponível em: <http://www.saxproject.org>. Acesso em: 10/12/2002.

[2] Jaxen: Java XPath Engine. Disponível em: <http://sourceforge.net/projects/jaxen>. Acesso em: 23/11/2002.

[3] Openccm. Disponível em: <http://www.objectweb.org/openccm/index.html>. Acesso em: 10/11/2002.

[4] The Pros and Cons of XML. Disponível em: <http://www.zapthink.com/reports/proscons-view.html>. Acesso em: 14/12/2002.

[5] The XMI Hackers' Homepage. Disponível em: <http://www.dcs.ed.ac.uk/home/pxs/XMI/>. Acesso em: 23/11/2002.

[6] Aldrich, J. and Chambers, C. and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. In *Proc. International Conference on Software Engineering*, 2002.

[7] Allen, R. and Garlan, D. A Formal Basis for Architectural Connections. *IEEE Transactions on Software Engineering*, 1997.

[8] Bachmann, F. and Bass, L. and Chastek, G. and Donohoe, P. and Peruzzi, F. The Architecture Based Design Method. Technical report, Carnegie Mellon University, 2000. CMU/SEI-2000-TR-001.

[9] Bass, L. and Clements, P. and Kazman, R. *Software Architecture in Practice*. Addison-Wesley, 1998.

[10] Bernardo, M. and Ciancarini, P. and Donatiello, L. On the Formalization of Architectural Types with Process Algebras. In *Proc. of FSE-8, ACM Prees*, 2000.

[11] Bernardo, M. and Ciancarini, P. and Donatiello, L. Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems. In *Proc. of WICSA 2001, IEEE-CS Prees*, 2001.

[12] Berry, G. and Boudol, G. The Chemical Abstract Machine. *Theoretical Computer Science*, 1992.

[13] Binns, P. and Vestal, S. Formal Real-Time Architecture Specification and Analysis. In *10th IEEE Workshop on Real-Time Operating Systems and Software*, 1993.

[14] Bishop, J. and Faria, R. Connectors in Configuration Programming Languages: Are They Necessary ? In *International Workshop on Configurable Distributed Systems*, 1996.

[15] Booch, G. and Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language: User Guide*. Object Technology Series. Addison Wesley, 1999.

[16] Bray, T. and Paoli, J. and Aperberg, C. M. and Maler, E. Extensible Markup Language (xml) 1.0 (Second Edition). Technical report, W3C Recomendation, 2000.

[17] Canal, C. and Pimentel, E. and Troya, J. On the Composition and Extension of Software Systems. In *Proc of FSE'97 FoCBS Workshop*, 1997.

[18] Clark, J. XSL Transformations (XSLT 2.0). 2002. Disponível em: <http://www.w3.org/TR/xslt20>. Acesso em: 10/12/2002.

[19] World Wide Web Consortium. Namespaces in XML. Disponível em: <http://www.w3.org/TR/REC-xml-names>. Acesso em: 10/12/2002, 1999.

[20] World Wide Web Consortium. Xml Path Language (XPath) Version 1.0. Disponível em: <http://www.w3.org/TR/xpath>. Acesso em: 10/12/2002, 1999.

[21] World Wide Web Consortium. Document Object Model (DOM). Disponível em: <http://www.w3.org/DOM>. Acesso em: 10/12/2002, 2001.

[22] World Wide Web Consortium. Extensible Stylesheet Language (XSL) Version 1.0. Disponível em: <http://www.w3.org/TR/xsl>. Acesso em: 10/12/2002, 2001.

[23] World Wide Web Consortium. Xml Linking Language Version 1.0. Disponível em: <http://www.w3.org/TR/xlink>. Acesso em: 10/12/2002, 2001.

[24] World Wide Web Consortium. Xml Pointer Language (XPointer) Version 1.0. Disponível em: <http://www.w3.org/TR/2001/WD-xptr-20010108>. Acesso em: 10/12/2002, 2001.

[25] World Wide Web Consortium. XML Schema Part 0: Primer. Disponível em: <http://www.w3.org/TR/xmlschema-0>. Acesso em: 10/12/2002, 2001.

[26] World Wide Web Consortium. XML Schema Part 2: Datatypes. Disponível em: <http://www.w3.org/TR/xmlschema-0>. Acesso em: 10/12/2002, 2001.

[27] Dashofy, Eric M. and Medvidovic, N. and Taylor, Richard N. Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, 1999.

[28] Dashofy, Eric M. and van der Hoek, A. and Taylor, Richard N. A Highly-Extensible, XML-based Architecture Description Language. In *Proceedings of the Working IEEEIFIP Conference on Software Architectures (WICSA 2001)*, 2001.

[29] Dashofy, Eric M. and van der Hoek, A. and Taylor, Richard N. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, 2002.

[30] Di Nitto, Elisabetta and Rosenblum, David S. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *International Conference on Software Engineering*, pages 13–22, 1999.

[31] Di Nitto, Elisabetta and Rosenblum, David S. On the Role of Style in Selecting Middleware and Underwear. In *Engineering Distributed Object 99, ICSE 99 workshop.*, 1999.

[32] Dummond, Y. and Girardet, D. and Oquendo, F. A Relationship Between Sequence and Statechart Diagrams. Technical report, LLP/CESALP Laboratory - University of Savoy, 2001.

[33] Egyed, A. and Grunbacher, P. and Medvidovic, N. Refinement and Evolution Issues in Bridging Requirements and Architecture: The CBSP Approach. In *Proc. of the From Software Requirements to Architectures Workshop (STRAW 2001)*, 2001.

[34] Egyed, A. and Medvidovic, N. Consistent Architectural Refinement and Evolution using the Unified Modeling Language. In *Proc. of the First Workshop on Describing Software Architecture with UML*, 2001.

[35] Eisenbach, S. and Paterson, R. π-Calculus Semantics for the Concurrent Configuration Language Darwin. In *Hawaii International Conference on Systems Science*, 1993.

[36] Emmerich, W. Software Engineering and Middleware: A Roadmap. *The Future of Software Engineering. ACM Press*, 2002.

[37] Emmerich, W. and Kaveh, N. Model Checking Distributed Objects. In H. Obbink and B. Balzer, editors, *Proceedings of the 4th International Software Architecture Workshop*, Limerick, Ireland, June 2000.

[38] Garlan, D. What is a Style? In *Proc. Dagshtul Workshop on Software Architecture*, 1995.

[39] Garlan, D. Style-Based Refinement for Software Architecture. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW2) and the International Workshop on Multiple Perspectives in Software Development (Viewpoints '96).* ACM Press, 1996.

[40] Garlan, D. Software Architecture: a Roadmap. In ACM Press, editor, *The Future of Software Engineering. Proceedings 22nd International Conference on Software Engineering*, 2000.

[41] Garlan, D. and Allen, R. and Ockerbloom, J. Exploiting Style in Architectural Design Environments. In *Proc. ACM SIGSOFT '94 Symposium on Foundations of Software Engineering*, 1994.

[42] Garlan, D. and Kompanek, A. and Cheng, S. Reconciling the Needs of Architectural Description with Object-Modeling Notations. Disponível em: <http://www-2.cs.cmu.edu/ able/publications/uml01/>. Acesso em: 10/12/2002, 2001.

[43] Garlan, D. and Kompanek. A. and Melton, R. and Monroe, R. Architectural Style: An Object-Oriented Approach. Technical report, Carnegie Mellon University, 1996.

[44] Garlan, D. and Monroe, R. T. and Wile, D. Acme: An Architectural Description Interchange Language. In *In Proceedings of CASCON'97*, 1997.

[45] Garlan, D. and Perry, D. Introduction to the Special Issue on Software Architecture. In *IEEE Transactions on Software Engineering*, volume 21, 1995.

[46] Garlan, D. and Shaw, M. An Introduction to Software Architecture. *In Advances in Software Engineering and Knowledge Engineering*, I, 1993.

[47] Giannakopoulou, D. and Kramer, J. and Cheung, S. C. Analysing the Behaviour of Distributed Systems using Tracta. *Journal of Automated Software Engineering*, 1999.

[48] Gomaa, H. and Wijesekera, D. The Role of UML, OCL and ADLs in Software Architecture. In *Proc. of the First Workshop on Describing Software Architecture with UML*, 2001.

[49] Object Management Group. Model-Driven Architecture. Disponível em: <http://www.omg.org/mda/>. Acesso em: 10/10/2002.

[50] Object Management Group. Corba Component Model - Joint Revised Submission, 1999.

[51] Object Management Group. The Commom Object Request Broker: Architecture and Specification Revision 2.4. In *OMG Technical Document formal/00-11-07*, 2000.

[52] Object Management Group. XML Metadata Language (XMI) Specification 1.1, 2000.

[53] Object Management Group. Commom Object Request Broker Architectuer Specification 3.0.1. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/02-11-01>. Acesso em: 10/12/2002, 2002.

[54] Harel, D. and Naamad, A. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[55] Hoare, C. A. R. Communicating Sequencial Processes. *Series in Computer Science. Prentice-Hall International*, 1985.

[56] Hursch, W. and Lopes, C. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, 1995.

[57] Inverardi, P. and Muccini, H. and Pelliccione, P. Checking Consistency Between Architectural Models Using SPIN. In *Proc. of the From Software Requirements to Architectures Workshop (STRAW 2001)*, 2001.

[58] Inverardi, P. and Uchitel, S. Proving Deadlock Freedom in Component-Based Programming. In *Proceedings FASE 2001*, LNCS 2029, Genova, April 2001.

[59] Jacobson, I. and Griss, M. and Jonsson, P. *Software Reuse - Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[60] Java Soft. Remote Method Invocation (RMI). Disponível em: <http://java.sun.com/products/jdk/rmi/>. Acesso em: 04/07/2003.

[61] Jelliffe, R. The Schematron Assertion Language 1.5. Disponível em: <http://www.ascc.net/xml/resource/schematron/schematron.html>. Acesso em: 10/12/2002.

[62] Justo, G. R. R. and Cunha, P. R. F. Programming Distributed Systems with Configuration Languages. In *International Workshop on Configurable Distributed Systems*, 1992.

[63] Justo, G. R. R. and Cunha, P. R. F. Framework for Developing Extensible and Reusable Parallel and Distributed Applications. In *IEEE Conf. on Algorithms and Architectures for Parallel Processing*, 1996.

[64] Kaveh, N. and Emmerich, W. Deadlock Detection in Distributed Object Systems. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 44–51, Vienna, Austria, April 2001.

[65] Kiczales, G. Aspect-Oriented Programming. Technical report, Xeroz, Palo Alto REsearch CenterCarnegie Mellon University, 1997. PARC Technical Report.

[66] Kramer, J. Configuration Programming - A Framework for the Development of Distributable Systems. In *The IEEE International Conference on Computer Systems and Software Engineering*, 1990.

[67] Lago, P. and Falcarin, P. Uml Requirements for Distributed Software Architectures. In *Proc. of the First Workshop on Describing Software Architecture with UML*, 2001.

[68] Ler, C. and Rosenblum, D. UML Component Diagrams and Software Architecture - Experiences from the Wren Project. In *1st ICSE Workshop on Describing Software Architecture with UML*, 2001.

[69] Luckham, D. C. and Augustin, L. M. and Kenny, J. J. and Veera, J. and Bryan, D. and Mann, W. Specification and Analysis of System Architecture Using Rapide. In *IEEE Transactions on Software Engineering*, 1995.

[70] Luer, C. and Rosenblum, D. Uml Component Diagrams and Software Architecture - Experiences from WREN Project. In *Proc. of the First Workshop on Describing Software Architecture with UML*, 2001.

[71] Magee, J. Behavioral Analysis of Software Architectures using LTSA. In *Proceedings of the 21st International Conference on Software Engineering*, pages 634–637, Los Angeles, California, United States, May 1999.

[72] Magee, J. and Dulay, N. and Eisenbach, S. and Kramer, J. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, 1995.

[73] Magee, J. and Dulay, N. and Kramer, J. Structuring Parallel and Distributed Programs. In *Proceedings of The IEEE International Workshop on Configuring Distributed Systems*, March 1992.

[74] Magee, J. and Tseng, A, and Kramer J. Composing distributed objects in CORBA. In *Proceedings of the Third International Symposium on Autonomous Decentralized Systems*, pages 257–63, Berlin, Germany, 9–11 1997. IEEE.

[75] Medvidovic, N. and Oreizy, P. and Robbins, J. E. and Taylor, R. N. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *In Proceedings of ACM SIGSOFT (Symposium on Foundations of Software Engineering)*, 1996.

[76] Medvidovic, N. and Rosenblum, D. Domains of Concerns in Software Architectures and Architectures Description Languages. In *USENIX Conference on Domain-Specific Languages*, 1997.

[77] Medvidovic, N. and Rosenblum, D. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Proc. of the First IFIP Working Conference on Software Architecture (WICSA1)*, 1999.

[78] Medvidovic, N. and Taylor, R. Separating Fact From Fiction in Software Architecture. In *International Software Architecture Workshop (ISAW3)*, 1998.

[79] Medvidovic, N. and Taylor, R. A Classification and Comparison Framework for Architecture Description Languages. In *IEEE Transactions on Software Engineering*, volume 26, 2000.

[80] Medvidovic, N. and Taylor, R. N. Architecture Description Languages. In *Software Engeneering ESEC/FSE'97*, 1997.

[81] Mendes, A. *Arquitetura de Software - Desenvolvimento Orientado para Arquitetura.* Editora Campus, 2002.

[82] Meszaros, G. Archi-Patterns - A Process Pattern Language for Defining Architectures. In *In Proceedings of PloP97*, 1997.

[83] Microsoft. Distributed Component Object Model (DCOM). Disponível em: <http://www.microsoft.com/com/tech/DCOM.asp>. Acesso em: 04/07/2003.

[84] Microsoft. Microsoft Message Queuing (MSMQ). Disponível em: <http://www.microsoft.com/windows2000/technologies/communications/msmq/default.asp>. Acesso em: 04/07/2003.

[85] Mikk, E. and Lakhnech, Y. and Petersohn, C. and Siegel, M. On Formal Semantics of Statecharts as supported by Statemate. *2nd BCS-FACS Northern Formal Methods Workshop. Springer-Verlag*, 1997.

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

[87] Milner, R. and Parrow, J. and Walker, D. A Calculus of Mobile Processes, parts I and II. *Journal of Informatin and Computation*, 1992.

[88] Moraes, M. Um Framework de Anáise e Projeto Baseado em Arquitetura de Software. *Dissertação de Mestrado. Universidade Federal de Pernambuco*, 2002.

[89] Orfali, R. and Harkey, D. *Client/Server Programming with Java and CORBA*. John Wiley & Sons Inc., 1998.

[90] Perry, D. and Alexander, L. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[91] Poole, J. Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *Proc. Workshop on Metamodeling and Adaptative Object Models*, 2001.

[92] The Apache XML Project. Xalan-Java version 2.4.1. Disponível em: <http://xml.apache.org/xalan-j/index.html>. Acesso em: 10/12/2002.

[93] Rausch, A. Toward a Software Architecture Specification Language based on UML and OCL. In *Proc. of the First Workshop on Describing Software Architecture with UML*, 2001.

[94] Rausch, A. Towards a Software Architecture Specification Language based on UML and OCL. In *1st ICSE Workshop on Describing Software Architecture with UML*, 2001.

[95] Razouk, R. Where Does Architecture End and Technology Begin ? In *Workshop on Evaluating Software Architectural Solutions 2000 (WESAS 2000)*, 2000.

[96] Schmerl, B. xACME: CMU Acme Extension to xArch. Disponível em: <http://www-2.cs.cmu.edu/ acme/pub/xAcme/guide.pdf>. Acesso em: 10/08/2002.

[97] Schmidt, D. and Vinoski, S. Comparing Alternative Client-side Distributed Programming Techniques. *SIGS C++ Report Magazine*, May 1995.

[98] Selic, B. On Modeling Architectural Structures with UML. In *Proc. of the First Workshop on Describing Software Architecture with UML*, 2001.

[99] Selic, B. On Modeling Architectural Structures with UML. In *1st ICSE Workshop on Describing Software Architecture with UML*, 2001.

[100] Shaw, M. and Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, 1996.

[101] Shaw, M. and DeLine, R. and Klein, D.V. Abstractions for Software Architecture and Tools for Support Them. In *IEEE Trans on Software Engineering*, 1995.

[102] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[103] Souza, Cidcley T. de. Um Ambiente de Desenvolvimento Orientado à Configuração utilizando Objetos Distribuídos. *Dissertação de Mestrado. Universidade Federal do Ceará*, 1996.

[104] Souza, Cidcley T. de and Cunha, Paulo R. F. Desenvolvimento Cooperativo de Aplicações Distribuídas Reconfiguráveis. In *Anais do XX Congresso Nacional da Sociedade Brasileira de Computação*, Curitiba, PR, 2000.

[105] Souza, Cidcley T. de and Cunha, Paulo R. F. Especificando Arquiteturas de Software em XML. In *XXVII Conferêcia Latino-Americana de Informáica*, Mérida, Venezuela, 2001.

[106] Souza, Cidcley T. de and Cunha, Paulo R. F. An Algebraic Approach for Specifying Reconfigurable Systems. In *Proc. International Information Technology Symposium. IEEE Press.*, 2002.

[107] Souza, Cidcley T. de and Cunha, Paulo R. F. Arquitetura e Comportamento na Descrição de Sistemas de Software. In *Simpósio Internacional de Tecnologias da Informação*, Florianópolis, SC, 2002.

[108] Souza, Cidcley T. de and Cunha, Paulo R. F. A Behavioral-Centric Approach for Specifying Component-Based Systems. In *Proc. XV International Conference on Systems Engineering (ICSeng 2002)*, Las Vegas, USA, 2002.

[109] Souza, Cidcley T. de and Cunha, Paulo R. F. Reusing Formal Specification of Components. In *Proc. 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, MIT, Cambridge, USA, 2002.

[110] Souza, Cidcley T. de and Cunha, Paulo R. F. Uma Linguagem de Padrões para o Projeto Sintático de Linguagens de Descrição de Arquitetura. *Revista do IME/UERJ*, 2002. ISSN:1413-9014.

[111] Souza, Cidcley T. de and Cunha, Paulo. R. F. A Calculus for Reconfigurable Componente-Based Systems. In *XXIX Latin-American Conference on Informatics. Accepted for Publication.*, La Paz, Bolivia, 2003.

[112] Souza, Cidcley T. de and Cunha, Paulo. R. F. CORBA Services: Conceitos e Implementação. *Relatório Técnico. Universidade Federal de Pernambuco*, 2003.

[113] Souza, Cidcley T. de and Cunha, Paulo. R. F. Descrição de Arquiteturas e Estilos em DraX. *Relatório Técnico. Universidade Federal de Pernambuco*, 2003.

[114] Souza, Cidcley T. de and Cunha, Paulo. R. F. Especificação de Estilos em Arquiteturas de Software. In *XXIX Conferência Latino-Americana de Informática. Aceito para Publicação.*, La Paz, Bolivia, 2003.

[115] Souza, Cidcley T. de and Cunha, Paulo R. F. and Souza, Jerffeson T. de. Uma Linguagem de Padrões para o Projeto Sintático de Linguagens de Descrição de Arquitetura. In *Conferência Latino-Americana de Linguagens de Padrões para Programação*, Rio de Janeiro, RJ, 2001.

[116] Souza, Cidcley T. de and Oliveira, Mauro. Abaco : Um Ambiente de Desenvolvimento Baseado em Objetos Distribuidos. In *XIII Simposio Brasileiro de Engenharia de Software*, Maringá, PR, April 1998.

[117] Souza, Cidcley T. de and Souza, M. Fátima C. and Cunha, Paulo. R. F. Álgebra de Processos na Especificação Semântica de Arquitetura de Software. In *XXIX Conferência Latino-Americana de Informática. Aceito para Publicação.*, La Paz, Bolivia, 2003.

[118] Storrle, H. Turning UML-Subsystems into Architectural Units. In *1st ICSE Workshop on Describing Software Architecture with UML*, 2001.

[119] Tidwell, D. *XSLT*. O'Reilly, 2001.

[120] Victor, B. The Mobility WorkBench Users' Guide. Technical report, Department of Computer Systems, Uppsala University, 1995.

Apêndice A

Códigos dos Estudos de Caso

Nesse apêndice apresentaremos os códigos finais gerados pelas ferramentas do ToolKit DraX para os Estudo de Caso 1, baseada em Java/CORBA, apresentados no Capítulo 8.

A.1 Arquivo IDL

```
module SystemRemoveVowels{

   interface RemoveVowelsItf {
      void getPhrase (in string phrase);
   };

   interface ShowFileItf {
      void showPhrase (in string phrase);
   };
};
```

A.2 CatFile.java

```
public class CatFile{
   org.omg.CORBA.ORB orb = null;
   org.omg.CosNaming.NamingContext ncRef = null;
   static org.omg.CORBA.Object RemoveVowelsRef = null;

   public static void main(String[] args){
      new CatFile(args);
   }

   CatFile(String[] args){
      orb = org.omg.CORBA.ORB.init(args,null);
      try{
         org.omg.CORBA.Object obj = orb.
                  resolve_initial_references("NameService");
         ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
      }catch ( org.omg.CORBA.ORBPackage.InvalidName in){
         System.out.println( "InvalidName");
         return ;
      }
      try{
         org.omg.CosNaming.NameComponent[] RemoveVowelsName = {
               new org.omg.CosNaming.NameComponent("Strings",),
               new org.omg.CosNaming.NameComponent("StringOperations",),
               new org.omg.CosNaming.NameComponent("removevowels",)};
         RemoveVowelsRef = ncRef.resolve(RemoveVowelsName);
      }catch ( java.lang.Exception e ){
         e.printStackTrace();
         return ;
      }

      while(true){
         System.out.print("Entre Mensagem:  ");
         byte[] buffer = new byte[30];
         try{
            System.in.read(buffer);
         }catch(Exception e){System.out.println("Error ");}

         String line = new String(buffer);
         this.output(line);
      }
   }


    private void output(String phrase){
       org.omg.CORBA.Request request = RemoveVowelsRef._request("getPhrase");
       request.add_in_arg().insert_string(phrase);
       request.invoke();
    }
 }
```

A.3 RemoveVowels.java

```
public class RemoveVowels extends
        SystemRemoveVowels.RemoveVowelsItfPOA{

   static org.omg.CORBA.ORB orb = null;
   static org.omg.CosNaming.NamingContext ncRef = null;
   static org.omg.PortableServer.POA rootpoa = null;
   static org.omg.CORBA.Object RemoveVowelsRef = null;
   static org.omg.CORBA.Object ShowFileRef = null;

   public static void main(String[] args){
      orb = org.omg.CORBA.ORB.init(args,null);
      try{
         rootpoa = org.omg.PortableServer.POAHelper.narrow(orb.
               resolve_initial_references("RootPOA"));
      }catch ( org.omg.CORBA.ORBPackage.InvalidName ex){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
      }
      try{
         rootpoa.the_POAManager().activate();
      }catch ( org.omg.PortableServer.POAManagerPackage.AdapterInactive ex){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
      }

      RemoveVowels RemoveVowelsObj = new RemoveVowels();
```

```
try{
   RemoveVowelsRef = rootpoa.servant_to_reference(RemoveVowelsObj);
}catch ( org.omg.PortableServer.POAPackage.ServantNotActive ex ){
   System.out.println( "Servant Not Active!");
   System.exit( 1 );
}catch ( org.omg.PortableServer.POAPackage.WrongPolicy wp ){
   System.out.println( "Wrong Policy!");
   System.exit( 1 );
}
try{
   org.omg.CORBA.Object obj = orb.
         resolve_initial_references("NameService");
   ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
}catch ( org.omg.CORBA.ORBPackage.InvalidName in) {
   System.out.println( "InvalidName");
   return ;
}
try{
   org.omg.CosNaming.NameComponent[] name1 = {
      new org.omg.CosNaming.NameComponent("Strings",)};
   org.omg.CosNaming.NamingContext cntx1 = ncRef.
            bind_new_context(name1);

   org.omg.CosNaming.NameComponent[] name2 = {
      new org.omg.CosNaming.NameComponent("StringOperations",)};
   org.omg.CosNaming.NamingContext endcntx = ncRef.
            bind_new_context(name2);

   org.omg.CosNaming.NameComponent[] RemoveVowelsName = {
      new org.omg.CosNaming.NameComponent("removevowels",)};

      endcntx.rebind(RemoveVowelsName, RemoveVowelsRef);
}catch ( org.omg.CosNaming.NamingContextPackage.AlreadyBound nf ){
   try{
      org.omg.CosNaming.NameComponent[] RemoveVowelsName = {
         new org.omg.CosNaming.NameComponent("Strings",),
         new org.omg.CosNaming.NameComponent("StringOperations",),
         new org.omg.CosNaming.NameComponent("removevowels",)};

         ncRef.rebind(RemoveVowelsName, RemoveVowelsRef);
   }catch(Exception e){}
}
```

```
   catch ( java.lang.Exception e ){
      e.printStackTrace();
      return ;
   }
   System.out.println("RemoveVowels ready and waiting ...");
   orb.run();
}

RemoveVowels(){
   try{
      org.omg.CORBA.Object obj = orb.
           resolve_initial_references("NameService");
      ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
   }
   catch ( org.omg.CORBA.ORBPackage.InvalidName in){
      System.out.println( "InvalidName");
      return ;
   }
   try{
      org.omg.CosNaming.NameComponent[] ShowFileName = {
         new org.omg.CosNaming.NameComponent("Strings",),
         new org.omg.CosNaming.NameComponent("StringOperations",),
         new org.omg.CosNaming.NameComponent("showfile",)};
         ShowFileRef = ncRef.resolve(ShowFileName);

   }catch ( java.lang.Exception e ){
      e.printStackTrace();
      return ;
   }
}

public void getPhrase(String phrase){
   String StringFinal = this.Remove(phrase);
   this.outPhrase(StringFinal);
}

private void outPhrase(String phrase){
   org.omg.CORBA.Request request = ShowFileRef._request("showPhrase");
   request.add_in_arg().insert_string(phrase);
   request.invoke();
}
```

```
    private String Remove(String data){

       char[] line = new char[100];
       int cont=0;
       for(int i=0; i<data.length(); i++){
          char str = data.charAt(i);
          if (str=='a' || str=='A' || str=='e' || str=='E' ||
             str=='i' || str=='I' || str=='o' || str=='O' ||
             str=='u' || str=='U'){
             line[cont]=data.charAt(i);
             cont++;
          }
       }
       return(new String(line));
    }
 }
```

A.4 ShowFile.java

```
public class ShowFile extends
        SystemRemoveVowels.ShowFileItfPOA{

   public static void main(String[] args){
      org.omg.CORBA.ORB orb = null;
      org.omg.CosNaming.NamingContext ncRef = null;
      org.omg.PortableServer.POA rootpoa = null;
      org.omg.CORBA.Object ShowFileRef = null;

      orb = org.omg.CORBA.ORB.init(args,null);

      try{
         rootpoa = org.omg.PortableServer.POAHelper.narrow(orb.
               resolve_initial_references("RootPOA"));
      }catch ( org.omg.CORBA.ORBPackage.InvalidName ex ){
         System.out.println( "Couldn't find RootPOA!");
         System.exit( 1 );
      }
```

```
try{
   rootpoa.the_POAManager().activate();
}catch ( org.omg.PortableServer.POAManagerPackage.AdapterInactive ex){
   System.out.println( "Couldn't find RootPOA!");
   System.exit( 1 );
}

ShowFile ShowFileObj = new ShowFile();

try{
   ShowFileRef = rootpoa.servant_to_reference(ShowFileObj);
}catch ( org.omg.PortableServer.POAPackage.ServantNotActive ex){
   System.out.println( "Servant Not Active!");
   System.exit( 1 );
}catch ( org.omg.PortableServer.POAPackage.WrongPolicy wp){
   System.out.println( "Wrong Policy!");
   System.exit( 1 );
}

try{
   org.omg.CORBA.Object obj = orb.
         resolve_initial_references("NameService");
   ncRef = org.omg.CosNaming.NamingContextHelper.narrow(obj);
}catch ( org.omg.CORBA.ORBPackage.InvalidName in ){
   System.out.println( "InvalidName");
   return ;
}

try{
   org.omg.CosNaming.NameComponent[] name1 = {
      new org.omg.CosNaming.NameComponent("Strings",)};
   org.omg.CosNaming.NamingContext cntx1 = ncRef.
         bind_new_context(name1);

   org.omg.CosNaming.NameComponent[] name2 = {
      new org.omg.CosNaming.NameComponent("StringOperations",)};
   org.omg.CosNaming.NamingContext endcntx = ncRef.
         bind_new_context(name2);

   org.omg.CosNaming.NameComponent[] ShowFileName = {
      new org.omg.CosNaming.NameComponent("showfile",)};

   endcntx.rebind(ShowFileName, ShowFileRef);
```

```
      }catch ( org.omg.CosNaming.NamingContextPackage.AlreadyBound nf ){
         try{
            org.omg.CosNaming.NameComponent[] ShowFileName = {
               new org.omg.CosNaming.NameComponent("Strings",),
               new org.omg.CosNaming.NameComponent("StringOperations",),
               new org.omg.CosNaming.NameComponent("showfile",)};
            ncRef.rebind(ShowFileName, ShowFileRef);
         }catch(Exception e){}
      }catch ( java.lang.Exception e ){
         e.printStackTrace();
         return ;
      }

      System.out.println("ShowFile ready and waiting ...");
      orb.run();
   }

   ShowFile(){

      System.out.println("ShowFile Working.");
   }

   public void showPhrase(String phrase){
      System.out.println(phrase);
   }
}
```