Utilização do paradigma Draco para implementar especificações Estelle na linguagem C++

Antônio Carlos Lima de Santana
Antônio Francisco do Prado, Wanderley Lopes de Souza
Departamento de Computação (DC)
Universidade Federal de São Carlos (UFSCar)
Cx. 676, 13565-905 São Carlos (SP)
{santana, prado, desouza}@dc.ufscar.br

Resumo

Extended State Transition Language (Estelle) é uma Técnica de Descrição Formal (TDF), padronizada pela International Organization for Standardization (ISO), que foi desenvolvida para a especificação formal de sistemas distribuídos e protocolos de comunicação. A partir da especificação de um sistema em Estelle, pode-se obter uma implementação automática, com o auxílio de ferramentas apropriadas. O objetivo principal deste artigo é o de mostrar como implementações de sistemas orientados a objetos, na linguagem C++, podem ser geradas a partir de suas especificações Estelle, utilizando o ambiente de transformação de software Draco.

Abstract

Extended State Transition Language (Estelle) is a Formal Description Technique (FDT) standardized by International Organization for Standardization (ISO) that was developed for the formal specification of distributed systems and communication protocols. A system implementation can be obtained automatically from its Estelle specification using appropriated tools. The main goal of this paper is to show how system object oriented implementations in the C++ language can be generated from their Estelle specifications using the software transformation environment Draco.

1. Introdução

As Técnicas de Descrição Formal (TDFs) surgiram para serem empregadas no projeto de sistemas de comunicação complexos, a fim de permitir a produção de especificações claras, concisas e livres de ambiguidades. O emprego de uma TDF facilita também a validação de especificações, a geração automática de implementações a partir das especificações e o teste de conformidade das implementações em relação as suas respectivas especificações.

Extended State Transition Language (Estelle) [ISO 88] é uma TDF padronizada pela International Organization for Standardization (ISO), que foi desenvolvida visando a especificação formal de sistemas distribuídos e protocolos de comunicação, sobretudo os relativos ao modelo de referência Open Systems Interconnection (OSI) [ISO 83a]. O modelo de Estelle é baseado numa Máquina de Estados Finita Estendida (MEFE), que combina dois tipos de notações: estados, interações e transições são descritos através de uma MEF, enquanto que variáveis, parâmetros e prioridades são descritos através de um subconjunto da linguagem de programação Pascal [ISO 83b]. Um tutorial sobre Estelle é apresentado em [LOP 89].

Nos últimos dez anos, várias ferramentas foram desenvolvidas para Estelle. Essas ferramentas podem ser agrupadas em duas grandes categorias: as que atuam na fase do projeto de um sistema, auxiliando na criação e na validação (verificação e/ou simulação) das especificações, e as que atuam na fase de implementação do sistema, auxiliando na geração do código e no teste. Em relação à segunda categoria, a maioria dos compiladores Estelle geram código Pascal [GER 83, LOP 88] ou código C [VUL 88, EWS 89], obrigando o usuário a realizar uma complementação manual significativa ao código gerado por essas ferramentas. Mais recentemente, foi desenvolvido o compilador PetDingo [SIS 91], que gera código C++ a partir de uma especificação Estelle.

Este artigo apresenta uma proposta para implementação, no ambiente de transformação de software Draco, da modelagem orientada a objeto de especificações Estelle descrita em [BAR 95a]. Nessa proposta busca-se um processo de transformação que possibilite a obtenção de implementações C++ orientadas a objeto a partir de especificações Estelle.

Na próxima seção, a modelagem orientada a objeto utilizada neste artigo é discutida, enquanto que a proposta de sua implementação no ambiente Draco é descrita na seção 3. Na seção 4, é descrito o processo de

transformação de especificações Estelle para C++, usando um estudo de caso. Finalmente, as conclusões e os trabalhos futuros são apresentados na seção 5.

2. Modelagem orientada a objeto de especificações Estelle

Uma especificação Estelle é composta de um conjunto de *módulos*, que se comunicam através de *interações* (mensagens). Um módulo pode ser refinado em submódulos, definindo um parentesco e uma estrutura hierárquica entre os módulos da especificação. Cada módulo é representado por uma caixa preta com *pontos de interação* de entrada/saída. A cada ponto de interação é associada uma fila de comprimento infinito, que é utilizada para armazenar as mensagens recebidas por aquele ponto. *Canais* de comunicação bidirecionais podem conectar pontos de interação, sendo que estes definem os conjuntos de interações a serem emitidas e recebidas por cada ponto. O comportamento de um módulo é descrito por uma Máquina de Estados Finita Estendida (MEFE).

Um conceito arquitetônico (e.g., módulo, interação, ponto de interação) ou um recurso (e.g., transição, canal, links de comunicação) de Estelle é geralmente implementado numa hierarquia de classes, na qual uma classe genérica(classe base) captura os elementos fundamentais desse conceito ou desse recurso e as classes derivadas(subclasses) capturam as suas especializações. Essa hierarquia contém classes de prateleira, comuns a todas as implementações, e classes especializadas que levam em conta os aspectos particulares de cada especificação Estelle. As classes de prateleira residem em uma biblioteca, e são referenciadas pelas classes especializadas que são geradas a cada nova implementação. As Figuras 2.1 e 2.2 apresentam as classes base, segundo as quais serão mapeadas as especificações Estelle. Os modelos orientados a objetos destas figuras usam a notação do método FUSION [FUS 94].

No diagrama da Figura 2.1, a partir da classe *Modulo* são derivadas, utilizando-se o princípio da herança, as classes *ModuloInativo, ModuloSistema, ModuloProcesso* e *ModuloAtividade*, enquanto que a partir da classe *ModuloSistema* são derivadas as classes *ModuloSistemaProcesso* e *ModuloSistemaAtividade*. No diagrama da Figura 2.2, as classes *PI_Fila_Individual* e *PI_Fila_Comum* são derivadas da classe *Ponto_de_Interacao*. A classe *PI_Remoto* serve para espelhar um ponto de interação em outro processo de sistema operacional ou outro processador.

Os serviços da classe *Interface_de_Comunicação* permitirão a comunicação entre os módulos e a classe *Tempo* servirá para controlar o estado de uma transição, em função dos parâmetros de tempo mínimo e máximo de espera para execução daquela transição, realizando a função da cláusula *delay* da TDF Estelle. Esses diagramas indicam também os principais atributos e métodos das classes representadas, sendo que uma descrição mais detalhada dos mesmos é realizada em [BAR 95b].

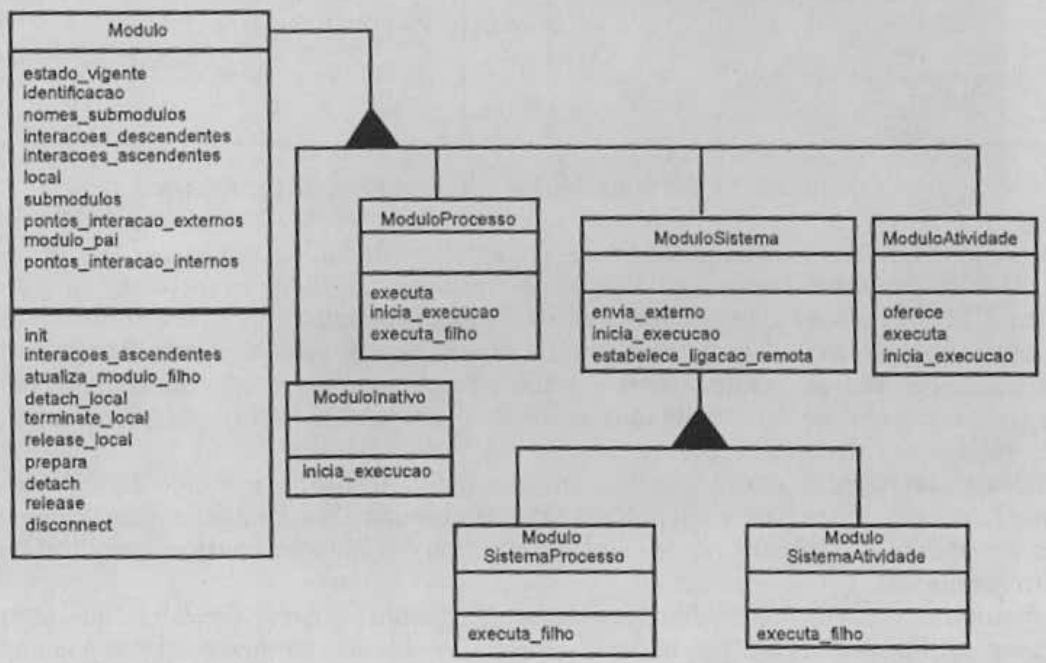

Figura 2.1 - Modelo do conceito arquitetônico Módulo

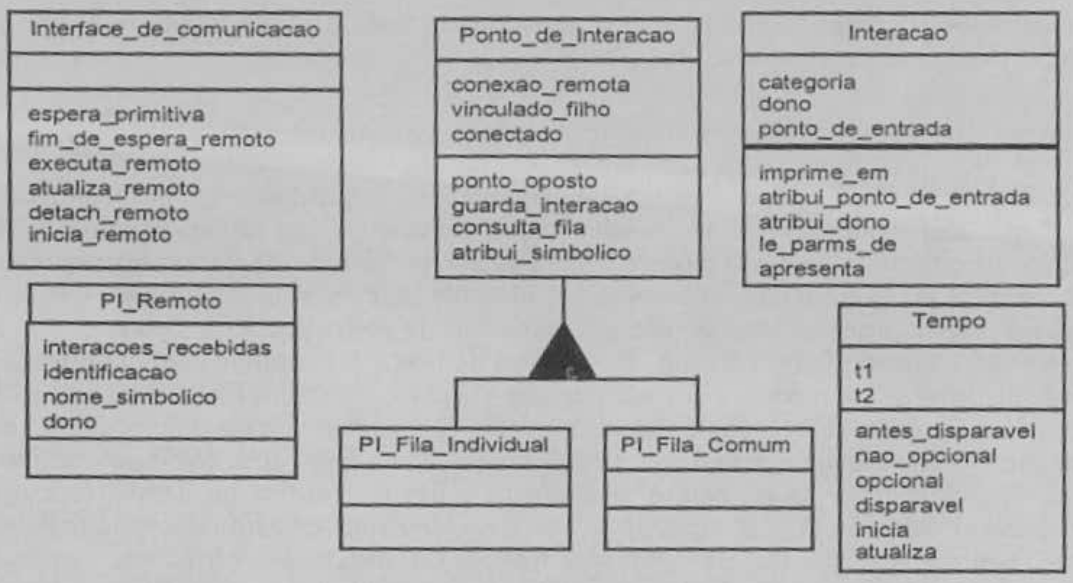

Figura 2.2 - Modelo para outros conceitos arquitetônicos

Seguindo a orientação dos modelos das Figuras 2.1 e 2.2, foram inicialmente encapsuladas as classes que aparecem em todos os programas C++ que implementem especificações Estelle. Essas classes, armazenadas em bibliotecas de prateleira, são referenciadas através de diretivas *#include* de C++. A Figura 2.3 mostra parte de uma destas bibliotecas, que corresponde à classe *Interacao* da Figura 2.2.

As classes de prateleira são referenciadas pelas classes que implementam aspectos específicos de cada especificação Estelle. Usando-se as classes de prateleira e as classes específicas obtém-se os correspondentes programas C++ correspondentes à especificação Estelle.

```
class Interacao {
   char *ponto_de_entrada, *dono;
   unsigned int categoria;
    Interacao() { ponto_de_entrada=dono=0 ; }
    ~Interacao() {
       if (ponto_de_entrada)
           delete(ponto_de_entrada);
       if (dono) delete(dono);     }
```

```
    void atribui_dono( char* x) ;
    void atribui_ponto_de_entrada( char* x);
    virtual istream& le_parms_de( istream& );
    virtual ostream& imprime_em( ostream& );
    virtual void apresenta( ostream&, int index=0);
    Interacao& operator=( Interacao& i) ;
    Interacao( Interacao& i);
};
```

Figura 2.3 - Parte da biblioteca de classes de prateleira

O aspecto dinâmico de um sistema especificado em Estelle é ditado por transições (ações) executadas pelas instâncias de módulos. Para se definir um módulo numa especificação Estelle, começa-se com a declaração de seu cabeçalho. Qualificadores colocados nos cabeçalhos dos módulos, juntamente com a forma de aninhamento dos mesmos, definem o paralelismo e o indeterminismo das transições de suas instâncias. Módulos com capacidade para realizar transições são chamados ativos e seus cabeçalhos devem obrigatoriamente ser qualificados. Módulos sem essa capacidade são chamados inativos e até podem ser qualificados, mas nesse caso os qualificadores não causam qualquer efeito.

Paralelismo assíncrono pode ocorrer apenas entre módulos qualificados como *ModuloSistemaProcesso* e *ModuloSistemaAtividade*, que possuem seu próprio ciclo de execução. Um ciclo de execução começa pela escolha de transições dos módulos definidos no seu interior e um novo ciclo só começa quando todas as transições escolhidas terminarem.

Módulos qualificados como *ModuloSistemaProcesso* e *ModuloProcesso* permitem que os módulos no seu interior disparem suas transições em paralelismo síncrono, no sentido em que cada um poderá executar uma e somente uma transição no ciclo de execução atual. Esses módulos podem ser refinados em combinações de módulos com atributo *MóduloProcesso* e *MóduloAtividade*.

Já os módulos com qualificadores *ModuloSistemaAtividade* e *ModuloAtividade* permitem a execução de uma transição de apenas um dos módulos interiores no ciclo de execução atual e a escolha da transição a ser executada é feita de forma indeterminística, a menos que o especificador tenha definido prioridades para as transições. Esses módulos só podem ser refinados em módulos com qualificador *ModuloAtividade*.

Os módulos definidos sem qualificador na especificação Estelle servem para auxiliar na estruturação da mesma, sendo mapeados na implementação como instâncias da classe *ModuloInativo*. Um módulo desse tipo, ao contrário dos outros, não realiza transições e portanto não tem a capacidade de dinamicamente criar, liberar e

mudar a configuração das conexões entre seus módulos-filho (módulos definidos no seu interior). Já um módulo ativo, tendo essa capacidade, atua como um tipo de supervisor sobre seus módulos-filho. Um módulo inativo pode conter qualquer tipo de módulo, porém um módulo com qualificador *ModuloSistemaProcesso* ou *ModuloSistema Atividade* só pode ter módulos inativos como ancentrais.

Cada módulo do sistema especificado, passível de ocupar um processo do sistema operacional, possui um programa principal (contendo uma rotina main de C++). Esse programa principal cria um objeto de uma classe descendente da classe *Modulo*, de mais alto nível na hierarquia de módulos implementada dentro desse processo. A partir desse ponto, o módulo cria a estrutura restante. A estrutura do programa principal depende do tipo de módulo no topo da hierarquia. Essa estrutura e os serviços especializados de cada tipo de módulo definem o dinamismo do sistema de acordo com a semântica operacional de Estelle.

Com relação à distribuição de processamento, é possível se especificar através de comentários especiais (embora não previsto em Estelle) em qual máquina um determinado módulo será executado. Em cada máquina onde for necessário executar um módulo da especificação Estelle, deverá haver um processo servidor de comunicação (um *daemon*, no caso do S.O. Unix). Dessa forma, um módulo poderá solicitar que o *daemon* remoto crie um novo processo (*fork*), carregue e execute (*execv*) o programa que implementa seu módulo filho. Para realizar o sincronismo entre módulos sendo executados em máquinas diferentes, é necessário um protocolo de sincronização, apresentado em [BAR 95b].

Para implementar a troca de mensagens entre os módulos, criou-se a classe Interface_de_Comunicação, cujos métodos utilizam os serviços de comunicação do sistema operacional (atualmente *sockets* Unix). Na implementação, os conceitos de alto nível de abstração ficaram em uma biblioteca enquanto que os detalhes dependentes da plataforma operacional ficaram em outra, aumentando a portabilidade do sistema. Assim, os recursos de comunicação via *sockets* (funções usando *socket*(), *bind*(), *listen*(), etc.) foram isolados em uma biblioteca, fornecendo serviços para a classe *Interface_de_Comunicação* e para o *daemon*. Toda vez que um módulo raiz de uma hierarquia de módulos é inicializado, é criado um fluxo de comunicação entre o processo ao qual este pertence e o processo ao qual pertence o seu módulo pai. Esse fluxo de comunicação é também utilizado pelos demais módulos desse processo.

3. Implementação do modelo proposto no ambiente Draco

O ambiente Draco emprega as idéias do paradigma Draco [NEI 80], o qual parte do princípio de que é possível desenvolver software baseado no reuso de abstrações de alto nível. Trata-se de um sistema transformacional orientado a domínios, onde especificações de software em linguagens de alto nível podem ser mapeadas automaticamente para linguagens executáveis. Um domínio para o Draco é constituído das seguintes partes:

- **linguagem**, cuja sintaxe é definida para permitir a escrita de programas do domínio. Para analisar os programas constroi-se um *parser* a partir da definição da gramática da linguagem, a qual tem associada ações semânticas para a geração de uma árvore de sintaxe abstrata (AST) das aplicações escritas no domínio. Essa AST particular é chamada Draco AST (DAST);
- bibliotecas de **transformações**. As transformações que mapeiam estruturas de uma linguagem em estruturas dessa mesma linguagem são chamadas de horizontais (intra-domínios), ao passo que as transformações que mapeiam estruturas de uma linguagem em estruturas de uma outra linguagem são chamadas de verticais (inter-domínios). Cada regra de transformação definida possui basicamente duas partes: o *Left hand side (Lhs)*, que descreve o padrão a ser encontrado nas descrições, e o *Right hand side (Rhs)*, que descreve o padrão de reescrita que substituirá a parte da descrição instanciada a partir do *Lhs*. Uma transformação na versão mais recente [LEI 95] da máquina Draco pode também disparar eventos e/ou alterar o fluxo de controle de aplicação de transformações através de pontos de controle, aos quais podem estar associados código para realizar essas tarefas. Os pontos de controle disponíveis na máquina Draco cobrem todas as possibilidades para o disparo de ações que estejam presentes em qualquer sistema transformacional.
- **prettyprinter**, que é um *unparser* responsável por mapear representações em sintaxe abstrata para a sintaxe concreta da linguagem, ou seja, que exibe a DAST da aplicação na linguagem de domínio.

A máquina Draco, que implementa as idéias do paradigma Draco, possibilita a produção automática de um programa executável a partir da especificação na linguagem do domínio da aplicação. Dada uma aplicação, o processo para sua transformação é iniciado pela verificação de sua sintaxe pelo *parser* do domínio. Outros *parsers* podem ser chamados de acordo com a necessidade, para produzir, juntamente com o *parser* principal, a chamada DAST. Sobre a DAST gerada são aplicadas as transformações das bibliotecas do domínio que ou otimizam a especificação no próprio domínio ou a transformam para outros domínios existentes no Draco. A Figura 3.1 mostra genericamente como uma especificação de entrada em um domínio é automaticamente transformada para outra especificação de saída no mesmo ou em outro domínio.

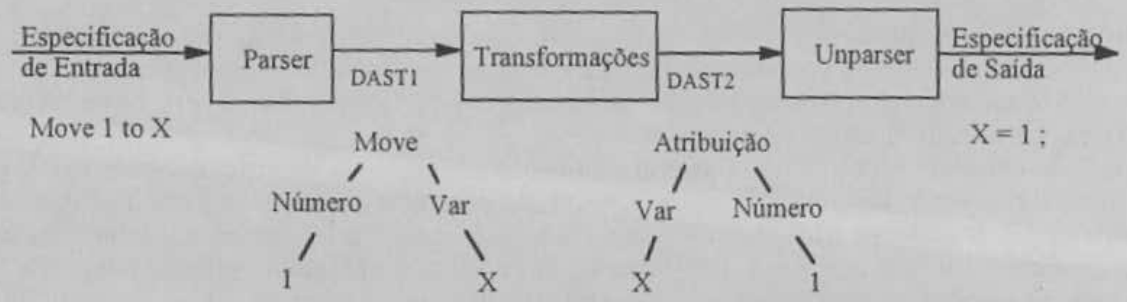

Figura 3.1 - Transformação usando Draco

Para possibilitar a transformação de especificações do domínio Estelle para o domínio executável C++, foi necessária a descrição desses dois domínios no ambiente Draco, através da linguagem e do *prettyprinter*. Para a definição das gramáticas de Estelle e de C++ foram utilizadas as especificações citadas em [ISO 88] e [STR 91] respectivamente. A Figura 3.2 mostra parte da gramática Estelle onde podem ser observadas, ao lado das regras, as ações semânticas (e.g., MakeNode) para construção das DASTs. Definidas essas gramáticas, foram gerados os *parsers* das linguagens e *unparsers* correspondentes aos *prettyprinters*, que analisam programas Estelle e C++ e constroem suas DASTs.

```
.....
e_mod_hdr_defn    : _DT_MODULE def_id e_mod_class p_nopoint_formal_part ';' e_ext_ip_decl_part
      e_exp_var_decl_part _DT_END
                              { $$=MakeNode("e_mod_hdr_defn1", $2, $3, $4, $6, $7, NULL); }
          | _DT_MODULE error _DT_END
                              { $$=MakeNode("e_mod_hdr_defn2", $2, NULL); }
          ;
e_mod_class       :           { $$=MakeNode("e_mod_class1", NULL); }
          | SYSTEMPROCESS     { $$=MakeNode("e_mod_class2", MakeLeaf($1), NULL); }
          | SYSTEMACTIVITY    { $$=MakeNode("e_mod_class3", MakeLeaf($1), NULL); }
          | PROCESS           { $$=MakeNode("e_mod_class4", MakeLeaf($1), NULL); }
          | ACTIVITY          { $$=MakeNode("e_mod_class5", MakeLeaf($1), NULL); }
          ;
.....
e_mod_body_heading : _DT_BODY def_id _DT_FOR appl_id ';'
                          { $$=MakeNode("e_mod_body_heading1", $2, $4, NULL); }     ;
```

Figura 3.2 - Parte da gramática Estelle

A Figura 3.3 mostra parte do *unparser* para o domínio Estelle. As ações à direita ("MODULE" #1, etc) são responsáveis em exibir a DAST na sintaxe Estelle.

```
...
e_mod_hdr_defn1       = "MODULE" #1 #2 #3
      ";" #4 #5 "END" ;
...
e_mod_hdr_defn2       = "MODULE" #1 "END" ;
e_mod_class2_SPEC#    = "SYSTEMPROCESS" ;
e_mod_class3_SPEC#    = "SYSTEMACTIVITY" ;
e_mod_class4_SPEC#    = "PROCESS" ;
e_mod_class5_SPEC#    = "ACTIVITY" ;
e_exp_var_decl_part1_SPEC# = ;
e_exp_var_decl_part2= "EXPORT" #1 ;
...
e_trans_clause1= "FROM" #1 ;
e_trans_clause2= "TO" #1 ;
e_trans_clause3= "PROVIDED" #1 ;
e_trans_clause4= "WHEN" #1 "." #2  #3 ;
e_trans_clause5= "DELAY" "(" #1 ")"    ;
```

Figura 3.3 - Parte do unparser para Estelle

Para o domínio Estelle, foram construídas também as bibliotecas de transformações verticais, que mapeiam especificações Estelle para C++ conforme os modelos das Figuras 2.1 e 2.2. A seguir será apresentado o processo de transformação de especificações Estelle para C++..

4. Transformação de especificações Estelle para C++

O processo de transformação começa pela análise de especificações Estelle pelo *parser* da linguagem Estelle. Estando sintaticamente correta, automaticamente é gerada a DAST da especificação Estelle e sobre essa DAST são aplicadas transformações que geram uma nova DAST na linguagem alvo C++. Usando o *unparser* da linguagem C++, obtém-se o código final transformado. Para mostrar com mais detalhes este processo de tranformação será utilizado um estudo de caso apresentado a seguir.

4.1 Estudo de caso - o Jogo do Demônio

O exemplo escolhido, embora simples, ilustra bem os conceitos importantes de Estelle, tais como refinamento, paralelismo e comunicação entre módulos. O *Jogo do Demônio* é um jogo de adivinhação onde uma entidade denominada *DEMONIO* gera tiros aleatoriamente. O objetivo do jogo é adivinhar quando o número de tiros é ímpar. Vários jogadores podem participar simultaneamente, sendo que as ações de um jogador não afetam os jogos dos demais. Os jogadores não precisam estar na mesma máquina ou sistema operacional, o que caracteriza um ambiente distribuído.

A arquitetura Estelle desse jogo, contendo os módulos *DEMONIO, MAQUINA, JOGADOR, GERENTE, JOGO, DISTRIBUIDOR*, os pontos de interação *G, D, P*, e os diversos níveis de refinamentos realizados, é apresentada na Figura 4.1. *Score, Venceu, Chute, Result, Novojogo, Fimjogo* e *Tiro* são as interações trocadas entre esses módulos.

O jogador tenta adivinhar o número de tiros gerados enviando um *Chute* ao sistema. O sistema responderá com *Ganhou* caso o jogador adivinhe e em caso contrário responderá com *Perdeu*. O sistema mantém um *Score*, inicializado com 0, para cada jogador. A cada *Ganhou* este Score é incrementado em uma unidade e para cada *Perdeu* este é decrementado em uma unidade. Cada jogador pode solicitar o *Result* do seu jogo, sendo que o sistema responderá com o seu *Score*.

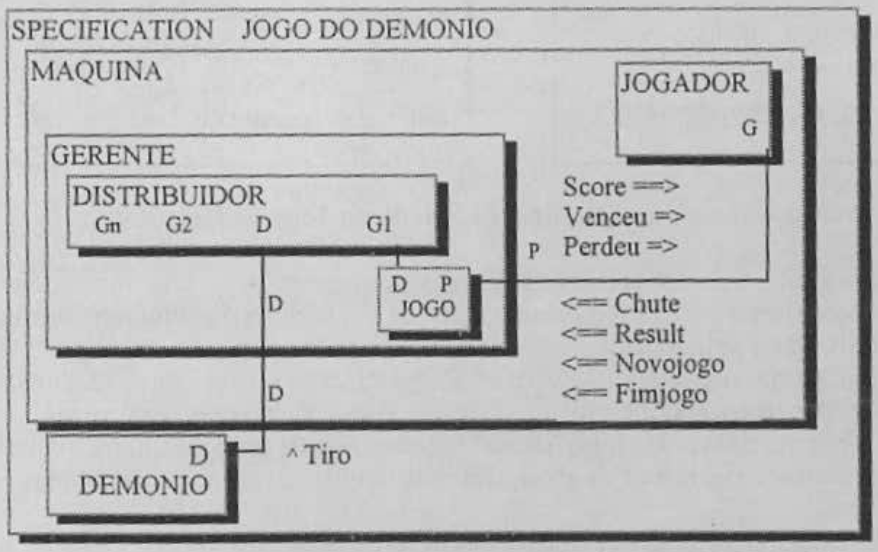

Figura 4.1 - Arquitetura Estelle do Jogo do Demônio

Para iniciar uma partida, um jogador deve realizar um *login* através de *Novojogo*. Neste momento, o sistema aloca um identificador único para esse jogador. Um *logout*, através de *Fimjogo*, desaloca o identificador desse jogador independentemente de outros identificadores que estão sendo utilizados pelo mesmo jogador.

Um módulo Estelle é constituído de um cabeçalho, que define o seu tipo e de um corpo, que define o seu comportamento. A expressão instância de módulo indica a associação, através da cláusula *init*, de um corpo a um cabeçalho. Em Orientação a Objeto (OO), essa mesma expressão indica a criação de um objeto de uma classe. Para eliminar essa ambiguidade, será utilizada a expressão "instância da classe ..." quando o assunto for pertinente a OO, mantendo a expressão "instância do módulo ..." quando se tratar de Estelle.

Na Figura 4.1, o módulo *GERENTE* assegura a criação e a eventual remoção das instâncias do módulo *JOGO*, em função das solicitações efetuadas pelos jogadores. Uma vez estabelecido um *JOGO*, o *GERENTE* vincula (*attach*) um canal *JOGOSERVIDOR* ao novo *JOGO*, de tal forma que as demais interações entre o *JOGADOR* e o *JOGO* não precisem da mediação do *GERENTE*. Uma variável exportada *FIM* é utilizada por um *JOGO*, para indicar ao seu pai (*GERENTE*) que seu *JOGADOR* terminou de jogar e, portanto, deve ser removido (*release*).

A Figura 4.2 mostra parte da especificação Estelle do Jogo do Demônio, correspondente à arquitetura ilustrada na Figura 4.1. Nessa especificação, as palavras reservadas de Estelle estão em letras minúsculas, enquanto que os identificadores estão em maiúsculas.

```
specification JOGO_DO_DEMONIO;
 module DEMONIO_CABEC systemprocess;
    ip D: DEMONIOSERVIDOR (PROVEDOR)
        individual queue;
 end; { DEMONIO_CABEC }
 body DEMONIO for DEMONIO_CABEC;
    { transições }
 end;

 module MAQUINA_CABEC;
    ip   D: DEMONIOSERVIDOR( USUARIO)
       individual queue;       end;
 body MAQUINA for MAQUINA_CABEC;
    const NUMJOGOS = 2;
    channel JOGOSERVIDOR
        (JOGADOR, MAQUINA);
      by JOGADOR:  CHUTE;  RESULT;
        NOVOJOGO;  FIMJOGO;
      by MAQUINA: VENCEU;  PERDEU;
          SCORE (VITORIAS: integer);
 module JOGADOR_CABEC systemprocess;
      ip G: JOGOSERVIDOR (JOGADOR)
       individual queue;
 end; { JOGADOR }
 body JOGADOR for JOGADOR_CABEC;
      {... }end;
 module GERENTE_CABEC systemprocess;
      ip  P: array [1..NUMJOGOS] of
          JOGOSERVIDOR (MAQUINA)
                common queue;
          : DEMONIOSERVIDOR (USUARIO)
                common queue;
 end; { GERENTE_CABEC }
 body GERENTE for GERENTE_CABEC;
```

```
 module DISTRIBUIDOR_CABEC process;
      ip  G: array [1..NUMJOGOS] of
        DEMONIOSERVIDOR (PROVEDOR)
          common queue;
        D: DEMONIOSERVIDOR (USUARIO)
              common queue;
        end; { DISTRIBUIDOR_CABEC }
  body DISTRIBUIDOR for
DISTRIBUIDOR_CABEC;
    { transições }
end; { DISTRIBUIDOR }
    module JOGO_CABEC process;
      ip  P: JOGOSERVIDOR (MAQUINA)
              common queue;
        D: DEMONIOSERVIDOR (USUARIO)
      common queue;
      export FIM : boolean ;   end;
            { JOGO_CABEC }
     body JOGO for JOGO_CABEC;
       var     CORRETOS: integer;
         state   PAR,  IMPAR;
         { inicialização }     end; { JOGO }
    end; { GERENTE }
    ...
    initialize     { criação do ambiente do jogo }
      begin init GERENTE_INSTANCIA with
                  GERENTE;
          attach D to
          GERENTE_INSTANCIA.D;
      end;     end;    {MAQUINA}
    ...
 initialize
   begin     {...}        end;
end. { specification JOGO DO DEMONIO }
```

Figura 4.2 - Parte da especificação Estelle do Jogo do Demônio

Conforme mostra a Figura 4.2, uma especificação Estelle começa pela palavra reservada *specification* e termina com um *end* e um ponto. Entre o *specification* e o *end*, a declaração dos módulos mostrados na Figura 4.1. Para os cabeçalhos é utilizada a palavra reservada **module**, enquanto que para os corpos é usada a palavra reservada **body**. Por exemplo, para o módulo DEMONIO foi especificado o cabeçalho DEMONIO_CABEC e o corpo DEMONIO. O cabeçalho desse módulo foi qualificado como *systemprocess*, o que implica que na implementação deverá haver uma classe *ModuloSistemaProcesso*. Tanto os cabeçalhos quanto os corpos de módulos terminam com a palavra reservada *end*. A especificação completa desse exemplo pode ser encontrada em [LOP 96].

Sobre esta especificação serão aplicadas as transformações descritas a seguir.

4.2 Transformação da especificação Estelle do Jogo do Demônio para C++

As bibliotecas de transformações são organizadas conforme mostram as Figura 4.3 e 4.4, para o caso do Jogo do Demônio. Declarações e inicializações globais são colocadas em *Global-Declaration* e *Global-Initialization*, respectivamente. Transformações são agrupadas em *Sets of Transforms*, que por sua vez são organizados em *Initialization*, *Transforms* e *End*. Na seção *Global-End* da biblioteca podem ser colocadas instruções que serão executadas após a aplicação de todos os *Sets of Transforms* contidos na mesma. Para facilitar o uso, as definições e inicializações globais mostradas na Figura 4.3 foram armazenadas em um arquivo (**globals.inc**), a ser referenciado através de diretiva *#include*.

Uma especificação Estelle pode conter vários módulos. Para cada módulo podem ser declarados vários corpos. No interior de cada corpo de módulo, podem ser declarados outros módulos, definindo assim uma hierarquia com diferentes níveis de profundidade.

O *Set of Transforms* **ShowModuleHierarchy** contém transformações responsáveis pela hierarquia dos módulos. Destacam-se neste *Set* as transformações **GetModuleHeader** e **GetBody**. **GetModuleHeader** reconhecerá os cabeçalhos de módulos existentes na especificação Estelle, gerando uma classe C++ para cada um. **GetBody** reconhecerá os corpos de módulos encontrados na especificação Estelle, também gerando uma classe C++ para cada um. **ShowModuleHierarchy** é chamado por outro conjunto de transformações, para implementar a hierarquia dos módulos de uma especificação Estelle. Para tal foi usado o método de disparo *Trigger:External* para indicar que **ShowModuleHierarchy** será disparado por outro conjunto de transformações.

```
// Arquivo pais.tfm
#include "globals.inc"
Set Of Transforms ShowModuleHierarchy

   Trigger: External
   Method
      Apply:  Single Step

Initialization: {{dast cpp.statement_list
   tem_corpo = 0 ; }}
      .....

Transform GetModuleHeader
   .....
```

```
Transform GetBody
   .....

Template ComposeAll
   Rhs: {{dast cpp.program
      [[ext_declaration *declarations]]
   }}
End: {{dast cpp.statement_list
   if (tem_corpo) {
    TEMPLATE("ComposeAll");
      MOVE(body_info.hxxfile,"declarations");
      PLACE_AT(body_info.hxxfile);
    END_TEMPLATE; } }}
```

Figura 4.3 - Conjunto de transformações ShowModuleHierarchy

O *Method Apply:Single Step* especifica a estratégia para a aplicação das transformações verticais ou horizontais. No caso trata-se de transformação vertical. O *Template* (transformação que contém o padrão de gravação) *ComposeAll* é chamado ao término da aplicação do conjunto **ShowModuleHierachy** para construir as classes correspondentes aos corpos de módulos da especificação Estelle.

A Figura 4.4 mostra parte do arquivo **globals.inc**, que contém as definições globais que permitem o intercâmbio de informações entre os *Set of Transforms*. As variáveis e funções auxiliares em *Global-Declaration* são visíveis a todo o conjunto de transformações **ShowModuleHierarchy.**

As definições relevantes de **globals.inc** e seus significados são listados a seguir:

- **pai, corpo e auxiliar** - pilhas do tipo pilha_de_pais para a determinação da relação hierárquica entre cabeçalhos e corpos de módulos. O campo *tipo* em cada elemento da pilha *pai* guardará a classe do módulo (process, activity, etc.).
- **depth, lastdepth e clastdepth** - usadas para se manter controle sobre a profundidade de aninhamento de cabeçalhos e corpos de módulos.
- **tchxxfile** - *string* que armazena o nome do arquivo que conterá a definição da classe em C++ gerada para representar um cabeçalho de módulo.
- **body_info** - estrutura para conter informações sobre um corpo de módulo.
- **tem_corpo** - sinaliza a presença de um corpo de módulo no interior do corpo de módulo sendo atualmente analisado.
- **Em_maiusculas (cadeiain, cadeiaout)** - função que copia o conteúdo do string *cadeiain* para o *string cadeiaout*, transformando todas as letras minúsculas em maiúsculas. Isso é necessário porque C++ é "*case sensitive*" e Estelle não.

```
// Arquivo globals.inc
Global-Declaration: {{dast cpp.statement_list
    typedef struct atributo {
        char nome[50];
        char tipo[50];    };
    class pilha_de_pais {
        struct atributo info [50] ;
        int top;
        public:
            pilha_de_pais() ;              // construtor
            void empilha(char wnome[50], char wtipo[50]) ;
            void desempilha() ;
            struct atributo* ultimo() ;   // retorna elemento do topo sem desempilhar
            int vazia() ;          };     // retorna verdadeiro se a pilha estiver vazia
    pilha_de_pais   pai , corpo, auxiliar ;
    int depth=1;              // profundidade de aninhamento de cabeçalho de módulo
    int lastdepth=1;          // profundidade do último cabeçalho de módulo encontrado
    int clastdepth=1;             // profundidade do último corpo de módulo encontrado
    char nome_do_pai[40]; // nome  do módulo pai do módulo corrente
    char tchxxfile[70];       // nome do arquivo tc.hxx a ser gerado
    .....
    struct {
        char hxxfile[70] ;    // nome do arquivo .hxx a ser gerado
        int  tem_initialize ;
        int  tem_transicao ;
        .....    } body_info ;
    int  tem_corpo;
    void Em_maiusculas( char * cadeiain, char * cadeiaout) { ..... }   }}
Global-Initialization: {{dast cpp.statement_list ...   }}
Global-End: {{dast cpp.statement_list   ....   }}
```

Figura 4.4 - Definições globais para as transformações

A transformação **GetModuleHeader**, pertencente ao conjunto de transformações **ShowModuleHierarchy**, é mostrada na Figura 4.5. **GetModuleHeader** reconhece a declaração de um cabeçalho de módulo Estelle e cria uma classe C++ para esse cabeçalho. Essa classe deve ser gravada no arquivo de trabalho (*workspace*) aberto para o corpo de módulo que contém o cabeçalho reconhecido. Por exemplo, quando **GetModuleHeader** encontra os cabeçalhos "module DISTRIBUIDOR_CABEC" e "module JOGO_CABEC", são gravadas as classes com esses nomes no arquivo _GERENTE.tc.hxx, pois o módulo GERENTE (Figura 4.1) contém os módulos DISTRIBUIDOR e JOGO.

```
Transform GetModuleHeader
    Lhs:
    {{dast estelle.e_body_decl
        module [[ID nome_do_cabecalho]]
            [[e_mod_class modclass]]      [[p_nopoint_formal_part no_point]] ;
            [[e_ext_ip_decl_part ips]]    [[e_exp_var_decl_part exp_vars]]
        end;
    }}
    Post-Match:  {{dast cpp.statement_list
    ..... // declarações de variáveis
    sprintf(nome_da_classe,"%s",expand("[[modclass]]"));
    if ( !strcmp(nome_da_classe,"e_mod_class1") )
        strcpy(nome_da_classe,"ModuloInativo");
    SET_LEAF_VALUE("nome_da_classe", nome_da_classe);
    .....
    // ****** CONTROLE DE ESCOPO DE CABEC. DE MÓDULOS
    for( ; depth <= lastdepth ; lastdepth--) // retira cabeçalhos irmãos da pilha
        pai.desempilha() ; // No caso de ser irmão do último módulo reconhecido
    // VERIFICACAO SEMÂNTICA : ANINHAMENTO
    Em_maiusculas(nome_da_classe, maiu_classe);
    erro_sem = ((!strcmp(maiu_classe, "SYSTEMPROCESS" ))   ||
            (!strcmp(maiu_classe,   "SYSTEMACTIVITY")))      &&
            (strcmp(pai.ultimo()->tipo,"ModuloInativo")) ;
```

```
if ( erro_sem )
    printf("ERRO SEMANTICO !! ==> Módulo SYSTEM só pode ter pai Inativo");
if ( depth >= lastdepth ) {  // empilha  o nome do cabeçalho corrente
        sprintf(nome_do_pai,"%s",expand("[[nome_do_cabecalho]]"));
        pai.empilha(nome_do_pai, nome_da_classe) ;
        lastdepth = depth ;  }
// **********************************
for ( ; depth <= clastdepth ; clastdepth--)  {
    auxiliar.empilha(corpo.ultimo()->nome, "");
    corpo.desempilha() ;        };                          // retira
sprintf(tchxxfile,"./output/_%s.tc.hxx", corpo.ultimo() );   // corpo pai do módulo atual
for ( ; !auxiliar.vazia() ; clastdepth++ ) {
    corpo.empilha(auxiliar.ultimo()->nome, "");             // devolve
    auxiliar.desempilha() ;  };
TEMPLATE("Mod_Header_Class");                               // invoca template
    TRANSPORT_VALUE("nome_do_cabecalho");                   // parâmetro
    TRANSPORT_VALUE("nome_da_classe");                      // parâmetro
    PLACE_AT(tchxxfile);                                    // grava no workspace do pai
END_TEMPLATE;
.....  }}
```

Figura 4.5 - Transformação GetModuleHeader

Os módulos *ModuloSistemaProcesso* e *ModuloSistemaAtividade* só podem ter módulos inativos como ancestrais. A verificação semântica sombreada na Figura 4.5 assegura que essa regra de aninhamento de módulos seja respeitada. A variável *nome_da_classe* contém o qualificador do cabeçalho do módulo que está sendo analisado e o objetivo dessa verificação é comparar o qualificador do cabeçalho de um módulo com o qualificador do cabeçalho do módulo que o contém.

Os nomes que aparecem entre [[e]] são variáveis Draco, ou seja, metavariáveis com tipos correspondentes às regras gramaticais de Estelle. Antes de aplicar a transformação **GetModuleHeader**, são aplicadas outras transformações que substituem essas metavariáveis por um padrão formatado da categoria sintática definida pelo tipo que aparece à sua esquerda. No caso da metavariável **nome_do_cabecalho** por exemplo, o tipo definido é **ID**.

A sintaxe das transformações é uma extensão de C++. Métodos auxiliares escritos em C++ podem ser utilizados normalmente nas transformações, como é o caso do método *expand*, que permite que uma variável Draco seja exportada como um *string*, para ser usado em uma variável C++.

Os trechos de programa na Figura 4.5 que usam a pilha *pai* (pai.empilha e pai.desempilha) implementam o controle do escopo de cabeçalhos de módulos. Dessa forma, é implementada a visibilidade dos cabeçalhos de módulos aninhados numa especificação Estelle.

Segue, na Figura 4.6, um exemplo de *Template*. No caso, **Mod_Header_Class** é responsável pela gravação da classe que representa um cabeçalho de módulo Estelle.

```
Template Mod_Header_Class

Rhs: {{dast cpp.ext_declaration.RC("common_types.tab")
    class [[IDENTIFIER nome_do_cabecalho]] : public [[CLASS_NAME  nome_da_classe]] {
    ...
    ....
    };

}}
```

Figura 4.6 - Template Mod_Header_Class

Os atributos e métodos da classe gerada por *GetModuleHeader* serão herdados pelas classes criadas pela transformação **GetBody**, apresentada na Figura 4.7. Para cada corpo de módulo é criado um arquivo para conter as classes que representam os cabeçalhos de módulo contidos nesse corpo. Por exemplo, quando **GetBody** encontra "body GERENTE for GERENTE_CABEC" na especificação Estelle, é aberto o arquivo _GERENTE.tc.hxx.

Para a gravação das classes correspondentes aos corpos de módulo reconhecidos na especificação Estelle, **GetBody** chama o *Template* **_MI_Struct**, passando-lhe como parâmetros as metavariáveis *nome_do_corpo* e *nome_do_cabec*, que contêm os nomes do corpo de um módulo e seu cabeçalho, respectivamente.

```
Transform GetBody

    Lhs:  {{dast estelle.e_body_decl
    body [[ID nome_do_corpo]] for [[ID nome_do_cabec]] ;
        [[e_body_decl *body_decls]]
        [[e_trans *trans]]
    end ;
    }}

    Post-Match: {{dast cpp.statement_list
    // declarações de variáveis
    tem_corpo = 1 ;
    sprintf(nome_do_cabec,"%s",expand("[[nome_do_cabec]]"));
    sprintf(nome_do_corpo,"%s",expand("[[nome_do_corpo]]"));
    .....
    // ******** CONTROLE DE ESCOPO DE CORPOS DE MÓDULOS
    for ( ; depth <= clastdepth ; clastdepth--)
        corpo.desempilha() ;                    // desempilha irmãos
    if ( depth >= clastdepth ) {
        corpo.empilha(nome_do_corpo, "") ;  // Empilhar, próximo corpo pode ser filho
        clastdepth = depth ; };
    .....
// ************************************************************
    sprintf(tchxxfile,"./output/_%s.tc.hxx", expand("[[nome_do_corpo]]"));
    CREATE_WORKSPACE(tchxxfile,"cpp");      // cria workspace próprio
    .....
    CREATE_WORKSPACE(body_info.hxxfile, "cpp");
    TEMPLATE("_MI_Struct");                         // chama o Template _MI_Struct
        TRANSPORT_VALUE("nome_do_corpo");   // parâmetro
        TRANSPORT_VALUE("nome_do_cabec");   // parâmetro
        MOVE(Variables, "variaveis") ;              // move workspace para metavariável
        PLACE_AT(body_info.hxxfile);
    END_TEMPLATE;
    .....
    depth++;
    TRANSFORM_VAR("body_decls", "ShowModuleHierarchy");
    depth--;
    .....
    }}
```

Figura 4.7 - Transformação GetBody

O T*emplate* **_MI_Struct**, mostrado na Figura 4.8, é chamado por *GetBody*, com o objetivo de implementar os aspectos de um módulo que são dependentes da implementação, gravando uma classe para cada corpo de módulo. Um detalhe importante desse *Template* é sua capacidade de decidir sobre a inclusão de determinados métodos e atributos na classe gerada. Por exemplo, a classe correspondente a um corpo de módulo só conterá o método "**seleciona_e_executa**" caso o módulo apresente transições Estelle. Outro exemplo é o método "**init**", que só fará parte da classe gerada se o módulo correspondente apresentar uma seção de inicialização (*initialize*).

```
Template _MI_Struct

  Pre-Apply: {{dast cpp.statement_list
  .... // declarações de variáveis locais
  sprintf(mi_nome_do_corpo, "__MI_%s", expand("[[nome_do_corpo]]"));
  sprintf(_nome_do_cabecalho, "_%s", expand("[[nome_do_cabec]]"));
  sprintf(__frame_nome_do_corpo, "__frame_%s", expand("[[nome_do_corpo]]"));

  if (body_info.tem_transicao)
      sprintf(wk_member_A,"%s","int seleciona_e_executa( int __dt ) ;");
  else
      sprintf(wk_member_A,"%s","");
  if (body_info.tem_initialize)
      sprintf(wk_member_B,"%s","int init() ; ");
  else
      sprintf(wk_member_B,"%s","");
  .....
  SET_LEAF_VALUE("mi_nome_do_corpo", mi_nome_do_corpo);
  .... // outros SET_LEAF_VALUE

  Rhs: {{dast cpp.ext_declaration.RC("common_types.tab")
    class [[IDENTIFIER mi_nome_do_corpo]] : [[CLASS_NAME _nome_do_cabecalho]] {
  [[CLASS_NAME __frame_nome_do_corpo]] __Miquadro;
  [[member_declaration* variaveis]]
  [[member_declaration wk_member_A]]
  [[member_declaration wk_member_B]]
  [[IDENTIFIER mi_nome_do_corpo]](istream& is) : [[IDENTIFIER _nome_do_cabecalho]] (is) {}
  [[IDENTIFIER mi_nome_do_corpo]]() : [[IDENTIFIER _nome_do_cabecalho]]() {}
  void mostra_LocVar(ostream& os, int idx);
  char** locVarNameList();
  void prepara_especifico();  };
  }}
```

Figura 4.8 - Template _MI_Struct

Diversos pontos de controle de aplicação de transformações disponíveis no Draco podem ser utilizados [LEI 94] para garantir e preservar a semântica de Estelle em C++. Os seguintes pontos de controle foram utilizados neste trabalho, demonstrando como o fluxo de controle de aplicação de transformações pode ser interceptado:

- *Lhs* - Descreve o padrão a ser encontrado nas descrições a serem transformadas. Uma vez que em Estelle a declaração de um cabeçalho de módulo é iniciada pela palavra reservada **module**, o processo de transformação deve reconhecer todas as ocorrências de **module,** contidas na especificação, e criar uma classe para cada uma, conforme mostrado na Figura 4.10. Assim por exemplo, a transformação **GetModuleHeader** é aplicada toda vez que é feita uma amarração da estrutura **module ... end;** com um trecho da especificação Estelle, como em **module** JOGO_CABEC ... **end;** da Figura 4.2.
- *Post-Match* - Permite a execução de atividades preparatórias da transformação logo após reconhecido o *Lhs*. No caso de **GetModuleHeader**, permite executar uma atividade de expansão de uma classe em arquivos com extensão **.tc.hxx** logo após reconhecida uma declaração de cabeçalho de módulo.
- *Pre-Apply* - Executado imediatamente antes da substituição do trecho reconhecido no *Lhs* pelo padrão descrito no *Rhs*. A expansão das classes para corpos de módulos nos arquivos **.hxx** é feita através do *Template* **_MI_Struct**. *Templates*, no Draco, são transformações que definem apenas sua fase de aplicação (*Pre-Apply* + *Rhs* + *Post-Apply*). No *Template* **_MI_Struct**, o ponto de controle *Pre-Apply* é responsável por tratar os parâmetros de entrada **nome_do_corpo** e **nome_do_cabec** para que as variáveis derivadas desses nomes possam ser disponibilizadas. Nesse ponto de controle também se decide sobre a inclusão de determinados métodos na classe a ser gerada, em função de análises realizadas em outros conjuntos de transformações. **SET_LEAF_VALUE** permite que uma variável Draco seja vinculada a um valor definido por um *string*, armazenado em uma variável C++. Dessa forma, o par **expand / SET_LEAF_VALUE** atua como uma via de comunicação de dados entre o Draco e o C++.
- *Rhs* - Define o padrão de reescrita que substitui parte da descrição instanciada a partir do *Lhs*. No caso da especificação **body** JOGO **for** JOGO_CABEC, o **Rhs** cria a *class* *__MI_JOGO* derivada de *JOGO_CABEC*, conforme a Figura 4.9. Toda classe gerada para implementar um corpo de módulo deve ter o prefixo **__MI_** (e.g., o corpo de módulo *JOGADOR* gera a classe *__MI_JOGADOR*).

Para cada corpo de módulo é gerado um arquivo com extensão **.hxx**. A Figura 4.9 mostra parte da classe gerada pelo transformador para implementar o corpo do módulo JOGO do *Jogo do Demônio*.

```
class __MI_JOGO : _JOGO_CABEC {
    __quadro_JOGO __Miquadro;
   int CORRETOS;
   int seleciona_e_executa( int __dt ) ;
   int init() ;
   __MI_JOGO(istream & is) : _JOGO_CABEC(is)   {}
   __MI_JOGO() : _JOGO_CABEC()   {}
   void mostra_LocVar(ostream & os,int idx);
   char ** locVarNameList();
   void prepara_especifico();
} ;
```

Figura 4.9 - Arquivo _JOGO.hxx gerado segundo o *Template* _MI_Struct

Além dos métodos construtores, com os mesmos nomes da classe, tem-se na classe gerada outros métodos e atributos que completam a implementação específica de um corpo de módulo. Assim por exemplo, o atributo **__MIquadro**, do tipo **__quadro_JOGO**, é específico para o módulo JOGO. Já o método **prepara_especifico** implementa a maneira particular com que cada módulo será inicializado. O método **seleciona_e_executa** escolhe e executa uma das transições do módulo JOGO. Os métodos **locVarNameList** e **mostra_LocVar** definem e apresentam variáveis locais de Estelle, respectivamente.

A Figura 4.10 mostra parte do arquivo __GERENTE.tc.hxx com as declarações de classes geradas pelo transformador para os cabeçalhos dos módulos DISTRIBUIDOR e JOGO. As classes geradas DISTRIBUIDOR_CABEC e JOGO_CABEC correspondentes aos corpos de módulos aninhados no corpo do módulo GERENTE herdam da classe de prateleira ModuloProcesso.

```
class DISTRIBUIDOR_CABEC : public ModuloProcesso{
   ...
};

class JOGO_CABEC  : public ModuloProcesso{
   ...
};
```

Figura 4.10 - Arquivo _GERENTE.tc.hxx gerado

Para completar o processo de transformação de Estelle para C++, outras transformações foram construídas, da mesma forma que estas aqui apresentadas. Para que se tenha uma idéia geral das classes geradas para o *Jogo do Demônio* e das estruturas que ligam estas classes, serão apresentadas a seguir os modelos orientados a objetos desta aplicação na notação do método FUSION, que possibilita uma representação em mais alto nível, facilitando o entendimento.

4.3 Modelos Orientados a Objetos da especificação Estelle do Jogo do Demônio

A implementação em C++ foi orientada pelos modelos dos conceitos arquitetônicos mostrados nas Figuras 2.1 e 2.2 e preservou a semântica de que uma especificação Estelle é composta de um conjunto de módulos que se comunicam através de pontos de interação. Assim, como resultado da implementação do *Jogo do Demônio* foram obtidos os modelos orientados a objetos das Figuras 4.11, 4.12 e 4.13, onde aparecem sombreadas as classes de prateleira anteriormente apresentadas nas Figuras 2.1 e 2.2.

A Figura 4.11 mostra as classes derivadas da classe de prateleira *ModuloProcesso*, que instanciam os módulos *JOGO* e *DISTRIBUIDOR*. Na hierarquia deste modelo tem-se que *__MI_DISTRIBUIDOR* e *__MI_JOGO* são classes derivadas de *DISTRIBUIDOR_CABEC* e *JOGO_CABEC*, respectivamente.

Objetos da classe *DISTRIBUIDOR_CABEC* possuem 2 ou mais ocorrências do ponto de interação *G* e apenas uma ocorrência do ponto de interação *D*. Objetos da classe *__MI_DISTRIBUIDOR* herdam automaticamente atributos e serviços das classes *DISTRIBUIDOR_CABEC* e *ModuloProcesso*. Esta herança inclui os pontos de interação *D* e *G*. Da mesma forma objetos da classe *__MI_JOGO* herdam os pontos de interação D e P.

Filas de Estelle podem ser individuais ou comuns aos pontos de interação. Para os módulos *DISTRIBUIDOR_CABEC* e *JOGO_CABEC*, todos os pontos de interação são do tipo *PI_Fila_Comum*, conforme pode ser verificado na especificação Estelle da Figura 4.2.

Para instanciar o módulo *JOGO*, é declarada a classe *__MI_JOGO*, de *JOGO_CABEC*, que possui dois pontos de interação *D* e *P*, do tipo *PI_Fila_Comum*.

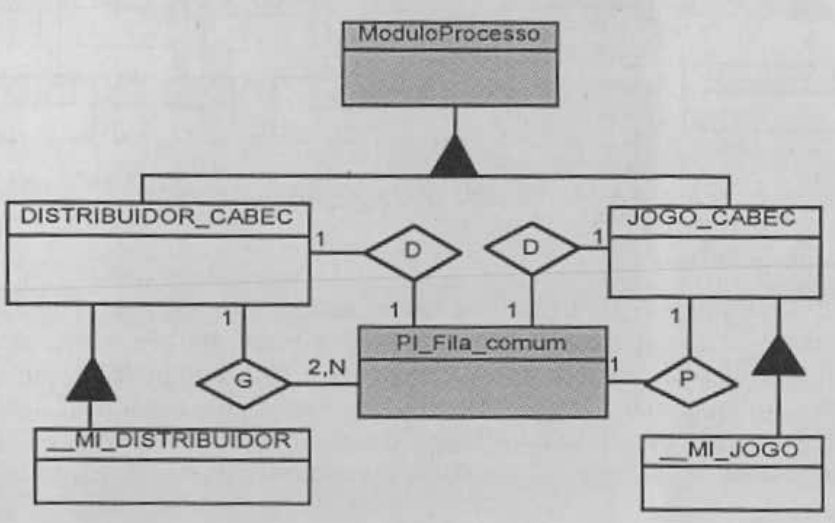

Figura 4.11 - Modelo do jogo para classes derivadas de *ModuloProcesso*

A Figura 4.12 apresenta três classes derivadas da classe de biblioteca *ModuloSistemaProcesso*: *DEMONIO_CABEC*, *JOGADOR_CABEC* e *GERENTE_CABEC*. Estas representam os cabeçalhos dos módulos *DEMONIO*, *JOGADOR* e *GERENTE* respectivamente, e servem como base para as classes *__MI_DEMONIO*, *__MI_JOGADOR* e *__MI_GERENTE*. Os pontos de interação *D* e *G* pertencentes a *DEMONIO_CABEC* e *JOGADOR_CABEC* são do tipo *PI_Fila_Individual*, ao passo que os pontos *P* e *D* pertencentes a *GERENTE_CABEC* são do tipo *PI_Fila_Comum*.

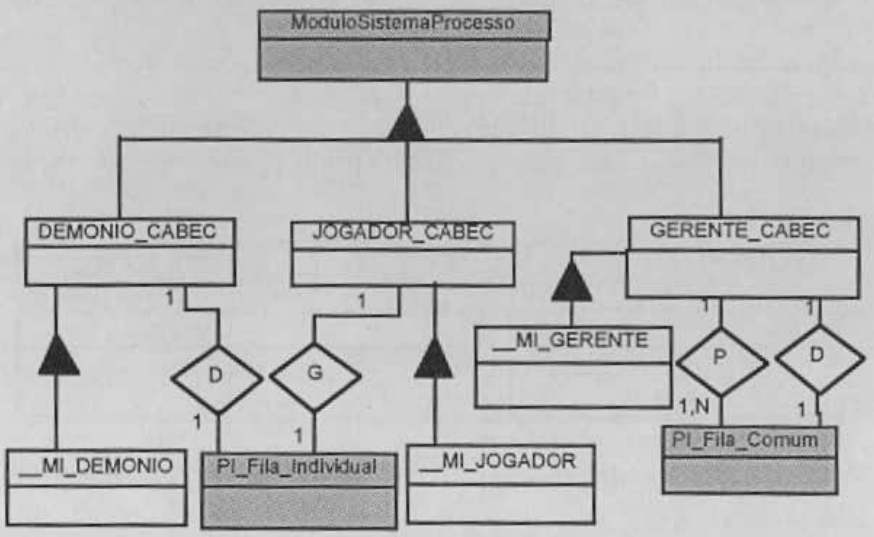

Figura 4.12 - Modelo do Jogo para classes derivadas de *ModuloSistemaProcesso*

Finalmente, a Figura 4.13 mostra as classes *MAQUINA_CABEC* e *__MI_JOGO_DO_DEMONIO* correspondentes aos módulos *MAQUINA* e *JOGO DO DEMONIO*, herdando atributos e métodos da classe de prateleira *ModuloInativo*. Nessa figura, aparecem também as classes derivadas de *Interacao* e a classe derivada de *Interface_de_comunicacao*, que são necessárias à implementação.

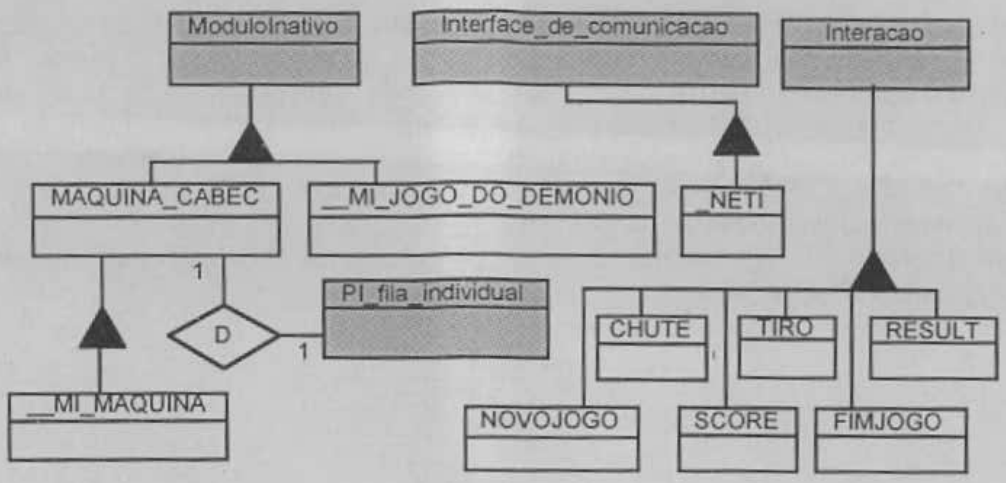

Figura 4.13 - Modelo do jogo para outras classes

4.4 Resultados e Comparações

Um levantamento das ferramentas para Estelle é apresentado em [BAR 95b], que citando [LIN 92], aponta genericamente o percentual de automatização dos compiladores para Estelle em cerca de 70%. Em experiências posteriores a [BAR 95b], citadas em [FAR 96], mostrou-se que o PetDingo pode chegar a percentuais maiores de automatização. Para um caso específico, chegou-se a obter um percentual de automatização de 90%.

O quadro da Figura 4.14 e as considerações a seguir comparam o Draco com o PetDingo, que é a ferramenta que apresenta atualmente o maior índice de automatização na transformação de especificações Estelle para uma linguagem executável.

Uma forma de reduzir a intervenção manual consiste em aproximar os níveis de abstração das linguagens fonte e destino. Diminuir o nível de abstração de Estelle é inconveniente, pois trata-se de uma TDF, que deve permitir a produção de especificações independentes das implementações, e também implicaria na modificação de um padrão. A solução foi obtida pelas facilidades de implementação oferecidas pela máquina Draco. Interagindo com a máquina Draco, o desenvolvedor pode, se necessário, tomar decisões em tempo de aplicação das transformações.

A ferramenta considerada, o PetDingo, é constituído de 2 produtos: o PET, um front-end que gera uma representação intermediária dos objetos encontrados na especificação, e o DINGO, que recupera esses objetos, gerando o código C++. Com a abordagem do Draco, a transformação está sendo realizada em uma única passagem.

O analisador sintático utilizado pelo PetDingo é o Bison, que reconhecidamente apresenta alguns problemas quanto à resolução de conflitos gramaticais, conforme [FRE 96]. Esses problemas foram resolvidos na máquina Draco.

Ainda, conforme mostra o quadro da Figura 4.14, no Draco, a linguagem alvo, para a qual as especificações Estelle são transformadas, pode ser C++ ou qualquer outra linguagem, permitindo uma grande portabilidade das especificações para diferentes plataformas. O programa gerado no Draco atende aos princípios da orientação a objetos, o que não só permite o reuso, como também facilita a integração com outros softwares orientados a objetos.

FERRAMENTA	LINGUAGEM ALVO	AUTOMATIZAÇÃO	ORIENTAÇÃO A OBJETOS
PetDingo	C++	90%	Parcial
Draco	C++ ou qualquer linguagem de programação	100%	Sim

Figura 4.14 - Quadro comparativo entre o Draco e o PetDingo

5. Conclusão

Como um dos resultados desse trabalho, definiu-se uma estratégia de implementação automática de especificações Estelle usando a ferramenta Draco. A importância desta estratégia vem do uso de um sistema transformacional que possibilita a implementação de Estelle para outras linguagens diferentes de C++. Pode-se por exemplo definir bibliotecas de transformações que transformem especificações Estelle para Pascal ou qualquer linguagem definida no Draco. A máquina Draco agrega várias técnicas de Engenharia de Software, apresentando uma versatilidade, conforme parcialmente demonstrado, superior aos meta-compiladores mais conhecidos.

O uso da ferramenta Draco no desenvolvimento de software é importante porque possibilita a geração automática do código a partir de especificações de alto nível. A manutenção dos sistemas gerados no Draco fica

facilitada, mesmo em Sistemas Distribuidos com diferentes protocolos, arquiteturas, sistemas operacionais e bancos de dados, porque o projetista pode realizar a tarefa de manutenção no nível de especificações da TDF Estelle.

Embora já existam alguns compiladores Estelle que permitem a geração semi-automática do código a partir das especificações, nossa contribuição vem do uso de uma tecnologia moderna [PRA 92, LEI 95], que pode ser ampliada e aperfeiçoada para viabilizar todo o processo de automatização.

Resultados importantes já foram obtidos usando a estratégia apresentada. O ambiente Draco tem se mostrado poderoso para permitir a transformação automática de modelos orientados a objetos de especificações Estelle para C++, conforme mostrado no caso do *Jogo do Demônio*. Uma ampla biblioteca de transformações já foi construída, o que possibilita, atualmente, a transformação automática de grande parte das especificações Estelle para C++. Trabalhos estão sendo desenvolvidos no sentido de ampliar essa biblioteca, visando implementar qualquer especificação Estelle.

Como trabalhos futuros temos: a ampliação da biblioteca de transformações para permitir a implementação de SDs mais complexos, a criação de outras bibliotecas para implementação em novas linguagens e sistemas operacionais, e a utilização de outras arquiteturas, tais como CORBA[OMG 92] e DCE[OSF 92], para suportar a computação distribuída.

6. Agradecimentos

Nossa gratidão ao colega Marcelo Sant'Anna, do projeto Draco-PUC, que forneceu o suporte não só necessário como decisivo para a compreensão dos detalhes operacionais da atual versão do ambiente Draco.

Este projeto teve ajuda parcial do CNPq, órgão brasileiro de fomento à pesquisa, e da Fundação Paulista de Tecnologia e Educação.

7. Referências

[BAR 95a] Barbosa, C.B. & Lopes de Souza, W. *Modelagem Orientada ao Objeto de Especificações Escritas em Estelle.* Anais do 13° Simpósio Brasileiro de Redes de Computadores, Belo Horizonte (MG), 22-26/05/1995, pp. 23-39.

[BAR 95b] Barbosa, C.B. *Modelagem orientada a objetos de especificações Estelle*, Dissertação de mestrado, Programa de Pós-Graduação em Ciência da Computação (PPG-CC), Universidade Federal de São Carlos (UFSCar), Jul 1995.

[EWS 89] ESPRIT Project 1265 - SEDOS. *EWS User's manual.* Bruxelas (Bélgica), 1989.

[FAR 96] Farias, C.R.G. & Lopes de Souza, W. & Barbosa, C.B. *Design and Implementation of Distributed Databases Using Estelle.* Anais do 11° Simpósio Brasileiro de Bancos de Dados, São Carlos(SP), 14-16/10/1996, pp.158-171.

[FRE 96] Freitas, F.G & Leite, J.C.S. & Sant'Anna, M. *Aspectos Implementacionais de um Gerador de Analisadores Sintáticos para o Suporte a Sistemas Transformacionais.* I Simpósio Brasileiro de Linguagens de Programação, B.Horizonte(MG), Set/1996.

[FUS 94] Coleman, D. et al *Object-Oriented Development - The Fusion Method*, Prentice Hall, 1994.

[GER 83] GERBER,G.W. *Une Méthode d'Implantation Automatisée de Systèmes Specifiés Formelement.* Dissertação de Mestrado, Université de Montréal, 1983.

[ISO 83a] ISO IS 7489. *Information Processing Systems - Basic Reference Model of Open Systems Interconnection.* 1983.

[ISO 83b] ISO IS 7185. *Programming Language Pascal.* 1983.

[ISO 88] ISO IS 9074. *Information Processing Systems - Open System Interconnection - Estelle - A Formal Description Technique based on an Extended State Transition Model.* 1988.

[LEI 94] Leite, J.C.S. & Freitas, F.G. & Sant'Anna, M. *Draco-PUC: a Technology Assembly for Domain Oriented Software Development.* 3rd. IEEE International Conference of Software Reuse, Rio de Janeiro (RJ), Nov/1994.

[LEI 95] Leite, J.C.S. & Prado, A.F. & Freitas, F.G. & Sant'Anna, M. *O Uso do Paradigma Transformacional no Porte de Programas Cobol*, 9° Simpósio Brasileiro de Engenharia de Software, Recife (PE), Out/1995.

[LIN 92] Lin, C.W. *Uma metodologia para implementação semi-automática de protocolos de comunicação.* Dissertação de Mestrado, Universidade Federal da Paraíba(UFPb), Campina Grande (PB), 1992.

[LOP 88] Lopes de Souza, W. & Ferneda, E. *A Compiler for a Formal Description Technique.* Computer Communications Systems, North-Holland, 1988, pp. 275-286.

[LOP 89] Lopes de Souza, W. *Estelle: uma técnica para a descrição formal de serviços e protocolos de comunicação.* Revista Brasileira de Computação, Vol.5 N.1, Jul/Set 1989, pp. 33-44.

[LOP 96] Lopes de Souza, W. & Prado, A. F. & Santana, A.C.L. *Implementação Automática de Especificações Estelle: um estudo de caso.* RT-DC 001/96, Departamento de Computação (DC), Universidade Federal de São Carlos (UFSCar), 1996.

[NEI 80] Neighbors, J. *Software Construction Using Components.* Tese de Doutorado, University of California at Irvine (EUA), Set/1980.

[OMG 92] Object Management Group and X/Open *The Common Object Request Broker: Architecture and Specification.* Document Number 91.12.1, 1992.

[OSF 92] Open Software Foundation *DCE Application Development Guide.* Cambridge, MA (EUA), 1992.

[PRA 92] Prado, A.F *Estratégia de Re-Engenharia de Software Orientada a Domínios.* Tese de Doutorado, Pontifícia Universidade Católica, Rio de Janeiro, Ago/1992.

[SIS 91] Sijelmassi, R. & Strausser, B. *The Distributed Implementation Generator: an overview and user guide.* Technical Report NCSL/SNA, Gaithersburg (EUA), 1991.

[STR 91] Stroustrup B. *The C++ Programming Language*, Addison-Wesley, 1991.

[VUL 88] Vuong, S.T & Lau, A.C & Chan, R.I. *Semiautomatic Implementation of Protocols Using an Estelle-C Compiler.* IEEE Transactions on Software Engineering, vol.14 (13), Mar 1988, pp. 384-393.