Um Compilador para a ADL Cbabel

Marcelo Bruno de Carvalho
DESC/FEN/UERJ
marc@ime.uerj.br

Alexandre Sztajnberg
DICC/IME/UERJ
alexszt@ime.uerj.br

Abstract

This paper presents the project of a compiler for the Architecture Description Language (ADL) CBabel, developed as part of the R-RIO framework. This compiler verifies software architecture descriptions, and generates appropriate files to integrate the compiled architecture into the framework's execution environment. These files contain configuration instructions, for loading the architecture, and a repository with the architecture meta-level information for runtime consulting. It is also presented a repository explorer, developed for viewing the repository meta-level information data.

Palavras Chaves: Compilador, arquiteturas de software, ADL, reconfiguração, reflexão.

1. Introdução

Este artigo apresenta o projeto de um compilador para a Linguagem de Descrição de Arquitetura de *Software* (ADL) CBabel, elemento do *framework* R-RIO [2]. Através de CBabel, requisitos funcionais e não-funcionais da arquitetura de *software* de uma aplicação podem ser descritos bem como seu mapeamento em artefatos de *software*. Em um estágio seguinte a arquitetura pode ser carregada para

execução e gerenciada dinamicamente.

Para que a linguagem CBabel seja eficazmente utilizada no contexto de R-RIO, é importante a disponibilidade de um compilador. Caso contrário, o usuário seria obrigado a analisar a arquitetura de *software* descrita em CBabel, e alimentar o ambiente de execução, manualmente.

O compilador desenvolvido verifica a consistência de descrições de arquiteturas de *software* e gera, como resultado, arquivos para integrar a arquitetura descrita ao ambiente de execução do *framework*. Estes arquivos contém instruções de configuração, para a carga da arquitetura, e um repositório contendo informações de meta-nível da arquitetura processada, que pode ser consultado durante sua execução com a finalidade de reconfigurá-la.

O restante do texto está estruturado da seguinte forma. Inicialmente, situa-se este trabalho em um contexto, examinado-se os fundamentos do *framework* R-RIO. Em seguida é apresentada a arquitetura do compilador. Logo após, detalhes da implementação do compilador são discutidos. Na seqüência, um exemplo ilustra a utilização do mesmo. Finalmente são apresentados os resultados do projeto e nossas observações finais.

2. R-RIO

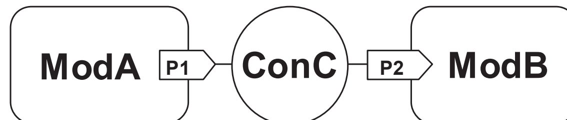
O *framework* R-RIO [3] possibilita a descrição, configuração, execução e manutenção de arquiteturas de *software*. R-RIO define uma linguagem para a descrição de arquiteturas de *software* (ADL) chamada CBabel. CBabel permite a descrição de aplicações a partir de componentes de *software* e suas interligações (aspectos funcionais) e, adicionalmente, seus aspectos não-funcionais como a distribuição, coordenação e qualidade de serviço (*quality of service* – QoS) [4]. Arquiteturas descritas em CBabel são carregadas, configuradas e executadas em um ambiente de suporte chamado Configurador. Aplicações executando sobre o Configurador admitem reconfiguração, mesmo estando em operação [1].

2.1 CBabel

Arquiteturas de *software* descritas em CBabel são baseadas em três elementos. Os **módulos** representam os componentes funcionais da arquitetura. Os **conectores** interligam e intermediam a interação entre módulo e encapsulam aspectos não-funcionais. As **portas** que representam os pontos de interação entre módulos e conectores e explicitam a interface destes elementos. Na figura 1(a) apresentamos a descrição de uma arquitetura simples e na figura 1(b) sua representação esquemática. Nas linhas 1-8 são descritas as classes de módulo **ModuloA** e **ModuloB.** Similarmente, descreve-se a classe de conector **ConnectorC** (linhas 9-11**)**. A configuração da arquitetura é descrita, indicando-se a criação de instâncias dos módulos e interligando os mesmos através do conector (linhas 12-14).

```
1    module ModuloA{
2        ...
3        out port P1;
4    }
5    module ModuloB{
6        ...
7        in port P2;
8    }
9    connector ConnectorC{
10       ...
11   } ConC;
12   instantiate ModuloA as ModA;
13   instantiate ModuloB as ModB;
14   link ModA to ModB by ConC;
```

(a)



(b)

Figura 1. (a) Listagem em Cbabel (b) representação da arquitetura.

CBabel inclui declarações (*start, stop, instantiate, link, block, resume*), que permitem a (re)configuração de um arquitetura. É possível também descrever componentes por composição, o que facilita a gerência de aplicações complexas. Além disso, contratos de interação, coordenação e QoS também podem ser descritos [2] em CBabel.

2.2 *Middleware* de R-RIO

O *middleware* de R-RIO, chamado de Configurador, é composto por um conjunto de elementos de software, que são distribuídos por nós de processamento onde módulos e conectores são instanciados e gerenciados. Este *middleware* utiliza os dados fornecidos pelo compilador de descrições CBabel (arquivo de comandos de carga e repositório de meta-nível) para criar imagens executáveis das arquiteturas. Ele também pode receber comandos ou *scripts* de configuração para reconfigurar as arquiteturas.

A figura 2 representa um diagrama dos elementos do *middleware*. Nele existe uma **API de configuração** que pode ser ativada diretamente, ou através de comandos submetidos ao **interpretador de comandos de configuração**. A **API de configuração** então dispara uma solicitação que é recebida pelo **gerente de configuração global**. Este por sua vez encaminha a solicitação ao **gerente de configuração local** do nó onde serão processados os comandos. Finalmente o comando é executado pelo **configurador executivo** do nó. Após cada reconfiguração das informações de meta-nível são sempre atualizadas pelos gerentes de configuração.

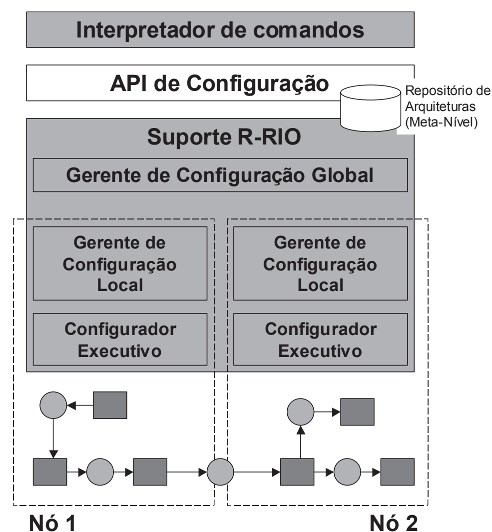


Figura 2. *Middleware* de R-RIO.

No modelo apresentado o gerente de configuração tem uma implementação centralizada. Também poderia ser feito de uma forma distribuída, utilizando diferentes instâncias de gerentes de configuração global para controlar diferentes partes de uma arquitetura em execução.

É importante ressaltar que o suporte R-RIO utiliza reflexão arquitetural através do repositório de meta-nível para executar devidamente os comandos de configuração (e reconfiguração). Por isso estes procedimentos são executados de maneira segura e fica mais evidente a importância da consistência do repositório.

3. Arquitetura do Compilador

A arquitetura do compilador foi desenvolvida com o objetivo de compilar descrições CBabel e preparar a

integração destas informações com o Configurador. Essa arquitetura será apresentada nas quatro seções a seguir:

- Entrada e Saída do Compilador;
- Processamento Interno;
- Modelo do Repositório de Dados de Meta-Nível;
- Construções CBabel Traduzidas pelo Compilador.

3.1 Entrada e Saída do Compilador

No projeto do compilador, uma descrição de arquitetura de *software* em CBabel é fornecida como entrada. O compilador processa esta descrição e, como resultado, fornece um arquivo de relatório, um arquivo de comandos de configuração e um arquivo de repositório de meta-nível (figura 3).

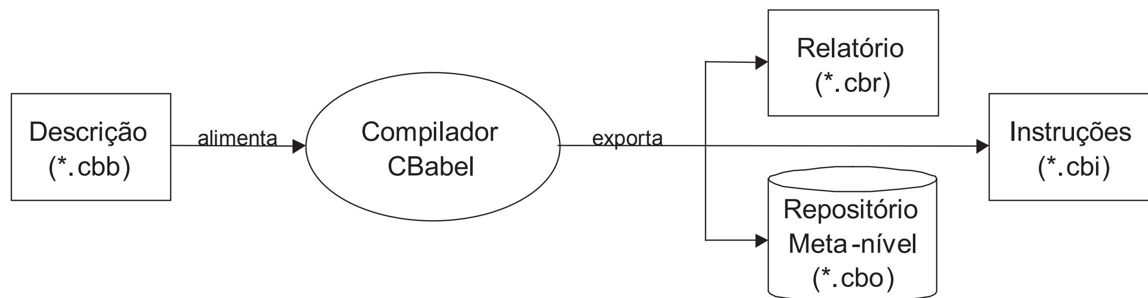


Figura 3. Entrada e saída de dados para o compilador CBabel.

O arquivo de relatório permite a conferência dos problemas encontrados pelo compilador, validando ou não a utilização dos demais arquivos para alimentar o Configurador.

O arquivo de configuração contém os comandos de configuração indicando como deve ser realizada a carga da arquitetura. Este arquivo é utilizado quando se deseja colocar a aplicação em execução.

O repositório de meta-nível contém um objeto que armazena os dados dos aspectos funcionais e não-funcionais da descrição de arquitetura processada. Este objeto é serializado a partir de uma classe Java, utilizada na construção da árvore de *parsing* no processo de compilação. Esta é uma parte importante da solução proposta para oferecer à aplicação a capacidade de inferir sobre sua própria arquitetura no momento de uma reconfiguração (reflexão arquitetural).

3.2. Processamento Interno

O processamento interno do compilador é dividido em três partes: primeiro passo, segundo passo e exportação. A figura 4 apresenta o diagrama de fluxo de processo de compilação em **UML** [5].

No primeiro passo são feitas análises sintáticas e semânticas da descrição de arquitetura. Nesse momento, os elementos (portas, módulos e conectores) que apresentam pendências são marcados para serem reavaliados. Ao final do primeiro passo o repositório e o relatório já se encontram parcialmente alimentados.

No segundo passo os elementos pendentes são revistos e os que apresentarem erros são excluídos do repositório. Ao final do segundo passo o repositório apresenta sua forma final. O relatório é complementado com os erros que foram encontrados e estatísticas da arquitetura descrita.

Na fase de exportação, com base na última forma do repositório, é criada a seqüência de comandos de configuração. Finalmente, os dados de configuração, de relatório e do repositório de meta-nível são exportados para arquivo.

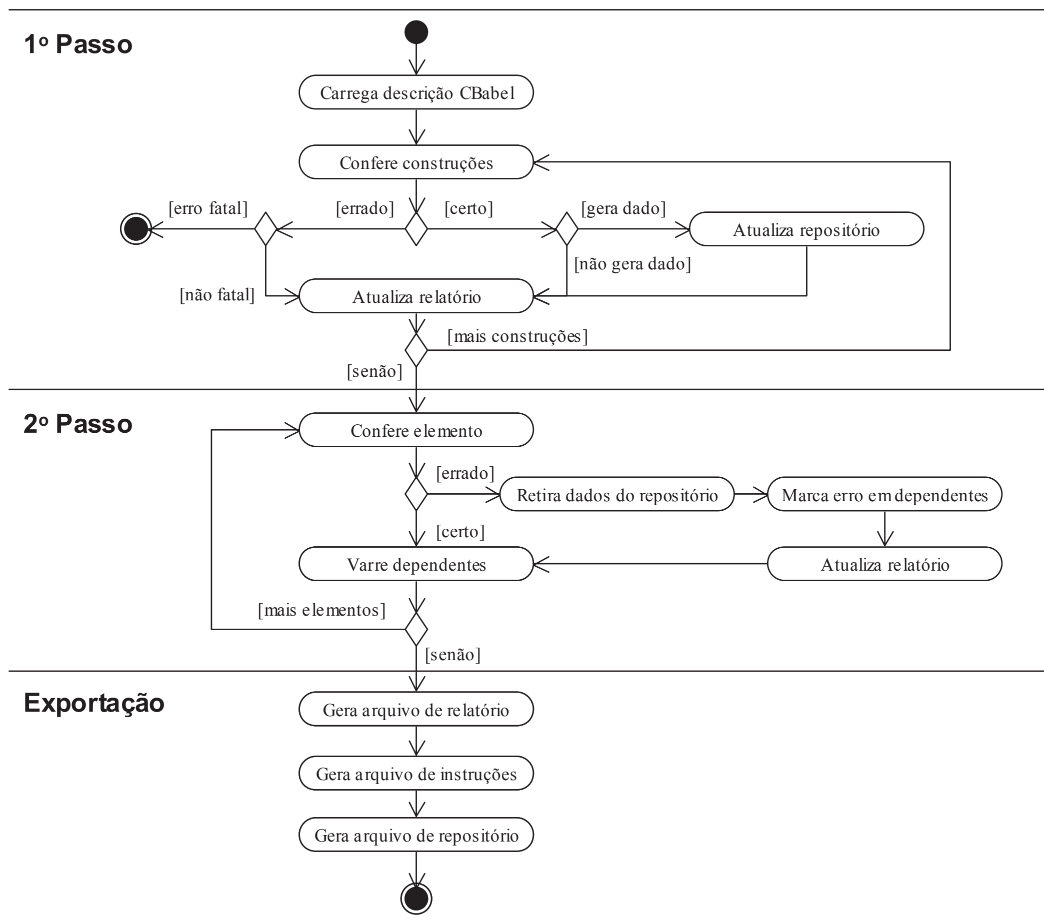


Figura 4. Fluxo do processo de compilação

3.3 Repositório de Dados de Meta-Nível

Os dados mais importantes gerados pelo compilador são os dados de meta-nível armazenados no arquivo de repositório. Estes dados são responsáveis pelo mapeamento da arquitetura em objetos executáveis no momento da execução e pelo armazenamento das informações sobre a configuração da aplicação. Sem estes dados o ambiente de execução e reconfiguração não poderiam realizar suas atividades. Tendo em vista a importância do repositório de meta-nível, iremos apresentar seu modelo conceitual.

Atualmente os dados de meta-nível são armazenados em uma estrutura de árvore com referência direta entre as instâncias das classes. Existem também dentro da estrutura da árvore, listas encadeadas que permitem a leitura seqüencial das instâncias de mesma classe/tipo. Desta forma houve uma simplificação dos métodos referentes ao gerenciamento dos dados e uma diminuição do esforço de programação. O repositório agora tem uma referência ao elemento raiz da árvore e a primeira instância de cada uma das listas encadeadas. A figura 5 apresenta os relacionamentos entre a classe repositório e as classes referenciadas.

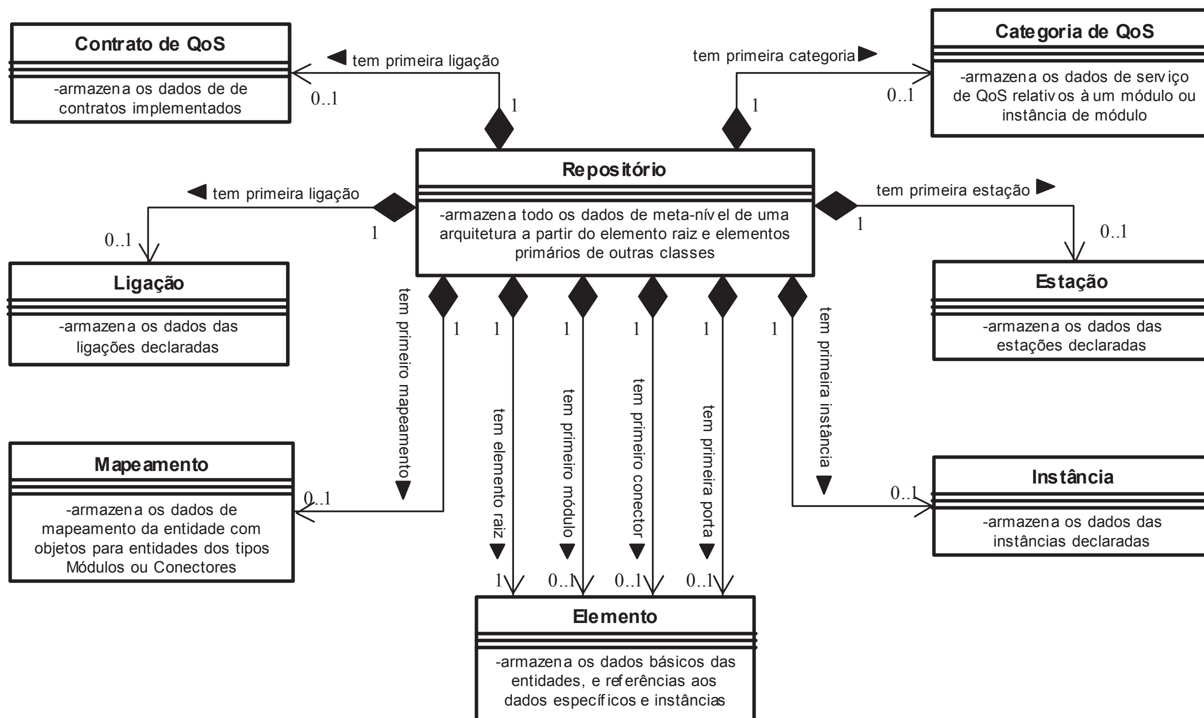


Figura 5. Relacionamento entre a classe do Repositório e as classes referenciadas

Os relacionamentos entre as classes que armazenam diretamente os dados de meta-nível estão representados na figura 6. A classe **Elemento** armazena os dados básicos dos elementos declarados (para módulos, conectores e portas). Em CBabel um **Elemento** pode ser estendido e ter filhos como mostra os auto-relacionamentos na figura.

De acordo com a necessidade classes auxiliares são utilizadas para o armazenamento de dados dos elementos. Essas classes são **Mapeamento**, **Parâmetros de Conector** e **Parâmetros de Porta**. A classe Mapeamento armazena as informações de referência do objeto executável. A classe **Parâmetros de Conector** armazena dados específicos de elementos do tipo conector e está associada a elementos do tipo porta que são coordenados pelo conector. Finalmente, a classe **Parâmetros de Porta** armazena dados específicos de portas e contém uma classe **Guarda** agregada a ela. Esta classe **Guarda** representa o guarda da porta e está associado a uma porta alternativa.

A classe **Instância** armazena os dados das instâncias de elementos. Os dados das ligações feitas entre instâncias são armazenados utilizando a classe **Ligação**. Tanto a classe **Instância** quanto a classe **Ligação** tem uma associação ao elemento em que foram declaradas.

Os relacionamentos que tem setas em sua linha de ligação representam a direção única de navegação da associação.

As classes **Elemento**, **Instância**, **Ligação**, **Estação** e **Mapeamento** têm a associação descrita como **tem por próximo**, que é exatamente a associação ao próximo objeto da lista encadeada daquela classe. Nas classes **Elemento** e **Instância** esta associação é bidirecional, representando uma lista duplamente encadeada.

Existem quatro classes para armazenar os dados relativos à Qualidade de Serviço (*QoS – Quality of Service*). **Categoria de QoS** é a classe que caracteriza um serviço de QoS através de propriedades armazenadas através da classe **Propriedade de QoS**. Esta última também é utilizada para armazenar os *constrains* contidos em um **Perfil de QoS**, que associa perfis de requisição ou provisão de uma **Categoria de QoS** por um dado módulo ou instância de módulo. Finalmente a classe **Contrato de QoS** armazena os dados para utilização de serviços de QoS entre duas instâncias.

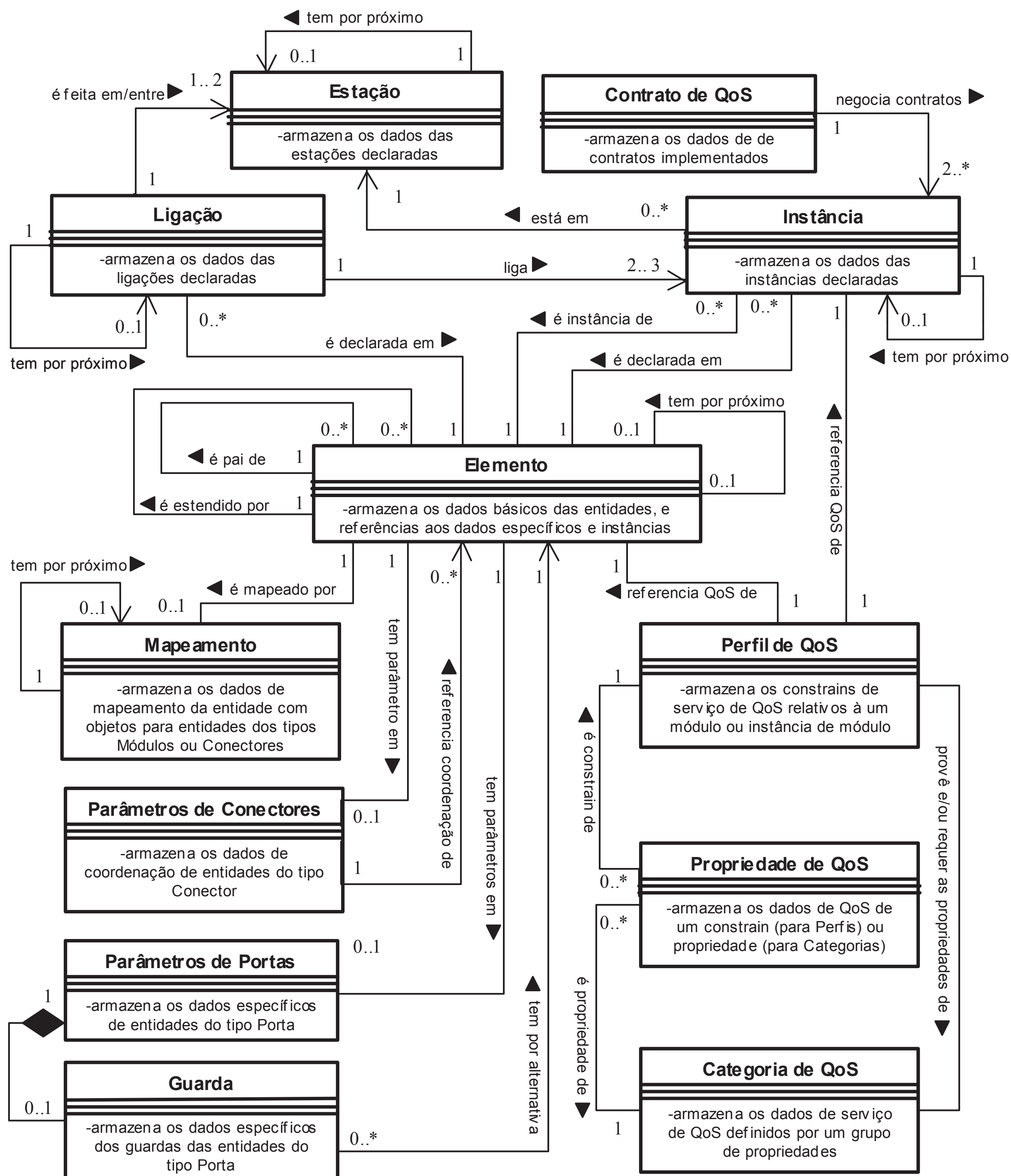


Figura 6. Associações entre as classes de armazenamento de dados de meta-nível

3.4. Construções Traduzidas pelo Compilador

A gramática implementada no compilador permite a tradução de todas as construções existentes na linguagem CBabel. Nesta seção elas serão apresentadas, organizadas em categorias.

<u>Construções relativas a Módulos</u>

- Declaração de módulos nomeados com declaração de instâncias e parâmetros como opcionais.
- Declaração de módulos não nomeados com declaração de instâncias obrigatória e parâmetros opcionais.

- Declaração de módulos estendidos com declaração de instâncias obrigatória e parâmetros opcionais.
- Declaração de módulos compostos utilizando elementos filho dos tipos módulo, conector e porta.
- Declaração de mapeamento do módulo ao objeto executável.
- Declaração de exportação (visibilidade externa) das portas do módulo.

Construções relativas a Conectores

- Declaração de conectores nomeados com declaração de instâncias e parâmetros como opcionais.
- Declaração de conectores não nomeados com declaração de instâncias obrigatória e parâmetros opcionais.
- Declaração de conectores estendidos com declaração de instâncias obrigatória e parâmetros opcionais.
- Declaração de conectores compostos utilizando elementos filho dos tipos conector e porta.
- Declaração de mapeamento do conector ao objeto executável.
- Declaração de exportação (visibilidade externa) das portas do conector.
- Declaração de variáveis de estado e de condição para contratos de coordenação.
- Declaração de contatos de coordenação dos tipos **Concurrent**, **Exclusive** e **Self-Exclusive**.

Construções relativas a Portas

- Declaração de portas com declaração de direção obrigatória e de instâncias, parâmetro de retorno e parâmetros como opcionais.
- Definição de portas nomeadas para futura extensão.
- Declaração de portas estendidas com declarações de instâncias e direção obrigatórias e de parâmetro de retorno e parâmetros como opcionais.
- Declaração com redefinição local de porta com declarações de direção, parâmetro de retorno, parâmetros e instâncias obrigatórias.
- Declaração de guardas para a porta.

Construções relativas a Estações

- Declaração de *alias* para estações reais.

Construções de Instanciação

- Declaração de instanciação no servidor local (*localhost*) com passagem de parâmetros opcional (para instâncias já declaradas).
- Declaração de instanciação em uma estação específica com passagem de parâmetros opcional (para instâncias já declaradas).
- Declaração de instanciação com nome e elemento de referência da instância em uma estação específica com passagem de parâmetros opcional (para instâncias não declaradas).

Construções de Ligação de Instâncias

- Declaração de ligação de instâncias simples (instâncias já instanciadas).
- Declaração de ligação de instâncias de nomes compostos, ex.: **teste.module2** (instâncias já instanciadas).

Construções de Inicialização de Instâncias

- Declaração de inicialização de uma instância já declarada e instanciada.

Construções de Qualidade de Serviço

- Declaração de categorias, perfis e contratos de QoS.

3.5. Navegador de Repositório

O **Navegador de Repositórios** é uma ferramenta que tem como objetivo permitir a visualização dos dados de meta-nível de arquivos do tipo *.cbo. Além disso ele fornece ferramentas de busca de elementos, estatísticas do repositório, e exportação de arquivos de descrição CBabel e de arquivos XML. A figura 7 apresenta a nova interface do Navegador de Repositórios que agora é do tipo *MDI* (*Multiple Document Interface*).

Nesta interface temos dois tipos de janelas: a janela do programa e a janela de documento. A janela do documento contém as funções de edição e estatística específicas de repositórios CBabel e permite a visualização dos dados de meta-nível. A janela do programa contém e gerencia a apresentação das janelas de documento abertas.

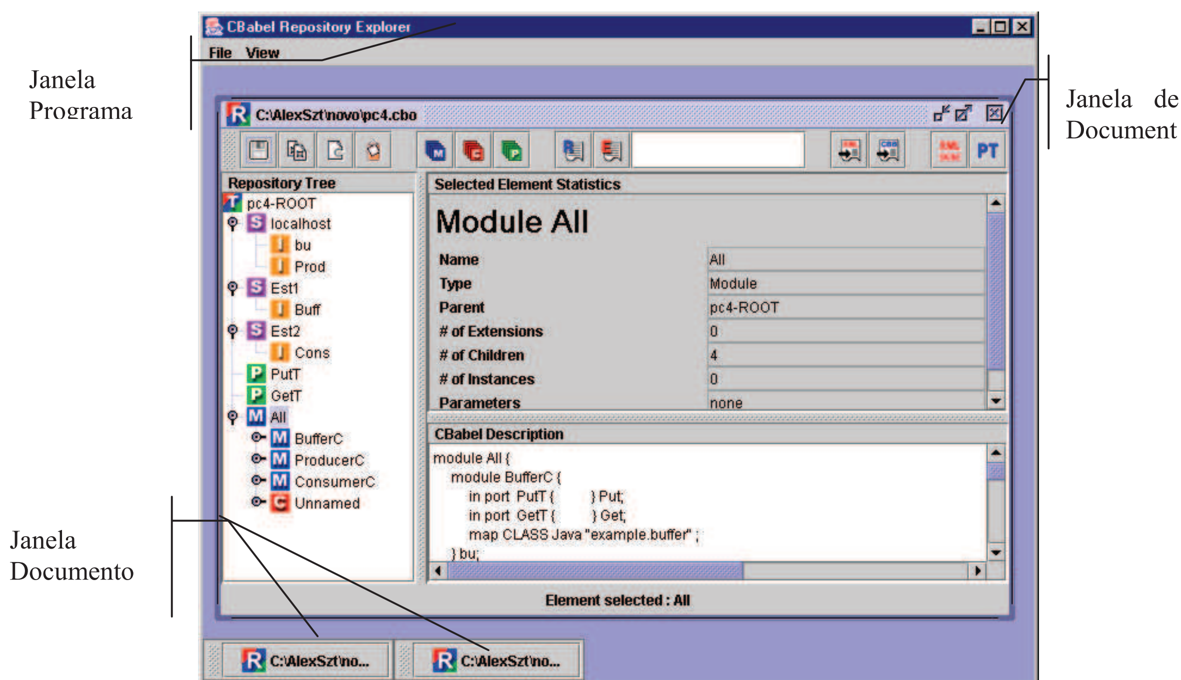


Figura 7. Interface do Navegador de Repositórios

Primeiramente vamos explicar a utilização da janela de documentos, começando com os três painéis de dados. O painel **Repository Tree** exibe a árvore do repositório, apresentando primeiro as estações ("S") e suas instâncias ("I"). Depois são apresentados os elementos do primeiro nível da arquitetura que podem conter filhos. Os elementos são identificados pelos ícones: "M" para módulos, "C" para conectores e "P" para portas.

O painel **Selected Element Statistics** apresenta as estatísticas do elemento selecionado. Caso selecionemos a raiz do repositório serão apresentadas as estatísticas da arquitetura como um todo.

O painel **CBabel Description** apresenta uma descrição CBabel correspondente ao elemento selecionado. Esta descrição é obtida através de um processo de engenharia reversa. Este painel pode apresentar também o elemento em forma de descrição XML. Neste caso o nome do painel será apresentado como **XML Description**. O tipo de descrição apresentada pode ser selecionado através da barra de botões da janela de documentos.

A barra de botões da janela documentos contém as funções de edição, pesquisa e visualização desta janela. Através do botão podemos salvar o repositório na forma atual. Os botões permitem respectivamente copiar, cortar e colar elementos em um documento e entre documentos diferentes (ainda em implementação). Para apresentarmos uma lista dos módulos, conectores ou portas devemos selecionar o botão correspondente ( ) . O botão seleciona o elemento raiz da arquitetura, apresentando as estatísticas da mesma. Para localizar um elemento devemos digitar seu nome na caixa de texto ao lado do botão e pressioná-lo. Através dos botões podemos exportar o repositório para descrições *.cbo ou arquivos XML. Pressionando o botão / alternamos a apresentação da descrição do elemento entre XML/CBabel. O botão EN/PT permite alternar entre os idiomas Inglês/Português.

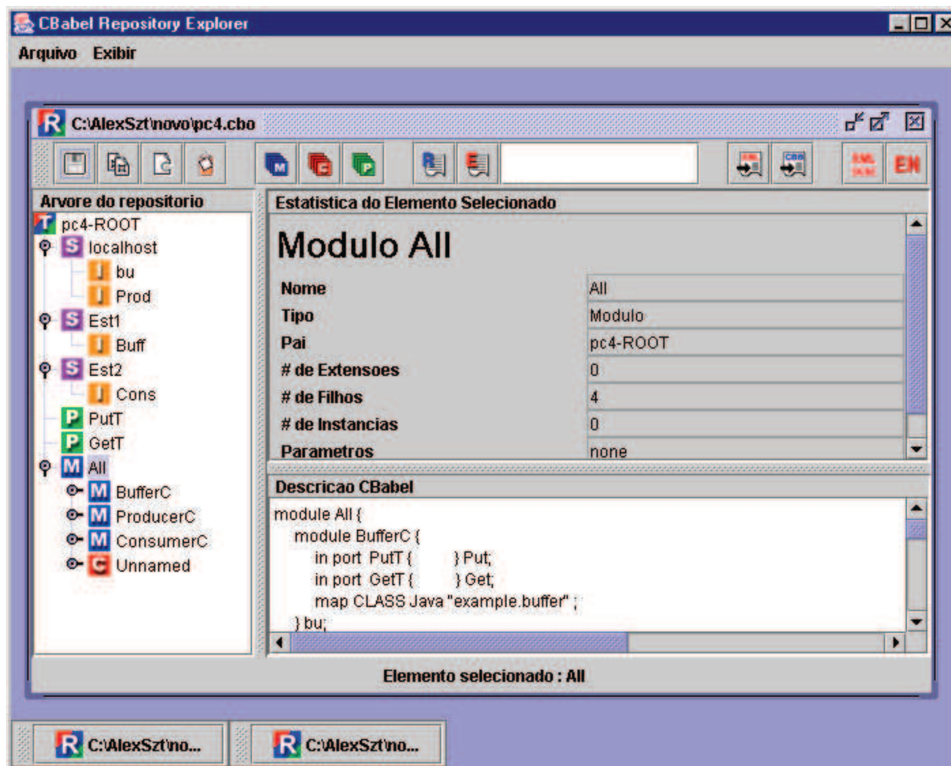


Figura 8. Interface do Navegador de Repositórios em Português

A janela do programa, de nome **CBabel Repository Explorer**, através do menu **File** permite a abertura de novos arquivos de repositório (**Open Repository**) e o encerramento do programa (**Close**). Nesta janela também é possível selecionar o idioma do programa (**Language**). Na figura 8 apresenta a interface em português. Podemos visualizar o conteúdo da área de transferência (**Clipboard**) através do menu **View**, será então apresentada a janela da figura 9.

Um exemplo de arquivo XML exportado (que pode ser visualizado no Internet Explorer 5.0) é apresentado na figura 10.

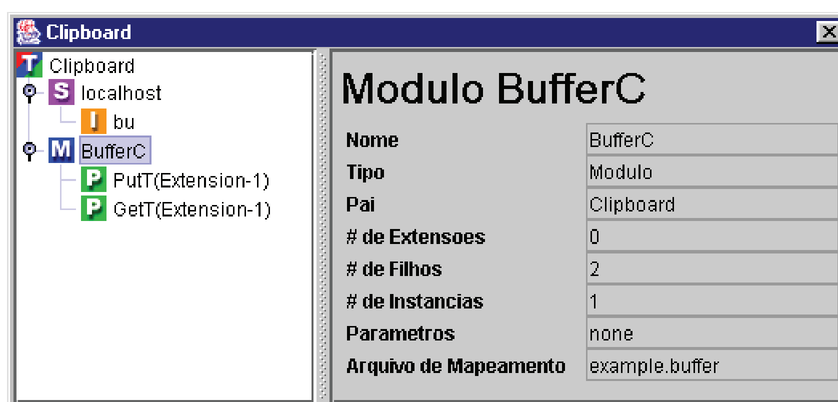


Figura 9. Janela de visualização da área de transferência (clipboard)

```
C:\marcelo\pc22.xml - Microsoft Internet Explorer
File  Edit  View  Favorites  Tools  Help

  <?xml version="1.0" ?>
- <architecture name="pc2" xmlns="x-schema:c:\marcelo\cbabelschema.xml">
    <station alias="localhost" />
  + <station alias="Est1">
  + <station alias="Est2">
  + <port name="PutT">
  - <port name="GetT">
      <extensions>2</extensions>
      <direction>none</direction>
      <returnparameter>string</returnparameter>
      <exported>false</exported>
    </port>
  - <module name="BufferC">
      <extensions>0</extensions>
    - <mapping>
        <object>CLASS</object>
        <language>Java</language>
        <reference>example.buffer</reference>
      </mapping>
    - <children>
      - <port name="PutT(Extension-1)">
```

Figura 10. Exemplo de arquivo XML exportado

4. Implementação

Na implementação do compilador foram utilizadas a linguagem Java e a ferramenta Java Compiler Compiler – JavaCC [5]. A implementação do compilador é dividida em quatro camadas, de acordo com a função e as ferramentas utilizadas em seu desenvolvimento. Essa divisão está representada na figura 11.

A ferramenta JavaCC é um gerador de *parser* para Java. A partir de uma gramática descrita em uma forma estendida de BNF, é possível descrever as especificações léxicas e gramaticais de CBabel. Com esta descrição o pré-processador JavaCC gera o código de uma série de classes Java que implementam o *parsing* e a verificação sintática. Atualmente o arquivo fonte da descrição da sintaxe de CBabel (**cbabel.jj**) contém 2429 linhas.

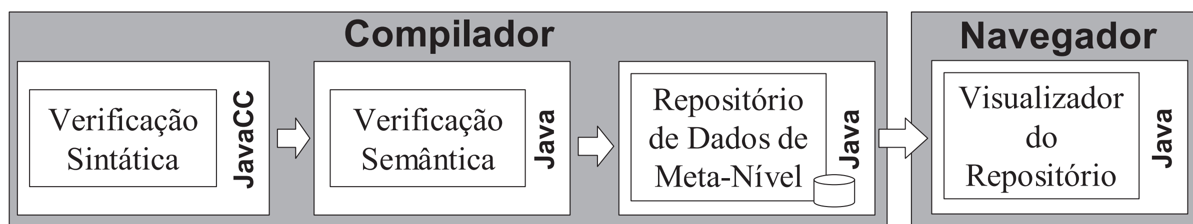


Figura 11. Camadas de implementação do compilador.

A verificação semântica fica ao encargo da classe Java **CBCManager**. Esta classe é, também, responsável pela construção e controle da árvore de *parsing* do compilador, e pela geração dos arquivos de saída. Esta classe contém, atualmente, 56 métodos e 1400 linhas de código.

A árvore de *parsing* está contida na classe Java **Repository** que conta com outras 13 classes para o armazenamento de informações de meta-nível: **Element**, **Instance**, **Map**, **Cpar**, **Ppar**, **Parameter**, **Guard**, **Link**, **Station**, **QoSCategory**, **QoSProfile**, **QoSProperty** e **QoSContract**. A classe **Repository** contém 21 métodos e 135 linhas de código. As outras classes contêm reunidas 2279 linhas de código.

No desenvolvimento do Navegador de Repositórios foram utilizadas 6 classes Java: **CBabelAPI**, **TDCB**, **DataPanel**, **DataLine**, **RepositoryFrame** e **RepositoryClipboard**. A classe **TDCB** é responsável pelo controle da interface gráfica de múltiplos documentos do navegador, contém 546 linhas de código, e utiliza a classe **RepositoryFrame** (703 linhas de código) para a navegação do repositório. A classe **CBabelAPI** (com 1260 linhas de código) se destaca como ferramenta na pesquisa do repositório.

5. Exemplo

O uso do compilador é demonstrado através de uma aplicação do tipo produtor-consumidor-buffer (listagem 1). Inicialmente são declarados os nós, **Est1** e **Est2** (linhas 1-2), onde os módulos irão executar. Nas linhas 3 e 4 são feitas as declarações de dois tipos de porta (**PutT** e **GetT**) usadas nos módulos. Nas linhas 5 a 9 é declarada a classe de módulo **BufferC**. Além das portas que compõem a interface (linhas 6-7), temos a cláusula **map** (linha 8) que indica o mapeamento do módulo em uma classe Java.

```
01   station Est1 is m1.ime.uerj.br ;
02   station Est2 is m2.ime.uerj.br ;

03   port int PutT (string Item);
04   port string GetT ;

05   module  BufferC {
06       out port PutT { } Putb;
07       in port GetT { } Getb;
08       map CLASS Java "example.buffer";
09   }

10   module  ProducerC {
11       out port PutT { } PutP;
12       map  CLASS Java "example.producer";
13   }

14   module  ConsumerC {
15       in port GetT { } GetC;
16       map  CLASS Java "example.consumer";
17   }
```

```
18   connector ConnMutex  {
19       exclusive{
20           out port PutT { } GPut;
21           out port GetT { } GGet;
22       }
23       in port PutT { } PPut;
24       in port GetT { } PGet;
25       map Class Java "connectors.log";
26   }Cmutex;

27   instantiate BufferC   as Buff at Est1;
28   instantiate ProducerC as Prod at Est2;
29   instantiate ConsumerC as Cons at Est2;
30   link Cons to Buff by Cmutex;
31   link Prod to Buff by Cmutex;
32   start Buff;
33   start Prod;
34   start Cons;
```

Listagem 1. Um arquivo (*.cbb) de descrição de arquitetura em CBabel.

A seguir temos a declaração das classes de módulo **ProducerC** (linhas 10-13) e **ConsumerC** (linhas 14-17). Na declaração da classe de conector **ConnMutex** (linhas 18-26) observa-se a especificação de um contrato de exclusão mútua (linhas 19-22) entre duas portas do conector **ConnMutex** (que serão ligadas diretamente ao *buffer*).

A topologia da arquitetura é configurada pela criação das instâncias dos componentes (linhas 27-29), seguida da ligação destas instâncias formando uma estrutura (linhas 30-31), e pela inicialização das mesmas (linhas 32-34).

A descrição da arquitetura da aplicação é submetida ao compilador para verificação de consistência e geração dos arquivos com instruções para a carga da arquitetura e o repositório de meta-nível. Em caso de erro na compilação é possível identificar o ponto em que o mesmo ocorreu.

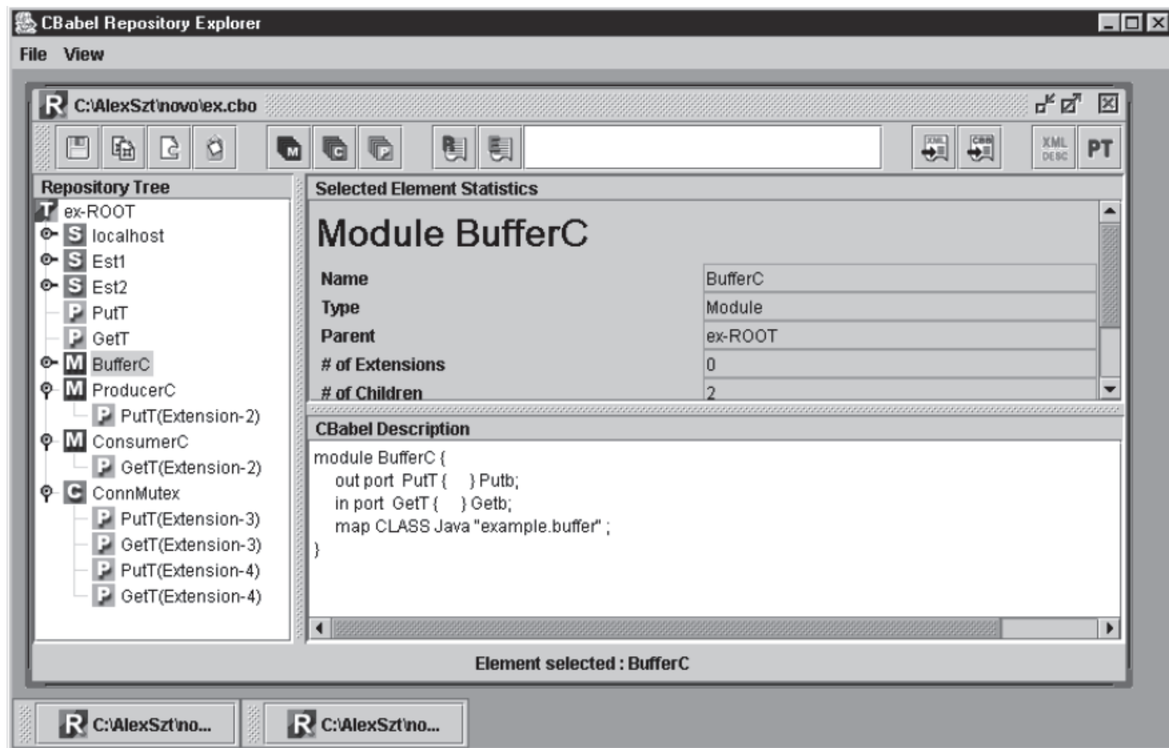


Figura 12. Visualização do repositório (*.cbo) gerado pelo compilador

A figura 12 apresenta a visualização do repositório exportado pelo compilador, utilizando o navegador. Na janela de documento do navegador, com interface *MDI* (*Multiple Document Interface*), podemos ver, no painel *Repository Tree*, os elementos declarados. O navegador apresenta também, no painel *CBabel Description*, a descrição CBabel correspondente ao elemento selecionado. Alternativamente, a descrição pode ser visualizada em formato **XML** como mostra a figura 13. O navegador também fornece ferramentas de exportação dos dados de meta-nível para arquivos CBabel ou XML.

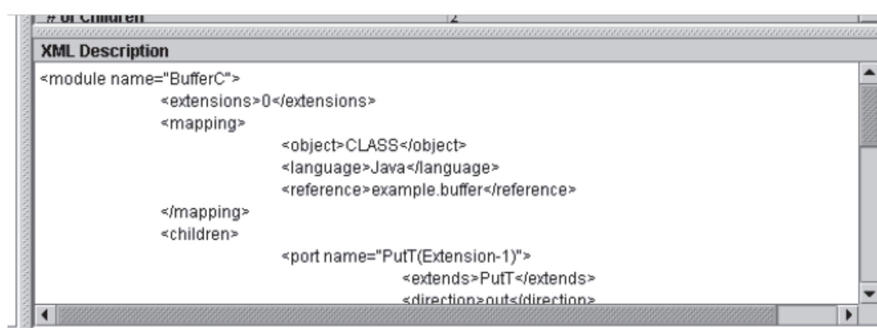


Figura 13. Visualização da descrição XML do repositório no navegador.

No arquivo de comandos de configuração (listagem 2) os comandos **instantiate** (linhas 1-3) contêm o nome da classe identificada na cláusula **map** de cada módulo. Também é utilizado o endereço do nó no lugar do *alias*. As substituições do endereço dos nós também ocorrem nos comandos **link** (linhas 4-5) e **start** (linhas 6-8).

```
1    instantiate example.buffer as Buff at buzios.gta.ufrj.br
2    instantiate example.producer as Prod("teste") at 127.0.0.1
3    instantiate example.consumer as Cons at copa.gta.ufrj.br
4    link Cons at copa.gta.ufrj.br to Buff at buzios.gta.ufrj.br with Log1
5    link Buff at buzios.gta.ufrj.br to Prod at 127.0.0.1 with Log2
6    start Buff at buzios.gta.ufrj.br
7    start Prod at 127.0.0.1
8    start Cons at copa.gta.ufrj.br
```

Listagem 2. Arquivo de instruções (*.cbi) gerado pelo compilador

Os arquivos gerados pelo compilador são, então, utilizados para colocar a arquitetura em execução sobre Configurador R-RIO (descrito em [1]).

6. Resultados

Nesta seção discutimos os resultados do desenvolvimento do projeto, que pode ser separado em três fases. Cada fase foi marcada por uma evolução da arquitetura do projeto, impondo mudanças na estrutura de sua implementação.

6.1. Resultados da 1ª Fase

Na primeira fase, foi implementado um compilador que fazia traduções das construções básicas da linguagem CBabel:

- Declaração e instanciação dos três tipos de elementos da linguagem;
- Composição de elementos;
- Mapeamento dos elementos aos programas objetos;
- Declaração de *alias* para estações reais;
- Declaração de ligações entre entidades;
- Declaração de *start* de instâncias.

O compilador exportava um arquivo de relatório de compilação e um arquivo de instruções para carga no ambiente de execução de R-RIO. Os dados de meta-nível da arquitetura eram utilizados no *parsing*, e depois eram descartados. Estes dados eram armazenados em classes, que por sua vez eram armazenados em vetores na classe responsável pela compilação (gerada pelo JavaCC). Existia um vetor para cada classe, que armazenava as instâncias que eram utilizadas. Essas instâncias se referenciavam utilizando seus índices nos vetores.

Quanto à implementação, eram utilizadas 7 classes Java para o armazenamento temporário (apenas no momento de compilação) dos dados de meta-nível, que totalizavam cerca de 400 linhas de código. O arquivo cbabel.jj, da ferramenta Java Compiler Compiler, era responsável pela verificação sintática e semântica do compilador e continha 1656 linhas de código.

6.2. Resultados da 2ª Fase

Na segunda etapa, foi implementada a tradução de novas construções CBabel. Dentre estas implementações destacam-se:

- Implementação das regras para declarações de elementos que estendem outros elementos de mesmo tipo.
- Implementação das regras para redefinição local de portas;
- Implementação de parâmetros opcionais para elementos dos tipos módulo e conector;
- Implementação de contratos de coordenação (*Concurrent*, *Exclusive*, *Self-Exclusive*);
- Implementação de variáveis de controle de coordenação (variável de estado e de condição);
- Implementação de guarda para as portas;
- Implementação de novas sintaxes para instanciação e ligação.

Com a implementação destas novas construções ocorreu uma remodelação da arquitetura do compilador, com acréscimo de funções. Essas modificações são citadas a seguir:

- Implementação de uma classe de repositório para armazenar todos os dados de meta-nível da árvore de *parsing*;
- Implementação da exportação para arquivo do objeto do repositório de meta-nível;
- Modificação da estrutura do repositório, que era vetorial, para uma estrutura de árvore;
- Acréscimo de uma classe para o armazenamento dos dados de meta-nível e implementação de referências diretas entre as instâncias das classes (as referências eram indexadas);
- Divisão do código do processo de compilação em dois arquivos, um para a sintaxe e outro para a semântica.

Além dos itens apresentados nas duas listas acima foi desenvolvida uma nova ferramenta, o Navegador de Repositórios. Com as seguintes funções:

- Navegação na arquitetura do repositório em uma estrutura de árvore e visualização dos dados de meta-nível do elemento selecionado;
- Ferramenta de busca de elementos;
- Visualização *on line* do elemento em descrição CBabel ou XML;
- Função de exportação do conteúdo do repositório para arquivo de Descrição CBabel ou XML;
- Interface em Português ou Inglês.

6.3. Resultados da 3ª Fase

Na terceira fase, foram acrescidas ao compilador as traduções das construções de Qualidade de Serviço (QoS) da linguagem CBabel:

- Declaração de categorias de QoS;
- Declaração de perfis de QoS, com papéis de provisão (*provide*) e requisição (*request*);
- Declaração de contratos de QoS entre instâncias.

Para satisfazer estes acréscimos, o repositório, recebeu a implementação de novas classes, totalizando 14 classes e 2414 linhas de código. O compilador também recebeu novas funções, contando agora com 3833 linhas de código dos dois arquivos que já existiam.

Além disso, foi implementada uma nova interface para múltiplos documentos no Navegador de Repositórios. Essa interface permite efetuar operações de corte-e-cola entre diferentes documentos (em implementação), fornecendo ao Navegador uma capacidade de edição de repositórios. Para satisfazer estas mudanças, o navegador desenvolvido agora conta com um total de 6 classes para a implementação de seus recursos, totalizando 3420 linhas de código.

6.4. Discussão dos Resultados

Os resultados obtidos durante as três etapas de desenvolvimento atingiram a proposta do projeto. Além do desenvolvimento de uma versão funcional do compilador, foi desenvolvida uma ferramenta auxiliar (o Navegador de Repositórios) que não fazia parte do projeto proposto.

Em relação à terceira etapa, ocorreu considerável avanço. Além da inclusão de construções de Qualidade de Serviço (*QoS – Quality of Service*) entre as construções traduzidas pelo compilador, a interface do navegador foi remodelada. Ao ser remodelada a interface, foi gerada a possibilidade de novas características como a transferência de informação de meta-nível entre repositórios.

A tabela a seguir apresenta os dados sobre a quantidade de linhas de código e arquivos obtidos ao final de cada etapa.

Tabela 1. Fases do projeto

Item	1ª Fase		2ª Fase		3ª Fase	
	Arqu.	Linhas	Arqu.	Linhas	Arqu.	Linhas
Compil.	1	1656	2	3350	2	3833
Reposit.	7	400	9	1500	14	2414
Naveg.	0	0	2	2070	6	3420
Total	8	2056	13	6920	22	9667

7. Conclusão

Apresentou-se o projeto de um compilador para a ADL CBabel, elemento do *framework* R-RIO. O compilador, processa descrições CBabel e exporta dados para a execução da arquitetura. Dentre os dados exportados, destaca-se um repositório que contém a própria árvore de *parsing* da compilação. O acesso às informações deste repositório através de reflexão arquitetural é importante tanto para a carga como para a reconfiguração da arquitetura descrita.

A integração com o Configurador é feita de forma semi-automática, através da leitura dos arquivos gerados pelo compilador. Esta integração poderá ser automatizada interligando-se o compilador/visualizador com o mesmo.

Na continuação deste trabalho está prevista uma melhoria das ferramentas do navegador de repositório, implementando uma representação gráfica dos elementos da arquitetura e da ligação das instâncias. Além disto, está sendo estudada a adequação do exportador XML à proposta de xADL [6]. Isso permitirá a reutilização das ferramentas e arquiteturas já desenvolvidas neste contexto.

8. Referências Bibliográficas

[1] LOBOSCO, M., LOQUES, O. G., SZTAJNBERG, A., "R-RIO: Um Ambiente para Suporte à Construção e à Evolução de Sistemas", *XIII Simpósio Brasileiro de Engenharia de Software 99 (Caderno de Ferramentas)*, pp. 53-56, Florianópolis, Outubro, 1999.

[2] SZTAJNBERG, A., LOBOSCO, M., LOQUES, O. G., "Configurando protocolos de interação na abordagem R-RIO", *In: anais do XIII Simpósio Brasileiro de Engenharia de Software*, pp. 29-45, Florianópolis, Outubro, 1999.

[3] SZTAJNBERG, A., *Flexibilidade e Separação de Interesses para a Concepção e Evolução de Aplicações Distribuídas*, Tese de D.Sc., PEE/ COPPE/UFRJ, Rio de Janeiro, Maio, 2002.

[4] SZTAJNBERG, A., LOQUES, O. G., "Reflection in the R-RIO Configuration Programming Environment", IEEE *Middleware*'2000, *Workshop on Reflective Middleware*, Palisades, NY, EUA, Abril, 2000.

[5] Sun Microsystems Inc. ,“Java Compiler Compiler - JavaCC”, 1996. http://www.webgain.com/products/java_cc/

[6] University of California, Irvine, “xADL”, 2000.http://www.isr.uci.edu/projects/ xarchuci