UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME SALUM RANGEL

ProTool: uma Ferramenta de Prototipação de Software para o Ambiente PROSOFT

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Daltro José Nunes
Orientador

Porto Alegre, agosto de 2003

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rangel, Guilherme Salum

ProTool: uma Ferramenta de Prototipação de Software para o Ambiente PROSOFT / Guilherme Salum Rangel. – Porto Alegre: Programa de Pós-Graduação em Computação, 2003.

223 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2003. Orientador: Daltro José Nunes.

1. Prototipação de Software. 2. Especificação Algébrica. 3. PROSOFT. 4. OBJ. I. Nunes, Daltro José. II. Título.

“Deus nos fez perfeitos e não escolhe os capacitados,capacita os escolhidos.
Fazer ou não fazer algo só depende de nossa vontade e perseverança.”
— ALBERT EINSTEIN

AGRADECIMENTOS

Agradeço a Deus pela vida, pelas oportunidades, pela proteção e pelas luzes para superar os momentos mais difíceis.

Aos meus pais e familiares, agradeço pela educação, carinho e estímulo.

Agradeço a Carina Cipolat, pelo amor, compreensão, amizade e incentivo na vida.

Agradeço ao meu orientador, Prof. Daltro José Nunes, pela oportunidade ímpar de ser seu aluno no mestrado. O considero um exemplo distinto, tanto como pessoa quanto como grande pesquisador engajado na vida acadêmica. Seu estilo que me fez descobrir o verdadeiro sentido da pesquisa e culminou na decisão por aprofundar meus estudos no doutorado. Gostaria de expressar minha gratidão também pelo incentivo e confiança no meu trabalho.

Aos membros do grupo PROSOFT pelas sugestões para aprimorar este trabalho. Em especial, gostaria de agradecer ao pesquisador Heribert Schlebbe, da Universidade de Sttutgart, pelo auxílio na implementação e pela excelente temporada em Stuttgart.

Aos professores Wilfried Brauer, da TU-München, Klaus Madlener, da Universidade de Kaiserslautern, Gregor Engels, da Universidade de Paderborn, Hartmut Ehrig, da TU-Berlin e Klaus Lagally, da Universidade de Stuttgart, pela oportunidade de visitar seus grupos de pesquisa e discutir o desenvolvimento desta dissertação.

Agradeço aos colegas do PPGC pela amizade e companhia durante esses anos. Em especial aos amigos Aline Loreto, André Martinotto, Diana Adamatti, Luciana Foss, Luciana Schroeder e Luis André Martins pelo companheirismo e incentivo durante o desenvolvimento deste trabalho.

Agradeço aos professores e funcionários do PPGC, que de uma forma ou de outra ajudaram na realização deste trabalho.

Ao CNPq por financiar este trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADS	Ambiente de Desenvolvimento de Software
ATO	Ambiente de Tratamento de Objetos
CASE	Computer-Aided Software Engineering
HDA	History Dependent Automata
ICS	Interface de Comunicação do Sistema
NSD	Nassi-Schneidermann Diagram
OO	Orientação à Objetos
PPGC	Programa de Pós-Graduação em Computação
ProTool	Prototyping Tool
PS	Processo de Software
RAD	Rapid Application Development
SO	Sistema Operacional
SRT	Sistema de Reescrita de Termos
SW-CMM	Software Capability Maturity Model
TAD	Tipo Abstrato de Dado
TD	Tipo de Dado
UFRGS	Universidade Federal do Rio Grande do Sul
VDM	Viena Development Method

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Dentre as principais áreas que constituem a Ciência da Computação, uma das que mais influenciam o mundo atual é a Engenharia de Software, envolvida nos aspectos científicos e tecnológicos do desenvolvimento de software.

No desenvolvimento de software, a fase de especificação dos requisitos é uma das mais importantes, visto que erros não detectados nesta são propagados para as fases posteriores. Quanto mais avançado estiver o desenvolvimento, mais caro custa reparar um erro introduzido nas fases iniciais, pois isto envolve reconsiderar vários estágios do desenvolvimento.

A prototipação permite que os requisitos do software sejam validados logo no início do desenvolvimento, evitando assim a propagação de erros. Paralelamente, a utilização de métodos formais visa revelar inconsistências, ambigüidades e falhas na especificação do software, que podem caso contrário, não serem detectadas. Usar a prototipação de software juntamente com uma notação formal enfatiza a especificação do problema e expõe o usuário a um sistema "operante" o mais rápido possível, de modo que usuários e desenvolvedores possam executar e validar as especificações dos requisitos funcionais.

O objetivo principal deste trabalho é instanciar uma técnica da área de Prototipação de Software que capacite o engenheiro de software gerar automaticamente protótipos executáveis a partir de especificações formais de tipos abstratos de dados, na notação PROSOFT-algébrico, visando a validação dos requisitos funcionais logo no início do desenvolvimento do software. Para tanto foi proposto um mapeamento da linguagem PROSOFT-algébrico para OBJ. Como OBJ possui um eficiente sistema de reescrita de termos implementado, a utilização deste propicia a prototipação de tipos abstratos de dados, especificados em PROSOFT-algébrico.

Os componentes envolvidos na definição deste trabalho, assim como o mapeamento entre as linguagens, foram especificados algebricamente e implementados no ambiente de desenvolvimento de software PROSOFT. A implementação serviu para validar o mapeamento proposto através de dois estudos de caso.

Por fim, são apresentadas as conclusões alcançadas e as atividades adicionais vislumbradas a partir do trabalho proposto.

Palavras-chave: Prototipação de Software, Especificação Algébrica, PROSOFT, OBJ.

ProTool: A Software Prototyping Tool for the PROSOFT Environment

ABSTRACT

Software Engineering constitutes an important field in the modern Computer Science by combining different management and technological strategies to support software development.

One of the most important phases of software development is the requirements specification. If errors are not detected in this stage they go to further phases. It is well known that the cost of error correction rises rapidly if errors are not detected until late in the development cycle.

Prototyping allows software requirements to be validated at the beginning of development, avoiding error propagations. In conjunction with, the use of formal methods promotes revealing inconsistencies, ambiguities and faults in software specification. Thus, without formal methods such errors might not be detected.

Combining prototyping with a formal notation allows focusing on the problem been specified and users can test an executable system as soon as possible in the development cycle. Hence, users and developers are able to execute and validate the functional requirements specification.

The main goal of this work is to instantiate a technique from Software Prototyping allowing the software engineer to automatically create executable prototypes from abstract data types specifications, in algebraic PROSOFT notation. This aims to detect and eliminate errors in functional requirements as early as possible in the software development. The idea is to create a mapping between algebraic PROSOFT and OBJ, because OBJ has an efficient term rewriting system implemented and through this translation abstract data types in algebraic PROSOFT can be prototyped in OBJ environment.

The required software components for the mapping were specified through algebraic specification techniques which, in turn, were used to derive Java based prototypes inside the Java-PROSOFT environment. The implementation was very useful to evaluate the concepts of the mapping using it in case studies.

Finally, achieved conclusions and expected future activities are presented.

Keywords: Software Prototyping,Algebraic Specification, Prosoft,OBJ.

1 INTRODUÇÃO

Dentre as principais áreas que constituem a Ciência da Computação, uma das que mais influenciam o mundo atual é a Engenharia de Software, envolvida nos aspectos científicos e tecnológicos do desenvolvimento de software. O software tornou-se base de sustentação para inúmeras organizações dos mais diversos ramos de atuação e, como conseqüência, muitos recursos são aplicados na busca de soluções que auxiliem a produção de software.

Apesar dos inúmeros avanços nesta área, muito ainda é discutido acerca da baixa qualidade e produtividade da indústria mundial de software, refletindo-se na insatisfação dos seus usuários e em prejuízos financeiros de enormes proporções.

No desenvolvimento de software, a fase de especificação dos requisitos é uma das mais importantes. Tipicamente nesta fase inicial do projeto de software muitos erros são introduzidos e geralmente não são detectados até que o sistema tenha sido totalmente implementado e testado. Grande parte destes erros são resultado de uma análise de requisitos de baixa fidelidade (LUTZ, 1993; SALLIS; TATE; MCDONELL, 1995; CURTIS; KRASNER; ISCOE, 1988) que não detecta ambiguidades e inconsistências nos requisitos do sistema em desenvolvimento. Conseqüentemente, estes erros são caros e difíceis de serem reparados, pois isto envolve reconsiderar todos os estágios do desenvolvimento (BOEHM, 1981; DAVIS, 1993).

Devido a grande complexidade das aplicações de software atuais, muitas vezes é necessário utilizar técnicas que facilitem a manipulação de requisitos cada vez maiores, em escala e funcionalidade, no desenvolvimento do software. Dentre estas técnicas auxiliares, a prototipação de software, utilizada logo no início do ciclo de vida do desenvolvimento de software, tem como objetivo reduzir o tempo total de desenvolvimento e aumentar a qualidade do software que está sendo construído (BOEHM; GRAY; SEEWALDT, 1984; PALVIA; NOSEK, 1990; GORDON; BIEMAN, 1994).

Este trabalho apresenta a utilização de uma técnica de prototipação de software que emprega um método algébrico tanto para especificar os tipos de dados do software em construção quanto gerar automaticamente protótipos executáveis para validação das mesmas.

1.1 Motivação e Objetivos do Trabalho

A prototipação de software visa reduzir os erros introduzidos na análise de requisitos através da validação precoce da especificação do software em desenvolvimento. Um protótipo executável permite que os estados do sistema sejam testados, por execução, bem antes do início da fase de implementação.

Segundo Luqi (LUQI, 1992; LUQI; GOGUEN, 1997), para cada dólar investido em prototipação, pode-se esperar um retorno de US$ 1,40 no decorrer do ciclo de vida do

desenvolvimento, pois os requisitos tendem a mudar cerca de 30 a 40% sem o auxílio da prototipação. Na fase de implementação, a atividade de prototipação é responsável por uma redução no esforço de programação em torno de 40%, de acordo com os experimentos de Boehm (BOEHM; GRAY; SEEWALDT, 1984).

Almejando reduzir ao máximo os erros introduzidos na análise dos requisitos funcionais de software, salienta-se, além da prototipação, a utilização de métodos formais, que são compostos por linguagens e ferramentas para especificar e verificar sistemas, baseados em fundamentos lógicos matemáticos (CLARKE et al., 1996). A utilização de métodos formais visa principalmente aumentar o entendimento do software, revelando inconsistências, ambigüidades e falhas que podem, caso contrário, não serem detectadas. Além disso, as especificações formais facilitam a modularização e o reuso na produção de software.

A utilização de um método de prototipação de software juntamente com uma notação formal enfatiza a especificação formal do problema e expõe o usuário a um sistema "operante" o mais rápido possível, de modo que usuários e desenvolvedores sejam capazes de executar e validar as especificações dos requisitos funcionais do sistema. Assim, os benefícios da especificação formal e da abordagem de prototipação para o desenvolvimento de software são combinados e uma descrição de sistema bem estruturada, executável e sem ambiguidade pode ser desenvolvida.

O mecanismo para prototipação proposto neste trabalho adota o ambiente de desenvolvimento de software PROSOFT (NUNES, 1992, 1994) para especificação e prototipação de tipos de dados. Este ambiente, estabelecido no PPGC-UFRGS, foi construído com o objetivo de suportar o desenvolvimento de software através da utilização do método de especificação formal PROSOFT-algébrico. Por ser um método algébrico, o PROSOFT-algébrico especifica tipos abstratos de dados e operações sobre estes tipos. O ambiente PROSOFT proporciona que especificações destes tipos abstratos de dados e suas respectivas operações sejam implementadas e integradas ao ambiente, passando a fazer parte do mesmo.

Muitos usuários do ambiente PROSOFT, entre eles estudantes de mestrado e doutorado, fazem uso deste ambiente quase que exclusivamente para implementação de sistemas de software, pois a versão disponível do ambiente não fornece recursos para validar as especificações dos tipos abstratos de dados, tornando-as mais suscetíveis à introdução de erros. A necessidade de suporte às atividades de validação das especificações algébricas, construídas no paradigma PROSOFT, motivou a proposta do presente trabalho.

Portanto, o objetivo principal deste trabalho é instanciar uma técnica da área de Prototipação de Software que capacite o engenheiro de software gerar automaticamente protótipos executáveis a partir de especificações de tipos abstratos de dados, na notação PROSOFT-algébrico, visando a validação dos requisitos funcionais logo no início do desenvolvimento do software. Este trabalho resultou na especificação formal do modelo de prototipação para o ambiente PROSOFT, chamado ProTool (**Pro**totyping **Tool**).

1.2 Organização do Texto

O texto está organizado conforme descrito abaixo:

Capítulo 2 descreve as abordagens para prototipação de software, assim como suas características. São também apresentadas as técnicas utilizadas na construção de protótipos e as principais linguagens usadas para prototipação.

Capítulo 3 apresenta a linguagem de especificação PROSOFT-algébrico e as principais características que norteiam o desenvolvimento de software no ambiente PROSOFT.

Capítulo 4 descreve a linguagem de especificação algébrica OBJ, que é utilizada para prototipar as especificações da notação PROSOFT-algébrico.

Capítulo 5 contém a descrição informal da motivação e dos requisitos necessários para tradução de especificações PROSOFT-algébrico para OBJ. São também apresentados: um comparativo entre estas duas notações; as regras que efetivam a tradução; e exemplos da tradução.

Capítulo 6 contém a especificação formal de todos os tipos de dados e operações que foram descritos no capítulo anterior.

Capítulo 7 mostra a implementação de um editor de especificações para o PROSOFT-algébrico. Também é mostrada a implementação do modelo ProTool no ambiente PROSOFT, assim como as instruções necessárias para uso do mesmo.

Capítulo 8 apresenta dois estudos de casos para validação do ProTool. O primeiro baseia-se em um sistema para locação de DVDs e CDs, enquanto o segundo visa a prototipação do ProTool usando o próprio ProTool para gerar os protótipos em OBJ.

Capítulo 9 apresenta as conclusões e as atividades futuras que podem ser realizadas a partir deste trabalho.

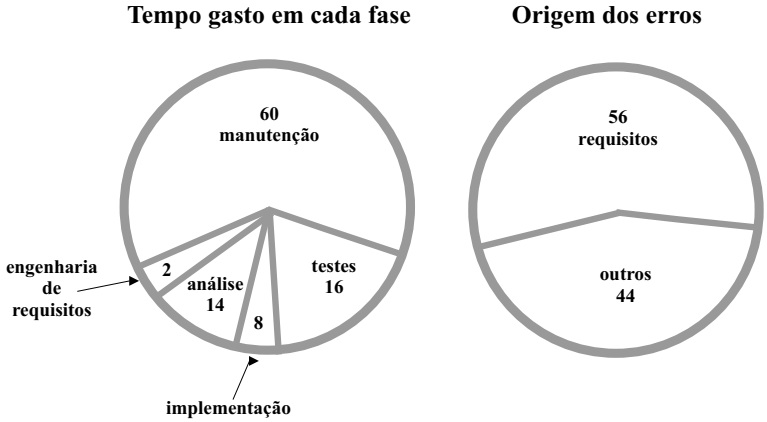
Anexo 1 apresenta todas as operações definidas na notação PROSOFT-algébrico, assim como suas funcionalidades e exemplos informais de uso.

Anexo 2 descreve a semântica dos tipos de dados do PROSOFT-algébrico em função das especificações em OBJ.

Anexo 3 descreve resumidamente a semântica das operações utilizadas na prototipação de classes recursivas do PROSOFT-algébrico.

2 PROTOTIPAÇÃO DE SOFTWARE

Apesar dos substanciais avanços dos métodos e ferramentas da Engenharia de Software nos últimos anos, a análise de requisitos ainda continua sendo um problema chave no desenvolvimento de softwares complexos. Um dos principais fatores que contribuíram para perpetuação deste problema é a carência de validações logo no início do ciclo do desenvolvimento de software. A validação dos requisitos é problemática pois freqüentemente estes requisitos não são perfeitamente compreendidos antes do início da fase de implementação do software.



Custo relativo para correção de erro

Fase	Custo relativo para reparação
Requisitos	1
Análise	5
Implementação	10
Teste de módulos	20
Teste de aceitação	50
Manutenção	200

Figura 2.1: A importância da validação antecipada dos requisitos. (traduzida de (WOOD; KANG, 1992))

Os dados estatísticos apresentados na figura 2.1 mostram a importância da validação dos requisitos logo no início do ciclo de vida do desenvolvimento. Boehm identificou que 54% de todos erros detectados em projetos de software, estudados na empresa TRW, foram encontrados após as fases de codificação e testes. A maioria destes erros (83%) foi atribuída à fase de análise ao invés da fase de implementação (17%) (BOEHM; MCCLEAN; URFRIG, 1975). DeMarco também relatou que 56% de todos erros detectados foram introduzidos na análise dos requisitos (TAVOLATO; VINCENA, 1984).

Muitos erros nos requisitos passam, sem serem detectados, para estágios mais avançados do desenvolvimento, e corrigir estes erros durante ou após a implementação é extre-

mamente mais caro (PAUL; SIMON, 1989).

Os tipos de erros mais freqüentes na análise de requisitos são de origem técnica, conforme ilustrado na figura 2.2. Basili (BASILI; WEISS, 1981) identificou que 77% dos erros encontrados nos requisitos do programa operacional de vôo da aeronave Navy A-7E foram erros de características não técnicas, dos quais 49% foram fatos incorretos e 31% foram omissões. Inconsistências e ambiguidade contabilizam 18% dos erros não técnicos.

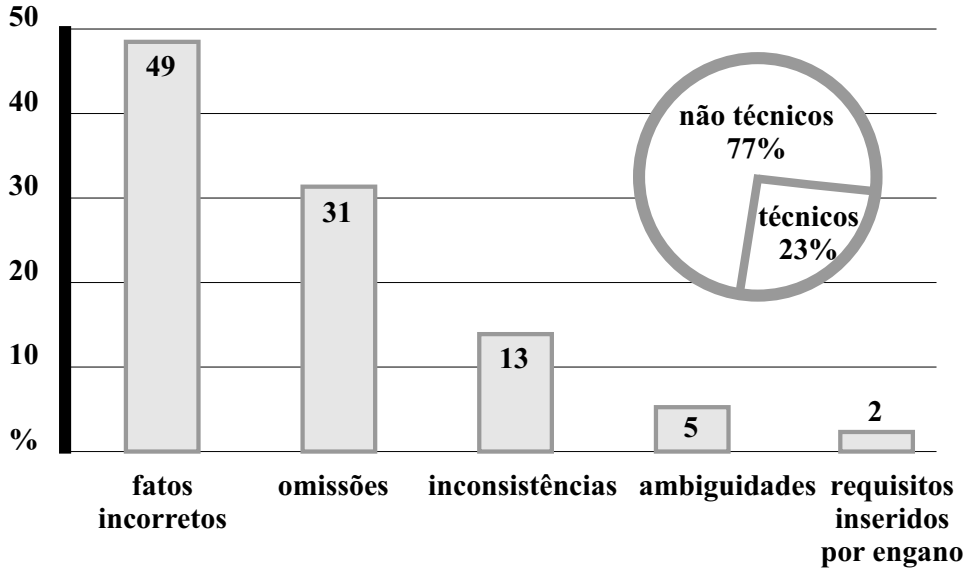


Figura 2.2: Tipos de erros nos requisitos (traduzida de (WOOD; KANG, 1992))

O modelo clássico para desenvolvimento de software, conhecido como cascata, supõe um projeto no qual os passos de trabalho são detalhados antes de serem executados e conseqüentemente contribui para a introdução de erros (ROYCE, 1970).

Na tentativa de superar as desvantagens do modelo em cascata, muitas abordagens de desenvolvimento de software foram propostas, entre elas: aprimoramento iterativo (BASILI; TURNER, 1975), prototipação rápida (GOMAA, 1983), prototipação evolucionária e incremental (FLOYD, 1984).

As abordagens de desenvolvimento de software que incorporam prototipação ganham respaldo pois têm provado serem capazes de dinamicamente responder às mudanças nos requisitos dos usuários (FLOYD, 1984), reduzindo a quantia de retrabalho necessária, além de viabilizar a identificação de problemas como ambiguidade, incompleteza e inconsistência na captura dos requisitos. A prototipação diminui os custos de desenvolvimento do software (BOEHM; GRAY; SEEWALDT, 1984; PALVIA; NOSEK, 1990; GORDON; BIEMAN, 1994), aumenta a comunicação entre as pessoas engajadas no desenvolvimento (ALAVI, 1984), ajuda a determinar a viabilidade técnica (FLOYD, 1984), é uma boa técnica de gerenciamento de riscos (TATE; VERNER, 1990) e resulta em um maior envolvimento e participação dos usuários no processo de desenvolvimento (NAUMAN; JENKINS, 1982).

No desenvolvimento de sistemas, a experiência tem mostrado que o produto final freqüentemente não atende as necessidades dos usuários. A razão mais comum para este problema é devido ao surgimento de novas idéias durante os estágios mais avançados de um projeto. E estas idéias não eram conhecidas no início, mas somente após a evolução do projeto.

A prototipação pode corrigir ambigüidades e mal entendimentos na fase de especifi-

cação de modo significantemente mais barato que corrigir um sistema após ele ter sido desenvolvido.

A revisão de prototipação de software mais conhecida na literatura é a de Floyd (FLOYD, 1984), que serve como base para muitos trabalhos nesta área. Segundo (FLOYD, 1984), a prototipação pode ser vista como um "processo" bem definido do ciclo de vida do desenvolvimento de software, ou uma "abordagem" que influencia o ciclo inteiro (BUDDE et al., 1992). O processo de prototipação pode encorajar o desenvolvimento eficiente de aplicações através da quebra de um sistema complexo em diversas partes menores, mais simples e portanto mais compreensíveis (KAUSHAAR; SHIRLAND, 1985).

No desenvolvimento de software podem ser utilizadas duas classes distintas de protótipos, ou seja, protótipos baseados em papel ou em software. Os protótipos baseados em papel não provêm funcionalidade, mas podem ser úteis para gerar idéias e elucidar os requisitos do usuário. A prototipação baseada em software é dividida segundo as abordagens de desenvolvimento abaixo:

Experimental – fornece aos usuários várias alternativas que ajudam a estimar a viabilidade do futuro sistema em termos de performance e outros aspectos técnicos, quando os requisitos são conhecidos;

Exploratória – usada principalmente para elicitar ou esclarecer os requisitos dos usuários. Ajuda os desenvolvedores a obterem novas idéias dos problemas e das tarefas dos usuários. Ajuda também a cristalizar as confusas necessidades dos usuários para um primeiro sistema;

Evolucionária – permite flexibilidade no processo de desenvolvimento de software tal que ele pode ser adaptado às mudanças organizacionais.

Kieback (KIEBACK et al., 1992) classifica os tipos de protótipos de acordo com as diferentes tarefas que eles executam:

Protótipo de Demonstração – usado pelos desenvolvedores para mostrar aos clientes a viabilidade do novo projeto e a primeira impressão do futuro sistema. Aborda somente uma parte limitada do sistema final, sendo esta geralmente a interface gráfica;

Protótipo Funcional – usado em paralelo a modelagem do software, onde um sistema temporário executável é construído. Embora cobrindo diversas funções e interface com usuário, o protótipo carece de propriedades importantes do sistema final, como tratamento de exceções, por exemplo. Usado para elucidar a definição dos requisitos e responder questões de projeto;

Protótipo Superfície – derivado das especificações para permitir o estudo de soluções alternativas, estimulando a criatividade dos desenvolvedores;

Protótipo Piloto – não existe distinção entre o protótipo e o sistema final. O protótipo é usado como um núcleo que evolui gradualmente até o sistema final.

Naumann e Jenkins (NAUMAN; JENKINS, 1982) caracterizaram a tarefa de prototipação contendo quatro passos mais um processo interativo que envolve tanto os desenvolvedores como os usuários:

1. as necessidades básicas dos usuários são identificadas;

2. um protótipo executável é construído;

3. o protótipo é então implementado e usado;

4. o sistema protótipo é revisado e aprimorado.

O processo de prototipação consiste essencialmente de diversos ciclos iterativos. O protótipo inicial, um modelo de software executável, é construído baseado na seleção inicial de funções ou necessidades identificadas pelos usuários. Uma demonstração do protótipo permite a revisão dos usuários. Sugestões dos usuários, críticas e aprimoramentos resultam na revisão do protótipo. Este ciclo é mantido para cada revisão. O processo de prototipação é então finalizado quando o objetivo desejado é alcançado.

O emprego da prototipação no ciclo de vida de desenvolvimento de software traz os seguintes benefícios:

- disponibilizar um meio tanto para experimentação de idéias quanto para acomodar decisões propostas pelos usuários;
- entregar rapidamente um sistema executável, para o usuário, que serve para avaliar o impacto do sistema proposto no ambiente da organização;
- estabelecer e validar os requisitos dos sistema a ser construído;
- minimizar os riscos do projeto visto que o protótipo pode estabelecer como o modelo funciona previamente à implementação final do sistema;
- reduzir custos e tempo do projeto de software;
- determinar a viabilidade do produto, bem como avaliar os custos de desenvolvimento;
- iniciar de um processo de aprendizado para usuários e desenvolvedores do sistema.

2.1 Abordagens para Desenvolvimento de Protótipos

Como apresentado anteriormente, Floyd (FLOYD, 1984) categorizou as abordagens de desenvolvimentos de software, que empregam prototipação, como sendo baseadas num destes três vetores: exploração, experimentação e evolução.

2.1.1 Abordagem Exploratória

As abordagens de desenvolvimento de software visam explorar os requisitos ou certos detalhes antes ou durante o desenvolvimento. Existem duas importantes abordagens que empregam este método: prototipação rápida descartável e modelo espiral (BOEHM, 1988). O primeiro método explora a completeza das especificações enquanto o segundo é uma tentativa para gerenciar os riscos no processo de desenvolvimento de software.

A prototipação rápida, também conhecida como descartável, ajuda os desenvolvedores a completarem o conjunto de requisitos dos usuários. Durante a fase de elicitação dos requisitos uma implementação parcial do sistema é construída (GOMAA, 1983). Os requisitos menos entendidos são os geralmente implementados primeiro. Quando o protótipo é construído, fatores de qualidade como eficiência, manutenibilidade, portabilidade, documentação e completeza não são considerados (KEUS, 1982). Usuários potenciais fornecem *feedback* aos desenvolvedores revisando o protótipo, que por sua vez

serve para refinar a especificação dos requisitos. Os desenvolvedores então seguem o projeto e implementação do sistema desde que haja um acordo entre usuários e desenvolvedores quanto aos requisitos. O protótipo é geralmente descartado embora algumas vezes o código possa ser reutilizado (DAVIS; BERSOFF; COMER, 1988).

2.1.2 Abordagem Experimental

Nesta abordagem, protótipos são construídos tal que a viabilidade das soluções propostas possam ser examinadas pelo uso experimental. O protótipo pode ser uma simulação funcional parcial demonstrando certos aspectos do sistema, uma simulação funcional total para demonstrar todas funções do sistema proposto, uma maquete de interface ou um "esqueleto" de programa mostrando a estrutura do sistema (MAYHEW; DEARNLEY, 1987; TATE, 1990).

2.1.3 Abordagem Evolucionária

Nesta abordagem, Floyd (FLOYD, 1984) identifica duas subcategorias distintas: incremental e evolucionária. Na primeira, o sistema evolui gradualmente através de incrementos parciais. A diferença fundamental entre desenvolvimento evolucionário e incremental é o projeto do sistema (INCE; HEKMATPOUR, 1987). No incremental assume-se que a maior parte dos requisitos do usuário são conhecidos no logo no início, enquanto que no evolucionário o protótipo é construído na área que todos os requisitos ainda não foram bem compreendidos (DAVIS; BERSOFF; COMER, 1988).

O desenvolvimento incremental é um método no qual um sistema parcial é implementado a partir de um projeto completo. Aspectos como funcionalidade e performance são gradativamente adicionados ao sistema parcial. Várias versões parciais são construídas, onde cada uma adiciona uma parte do projeto original (INCE; HEKMATPOUR, 1987). Assim, o sistema é construído de tal forma que facilite a incorporação de requisitos adicionais.

O desenvolvimento evolucionário é adotado no contexto de um ambiente dinâmico e mutável. Este tipo de desenvolvimento necessita uma seqüência de ciclos de reprojeto, reimplementação e reavaliação, sem qualquer esforço para capturar previamente um conjunto completo de requisitos (FLOYD, 1984). Primeiramente, um protótipo dos requisitos parcialmente conhecidos é implementado. Quando os usuários usam o protótipo, um entendimento mais aprofundado dos requisitos é obtido e novos requisitos são então implementados. O sistema final evolui de maneira tradicional (DAVIS; BERSOFF; COMER, 1988), onde cada protótipo sucessor explora novas necessidades dos usuários, e refina a funcionalidade que havia sido implementada. Enquanto que a seqüência de protótipos é talvez convergente em direção aos objetivos móveis, eles nunca serão alcançados, pois os requisitos estão sujeitos a mudanças contínuas (TATE, 1990). Entretanto, o sistema está sempre em evolução contínua e não há nenhuma caracterização de uma fase manutenção (FLOYD, 1984).

2.2 Técnicas de Prototipação

As técnicas de prototipação de software devem permitir o rápido desenvolvimento de um protótipo. De acordo com (SOMMERVILLE, 2001), entre as técnicas que têm sido usadas para prototipação de software, destacam-se:

- linguagens de especificação formal executáveis;

- linguagens de programação de alto nível;
- geradores de aplicação e linguagens de quarta geração;
- composição de componentes reutilizáveis.

Cabe ressaltar que elas não são mutuamente exclusivas. Por exemplo, uma parte do software pode ser construída por um gerador de aplicação e ligada a componentes de software já existentes.

2.2.1 Linguagens de Especificação Executáveis

Desenvolver um protótipo a partir de uma especificação formal é um atrativo que combina uma especificação não ambígua com um protótipo executável. Esta técnica tem a vantagem de não gerar custos adicionais para desenvolver o protótipo após a especificação ter sido escrita. Entretanto, existem algumas dificuldades práticas na aplicação desta abordagem:

- as interfaces gráficas com usuário não podem ser prototipadas usando esta técnica;
- a especificação formal requer uma análise detalhada do sistema. Portanto, muito tempo pode ser gasto na modelagem detalhada de funções do sistema que são rejeitadas após a avaliação do protótipo;
- o sistema executável é geralmente lento e ineficiente. Conseqüentemente, os usuários podem ter uma falsa impressão do sistema assim como podem não usá-lo da mesma forma que usariam uma versão mais eficiente;
- as especificações executáveis testam somente os requisitos funcionais do sistema. Em alguns casos, as características não funcionais do sistema são particularmente importantes que tornam o protótipo algo de valor limitado.

Alguns destes problemas têm sido considerados pelos desenvolvedores de linguagens de especificação (por exemplo, Prosoft-algébrico (NUNES, 1992)), que têm integrado interfaces gráficas com usuário, permitindo o desenvolvimento rápido de protótipos através de interfaces mais ergonômicas.

Dentre as linguagens formais executáveis mais utilizadas na prototipação de tipos abstratos de dados, destacam-se: OBJ (GOGUEN et al., 2000), Haskell (JONES; PETERSON, 1999; HUDAK; PETERSON; FASEL, 1999), Act-One (EHRIG; MAHR, 1985; HANSEN, 1987), CASL (ASTESIANO et al., 2002) e Prolog (ARMSTRONG; VIRDING; WILLIAMS, 1992).

2.2.2 Linguagens de Programação de Alto Nível

As linguagens de programação de alto nível incluem poderosas facilidades para processamento de dados. Elas simplificam a implementação do software porque fornecem recursos que deveriam ser construídos em linguagens mais primitivas.

É importante ressaltar que diferentes linguagens são usadas para diferentes tipos de sistemas, dependendo do domínio de aplicação. A tabela 2.1 lista algumas linguagens de programação utilizadas para prototipação.

Tabela 2.1: Linguagens de programação

Linguagem	Tipo	Domínio de aplicação
Smaltalk	orientado a objeto	sistemas interativos
Prolog	lógica	processamento simbólico
Lisp	baseada em listas	processamento simbólico
Miranda	funcional	processamento simbólico
SETL	baseada em conjuntos	processamento simbólico
APL	matemática	sistemas científicos

Dependendo do tipo de protótipo a ser construído, pode-se adotar uma abordagem que combine várias linguagens de programação. Diferentes partes do sistema podem ser programadas em diferentes linguagens e um *framework* de comunicação estabelecido entre as partes.

Cabe citar que não existe uma linguagem ideal para prototipação de sistemas grandes, pois suas partes tendem a ser muito diversas. A vantagem da abordagem que combina linguagens é que uma linguagem mais apropriada pode ser escolhida para cada parte lógica do sistema. A desvantagem é que pode ser difícil estabelecer um *framework* de comunicação no qual múltiplas linguagens possam se comunicar, pois as entidades usadas nas linguagens podem ser muito diferentes. Conseqüentemente, uma codificação muito grande pode ser necessária para traduzir entidades de uma linguagem em outras de uma segunda linguagem.

2.2.3 Linguagens de Quarta Geração

As técnicas de quarta geração abrangem um amplo conjunto de linguagens de emissão de relatórios, consulta a banco de dados, geradores de aplicações e programas, além de outras linguagens de altíssimo nível. Uma vez que capacitam o engenheiro de software a gerar códigos executáveis rapidamente, as linguagens de quarta geração tornam-se boas candidatas para a prototipação rápida.

2.2.4 Componentes de Software Reutilizáveis

Outra técnica de prototipação rápida é montar, em vez de construir o protótipo, usando componentes de software já existentes. Um componente de software pode ser uma estrutura de dados (ou banco de dados), um programa ou um componente procedimental (módulo). Em cada caso, o componente de software deve ser projetado de tal forma que possa ser reutilizado sem um conhecimento detalhado do seu funcionamento interno.

2.3 Especificação Formal no Desenvolvimento de Software

Uma questão tradicional na Engenharia de Software é se o sistema proposto é realmente uma solução para o problema considerado. Uma maneira de responder esta pergunta é através do uso de métodos formais, cuja idéia é focar-se na modelagem formal do sistema em construção, abstraindo seus comportamentos menos importantes. Esse modelo formal pode ser empregado para analisar o comportamento do sistema, a fim de garantir que o modelo construído possui o comportamento correto e as propriedades desejadas. Entre os métodos formais mais conhecidos, citam-se: VDM (JONES, 1986), Z (SPIVEY, 1992), OBJ (GOGUEN et al., 2000), Larch (GUTTAG; HORNING, 1993),

CASL (ASTESIANO et al., 2002), álgebra de processos (HENNESSY, 1988), Gramática de Grafos (ROZENBERG, 1997; EHRIG et al., 1999), Redes de Petri (PETERSON, 1981; REISIG, 1985) e CCS (MILNER, 1989).

Os métodos formais podem ser totalmente integrados no ciclo de vida clássico do desenvolvimento de software, como mostrado em (COHEN, 1989). Na fase de análise, eles podem ser usados juntamente com outras técnicas semi-formais. Na fase de projeto, a especificação pode sofrer uma série de refinamentos e iterações até chegar na implementação. O uso de raciocínio formal e refinamentos ajuda em assegurar a correteza de cada versão do software em desenvolvimento.

Em alguma etapa do desenvolvimento a descrição informal dos requisitos precisa ser formalizada. É essencial que a formalização capture os requisitos contidos na especificação informal de modo mais correto e completo possível. Infelizmente, este processo é inerentemente tendencioso a erros e não pode ser completamente automatizado e provado (COHEN, 1989). Outra dificuldade é o fato dos requisitos do usuário serem freqüentemente mal definidos e usualmente evoluírem através do ciclo de desenvolvimento. Portanto, não existe uma especificação "completa".

Na maioria dos projetos em que especificações formais são usadas, especificações informais e formais são combinadas. Alguns componentes e passos são formalizados enquanto outros não. A decisão de usar especificações formais depende principalmente do grau de criticidade do componente de software, em termos de conseqüências de uma falha (vidas humanas, custo, etc), da complexidade de seus requisitos e seu desenvolvimento.

As linguagens de especificação concentram-se na funcionalidade e no projeto das estruturas, deixando algoritmos e detalhes de implementação para uma posterior fase de codificação. Linguagens de especificação declarativas, assim como o OBJ, permitem que o desenvolvimento de uma especificação siga uma abordagem incremental, onde o primeiro estágio de especificação pode consistir simplesmente em nomear as entidades que irão compor o sistema, deixando para estágios posteriores o refinamento das funcionalidade de tais entidades.

Os métodos de especificação algébrica fornecem um conjunto de técnicas para abstração de dados e para especificação, validação e análise das estruturas de dados construídas (LEEUWEN, 1994). Este métodos podem ser considerados, neste contexto, uma linha de especificação formal mais evoluída, tanto pela teoria subjacente quanto pelas suas eficientes implementações existentes.

2.3.1 Validação de Especificações Formais

Mesmo parecendo óbvio, é sempre útil lembrar que uma especificação formal pode estar errada. Ela pode estar errada por dois motivos: houve algum mal entendimento dos requisitos dos usuários ou algum erro na expressão destes requisitos. Uma grande vantagem do uso de especificações formais é que as ferramentas existentes podem analisá-las e algumas vezes animá-las. E, portanto, estas ferramentas podem guiar a atividade de validação[1] das especificações dos requisitos funcionais do software.

No desenvolvimento tradicional de software um dos principais problemas é o enorme tempo entre a especificação de um sistema e sua validação. Visto que a implementação é a primeira versão executável do sistema, a validação é somente possível após muitas decisões já terem sido tomadas. Conseqüentemente, a validação tem que lidar com uma quantidade enorme de detalhes de implementação e, possivelmente, muitas decisões de

[1] Validação significa verificar a correspondência entre os requisitos informais e as especificações formais. Para o caso geral, nenhum método formal pode verificar esta correspondência (HOARE, 1987).

projeto deverão serem refeitas. As especificações executáveis, entretanto, permitem validar antecipadamente os requisitos funcionais sob a ótica do problema, ou seja, em um nível de abstração muito alto que não contém detalhes de implementação.

As especificações executáveis também servem como protótipos que permitem experimentar diferentes requisitos, ou mesmo usar uma abordagem evolucionária para o desenvolvimento de software. Isto é especialmente importante, pois em muitos projetos os requisitos não podem ser inicialmente descritos de forma completa e precisa.

Entretanto, o uso de métodos formais não garante a priori que o software em desenvolvimento será correto. Mas, por outro lado, ele pode aumentar muito o entendimento sobre o sistema, revelando inconsistências, ambigüidades e incompletudes, que poderiam, de outro modo, passar desapercebidas.

2.3.2 Métodos de Validação

Existem algumas técnicas que, empregadas junto às especificações formais, podem ajudar a validar as especificações de acordo com os requisitos informais. A seguir elas são brevemente descritas.

A *prova de propriedades* de uma especificação, através de provadores de teoremas, parece ser a primeira opção quando métodos formais estão sob consideração. Em muitos casos, a árvore de prova tende a ser infinita, assim impossibilitando provas automáticas, para o caso geral.

A *interpretação* é também uma outra alternativa quando se trata de especificações formais. Técnicas baseadas na lógica, tais como reescrita de termos ou algoritmos de resolução, fornecem uma base poderosa para executar especificações escritas em alguma lógica restrita. Quase todas as linguagens de especificação contêm subpartes nas quais é possível escrever especificações executáveis. Este é o caso das especificações algébricas, quando os axiomas são restritos a cláusulas equacionais.

A *geração automática de código* a partir de especificações formais tem sido estudada por muito tempo. Quase todos ambientes de especificação formal tem um gerador de código. A geração de código facilita consideravelmente a disciplina de programação, assegurando que o código gerado representa exatamente o sistema especificado e que ambos têm as mesmas propriedades.

Na construção de software, é possível garantir que uma implementação satisfaz sua especificação através da prova de propriedades. Mas mesmo assim, não se pode garantir que esta implementação realmente satisfaz os requisitos do usuário. Considere um exemplo no qual uma especificação formal foi criada, mas esta não corresponde fielmente aos reais requisitos do usuário. A implementação pode até ser provada correta de acordo com a especificação formal, mas a implementação não é uma solução para o problema proposto pelo usuário. Por isso, validar por execução é em muitos casos imprescindível.

É importante ressaltar que tanto a interpretação como a geração de código geram tipos complementares de protótipos executáveis que auxiliam a tarefa de validação. A interpretação de uma especificação geralmente é feita através de sistemas de reescrita de termos, como em OBJ (GOGUEN et al., 2000). Uma limitação comum desta técnica é que os protótipos só podem ser executados no ambiente de especificação. Se o software que está sendo construído precisa interagir com módulos já existentes, então deve-se utilizar a técnica de geração de código. Note-se que a geração de código é válida para situações nas quais a especificação formal já está em fase avançada. Resumidamente, a interpretação é útil para validar requisitos funcionais enquanto a geração de código é útil na validação de requisitos não funcionais como: interface gráfica, manutenibilidade, performance, segu-

rança, etc.

2.4 Linguagens Utilizadas na Prototipação de Software

Para a concepção da ferramenta de prototipação de software proposta nesta dissertação foi necessário realizar um estudo acerca das principais linguagens utilizadas para prototipação. As linguagens estudadas são descritas, nesta seção, de maneira resumida, abordando somente as principais características de cada linguagem, pois o estudo completo de cada uma está fora do escopo do presente trabalho. Em (RANGEL, 2001) foram estudadas três linguagens declarativas (tabela 2.2) amplamente utilizadas na construção de protótipos.

Tabela 2.2: Linguagens para prototipação de software estudadas

Linguagem	**Paradigma**
OBJ	Especificação algébrica
Haskell	Funcional
Prolog	Lógica

Deve-se evidenciar que na literatura especializada existem muitas outras linguagens que podem servir como meio de prototipação, cada qual voltada para um domínio de aplicação. A decisão por linguagens declarativas ocorreu devido elas focarem o problema a ser resolvido em vez de focarem os passos necessários para solucionar o problema, como nas linguagens imperativas. Uma linguagem declarativa fornece poderosos recursos, com alto nível de abstração, para especificação de software. Enquanto que o uso de linguagens imperativas sobrecarrega a equipe de desenvolvimento com tarefas além do problema a ser solucionado (como gerenciamento de memória, ponteiros, etc.), o que torna a fase de análise bem mais difícil e propensa a introdução de erros.

Os aspectos mais relevantes de cada linguagem são apresentados a seguir.

2.4.1 OBJ

O método de especificação formal OBJ foi projetado por Joseph Goghen como uma extensão da teoria algébrica de tipos abstratos de dados para tratamento de erros e funções parciais, cujo objetivo é escrever e testar especificações algébricas. O OBJ, além das vantagens intrínsecas dos métodos de especificação formal, apresenta dois importantes aspectos: é parametrizado, permitindo portanto a reutilização de software e é executável, o que proporciona a interpretação direta da especificação.

Uma especificação em OBJ consiste em nomes de *sorts* e de operações sobre estes *sorts*, cuja funcionalidade é dada por um conjunto de equações. Estas equações são similares às definidas nas linguagens de programação funcional e podem ser usadas para testar a especificação, pois cada equação define a semântica de uma determinada operação algébrica. Desta forma, uma especificação pode ser vista como um protótipo de sua implementação real, de modo que desenvolvedores e usuários finais possam realizar experimentos com a especificação, descobrindo falhas e omissões na especificação ou na definição dos requisitos, corrigindo-se problemas antes da fase de codificação. Assim, a linguagem OBJ pode ser usada para um processo de desenvolvimento interativo baseado em prototipação, onde aspectos importantes são melhorados incrementalmente através da

resposta do usuário durante todo o ciclo de vida do desenvolvimento de software. Além disso, aplicações podem ser escritas e executadas diretamente em OBJ, sem a necessidade de codificação alguma.

O OBJ foi projetado para a semântica algébrica: suas declarações introduzem símbolos para sorts e funções, suas declarações são equações, e suas computações são provas equacionais. Então, uma especificação em OBJ realmente *é* uma teoria equacional, e toda computação OBJ realmente *prova* algum teorema sobre tal teoria. Da mesma maneira que o raciocínio equacional pode ser usado para provar propriedades de números ou conjuntos, isto permite os desenvolvedores provarem (usando o próprio OBJ) propriedades de suas especificações.

A grande utilidade do OBJ reside na sua ajuda para especificar estruturas de dados algebricamente de maneira correta. É bastante difícil escrever especificações sem cometer erros. A utilização de ferramentas automatizadas, como o ambiente OBJ3[2], torna-se portanto fundamental para que especificações sejam efetivamente utilizadas em situações reais de desenvolvimento de software.

Características da linguagem OBJ

OBJ3 é o mais recente de uma série de ambientes que consiste de um interpretador e um ambiente que tem as seguintes propriedades:

1. OBJ3 é composto por um sistema lógico $\mathcal{L}$ tal que

   - as declarações em especificações OBJ são sentenças em $\mathcal{L}$,
   - a semântica denotacional de uma especificação $\mathcal{P}$ em OBJ é:
     - um modelo inicial de $\mathcal{P}$, se $\mathcal{P}$ é um "objeto" (tem semântica inicial);
     - uma "variedade" de modelos de $\mathcal{P}$, se $\mathcal{P}$ é uma "teoria" (tem semântica *loose*).
   - a semântica operacional é dada através de um eficiente sistema de dedução em $\mathcal{L}$.

2. OBJ3 suporta programação parametrizada, que é um modo bem flexível de reuso e estruturação de especificações. A notação OBJ tem uma expressividade de mais alta ordem usando somente uma lógica de primeira ordem, através das seguintes características:

   - objetos contêm código executável e teorias definem propriedades;
   - módulos parametrizados, com teorias para definir suas interfaces;
   - visões para definir instantiações de módulos genéricos;
   - expressões módulo que descrevem sistemas complexos como interconexões de módulos possivelmente parametrizados.

3. OBJ3 é baseado na lógica equacional de sorts ordenados, que fornece uma base rigorosa para:

   - tipos definidos pelo usuário;

[2]OBJ é a linguagem de especificação, enquanto que OBJ3 é o ambiente que utiliza a notação OBJ.

- tratamento de exceções;
- herança múltipla de operações;
- sobrecarga de operações.

4. O OBJ é uma linguagem cuja tipagem é muito forte e ao mesmo tempo flexível, o que permite:

   - a detecção de expressões incorretas antes de sua execução;
   - a distinção de conceitos da lógica;
   - a documentação destas diferenças aumenta a legibilidade da especificação.

5. OBJ3 também suporta algumas características como:

   - definição de sintaxe no formato mixfix com precedência de avaliação;
   - um sistema de módulos pré-definidos para implementação de tipos abstratos de dados básicos, como números e caracteres;
   - hierarquia de importação de módulos.

2.4.2 Prolog

Prolog (**Pro**gramming **Log**ic) é uma linguagem de alto nível de abstração baseada em lógica, cuja principal utilização reside no domínio da programação simbólica, não-numérica, sendo especialmente adequada à solução de problemas envolvendo objetos e relações entre objetos. O advento da linguagem Prolog reforçou a tese de que a lógica é um formalismo conveniente para representar e processar conhecimento. Seu uso evita que o programador descreva os procedimentos necessários para a solução de um problema, permitindo que ele expresse declarativamente apenas sua estrutura lógica, através de fatos, regras e consultas.

Um programa em lógica é a resolução de um determinado problema através da utilização de sentenças da lógica. Em linguagens de programação imperativas, como por exemplo Pascal ou C, uma seqüência de instruções é executada uma após a outra, que no fim converge em um resultado. Por outro lado, um programa em lógica assemelha-se mais a um banco de dados contendo várias informações e relações do tipo “o homem é inteligente” ou “X é inteligente se X é homem”.

O foco da programação em lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, os sistemas de programação em lógica reduzem a execução de programas à pesquisa da refutação das sentenças do programa em conjunto com a negação da sentença que expressa a consulta, seguindo a regra: uma refutação é a dedução de uma contradição.

Características da Linguagem Prolog

As características mais marcantes dos sistemas de programação em lógica, e da linguagem Prolog em particular, são as seguintes:

Especificações são Programas A linguagem de especificação é entendida pela máquina e é, por si só, uma linguagem de programação. Naturalmente, o refinamento de especificações é mais efetivo do que o refinamento de programas. Um número ilimitado de cláusulas diferentes podem ser usadas e predicados (procedimentos) com

qualquer número de argumentos são possíveis. Não há distinção entre programa e dados. As cláusulas podem ser usadas com grande vantagem sobre as construções convencionais para a representação de tipos abstratos de dados. A adequação da lógica para a representação simultânea de programas e suas especificações torna o Prolog um instrumento especialmente útil para o desenvolvimento de ambientes e protótipos.

Capacidade Dedutiva O conceito de computação confunde-se com o de (passo de) inferência. A execução de um programa é a prova do teorema representado pela consulta formulada, com base nos axiomas representados pelas cláusulas (fatos e regras) do programa.

Não-determinismo As computações podem apresentar múltiplas respostas, da mesma forma que podem solucionar múltiplas e aleatórias condições de entrada. Através de um mecanismo especial, denominado *backtracking*, uma série de resultados alternativos podem ser obtidos.

Reversibilidade das Relações Também conhecido como *computação bidirecional*, os argumentos de um procedimento podem alternativamente, em diferentes chamadas, representar parâmetros de entrada, em determinados casos, ou de saída, em outros casos. Logo, os procedimentos podem ser projetados para atender múltiplos propósitos. A execução pode ocorrer em qualquer sentido, dependendo do contexto. Por exemplo, o mesmo procedimento para inserir um elemento no topo de uma pilha qualquer, pode ser usado em sentido contrário, para remover o elemento que estiver no topo desta pilha.

Tríplice Interpretação dos Programas em Lógica Um programa em lógica pode ser semanticamente interpretado de três modos distintos:

1. Por meio da semântica declarativa, inerente à lógica;
2. Por meio da semântica axiomática, onde as cláusulas dos programas são vistas como entrada para um método de prova; e
3. Por meio da semântica operacional, onde as cláusulas são vistas como comandos para um procedimento particular de prova por refutação.

Essas três interpretações são intercambiáveis. Geralmente escolhe-se a abordagem que se mostrar mais vantajosa ao problema que se pretende solucionar.

Recursão A recursão, em Prolog, é a forma natural de ver e representar dados e programas. Entretanto, na sintaxe da linguagem não há iterações do tipo *for* ou *while* (apesar de poderem ser facilmente programadas), simplesmente porque eles são absolutamente desnecessárias. Também são dispensados comandos de atribuição. Uma estrutura de dados contendo variáveis livres pode ser retornada como a saída de um procedimento. Essas variáveis livres podem ser posteriormente instanciadas por outros procedimentos produzindo o efeito de atribuições implícitas a estruturas de dados. Onde for necessário, variáveis livres são automaticamente agrupadas por meio de referências transparentes ao programador. Assim, as variáveis lógicas tem um potencial de representação significativamente maior do que oferecido por operações de atribuição e referência nas linguagens convencionais.

2.4.3 Haskell

Haskell é uma linguagem de programação puramente funcional. O objetivo deste tipo de linguagem é imitar as funções matemáticas o máximo possível. Uma importante característica das linguagens funcionais é que elas não produzem efeitos colaterais. Então, dados os mesmos conjuntos de argumentos, uma função matemática produz sempre o mesmo valor como resultado da computação. A propriedade de transparência referencial significa que pode-se ver uma declaração de função

$$fx = e$$

como uma igualdade matemática entre fx e e. Então, pode-se substituir qualquer ocorrência de uma chamada por uma função sem nenhuma mudança semântica da expressão.

Grande parte do ciclo de vida do software é gasto na especificação, projeto e manutenção, e não na programação. As linguagens funcionais são excelentes para escrever especificações de tipos abstratos de dados nas quais podem ser executadas e então testadas. Tal especificação é então o primeiro protótipo do programa final.

Características da Linguagem Haskell

Examinando os benefícios de Haskell e da programação funcional, pode-se evidenciar:

Redução Os programas funcionais tendem a ser muito mais concisos que seus semelhantes imperativos.

Fácil entendimento Os programas funcionais geralmente são fáceis de entender. Uma pessoa pode ser capaz de entender um programa sem conhecimentos prévios de Haskell ou do problema especificado.

Não há erros de execução A maior parte das linguagens funcionais, Haskell em particular, são fortemente tipadas, eliminando uma enorme classe de erros que facilmente ocorrem em tempo de compilação.

Reutilização de código A tipagem forte está disponível em muitas linguagens imperativas, como por exemplo Ada. Entretanto, o sistema de tipos do Haskell é muito menos restritivo pois usa o conceito de polimorfismo. Por exemplo, um programa para ordenação de listas pode ser usado para ordenar listas de strings, listas de reais, listas de listas, etc. Portanto, o polimorfismo aumenta a reutilização de código.

Colagem forte Uma outra característica poderosa é avaliar somente as partes do programa que são necessárias para a resposta, isto é chamado de avaliação preguiçosa (*lazy*) ou avaliação sob demanda. As estruturas de dados são avaliadas somente quando necessárias para formar a resposta, e partes delas podem não ser avaliadas em nenhum momento. Isto constitui uma forma capaz de compor programas existentes, viabilizando a reutilização de programas ou partes de programas e permitindo a escrita de programas mais modulares.

Abstrações poderosas Em geral, as linguagens funcionais oferecem poderosos recursos de abstração. Uma abstração permite a definição de um objeto cujo comportamento interno é “escondido”. Um mecanismo de abstração disponível nas linguagens funcionais é a função de mais alta ordem. Em Haskell, uma função é uma estrutura

básica que pode ser livremente passada para outras funções, retornada como resultado de uma função, armazenada em uma estrutura de dados, e assim por diante. Isto evidencia que o uso correto das funções de mais alta ordem pode substancialmente aumentar a estrutura e a modularidade de muitos programas.

2.5 Considerações Finais

Neste capítulo foi apresentada a revisão de literatura da área de Prototipação de Software. Primeiramente foram mostrados os principais fatores que motivam a utilização de um processo de prototipação no desenvolvimento de software. Na seqüência, foram discutidas as abordagens e técnicas utilizadas na construção de protótipos.

Devido a este trabalho adotar um método de especificação formal para prototipação de software, foi especialmente adicionada um seção que menciona, brevemente, os principais conceitos de especificação formal para o desenvolvimento de software. Também é discutida a importância da validação das especificações formais e quais técnicas são empregadas para tanto.

Por fim, são apresentadas três linguagens declarativas (OBJ, Prolog e Haskell) que estão intimamente ligadas à construção de protótipos. Em se tratando de especificação formal de tipos abstratos de dados e prototipação dos mesmos, foi escolhida a linguagem OBJ, que além prover recursos para especificação formal e prototipação, possui uma semântica formal bem definida. Logo, um mapeamento entre as notações PROSOFT-algébrico e OBJ, possibilita a prototipação de TAD do PROSOFT-algébrico em OBJ. É relevante mencionar que um estudo detalhado acerca das três linguagens está fora do escopo do presente trabalho.

No próximo capítulo são abordados os principais conceitos da notação PROSOFT-algébrico, que é utilizada para especificar tipos abstratos de dados de sistemas de software.

3 PROSOFT

O PROSOFT é um projeto desenvolvido no Instituto de Informática da Universidade Federal do Rio Grande do Sul (UFRGS), sob coordenação do Prof. Dr. Daltro José Nunes. O principal objetivo deste projeto é construir um ambiente integrado de desenvolvimento de software para auxiliar a construção de software através do uso de métodos formais.

O ambiente, também chamado PROSOFT, visa apoiar o engenheiro de software desde a fase de análise do problema, até a fase de construção do programa, que representa uma solução (NUNES, 1994). Este ambiente é sustentado por uma base teórica sólida de tal forma que propriedades como completeza e consistência do software em construção possam ser verificadas.

O ambiente PROSOFT proporciona que novas ferramentas de desenvolvimento sejam especificadas e implementadas, para posterior integração ao mesmo. Isto somente é possível devido à integração funcional[1], sintática[2] e semântica[3] que o ambiente provê para suas ferramentas.

O ambiente PROSOFT se baseia nos seguintes conceitos (NUNES, 1994):

- Estratégia *data-driven*, que determina que para encontrar a solução do problema, as estruturas de dados devem ser definidas antes das operações (NUNES, 1992);
- Conceito de modelos, onde a solução de um problema é o modelo de alguma teoria, por exemplo VDM (JONES, 1986);
- Cálculo Lambda;
- Conceito de Tipo Abstrato de Dados (TAD), onde tipos mais complexos podem ser definidos a partir da instanciação de tipos mais simples (WATT, 1991);
- Método algébrico (WATT, 1991).

As vantagens da utilização de um ambiente na construção de software são largamente conhecidas (BALZERT, 1987), resumidamente: aumenta a qualidade, a produtividade, a correteza e a robustez do software em desenvolvimento.

Muitas ferramentas desenvolvidas em trabalhos de mestrado e doutorado foram especificadas usando o paradigma do PROSOFT. Dentre os últimos trabalhos, citam-se:

[1]Por integração funcional, entende-se que cada ferramenta do ambiente deve ter uma função bem definida e interagir com outras através de sua interface.

[2]A integração sintática determina que dados gerados por uma ferramenta possam ser lidos e usados por outras.

[3]A integração semântica, por sua vez, determina que as operações de uma ferramenta possam ser construídas a partir de outras ferramentas.

meta-modelo para reutilização de processos de software (PS) (REIS, 2003a), abordagem para execução de PS (REIS, 2003b), desenvolvimento cooperativo de software (REIS, 1998a), gerenciador de PS (REIS, 1998b), sistema especialista para o desenvolvimento de software (MORAES, 1997), modelo para decisões em grupo no desenvolvimento de software (ALVES, 2003), ferramenta para reutilização de especificações de requisitos (PIMENTA, 1998), mecanismo para visualização de PS (RABELO, 2003) e componentes de percepção visual (NUNES, 2001).

Para um melhor entendimento das próximas seções, deve-se fazer a distinção entre *PROSOFT-Java* e *PROSOFT-algébrico*. O primeiro é o nome dado à implementação do ambiente em Java. O segundo corresponde à notação formal que é utilizada para especificação de software segundo o paradigma PROSOFT.

3.1 Notação PROSOFT-Algébrico

O PROSOFT, como método[4] de especificação formal, apóia a reutilização e a modularização das especificações de TADs. A modularização torna possível "quebrar" o problema em partes menores e portanto mais fáceis de solucionar. A reutilização simplifica a especificação do TAD, tornando-a mais simples e mais rápida de ser construída.

Dado um desenvolvimento de software, os tipos de dados deste software podem ser especificados utilizando o PROSOFT-algébrico. No PROSOFT-algébrico cada ATO (Ambiente de Tratamento de Objetos) especifica somente um TAD. A construção de um ou mais ATOs representa uma solução do problema (software). Como visto na figura 3.1, os ATOs são integrados através da Interface de Comunicação de Sistema (ICS).

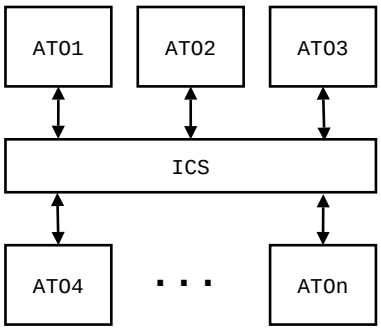


Figura 3.1: Estrutura do PROSOFT

3.1.1 ATO

De acordo com a figura3.2, um ATO é composto de cinco partes. A classe define o TAD do ATO através da instanciação de especificações parametrizadas. A segunda parte define as importações. As partes seguintes especificam a funcionalidade (assinatura) das novas operações algébricas sobre o tipo de dado, as variáveis formais e a semântica das novas operações (axiomas).

3.1.1.1 Classe

No PROSOFT-algébrico existem dois grupos de tipos de dados: primitivos (Integer, Boolean, String, Date e Time) e compostos (Conjunto, Lista, Mapeamento, Registro e

[4]O PROSOFT-algébrico provê recursos somente para especificação de tipos abstratos de dados.

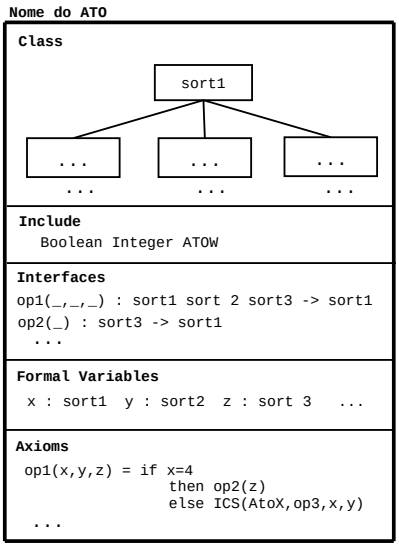


Figura 3.2: Componentes do ATO algébrico

União Disjunta). A classe de um ATO é definida através da instanciação dos tipos compostos. Exemplos de classes são: lista de inteiros, lista de conjuntos de booleanos, etc.

Para facilitar a utilização do método PROSOFT-algébrico, foi definida uma poderosa representação gráfica, inspirada nos diagramas de Jackson (JACKSON, 1985), para definição das classes. Cada tipo composto possui uma representação gráfica na forma de árvore. Na instanciação, a raiz da árvore descreve o sort definido pelo ATO, os nodos são tipos de dados compostos e as folhas são referências a outros tipos de dados.

Os tipos compostos e suas representações gráficas são mostrados na figura 3.3 através de exemplos. Para cada tipo composto e primitivo do PROSOFT, existem operações[5] geradoras, modificadoras e observadoras.

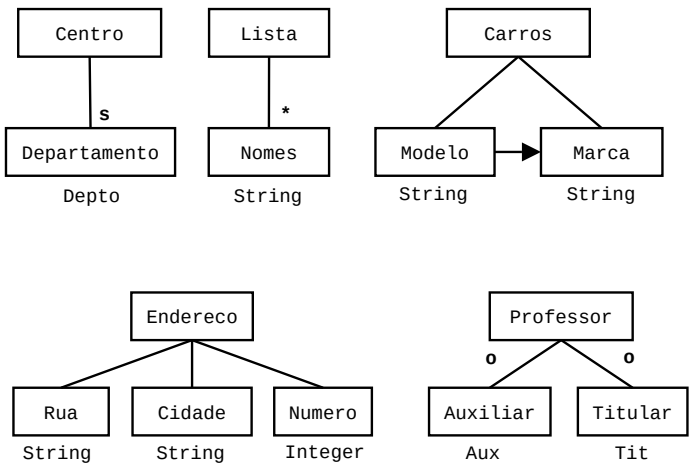


Figura 3.3: Representação gráfica dos tipos compostos do PROSOFT

Abaixo são mostradas as especificações textuais correspondentes a cada classe da figura 3.3.

specification ATOCentro

include instantiation of SET **by** ATODepto

[5]No anexo 1 podem ser encontradas as operações sobre cada tipo do PROSOFT-algébrico.

```
            using Depto for Element
            renamed using Centro for Set
                          Departamento for Depto
end specification

specification ATOLista
    include instantiation of LIST by STRING
        renamed using Nomes for Element
                      Lista for List
        using String for Nomes
end specification

specification ATOCarros
    include instantiation of MAP by STRING,ATOCaracteristicas
        renamed using Carros for Map
                      Modelo for Domain
                      Detalhes for Range
        using String for Modelo
              Caracteristicas for Range
end specification

specification ATOEndereco
    include instantiation of REGISTER by STRING,STRING,INT
        using String for D1
              String for D2
              Int for D3
        renamed using Endereco for Register
                      Rua for tag1
                      Cidade for tag2
                      Numero for tag3
end specification

specification ATOProfessor
    include instantiation of DISJOINT-UNION by ATOAux,ATOTit
        using Aux for D1
              Tit for D2
        renamed using Professor for DisjointUnion
                      Auxiliar for Aux
                      Auxiliar for tag1
                      Titular for Tit
                      Titular for tag2
end specification
```

3.1.1.2 Cláusula Include

Cada ATO algébrico manipula somente os tipos de dados que pertencem a sua classe. Em alguns casos é necessário também dispor de outros tipos de dados para definir novas operações neste ATO. Estes tipos de dados, que não pertencem a classe, são importados no ATO através da cláusula *Include* e podem ser primitivos ou também outros ATOs.

3.1.1.3 Operações Algébricas

A instanciação de uma classe importa automaticamente todas as operações pré-definidas dos tipos compostos e primitivos que compõem a classe. Por exemplo, se a classe é uma lista de strings, todas operações de criação e manipulação de listas e de strings são automaticamente incluídas na definição do ATO.

Para a definição de novas operações no ATO, se utiliza a interface para declarar a assinatura[6] de cada operação. A assinatura é composta por uma string que representa o nome da operação, seguida de seu respectivo *rank*[7], como ilustrado no exemplo abaixo:

```
include-client :  Clients Client -> Clients
```

Para cada nova operação definida num ATO, deve-se criar pelo menos um axioma para lhe dar semântica. Um axioma geralmente usa variáveis formais para generalizar sua definição. Abaixo são definidas as variáveis `client` e `clients`, com seus respectivos sorts.

```
client :  Client
clients :  Clients
```

A semântica da operação `include-client` é dada pelo axioma:

```
include-client(clients,client)=
if not(exist-client(clients,ICS(Client,get-clientcode,client)))
    then add(client,clients)
    else clients
```

Nos métodos algébricos convencionais, variáveis são utilizadas nos axiomas. O PROSOFT-algébrico conta com o artifício "_", que facilita a construção de axiomas e conseqüentemente aumenta a legibilidade dos mesmos. Utiliza-se o símbolo "_" no lugar de variáveis que representam termos do lado esquerdo de um axioma que não são modificados no lado direito. O exemplo abaixo ilustra o uso deste recurso.

```
addterm(newterm,reg-Term(_,terms))
  = reg-Term(_,cons(newterm,terms))
```

A versão do axioma acima, em uma linguagem algébrica convencional, requer o uso de uma variável ao invés de "_".

3.1.1.4 ICS

A Interface de Comunicação do Sistema tem função de integrar os diversos ATOs. Qualquer termo só pode ser criado ou modificado pelo ATO que contém sua classe. Assim, quando um ATO necessita processar termos que não pertencem à sua classe, ele deve requisitar operações, via ICS, dos ATOs que definem estes termos. A ICS é também uma operação algébrica, e sua sintaxe é:

---

[6] No PROSOFT, as operações são de 1^{a} ordem.

[7] O *rank* de uma operação é composto pela aridade da operação seguido do sort valor (resultado) da mesma.

ICS (nome do ATO, nome da operação, argumentos)

Um exemplo de chamada ICS é mostrado na figura 3.4. Segundo (NUNES, 1994), a pesquisa da operação é feita de cima para baixo dentro do ATO, se o nome do ATO coincidir e o nome da operação for uma operação pertencente ao ATO, os argumentos serão ligados aos parâmetros formais e então será executada a operação.

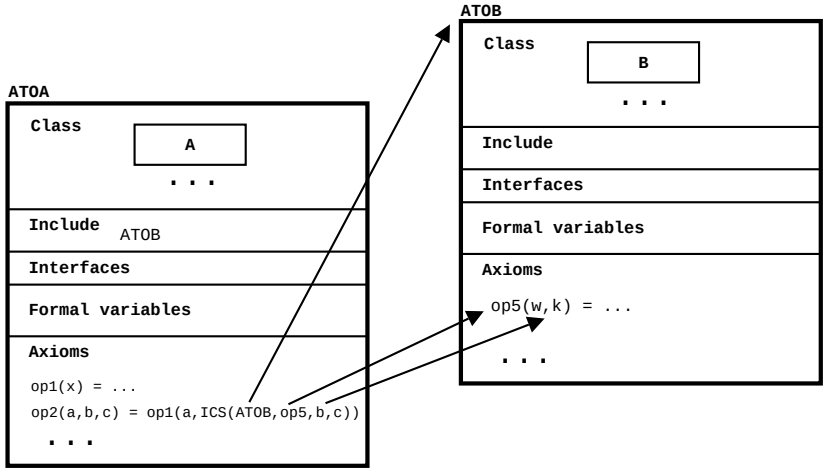


Figura 3.4: Exemplo de chamada ICS

De acordo com (NUNES, 2003), o conceito de modularização através da ICS evidencia que as operações algébricas do PROSOFT são monádicas, ou seja, cada operação trata somente o tipo de dado definido na classe do seu ATO. Considere a criação de uma nova operação chamada `op`, cuja assinatura é

```
op :  S1 S2 S3 ...  Sn -> S
```

onde `Si` e `S` são sorts. Esta operação `op` é interpretada, como no cálculo lambda, assim:

```
op :  S1 -> (S2 S3 ...  Sn -> S)
```

A operação `op`, pertencente ao ATO que define o sort `S1`, tem como domínio o sort `S1` e como imagem operações de `S2 S3 ... Sn -> S`. Dependendo do valor de `S1`, será escolhido uma das operações de `S2 S3 ... Sn -> S`. Os valores de `S1` podem ser interpretados como estados. Assim, o estado de `S1` vai determinar qual operação será escolhida. Esse processo é recorrente. No final da computação, tem-se, como resultado, a aplicação de uma operação, criando ou alterando o estado de `S`.

3.1.2 Estratégia para Construção de ATOs Algébricos

Na fase de análise do desenvolvimento de software, na maioria das vezes, o problema a ser solucionado é dividido em partes menores para facilitar sua resolução. O lado esquerdo da figura 3.5 ilustra os requisitos informais do software divididos em *n* problemas. Estes problemas menores podem ser analisados pela equipe de desenvolvimento que criará documentos de requisitos (informais e/ou semi-formais) descrevendo o software. O emprego de uma notação formal na construção de software requer que, em algum instante, os documentos de requisitos sejam formalizados. A especificação formal permite

que ambiguidades e inconsistências nos requisitos sejam reveladas, forçando o retorno à fase de análise dos requisitos.

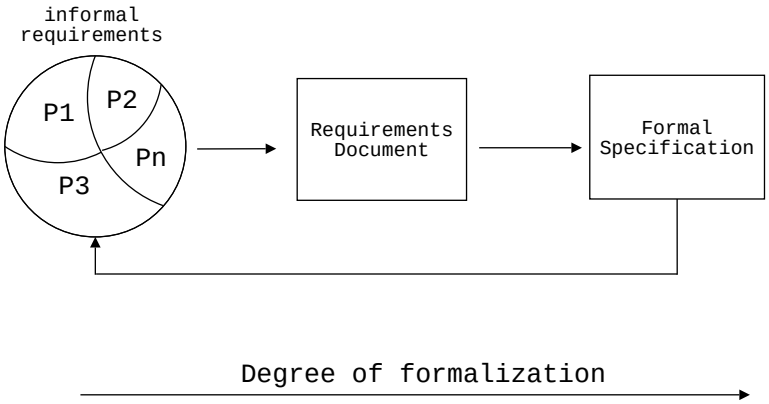


Figura 3.5: Etapas de formalização dos requisitos

No PROSOFT, o desenvolvedor é incentivado a modularizar a formalização, ou seja, a especificação do software pode conter diversos ATOs, cada qual solucionando um problema específico. Tomando como exemplo o problema da figura 3.5, a especificação formal na notação PROSOFT-algébrico conteria no mínimo *n* ATOs.

Para criar um ATO, o processo segue a estratégia *data-driven*, onde primeiramente são especificados os dados para depois serem criadas as operações sobre estes dados. Portanto, o desenvolvimento de um ATO algébrico demanda que o especificador crie basicamente:

1. a classe do ATO (TAD do ATO);

2. as operações que atuam sobre a classe.

Entre as etapas 1 e 2, existem outros passos intermediários para definição de importações (se necessário) do ATO e das variáveis formais que serão utilizadas nos axiomas. É importante ressaltar que o passo 2 consiste em definir a assinatura de cada operação criada para depois fornecer sua semântica através de axiomas.

A especificação algébrica convencional (WATT, 1991) é muito livre na definição semântica das operações. Se uma operação envolve sorts de especificações importadas, a definição da semântica desta operação pode conter muitos passos de computação, alguns sobre termos da especificação local e outros das especificações importadas. Este modo de especificar software tende a distrair o especificador do problema a ser solucionado, uma vez que é possível efetuar diretamente as computações sobre todos os sorts da especificação.

Com o recurso de modularização, provido pela operação ICS, é diferente. O especificador limita-se a definir operações de cada ATO somente com os tipos de dados da classe do ATO. Assim, quando uma operação chama uma operação de outro ATO, via ICS, o especificador não precisa se preocupar com os detalhes oriundos do tipo de dado do ATO chamado pela ICS. As chamadas ICS nos axiomas forçam o especificador a focar muito mais sua atenção no problema local que está sendo solucionado no ATO. Considere um exemplo que contêm os seguintes ATOs:

- `ATOData`: define as estruturas de dados do software;

- `ATOUse-Cases`: define diagramas use cases;

- `ATOAlgebraicProsoft`: define especificações na notação PROSOFT-algébrico.

A partir dos ATOs acima, poderia-se criar outro ATO, chamado `ATOCreate-Spec`, responsável por interpretar os requisitos funcionais (dados e sequências) do software em construção e guiar o desenvolvedor no processo de criação da especificação formal na notação PROSOFT. A definição dos axiomas da operação

```
generate-algeb-spec :  Data Use-Cases -> AlgebraicProsoft
```

conteria todo o processamento necessário para criar a especificação algébrica a partir das estruturas de dados e diagramas use cases, sem ter que se preocupar com os pormenores dos tipos `Data`, `Use-Cases` e `AlgebraicProsoft` já existentes. Logo, os ATOs podem ser considerados caixas-preta, onde sua funcionalidade é dada pelas operações que aparecem na interface do ATO. Note-se que ao criar o `ATOCreate-Spec`, o especificador foca sua atenção principalmente no processo de geração de especificações formais.

3.2 PROSOFT-Java

PROSOFT-Java é a mais recente implementação do paradigma PROSOFT, na forma de um ambiente homogêneo e integrado para desenvolvimento de software. Como o próprio nome sugere, a implementação foi realizada na linguagem Java. A atual versão do ambiente PROSOFT é resultado do esforço cooperativo entre estudantes e pesquisadores do PPGC-UFRGS e da Fakultät Informatik, da Universität Stuttgart (Alemanha), sob coordenação do Prof. Daltro José Nunes.

Segundo (NUNES, 2003), a abordagem do PROSOFT sugere que o desenvolvimento completo de um módulo de software siga duas etapas distintas: uma de especificação e outra de implementação de acordo com a especificação. Esta separação é muito útil do ponto de vista organizacional, pois qualquer processo que precise usar um módulo de software (ATO) pode fazê-lo examinando somente sua definição. Não é necessário esperar que o ATO esteja totalmente implementado, nem é necessário compreender sua implementação.

Outra vantagem desta técnica está na reusabilidade de ATOs, ou seja, um novo ATO pode ser construído utilizando outros já existentes. Este mecanismo de referência a ATOs existentes possibilita a criação de ATOS em função de outros, definindo sistemas cada vez mais complexos, sem que a complexidade do esforço de programação aumente na mesma proporção. Além do mais, a inclusão de um novo ATO no sistema é possível sem alterações nos ATOs já implementados, o que faz do PROSOFT um ambiente facilmente estendível.

É importante observar que existe uma correspondência entre ATOs algébricos e ATOs Java, conforme ilustrado na figura 3.6. Esta figura mostra que o conceito de TAD do PROSOFT-Algébrico é diretamente mapeado para a implementação em PROSOFT-Java.

O ambiente PROSOFT possui uma ferramenta para edição das classes algébricas que também gera código-fonte Java para implementá-las. A versão atual deste gerador de código, as operações adicionais, especificadas algebricamente no ATO, devem ser traduzidas manualmente para métodos escritos de acordo com a sintaxe da linguagem Java (GOSLING et al., 2000). A codificação manual não se restringe somente às opera-

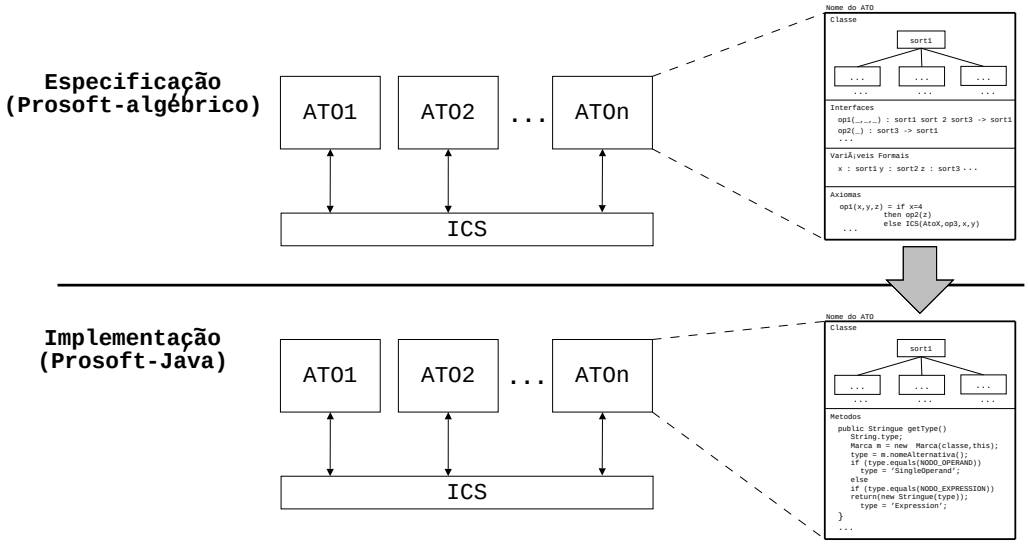


Figura 3.6: Correspondência entre ATOs algébricos e ATOs Java

ções adicionais. A interface gráfica com o usuário também deve ser manualmente implementada de acordo com os componentes visuais disponíveis do ambiente PROSOFT-Java.

3.3 Considerações Finais

Neste capítulo foi apresentada a notação PROSOFT-algébrico e suas principais características. O método algébrico foi escolhido pois é o mais adequado para especificar formalmente TADs. Este trabalho adota a notação PROSOFT-algébrico devido as vantagens intrínsicas deste método. Tais vantagens serão melhor detalhadas no capítulo 5.

Em (REIS, 2003a), o leitor encontra, em detalhes, o atual processo de implementação de ATOs Java no ambiente PROSOFT. A descrição completa do desenvolvimento de ATOs Java está disponibilizada em (SCHLEBBE; SCHIMPF, 1997; SCHLEBBE, 2002).

No próximo capítulo é apresentada a linguagem OBJ, usada como base para geração de protótipos executáveis dos ATOs algébricos.

4 OBJ

Neste capítulo é apresentado o método de especificação formal OBJ, projetado por Joseph Goghen (GOGUEN et al., 2000) como uma extensão da teoria algébrica de tipos abstratos de dados para tratamento de erros e de funções parciais, cujo objetivo é escrever e testar especificações algébricas. O OBJ, além das vantagens intrínsecas dos métodos de especificação formal, apresenta dois importantes aspectos: é parametrizado, permitindo portanto a reutilização de software e é executável, o que proporciona a implementação direta da especificação.

A linguagem OBJ utiliza equações algébricas para especificar tipos abstratos de dados que podem ser definidos independentemente de qualquer representação e especificação. A grande utilidade do método OBJ reside na sua ajuda para especificar estruturas de dados algebricamente de maneira correta. É bastante difícil escrever especificações sem cometer erros. A utilização de ferramentas automatizadas, como o OBJ, torna-se portanto, fundamental para que esses tipos de especificações sejam efetivamente utilizadas.

A notação OBJ é caracterizada por sua tipagem muito forte e ao mesmo tempo flexível. Isto permite detecção de expressões incorretas antes de sua execução, separação de conceitos da lógica que são intuitivamente distintos e a documentação destas diferenças aumenta a legibilidade da especificação.

OBJ foi projetado para ser uma linguagem executável de especificação algébrica. Uma declaração em OBJ introduz símbolos para sorts e equações, suas operações são equações e suas computações são provas equacionais de teoremas. Assim, uma especificação em OBJ é na realidade uma teoria equacional e toda computação em OBJ prova algum teorema sobre esta teoria. Ou seja, utilizando-se a própria linguagem de especificação, é possível provar propriedades da especificação construída.

OBJ3 é a versão mais recente dos sistemas OBJ. Esta versão é implementada em Common Lisp e C, sendo baseada na utilização de álgebras de sorts ordenados e na programação parametrizada, assim como na lógica equacional. A possibilidade de utilização de notação pré, pós e infixada, subsorts, módulos parametrizáveis, visões, reescrita de módulos por associatividade, comutatividade e/ou identidade e incorporação de código Lisp fornece um ambiente extensível e flexível para especificação, prototipação e construção de software, assim como construção de linguagens e ambientes para prova de teoremas. Uma grande quantidade de referências sobre sistemas construídos a partir da linguagem OBJ3 pode ser encontrada em (GOGUEN et al., 2000; GOGUEN; COLEMAN; GALLIMORE, 1992) .

4.1 Especificação algébrica em OBJ

Uma especificação em OBJ consiste de nomes de sorts e de operações sobre estas entidades, cuja funcionalidade é dada por um conjunto de equações. Estas equações são similares as definidas nas linguagens de programação funcional e podem ser usadas para testar a especificação. Desta maneira, uma especificação pode ser vista como um protótipo, de modo que projetistas e usuários finais podem realizar experimentos com a especificação, descobrindo falhas e omissões na especificação ou na definição dos requisitos, corrigindo os problemas antes da fase de implementação. Assim, a linguagem OBJ pode ser usada para um processo de desenvolvimento interativo baseado em prototipação, onde aspectos importantes são melhorados incrementalmente através do *feedback* do usuário durante todo o desenvolvimento de software. Além disso, aplicações podem ser escritas e executadas diretamente em OBJ, sem a necessidade de codificação alguma.

A especificação algébrica é dividida em duas partes: *assinatura* e *axiomas*. A assinatura define os *sorts* (ou "espécies") que estão sendo especificados, os símbolos que designam *operações* e suas funcionalidades. Os axiomas são sentenças lógicas que descrevem a semântica das operações. Considerando os tipos abstratos de dados, não é necessário definir suas representações dos valores, pois utiliza-se os axiomas para fazer asserções sobre as relações entre as operações algébricas.

Uma especificação escrita na linguagem OBJ tem uma estrutura modular, cuja unidade básica é chamada de objeto. É num objeto que está encapsulado o código executável. Outros módulos que podem constar em uma especificação OBJ são as teorias e as visões.

4.2 Assinatura

A assinatura de uma especificação algébrica é uma coleção de declarações de sorts e operações definidas sobre esses sorts.

4.2.1 Sorts

A descrição de uma especificação, seja de linguagens de programação ou de programas, envolve a definição de espécies como números inteiros, variáveis, vetores, expressões numéricas e booleanas e outros programas. Estas espécies são formalmente denominadas sorts.

Para um sort ser completamente especificado, deve-se definir as operações que podem ser aplicadas sobre ele. Por exemplo, pode-se efetuar operações aritméticas sobre números inteiros, somar todos os elementos de um vetor (se for um vetor de inteiros), efetuar operações lógicas sobre expressão booleanas e construir programas a partir da composição de outros programas.

Os nomes de sorts podem conter quaisquer caracteres, com exceções dos caracteres vírgula, branco, parênteses e subscrito "_". A declaração de sorts apresenta a seguinte sintaxe:

`sort` $\langle Sort \rangle$ `.`

onde $\langle Sort \rangle$ é um nome de sort, como no exemplo abaixo:

```
sort Bits .
```

Existem alguns sorts pré-definidos na linguagem OBJ, que não podem ser redefinidos. São eles:

- Int - números inteiros;
- Nat - números naturais;
- Rat - números racionais;
- Float - números em ponto flutuante;
- Bool - valores lógicos;
- Id - identificadores;
- Qid - identificadores que iniciam com apóstrofe (exemplo: 'abc).

4.2.2 Subsorts

É possível especificar que os elementos pertencentes a um sort também pertençam a outro sort, declarando um sort como sendo subconjunto de outro sort. A sintaxe é a seguinte:

`subsort` $\langle Sort \rangle$ `<` $\langle Sort \rangle$ `.`

`subsorts` $\langle SortList \rangle$ `<` $\langle SortList \rangle$ `< ...`

O significado desta definição é que o conjunto de objetos do primeiro sort é um subconjunto (não necessariamente o próprio) do conjunto de objetos do segundo sort. Na segunda forma de apresentação, cada elemento da primeira lista de sorts correspondente a um elemento da segunda lista, com o mesmo significado colocado antes. Exemplo:

```
obj BITS is
 sorts Bit Bits .
 subsorts Bit < Bits .
 ...
endo
```

No exemplo acima é especificado que o sort `Bit` é um subconjunto do sort `Bits`.

4.2.3 Operações

Além dos sorts, as operações definidas sobre ele também constituem a assinatura. A sintaxe para declaração de uma operação é a seguinte:

`op` $\langle OpForm \rangle$ `:` $\langle SortList \rangle$ `->` $\langle Sort \rangle$ $[Attr]$ `.`

onde $\langle OpForm \rangle$ é uma string não vazia, possivelmente contendo múltiplos *tokens* separados por espaço. Operações na forma padrão não devem incluir o caractere subscrito "_". Todos os sorts usados na declaração de uma operação devem ter sido previamente declarados.

A declaração de uma operação é feita, portanto, definindo o nome da operação, uma lista de sorts que serão seus argumentos e o sort do seu resultado. Na declaração também pode ser definidos atributos para a operação. A forma padrão para declaração de uma operação é dada como no exemplo:

```
op push :  Stack Int -> Stack .
```

que declara a sintaxe para termos da forma `push(X,Y)` de sort `Stack`, onde `X` é de sort `Stack` e `Y` de sort `Int`. A forma padrão é prefixada, ou seja, para criar um termo desta operação os argumentos devem ser separados por vírgula e inseridos dentro de um par de parênteses.

Uma outra forma para declaração de operações é através da sintaxe mixfix. Esta possibilita a declaração de operações segundo notação pré-fixada, pós-fixada, infixada ou qualquer combinação arbitrária entre elas. A declaração mixfix de uma operação utiliza *place-holders* (caracter subscrito "_"), que indicam onde os argumentos devem aparecer. Um exemplo de declaração mixfix é a operação `if-then-else`:

```
op if_then_else_fi :  Bool Int Int -> Int .
```

4.2.4 Atributos para operações

Ao declarar uma operação, é possível definir certas propriedades da operação através de atributos explícitos. Estas propriedades podem ser axiomas, tais como associatividade, comutatividade, idempotência e identidade. A utilização destes atributos acarreta em conseqüências sintáticas e semânticas, além de afetarem a ordem de avaliação da operação.

Os principais atributos que podem ser incluídos numa operação são os seguintes:

- assoc - associatividade;
- comm - comutatividade;
- idem - idempotência;
- id: <Termo> - identidade;
- prec<inteiro> - precedência.

A utilização dos atributos pode facilitar em parte a definição de especificações. No entanto, alguns cuidados precisam ser tomados e dizem respeito a cada atributo em particular (GOGUEN et al., 2000; MALCOLM; GOGUEN, 1996). Nesta seção foi citada somente a existência do conceito de atributos para operadores em OBJ. Mais detalhes sobre cada atributo podem ser encontrados em (GOGUEN et al., 2000).

4.3 Programação Parametrizada

Objetos, teorias, visões e expressões módulo fornecem um suporte formal para o reuso de especificações. A idéia básica da programação parametrizada é a alta forma de abstração, ou seja, dividir o código em pedaços menores altamente parametrizados e então construir novas especificações a partir de módulos existentes pela instanciação de parâmetros e transformação de módulos.

Uma teoria define a interface de um modulo parametrizado, ou seja, a estrutura e as propriedades requeridas pelo parâmetro atual para uma instanciação. Uma visão expressa que um certo módulo satisfaz uma certa teoria em uma maneira (alguns módulos podem satisfazer algumas teorias em mais de uma maneira). Assim, uma visão descreve uma amarração do parâmetro atual com uma teoria de interface. A instanciação de um módulo parametrizado com um parâmetro atual, usando uma visão em particular, produz um novo módulo. As expressões módulo descrevem interconexões complexas de módulos, potencialmente envolvendo instanciação, adição e renomeação de módulos.

Como um exemplo de programação parametrizada, considere o módulo parametrizado `LEXL[X]`, que fornece listas de `Xs` com ordenação lexicográfica, onde o parâmetro `X` pode ser instanciado com qualquer conjunto parcialmente ordenado. Usando os identificadores `QIDL` com sua usual ordenação (lexicográfica), então `LEXL[QIDL]` fornece a ordenação lexicográfica em listas de palavras (exemplo: títulos de livros). E `LEXL[LEXL [QIDL]]` retorna a ordenação lexicográfica em listas de frases (exemplo: listas de títulos de livros).

4.3.1 Objetos

Em OBJ, as assinaturas aparecem como parte de um módulo. Estes módulos podem ser objetos ou teorias. A distinção entre objetos e teorias é sutil à primeira vista. Um objeto é usado para definir uma estrutura padrão fixa que contém certas entidades abstratas, geralmente tipos abstratos de dados como inteiros, vetores ou valores booleanos. Por outro lado, uma teoria é usada para definir uma classe de estruturas similares, como por exemplo: grafos, autômatos ou grupos.

Um exemplo simples de objeto é o que especifica os números naturais. O objeto `NAT` abaixo introduz o sort `Nat`, que contém duas operações (`0` e `s`).

```
obj NAT is
 sort Nat .
 op 0 :  -> Nat .
 op s_ :  Nat -> Nat .
endo
```

A seguir, é definido um objeto para expressões numéricas que são construídas a partir dos números naturais.

```
obj EXPNAT is
 sort Exp .
 op 0 :  -> Exp .
 op s_ :  Exp -> Exp .
 op _+_ :  Exp Exp -> Exp .
 op _*_ :  Exp Exp -> Exp .
endo
```

A constante `0`, definida anteriormente como um elemento de sort `Nat`, agora é definida como um elemento de sort `Exp`. Da mesma forma, a operação `s` também é redefinida como um elemento de sort `Exp`. As duas operações restantes (soma e multiplicação), são definidas como operações binárias infixadas que recebem duas expressões e retornam uma expressão.

A princípio, esta redefinição de operações com mesmo nome sobre sorts distintos pode gerar certa confusão. No entanto, devido a existência de sobrecarga (*overloading*) de operações em OBJ, é possível definir operações com mesmo nome sobre diferentes argumentos. Mas, este tipo de facilidade pode causar confusão. Por exemplo, as declarações

```
op _+_ :  Nat Nat -> Nat .
op _+_ :  Int Int -> Int .
op _+_ :  Exp Exp -> Exp .
```

indicam que a soma de dois naturais é um natural, que a soma de dois inteiros é um inteiro e que a soma de duas expressões é uma expressão. A primeira vista, `3 + 5` pode pertencer a qualquer um dos três sorts acima. No entanto, neste caso particular, o resultado é indiferente, pois para qualquer uma das interpretações o resultado será `8`, que pode ser visto tanto quanto um natural, um inteiro ou uma expressão.

Como mostrado anteriormente, a linguagem OBJ tem recurso para definição de subsorts. A idéia intuitiva por trás deste construtor é que alguns conjuntos de termos podem estar contidos dentro de outros conjuntos. Por exemplo, os termos de números naturais estão contidos no conjunto dos termos inteiros, e ambos por sua vez são parte do conjunto de expressões, cujo sort é `Exp`. Em OBJ pode-se indicar este fato escrevendo-se:

```
subsort Nat < Int < Exp .
```

A declaração de subsorts como parte da assinatura leva a uma relação de ordem parcial[1] entre os conjuntos de termos. Este tipo de assinatura é denominada *assinatura de sorts ordenados*.

4.3.2 Teorias

Além dos objetos, também podem ser especificadas teorias. Uma teoria define uma classe de estruturas semelhantes. A assinatura de um autômato finito, por exemplo, pode ser expressa como

```
th AUTOM is
 sorts Input State Output .
 op i :  -> State .
 op f :  Input State -> State .
 op g :  State -> Output .
endth
```

A palavra reservada `th` define o início da definição de uma teoria, enquanto que `endth` indica o término.

4.3.3 Módulos Parametrizados

Visando a reutilização de código, o OBJ permite a parametrização de módulos. O objeto `LIST` abaixo possibilita criar listas de elementos naturais ou inteiros (por exemplo), sem a necessidade de reescrever todas equações que definem o comportamento de

[1] Uma relação de ordem parcial sobre um conjunto A é uma relação binária em A que é reflexiva, transitiva e anti-simétrica.

uma lista.

```
obj LIST[X ::  TRIV] is
 sorts List NeList .
 subsorts Elt < NeList < List .
 op nil :  -> List .
 op __ :  List List -> List [assoc id:  nil prec 9] .
 op __ :  NeList List -> List [assoc prec 9] .
 op head_ :  NeList -> Elt .
 op tail_ :  NeList -> List .
 op empty?_ :  List -> Bool .
 var X : Elt .
 var L : List .
 eq head (X L) = X .
 eq tail (X L) = L .
 eq empty?  L = L == nil .
endo
```

Os módulos podem ter mais de um parâmetro formal. Um módulo com dois parâmetros tem uma interface com sintaxe `[X ::  TH1,Y ::  TH2]`, e se as duas teorias são a mesma `[X Y ::  TH]`. Exemplo:

```
obj 2TUPLE[C1 ::  TRIV, C2 ::  TRIV] is
 sort 2Tuple .
 op «_;_» :  Elt.C1 Elt.C2 -> 2Tuple .
 op 1*_ :  2Tuple -> Elt.C1 .
 op 2*_ :  2Tuple -> Elt.C2 .
 var E1 :  Elt.C1 .
 var E2 :  Elt.C2 .
 eq 1* « E1 ; E2 » = E1 .
 eq 2* « E1 ; E2 » = E2 .
endo
```

Note-se o uso de qualificadores nos sorts `Elt.C1` e `Elt.C2`.

4.4 Variáveis e Equações

Para definir adequadamente um objeto ou uma teoria, freqüentemente é necessário utilizar equações. Uma equação em OBJ é um par de termos ligados por um sinal de igualdade. As equações são escritas de modo declarativo e interpretadas operacionalmente como regras de reescrita, que trocam instâncias obtidas por substituição no lado esquerdo pelas correspondentes instanciações do lado direito das regras. A definição de uma equação é dada pela sintaxe:

`eq` $\langle Term \rangle$ `=` $\langle Term \rangle$ `.`

onde dois termos válidos da especificação são usados. Um exemplo de equação seria `eq X + Y = Y + X .`. Esta equação define a propriedade comutativa da adição, ou

seja, dois termos podem ser somados em qualquer ordem sem afetar o resultado da soma.

Uma equação é considerada válida se todos os símbolos usados em ambos os termos tenham sido previamente declarados. No exemplo acima, as variáveis `X` e `Y` e a operação `_+_` já deveriam ter sido declaradas na assinatura.

As variáveis têm papel fundamental para escrever equações genéricas sem precisar tratar todos os casos possíveis. A sintaxe para declaração de variáveis é a seguinte:

`vars` $\langle VarIdList \rangle$ `:` $\langle Sort \rangle$ `.`

onde os nomes de variáveis devem ser separados por espaços em branco. Por exemplo:

```
vars X Y : Exp .
```

A palavra chave `var` é semelhante à `vars`, mas para declaração de uma única variável.

4.5 Visões

Um módulo pode satisfazer uma teoria de várias formas, e mesmo que exista somente uma forma, esta pode ser relativamente difícil de se encontrar. Além disso, um objeto somente poderá ter um modelo, que é a sua álgebra inicial. As visões são uma notação para descrever as diversas formas que os módulos podem satisfazer as teorias.

Mais formalmente, uma visão ϕ de uma teoria `T` para um módulo `M`, indicada por ϕ : `T` $\Rightarrow$ `M`, consiste de um mapeamento dos sorts de `T` para os sorts de `M` (preservando a relação de subsorts) e de um mapeamento das operações de `T` para as operações de `M`, preservando aridade, valor do sort, e os significados dos atributos assoc, idem, id: e idr:, tal que cada equação em `T` é verdadeira para cada modelo de `M`.

O mapeamento de sorts é dado pela sintaxe:

```
sort S1 to S1' .
sort S2 to S2' .
...
```

O mapeamento de operações é expresso com a sintaxe:

```
op o1 to o1' .
op o2 to o2' .
...
```

A sintaxe para a definição de uma visão requer que sejam designados nomes para os módulos origem e destino, e usualmente, um nome para a visão. Para mostrar um exemplo de visão, primeiramente será apresentada a teoria abaixo.

```
th POSET
 sort Elt .
 op _<_ :  Elt -> Elt .
 vars E1 E2 E3 :  Elt .
 eq (E1 < E1) = true .
```

```
 cq (E1 < E3) = true if E1 < E2 and E2 < E3 .
endth
```

Esta teoria define conjuntos parcialmente ordenados, ou seja, conjuntos que possuem uma relação de ordem parcial sobre si próprios. Essa relação é definida na teoria `POSET` pelo operador `_<_`. Existem vários conjuntos com suas respectivas relações de ordem que podem satisfazer essa teoria. Um desses conjuntos é o dos números naturais, com as suas relações ou divisibilidade, como exemplos de possíveis relações de ordem parcial. A seguinte visão mostra um dos possíveis mapeamentos entre `POSET` e `NAT`:

```
view NATG from POSET to NAT is
 sort Elt to Nat .
 op _<_ to _>_ .
endv
```

Esta é a visão `NATG`, de `POSET` para `NAT`, sendo que o operador `_<_` de `POSET` é interpretado como `_>_` em `NAT`. A palavra reservada `to` relaciona os sorts e operações formais definidos na teoria com os sorts e operações reais instanciados na visão. A definição de uma visão garante que a instanciação do módulo satisfaça os requisitos semânticos da teoria.

4.6 Exceções e Retrações

A maioria das linguagens de especificação algébricas não têm bons fundamentos semânticos e flexibilidade suficiente para o tratamento de exceções. As linguagens de especificação algébricas geralmente utilizam funções parciais para esses casos, ou seja, funções que simplesmente são indefinidas sobre determinadas condições. Ainda que esta abordagem seja satisfatória em alguns casos, não é permitida a emissão de mensagens de erros ou a recuperação dos erros. Em OBJ existe o conceito de retrações, que permite que operações possam ser aplicadas a argumentos fora do domínio de definição.

Como exemplo, considere os conjuntos dos números naturais e racionais, com sorts `Nat < Rat`. Considere agora `(2 + 2)`, sendo que a operação `_+_` foi definida apenas para os racionais. Neste caso, não há nenhum problema, pois `2` é um número natural e `Nat < Rat`, ou seja, os naturais estão contidos nos racionais. Considerando-se o caso `(- 4 / -2)!`, sendo que o operador `!` (fatorial) foi definido apenas para os naturais. Nesse caso, fica a dúvida se `(-4 / -2)`, que é definido como um racional, pode ser reduzido para um natural. O *parser* do OBJ3 irá inserir uma "retração", que é uma operação especial (nesse caso denotado por `r:Rat>Nat` e tendo aridade `Rat` e coaridade `Nat`), que será removida durante a execução, caso os subtermos consigam gerar um natural, mas caso contrário, ela permanece como uma mensagem de erro. Dessa forma, o parser irá transformar o termo `(-4 / -2)!` em `(r>Rat>Nat(-4 / -2))!`, que em tempo de execução se torna `(r:Rat>Nat(2))!` e em seguida em `(2)!`, usando a seguinte equação interna de retração:

`r:Rat>Nat(X) = X`, sendo `X` uma variável de sort `Nat`.

As operações de retração, assim como suas equações, são geradas automaticamente pelo OBJ3. As retrações são inseridas pelo parser quando forem necessárias, e não serão

vistas pelo usuário do sistema, se não houverem erros na redução.

4.7 Importação de Módulos

O OBJ3 permite quatro tipos de importação de módulos dentro de um outro módulo. A finalidade da importação é facilitar a estruturação do código em módulos, cada um com um propósito específico, facilitando a reutilização e o *debug* do código. Quando existem muitos módulos, é necessário tornar a estrutura hierárquica dos módulos explícita, de forma que quando um módulo utilize sorts ou operações declarados em outro módulo, o outro módulo seja explicitamente importado pelo primeiro, e que seja previamente definido no objeto. Dessa forma, especificações em OBJ podem ser vistas como um "grafo acíclico" de módulos abstratos.

A sintaxe para importar módulos em OBJ é seguinte:

$\langle Import \rangle$ $\langle Module \rangle$ `.`

onde $\langle Import \rangle$ pode ser **protecting**, **extending**, **including** ou **using**, e $\langle Module \rangle$ é uma expressão módulo, tal como o objeto `NAT`. As abreviações **pr**, **ex**, **inc** e **us** são abreviações para cada modo de importação.

Por convenção, se um módulo `M` importa um módulo `M'` que importa um módulo `M''`, então `M''` também é importado em `M`, isto é, qualquer tipo de importação é uma relação transitiva. Um dado módulo `M'` pode somente ser importado em `M` com um único modo de importação. Módulos que são importados mais de uma vez devido a transitividade são considerados como "compartilhados".

O significado dos modos de importação está relacionado com a semântica da álgebra inicial dos objetos, sendo que a importação de um módulo `M'` num módulo `M` pode ser:

protecting o módulo `M` não adiciona novos itens de dados dos sorts do módulo `M'`, e também não identifica nenhum item de dado antigo de `M'` (sem refugo e sem confusão);

extending as equações do módulo `M` não identificam itens de dados antigos dos sorts de `M'` (sem confusão);

including ou **using** nesse caso, não há garantia nenhuma.

O ambiente OBJ3 não verifica se as declarações de importação são corretas. Todavia, declarações de importação incorretas podem resultar em reduções incompletas em alguns casos, ou até mesmo reduções ineficientes em outros.

A seguir, é apresentado um exemplo simples de importação, onde o objeto `NAT`, que define o sort `Nat`, é importado em modo protegido.

```
obj STACK-OF-NAT is
 sorts Stack NeStack .
 subsort NeStack < Stack .
 protecting NAT .
 op empty :  -> Stack .
 op push :  Nat Stack -> NeStack .
 ...
```

4.8 Exemplo de Especificação em OBJ

Nesta seção é mostrado um exemplo de um objeto que especifica o tipo abstrato de dados fila. As operações construtoras deste tipo são: `mtq` (fila vazia) e `append` (adiciona um número natural na fila). A operação `conc` é responsável pela concatenação de duas filas. A operação `ismt` verifica se uma fila é vazia, enquanto que `invert` computa a inversão dos elementos da fila. O objeto que especifica o tipo fila e suas respectivas operações é o mostrado abaixo.

```
obj MyQueue is
  sorts Queue, Bool .
  protecting NAT .
  op mtq :  -> Queue .
  op append :  Queue Nat -> Queue .
  op _conc_ :  Queue Queue -> Queue .
  op ismt_ :  Queue -> Bool .
  op invert_ :  Queue -> Queue .
  var q, r :  Queue .
  var c :  Nat .
  eq (q)conc(mtq) = q .
  eq (q)conc(append(r,c)) = append((q)conc(r),c) .
  eq ismt(mtq) = true .
  eq ismt(append(q,c)) = false .
  eq invert(mtq) = mtq .
  eq invert(append(q,c)) = append(mtq,c)conc(invert(q)) .
endo
```

Para demonstrar que esta especificação atende corretamente aos requisitos do axioma para inverter filas, utilizou-se um exemplo para comprovação. Este exemplo é composto por uma fila de entrada (←12345←) que terá seus elementos invertidos de posição e o resultado será a fila ←54321←. Abaixo é mostrada a redução no ambiente OBJ3.

```
[rangel@tetis obj]$ ./obj.exe
                     \|||||||||||||||||/
                   --- Welcome to OBJ3 ---
                     /|||||||||||||||||\
          OBJ3 version 2.0  built: 2000 Jun 20 Tue 20:18:45
             Copyright 1988,1989,1991 SRI International
                    2001 Jul 2 Mon 13:12:19
OBJ> in mq.obj
==========================================
obj MyQueue
OBJ>reduce invert(append(append(append(append(append(mtq,1)
,2),3),4),5)) .
reduce in MyQueue : invert append(append(append(append(
append(mtq,1),2),3),4),5)
rewrites: 21
result Queue: append(append(append(append(append(mtq,5),4),
3),2),1)
OBJ>
```

4.9 Considerações Finais

Neste capítulo foi apresentada a linguagem de especificação algébrica OBJ e suas principais características. Esta linguagem é amplamente utilizada pois ela fornece recursos muito poderosos para especificação algébrica. Outro fator que ajudou a disseminar o OBJ foi a implementação do seu sistema de reescrita de termos (SRT).

Muitas linguagens algébricas dispõem somente de um gerador de código para alguma linguagem de programação (como C++ ou Java, por exemplo). O ambiente OBJ3, por sua vez, possui uma eficiente implementação de um SRT, utilizado interativamente por linha de comando. A grande vantagem do OBJ3 é que o usuário pode reduzir termos visualizando explicitamente como estão sendo computadas cada redução.

Além do SRT do OBJ3, o ambiente gera código fonte em C++. A implementação atual deste gerador de código ainda é bem limitada e não suporta muitos dos recursos fornecidos pela linguagem. O leitor interessado pode encontrar mais detalhes deste gerador em (GOGUEN, 2001). Para estudos mais aprofundados acerca do OBJ indica-se o estudo da semântica formal da linguagem, que pode ser encontrada em (GOGUEN; MESEGUER, 1992).

O próximo capítulo apresenta a tradução de PROSOFT-algébrico para OBJ, que é o centro desta dissertação.

5 MODELO PROPOSTO DE PROTOTIPAÇÃO DE SOFTWARE PARA O PROSOFT

Segundo (LUQI, 1992; LUQI; GOGUEN, 1997), os requisitos do sistema mudam entre 30 e 40% sem o uso de uma atividade de prototipação no decorrer do desenvolvimento. A prototipação rápida de software é uma abordagem que visa reduzir os erros introduzidos na análise de requisitos através da validação precoce da especificação do software em desenvolvimento.

A utilização de um método de prototipação de software juntamente com uma notação formal enfatiza a especificação formal do problema e expõe o usuário a um sistema "operante" o mais rápido possível, de modo que usuários e desenvolvedores sejam capazes de executar e validar a especificação dos requisitos funcionais do sistema. Assim, os benefícios da especificação formal e da abordagem de prototipação para o desenvolvimento de software são combinados e uma descrição de sistema bem estruturada, executável e sem ambiguidade pode ser desenvolvida.

Com o objetivo de aumentar a qualidade do software, várias tecnologias vêm sendo experimentadas no sentido de apoiar o ciclo de vida do software. Um dos esforços mais significativos corresponde à definição de metodologias voltadas a disciplinar o processo de desenvolvimento através do estabelecimento de etapas bem definidas, proporcionando, desta forma, um mecanismo de controle para o processo. Como conseqüência, muitas organizações de desenvolvimento de software buscam a maturidade no processo de software, usando medidas como o SEI-Capability Maturity Model for Software (SW-CMM (PAULK et al., 1993)) para estruturar as iniciativas de melhoria de processo.

O modelo SW-CMM propõe cinco estágios, sendo que uma organização que desenvolve software pode se encaixar desde o nível inicial (1), onde o desenvolvimento é caótico, até o nível otimizado (5), onde o gerenciamento de software é ideal. Nesse modelo, algumas áreas-chave são identificadas para que uma organização passe de um nível ao outro do modelo. Um pré-requisito para atingir o nível 4 do SW-CMM consiste em garantir que o sistema em desenvolvimento foi bem compreendido quanto aos requisitos, o que é facilitado com o uso de especificações formais.

A especificação formal de softwares do mundo real considera a existência de muitos módulos distintos. Estes módulos podem ser classificados quanto ao modelo de computação adotado. O modelo de computação, ou modo como cada módulo executa suas operações, pode ser dividido em computação sequencial e computação concorrente. Por computação sequencial entende-se que cada operação é executada sequencialmente, ou seja, uma após a outra. Por outro lado, a computação concorrente permite que operações sejam executadas paralelamente, isto é, várias operações podem ser executadas ao mesmo tempo.

De acordo com (NUNES, 2003), na especificação de sistemas concorrentes, paralelos, móveis, etc, os objetos passivos, denominados dados[1], podem ser tratados[2] por objetos ativos, denominados agentes[3]. Quando agentes entram em sincronismo, alguns enviam dados e outros recebem.

No cálculo CCS (MILNER, 1989), por exemplo, agentes podem se comunicar para trocar dados. Na figura abaixo, estão representados quatro agentes: `ag1`, `ag2`, `ag3` e `ag4`. O agente `ag1` pode oferecer pela sua porta $\overline{\texttt{T}}$ um dado `v`. Os agentes `ag2`, `ag3` e `ag4` podem receber este dado pela porta `T`.

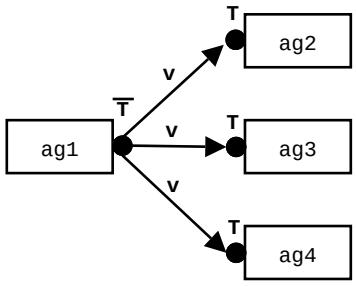


Figura 5.1: Exemplo de agentes concorrentes em CCS

Quando o agente `ag1` estiver pronto para oferecer o dado `v` e os agentes `ag2`, `ag3` e `ag4` estiverem prontos para receber, então, não deterministicamente, um deles recebe o dado `v`.

Para um software que envolve ambos os modelos de computação, geralmente utiliza-se um método para especificar a concorrência entre os agentes e outro para especificar os tipos de dados que são trocados entre os agentes concorrentes. Os métodos mais conhecidos que abordam concorrência, presentes na literatura, são: Gramática de Grafos (ROZENBERG, 1997; EHRIG et al., 1999), Redes de Petri (PETERSON, 1981; REISIG, 1985), LOTOS (BOLOGNESI; BFINKSMA, 1987), CCS (MILNER, 1989) e cálculo-π (MILNER; PARROW, 1992; MILNER; PARROW; WALKER, 1992). Para a especificação formal dos tipos abstratos de dados o método mais adequado a ser utilizado é o método algébrico (WATT, 1991). Entre os métodos algébricos estão: PROSOFT-algébrico (NUNES, 1992, 1994), OBJ (GOGUEN et al., 2000), Act-One (EHRIG; MAHR, 1985; HANSEN, 1987), RAISE (GEORGE, 1991), Larch (GUTTAG; HORNING, 1993) e CASL (ASTESIANO et al., 2002).

O contexto deste trabalho consiste em prototipar tipos abstratos de dados da notação PROSOFT-algébrico, desconsiderando aspectos como concorrência, paralelismo ou mobilidade entre agentes do software. Portanto, o modelo de computação é sequencial, onde cada operação, sobre um tipo de dado, é executada sequencialmente.

Este capítulo apresenta o modelo de prototipação **Pro**totyping**Tool**, projetado para o ADS PROSOFT, incluindo as características da tradução entre as linguagens PROSOFT-algébrico e OBJ.

[1] dados, também denominados valores, termos, informações etc.
[2] tratados, criados, consultados, alterados, mudados, transformados, etc.
[3] agentes, também denominados de processos, tasks, threads, módulos, etc.

5.1 O Diferencial do PROSOFT-algébrico

O método PROSOFT-algébrico foi escolhido como linguagem para prototipação de tipos abstratos de dados devido ao seu diferencial sobre os outros métodos algébricos. Para explicar melhor este diferencial, deve-se abordar as duas facetas do PROSOFT: a notação e o ambiente.

A notação PROSOFT-algébrico é muito importante pelo grande benefício que traz à especificação algébrica. Este benefício chama-se *classe*. Conforme mostrado no capítulo 3, uma classe no PROSOFT tem uma representação gráfica, que facilita a criação de tipos de dados complexos através da instanciação de tipos de dados mais simples. A notação gráfica usada nas classes dos ATOs torna as classes mais fáceis de serem construídas, analisadas e entendidas do que a correspondente textual. Dispondo deste recurso visual[4], a definição das operações sobre o tipo e suas respectivas semânticas é simplificada, visto que o especificador pode raciocinar sobre o tipo de dado analisando sua representação gráfica.

Outra vantagem do PROSOFT-algébrico é a metodologia para construção dos ATOs, que incentiva modularizar a especificação e focar a atenção do desenvolvedor principalmente no problema que está sendo definido em cada ATO.

Para um desenvolvedor utilizar um método de especificação algébrico, ele precisa dispor de conhecimentos acerca da teoria algébrica. Por outro lado, devido a facilidade intrínseca do método, o PROSOFT-algébrico permite que usuários menos experientes sejam capazes de definir tipos de dados e interfaces de operações sobre estes tipos, somente com conhecimento oriundo das linguagens de programação.

Mais um diferencial do PROSOFT é a implementação do ambiente, chamado PROSOFT-Java. Para facilitar a vida do desenvolvedor, existe no ambiente uma ferramenta que gera a implementações de ATOs algébricos na linguagem Java. Estas implementações geradas respeitam um padrão de comunicação entre os ATOs-Java através da ICS-Java (implementação da ICS algébrica), como mostrado no capítulo 3. Baseado neste padrão, novas ferramentas (implementações de ATOs algébricos) podem ser facilmente agregadas ao ambiente, tornando-se parte dele. Em outras palavras, o ambiente PROSOFT-Java é auto-estendível e este conceito não existe em outro método de especificação algébrico. O processo que gera código Java a partir dos ATOs algébricos é descrito em maiores detalhes no capítulo 7.

5.2 Alternativas para Geração de Protótipos

Antes de iniciar a discussão acerca das alternativas para geração de protótipos executáveis para o PROSOFT-algébrico, é importante lembrar as técnicas utilizadas para validação de especificações formais.

No capítulo 2 foram descritas, sucintamente, as técnicas: prova de teoremas, geração automática de código e interpretação. As duas primeiras técnicas são usadas para provar propriedades da especificação formal, enquanto que as duas últimas fornecem um protótipo que possibilita a validação dos requisitos do software por execução. Em todas as técnicas o objetivo é tentar provar que a especificação captura os requisitos propostos pelos usuários do software em desenvolvimento. Mas, por outro lado, nenhuma técnica pode garantir plenamente que existe uma correspondência entre os requisitos informais e as especificações formais (HOARE, 1987).

[4] A definição de operações e axiomas não têm representação gráfica.

Dada uma especificação formal, muitas propriedades podem ser provadas usando um provador de teoremas ou um verificador de modelos. Considere o caso no qual a equipe de desenvolvimento criou uma especificação formal de acordo com o que eles entenderam ser os requisitos do usuário, mas esta não corresponde fielmente aos reais requisitos. Propriedades seriam provadas sobre a especificação, e o desenvolvimento avançaria para fases posteriores, podendo chegar até na implementação para descobrir que o software criado não é uma solução adequada para o problema proposto pelo usuário. Nestes casos que a validação por execução é muito útil, pois permite que usuários e desenvolvedores "testem" a especificação formal para validar os reais requisitos do usuário. É baseado nesta motivação que este trabalho apresenta a utilização de uma técnica para executar as especificações de tipos abstratos de dados, segundo a notação PROSOFT-algébrico.

Como o PROSOFT é um método algébrico, existem duas técnicas que podem ser adotadas para obter protótipos executáveis. A primeira seria a geração automática de código, em alguma linguagem de programação, que implementasse a especificação. Esta técnica não é adequada pois a especificação já deve estar bem detalhada, o que muitas vezes não é possível logo no início do desenvolvimento.

A segunda técnica consiste em interpretar a especificação algébrica em um sistema de reescrita de termos. A grande vantagem desta técnica reside na possibilidade de poder testar as especificações mesmo com um nível de abstração bem alto, onde nem todos detalhes da especificação foram completamente definidos. Esta técnica apresenta como desvantagem a impossibilidade de poder testar o protótipo no ambiente real onde o software será executado, ou mesmo com componentes de software já existentes. Esta desvantagem é suprida pela geração de código, por isso as duas técnicas podem ser consideradas complementares.

Conseqüentemente, um mecanismo de prototipação para o PROSOFT-algébrico envolveria o desenvolvimento de algoritmos para implementar a reescrita de termos. Assim especificações de ATOs algébricos poderiam ser executadas através da redução de termos algébricos.

A primeira vista, a solução mais adequada seria a definição de um sistema de reescrita de termos para o PROSOFT-algébrico. Analisando mais profundamente, vê-se que esta solução envolve a implementação de algoritmos extremamente complexos.

Como alternativa, propõe-se o mapeamento da notação PROSOFT-algébrico para uma outra linguagem que já disponha uma implementação. Em (RANGEL, 2001), foram estudadas as linguagens Haskell, Prolog e OBJ com intuito de escolher qual seria a melhor para o mapeamento proposto. Note-se que o foco de estudo foram as linguagens declarativas, pois elas enfatizam o que é realmente o problema sem ter que definir como ele deve ser solucionado, como nas linguagens imperativas.

A decisão pelo mapeamento visa a reutilização de implementações já existentes e bem aperfeiçoadas. A literatura apresenta casos semelhantes. Entre eles é o que mapeia especificações em cálculo-π (MILNER; PARROW, 1992; MILNER; PARROW; WALKER, 1992) para History Dependent Automata(HDA) (MONTANARI; PISTORE, 1998). Neste caso, já existiam muitas ferramentas implementadas para simulação e prova de propriedades para HDA, mas queria-se estudar como provar propriedades no cálculo-π. Como não existiam ferramentas implementadas para cálculo-π, optou-se pela definição de um mapeamento entre cálculo-π e HDA. E através deste mapeamento, as ferramentas já existentes para HDA também poderiam ser usadas para provar propriedades de especificações em cálculo-π.

A proposta apresentada neste trabalho, portanto, em linhas gerais, corresponde à es-

pecificação de um tradutor entre a notação PROSOFT- algébrico (apresentada no capítulo 3) e a notação OBJ (apresentada no capítulo 4). A escolha da linguagem OBJ é devido a dois motivos: semelhança com o PROSOFT-algébrico e existência de uma implementação muito eficiente de um sistema de reescrita de termos.

A prototipação de tipos abstratos de dados proposta neste trabalho pode ser classificada, segundo abordagem e técnica empregadas, como uma instância da abordagem exploratória que emprega a técnica de especificações formais executáveis.

5.3 Comparativo entre PROSOFT-algébrico e OBJ

Nesta seção é traçado um comparativo entre as notações PROSOFT-algébrico e OBJ, conforme a tabela abaixo.

Tabela 5.1: Características comparadas

Característica	**PROSOFT**	**OBJ**
Lógica equacional	multisortida de 1^a ordem	sorts ordenados de 1^a ordem
Modularização	Sim	Sim
Módulo principal	ATO	objeto
Especificações parametrizadas	Sim	Sim
Importação de módulos	precária	sofisticada
Notação gráfica	Sim (classe)	Não
Legibilidade	boa	ruim
Poder de expressão	funções totais	funções parciais
Sintaxe	mixfix	mixfix
Atributos para operações	Não	Sim
Equações condicionais	if-then-else	if-then-else e equação condicional
Semântica formal definida	Não	Sim
Ambiente implementado	Sim	Sim
Sistema de reescrita de termos	Não	Sim
Implementação do ambiente	Java	Lisp e C
SO compatíveis	Vários	Linux
Provador de Teoremas	Não	Sim
Gerador de código	Sim (Java)	Sim (C++)
Parser	Não	Sim
Tipos básicos	set, list, map, reg, du, boolean, integer, string, date, time	Truth-Value, Bool, Nat, Int, Rat, Float, Qid, Qidl, Id, Tuple2, Tuple3
Metodologia de projeto	Sim	Não

5.3.1 Modularização

No PROSOFT-algébrico a modularização dá-se através da definição de vários ATOs. Já em OBJ, a modularização é permitida pela definição de vários objetos.

5.3.2 Especificações Parametrizadas

No PROSOFT, usam-se os tipos básicos parametrizados (lista, conjunto, mapeamento, registro e união disjunta) para definir a classe do ATO. Note-se que a instanciação dos tipos parametrizados já existentes é fácil. Por outro lado, para o usuário definir (e usar) seus ATOs parametrizados não é trivial, pois requer não somente a definição da especificação algébrica parametrizada, mas também a definição da representação gráfica do tipo. E estas definições devem ser incorporadas à implementação da ferramenta responsável pela criação das classes gráficas. Um exemplo de ATO parametrizado seria um que define o tipo pilha e a instanciação deste tipo poderia ser aplicada a qualquer outro tipo (pilha de inteiros, pilha de strings, pilha de pilha de listas de strings, etc.).

Em OBJ, o especificador tem a liberdade de criar seus objetos parametrizados e utilizá-los diretamente, sem ter que lidar com a implementação do ambiente. No OBJ o especificador define objetos parametrizados de uma forma rigorosa, pois teorias são usadas para especificar os requisitos mínimos que devem ser atendidos pelo módulo que instanciará os parâmetros formais do objeto parametrizado.

Considere a especificação do objeto parametrizado `LEXL[X :: POSET]` que fornece listas de `X`s segundo a ordenação lexicográfica entre elementos. A teoria `POSET` define as propriedades de conjuntos parcialmente ordenado para que `X` seja somente instanciado com conjuntos parcialmente ordenados. Exemplos de instanciação do objeto `LEXL` são: `LEXL[NAT]`, `LEXL[LEXL[NAT]]`, etc.

5.3.3 Importação de módulos

Se uma linguagem tem o conceito de modularização, ela deve fornecer mecanismos para importação de módulos para composição de especificações maiores pela importação e/ou instanciação de outras especificações.

Tanto no PROSOFT como no OBJ, a instanciação de módulos é feita de modo semelhante. A diferença fica por conta da notação gráfica da classe contra a notação puramente textual empregada no OBJ. Além desta diferença, o especificador tem mais liberdade para renomear sorts e operações em OBJ.

Sob o ponto de vista da importação de módulos, o PROSOFT não tem nenhuma referência explícita a este recurso, apesar de ser utilizado em especificações de trabalhos anteriores a este. A importação de outros ATOs que não pertenciam à classe era dada através de chamadas à operação ICS, pois uma chamada ICS é uma forma de importação de ATOs.

Em OBJ, a importação de módulos é bem desenvolvida, oferecendo vários tipos de importação de módulos, como visto no capítulo 4.

5.3.4 Legibilidade

Considerando a legibilidade das especificações, a diferença entre PROSOFT-algébrico e OBJ é grande. Primeiro, a notação gráfica para representação das classes PROSOFT torna o tipo de dado do ATO bem mais fácil de “ler” que seu similar em OBJ. Abaixo é mostrado um exemplo que compara a classe de um ATO com uma instanciação em OBJ.

A instanciação de um objeto parametrizado (`REGISTER3`) que é equivalente à classe da figura 5.2 é a seguinte:

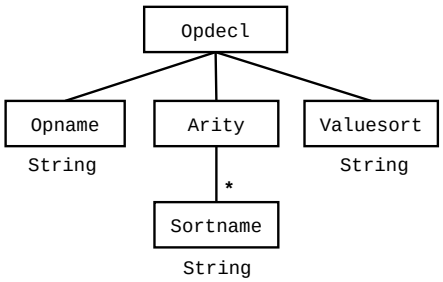


Figura 5.2: Hierarquia de módulos em PROSOFT algébrico

```
obj ATOOpdecl is
 pr (REGISTER3 *(sort Register to Opdecl,op reg to reg-Opdecl,
    op select1 to select-Opname,op select2 to select-Arity,
    op select3 to select-Valuesort))[STRING,(LIST *(sort List
    to Arity))[STRING],STRING] .
 ...
endo
```

No que se refere ao restante da especificação, a diferença fica por conta da estrutura indentada dos axiomas em PROSOFT contra uma única string que define uma equação em OBJ. A seguir é ilustrada a semântica de uma operação em PROSOFT e em OBJ. O axioma da operação `fillarity` no PROSOFT é

```
fillarity(int,opdecl)
=
if isemptyarity(int)
 then opdecl
 else
  if sortarity(int) == "Boolean"
   then fillarity(delarity("Boolean",int),ins-arity("Bool",opdecl))
   else
    if sortarity(int) == "Integer"
     then fillarity(delarity("Integer",int),ins-arity("Int",opdecl))
     else fillarity(delarity(sortarity(int),int),ins-arity(sortarity
            (int),opdecl))
```

enquanto que em OBJ é:

```
eq fillarity(int,opdecl) = if isemptyarity(int) then opdecl
else if sortarity(int) == "Boolean" then fillarity(delarity(
"Boolean",int),ins-arity("Bool",opdecl)) else if sortarity(
int) == "Integer" then fillarity(delarity("Integer",int),
ins-arity("Int",opdecl)) else fillarity(delarity(sortarity(
int),int),ins-arity(sortarity(int),opdecl)) fi fi fi .
```

5.3.5 Atributos para operações

Os atributos de operações especificam certas propriedades da operação algébrica. Entre estas propriedades estão: associatividade, comutatividade, idempotência, identidade, etc.

Na notação OBJ, é possível definir estas propriedades explicitamente na definição da assinatura da operação. Por exemplo, a operação

```
op _or_ :  Bool Bool -> Bool [comm] .
```

indica que esta operação é uma operação binária infixada comutativa sobre valores booleanos. É importante lembrar que a reescrita de termos, envolvendo operações associativas/comutativas, é um problema NP completo que inviabiliza a implementação de uma solução ótima para todos os casos. Por isto, em especificações grandes, não se pode esperar que operações com estas propriedades sejam sempre executadas rapidamente.

No PROSOFT-algébrico não é possível definir estas propriedades explicitamente, mas podem ser definidos axiomas que especificam estas propriedades. Portanto, para definir a propriedade comutativa para a operação `or` é necessário o seguinte axioma **`m or n = n or m`**, onde **`m`** e **`n`** são variáveis de sort Boolean.

5.3.6 Implementação do ambiente

O ambiente PROSOFT-Java como o próprio nome sugere foi implementado na linguagem Java, enquanto que o ambiente OBJ3 foi implementado em Lisp e C.

5.3.7 Sistemas operacionais compatíveis

Devido ao fato do PROSOFT-Java ter sido implementado em Java, qualquer sistema operacional com uma máquina virtual Java pode rodar o ambiente. Por outro lado, o OBJ3 é de uso restrito ao SO Linux.

5.3.8 Provador de teoremas

O PROSOFT-algébrico não possui nenhum provador de teoremas implementado. Por outro lado, no OBJ, uma computação em OBJ3 já é uma prova de teoremas, ou seja, cada redução efetuada é uma prova, pois as especificações são teorias lógicas e a computação é realmente uma dedução. Um assistente de provas usado para esta notação é o Kumo (GOGUEN et al., 2000).

5.3.9 Gerador de código

O PROSOFT-Java possui um gerador de código Java para as classes dos ATOs, mas a versão atual do gerador não abrange os outros elementos (importação, interfaces de operações, variáveis e axiomas) do ATO algébrico.

A última versão do ambiente OBJ3 fornece um gerador de código para C++. Este gerador é de uso bem limitado, pois não é possível a utilização de muitos recursos na notação OBJ, como: tipos pré-definidos (Nat, Int, Bool, Id, etc.), teorias, programação paramatrizada e atributos de operações.

5.3.10 Parser

A atual versão do PROSOFT-Java tem somente um editor para especificar a classe do ATO algébrico. Até então não existia nenhum editor (nem parser) para especificação dos elementos restantes do ATO. A implementação do tradutor proposto neste trabalho demandou a implementação de um editor completo de ATOs algébricos, como pode ser visto no capítulo 7.

O OBJ3 possui um parser bem eficiente para reconhecimento de sua notação. Entre os destaques do parser estão os avisos que ele fornece ao usuário, indicando possíveis erros sintáticos/semânticos na especificação.

5.4 ProTool: Tradutor de Especificações

A especificação de um software do mundo real demanda que o software seja dividido em diversos módulos para facilitar o entendimento de cada parte específica e ao mesmo tempo incentivar a reutilização destes módulos no desenvolvimento de outros sistemas.

A especificação de um software, em PROSOFT-algébrico, requer, muitas vezes, a definição de vários ATOs algébricos. Desse modo, a tradução de uma especificação de software, em PROSOFT-algébrico, para OBJ demanda que todos os ATOs algébricos, pertencentes a esta especificação, sejam traduzidos para objetos em OBJ. Cada objeto criado em OBJ corresponde a um protótipo executável[5] do ATO algébrico. A figura 5.3 ilustra abstratamente este processo de tradução.

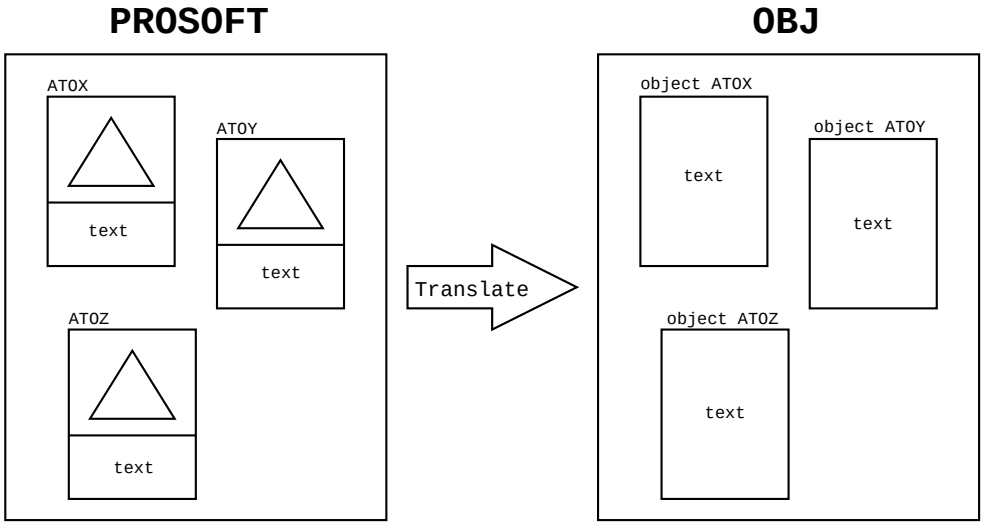


Figura 5.3: Tradução de ATOs algébricos para objetos em OBJ

No lado esquerdo da figura aparece uma especificação em PROSOFT-algébrico que engloba: `ATOX`, `ATOY` e `ATOZ`. O triângulo dentro de cada ATO simboliza a classe gráfica que define seu tipo de dado. Abaixo do triângulo aparece `text` que denota importações, interfaces de operações, variáveis formais e axiomas do ATO. A seta `Translate` indica a tradução dos ATOs algébricos para objetos em OBJ, que é a parte central deste trabalho. No lado direito da figura aparece o resultado da função `Translate` que são os objetos OBJ que prototipam os ATOs algébricos. Note-se que o nome de cada objeto criado em OBJ utiliza o mesmo nome do ATO, ou seja, o `ATOX` em PROSOFT-algébrico gera o objeto chamado `ATOX` e assim por diante. O elemento `text` representa a especificação puramente textual de cada objeto OBJ, mas este elemento não é igual ao `text` do ATO.

O processo indicado na figura 5.3 pode ser simplificado para traduzir somente um ATO para um objeto em OBJ, como mostrado na figura 5.4.

A execução dos protótipos dos ATOs algébricos segue conforme a figura 5.5. Cada objeto, em OBJ, criado a partir do ATO algébrico, deve ser carregado dentro do ambiente OBJ3 para permitir a redução de termos algébricos sobre este objeto. Na seqüencia,

[5]Executável através da reescrita de termos do ambiente OBJ3.

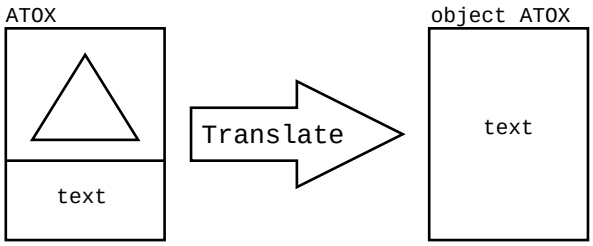


Figura 5.4: Tradução de um ATO algébrico para um objeto em OBJ

o desenvolvedor pode realizar reduções de termos algébricos sobre estes objetos para computar as operações inseridas no termo. O ambiente OBJ3 é responsável por computar cada operação de acordo com as equações que definem sua semântica. Como se sabe da teoria algébrica, este termo de entrada é reduzido até o ponto que todas operações foram completamente computadas. Este termo resultante é chamado de forma normal do termo de entrada. Analisando o termo de entrada e o termo na forma normal o desenvolvedor é capaz de verificar se cada operação especificada atende realmente os requisitos informais. Mais detalhes de como proceder estas reduções podem ser encontrados no capítulo 7.

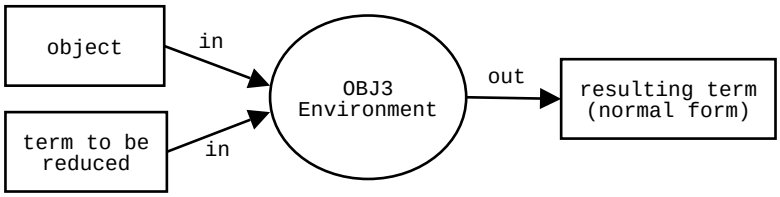


Figura 5.5: Redução de termos usando o ambiente OBJ3

Quando o software possui muitos tipos de dados e muitas operações, a especificação provavelmente conterá muitos módulos que são conectados para formar a especificação completa. Nestes casos que a notação PROSOFT leva grande vantagem sobre a notação OBJ. Como mostrado no comparativo entre as duas notações, quanto maior a especificação dos tipos de dados pior a legibilidade das especificações OBJ, devido a notação puramente textual. Ao utilizar o OBJ para especificar softwares grandes, a equipe de desenvolvimento tende a ficar mais propensa a introduzir erros na especificação. Isto porque para criar as especificações a equipe necessita focar-se não somente na definição do problema, mas também na complexidade da notação formal utilizada. Assim, justifica-se o uso da notação PROSOFT em vez de utilizar diretamente a OBJ para especificação.

As próximas seções deste capítulo mostram os detalhes do processo de tradução de um ATO algébrico para um objeto OBJ.

5.4.1 Introdução ao Mapeamento de PROSOFT-algébrico para OBJ

Um ATO algébrico é composto por cinco partes (classe, importação, interface, variáveis, axiomas) que são utilizados para criar automaticamente um objeto, em OBJ, que prototipa a especificação deste ATO.

A classe de um ATO é uma representação gráfica para a instanciação dos tipos de dados existentes no PROSOFT-algébrico. Por outro lado, em OBJ a instanciação de especificações já existentes é dada por uma sintaxe puramente textual.

A figura 5.6 mostra a tradução da classe do `ATOText` para um objeto em OBJ. A classe mostrada no lado esquerdo da figura é a instanciação do tipo lista com elementos

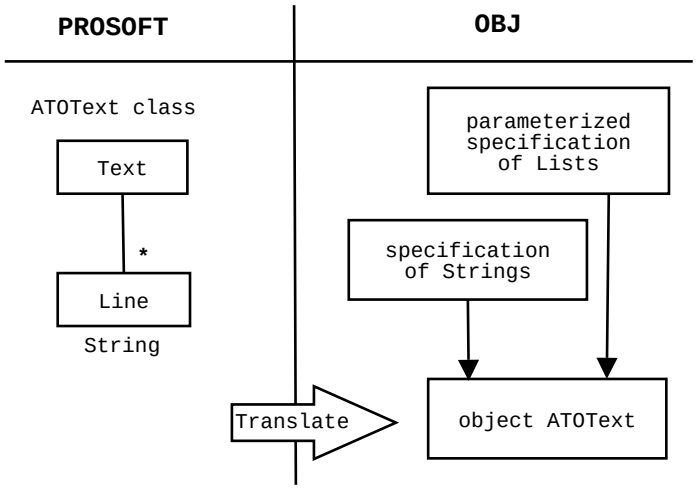


Figura 5.6: Exemplo de tradução de classe

string. Para criar em OBJ uma especificação equivalente, seria necessário dispor de uma especificação parametrizada para listas e uma especificação para o tipo string. As setas verticais da figura simbolizam que a especificação parametrizada de lista é instanciada com a especificação de string, formando o objeto que prototipa a classe do `ATOText`.

Portanto, para tornar possível o mapeamento dos tipos do PROSOFT diretamente em OBJ, criou-se objetos parametrizados, em OBJ, para os tipos compostos (conjunto, lista, mapeamento, registro e união disjunta) do PROSOFT-algébrico. Além dos tipos compostos, foram criados objetos que especificam os tipos primitivos disponíveis (string, boolean, integer, date e time) no PROSOFT.

Como não existia a especificação formal dos tipos de dados do PROSOFT-algébrico, a criação das especificações equivalentes em OBJ foram feitas através da análise da tabela[6] de construtores de cada tipo composto, como mostrado em (REIS, 1998a).

Para os tipos primitivos do PROSOFT, não existia nenhuma documentação citando as operações algébricas disponíveis, muito menos suas respectivas semânticas. Logo, para especificar estes tipos em OBJ, foram criados objetos que definem a semântica de cada tipo. Como os tipos boolean e integer já existiam em OBJ, estes foram usados para dar semântica aos respectivos do PROSOFT. As especificações dos tipos restantes (string, date e time) foram criadas e podem ser encontradas no anexo B.

Inicialmente, almejava-se criar especificações parametrizadas em OBJ para os tipos compostos do PROSOFT-algébrico, mas seria impossível gerar especificações parametrizadas genéricas para os tipos registro e união disjunta. Para os tipos lista, conjunto e mapeamento, foi possível criar uma especificação genérica o suficiente que torna possível instanciar listas, conjuntos e mapeamentos de qualquer outro tipo de dado.

A definição matemática de um registro é um produto cartesiano de vários domínios. Por sua vez, a definição matemática de uma união disjuntiva é a união de vários domínios que não têm elementos em comum. Ambos tipos têm uma característica em comum, ou seja, um número ilimitado de domínios. Portanto, a solução adotada para os tipos registro e união disjunta é semelhante, visto que a causa do problema é idêntica.

Foram realizadas várias tentativas na busca por uma especificação parametrizada genérica para qualquer número de domínios, mas nenhuma teve sucesso. Buscou-se na lite-

[6]Esta tabela continha o nome de cada operação algébrica, sua respectiva funcionalidade (rank da operação) e exemplos informais de uso.

ratura especializada uma solução para este problema, mas diversos autores adotavam artifícios, como intervalos ($1 \ldots n$), para referenciar um número indeterminado de domínios. Tais artifícios impossibilitam que uma especificação concreta seja formulada e então executada por um sistema de reescrita de termos.

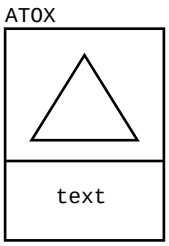
A solução adotada consiste em criar uma especificação parametrizada para cada caso particular dos tipos registro e união disjunta. Por exemplo, se na classe existe um registro de três domínios é criado um objeto parametrizado para registros com exatamente três domínios. Assim, uma classe do PROSOFT pode ser traduzida para uma instanciação de objetos parametrizados em OBJ.

Uma solução semelhante é adotada pela linguagem algébrica Larch (GUTTAG; HORNING, 1993), onde são usadas abreviações na especificação para definir os tipos produto cartesiano (registro) e união disjunta. Tais abreviações tornam o código mais legível, pois poupam o usuário de definir todas operações algébricas que especificam o tipo de dado. Exemplos de uso destas abreviações podem ser encontrados em (GUTTAG; HORNING, 1993).

Por fim, a tradução de um ATO algébrico para um objeto em OBJ consiste de traduzir os seguintes elementos do ATO: classe, importação, interfaces de operações, variáveis formais e axiomas.

5.4.2 Criando um Objeto a partir do ATO Algébrico

O primeiro passo para tradução de um ATO consiste em criar um objeto em OBJ que seja equivalente ao ATO. No exemplo abaixo é ilustrado este processo.




```
object ATOX is
 ...
endo
```

Figura 5.7: Tradução de um ATO para um objeto

Considere o ATOX da figura 5.7. A estratégia é criar um objeto com o mesmo nome do ATO, como mostrado no lado direito da figura. Os três pontos indicam o corpo da especificação do objeto, que será gerado a partir da tradução da classe, importações, interfaces, variáveis formais e axiomas do ATO.

5.4.3 Tradução da Classe

A tradução da classe do ATO emprega duas estratégias distintas, dependendo se existe ou não recurssividade. Entende-se por recursão quando um nodo folha da classe referencia o nodo raiz.

5.4.3.1 Tradução da Classe sem Recursão

Quando a classe de um ATO não é recursiva, sua tradução para OBJ consiste em instanciar os objetos parametrizados que foram criados para definir os tipos de dados do PROSOFT em OBJ. O anexo B contém as especificações que definem os tipos parametrizados (conjunto, lista, mapeamento, registro e união disjunta) e os tipos primitivos (string, boolean, integer, date e time). Os objetos parametrizados seguem o padrão da notação

OBJ:

```
SET[E]
LIST[E]
MAP[D,R]
REGISTER2[D1,D2]
...
REGISTERn[D1,...,Dn]
DISJOINTUNION2[D1,D2]
...
DISJOINTUNIONn[D1,...,Dn]
```

onde `SET` é o nome do objeto parametrizado que especifica o tipo conjunto. Entre colchetes aparecem os parâmetros formais de cada especificação parametrizada.

Para cada tipo primitivo do PROSOFT existe um objeto que o define, cujos nomes são: `STRING`, `BOOL`, `INT`, `DATE` e `TIME`. Portanto, exemplos de instanciações dos objetos parametrizados são: `SET[STRING]` (conjunto de strings), `MAP[STRING,DATE]` (mapeamento de strings para datas), `REGISTER3[STRING,INT,BOOL]` (produto cartesiano de string, inteiro e valores booleanos), etc. Estas instanciações utilizam um recurso do OBJ chamado *visões default*. Este tipo de visão é uma abreviação que captura sua noção intuitiva, e portanto, economizando ter que defini-la explicitamente.

Para criar os tipos de dados da classe de um ATO dentro de um objeto OBJ, faz-se a importação e instanciação dos objetos parametrizados que correspondem à classe do ATO algébrico. As importações em OBJ requerem um modo de importação. Para as classes, o utilizado é o *protecting*, que preserva a semântica inicial de cada objeto (sem confusão nem refugo).

As instanciações em OBJ permitem a renomeação de sorts e operações, utilizando a sintaxe

```
(NOME-OBJETO-PARAMETRIZADO *(sort oldsort to newsort,
                             op oldop to newop))[Y1,...,Yn] .
```

onde as renomeações aparecem dentro do parênteses, após o símbolo *****, e separadas por vírgula.

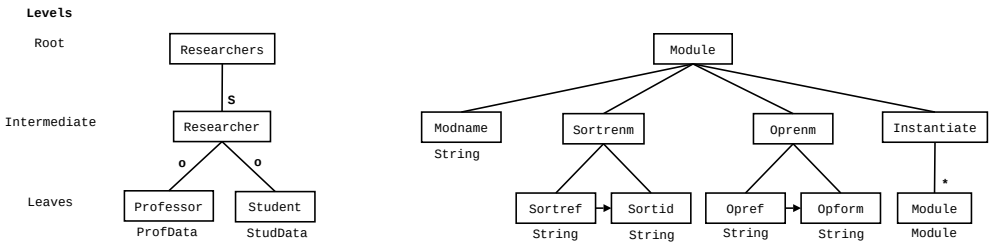


Figura 5.8: Exemplos de classes não recursivas

A figura 5.8 ilustra os níveis das classes. O nível raiz define o sort principal do tipo de dado que a classe especifica. A classe da esquerda especifica um conjunto de registros, cujo sort principal é `Researchers`. Os nomes que aparecem dentros dos retângulos são usados tanto para renomeação de sorts, quanto para renomeação de operações da especificação parametrizada.

Ao gerar a instanciação em OBJ que corresponde ao tipo de dado definido na classe do ATO, deve-se renomear os sorts e operações mostrados na tabela 5.2.

Tabela 5.2: Sorts e operações dos objetos parametrizados que são renomeados na instanciação

Objeto	Sort	Operação
SET	Set	–
LIST	List	–
MAP	Map	–
REGISTER*n*	Register *n*	reg select1 select*n*
DISJOINTUNION*n*	DisjointUnion*n*	apply1 . . . apply*n* is1 . . . is*n* get1 . . . get*n*

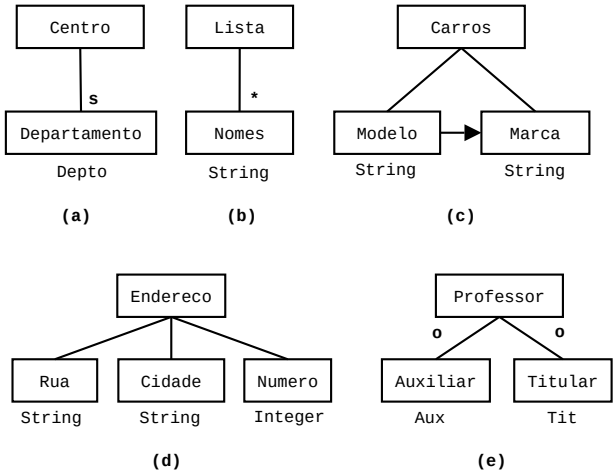


Figura 5.9: Exemplos de tradução de classes sem recursão

Para ilustrar melhor a tradução da classe de um ATO, considere as classes da figura 5.9. A tradução da classe do ATOCentro (item **a**) para OBJ é

```
pr (SET *(sort Set to Centro))[(ATODepto *(sort Depto to
                                          Departamento))] .
```

onde o objeto parametrizado `SET` é instanciado com o tipo de dado definido no `ATODepto`. O sort `Set` é renomeado para `Centro` e o sort `Depto` para `Departamento`. As traduções das outras classes (**b**,**c**,**d** e **e**) da figura 5.9 são mostradas a seguir:

```
pr (LIST *(sort List to Lista))[STRING] .

pr (MAP *(sort Map to Carros))[STRING,(ATOCaracteristicas *(sort
                                Caracteristicas to Detalhes))] .

pr (REGISTER3 *(sort Register to Endereco,op reg to reg-Endereco,
```

```
            op select1 to select-Rua,op select2 to select-Cidade,
            op select3 to select-Numero))[STRING,STRING,INT] .

pr (DISJOINTUNION2 *(sort DisjointUnion to Professor,op apply1 to
               Professor-Auxiliar,op apply2 to Professor-Titular,
               op get1 to get-Auxiliar,op get2 to get-Titular,
               op is1 to is-Auxiliar,op is2 to is-Titular))
               [(ATOAux *(sort Aux to Auxiliar)),(ATOTit
               *(sort Tit to Titular))] .
```

As operações dos tipos registro e união disjunta são renomeadas segundo o padrão mostrado nos exemplos acima. A tabela 5.3 apresenta os sorts e as operações do objeto parametrizado antes e depois da renomeação.

Tabela 5.3: Renomeações na instanciação dos items **d** e **e** da figura 5.9

Sorts e operações	
Antes	**Depois**
Register	Endereco
reg	reg-Endereco
select1	select-Rua
select2	select-Cidade
select3	select-Numero
DisjointUnion	Professor
apply1	Professor-Auxiliar
apply2	Professor-Titular
get1	get-Auxiliar
get2	get-Titular
is1	is-Auxiliar
is2	is-Titular

Como pode ser visto, o processo de tradução da classe é *top-down*, ou seja, começa pela raiz da classe para depois traduzir os nodos folha. Se a classe conter mais de um tipo composto, o processo de tradução é recursivo. As traduções das classes da figura 5.8 são:

```
pr (SET *(sort Set to Researchers))[(DISJOINTUNION2
   *(sort DisjointUnion to Researcher,op apply1 to
   Reseacher-Professor,op apply2 to Reseacher-Student,
   op is1 to is-Professor,op is2 to is-Student,
   op get1 to get-Professor,op get2 to get-Student))
   [(ATOPROFDATA *(sort ProfData to Professor)),
   (ATOStudData *(sort StudData to Student))]] .


pr (REGISTER4 *(sort Register to Module,op reg to reg-Module,
      op select1 to select-Modname,op select2 to select-Sortrenm,
      op select3 to select-Oprenm,op select4 to
      select-Instantiate))[STRING,(MAP *(sort Map to Sortrenm))
```

```
[STRING,STRING],(MAP *(sort Map to Oprenm))[STRING,STRING],
(LIST *(sort List to Instantiate))[ATOMODULE]] .
```

Cabe ressaltar que nas classes que possuem tipos primitivos em suas folhas, alguns elementos da classe gráfica não tem correspondente na textual em PROSOFT-algébrico, e portanto, não tem correspondente na tradução para OBJ.

5.4.4 Tradução da Classe com Recursão

A notação PROSOFT provê recursos para a definição de tipos abstratos de dados recursivos. Entende-se por recursão a referência à própria classe, como no exemplo da figura 5.10.

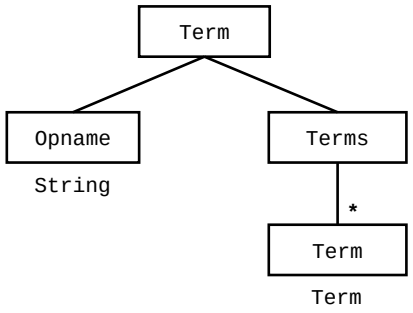


Figura 5.10: Exemplo de classe com recursão direta

No Prosoft-algébrico existem dois tipos de recursão, direta e indireta. A recursão direta ocorre quando um nodo folha da classe referencia a própria classe, como na figura 5.10. Por outro lado, a recursão indireta é caracterizada quando:

1. um ATO dado referencia um ATO' em sua classe; e

2. o ATO' referencia o ATO em sua classe.

É importante ressaltar que a recursão indireta envolve sempre mais de um ATO. Considere o exemplo da figura 5.11. Neste, a classe do ATOXX referencia, em uma de suas folhas, o ATOYY. A classe do ATOYY tem uma folha referenciando o ATOZZ, que por sua vez tem uma folha referenciando o ATOXX. Portanto, este é um caso de recursão indireta com somente uma recursão. No caso geral, um número indeterminado de recursões é permitido, mesmo que em níveis diferentes[7].

A primeira tentativa em traduzir classes recursivas seguia a mesma estratégia das classes sem recursão. Logo, verificou-se que seria impossível, pois a linguagem OBJ não fornece recursos explícitos para tanto.

No exemplo em OBJ abaixo, a recursão funciona devido ao ambiente OBJ3 detectar a recursão, para então, criar automaticamente um submódulo contendo todas as definições do módulo corrente (SELF-REF) até o ponto que aparece a recursão. Logo, a recursividade em OBJ funciona nestes casos muito simples, não satisfazendo os requisitos[8] das classes recursivas dos ATOs algébricos.

[7]Por exemplo, uma segunda recursão seria a classe do ATOYY referenciar também a classe do ATOXX. Neste caso existiriam recursões indiretas no nível 2 e 3.

[8]Uma classe recursiva pode ter vários nodos folha contendo a recursividade.

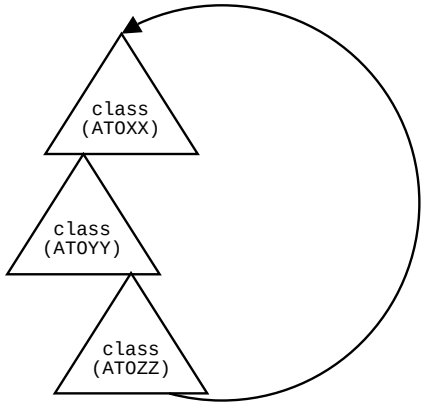


Figura 5.11: Exemplo de classe com recursão indireta

```
obj SELF-REF is
  sort A .
  op a :  -> A .
  pr LIST[A] .
endo
```

A tradução proposta neste trabalho limita-se às classes com recursão direta, pois o tratamento de recursões indiretas é bem mais complexo, devido a possibilidade de haver muitos ATOs na recursão. E, mais ainda, deve-se garantir que pelo menos um dos ATOs da recursão indireta possui o tipo união disjunta, possibilitando a "saída" da recursão. Cabe evidenciar que as recursões diretas são as mais utilizadas pelos especificadores.

A estratégia empregada para superar o problema de simular a recursão direta em OBJ consiste em criar as especificações de todos os tipos compostos que aparecem na classe, declarando sorts, operações, variáveis e axiomas para cada tipo. Obviamente, o objeto resultante da tradução é um tanto difícil de ser entendido, pois trata-se de uma especificação *flat* de todas especificações dos tipos compostos da classe recursiva. Considerando a classe recursiva direta da figura 5.10, a tradução para OBJ resulta no objeto:

```
obj ATOTerm is
  sort Term .
  pr STRING .
  sort Terms .
  *** operações, variáveis e axiomas para registro
  op reg-Term :  String Terms -> Term .
  op select-Opname :  Term -> String .
   ...
  *** operações, variáveis e axiomas para lista
  op emptylist :  -> Terms .
  op cons :  Term Terms -> Terms .
   ...
endo
```

No exemplo da figura 5.10, o nodo folha com a recursão direta contém o mesmo sort (`Term`) da raiz da classe. Caso fosse diferente, não seria utilizado para renomeação de sort, mas sim para renomear operações, se o nodo raiz da classe é um registro ou

uma união disjunta. No capítulo 6, a especificação do ATOClass apresenta uma classe recursiva direta bem mais complexa que a do ATOTerm.

A tradução de uma classe recursiva é dividida em duas etapas. A primeira consiste em declarar os sorts dos tipos compostos que aparecem na classe e importar tipos primitivos e ATOs que os nodos folha da classe referenciam. Mais detalhadamente, este primeiro passo consiste em:

- Declarar o sort principal da classe (nome do nodo raiz);
- Importar e renomear ATOs que apareçam em nodos folha da classe;
- Importar tipos primitivos (nodos folha);
- Declarar sorts para os tipos compostos da classe (nodos intermediários).

A segunda etapa é responsável por criar no objeto todas as operações algébricas que definem cada tipo composto (conjunto, lista, mapeamento, registro e união disjunta) que aparece na classe. Nesta tarefa de criação são declaradas as assinaturas das operações, as variáveis formais e os axiomas.

As operações criadas são basicamente as mesmas, a menos das renomeações de sorts e operações, dos objetos parametrizados dos tipos compostos mostrados no anexo B. Por exemplo, as operações que dão semântica ao tipo registro da figura 5.10, são:

```
op reg-Term : String Terms -> Term . op select-Opname : Term ->
String . op select-Terms : Term -> Terms . var x : String . var y
: Terms . eq select-Opname(reg-Term(x,y)) = x . eq
select-Terms(reg-Term(x,y)) = y .
```

Para especificar toda a classe da figura 5.10, deveriam ser acrescidas as assinaturas de operações, variáveis e equações para o tipo lista (`Terms`) e importar o tipo String. Exemplos completos dos objetos *flat*, gerados pela tradução, podem ser encontrados no capítulo 8.

5.4.5 Tradução da Importação

As versões do PROSOFT anteriores a este trabalho não incluíam a importação explícita nos ATOs algébricos. A motivação para definir explicitamente as importações é devido a duas razões. A primeira é definir explicitamente o que está sendo importado e também prever futuramente modos de importações mais avançados no PROSOFT-algébrico. A segunda razão é facilitar a implementação do editor de ATOs algébricos, que é mostrado no capítulo 7. A importação é dada pela cláusula *Include* do ATO. Cabe ressaltar que atualmente todos os dados contidos na cláusula *Include* poderiam ser obtidos a partir das interfaces de operações e das chamadas ICS.

A tradução das importações do ATO é direta. O modo de importação utilizado[9] no PROSOFT é equivalente ao modo de importação *protecting* do OBJ. O exemplo abaixo ilustra a tradução.

[9]Quando a notação PROSOFT for estendida para incorporar modos de importações mais sofisticados, a tradução de importações deve ser revista.

```
Include
Boolean Integer ATOX String ATOY
```

Para cada módulo da cláusula *Include* do ATO deve ser criado uma linha de código OBJ que corresponde à importação do módulo. Considerando a lista de importações acima, sua tradução seria:

```
pr BOOL .
pr INT .
pr ATOX .
pr STRING .
pr ATOY .
```

Na tradução os nomes que aparecem na lista *Include* do ATO são convertidos para letras maiúsculas, que em OBJ representam os nomes dos objetos que especificam o TAD. Usa-se os objetos `BOOL` e `INT` do OBJ para dar semântica aos tipos Boolean e Integer do PROSOFT-algébrico.

5.4.6 Tradução das Interfaces de Operações

A tradução das interfaces de operações definidas pelo usuário do ATO também é direta, apenas prestando atenção à renomeação dos sorts *Boolean* para *Bool* e *Integer* para *Int*. Considere as seguintes interfaces de operações de um ATO:

```
del-oprenm    :  String Module -> Module
isempty-inst  :  Module -> Boolean
```

A tradução para a notação OBJ é:

```
op del-oprenm :  String Module -> Module .
op isempty-inst :  Module -> Bool .
```

Ou seja, deve-se somente acrescentar a palavra-chave **op** antes da declaração e o ponto após o sort valor da operação.

5.4.7 Tradução das Variáveis Formais

Para traduzir as variáveis formais usadas nos axiomas de um ATO, basta renomear os sorts “Boolean” e “Integer” (quando existirem variáveis destes sorts) e declarar as variáveis seguindo a notação OBJ. As seguintes declarações de variáveis de um ATO

```
terms   :  Terms
opname  :  String
x       :  Boolean
```

são traduzidas para:

```
var terms :  Terms .
var opname :  String .
var x :  Bool .
```

Na tradução são acrescidos: a palavra-chave **`var`** e um ponto no final da declaração da variável.

5.4.8 Tradução dos Axiomas

A tradução de axiomas envolve traduzir os construtores da linguagem origem para os construtores da linguagem destino. Como os construtores do PROSOFT-algébricos foram especificados em OBJ, a tradução de construtores é direta, com exceção das operações ICS e do artifício "_" do PROSOFT que não têm correspondentes em OBJ.

Como visto no capítulo 3, a utilização do símbolo "_" serve para identificar termos algébricos que não são usados no lado direito da definição do axioma. Um axioma contendo "_" pode ser convertido facilmente para um equivalente que utilize variáveis formais. Por exemplo, o axioma abaixo envolvendo o artifício "_"

```
calculate-bill(reg-VCDRShop(_,ds,ren,_),dcod,today)
 = ICS(ATODiscs,get-price,ds,dcod) *
       calc-days(ICS(ATORentals,get-loandate,ren,dcod),today)
```

é equivalente a um axioma sem "_", como

```
calculate-bill(reg-VCDRShop(cs,ds,ren,ms),dcod,today)
 = ICS(ATODiscs,get-price,ds,dcod) *
       calc-days(ICS(ATORentals,get-loandate,ren,dcod),today)
```

onde as variáveis **`cs`** e **`ms`** devem ser declaradas com seus respectivos sorts.

A tradução de axiomas proposta aqui não considera a utilização do artifício "_". Para não privar os usuários do PROSOFT deste recurso, este é disponibilizado na implementação do editor de ATOs (ver capítulo 7) e o editor é responsável pela criação de variáveis formais para substituição nos axiomas e posterior tradução para OBJ.

Outro detalhe a ser considerado é a operação ICS. Como um ATO pode instanciar outro ATO em sua classe, ou então importar este ATO (através da cláusula *Include*), as operações que são chamadas via ICS podem ser chamadas diretamente em OBJ. Logo, a tradução do axioma mostrado acima é traduzido para:

```
calculate-bill(reg-VCDRShop(cs,ds,ren,ms),dcod,today)
 = get-price(ds,dcod) * calc-days(get-loandate(ren,dcod),today)
```

5.5 Considerações Finais

Neste capítulo foi apresentado o modelo de prototipação ProTool para o ADS PROSOFT. O objetivo deste capítulo foi descrever informalmente as regras necessárias para efetivar a tradução de ATOs algébricos para objetos em OBJ.

Um problema inerente dos métodos algébricos é a legibilidade dos termos. Quanto maior for a especificação do tipo abstrato de dado, maiores serão os termos algébricos, que por sua vez ficam difíceis de serem criados e interpretados pelos desenvolvedores de software. O PROSOFT-algébrico e seu conceito de classe gráfica facilitam de certa forma a criação dos termos algébricos, pois o entendimento dos tipos abstratos de dados, de especificações puramente textual, é bem complexo.

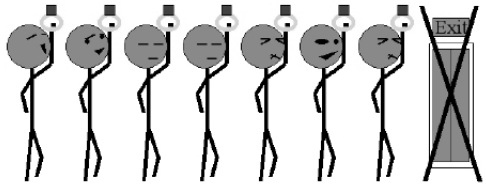


Figura 5.12: RV para Fila

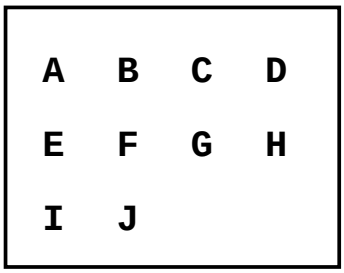


Figura 5.13: RV para conjunto de strings

Existem propostas que aumentam a legibilidade dos termos algébricos através da utilização de gráficos para simbolizar estes termos. Em (BARDOHL; CLASEN, 1994) é proposta uma linguagem visual para geração de figuras a partir de termos algébricos. A notação algébrica empregada é o ACT-ONE (EHRIG; MAHR, 1985; HANSEN, 1987), mas como a linguagem gráfica é flexível, ela pode ser definida para outras notações algébricas, como o PROSOFT-algébrico e o OBJ, por exemplo. Nas figuras 5.12 e 5.13 são apresentadas duas representações visuais (RV): uma para um termo que define uma fila e outra para um termo que define um conjunto de strings.

No próximo capítulo é mostrada a formalização das estruturas de dados e operações empregadas na tradução.

6 ESPECIFICAÇÃO FORMAL DO PROTOOL

Este capítulo é responsável pela formalização do modelo ProTool, o qual englobou a especificação de 13 tipos de dados e derivou a implementação de um protótipo para edição de ATOs algébricos juntamente com o tradutor para OBJ.

Mais uma vez, o objetivo do ProTool é analisar a especificação de um ATO algébrico e gerar um código OBJ (textual) para ser carregado no ambiente OBJ3.

Os principais ATOS que formalizam o ProTool são:

ATOMetaATO – define a classe MetaATO. Termos desta classe são ATOs algébricos;

ATOOBJ – define a classe OBJ. Termos desta classe são objetos OBJ;

ATOText – define uma lista de strings. Um termo da classe Text é o resultado da tradução de um ATO algébrico para OBJ.

Cabe ressaltar que no ATOMetaATO e no ATOOBJ especifica-se somente a estrutura das notações, desconsiderando alguns detalhes sintáticos.

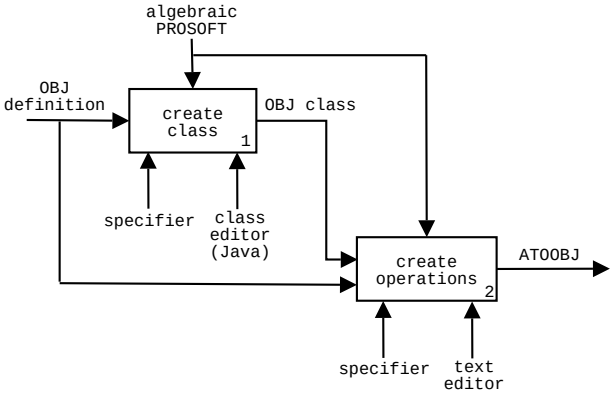


Figura 6.1: Processo de criação do ATOOBJ

O processo de formalização do ProTool envolveu, primeiramente, a definição do ATOOBJ, conforme mostrado no diagrama SADT da figura 6.1. A entrada da atividade 1 é a definição da linguagem OBJ, de acordo com seus construtores. Nesta definição foi utilizada a notação PROSOFT para especificar a estrutura do OBJ. E para tanto, utilizou-se a ferramenta de edição de classes do PROSOFT-Java para criar a classe OBJ. Na atividade 2 são especificadas as operações algébricas necessárias para construir termos da classe OBJ. Além destas operações, foram também definidas operações para traduzir uma classe OBJ em uma lista de strings (classe Text).

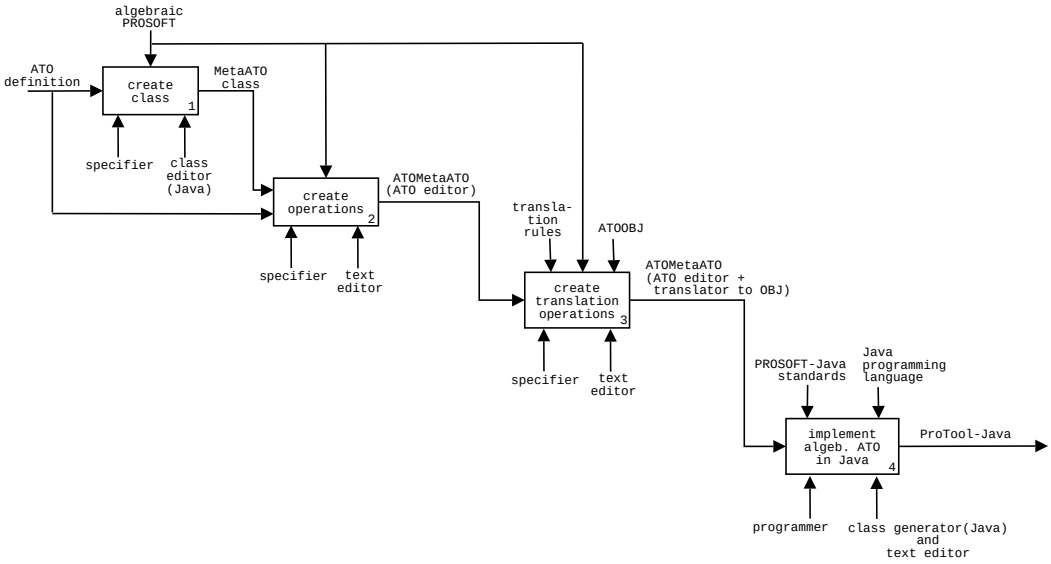


Figura 6.2: Processo de criação do ATOMetaATO e implementação do ProTool-Java

A etapa seguinte de formalização englobou a definição do `ATOMetaATO`, o estabelecimento da tradução e a implementação dos ATOs do ProTool no ADS PROSOFT, gerando a ferramenta ProTool-Java. O diagrama SADT da figura 6.2 ilustra este processo. As atividades `1` e `2` criam a classe MetaATO e as operações para construir termos da classe. A saída da atividade `2` é o `ATOMetaATO` que define o editor de ATOs algébricos. Na atividade `3`, são criadas as operações de tradução para OBJ, e portanto, a saída desta atividade é a especificação formal que contém o editor de ATOs algébricos e o tradutor para OBJ. Nas atividades de construção das operações algébricas dos ATOs, foi utilizado um editor[1] de texto simples, pois ainda não existia uma ferramenta específica.

A atividade `4` é responsável pela implementação do ProTool no ADS PROSOFT. Para tanto, foi utilizado o gerador de código Java para implementar as classes. As operações, por sua vez, foram implementadas manualmente usando um editor de textos. A saída desta atividade é a implementação da ferramenta ProTool-Java, que é composta pelo editor de ATOs algébricos e pelo tradutor para OBJ. Cabe ressaltar que entre as atividades `3` e `4` os ATOs foram validados, como mostrado no capítulo 8.

O diagrama SADT da figura 6.3 mostra o caso geral de utilização da ferramenta ProTool-Java na construção de ATOs algébricos. A entrada da atividade `1` consiste nos requisitos do software em desenvolvimento. Tais requisitos são utilizados para guiar o especificador na tarefa de criação das classes dos ATOs que formalizarão o software. Seguindo a estratégia *data-driven* do PROSOFT, na atividade `2` são definidas as operações que atuam sobre as classes. Logo, como saída se tem os ATOs algébricos que formalizam o software. Na atividade `3`, os ATOs construídos podem ser traduzidos para OBJ, para fins de validação. Ao gerar os objetos OBJ, são utilizadas as regras de tradução formalizadas neste capítulo. A atividade `4` ilustra o processo de redução de termos no ambiente OBJ3, onde com os resultados das formas normais, o desenvolvedor tem recursos para verificar se o que ele especificou atende realmente aos requisitos do software.

Os três principais ATOs do ProTool estão relacionados como ilustrado na figura 6.4.

No `ATOMetaATO` são definidas operações para construção de termos desta classe que são usadas para editar as especificações de ATOs. Além destas operações, este ATO pos-

[1]Editor de texto como o Notepad do MS Windows, por exemplo.

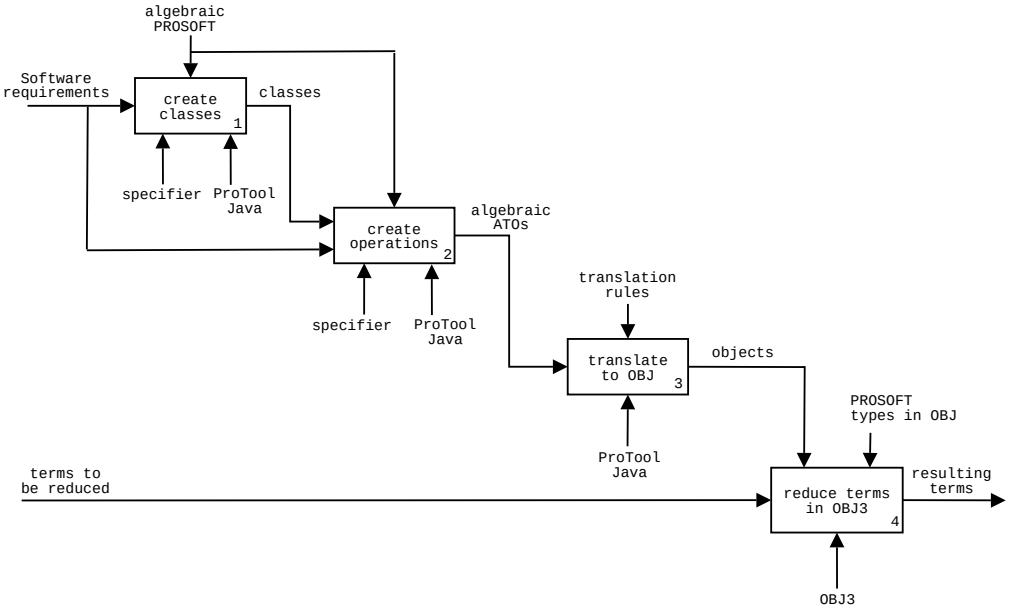


Figura 6.3: Utilização do ProTool-Java no desenvolvimento de ATOs algébricos

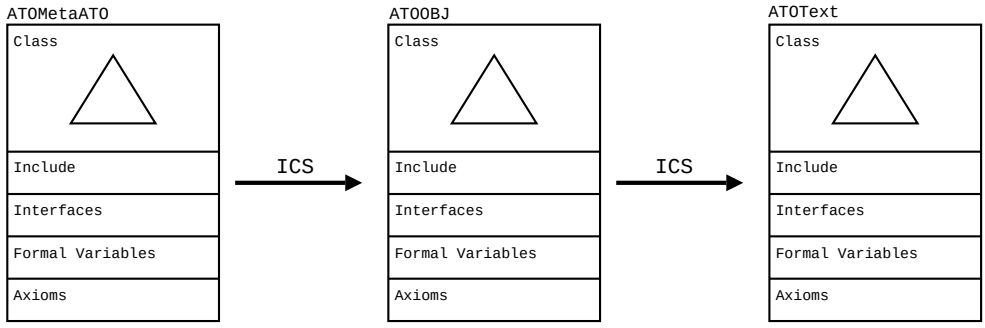


Figura 6.4: ATOs envolvidos na especificação formal do ProTool

sui operações que formalizam a tradução de um ATO algébrico para OBJ. Esta operação de tradução, chamada `phi`, toma como argumento um termo da classe MetaATO e retorna como resultado um termo da classe do `ATOOBJ`. Como cada ATO trata somente seu tipo de dado, esta operação `phi` é chamada via ICS, como na figura 6.4. Portanto, um termo da classe MetaATO representa a especificação de um ATO algébrico.

No `ATOOBJ` estão as operações que constroem termos da classe OBJ, que representam especificações de objetos OBJ. Como o objetivo é traduzir um ATO para código textual OBJ, são também definidas as operações que traduzem um termo da classe OBJ para a classe Text do `ATOText`. Neste passo de tradução são acrescentados todos os detalhes sintáticos necessários.

O `ATOText` por sua vez, contém operações para construção e manipulação de listas de string, que são necessárias para gerar o código textual do objeto OBJ.

Uma tradução, segundo a figura 6.4, é a seguinte: dado um ATO algébrico (descrito no MetaATO) este é traduzido para um termo da classe OBJ, que é então traduzido para a classe Text. O termo resultante da classe Text é o resultado final da tradução de um ATO algébrico para um objeto OBJ textual.

6.1 Hierarquia das classes do ProTool

Com intuito de aumentar o reuso de especificações, os ATOs `ATOMetaATO` e `ATOOBJ` foram modularizados em diversos ATOs, conforme mostrado na hierarquia de classes da figura 6.5.

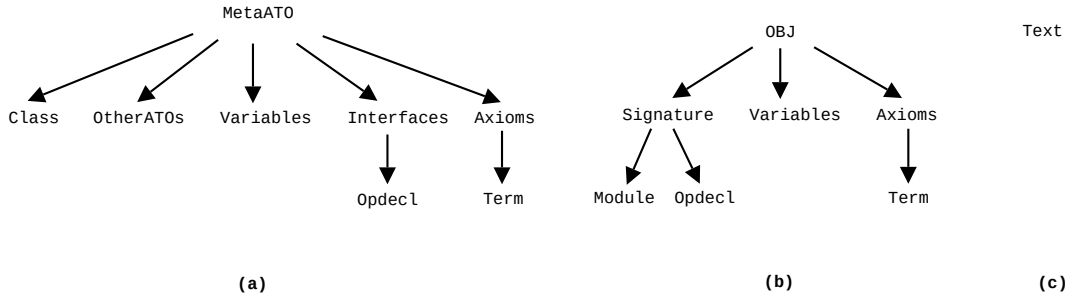


Figura 6.5: Hierarquia de ATOs do ATOMetaATO e ATOOBJ

Em **(a)** são mostrados os ATOs usados na especificação da classe do `ATOMetaATO`. Em **(b)**, os ATOs do `ATOOBJ` e em **(c)** aparece o `ATOText`, que não tem nenhuma dependência. Devido a similaridade entre as notações PROSOFT e OBJ, alguns ATOs são usados em ambas definições. Estes ATOs são: `ATOVariables`, `ATOOpdecl`, `ATOAxioms` e `ATOTerm`.

Conforme a figura 6.5, os ATOs que fazem parte do modelo ProTool são:

- **ATOMetaATO**: define a estrutura para construção de um ATO algébrico e sua respectiva tradução para OBJ;
- **ATOClass**: define a classe de um ATO;
- **ATOOtherATOs**: define as referências aos tipos primitivos e/ou ATOs externos (que não pertençam à classe) usados na construção de operações;
- **ATOVariables**: define a declaração de variáveis;
- **ATOInterfaces**: define uma lista de declarações de operações;
- **ATOOpdecl**: define a assinatura de uma operação algébrica;
- **ATOAxioms**: define a semântica das operações declaradas na especificação;
- **ATOTerm**: define um termo algébrico;
- **ATOOBJ**: define a estrutura de um objeto OBJ;
- **ATOSignature**: define a assinatura de um módulo OBJ;
- **ATOModule**: define os módulos usados no objeto OBJ;
- **ATOText**: define uma lista de strings.

As definições das operações que traduzem o MetaATO para OBJ são facilitadas pela criação de operações no `ATOMetaATO` que chamam, via ICS, as operações do `ATOOBJ`. A experiência evidencia que a utilização de muitas chamadas ICS, na definição de uma

operação, acabam dificultando o compreendimento devido ao tamanho exagerado dos termos no lado direito do axioma. Portanto, a estratégia adotada aumenta a legibiliadde da semântica da tradução, uma vez que os termos usados nos axiomas não envolvem diretamente chamadas ICS.

A maioria das operações de tradução estão localizadas na definição do `ATOMetaATO`. A tradução de classes para OBJ estão localizadas no `ATOClass`, devido a complexidade envolvida na tradução das mesmas.

6.2 ATOMetaATO

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **createMetaATO**: cria um termo da classe MetaATO somente com o nome do ATO;
- **addprimitive**: insere um tipo primitivo na classe;
- **addlist**: insere o tipo lista na classe;
- **addset**: insere o tipo conjunto na classe;
- **addmap**: insere o tipo mapeamento na classe;
- **addregister**: insere o tipo registro na classe;
- **adddu**: insere o tipo união disjunta na classe;
- **ins-extref**: insere referência a um ATO ou um primitivo;
- **getextref**: retorna um nome do primitivo ou ATO;
- **delextref**: apaga ATO ou primitivo do conjunto de referências;
- **isemptyextref**: verifica se o conjunto de referências é vazio;
- **addinterface**: insere uma declaração de operação;
- **createopint**: cria uma assinatura de operação, inicialmente com o nome da operação e seu sort valor;
- **includeoparity**: insere os sorts da aridade em uma assinatura de operação;
- **opname**: dada uma assinatura de operação, retorna o nome desta operação;
- **isemptyarity**: verifica se a aridade de uma operação é vazia;
- **valuesort**: retorna o sort valor de uma operação;
- **sortarity**: retorna o primeiro sort da aridade de uma operação;
- **delarity**: apaga um sort da aridade de uma operação;
- **isemptyints**: verifica se existem declarações de operações;
- **getint**: retorna a assinatura de operação;

- **delint**: apaga uma declaração de operação;
- **addvariable**: insere uma declaração de variável;
- **getvname**: retorna o nome de uma variável;
- **getvsort**: retorna o sort de uma variável;
- **delvar**: apaga uma declaração de variável;
- **isemptyvars**: verifica se existem declarações de variáveis;
- **addaxiom**: insere um axioma;
- **delaxiom**: apaga um axioma;
- **get-t-lhs**: retorna o lado esquerdo de um axioma;
- **get-t-rhs**: retorna o lado direito de um axioma;
- **isemptyaxs**: verifica se existem axiomas;
- **addop-in-term**: insere uma operação dentro de um termo;
- **addterm-in-term**: insere um subtermo em um termo;
- **getopname**: retorna o nome da operação de um termo;
- **getsubterms**: retorna os subtermos de um termo;
- **hassubterms**: verifica se um termo tem subtermos;
- **termstail**: retorna os subtermos de um termo, sem o primeiro subtermo;
- **termshead**: retorna o primeiro subtermo de um termo;
- **makeobj**: cria um objeto OBJ só com o nome;
- **impmod**: importa um módulo em OBJ;
- **ins-opdecl**: insere uma declaração de operação em OBJ;
- **makemod**: cria um módulo em OBJ;
- **opdecl**: cria uma declaração de operação somente com nome da operação e seu sort valor (em OBJ);
- **ins-arity**: insere um sort na aridade de uma operação (em OBJ);
- **ins-var**: insere uma declaração de variável em OBJ;
- **ins-equation**: insere uma equação em OBJ;
- **mk-op-in-term**: insere uma operação em um termo;
- **ins-term-in-term**: insere uma operação dentro de um termo (em OBJ);

- **ins-terms-in-term**: insere um subtermo em um termo (em OBJ);
- **t-to-terms**: converte um termo para uma lista de termos (em OBJ);
- **link-terms**: concatena duas listas de termos (em OBJ);
- **isrecclass**: verifica se uma classe é recursiva;
- **get-classname**: retorna o sort principal de uma classe;
- **phi-class**: traduz uma classe para OBJ;
- **phi-sorts**: analisa uma classe e declara sorts em OBJ;
- **phi-rclass**: traduz uma classe recursiva para OBJ;
- **phi**: traduz um MetaATO para OBJ;
- **phi-extref**: traduz as referências externas para OBJ;
- **phi-ints**: traduz as declarações de operações para OBJ;
- **fillarity**: operação auxiliar de phi-ints;
- **phi-vars**: traduz declaração de variáveis para OBJ;
- **phi-axioms**: traduz axiomas para equações OBJ;
- **phi-term**: traduz um termo para OBJ;
- **phi-subterms**: traduz subtermos para OBJ;
- **extract-text**: analisa um termo de OBJ e insere detalhes sintáticos para gerar código OBJ textual.

CLASS

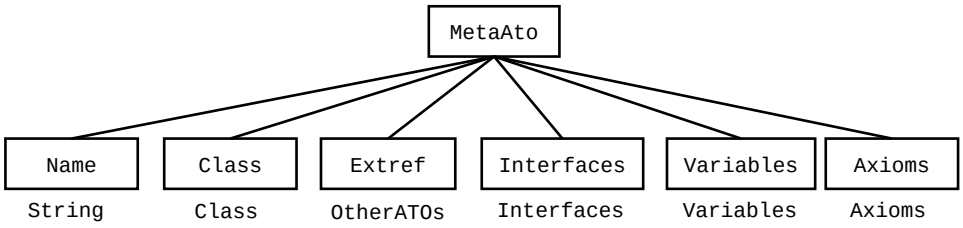


Figura 6.6: Classe que define MetaATO

INCLUDE

```
ATOText
```

INTERFACES

```
createMetaATO   :  String                  →  MetaATO
addprimitive    :  String String MetaATO   →  MetaATO
addlist         :  String MetaATO          →  MetaATO
addset          :  String MetaATO          →  MetaATO
addmap          :  String MetaATO          →  MetaATO
addregister     :  String Integer MetaATO  →  MetaATO
adddu           :  String Integer MetaATO  →  MetaATO
ins-extref      :  String MetaATO          →  MetaATO
getextref       :  MetaATO                 →  String
delextref       :  String MetaATO          →  MetaATO
isemptyextref   :  MetaATO                 →  Boolean
addinterface    :  Interface MetaATO       →  MetaATO
createopint     :  String String           →  Interface
includeoparity  :  String Interface        →  Interface
opname          :  Interface               →  String
isemptyarity    :  Interface               →  Boolean
valuesort       :  Interface               →  String
sortarity       :  Interface               →  String
delarity        :  String Interface        →  Interface
isemptyints     :  Interfaces              →  Boolean
getlint         :  Interfaces              →  Interface
delint          :  Interface Interfaces    →  Interfaces
addvariable     :  String String MetaATO   →  MetaATO
getvname        :  MetaATO                 →  String
getvsort        :  String MetaATO          →  String
delvar          :  String MetaATO          →  MetaATO
isemptyvars     :  MetaATO                 →  Boolean
addaxiom        :  Term Term MetaATO       →  MetaATO
delaxiom        :  Term MetaATO            →  MetaATO
get-t-lhs       :  MetaATO                 →  Term
```

```
get-t-rhs          :  Term MetaATO            →  Term
isemptyaxs         :  MetaATO                 →  Boolean
addop-in-term      :  String                  →  Term
addterm-in-term    :  Term Term               →  Term
getopname          :  Term                    →  String
getsubterms        :  Term                    →  Terms
hassubterms        :  Term                    →  Boolean
termstail          :  Terms                   →  Terms
termshead          :  Terms                   →  Term
makeobj            :  String                  →  OBJ
impmod             :  String Module OBJ       →  OBJ
ins-opdecl         :  Opdecl OBJ              →  Signature
makemod            :  String                  →  Module
opdecl             :  String String           →  Opdecl
ins-arity          :  String Opdecl           →  Opdecl
ins-var            :  String String OBJ       →  OBJ
ins-equation       :  Term Term OBJ           →  OBJ
mk-op-in-term      :  String                  →  Term
ins-term-in-term   :  Term Term               →  Term
ins-terms-in-term  :  String Terms            →  Term
t-to-terms         :  Term                    →  Terms
link-terms         :  Terms Terms             →  Terms
isrecclass         :  Class String            →  Boolean
get-classname      :  Class                   →  String
phi-class          :  Class                   →  Module
phi-sorts          :  Class OBJ String        →  OBJ
phi-rclass         :  Class Class OBJ         →  OBJ
phi                :  MetaATO                 →  OBJ
phi-extref         :  MetaATO MetaATO OBJ     →  OBJ
phi-ints           :  MetaATO Interfaces OBJ  →  OBJ
fillarity          :  Interface Opdecl        →  Opdecl
phi-vars           :  MetaATO MetaATO OBJ     →  OBJ
phi-axioms         :  MetaATO MetaATO OBJ     →  OBJ
phi-term           :  Term                    →  Term
phi-subterms       :  Terms                   →  Terms
extract-text       :  OBJ                     →  Text
```

FORMAL VARIABLES

```
name prim sort   :  String
n                :  Integer
extref           :  Extref
ints             :  Interfaces
int              :  Interface
axioms           :  Axioms
t1 t2            :  Term
obj              :  OBJ
op opdecl        :  Opdecl
oldsort newsort  :  String
oldop newop      :  String
```

```
 vars   :  Variables
 class  :  Class
 ma     :  MetaATO
 terms  :  Terms
```

AXIOMS

```
createMetaATO(name)
= reg-MetaATO(name,ICS(ATOClass,createclass,"un"),emptyset,
               emptylist,emptymap,emptymap)

addprimitive(name,prim,reg-MetaATO(_,class,_,_,_,_))
= reg-MetaATO(_,ICS(ATOClass,addPrimitive,name,prim,class),_,_,_,_)

addlist(name,reg-MetaATO(_,class,_,_,_,_))
= reg-MetaATO(_,ICS(ATOClass,addList,name,class),_,_,_,_)

addset(name,reg-MetaATO(_,class,_,_,_,_))
= reg-MetaATO(_,ICS(ATOClass,addSet,name,class),_,_,_,_)

addmap(name,reg-MetaATO(_,class,_,_,_,_))
= reg-MetaATO(_,ICS(ATOClass,addMap,name,class),_,_,_,_)

addregister(name,n,reg-MetaATO(_,class,_,_,_,_))
= reg-MetaATO(_,ICS(ATOClass,addRegister,name,n,class),_,_,_,_)

adddu(name,n,reg-MetaATO(_,class,_,_,_,_))
= reg-MetaATO(_,ICS(ATOClass,addDisjointUnion,name,n,class),_,_,_,_)

ins-extref(name,reg-MetaATO(_,_,extref,_,_,_))
= reg-MetaATO(_,_,ICS(ATOOtherATOs,addato,name,extref),_,_,_)

getextref(reg-MetaATO(_,_,extref,_,_,_))
= ICS(ATOOtherATOs,getatoname,extref)

delextref(name,reg-MetaATO(_,_,extref,_,_,_))
= reg-MetaATO(_,_,ICS(ATOOtherATOs,deleteato,name,extref),_,_,_)

isemptyextref(reg-MetaATO(_,_,extref,_,_,_))
= ICS(ATOOtherATOs,isemptyotheratos,extref)

addinterface(int,reg-MetaATO(_,_,_,ints,_,_))
= reg-MetaATO(_,_,_,ICS(ATOInterfaces,addinterface,int,ints),_,_)

createopint(name,sort) = ICS(ATOInterfaces,createint,name,sort)

includeoparity(sort,int) = ICS(ATOInterfaces,addarity,sort,int)

opname(int) = ICS(ATOInterfaces,getop,int)

isemptyarity(int) = ICS(ATOInterfaces,isemptyaritylist,int)
```

```
valuesort(int) = ICS(ATOInterfaces,getvsort,int)

sortarity(int) = ICS(ATOInterfaces,getsarity,int)

delarity(sort,int) = ICS(ATOInterfaces,deletesarity,sort,int)

isemptyints(ints) = ICS(ATOInterfaces,isemptyinterfaces,ints)

getint(ints) = ICS(ATOInterfaces,getinterface,ints)

delint(int,ints) = ICS(ATOInterfaces,deleteinterface,int,ints)

addvariable(name,sort,reg-MetaATO(_,_,_,_,vars,_))
= reg-MetaATO(_,_,_,_,ICS(ATOVariables,addvar,name,sort,vars),_)

getvname(reg-MetaATO(_,_,_,_,vars,_))
= ICS(ATOVariables,getvarname,vars)

getvsort(name,reg-MetaATO(_,_,_,_,vars,_))
= ICS(ATOVariables,getvarsort,name,vars)

delvar(name,reg-MetaATO(_,_,_,_,vars,_))
= reg-MetaATO(_,_,_,_,ICS(ATOVariables,deletevar,name,vars),_)

isemptyvars(reg-MetaATO(_,_,_,_,vars,_))
= ICS(ATOVariables,isemptyvariables,vars)

addaxiom(reg-MetaATO(_,_,_,_,_,axioms),t1,t2)
= reg-MetaATO(_,_,_,_,_,ICS(ATOAxioms,addaxiom,t1,t2,axioms))

delaxiom(t,reg-MetaATO(_,_,_,_,_,axioms))
= reg-MetaATO(_,_,_,_,_,ICS(ATOAxioms,deleteaxiom,t,axioms))

get-t-lhs(reg-MetaATO(_,_,_,_,_,axioms))
= ICS(ATOAxioms,get-term-lhs,axioms)

get-t-rhs(t,reg-MetaATO(_,_,_,_,_,axioms))
= ICS(ATOAxioms,get-term-rhs,t,axioms)

isemptyaxs(reg-MetaATO(_,_,_,_,_,axioms))
= ICS(ATOAxioms,isemptyaxioms,axioms)

addop-in-term(name) = ICS(ATOTerm,addop,name)

addterm-in-term(t1,t2) = ICS(ATOTerm,addterm,t1,t2)

getopname(t) = ICS(ATOTerm,getopname,t)

getsubterms(t) = ICS(ATOTerm,getterms,t)
```

```
hassubterms(t) = ICS(ATOTerm,hasparameters,t)

termstail(terms) = ICS(ATOTerm,parameterstail,terms)

termshead(terms) = ICS(ATOTerm,parametershead,terms)

makeobj(name) = ICS(ATOOBJ,createOBJ,name)

impmod(mode,mod,obj) = ICS(ATOOBJ,addimp,mode,mod,obj)

ins-opdecl(op,obj) = ICS(ATOOBJ,insertop,op,obj)

makemod(name) = ICS(ATOOBJ,createmodule,name)

opdecl(name,sort) = ICS(ATOOBJ,declareop,name,sort)

ins-arity(sort,opdecl) = ICS(ATOOBJ,addarity,sort,opdecl)

ins-var(name,sort,obj) = ICS(ATOOBJ,addvar,name,sort,obj)

ins-equation(t1,t2,obj) = ICS(ATOOBJ,addeq,t1,t2,obj)

mk-op-in-term(name) = ICS(ATOOBJ,add-op-in-term,name)

ins-term-in-term(t1,t2) = ICS(ATOOBJ,addterm-in-term,t1,t2)

ins-terms-in-term(name,terms) = ICS(ATOOBJ,add-terms,name,terms)

t-to-terms(t) = ICS(ATOTerm,termtoterms,t)

link-terms(t1,t2) = ICS(ATOTerm,concatterms,t,tt)

isrecclass(class,name) = ICS(ATOClass,isrec,class,name)

get-classname(class) = ICS(ATOClass,getname,class)

phi-class(class) = ICS(ATOClass,phic,class)

phi-sorts(class,obj,name) = ICS(ATOClass,sorts-cr,class,obj,name)

phi-rclass(class,class,obj) = ICS(ATOClass,phicr,class,class,obj)


phi(ma)
=
if isrecclass(select-Class(ma),get-classname(select-Class(ma)))
 then phi-extref(ma,ma,phi-rclass(select-Class(ma),select-Class(ma),
          phi-sorts(select-Class(ma),makeobj("ATO"++select-Name(ma)),
          get-classname(select-Class(ma)))))
 else phi-extref(ma,ma,impmod("pr",phi-class(select-Class(ma)),
          makeobj("ATO"++select-Name(ma))))
```

```
phi-extref(ma,reg-MetaATO(_,_,extref,_,_,_),obj)
=
if isemptyextref(extref)
 then phi-ints(ma,select-Interfaces(ma),obj)
 else
  if getextref(extref) == "Boolean"
   then phi-extref(ma,reg-MetaATO(_,_,delextref("Boolean",extref),
               _,_,_),impmod("pr",makemod("BOOL"),obj))
   else
    if getextref(extref) == "Integer"
     then phi-extref(ma,reg-MetaATO(_,_,delextref("Integer",extref),
                _,_,_),impmod("pr",makemod("INT"),obj))
     else phi-extref(ma,reg-MetaATO(_,_,delextref(getextref(extref)
              ,extref),_,_,_),impmod("pr",makemod(ucase(getextref(
              extref))),obj))


phi-ints(ma,ints,obj)
=
if isemptyints(ints)
 then phi-vars(ma,ma,obj)
 else
  if valuesort(getint(ints)) == "Boolean"
   then phi-ints(ma,delint(getint(ints),ints),ins-opdecl(fillarity(
          getint(ints),opdecl(opname(getint(ints)),"Bool"),obj)
   else
    if valuesort(getint(ints)) == "Integer"
     then phi-ints(ma,delint(getint(ints),ints),ins-opdecl(
            fillarity(getint(ints),opdecl(opname(getint(ints)),
            "Int"),obj)
     else phi-ints(ma,delint(getint(ints),ints),ins-opdecl(
             fillarity(getint(ints),opdecl(opname(getint(ints)),
             valuesort(getint(ints))),obj)

fillarity(int,opdecl)
=
if isemptyarity(int)
 then opdecl
 else
  if sortarity(int) == "Boolean"
   then fillarity(delarity("Boolean",int),ins-arity("Bool",opdecl))
   else
    if sortarity(int) == "Integer"
     then fillarity(delarity("Integer",int),ins-arity("Int",opdecl))
     else fillarity(delarity(sortarity(int),int),ins-arity(sortarity
            (int),opdecl))


phi-vars(ma,reg-MetaATO(_,_,_,_,vars,_),obj)
```

```
=
if isemptyvars(vars)
 then phi-axioms(ma,ma,obj)
 else phi-vars(ma,reg-MetaATO(_,_,_,_,delvar(getvname(reg-MetaATO(
        _,_,_,_,vars,_)),reg-MetaATO(_,_,_,_,vars,_)),_),
        ins-var(getvname(reg-MetaATO(_,_,_,_,vars,_)),
        getvsort(getvname(reg-MetaATO(_,_,_,_,vars,_)),
        reg-MetaATO(_,_,_,_,vars,_)),obj))


phi-axioms(ma,reg-MetaATO(_,_,_,_,_,axioms),obj)
=
if isemptyaxs(axioms)
 then obj
 else phi-axioms(ma,delaxiom(t,reg-MetaATO(_,_,_,_,_,axioms)),
        ins-equation(phi-term(get-t-lhs(reg-MetaATO(_,_,_,_,_,
        axioms)),phi-term(get-t-rhs(get-t-lhs(reg-MetaATO(_,_,
        _,_,_,axioms)),reg-MetaATO(_,_,_,_,_,axioms))),obj))


phi-term(t)
=
if isin(ucase(getopname(t)),"ICS")
 then ins-terms-in-term(termshead(getsubterms(t)),phi-subterms(
        termstail(getsubterms(t))))
 else
   if hassubterms(t)
    then ins-terms-in-term(getopname(t),phi-subterms(
             getsubterms(t)))
    else mk-op-in-term(getopname(t))

phi-subterms(terms)
=
if terms == emptylist
 then terms
 else link-terms(t-to-terms(phi-term(termshead(terms))),
         phi-subterms(termstail(terms)))


extract-text(obj) = ICS(ATOOBJ,phi-txt,obj)
```

6.3 ATOClass

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **createclass**: cria uma classe somente com o nome;
- **addPrimitive**: insere um tipo primitivo na classe;
- **addList**: insere o tipo lista na classe;
- **addSet**: insere o tipo conjunto na classe;

- **addMap**: insere o tipo mapeamento na classe;
- **addRegister**: insere o tipo registro na classe;
- **addDisjointUnion**: insere o tipo união disjunta na classe;
- **createMembers**: cria os domínios do tipo registro;
- **createAlternatives**: cria os domínios do tipo união disjunta;
- **path, teta, lambda**: operações auxiliares de caminhamento para criar uma classe;
- **is-closed**: verifica se uma classe é fechada (não tem nodos folha anônimos);
- **isrec**: verifica se uma classe é recursiva;
- **mk-mod**: cria um módulo (em OBJ);
- **rnm-op**: renomeia uma operação (em OBJ);
- **rnmsort**: renomeia um sort (em OBJ);
- **inst-mod**: instancia um módulo (em OBJ);
- **decl-sort**: declara um sort (em OBJ);
- **imp-mod**: importa um módulo (em OBJ);
- **count**: conta o número de domínios de um registro ou união disjunta;
- **inttostr**: converte um inteiro em string (operação limitada de 1 até 10);
- **ren-reg-ops**: renomeia operações do tipo registro em OBJ;
- **ren-du-ops**: renomeia operações do tipo união disjuntiva em OBJ;
- **phic**: traduz uma classe sem recursão em um módulo OBJ;
- **sorts-cr**: analisa uma classe recursiva para declarar, em OBJ, os sorts desta classe e importar os tipos primitivos usados na classe;
- **isPrimitive**: verifica se uma classe é somente composta por um tipo primitivo;
- **get-ref**: retorna o tipo primitivo de uma classe;
- **getdomain**: retorna o sort do domínio de uma classe mapeamento;
- **getrange**: retorna o sort da imagem de uma classe mapeamento;
- **isList**: verifica se uma classe é do tipo lista;
- **isSet**: verifica se uma classe é do tipo conjunto;
- **isMap**: verifica se uma classe é do tipo mapeamento;
- **isRegister**: verifica se uma classe é do tipo registro;

- **isDisjointUnion**: verifica se uma classe é do tipo união disjunta;
- **getname**: retorna o nome do nodo raiz de uma classe;
- **getsort**: retorna o sort de uma classe primitiva;
- **reg-list**: operação auxiliar para renomeação de operações de registro em OBJ;
- **du-list**: operação auxiliar para renomeação de operações de união disjunta em OBJ;
- **phicr**: traduz uma classe recursiva para OBJ.

CLASS

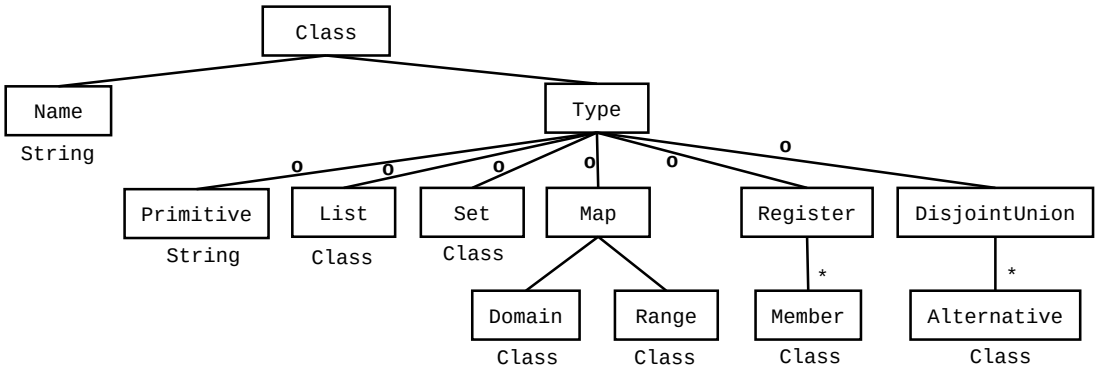


Figura 6.7: Classe que define Class

INCLUDE

```
Boolean Integer ATOOBJ
```

INTERFACES

```
createclass         :  String                →  Class
addPrimitive        :  String String Class   →  Class
addList             :  String Class          →  Class
addSet              :  String Class          →  Class
addMap              :  String Class          →  Class
addRegister         :  String Integer Class  →  Class
addDisjointUnion    :  String Integer Class  →  Class
createMembers       :  Integer               →  Register
createAlternatives  :  Integer               →  DisjointUnion
path                :  Class Class           →  Class
teta                :  Type Class            →  Type
lambda              :  DisjointUnion Class   →  DisjointUnion
lambda              :  Register Class        →  Register
is-closed           :  Class                 →  Boolean
isrec               :  Class String          →  Boolean
isrec               :  Register String       →  Boolean
```

```
isrec           :  DisjointUnion String                  →  Boolean
phic            :  Class                                 →  Module
count           :  Register                              →  Integer
count           :  DisjointUnion                         →  Integer
inttostr        :  Integer                               →  String
ren-reg-ops     :  Module Register Integer               →  Module
ren-du-ops      :  Module DisjointUnion Integer String   →  Module
mk-mod          :  String                                →  Module
rnm-op          :  String String Module                  →  Module
rnmsort         :  String String Module                  →  Module
inst-mod        :  Module Module                         →  Module
decl-sort       :  String OBJ                            →  OBJ
imp-mod         :  String Module OBJ                     →  OBJ
rnm-du-ops      :  String String String Module           →  Module
sorts-cr        :  Class OBJ String                      →  OBJ
sorts-cr        :  Register OBJ String                   →  OBJ
sorts-cr        :  DisjointUnion OBJ String              →  OBJ
isPrimitive     :  Class                                 →  Boolean
get-ref         :  Class                                 →  String
phicr           :  Class Class OBJ                       →  OBJ
getdomain       :  Class                                 →  String
getrange        :  Class                                 →  String
isList          :  Class                                 →  Boolean
isSet           :  Class                                 →  Boolean
isMap           :  Class                                 →  Boolean
isRegister      :  Class                                 →  Boolean
isDisjointUnion :  Class                                 →  Boolean
getname         :  Class                                 →  String
getsort         :  Class                                 →  String
reg-list        :  Register Module                       →  Module
du-list         :  DisjointUnion Module                  →  Module
aux-reg         :  Register Class OBJ                    →  OBJ
aux-du          :  DisjointUnion Class OBJ               →  OBJ
```

FORMAL VARIABLES

```
    newclass class m a range domain  :  Class
    list set oldclass                :  Class
    type type2 type3                 :  Type
    name name2 name3 cname           :  String
    prim mode namelist               :  String
    n                                :  Integer
    mlist                            :  Register
    alist                            :  DisjointUnion
    mod mod1                         :  Module
    obj                              :  OBJ
```

AXIOMS

```
createclass(name) = reg-Class(name,Type-Primitive("ut")

addPrimitive(name,prim,oldclass)
= path(oldclass,reg-Class(name,Type-Primitive(prim)))
```

```
addList(name,oldclass)
= path(oldclass,reg-Class(name,Type-List(reg-Class("un",
       Type-Primitive("ut")))))

addSet(name,oldclass)
= path(oldclass,reg-Class(name,Type-Set(reg-Class("un",
       Type-Primitive("ut")))))

addMap(name,oldclass)
= path(oldclass,reg-Class(name,Type-Map(reg-Map(reg-Class(
      "un",Type-Primitive("ut")),reg-Class("un",
      Type-Primitive("ut"))))))

addRegister(name,n,oldclass)
= path(oldclass,reg-Class(name,Type-Register(createMembers(n))))

addDisjointUnion(name,n,oldclass)
= path(oldclass,reg-Class(name,Type-DisjointUnion(
      createAlternatives(n))))

createMembers(n)
=
if n > 0
 then concat(cons(reg-Class("un",Type-Primitive("ut")),emptylist),
         createMembers(n-1))
 else emptylist

createAlternatives(n)
=
if n > 0
 then concat(cons(reg-Class("un",Type-Primitive("ut")),emptylist),
         createAlternatives(n-1))
 else emptylist

path(reg-Class(name,type),newclass)
=
if (name == "un")
 then newclass
 else reg-Class(name,teta(type,newclass))

teta(Type-List(reg-Class(name,type)),reg-Class(name2,type2))
=
if (name == "un")
 then Type-List(reg-Class(name2,type2))
 else Type-List(reg-Class(name,teta(type,reg-Class(name2,type2))))

teta(Type-Set(reg-Class(name,type)),reg-Class(name2,type2))
=
if (name == "un")
 then Type-Set(reg-Class(name2,type2))
```

```
 else Type-Set(reg-Class(name,teta(type,reg-Class(name2,type2))))

teta(Type-DisjointUnion(reg-Class(cons(reg-Class(name,type),alist))),
    reg-Class(name2,type2))
=
if name == "un"
 then Type-DisjointUnion(reg-Class(cons(reg-Class(name2,type2),
           alist)))
 else Type-DisjointUnion(reg-Class(lambda(cons(reg-Class(name,
           type),alist),reg-Class(name2,type2))))

lamda(cons(reg-Class(name,type),alist),newclass)
=
if is-closed(reg-Class(name,type))
 then concat(cons(reg-Class(name,type),emptylist),lambda(alist,
       newclass))
 else cons(reg-Class(name,teta(type,newclass)),alist)

teta(Type-Register(reg-Class(cons(reg-Class(name,type),mlist))),
   reg-Class(name2,type2))
=
if name == "un"
 then Type-Register(reg-Class(cons(reg-Class(name2,type2),mlist)))
 else Type-Register(reg-Class(lambda(cons(reg-Class(name,type),
         mlist),reg-Class(name2,type2))))

lamda(cons(reg-Class(name,type),mlist),newclass)
=
if is-closed(reg-Class(name,type))
 then concat(cons(reg-Class(name,type),emptylist),lambda(mlist,
         newclass))
 else cons(reg-Class(name,teta(type,newclass)),mlist)

teta(Type-Map(reg-Map(reg-Class(name,type),reg-Class(name2,type2))),
    reg-Class(name3,type3))
=
if name == "un"
 then Type-Map(reg-Map(reg-Class(name3,type3),reg-Class(name2,type2)))
 else
  if not(is-closed(reg-Class(name,type)))
   then Type-Map(reg-Map(reg-Class(name,teta(type,reg-Class(name3,
             type3))),reg-Class(name2,type2)))
   else
    if (name2 == "un")
     then Type-Map(reg-Map(reg-Class(name,type),reg-Class(name3,
             type3)))
     else Type-Map(reg-Map(reg-Class(name,type),reg-Class(name2,
             teta(type2,reg-Class(name3,type3)))))


is-closed(reg-Class(_,Type-Primitive(prim)))
```

```
= if prim == "ut" then false else true

is-closed(reg-Class(_,Type-List(list))) = is-closed(list)

is-closed(reg-Class(_,Type-Set(set))) = is-closed(set)

is-closed(reg-Class(_,Type-Map(reg-Map(domain,range))))
= is-closed(domain) and is-closed(range)

is-closed(reg-Class(_,Type-Register(emptylist))) = false

is-closed(reg-Class(_,Type-Register(cons(m,mlist))))
= is-closed(m) and is-closed(reg-Class(_,Type-Register(mlist)))

is-closed(reg-Class(_,Type-DisjointUnion(emptylist))) = false

is-closed(reg-Class(_,Type-DisjointUnion(cons(a,alist))))
= is-closed(a) and is-closed(reg-Class(_,Type-DisjointUnion(alist)))

isrec(reg-Class(_,Type-List(list)),name) = isrec(list,name)

isrec(reg-Class(_,Type-Set(set)),name) = isrec(set,name)

isrec(reg-Class(_,Type-Map(reg-Map(domain,range))),name)
= isrec(domain,name) or isrec(range,name)

isrec(reg-Class(_,Type-Primitive(prim)),name)
= if prim == name then true else false

isrec(cons(m,mlist),name)
=
if mlist == emptylist
 then isrec(m,name)
 else isrec(m,name) or isrec(mlist,name)

isrec(cons(a,alist),name)
=
if alist == emptylist
 then isrec(a,name)
 else isrec(a,name) or isrec(alist,name)

isrec(reg-Class(_,Type-Register(mlist)),name)
= isrec(mlist,name)

isrec(reg-Class(_,Type-DisjointUnion(alist)),name)
= isrec(alist,name)

mk-mod(name) = ICS(ATOOBJ,createmodule,name)

rnm-op(oldop,newop,mod) = ICS(ATOOBJ,renameop,oldop,newop,mod)
```

```
rnmsort(oldsort,newsort,mod)
= ICS(ATOOBJ,renamesort,oldsort,newsort,mod)

inst-mod(mod,mod1) = ICS(ATOOBJ,instmodule,mod,mod1)

decl-sort(name,obj) = ICS(ATOOBJ,declaresort,name,obj)

imp-mod(mode,mod,obj) = ICS(ATOOBJ,addimp,mode,mod,obj)

count(emptylist) = 0
count(cons(m,members)) = 1 + count(members)
count(cons(a,alternatives)) = 1 + count(alternatives)

inttostr((1).Integer) = (1).String
inttostr((2).Integer) = (2).String
inttostr((3).Integer) = (3).String
inttostr((4).Integer) = (4).String
inttostr((5).Integer) = (5).String
inttostr((6).Integer) = (6).String
inttostr((7).Integer) = (7).String
inttostr((8).Integer) = (8).String
inttostr((9).Integer) = (9).String
inttostr((10).Integer) = (10).String

ren-reg-ops(mod,cons(m,mlist),n)
=
if mlist =/= emptylist
 then ren-reg-ops(rnm-op("select"++inttostr(n),"select-"++
          getname(m),mod),mlist,n - 1)
 else rnm-op("select"++inttostr(n),"select-"++getname(m),mod)

ren-du-ops(mod,cons(a,alist),n,name)
=
if alist =/= emptylist
 then ren-du-ops(rnm-op("get"++ ++inttostr(n),"get\-"++
         getname(a),rnm-op("is"++inttostr(n),"is\-"++getname(a),
         rnm-op("apply"++inttostr(n),name++getname(a),mod))),
         alist,n - 1,name)
 else rnm-op("get"++ ++inttostr(n),"get-"++getname(a),rnm-op
         ("is"++inttostr(n),"is\-"++getname(a),rnm-op("apply"++
         inttostr(n),name++getname(a),mod)))


rnm-du-ops(name,prim,namelist,mod)
=
if not(isin("\#",namelist)) and length(namelist) > 1
 then mod
 else rnm-du-ops(name,prim,extract(locate("\#",namelist) + 1,
        length(namelist),namelist),rnm-op(prim++"\-"++extract(1,
        locate("\#",namelist) - 1,namelist),name++"-++extract(1,
        locate("\#",namelist) - 1,namelist),mod))
```

```
phic(reg-Class(name,Type-List(list)))
= inst-mod(phic(list),rnmsort("List",name,mk-mod("LIST")))

phic(reg-Class(name,Type-Set(set)))
= inst-mod(phic(set),rnmsort("Set",name,mk-mod("SET")))

phic(reg-Class(name,Type-Map(reg-Map(domain,range))))
= inst-mod(phic(range),inst-mod(phic(domain),sortrnm("Map",
     name,mk-mod("MAP"))))

phic(reg-Class(name,Type-Register(cons(m,mlist))))
= inst-mod(phic(reg-Class(name,Type-Register(mlist))),
     inst-mod(phic(m),ren-reg-ops(rnm-op("reg","reg\-"++name,
     rnmsort("Register"++inttostr(count(cons(m,mlist))),name,
     mk-mod("REGISTER"++inttostr(count(cons(m,mlist)))))))),
     cons(m,mlist),count(cons(m,mlist)))))

phic(reg-Class(name,Type-DisjointUnion(cons(a,alist))))
= inst-mod(phic(reg-Class(name,Type-DisjointUnion(alist))),
     inst-mod(phic(a),ren-du-ops(rnmsort("DisjointUnion"++
     inttostr(count(cons(a,alist))),name,mk-mod("DISJOINTUNION"++
     inttostr(count(cons(a,alist))))),cons(a,alist),count(cons(a,
     alist)),name)))

phic(reg-Class(name,Type-Primitive(prim)))
=
if prim == "Boolean"
 then mk-mod("BOOL")
 else
  if prim == "Integer"
   then mk-mod("INT")
   else
    if prim == "String" or prim == "Date" or prim == "Time"
     then mk-mod(ucase(prim))
     else rnmsort(prim,name,mk-mod("ATO"++prim))



sorts-cr(reg-Class(name,Type-List(list)),obj,cname)
= sorts-cr(list,decl-sort(name,obj),cname)

sorts-cr(reg-Class(name,Type-Set(set)),obj,cname)
= sorts-cr(set,decl-sort(name,obj),cname)

sorts-cr(reg-Class(name,Type-Map(reg-Map(domain,range)),obj,cname)
= sorts-cr(range,sorts-cr(domain,decl-sort(name,obj),cname),cname)

sorts-cr(reg-Class(name,Type-Register(mlist)),obj,cname)
```

```
= sorts-cr(mlist,decl-sort(name,obj),cname)

sorts-cr(cons(m,mlist),obj,cname)
= if mlist == emptylist
then sorts-cr(m,obj,cname)
else sorts-cr(m,sorts-cr(mlist,obj,cname),cname)

sorts-cr(reg-Class(name,Type-DisjointUnion(alist)),obj,cname)
= sorts-cr(alist,decl-sort(name,obj),cname)

sorts-cr(cons(a,alist),obj,cname)
= if alist == emptylist
then sorts-cr(a,obj,cname)
else sorts-cr(a,sorts-cr(alist,obj,cname),cname)


sorts-cr(reg-Class(name,Type-Primitive(prim)),obj,cname)
=
if prim == "Boolean"
 then imp-mod("pr",mk-mod("BOOL"),obj)
 else
  if prim == "Integer"
   then imp-mod("pr",mk-mod("INT"),obj)
   else
    if prim == "String" or prim == "Date" or prim == "Time"
     then imp-mod("pr",mk-mod(ucase(prim)),obj)
     else
      if prim == cname
        then obj
        else imp-mod("pr",rnmsort(prim,name,mk-mod("ATO"++prim)),obj)



isPrimitive(reg-Class(name,type)) = is-Primitive(type)

get-ref(reg-Class(name,Type-Primitive(prim))) = prim


getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
   Type-Primitive(prim)),_))))
=
if prim == "Boolean"
 then "Bool"
 else
  if prim == "Integer"
   then "Int"
   else
    if prim == "String" or prim == "Date" or prim == "Time"
     then prim
     else name2
```

```
getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
 Type-List(list)),_)))) = name2

getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
 Type-Set(set)),_)))) = name2

getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
 Type-Map(map)),_)))) = name2

getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
 Type-Register(mlist)),_)))) = name2

getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
 Type-DisjointUnion(alist)),_)))) = name2


getrange(reg-Class(_,Type-Map(reg-Map(_,reg-Class(name2,
 Type-Primitive(prim))))))
=
if prim == "Boolean"
 then "Bool"
 else
  if prim == "Integer"
   then "Int"
   else
    if prim == "String" or prim == "Date" or prim == "Time"
     then prim
     else name2


getrange(reg-Class(name,Type-Map(reg-Map(_,reg-Class(name2,
 Type-List(list)))))) = name2

getrange(reg-Class(name,Type-Map(reg-Map(_,reg-Class(name2,
 Type-Set(set)))))) = name2

getrange(reg-Class(name,Type-Map(reg-Map(_,reg-Class(name2,
 Type-Map(map)))))) = name2

getrange(reg-Class(name,Type-Map(reg-Map(_,reg-Class(name2,
 Type-Register(mlist)))))) = name2

getrange(reg-Class(name,Type-Map(reg-Map(_,reg-Class(name2,
 Type-DisjointUnion(alist)))))) = name2


isList(reg-Class(_,type))
= if is-List(type) == true then true else false

isSet(reg-Class(_,type))
= if is-Set(type) == true then true else false
```

```
isMap(reg-Class(_,type))
= if is-Map(type) == true then true else false

isRegister(reg-Class(_,type))
= if is-Register(type) == true then true else false

isDisjointUnion(reg-Class(_,type))
= if is-DisjointUnion(type) == true then true else false


getname(class) = select-Name(class)

getsort(reg-Class(name,Type-Primitive(prim)))
=
if prim == "Boolean"
 then "Bool"
 else
  if prim == "Integer"
   then "Int"
   else
    if prim == "String" or prim == "Date" or prim == "Time"
     then prim
     else name


getsort(reg-Class(name,Type-List(list))) = name

getsort(reg-Class(name,Type-Set(set))) = name

getsort(reg-Class(name,Type-Map(map))) = name

getsort(reg-Class(name,Type-Register(mlist))) = name

getsort(reg-Class(name,Type-DisjointUnion(alist))) = name


reg-list(cons(m,mlist),mod)
=
if mlist == emptylist
 then mod
 else reg-list(mlist,rnm-op(getname(m),getsort(m),mod))


du-list(cons(a,alist),mod)
=
if alist == emptylist
 then mod
 else du-list(alist,rnm-op(getname(a),getsort(a),mod))
```

```
phicr(reg-Class(name,Type-List(list)),class,obj)
=
if not(isPrimitive(list))
 then phicr(list,class,imp-mod("create\-spec",rnmsort(name,
        getname(list),mk-mod("LIST")),obj))
 else
  if get-ref(list) == getname(class)
   then imp-mod("create\-spec",rnmsort(name,get-ref(list),
          mk-mod("LIST")),obj)
   else
    if get-ref(list) == "Boolean"
     then imp-mod("create\-spec",rnmsort(name,"Bool",mk-mod(
             "LIST")),obj)
     else
      if get-ref(list) == "Integer"
       then imp-mod("create\-spec",rnmsort(name,"Int",mk-mod(
             "LIST")),obj)
       else
        if get-ref(list) == "String" or get-ref(list) == "Date"
              or get-ref(list) == "Time"
         then imp-mod("create\-spec",rnmsort(name,get-ref(list),
                mk-mod("LIST")),obj)
         else imp-mod("create\-spec",rnmsort(name,getname(list),
                mk-mod("LIST")),obj)



phicr(reg-Class(name,Type-Set(set)),class,obj)
=
if not(isPrimitive(set))
 then phicr(set,class,imp-mod("create\-spec",rnmsort(name,getname(
         set),mk-mod("SET")),obj))
 else
  if get-ref(set) == getname(class)
   then imp-mod("create\-spec",rnmsort(name,get-ref(set),mk-mod(
         "SET")),obj)
   else
    if get-ref(set) == "Boolean"
     then imp-mod("create\-spec",rnmsort(name,"Bool",mk-mod("SET"))
            ,obj)
     else
      if get-ref(set) == "Integer"
       then imp-mod("create\-spec",rnmsort(name,"Int",mk-mod("SET")),
              obj)
       else
        if get-ref(set) == "String" or get-ref(set) == "Date" or
              get-ref(set) == "Time"
         then imp-mod("create\-spec",rnmsort(name,get-ref(set),
                mk-mod("SET")),obj)
         else imp-mod("create\-spec",rnmsort(name,getname(set),
                mk-mod("SET")),obj)
```

```
phicr(reg-Class(name,Type-Map(reg-Map(domain,range))),class,obj)
=
if not(isPrimitive(domain)) and not(isPrimitive(range))
 then phicr(domain,class,phicr(range,class,imp-mod("create\-spec",
         rnm-op(getname(range),"\empty",rnmsort(name,getname(domain),
         mk-mod("MAP"))),obj)))
 else
  if not(isPrimitive(domain)) and isPrimitive(range)
   then
    if get-ref(range) == getname(class)
     then phicr(domain,class,imp-mod("create\-spec",
             rnm-op(get-ref(range),"\empty",rnmsort(name,getname(
             domain),mk-mod("MAP"))),obj))
     else
      if get-ref(range) == "Boolean"
       then phicr(domain,class,imp-mod("create\-spec",
              rnm-op("Bool","\empty",rnmsort(name,getname(domain),
              mk-mod("MAP"))),obj))
       else
        if get-ref(range) == "Integer"
         then phicr(domain,class,imp-mod("create\-spec",
               rnm-op("Int","\empty",rnmsort(name,getname(domain),
               mk-mod("MAP"))),obj))
         else
          if get-ref(range) == "String" or get-ref(range) == "Date"
                or get-ref(range) == "Time"
           then phicr(domain,class,imp-mod("create\-spec",
                  rnm-op(get-ref(range),"\empty",rnmsort(name,getname(
                  domain),mk-mod("MAP"))),obj))
           else phicr(domain,class,imp-mod("create\-spec",rnm-op(
                  getname(range),"\empty",rnmsort(name,getname(domain),
                  mk-mod("MAP"))),obj))

   else
    if isPrimitive(domain) and not(isPrimitive(range))
     then
      if get-ref(domain) == getname(class)
       then phicr(range,class,imp-mod("create\-spec",
             rnm-op(getname(range),"\empty",rnmsort(name,get-ref(domain),
             mk-mod("MAP"))),obj))
       else
        if get-ref(domain) == "Boolean"
         then phicr(range,class,imp-mod("create\-spec",
               rnm-op(getname(range),"\empty",rnmsort(name,"Bool",mk-mod(
               "MAP"))),obj))
         else
          if get-ref(domain) == "Integer"
           then phicr(range,class,imp-mod("create\-spec",
              rnm-op(getname(range),"\empty",rnmsort(name,"Int",mk-mod(
```

```
              "MAP"))),obj))
            else
             if get-ref(domain) == "String" or get-ref(domain) == "Date"
                    or get-ref(domain) == "Time"
              then phicr(range,class,imp-mod("create\-spec",rnm-op(
                      getname(range),"\empty",rnmsort(name,get-ref(
                      domain),mk-mod("MAP"))),obj))
              else phicr(range,class,imp-mod("create\-spec",rnm-op(
                      getname(range),"\empty",rnmsort(name,getname(
                      domain),mk-mod("MAP"))),obj))

     else imp-mod("create\-spec",rnm-op(get-ref(range),"\empty",
            rnmsort(name,get-ref(domain),mk-mod("MAP"))),obj))



phicr(reg-Class(name,Type-Register(mlist)),class,obj)
= aux-reg(mlist,class,imp-mod("create\-spec",reg-list(mlist,rnmsort(
      name,"\empty",mk-mod("REGISTER"))),obj))

aux-reg(cons(m,mlist),class,obj)
=
if mlist == emptylist
 then
   if isPrimitive(m)
    then obj
    else phicr(m,class,obj)
 else
   if isPrimitive(m)
    then obj
    else phicr(m,class,aux-reg(mlist,class,obj))

phicr(reg-Class(name,Type-DisjointUnion(alist)),class,obj)
= aux-du(alist,class,imp-mod("create\-spec",du-list(alist,rnmsort(name,
   "\empty",mk-mod("DISJOINTUNION"))),obj))

aux-du(cons(a,alist),class,obj)
=
if alist == emptylist
 then
   if isPrimitive(a)
    then obj
    else phicr(a,class,obj)
 else
   if isPrimitive(a)
    then obj
    else phicr(a,class,aux-du(alist,class,obj))
```

6.4 ATOOtherATOs

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addato**: insere uma referência a um ATO ou um primitivo;
- **getatoname**: retorna um nome do primitivo ou ATO;
- **deleteato**: apaga um ATO ou primitivo do conjunto de referências;
- **isemptyotheratos**: verifica se o conjunto de referências é vazio.

CLASS

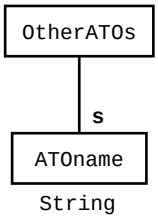


Figura 6.8: Classe OtherATOs

INCLUDE

```
Boolean
```

INTERFACES

```
addato           :  String OtherATOs  →  OtherATOs
getatoname       :  OtherATOs         →  String
deletaato        :  String OtherATOs  →  OtherATOs
isemptyotheratos :  OtherATOs         →  Boolean
```

FORMAL VARIABLES

```
name name2  :  String
oatos       :  OtherATOs
```

AXIOMS

```
addato(name,otaos) = cons(name,oatos)

getatoname(add(name,oatos)) = name

deleteato(name,oatos) = delete(name,oatos)

isemptyotheratos(oatos) = isempty(oatos)
```

6.5 ATOVariables

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addvar**: insere uma declaração de variável;
- **getvarname**: dada uma declaração de variável, retorna o nome da variável;
- **getvarsort**: dada uma declaração de variável, retorna o sort da variável;
- **deletevar**: apaga uma declaração de variável;
- **isemptyvariables**: verifica se existem declarações de variáveis.

CLASS

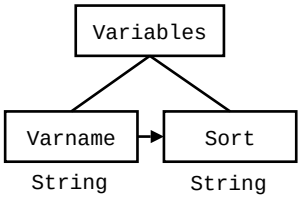


Figura 6.9: Classe que define Variables

INCLUDE

```
Boolean
```

INTERFACES

```
addvar           :  String String Variables  →  Variables
getvarname       :  Variables                →  String
getvarsort       :  String Variables         →  String
deletevar        :  String Variables         →  Variables
isemptyvariables :  Variables                →  Boolean
```

FORMAL VARIABLES

```
vname  :  String
vsort  :  String
vars   :  Variables
```

AXIOMS

```
addvar(vname,vsort,vars) = modify(vname,vsort,vars)

getvarname(modify(vname,vsort,vars)) = vname

getvarsort(vname,vars) = imageof(vname,vars)

deletevar(vname,vars) = restrictwith(vars,add(vname,emptyset))

isemptyvariables(vars) = isempty(vars)
```

6.6 ATOInterfaces

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addinterface**: insere uma assinatura de operação algébrica;
- **createint**: cria uma assinatura de operação, inicialmente com o nome da operação e seu sort valor;
- **addarity**: insere os sorts da aridade em uma assinatura de operação;
- **getop**: dada uma assinatura de operação, retorna o nome desta operação;
- **isemptyaritylist**: verifica se a aridade de uma operação é vazia;
- **getvsort**: retorna o sort valor de uma assinatura de operação;
- **getsarity**: retorna o sort da aridade de uma assinatura de operação;
- **deletesarity**: apaga um sort da aridade de uma operação;
- **isemptyinterfaces**: verifica se existe declarações de operações;
- **getinterface**: retorna uma declaração de operação;
- **deleteinterface**: apaga uma declaração de operação.

CLASS

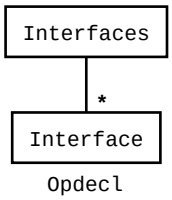


Figura 6.10: Classe que define Interfaces

INCLUDE

INTERFACES

```
addinterface      :  Interface Interfaces  →  Interfaces
createint         :  String String         →  Interface
addarity          :  String Interface      →  Interface
getop             :  Interface             →  String
isemptyaritylist  :  Interface             →  Boolean
getvsort          :  Interface             →  String
getsarity         :  Interface             →  String
deletesarity      :  String Interface      →  Opdecl
isemptyinterfaces :  Interfaces            →  Boolean
getinterface      :  Interfaces            →  Interface
deleteinterface   :  Interface Interfaces  →  Interfaces
```

FORMAL VARIABLES

```
ints         :  Interfaces
int          :  Interface
opname sort  :  String
```

AXIOMS

```
addinterface(int,ints) = cons(int,ints)

createint(opname,sort) = ICS(ATOOpdecl,createopdecl,opname,sort)

addarity(sort,int)
= ICS(ATOOpdecl,addsortarity,sort,int)

getop(int) = ICS(ATOOpdecl,getopname,int)

isemptyaritylist(int) = ICS(ATOOpdecl,isemptyarity,int)

getvsort(int) = ICS(ATOOpdecl,getvaluesort,int)

getsarity(int) = ICS(ATOOpdecl,getsortarity,int)

deletesarity(sort,int) = ICS(ATOOpdecl,deletesortarity,sort,int)

isemptyinterfaces(ints) = isempty(ints)

getinterface(ints) = head(int)

deleteinterface(int,ints) = delete(int,ints)
```

6.7 ATOOpdecl

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **createopdecl**: cria uma declaração de operação;
- **addsortarity**: insere um sort na aridade de uma operação;
- **getopname**: retorna o nome de uma operação;
- **isemptyarity**: verifica se a aridade da operação é vazia;
- **getvaluesort**: retorna o sort valor de uma operação;
- **getsortarity**: retorna a última declaração de operação declarada;
- **deletesortarity**: apaga um sort da aridade de uma operação.

CLASS

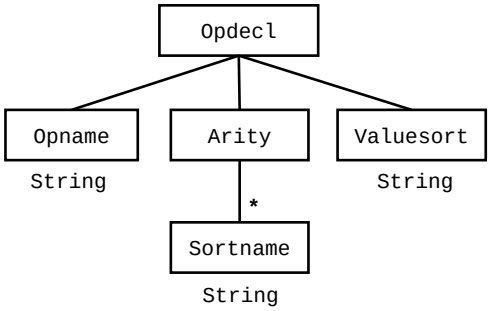


Figura 6.11: Classe que define Opdecl

INCLUDE

```
Boolean
```

INTERFACES

```
createopdecl     :  String String  →  Opdecl
addsortarity     :  String Opdecl  →  Opdecl
getopname        :  Opdecl         →  String
isemptyarity     :  Opdecl         →  Boolean
getvaluesort     :  Opdecl         →  String
getsortarity     :  Opdecl         →  String
deletesortarity  :  String Opdecl  →  Opdecl
```

FORMAL VARIABLES

```
aritylist    :  Arity
sort opname  :  String
opdecl       :  Opdecl
```

AXIOMS

```
createopdecl(opname,sort) = reg-Opdecl(opname,emptylist,sort)

addsortarity(sort,reg-Opdecl(_,aritylist,_))
= reg-Opdecl(_,cons(sort,aritylist),_)

getopname(opdecl) = select-Opname(opdecl)

isemptyarity(reg-Opdecl(_,aritylist,_)) = isempty(aritylist)

getvaluesort(opdecl) = select-Valuesort(opdecl)

getsortarity(opdecl) = head(select-Arity(opdecl))

deletesortarity(sort,reg-Opdecl(_,aritylist,_))
= reg-Opdecl(_,delete(sort,aritylist),_)
```

6.8 ATOAxioms

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addaxiom**: insere um axioma;
- **deleteaxiom**: apaga um axioma;
- **get-term-lhs**: retorna o lado esquerdo de um axioma;
- **get-term-rhs**: retorna o lado direito de um axioma;
- **isemptyaxioms**: verifica se existem axiomas.

CLASS

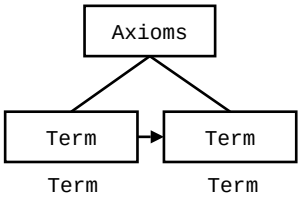


Figura 6.12: Classe que define Axioms

INCLUDE

INTERFACES

```
addaxiom       :  Term Term Axioms  →  Axioms
deleteaxiom    :  Term Axioms       →  Axioms
get-term-lhs   :  Axioms            →  Term
get-term-rhs   :  Term Axioms       →  Term
isemptyaxioms  :  Axioms            →  Boolean
```

FORMAL VARIABLES

```
axioms  :  Axioms
t tt    :  Term
terms   :  Terms
```

AXIOMS

```
addaxiom(t,tt,axioms) = modify(t,tt,axioms)

deleteaxiom(t,axioms) = restrictwith(axioms,add(t,emptyset))

get-term-lhs(modify(t,tt,axioms)) = t

get-term-rhs(t,axioms) = imageof(t,axioms)

isemptyaxioms(axioms) = isempty(axioms)
```

6.9 ATOTerm

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addop**: cria um termo algébrico;
- **addterm**: insere um termo dentro de outro termo;
- **getopname**: retorna o nome da operação de um termo;
- **addterms**: insere uma lista de termos dentro de um termo;
- **getterms**: retorna os subtermos de um termo;
- **hasparameters**: verifica se um termo tem subtermos;
- **parameterstail**: retorna os subtermos de um termo, sem o primeiro subtermo;
- **parametershead**: retorna o primeiro subtermo de um termo;
- **termslength**: conta quantos subtermos existem dentro de um termo;
- **termtoterms**: converte um termo para uma lista de termos;
- **concatterms**: concatena duas listas de termos.

CLASS

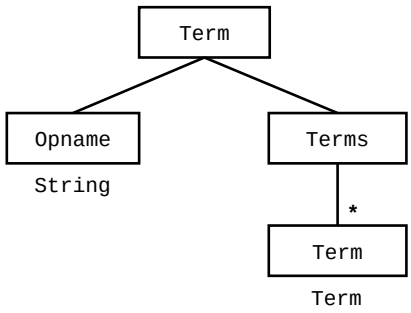


Figura 6.13: Classe que define Term

INCLUDE

```
Boolean Integer
```

INTERFACES

```
addop      :  String         →  Term
addterm    :  Term Term      →  Term
getopname  :  Term           →  String
addterms   :  String Terms   →  Term
getterms   :  Term           →  Terms
```

```
hasparameters   :  Term         →  Boolean
parameterstail  :  Terms        →  Terms
parametershead  :  Terms        →  Term
termslength     :  Terms        →  Integer
termtoterms     :  Term         →  Terms
concatterms     :  Terms Terms  →  Terms
```

FORMAL VARIABLES

```
    opname        :  String
    terms         :  Terms
    newterm t tt  :  Term
```

AXIOMS

```
addop(opname) = reg-Term(opname,emptylist)

addterm(newterm,reg-Term(_,terms))
= reg-Term(_,cons(newterm,terms))

getopname(t) = select-Opname(t)

addterms(opname,terms) = reg-Term(opname,terms)

getterms(t) = select-Terms(t)

hasparameters(reg-Term(_,terms)) = isempty(terms)

parameterstail(terms) = tail(terms)

parametershead(terms) = head(terms)

termslength(terms) = length(terms)

termtoterms(t) = cons(t,emptylist)

concatterms(t,tt) = concat(t,tt)
```

6.10 ATOOBJ

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **createOBJ**: cria um objeto somente com o nome;
- **declaresort**: insere uma declaração de sort;
- **delsortdecl**: apaga uma declaração de sort;
- **isemptysortdecl**: verifica se existem declarações de sort;
- **firstsortdecl**: retorna o primeiro sort declarado;
- **addimp**: insere uma importação de módulo;

- **del-imp**: apaga uma importação de módulo;
- **putimps**: insere uma lista de módulos importados;
- **getimps**: retorna uma lista de módulos importados;
- **isemptyimps**: verifica se existe importação de módulos;
- **lastimpmode**: retorna o modo de importação do último módulo importado;
- **lastimpmod**: retorna o último módulo importado;
- **insopdecl**: insere declaração de operação;
- **isemptyopdecl**: verifica se existe declaração de operação;
- **firstopdecl**: retorna a primeira declaração de operação;
- **delopdecl**: apaga uma declaração de operação;
- **mod-name**: retorna o nome do módulo;
- **createmodule**: cria um módulo, somente com seu nome;
- **renamesort**: insere uma renomeação de sort;
- **renameop**: insere uma renomeação de operação;
- **instmodule**: instancia um módulo dentro de outro;
- **isempty-sortrenm**: verifica se existem renomeações de sorts;
- **get-oldsort**: retorna o sort que será renomeado;
- **get-newsort**: retorna a renomeação de um sort;
- **del-sortrenm**: apaga uma renomeação de sort;
- **isempty-oprenm**: verifica se existem renomeações de operações;
- **get-oldop**: retorna a operação que será renomeada;
- **get-newop**: retorna a renomeação de uma operação;
- **del-oprenm**: apaga uma renomeação de operação;
- **isempty-inst**: verifica se um módulo tem instanciações;
- **last-module**: retorna o primeiro módulo instanciado;
- **del-instmod**: apaga uma instanciação de módulo;
- **declareop**: cria uma declaração de operação;
- **addarity**: insere um sort na aridade de uma operação;
- **get-opname**: retorna o nome de uma operação;

- **isempty-arity**: verifica se a aridade da operação é vazia;
- **get-valuesort**: retorna o sort valor de uma operação;
- **get-sortarity**: retorna a última declaração de operação declarada;
- **del-sortarity**: apaga um sort da aridade de uma operação;
- **addvar**: insere uma declaração de variável;
- **getvarname**: dada uma declaração de variável, retorna o nome da variável;
- **getvarsort**: dada uma declaração de variável, retorna o sort da variável;
- **delvar**: apaga uma declaração de variável;
- **isempty-vars**: verifica se existem declarações de variáveis;
- **addeq**: insere uma equação;
- **del-eq**: apaga uma equação;
- **get-lhs**: retorna o lado esquerdo de uma equação;
- **get-rhs**: retorna o lado direito de uma equação;
- **isempty-eqs**: verifica se existem equações;
- **add-op-in-term**: cria um termo algébrico;
- **add-term-in-term**: insere um termo dentro de outro termo;
- **add-terms**: insere uma lista de termos dentro de um termo;
- **get-opname**: retorna o nome da operação de um termo;
- **getparams**: retorna os subtermos de um termo;
- **hasparams**: verifica se um termo tem subtermos;
- **paramstail**: retorna os subtermos de um termo, sem o primeiro subtermo;
- **paramshead**: retorna o primeiro subtermo de um termo;
- **terms-length**: conta quantos subtermos existem dentro de um termo;
- **ins-line**: insere uma string na lista;
- **ins-text**: concatena duas listas de strings;
- **invert-text**: inverte a lista de strings;
- **phi-txt**: converte um objeto OBJ para a notação puramente textual (lista de strings);
- **phi-sortdecl**: traduz as declarações de sorts para texto;
- **phi-imp**: traduz a importação de módulos para texto;

- **isprimtypes**: verifica se é um módulo primitivo;
- **phi-mod**: traduz módulos para texto;
- **rnmsort**: traduz uma renomeação de sort para texto;
- **rnmops**: traduz uma renomeação de operação para texto;
- **aux-mod**: operação auxiliar para módulos;
- **phi-spec**: cria especificações quando a classe é recursiva;
- **create-spec-list**: cria operações para criação e manipulação de listas;
- **create-spec-set**: cria operações para criação e manipulação de conjuntos;
- **create-spec-map**: cria operações para criação e manipulação de mapeamentos;
- **create-spec-reg**: cria operações para criação e manipulação de registros;
- **create-spec-du**: cria operações para criação e manipulação de união disjuntivas;
- **phi-opdecl**: traduz as declarações de operações para texto;
- **str-opdecl**: operação auxiliar de phi-opdecl;
- **str-arity**: operação auxiliar de phi-opdecl;
- **phi-vars**: traduz variáveis para texto;
- **phi-eqs**: traduz equações para texto;
- **phi-t**: traduz um termo para texto;
- **sub-ts**: traduz subtermos para texto;
- **mixfix**: traduz operações mixfix para texto.

Para compreender a semântica das operações `create-spec-list`,`create-spec-set`, `create-spec-map`, `create-spec-reg`, `create-spec-du` verifique o anexo C. A especificação formal destas operações aumentaria muito o tamanho da especificação deste ATO. Além disso dificultaria o entendimento da semântica destas operações, visto que cada definição ocuparia quase uma página de texto.

CLASS

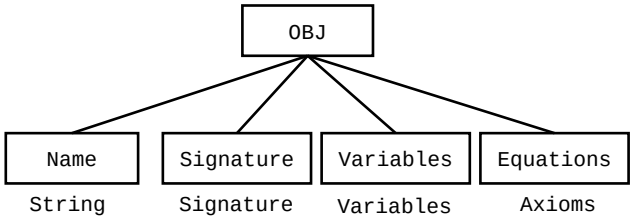


Figura 6.14: Classe que define OBJ

INCLUDE

ATOText

INTERFACES

```
createOBJ         :  String                       →  OBJ
declaresort       :  String OBJ                   →  OBJ
delsortdecl       :  String OBJ                   →  OBJ
isemptysortdecl   :  OBJ                          →  Boolean
firstsortdecl     :  OBJ                          →  String
addimp            :  String Module OBJ            →  OBJ
del-imp           :  String Module Importations   →  Importations
putimps           :  Importations OBJ             →  OBJ
getimps           :  OBJ                          →  Importations
isemptyimps       :  OBJ                          →  Boolean
lastimpmode       :  OBJ                          →  String
lastimpmod        :  OBJ                          →  Module
insopdecl         :  Opdecl OBJ                   →  Signature
isemptyopdecl     :  OBJ                          →  Boolean
firstopdecl       :  OBJ                          →  Opdecl
delopdecl         :  Opdecl OBJ                   →  Signature
mod-name          :  Module                       →  String
createmodule      :  String                       →  Module
renamesort        :  String String Module         →  Module
renameop          :  String String Module         →  Module
instmodule        :  Module Module                →  Module
isempty-sortrenm  :  Module                       →  Boolean
get-oldsort       :  Module                       →  String
get-newsort       :  String Module                →  String
del-sortrenm      :  String Module                →  Module
isempty-oprenm    :  Module                       →  Boolean
get-oldop         :  Module                       →  String
get-newop         :  String Module                →  String
del-oprenm        :  String Module                →  Module
```

```
isempty-inst      :  Module             →  Boolean
last-module       :  Module             →  Module
del-instmod       :  Module Module      →  Module
declareop         :  String String      →  Opdecl
addarity          :  String Opdecl      →  Opdecl
get-opname        :  Opdecl             →  String
isempty-arity     :  Opdecl             →  Boolean
get-valuesort     :  Opdecl             →  String
get-sortarity     :  Opdecl             →  String
del-sortarity     :  String Opdecl      →  Opdecl
addvar            :  String String OBJ  →  OBJ
getvarname        :  OBJ                →  String
getvarsort        :  String OBJ         →  String
delvar            :  String OBJ         →  Variables
isempty-vars      :  OBJ                →  Boolean
addeq             :  Term Term OBJ      →  OBJ
del-eq            :  Term OBJ           →  OBJ
get-lhs           :  OBJ                →  Term
get-rhs           :  Term OBJ           →  Term
isempty-eqs       :  OBJ                →  Boolean
add-op-in-term    :  String             →  Term
add-term-in-term  :  Term Term          →  Term
add-terms         :  String Terms       →  Term
get-opname        :  Term               →  String
getparams         :  Term               →  Terms
hasparams         :  Term               →  Boolean
paramstail        :  Terms              →  Terms
paramshead        :  Terms              →  Term
terms-length      :  Terms              →  Integer
phi-txt           :  OBJ                →  Text
ins-line          :  String Text        →  Text
ins-text          :  Text Text          →  Text
invert-text       :  Text               →  Text
phi-sortdecl      :  OBJ Text           →  Text
phi-imp           :  OBJ Text           →  Text
isprimtypes       :  String             →  Boolean
phi-mod           :  Module String      →  String
rnmsort           :  Module             →  String
rnmops            :  Module String      →  String
aux-mod           :  Module String      →  String
phi-spec          :  Module Text        →  Text
create-spec-set   :  Module Text        →  Text
create-spec-list  :  Module Text        →  Text
create-spec-map   :  Module Text        →  Text
create-spec-reg   :  Module Text        →  Text
create-spec-du    :  Module Text        →  Text
```

```
phi-opdecl  :  OBJ Text       →  Text
str-opdecl  :  Opdecl         →  String
str-arity   :  Opdecl String  →  String
phi-vars    :  OBJ Text       →  Text
phi-eqs     :  OBJ Text       →  Text
phi-t       :  Term String    →  String
sub-ts      :  Terms String   →  String
mixfix      :  String Terms   →  String
```

FORMAL VARIABLES

```
    name opname      :  String
    sort mode str    :  String
    sig              :  Signature
    opdecl op        :  Opdecl
    vars             :  Variables
    eqs              :  Equations
    t t1 t2          :  Term
    terms            :  Terms
    mod mod1         :  Module
    prmode           :  String
    oldsort newsort  :  String
    oldop newop      :  String
    imps             :  Importations
    obj              :  OBJ
    txt txt1         :  Text
    line             :  String
```

AXIOMS

```
createOBJ(name)
= reg-OBJ(name,ICS(ATOSignature,reg-Signature(emptylist,emptylist,
    emptylist)),ICS(ATOVariables,emptymap),ICS(ATOAxioms,emptymap))

declaresort(sort,reg-OBJ(_,sig,_,_))
= reg-OBJ(_,ICS(ATOSignature,addsortdecl,sort,sig),_,_)

delsortdecl(sort,reg-OBJ(_,sig,_,_))
= reg-OBJ(_,ICS(ATOSignature,delsortdecl,sort,sig),_,_)

isemptysortdecl(reg-OBJ(_,sig,_,_))
= ICS(ATOSignature,isemptysortdecl,sig)

firstsortdecl(reg-OBJ(_,sig,_,_))
= ICS(ATOSignature,firstsortdecl,sig)

addimp(mode,mod,reg-OBJ(_,sig,_,_))
= reg-OBJ(_,ICS(ATOSignature,addimp,mode,mod,sig),_,_)

del-imp(mode,mod,imps) = ICS(ATOSignature,delimp,mode,mod,imps)

putimps(imps,reg-OBJ(_,sig,_,_))
= reg-OBJ(_,ICS(ATOSignature,putimps,imps,sig),_,_)
```

```
getimps(reg-OBJ(_,sig,_,_)) = ICS(ATOSignature,getimps,sig)

isemptyimps(reg-OBJ(_,sig,_,_)) = ICS(ATOSignature,isemptyimp,sig)

lastimpmode(reg-OBJ(_,sig,_,_)) = ICS(ATOSignature,lastimpmode,sig)

lastimpmod(reg-OBJ(_,sig,_,_)) = ICS(ATOSignature,lastimpmod,sig)

insopdecl(opdecl,reg-OBJ(_,sig,_,_))
= reg-OBJ(_,ICS(ATOSignature,addopdecl,opdecl,sig),_,_)

isemptyopdecl(reg-OBJ(_,sig,_,_))
= ICS(ATOSignature,isemptyopdecl,sig)

firstopdecl(reg-OBJ(_,sig,_,_))
= ICS(ATOSignature,firstopdecl,sig)

delopdecl(opdecl,reg-OBJ(_,sig,_,_))
= reg-OBJ(_,ICS(ATOSignature,delopdecl,opdecl,sig),_,_)

mod-name(mod) = ICS(ATOModule,getmodname,mod)

createmodule(name) = ICS(ATOModule,createmod,name)

renamesort(oldsort,newsort,mod)
= ICS(ATOModule,rnmsort,oldsort,newsort,mod)

renameop(oldop,newop,mod) = ICS(ATOModule,rnmop,oldop,newop,mod)

instmodule(mod,mod1) = ICS(ATOModule,instmod,mod,mod1)

isempty-sortrenm(mod) = ICS(ATOModule,isemptysortrenm,mod)

get-oldsort(mod) = ICS(ATOModule,getoldsort,mod)

get-newsort(sort,mod) = ICS(ATOModule,getnewsort,sort,mod)

del-sortrenm(sort,mod) = ICS(ATOModule,delsortrenm,sort,mod)

isempty-oprenm(mod) = ICS(ATOModule,isemptyoprenm,mod)

get-oldop(mod) = ICS(ATOModule,getoldop,mod)

get-newop(opname,mod) = ICS(ATOModule,getnewop,opname,mod)

del-oprenm(opname,mod) = ICS(ATOModule,deloprenm,opname,mod)

isempty-inst(mod) = ICS(ATOModule,isemptyinstantiate,mod)

last-module(mod) = ICS(ATOModule,lastmodule,mod)
```

```
del-instmod(mod,mod1) = ICS(ATOModule,delinstmod,mod,mod1)

declareop(opname,sort) = ICS(ATOOpdecl,createopdecl,opname,sort)

addarity(sort,opdecl) = ICS(ATOOpdecl,addsortarity,sort,opdecl)

get-opname(opdecl) = ICS(ATOOpdecl,getopname,opdecl)

isempty-arity(opdecl) = ICS(ATOOpdecl,isemptyarity,opdecl)

get-valuesort(opdecl) = ICS(ATOOpdecl,getvaluesort,opdecl)

get-sortarity(opdecl) = ICS(ATOOpdecl,getsortarity,opdecl)

del-sortarity(sort,opdecl)
= ICS(ATOOpdecl,deletesortarity,opdecl,sort)

addvar(name,sort,reg-OBJ(_,_,vars,_))
= reg-OBJ(_,_,ICS(ATOVariables,addvar,name,sort,vars),_)

getvarname(reg-OBJ(_,_,vars,_))
= ICS(ATOVariables,getvarname,vars)

getvarsort(name,reg-OBJ(_,_,vars,_))
= ICS(ATOVariables,getvarsort,name,vars)

delvar(name,reg-OBJ(_,_,vars,_))
= ICS(ATOVariables,deletevar,name,vars)

isempty-vars(reg-OBJ(_,_,vars,_))
= ICS(ATOVariables,isemptyvariables,vars)

addeq(t,t1,reg-OBJ(_,_,_,eqs))
= reg-OBJ(_,_,_,ICS(ATOAxioms,addaxiom,t,t1,eqs))

del-eq(reg-OBJ(_,_,_,eqs),t) = ICS(ATOAxioms,deleteaxiom,eqs,t)

get-lhs(reg-OBJ(_,_,_,eqs)) = ICS(ATOAxioms,get-term-lhs,eqs)

get-rhs(t,reg-OBJ(_,_,_,eqs)) = ICS(ATOAxioms,get-term-rhs,t,eqs)

isempty-eqs(reg-OBJ(_,_,_,eqs)) = ICS(ATOAxioms,isemptyaxioms,eqs)

add-op-in-term(name) = ICS(ATOTerm,addop,name)

add-term-in-term(t1,t2) = ICS(ATOTerm,addterm,t1,t2)

add-terms(name,terms) = ICS(ATOTerm,addterms,name,terms)

get-opname(t) = ICS(ATOTerm,getopname,t)
```

```
getparams(t) = ICS(ATOTerm,getterms,t)

hasparams(t) = ICS(ATOTerm,hasparameters,t)

paramstail(terms) = ICS(ATOTerm,parameterstail,terms)

paramshead(terms) = ICS(ATOTerm,parametershead,terms)

terms-length(terms) = ICS(ATOTerm,termslength,terms)

ins-line(line,txt) = ICS(ATOText,addline,line,txt)

ins-text(txt,txt1) = ICS(ATOText,addtext,txt,txt1)

invert-text(txt) = ICS(ATOText,inverttext,txt)

phi-txt(obj,txt)
= phi-sortdecl(obj,ins-line("obj \space"++select-Name(obj)++
       "\space is",ICS(ATOText,emptylist)))

phi-sortdecl(obj,txt)
=
if isemptysortdecl(obj)
 then phi-imp(obj,txt)
 else phi-sortdecl(delsortdecl(obj,lastsortdecl(obj)),ins-line(
        "sort \space "++lastsortdecl(obj)++"\space .",txt))

phi-imp(obj,txt)
=
if isemptyimps(obj)
  then phi-opdecl(obj,txt)
  else
    if lastimpmode(obj) == "create\-spec"
      then phi-imp(putimps(del-imp("create\-spec",lastimpmod(obj),
               getimps(obj)),obj),add-text(txt,phi-spec(lastimpmod(
               obj),emptylist)))
      else phi-imp(putimps(del-imp("pr",lastimpmod(obj),getimps(
               obj)),obj),ins-line("\space pr \space "++phi-mod(
               lastimpmod(obj),"\empty")++"\space .",txt))

isprimtypes(name)
= if name == "BOOL" or name == "INT" or name == "STRING" or
     name == "DATE" or name == "TIME" then true else false

phi-mod(mod,str)
=
if isprimtypes(mod-name(mod))
 then str++mod-name(mod)
 else
  if isin("ATO",mod-name(mod)) and isempty-sortrenm(mod)
```

```
   then str++mod-name(mod)
   else
    if isin("ATO",mod-name(mod))
     then str++"\paropen"++mod-name(mod)++"\space \*\paropen sort
              \space"++get-oldsort(mod)++"\space to \space"++
              get-newsort(get-oldsort(mod),mod)++"\parclose\parclose"
     else
      if (mod-name(mod) == "LIST") or (mod-name(mod) == "SET")
       then str++"\paropen"++mod-name(mod)++"\space \*\paropen sort
              \space"++get-oldsort(mod)++"\space to \space"++
              get-newsort(get-oldsort(mod),mod)++"\parclose\parclose\["
              ++phi-mod(last-module(mod),"\empty")++"\]"
       else
        if mod-name(mod) == "MAP"
         then
          str++"\paropen"++mod-name(mod)++"\space \*\paropen sort
            \space"++get-oldsort(mod)++"\space to \space"++
            get-newsort(get-oldsort(mod),mod)++"\parclose\parclose\["
            ++phi-mod(last-module(mod))++"\comma"++phimod(
            last-module(del-instmod(last-module(mod),mod)),"\empty")
            ++"\]"
         else
          if isin("REGISTER",mod-name(mod)) or (isin("DISJOINTUNION",
               mod-name(mod)))
           then
             str++"\paropen"++mod-name(mod)++"\space \*\paropen sort
               \space"++rnmsort(mod)++rnmops(mod,"\empty")++
               "\parclose\parclose\["++aux-mod(mod,"\empty")++"\]"
           else str


rnmsort(mod)
= get-oldsort(mod)++"\space to \space"++get-newsort(get-oldsort(mod),
    mod)

rnmops(mod,str)
=
if isempty-oprenm(mod)
 then str
 else "\comma op \space "++get-oldop(mod)++"\space to \space"++
        get-newop(get-oldop(mod),mod)++rnmops(del-oprenm(get-oldop(
        mod),mod),"\empty")

aux-mod(mod,str)
=
if isempty-inst(mod)
 then str
 else phi-mod(last-module(mod),"\empty")++aux-mod(del-instmod(
         last-module(mod),mod),"\empty"))

phi-spec(mod,txt)
```

```
=
if mod-name(mod) == "SET"
  then create-spec-set(mod,txt)
  else
   if mod-name(mod) == "LIST"
     then create-spec-list(mod,txt)
     else
      if mod-name(mod) == "MAP"
       then create-spec-map(mod,txt)
       else
        if mod-name(mod) == "REGISTER"
         then create-spec-reg(mod,txt)
         else
          if mod-name(mod) == "DISJOINTUNION"
           then create-spec-du(mod,txt)
           else txt

phi-opdecl(obj,txt) = if isemptyopdecl(obj)
 then phi-vars(obj,txt)
 else phi-opdecl(delopdecl(lastopdecl(obj),obj),ins-line(str-opdecl(
         last-opdecl(obj)),txt))


str-opdecl(op)
= "\space op \space "++get-opname(op)++"\space \: \space"++str-arity(
     op,"\empty")++"\space \-\> \space"++get-valuesort(op)++"\space\."

str-arity(op,str)
=
if isempty-arity(op)
 then str
 else str++"\space"++get-sortarity(op)++str-arity(del-sortarity(
        get-sortarity(op),op),"\empty")


phi-vars(obj,txt)
=
if isempty-vars(obj)
 then phi-eqs(obj,txt)
 else phi-vars(delvar(getvarname(obj),obj),ins-line("\space var
       \space"++getvarname(obj)++"\space \: \space "++
       getvarsort(getvarname(obj),obj)++"\.",txt))


phi-eqs(obj,txt)
=
if isempty-eqs(obj)
 then ins-line("endo",txt)
 else phi-eqs(del-eq(get-lhs(obj),obj),ins-line("\space eq
         \space "++phi-t(get-lhs(obj),"\empty")++"\space \=
         \space"++phi-t(get-rhs(get-lhs(obj),obj),"\empty")++
```

```
         "\space\.",txt))


phi-t(t,str)
=
if not(hasparams(t))
 then get-opname(t)             *** variable or constructor
 else
  if not(isin("\underline",get-opname(t)))
   then get-opname(t)++"\paropen"++sub-ts(getparams(t),"\empty")
            ++"\parclose"   *** prefix op without place-holders
   else mixfix(get-opname(t),getparams(t))  *** mixfix


sub-ts(terms,str)
=
if terms == ICS(ATOTerm,emptylist)
 then str
 else
  if terms-length(terms) == 1
    then str++phi-t(paramshead(terms),"\empty")
    else sub-ts(paramstail(terms),str++phi-t(paramshead(terms),
           "\empty")++"\comma")


mixfix(opname,terms)
=
if terms == emptylist
 then opname
 else mixfix(rep1("\underline","\space"++phi-t(paramshead(terms)),
        "\empty")++"\space",opname),paramstail(terms))
```

6.11 ATOSignature

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addsortdecl**: insere uma declaração de sort;
- **delsortdecl**: apaga uma declaração de sort;
- **isemptysortdecl**: verifica se existem declarações de sort;
- **firstsortdecl**: retorna o primeiro sort declarado;
- **addimp**: insere uma importação de módulo;
- **delimp**: apaga uma importação de módulo;
- **putimps**: insere várias importações em um módulo;
- **getimps**: retorna os módulos importados;

- **isemptyimp**: verifica se existem importações de módulos;
- **lasimpmode**: retorna o modo de importação da primeira importação de módulos;
- **lastimpmod**: retorna o primeiro módulo importado;
- **addopdecl**: insere uma declaração de operação na assinatura;
- **isemptyopdecl**: verifica se existem declarações de operações;
- **firstopdecl**: retorna a primeira declaração de operação;
- **delopdecl**: apaga uma declaração de operação;

CLASS

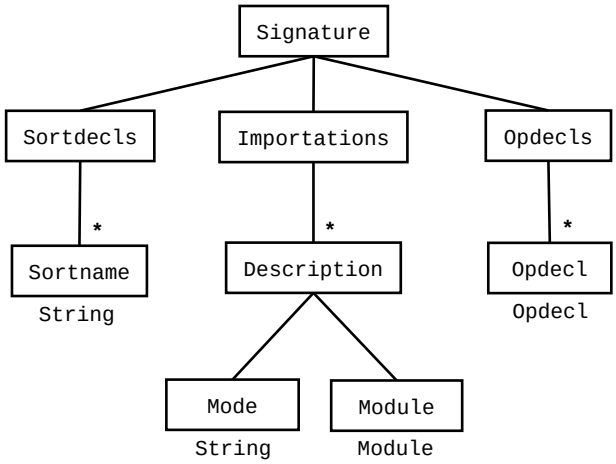


Figura 6.15: Classe que define Signature

INCLUDE

INTERFACES

```
addsortdecl      :  String Signature           →  Signature
delsortdecl      :  String Signature           →  Signature
isemptysortdecl  :  Signature                  →  Boolean
firstsortdecl    :  Signature                  →  String
addimp           :  String Module Signature    →  Signature
delimp           :  String Module Importations →  Importations
putimps          :  Importations Signature     →  Signature
getimps          :  Signature                  →  Importations
isemptyimp       :  Signature                  →  Boolean
lastimpmode      :  Signature                  →  String
lastimpmod       :  Signature                  →  Module
addopdecl        :  Opdecl Signature           →  Signature
isemptyopdecl    :  Signature                  →  Boolean
firstopdecl      :  Signature                  →  Opdecl
delopdecl        :  Opdecl Signature           →  Signature
```

FORMAL VARIABLES

```
sig           :  Signature
newsort sort  :  String
s             :  Sortdecls
ops           :  Opdecls
newop op      :  Opdecl
imps          :  Importations
impmode mode  :  String
mod mod1      :  Module
```

AXIOMS

```
addsortdecl(newsort,reg-Signature(s,_,_))
= reg-Signature(cons(newsort,s),_,_)

delsortdecl(sort,reg-Signature(s,_,_))
= reg-Signature(delete(sort,s),_,_)

isemptysortdecl(reg-Signature(s,_,_)) = isempty(s)

firstsortdecl(reg-Signature(s,_,_)) = last(s)

addimp(impmode,mod,reg-Signature(_,imps,_))
= reg-Signature(_,cons(reg-Description(impmode,mod),imps),_)

delimp(mode,mod1,cons(reg-Description(impmode,mod),imps))
= if (impmode == mode) and (mod == mod1)
then imps
else cons(reg-Description(impmode,mod),delimp(imps,mode,mod1))

putimps(imps,reg-Signature(_,_,_)) = reg-Signature(_,imps,_)

getimps(sig) = select-Importations(sig)

isemptyimp(reg-Signature(_,imps,_))
= isempty(imps)

lastimpmode(reg-Signature(_,imps,_)) = select-Mode(last(imps))

lastimpmod(reg-Signature(_,imps,_)) = select-Module(last(imps))

addopdecl(newop,reg-Signature(_,_,ops))
= reg-Signature(_,_,cons(newop,ops))

isemptyopdecl(reg-Signature(_,_,ops)) = isempty(ops)

firstopdecl(reg-Signature(_,_,ops))) = head(op)

delopdecl(op,reg-Signature(_,_,ops)) = delete(op,ops)
```

6.12 ATOModule

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **createmod**: cria um módulo, somente com seu nome;
- **getmodname**: retorna o nome do módulo;
- **rnmsort**: insere uma renomeação de sort;
- **rnmop**: insere uma renomeação de operação;
- **instmod**: instancia um módulo dentro de outro;
- **isemptysortrenm**: verifica se existem renomeações de sorts;
- **getoldsort**: retorna o sort que será renomeado;
- **getnewsort**: retorna a renomeação de um sort;
- **delsortrenm**: apaga uma renomeação de sort;
- **isemptyoprenm**: verifica se existem renomeações de operações;
- **getoldop**: retorna a operação que será renomeada;
- **getnewop**: retorna a renomeação de uma operação;
- **deloprenm**: apaga uma renomeação de operação;
- **isemptyinstantiate**: verifica se um módulo tem instanciações;
- **lastmodule**: retorna o primeiro módulo instanciado;
- **delinstmod**: apaga uma instanciação de módulo.

CLASS

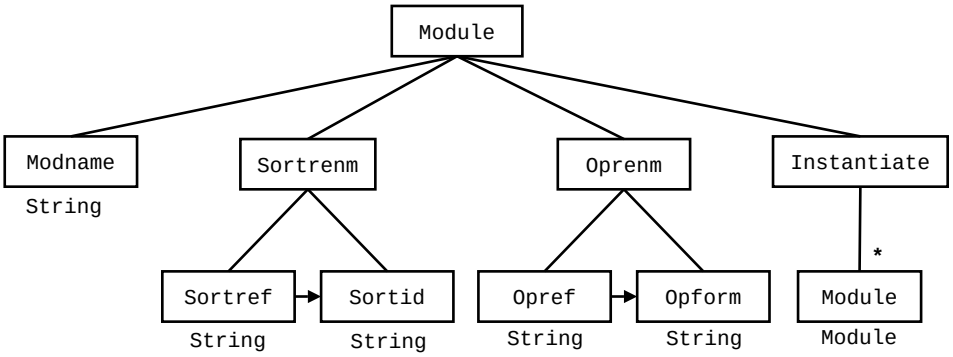


Figura 6.16: Classe que define Module

INCLUDE

Boolean

INTERFACES

```
createmod          :  String                →  Module
getmodname         :  Module                →  String
rnmsort            :  String String Module  →  Module
rnmop              :  String String Module  →  Module
instmod            :  Module Module         →  Module
isemptysortrenm    :  Module                →  Boolean
getoldsort         :  Module                →  String
getnewsort         :  String Module         →  String
delsortrenm        :  String Module         →  Module
isemptyoprenm      :  Module                →  Boolean
getoldop           :  Module                →  String
getnewop           :  String Module         →  String
deloprenm          :  String Module         →  Module
isemptyinstantiate :  Module                →  Boolean
lastmodule         :  Module                →  Module
delinstmod         :  Module Module         →  Module
```

FORMAL VARIABLES

```
name sort opname  :  String
oldsort newsort   :  String
sortmap           :  Sortrenm
opmap             :  Oprenm
oldop newop       :  String
mod               :  Module
modlist           :  Instantiate
```

AXIOMS

```
createmod(name) = reg-Module(name,emptymap,emptymap,emptylist)

getmodname(mod) = select-Modname(mod)

rnmsort(oldsort,newsort,reg-Module(_,sortmap,_,_))
= reg-Module(_,modify(oldsort,newsort,sortmap),_,_)

rnmop(oldop,newop,reg-Module(_,_,opmap,_))
= reg-Module(_,_,modify(oldop,newop,opmap),_)

instmod(mod,reg-Module(_,_,_,modlist))
= reg-Module(_,_,_,cons(mod,modlist))

isemptysortrenm(reg-Module(_,sortmap,_,_)) = isempty(sortmap)

getoldsort(reg-Module(_,modify(oldsort,newsort,sortmap),_,_) = oldsort

getnewsort(sort,reg-Module(_,sortmap,_,_)) = imageof(sort,sortmap)
```

```
delsortrenm(sort,reg-Module(_,sortmap,_,_))
= reg-Module(_,restrictwith(sortmap,add(sort,emptyset)),_,_)

isemptyoprenm(reg-Module(_,_,opmap,_)) = isempty(opmap)

getoldop(reg-Module(_,_,modify(oldop,newop,opmap),_) = oldop

getnewop(opname,reg-Module(_,_,opmap,_)) = imageof(opname,opmap)

deloprenm(opname,reg-Module(_,_,opmap,_))
= reg-Module(_,_,restrictwith(opmap,add(opname,emptyset)),_)

isemptyinstantiate(reg-Module(_,_,_,modlist)) = isempty(modlist)

lastmodule(reg-Module(_,_,_,modlist)) = last(modlist)

delinstmod(mod,reg-Module(_,_,_,modlist))
= reg-Module(_,_,_,delete(mod,modlist))
```

6.13 ATOText

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **addline**: insere uma string na lista;
- **addtext**: concatena duas listas de strings;
- **inverttext**: inverte a lista de strings.

CLASS

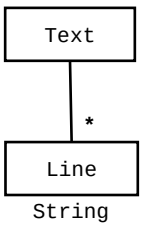


Figura 6.17: Classe que define Text

INCLUDE

INTERFACES

```
addline    :  String Text  →  Text
addtext    :  Text Text    →  Text
inverttext :  Text         →  Text
```

FORMAL VARIABLES

```
  txt txt1  :  Text
  line      :  String
```

AXIOMS

```
addline(line,txt) = cons(line,txt)

addtext(txt,txt1) = concat(txt,txt1)

inverttext(txt) = invert(txt)
```

6.14 Considerações Finais

Neste capítulo foram apresentados os processos utilizados para formalizar o ProTool e os respectivos ATOs algébricos resultantes desses processos. Além disso, também foi apresentada a seqüência de tradução de um ATO algébrico para um objeto OBJ puramente textual, que representa o protótipo executável do ATO.

A modularização imposta nas classes `MetaATO` e `OBJ` permitiu a reutilização de subclasses em ambas especificações. Por exemplo, um mesmo ATO especifica declarações de variáveis em PROSOFT e em OBJ, pois desconsiderando detalhes sintáticos a estrutura de declaração de ambas as linguagens é a mesma.

No `ATOMetaATO`, além das operações de tradução para OBJ, também foram especificadas operações para criação de termos da classe `MetaATO`, que representam ATOs algébricos. Tais operações serviram como base para derivar a implementação de um editor de ATOs algébricos no ambiente PROSOFT-Java.

O próximo capítulo aborda a implementação do ProTool em PROSOFT-Java. Esta ferramenta, produto da implementação das especificações deste capítulo, é tanto o editor de ATOs algébricos como o prototipador de ATOs em OBJ.

7 IMPLEMENTAÇÃO DO PROTOOL EM PROSOFT-JAVA

Este capítulo apresenta a implementação da ferramenta de prototipação ProTool no ambiente PROSOFT-Java. Os ATOs implementados são relacionados entre si de acordo com a arquitetura apresentada no capítulo 5.

Embora o objetivo principal deste trabalho seja prototipar ATOs, deve-se disponibilizar uma ferramenta para edição completa de ATOs algébricos, pois a implementação do ambiente PROSOFT-Java dispunha somente de um editor de classes. Para suprir esta necessidade, foi implementado um editor[1] de ATOs algébricos no PROSOFT-Java.

O editor de ATOs provê recursos tanto para criação de novos ATOs quanto para reutilização dos ATOs já existentes. Foi necessário definir o `ATOATO` para armazenar as dependências entre os ATOs envolvidos numa especificação de software. O lado esquerdo da figura 7.1 mostra a classe deste ATO.

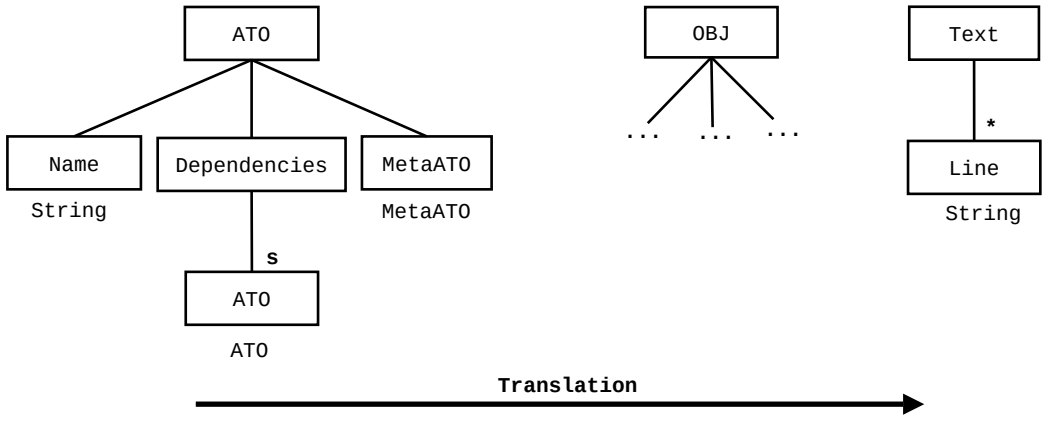


Figura 7.1: Estrutura da implementação do ProTool

As dependências de um ATO são os ATOs que pertencem aos nodos folha da classe e os ATOs incluídos pela cláusula *Include*. O `ATOATO` é usado somente como uma estrutura auxiliar para a implementação do editor de ATOs, portanto, suas operações não foram especificadas formalmente.

Analisando a figura 7.1, vê-se a geração de código textual OBJ a partir das especificações em PROSOFT-algébrico. Dados um ou mais ATOs (armazenados no `ATOATO`, estes são primeiramente traduzidos para a classe OBJ[2] para posterior tradução para uma lista de strings (classe `Text`) que corresponde a um objeto textual. A implementação do

[1] Este editor implementa os tipos de dados, funções e regras do ATOMetaATO.

[2] A classe OBJ da figura é somente um esboço da classe especificada no capítulo anterior.

ProTool também é responsável por “ler” esta lista de strings para gerar o arquivo texto a ser carregado no OBJ3.

Os ambientes PROSOFT-Java e OBJ3, apesar de rodarem no mesmo sistema operacional (Linux), não podem ser integrados, pois a implementação do OBJ3 é muito restrita quanto aos recursos disponíveis para interação com usuário. A idéia inicial era chamar o ambiente OBJ3 de dentro do PROSOFT-Java. Assim, seria possível carregar automaticamente os objetos gerados pelo ProTool no OBJ3. Mas o OBJ3 interpreta linhas de comandos e não fornece nenhum recurso para ser chamado e ao mesmo tempo carregar um arquivo (com o código OBJ a ser executado). Portanto este processo é manual.

7.1 Integração da Ferramenta ProTool no Ambiente PROSOFT-Java

A figura 7.2 mostra como deve ser chamada a ferramenta ProTool no ambiente PROSOFT-Java. E, na figura 7.3 aparece a tela principal da ferramenta ProTool.

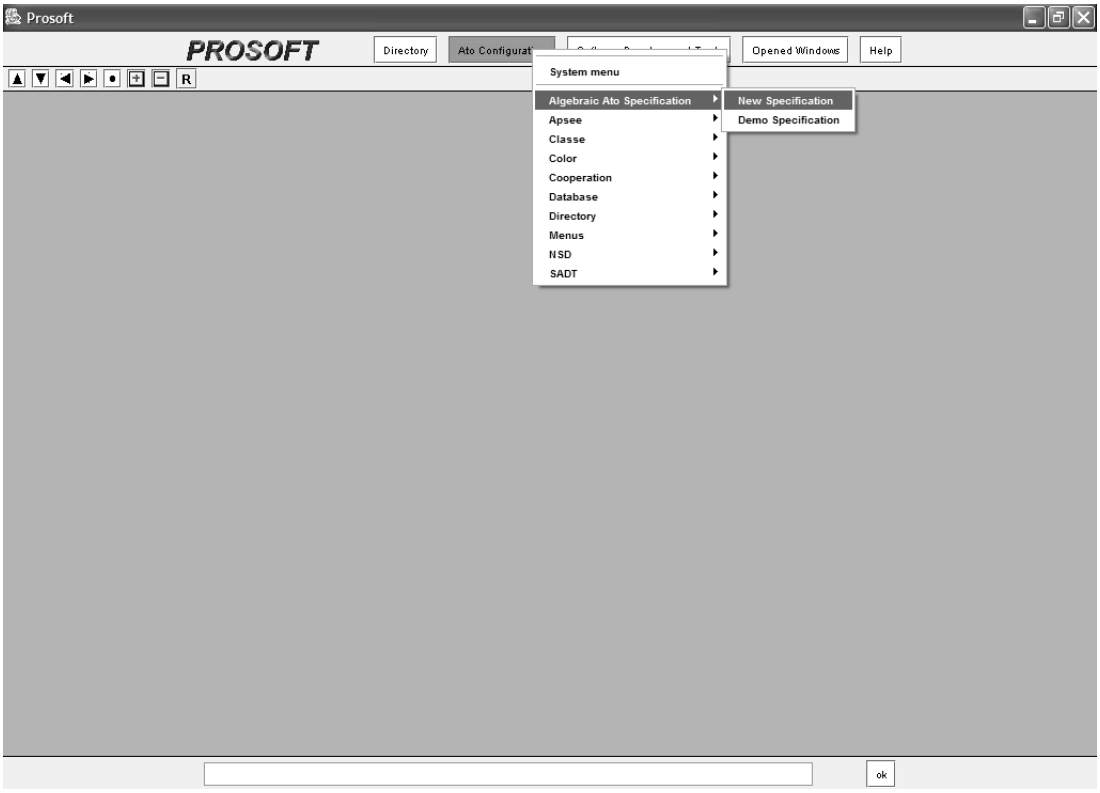


Figura 7.2: Tela principal do ambiente PROSOFT

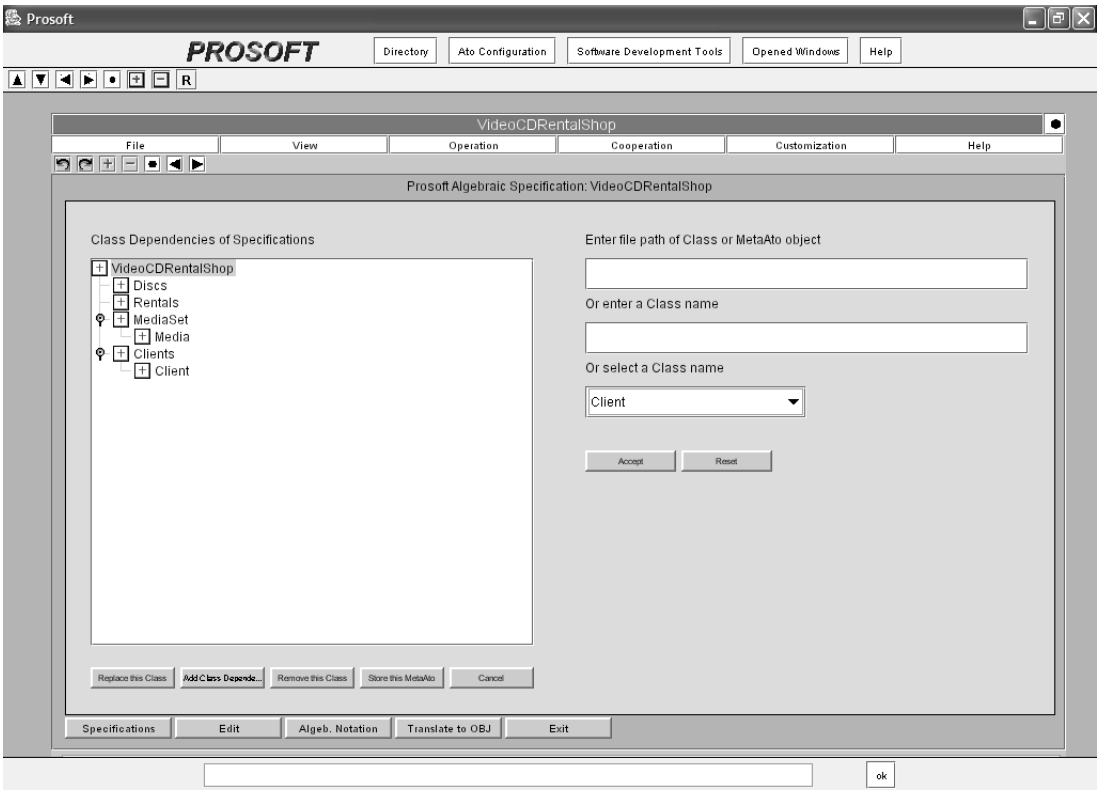


Figura 7.3: Tela principal do ProTool

7.2 Criação de Classes

A ferramenta para criação e manipulação de classes já existia na implementação do ambiente PROSOFT, como pode ser visto na figura 7.4.

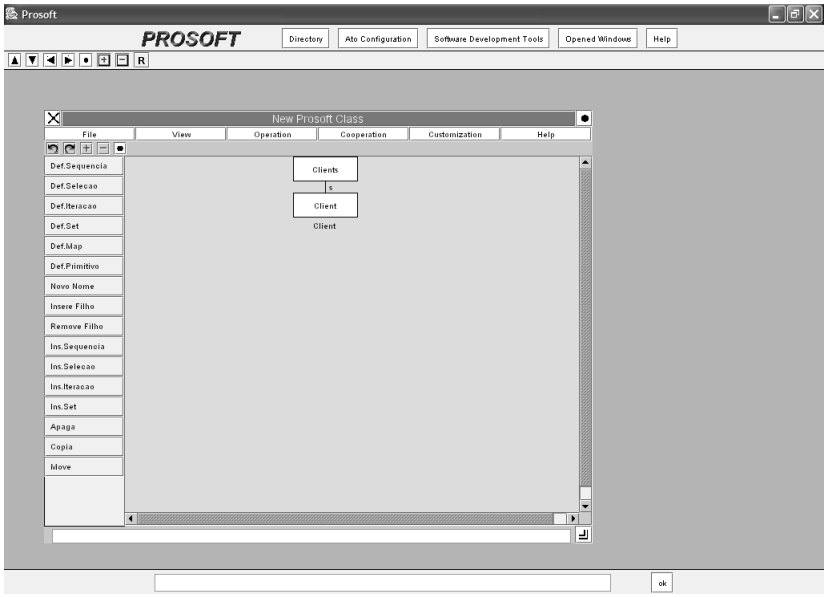


Figura 7.4: Ferramenta para edição de classes

A rigor é aconselhável criar todas as classes dos ATOs que serão utilizados por uma especificação, pois elas servirão para estabelecer a dependência entre os ATOs, como será mostrado adiante.

7.3 Principais Funcionalidades do ProTool

Na parte inferior da figura 7.3 aparecem os botões com as principais funcionalidade do editor/prototipador ProTool. A descrição de cada uma é a seguinte:

- **Specifications**: mostra as dependências entre ATOs;
- **Edit**: ao clicar neste botão, surgem as opções:
  - *Show Class*: mostra a classe do ATO selecionado;
  - *Include*: edita as importações do ATO;
  - *Interfaces*: edita as interfaces de operações;
  - *Formal Variables*: edita as variáveis formais;
  - *Axioms*: edita os axiomas.
- **Algeb. Notation**: ao clicar neste botão, surgem as opções:
  - *Show Algeb. Notation of this ATO*: mostra a especificação textual do ATO corrente;
  - *Show Algeb. Notation of Specification*: mostra a especificação textual da especificação completa (todos ATOs);

  - *Print Algeb. Notation of this ATO*: salva em arquivo a especificação textual do ATO corrente;
  - *Print Algeb. Notation of Specification*: salva em arquivo a especificação textual da especificação completa (todos ATOs).

- **Translate to OBJ**: ao clicar neste botão, surgem as opções:
  - *Show OBJ Notation of this ATO*: mostra a tradução do ATO corrente para OBJ;
  - *Show OBJ Notation of Specification*: mostra a tradução da especificação completa para OBJ;
  - *Print OBJ Notation of this ATO*: salva em arquivo a tradução do ATO corrente para OBJ;
  - *Print OBJ Notation of Specification*: salva em arquivo a tradução da especificação completa para OBJ.

- **Exit**: sair do ProTool.

7.4 Estabelecendo a Dependência entre ATOs

Uma especificação de software em PROSOFT é composta por um ATO principal que, possivelmente, utiliza vários outros ATOs. Portanto, após a definição das classes, deve-se estabelecer a hierarquia dos ATOs envolvidos, conforme mostrado na figura 7.3.

Estas dependências explícitas são necessárias pois pode-se carregar um ATO já criado pelo editor ou então criar um novo ATO a partir da classe criada anteriormente. Cada ATO é armazenado em arquivos diferentes e o ambiente precisa saber quais arquivos serão utilizados.

Neste formulário ainda existem operações para: criar uma dependência, remover dependência, substituir uma classe e ainda salvar um ATO em arquivo.

7.5 Editando as Importações de um ATO

Estão disponíveis para importação todos os tipos primitivos do PROSOFT, além dos ATOs que já foram carregados nas dependências da especificação. Para importar um ATO ou um primitivo, basta escolher uma opção listada na caixa de seleção.

7.6 Editando as Interfaces de Operações

A princípio existiam duas alternativas para implementar a edição de ATOs. Uma seria implementar um *parser* para reconhecer a linguagem do PROSOFT-algébrico, mas a implementação desta requer um esforço de programação muito grande e deixa o usuário muito "livre" para escrever especificações sintaticamente incorretas. A segunda alternativa, que foi adotada, baseia-se na edição dos ATOs através de caixas de texto e de caixas de seleção. Assim, o usuário fica limitado a "montar" o ATO usando somente o que já foi definido anteriormente e, portanto, reduzindo a probabilidade do usuário cometer erros sintáticos.

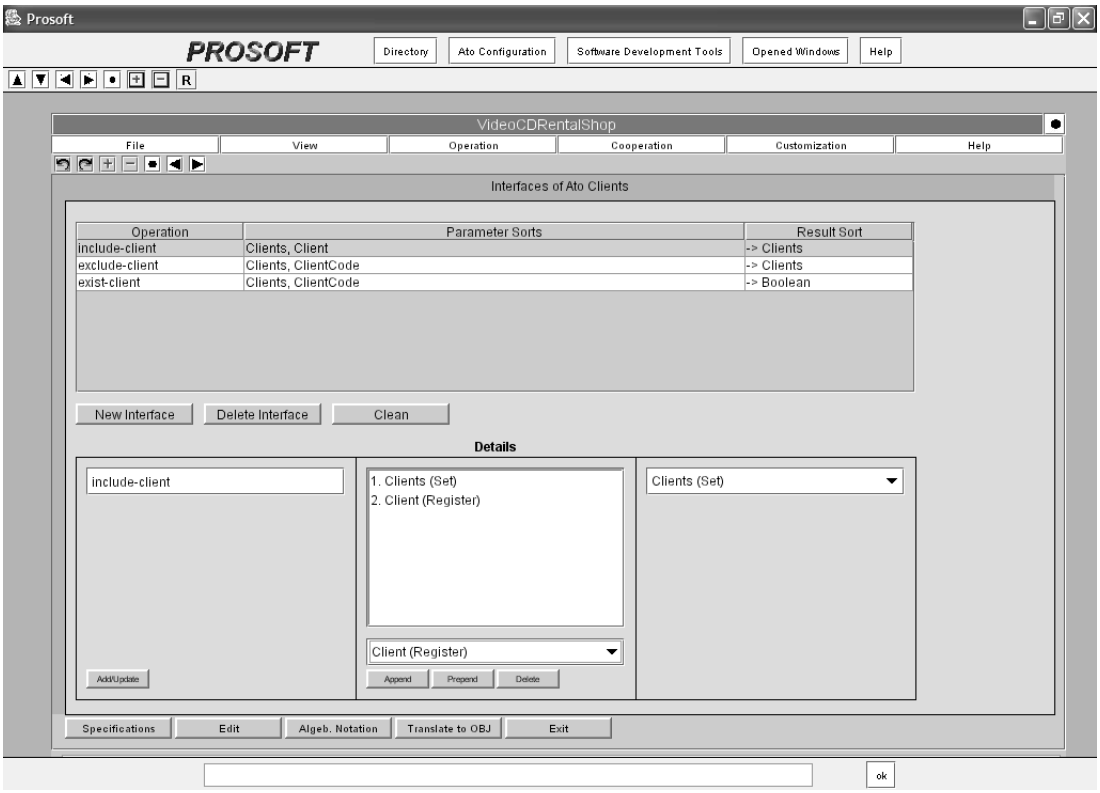


Figura 7.5: Edição de interfaces

Baseado nesta estratégia a declaração da assinatura (interface) de uma operação consiste na inserção do:

1. **Nome da operação**: Nome que representa a operação. Se for inserido um nome de acordo com a notação mixfix, deve-se usar *place-holders* (“_”) para cada argumento da operação. Se o nome não conter *place-holders* a operação é considerada pré-fixada;

2. **Sort(s) da aridade**: Nesta etapa são inseridos os sorts que fazem parte da aridade da operação. Podem ser usados sorts de qualquer ATO que faça parte da classe ou da cláusula “Include” do ATO;

3. **Sort valor**: Idem à etapa anterior, mas somente um sort é inserido, pois a lógica do PROSOFT-algébrico é de 1^a ordem.

De acordo com a figura 7.5, existem botões para inserir e remover cada parte que compõe uma interface.

7.7 Editando as Variáveis Formais

A criação e manipulação das variáveis formais de um ATO é realizada como mostra a figura 7.6.

Para criar uma nova variável formal deve-se inserir:

1. **Nome da variável**: String composta de letras e/ou números, sem espaços em branco no meio;

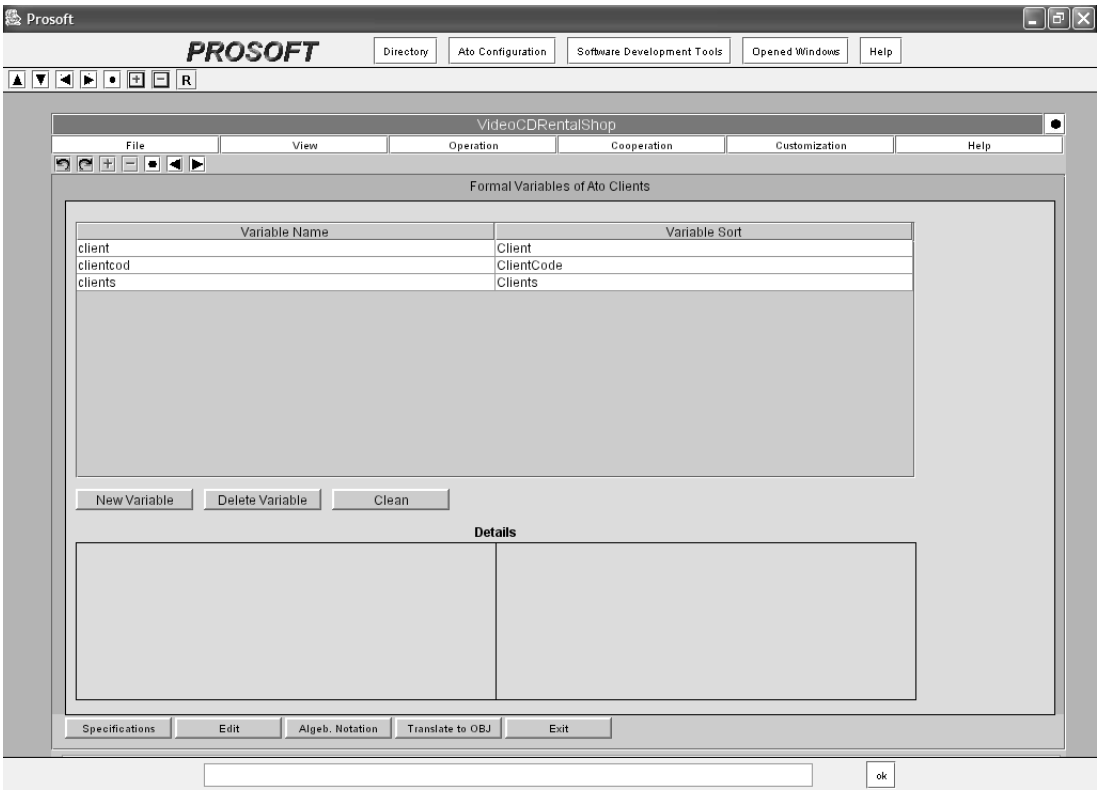


Figura 7.6: Edição das variáveis formais

2. **Sort da variável**: O sort pode ser qualquer um da classe ou dos ATOs que pertencem a cláusula “Include”. A escolha do sort é feita através da caixa de seleção que lista todos os sorts disponíveis.

A edição de variáveis conta também com opções para criar e apagar variáveis.

7.8 Editando os Axiomas

A edição de axiomas também utiliza a estratégia de “montagem” através do que já havia sido especificado. A figura 7.7 mostra o formulário para editar axiomas. A figura 7.8 mostra a definição dos axiomas segundo a estratégia de “montagem”.

Para inserir um novo axioma, deve-se primeiramente escolher para qual operação será dada a semântica. Isto é feito através da seleção da operação em uma caixa de seleção, que lista todas operações declaradas anteriormente na interface.

Os passos a seguir consistem em definir os campos que são listados como <*undefined*>. Nestes campos <*undefined*> podem ser inseridos operações sobre o sort deste argumento, variáveis formais, outras operações ou o símbolo “_”. A definição do axioma deve ser feita até todos elementos <*undefined*> serem definidos.

7.9 ATO – Representação Textual

A qualquer momento, o usuário pode visualizar a especificação do ATO através do formulário da figura 7.9. Este formulário conta com opções para visualizar somente um ATO ou então a especificação completa (todos ATOs da especificação). Além de visualizar, o usuário pode salvar a especificação em arquivo texto.

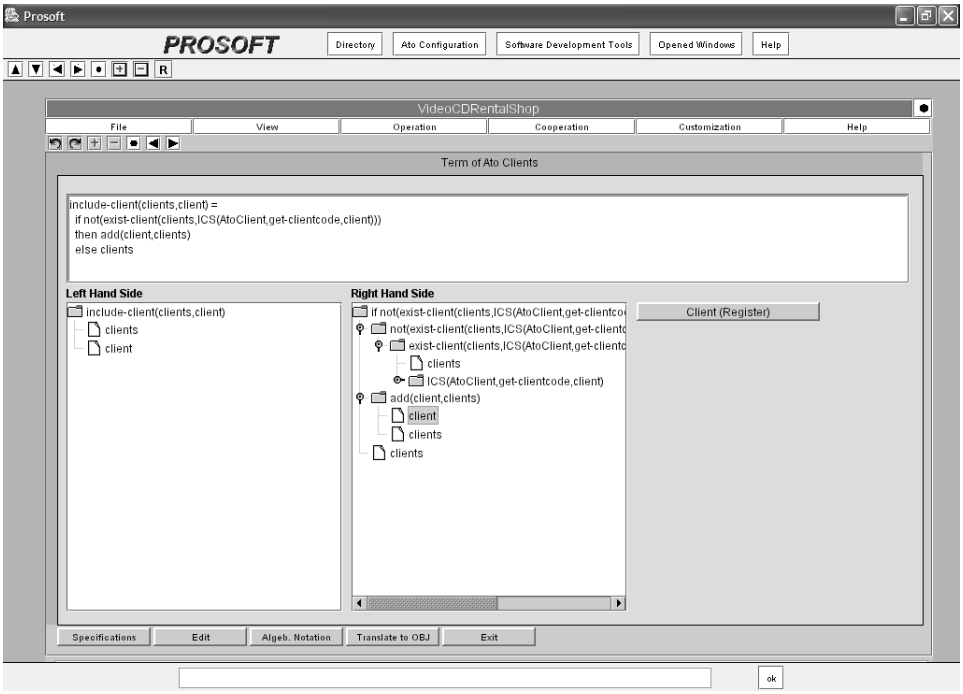


Figura 7.7: Edição dos axiomas

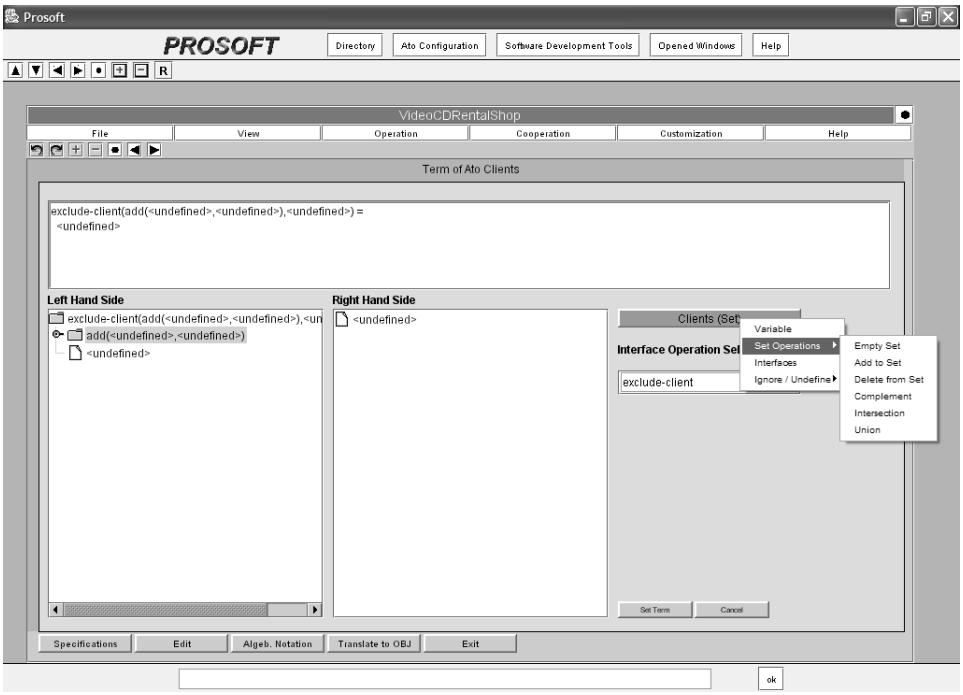


Figura 7.8: Definindo os axiomas

7.10 Código OBJ Resultante

A figura 7.10 mostra a tradução para OBJ, sendo que a qualquer momento o usuário pode solicitá-la. Assim como na visualização da notação textual do PROSOFT-algébrico, a tradução para OBJ pode considerar somente um ATO ou então todos os ATOs da especificação. Também existe a possibilidade de salvar a tradução em arquivo texto.

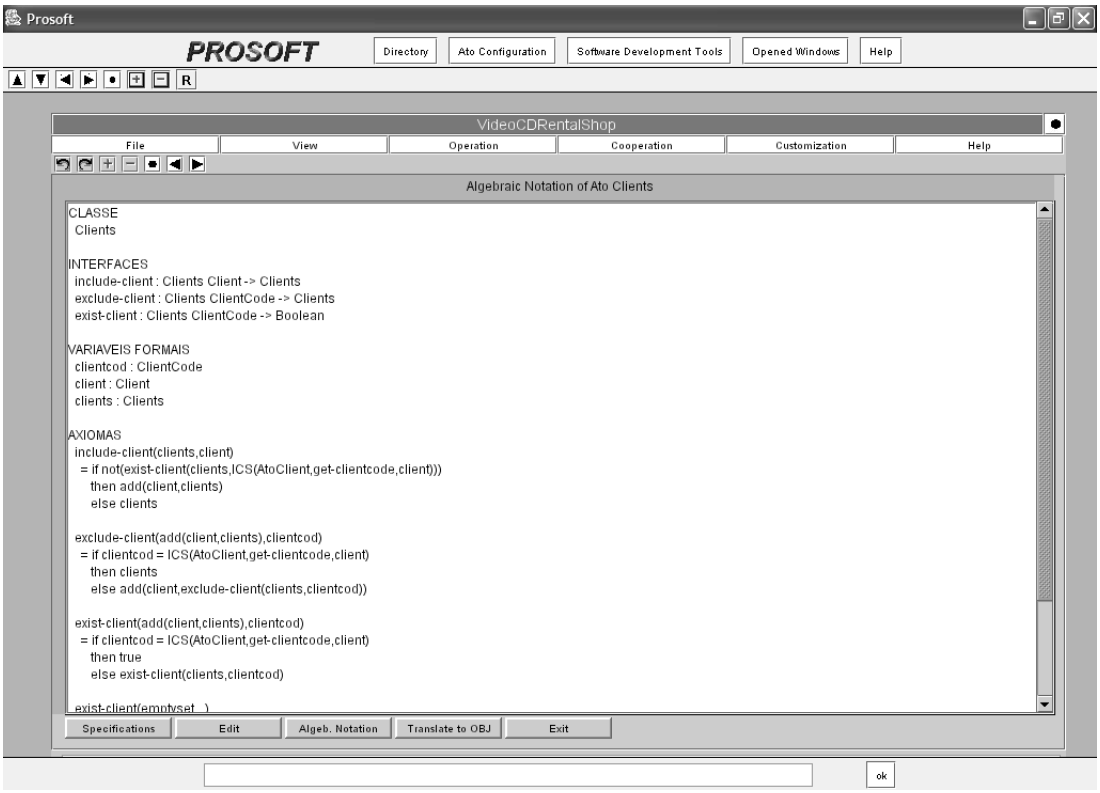


Figura 7.9: Formulário da representação textual do ATO

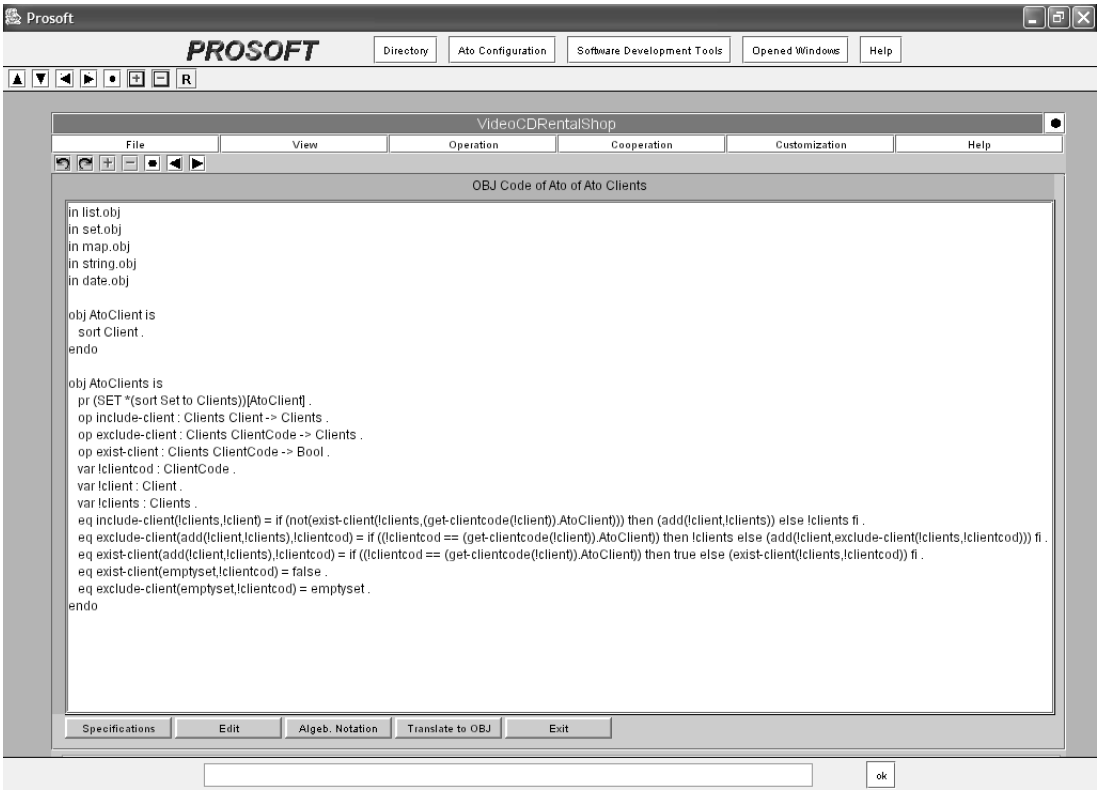


Figura 7.10: Código OBJ resultante da tradução

7.11 Uso do Ambiente OBJ3 para Reescrita de Termos

Como citado no início deste capítulo, não foi possível integrar os ambientes PROSOFT-Java e OBJ3. Portanto, esta seção traz algumas dicas para o usuário proceder a redução

de termos no OBJ3.

Primeiramente, deve-se traduzir a especificação editada no ProTool para OBJ e esta deve ser armazenada em um arquivo texto. Este arquivo conterá todos os objetos OBJ necessários para computar reduções de termos sobre a especificação.

Para executar o ambiente OBJ3, no Linux, deve-se entrar no diretório que o ambiente está instalado e copiar o arquivo que contém a tradução dos ATOs. Para chamar o OBJ3 digita-se na linha de comando

./obj3

e para carregar a tradução dos ATOs, digita-se:

in nome-do-arquivo.obj

O comando **in** faz o OBJ3 carregar todos os objetos deste arquivo no ambiente. Tendo carregado a especificação no OBJ3, então o usuário pode efetuar reduções de termos através do comando:

red *TERMO* .

O comando **red** é seguido pelo termo que será reduzido, seguido de um "ponto" que delimita o fim do comando. Este comando faz o ambiente OBJ3 reduzir o termo dado para sua forma normal, de acordo com a especificação carregada no ambiente.

Para a construção dos termos algébricos, segundo a notação PROSOFT-algébrico, o usuário deve seguir a teoria algébrica (WATT, 1991). O único detalhe que deve ser ressaltado diz respeito às chamadas ICS, que não existem em OBJ. Portanto o usuário deve criar seus termos algébricos na mesma notação especificada no PROSOFT, chamando diretamente a operação que a ICS solicita. Por exemplo: o termo do PROSOFT-algébrico

op1(. . . ,ICS(ATO2,op2,a,b,c),. . .)

em OBJ (sem a chamada ICS) é:

op1(. . . ,op2(a,b,c),. . .)

Através das reduções de termos, o usuário tem a possibilidade de validar se o que ele especificou é realmente uma solução para o problema que está sendo resolvido. A definição de uma metodologia para validação dos requisitos funcionais das especificações está fora do escopo deste trabalho.

7.12 Ferramentas Relacionadas

Nesta seção são apresentadas outras ferramentas para prototipação de tipos abstratos de dados baseadas em métodos formais. As ferramentas escolhidas para serem comparadas com o ProTool foram:

Larch Prover Larch Shared Language (LSL) (GUTTAG; HORNING, 1993) é uma linguagem para especificar propriedades de tipos abstratos de dados em lógica multisortida de 1^a ordem. O Larch Prover (LP), serve como um assistente de provas para raciocinar sobre especificações Larch e guiar as provas. LP é um assistente de provas interativo ao invés de um provador automático de teoremas. A linguagem Larch conta ainda com recursos de verificar inconsistências entre um programa e sua especificação, sendo que os programas podem ser escritos em ANSI C, CLU, Ada, Module-3, Smalltalk-80, C++, Standard ML, VHDL e Java.

RAISE Tools RAISE[3] (GEORGE, 1991) utiliza a notação *RAISE Specification Language* (RSL), inspirada em outras como VDM, CSP e Act-One. Além da notação formal, RAISE conta com um método para guiar as atividades mais relevantes do desenvolvimento de software, tais como: captura dos requisitos e gerenciamento de projeto. Este método é baseado no paradigma de refinamentos graduais e não é uma "receita de bolo", apenas fornece um *framework* e guias para o processo de desenvolvimento de software. O método RAISE permite aos usuários selecionar o nível de formalidade mais apropriado para circunstâncias particulares, padrões de projeto, etc.

CASL Tool Set (CATS) CASL (ASTESIANO et al., 2002) é uma linguagem de especificação formal desenvolvida pelo grupo de pesquisa CoFI. CASL nasceu como uma tentativa de padronizar o método algébrico, por isso ela incorpora muitas características de outros métodos algébricos. CATS é um conjunto integrado de ferramentas, como: parser, static checker, pretty printer para LaTeX e outras facilidades para visualizar assinaturas de especificações e as estruturas em grafo das especificações.

OBJ3 é o interpretador de especificações OBJ (GOGUEN et al., 2000). O ambiente OBJ3 é formado por uma implementação de um sistema de reescrita de termos muito eficiente.

Todas estas linguagens contam com algum tipo de suporte de ferramentas que auxiliam o desenvolvedor de software na tarefa de especificação e verificação. Estudos da agência espacial americana (NASA), resultaram numa taxonomia para ferramentas de suporte a métodos formais. Segundo esta taxonomia, apresentada em (CROW et al., 1995), uma ferramenta pode oferecer os seguintes componentes:

- Interface Gráfica: integra os componentes da ferramenta;
- Parser: verifica a consistência sintática de especificações e produz uma representação interna usada por outros componentes da ferramenta;
- Pretty-printer: traduz a representação interna da especificação em um formato padrão de visualização;
- Typechecker: verifica a consistência semântica da especificação;
- Prover (proof checker): realiza provas sobre uma especificação;
- Outros Recursos:
  - Browser: produz referências cruzadas e as gerencia. Particularmente útil para grandes especificações, possivelmente espalhadas sobre vários arquivos.
  - Status Recorder: mantém e relata o status da especificação, ou seja, se ela já passou pelo parser, pelo type checking, etc;
  - Pretty-printer: fornece formatação customizada para especificação e provas.

[3]RAISE é um acrônimo para *"Rigorous Approach Industrial Software Engineering"*.

7.12.1 Comparativo do ProTool com Outras Ferramentas

Na tabela 7.1, as ferramentas são comparadas de acordo com diversas funcionalidades, ou seja, se elas oferecem recursos para: editar a especificação dentro da ferramenta; verificar a sintaxe da especificação; type checking; provar teoremas; animar/executar especificação; interfaceamento gráfico com usuário; formatar especificação em algum formato; gerar código fonte em linguagem de programação. A última característica lista os sistemas operacionais em que cada ferramenta pode ser executada.

Tabela 7.1: Comparativo entre ferramentas de especificação formal

Característica	**Ferramenta**				
	LP	**RAISE**	**CATS**	**OBJ3**	**ProTool**
Editor	Não	Sim	Não	Não	Sim
Verificador de sintaxe	Sim	Sim	Sim	Sim	Sim
Type Checking	Sim	Sim	Sim	Sim	Sim
Provador de Teoremas	Sim	Sim	Externo	Externo	Externo
Animação/Execução	Não	Externo	Externo	Sim	Externo
Interface gráfica	Não	Não	Não	Não	Sim
Pretty-printer	Não	LaTeX e ASCII	LaTeX, Postscript, HMTL e PDF	Não	ASCII
Gerador de código	Não	Ada e C++	–	C++	Java
Sistema Operacional	Linux e Solaris	Windows e Linux	Linux e Solaris	Linux	Windows e Linux

Na característica animação/execução, somente LP não tinha um meio explícito de execução, apesar de ter um sistema de reescrita de termos para computar as provas. RAISE não é executável, mas em alguns casos a especificação pode ser traduzida automaticamente para uma linguagem imperativa. CATS não tinha nenhum meio próprio de execução. Para executar especificações em CASL, estas são traduzidas para serem utilizadas nos sistemas de reescrita ELAN (BOROVANSKY et al., 2000) e ASF+SDF (BRAND et al., 2001). OBJ3 possui um eficiente sistema de reescrita de termos. No ProTool, as especificações são executadas através da tradução para OBJ.

No quesito provador de teoremas, CATS não fornece nenhum provador próprio, mas as especificações podem ser exportadas para utilizar os provadores CASLtoPVS, HOL-CASL, INKA e KIV. Para provar teoremas na notação OBJ utilização o Kumo (GOGUEN et al., 2000), que não faz parte do ambiente OBJ3. No ProTool não é possível provar teoremas diretamente em PROSOFT-algébrico, mas sim utilizando a tradução para OBJ.

7.13 Gerador de Código Java do ADS PROSOFT

As seções anteriores mostraram a ferramenta ProTool, que edita e prototipa ATOs algébricos. Através da execução dos ATOs algébricos no ambiente OBJ3, é possível efetuar verificações sobre a especificação para validá-la de acordo com os requisitos informais do software em desenvolvimento.

Uma vez que a especificação do ATO algébrico esteja bem definida, pode-se utilizar uma ferramenta que gera código Java que implementa o ATO algébrico dentro do ADS PROSOFT-Java.

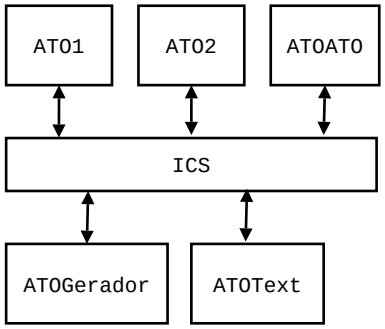


Figura 7.11: Exemplo de estrutura do ambiente PROSOFT-Java

Considere o exemplo da figura 7.11. Neste exemplo o ambiente PROSOFT-Java é composto por quatro ATOs-Java implementados:

ATO1 – ATO qualquer;

ATO2 – ATO qualquer;

ATOATO – editor de ATOs algébricos;

ATOGerador – ferramenta que implementa um ATO algébrico em Java;

ATOText – ferramenta que define uma lista de strings (texto) e suas respectivas operações para manipulação de texto.

Se o usuário do ambiente deseja criar uma especificação de um editor de código fonte C, por exemplo, um novo ATO algébrico deve ser criado no ProTool de modo que especifique os tipos de dados necessários e as respectivas operações. Um exemplo de operação do editor C é a que identifica os construtores utilizados na linguagem de programação C.

Para especificar o novo ATO algébrico que define o editor em C é utilizada a ferramenta ProTool que está no `ATOATO`. Este novo ATO é chamado de `ATOEditorC`, que utiliza recursos de um ATO já existente no ambiente, o `ATOText`. Até então foi criado o ATO algébrico`ATOEditorC`. Para implementar este editor C dentro do ambiente PROSOFT-Java é utilizada a ferramenta `ATOGerador`, conforme ilustrado na figura 7.12.

O processo descrito na figura 7.12 inicia com a geração de código Java que implementa a classe algébrica do `ATOEditorC`. O próximo passo é dado por um processo manual que consiste em implementar as operações definidas no ATO e a interface gráfica[4] no ambiente PROSOFT. Portanto, o resultado do processo *Implementa ops do usuário e interface gráfica* é um código fonte em Java que implementa o `ATOEditorC` algébrico. Cabe ressaltar que o `ATOGerador`, responsável pela codificação do ATO-Java, também poderia implementar as operações do ATO, mas a versão atual deste gerador não fornece este recurso.

[4] A interface gráfica de um ATO-Java é a tela que serve para interação com o usuário final da ferramenta.

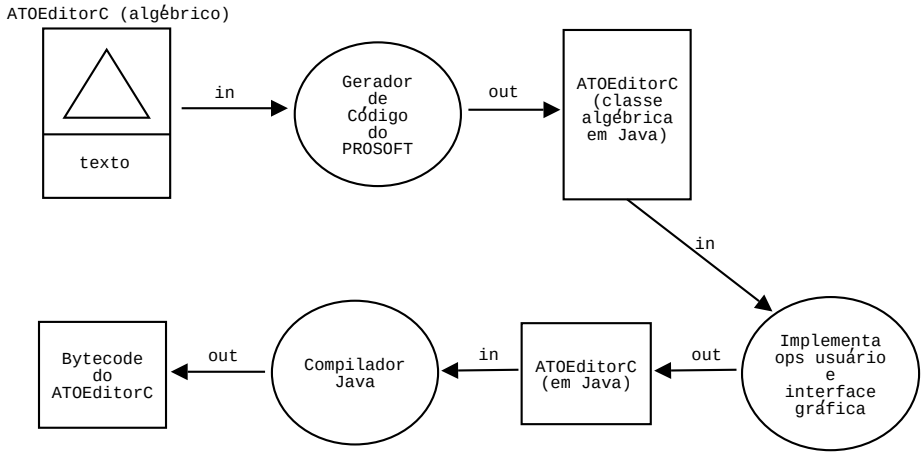


Figura 7.12: Processo de inclusão do novo ATO-Java no ambiente PROSOFT

A partir de então, o usuário precisa compilar o código fonte Java utilizando um compilador Java, produzindo o bytecode[5] deste ATO. Na figura o processo está simplificado para melhorar o entendimento, pois o processo *Compilador Java* deve não somente compilar o código fonte do ATO, mas também recompilar todos os arquivos fontes do ambiente PROSOFT para então obter o novo ambiente com a ferramenta `ATOEditorC` implementada dentro dele, como mostrado na figura 7.13.

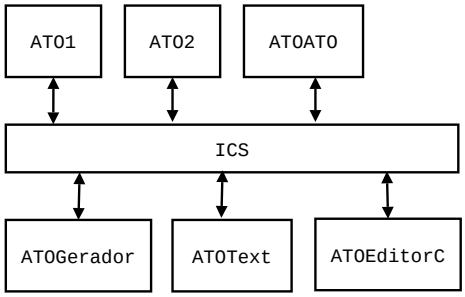


Figura 7.13: Estrutura do PROSOFT-Java depois da implementação do `ATOEditorC`

7.14 Considerações Finais

Neste capítulo foi apresentada a implementação da ferramenta ProTool. Esta implementação consiste em editor de ATOs algébricos e o correspondente tradutor para OBJ. Foi mostrado, brevemente, como o usuário deve interagir tanto com o ProTool quanto com o OBJ3 para validar as especificações dos ATOs.

Por fim, foi apresentado o gerador de código Java, já existente no PROSOFT-Java, que é um instrumento muito útil para gerar o esqueleto da implementação (em Java) do ATO algébrico. Uma visão mais detalhada sobre o processo de codificação de um ATO-Java pode ser encontrada em (REIS, 2003a).

No próximo capítulo são apresentados os estudos de caso que foram usados para validação da ferramenta ProTool.

[5]Bytecode é o código gerado pelo compilador Java que é capaz de ser executado por qualquer máquina virtual Java.

8 ESTUDOS DE CASO

Esse capítulo descreve o uso prático da ferramenta ProTool em situações que envolvem a construção de especificações de software para problemas reais. Esses estudos de casos foram conduzidos com o objetivo de avaliar tanto o modelo de prototipação proposto quanto as operações para edição dos ATOs algébricos.

Foram escolhidos dois estudos de caso para validar o modelo ProTool. O primeiro é um sistema de locadora de CDs e DVDs que tem como premissa básica o fácil entendimento de suas funcionalidades devido à baixa complexidade dos tipos de dados e suas operações.

O segundo estudo de caso é a prototipação da especificação do próprio ProTool, que serviu para demonstrar seu funcionamento e validar as especificações dos ATOs descritos no capítulo 6 antes da implementação no ADS PROSOFT-Java.

Em virtude da documentação completa gerada pelas reduções de termos sobre as especificações ocupar um número considerável de páginas, esse capítulo descreve resumidamente algumas das principais experimentações.

8.1 Estudo de Caso 1: Locadora de CDs e DVDs

Para exemplificar é apresentado nesta seção um exemplo completo de um sistema especificado de acordo com o paradigma do Prosoft. O exemplo apresentado a seguir apresenta a especificação do controle de uma videlocadora, armazenando informações acerca dos empréstimos dos discos (CDs e DVDs) para os clientes.

8.1.1 Descrição Informal

O sistema de videolocadora especificado objetiva mostrar as principais características de um sistema de locadora de CDs e DVDs. Para facilitar o entendimento, este sistema, é obviamente uma versão simplificada de um sistema real de locadora. Uma locadora é composta basicamente dos seguintes cadastros:

Clientes – armazena dados pessoais do cliente, como: nome do cliente, endereço e telefone de contato;

Info Media – armazena as informações acerca dos CDs e DVDs, como: título, lista de atores (DVD), lista de músicas (CD), gênero e ano de lançamento;

Discos – armazena informações acerca dos discos que a locadora têm disponível no acervo. Um dos dados do disco é o valor da diária cobrada pela locação. Como por exemplo: existem quatro discos do DVD “Matrix Reloaded” e sua diária é 3 reais;

Empréstimos – armazena os empréstimos efetuados pelos clientes, ou seja, qual disco foi locado por um determinado cliente.

Baseado nos cadastros acima, deve-se oferecer operações que possibilitem o funcionamento da locadora. Abaixo é apresentada uma descrição em alto nível para cada operação:

- **include-client**: inclui o cadastro de cliente;
- **exclude-client**: exclui o cadastro de cliente;
- **include-media**: inclui o cadastro com os dados de um CD/DVD;
- **add-track**: insere uma música no cadastro de um CD;
- **add-actor**: insere um ator no cadastro de um DVD;
- **exclude-media**: exclui o cadastro de uma media (CD ou DVD);
- **include-disc**: inclui o cadastro de um disco para um determinado CD/DVD;
- **exclude-disc**: exclui o cadstro de um disco para um determinado CD/DVD;
- **lend-disc**: efetua o empréstimo de um disco (CD ou DVD);
- **give-disc-back**: efetua a devolução de um disco;
- **calculate-bill**: calcula o valor a ser pago pela locação.

Abaixo é descrito o empréstimo de um disco para um cliente, desde a fase que o cliente retira o disco, até quando ele retorna o disco á locadora e efetua o respectivo pagamento.

1. O cliente retira o DVD “A” e o funcionário da locadora efetua a entrada desta locação no sistema. Entre os dados armazenados estão o código do cliente, o código do disco locado e o dia da retirada. A operação responsável por este passo é a **lend-disc**.
2. Para devolver um disco locado, o funcionário entra no sistema somente com o código do disco. Este código localiza os dados do empréstimo para ser calculado o número de dias que o cliente ficou com o disco. Este número de dias locados é multiplicado pelo valor da diária (informação do disco) e resulta o valor que deve ser pago pelo cliente. A operação que calcula este pagamento é a **calculate-bill**.
3. Por fim, o funcionário efetua a baixa do empréstimo do disco, que é feita através do código do disco fornecido no passo 2. A operação responsável por esta funcionalidade é a **give-disc-back**.

Baseado na descrição informal mostrada acima, foram criados ATOs algébricos que especificam os tipos de dados e operações necessárias para o funcionamento da locadora proposta. Estes ATOs são:

ATOVCDRShop – neste ATO são integrados todos os ATOs abaixo, criando a especificação completa do sistema da locadora;

ATOClients – conjunto de todos os clientes da locadora;

ATOClient – cadastro do cliente;

ATODiscs – cadastro dos discos existentes na locadora;

ATORentals – informações acerca dos empréstimos efetuados na locadora;

ATOMediaSet – neste ATO são armazenados em um conjunto as informações de todos os CDs e DVDs da locadora;

ATOMedia – cadastro das informações de um CD ou DVD, como o título, gênreo, etc.

A seguir são mostradas as especificações dos ATOs citados acima.

8.1.2 ATOVCDRShop

O ATO principal do sistema da locadora é o ATOVCDRShop (**V**ideo**CDR**ental**S**hop), que possui como componentes instâncias das classes Clients, Rentals, MediaSet e Discs.

CLASS

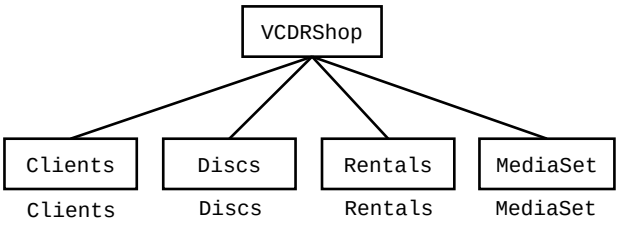


Figura 8.1: Definição da classe do ATOVCDRShop

INCLUDE

INTERFACES

```
include-client  :  VCDRShop Client                      ->  VCDRShop
exclude-client  :  VCDRShop String                      ->  VCDRShop
include-media   :  VCDRShop Media                       ->  VCDRShop
add-track       :  VCDRShop String String               ->  VCDRShop
add-actor       :  VCDRShop String String               ->  VCDRShop
exclude-media   :  VCDRShop String                      ->  VCDRShop
include-disc    :  VCDRShop String String Date Integer  ->  VCDRShop
exclude-disc    :  VCDRShop String                      ->  VCDRShop
lend-disc       :  VCDRShop String String Date          ->  VCDRShop
give-disc-back  :  VCDRShop String                      ->  VCDRShop
calc-days       :  Date Date                            ->  Integer
calculate-bill  :  VCDRShop String Date                 ->  Integer
```

FORMAL VARIABLES

```
c    :  Client
cs   :  Clients
ren  :  Rentals
```

```
ccod mcod tr ac dcod mcod  :  String
ms                         :  MediaSet
m                          :  Media
ds                         :  Discs
pri                        :  Integer
today                      :  Date
pdate d1 d2                :  Date
```

AXIOMS

```
include-client(reg-VCDRShop(cs,_,_,_),c)
= reg-VCDRShop(ICS(ATOClients,include-client,cs,c),_,_,_)

exclude-client(reg-VCDRShop(cs,_,ren,_),ccod)
= if ICS(ATORentals,client-has-disc,ren,ccod)
      then reg-VCDRShop(cs,_,ren,_)
      else reg-VCDRShop(ICS(ATOClients,exclude-client,cs,ccod),
             _,ren,_)

include-media(reg-VCDRShop(_,_,_,ms),m)
= reg-VCDRShop(_,_,_,ICS(ATOMediaSet,include-media,ms,m))

add-track(reg-VCDRShop(_,_,_,ms),mcod,tr)
= reg-VCDRShop(_,_,_,ICS(ATOMediaSet,add-track,ms,mcod,tr))

add-actor(reg-VCDRShop(_,_,_,ms),mcod,ac)
= reg-VCDRShop(_,_,_,ICS(ATOMediaSet,add-actor,ms,mcod,ac))

exclude-media(reg-VCDRShop(_,ds,_,ms),mcod)
= if ICS(ATODiscs,exist-disc-media,ds,mcod)
     then reg-VCDRShop(_,ds,_,ms)
     else reg-VCDRShop(_,ds,_,ICS(ATOMediaSet,exclude-media,ms,
             mcod))

include-disc(reg-VCDRShop(_,ds,_,ms),dcod,mcod,pdate,pri)
= if ICS(ATOMediaSet,exist-media,ms,mcod)
     then reg-VCDRShop(_,ICS(ATODiscs,include-disc,ds,dcod,mcod,
            pdate,pri),_,ms)
     else reg-VCDRShop(_,ds,_,ms)

exclude-disc(reg-VCDRShop(_,ds,ren,_),dcod)
= if dcod belongsto domain(ren)
     then reg-VCDRShop(_,ds,ren,_)
     else reg-VCDRShop(_,ICS(ATODiscs,exclude-disc,ds,dcod),ren,
              _)

lend-disc(reg-VCDRShop(_,_,ren,_),dcod,ccod,today)
= reg-VCDRShop(_,_,ICS(ATORentals,lend-disc,ren,dcod,ccod,today)
   ,_)

give-disc-back(reg-VCDRShop(_,_,ren,_),dcod)
= reg-VCDRShop(_,_,ICS(ATORentals,give-disc-back,ren,dcod),_)
```

```
calc-days(d1,d2)
= toint(d2) - toint(d1)

calculate-bill(reg-VCDRShop(_,ds,ren,_),dcod,today)
= ICS(ATODiscs,get-price,ds,dcod) *
        calc-days(ICS(ATORentals,get-loandate,ren,dcod),today)
```

Código OBJ resultante da tradução:

```
obj ATOVCDRShop is
 pr (REGISTER4 *(sort Register to VCDRShop,op reg to reg-VCDRShop,
      op select1 to select-Clients,op select2 to select-Discs,
      op select3 to select-Rentals,op select4 to select-MediaSet))
      [ATOClients,ATODiscs,ATORentals,ATOMediaSet] .
 op include-client : VCDRShop Client -> VCDRShop .
 op exclude-client : VCDRShop String -> VCDRShop .
 op add-track : VCDRShop String String -> VCDRShop .
 op add-actor : VCDRShop String String -> VCDRShop .
 op include-media : VCDRShop Media -> VCDRShop .
 op exclude-media : VCDRShop String -> VCDRShop .
 op include-disc : VCDRShop String String Date Int -> VCDRShop .
 op exclude-disc : VCDRShop String -> VCDRShop .
 op lend-disc : VCDRShop String String Date -> VCDRShop .
 op give-disc-back : VCDRShop String -> VCDRShop .
 op calculate-bill : VCDRShop String Date -> Int .
 op calc-days : Date Date -> Int .
 var !c : Client .
 var !cs : Clients .
 var !ren : Rentals .
 var !ccod : String .
 var !ms : MediaSet .
 var !m : Media .
 var !mcod : String .
 var !tr : String .
 var !ac : String .
 var !ds : Discs .
 var !dcod : String .
 var !mcod : String .
 var !pdate : Date .
 var !pri : Int .
 var !today : Date .
 var !d1 : Date .
 var !d2 : Date .
 var !Discs : Discs .
 var !Rentals : Rentals .
 var !MediaSet : MediaSet .
 var !Clients : Clients .
 eq include-client(reg-VCDRShop(!cs,!Discs,!Rentals,!MediaSet),!c)
= reg-VCDRShop(include-client(!cs,!c),!Discs,!Rentals,!MediaSet) .
 eq exclude-client(reg-VCDRShop(!cs,!Discs,!ren,!MediaSet),!ccod)
= if client-has-disc(!ren,!ccod) then reg-VCDRShop(!cs,!Discs,!ren,
```

```
!MediaSet) else reg-VCDRShop(exclude-client(!cs,!ccod),!Discs,!ren,
!MediaSet) fi .
 eq add-track(reg-VCDRShop(!Clients,!Discs,!Rentals,!ms),!mcod,!tr)
= reg-VCDRShop(!Clients,!Discs,!Rentals,add-actor(!ms,!mcod,!tr)) .
 eq add-actor(reg-VCDRShop(!Clients,!Discs,!Rentals,!ms),!mcod,!ac)
= reg-VCDRShop(!Clients,!Discs,!Rentals,add-actor(!ms,!mcod,!ac)) .
 eq include-media(reg-VCDRShop(!Clients,!Discs,!Rentals,!ms),!m)
= reg-VCDRShop(!Clients,!Discs,!Rentals,include-media(!ms,!m)) .
 eq exclude-media(reg-VCDRShop(!Clients,!ds,!Rentals,!ms),!mcod)
= if exist-disc-media(!ds,!mcod) then reg-VCDRShop(!Clients,!ds,
!Rentals,!ms) else reg-VCDRShop(!Clients,!ds,!Rentals,exclude-media(
!ms,!mcod)) fi .
 eq include-disc(reg-VCDRShop(!Clients,!ds,!Rentals,!ms),!dcod,!mcod,
!pdate,!pri) = if (exist-media(!ms,!mcod)).ATOMediaSet then
reg-VCDRShop(!Clients,include-disc(!ds,!dcod,!mcod,!pdate,!pri),
!Rentals,!ms) else reg-VCDRShop(!Clients,!ds,!Rentals,!ms) fi .
 eq exclude-disc(reg-VCDRShop(!Clients,!ds,!ren,!MediaSet),!dcod)
= if (!dcod belongsto domain(!ren)) then reg-VCDRShop(!Clients,!ds,
!ren,!MediaSet) else reg-VCDRShop(!Clients,exclude-disc(!ds,!dcod),
!ren,!MediaSet) fi .
 eq lend-disc(reg-VCDRShop(!Clients,!Discs,!ren,!MediaSet),!dcod,
 !ccod,!today) = reg-VCDRShop(!Clients,!Discs,lend-disc(!ren,!dcod,
 !ccod,!today),!MediaSet) .
 eq give-disc-back(reg-VCDRShop(!Clients,!Discs,!ren,!MediaSet),
!dcod) = reg-VCDRShop(!Clients,!Discs,give-disc-back(!ren,!dcod),
!MediaSet) .
 eq calc-days(!d1,!d2) = toint(!d2) - toint(!d1) .
 eq calculate-bill(reg-VCDRShop(!Clients,!ds,!ren,!MediaSet),!dcod,
!today) = get-price(!ds,!dcod) * calc-days(get-loandate(!ren,!dcod),
!today) .
endo
```

8.1.3 ATOClients

CLASS

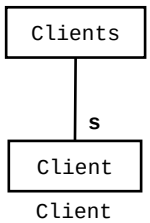


Figura 8.2: Definição da classe do ATOClients

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **include-client**: inclui um cadastro de cliente;
- **exclude-client**: exclui um cadastro de cliente;
- **exist-client**: verifica se um cliente já possui cadastro.

INCLUDE

```
   Boolean
```

INTERFACES

```
  include-client  :  Clients Client  ->  Clients
  exclude-client  :  Clients String  ->  Clients
  exist-client    :  Clients String  ->  Boolean
```

FORMAL VARIABLES

```
  clients    :  Clients
  client     :  Client
  clientcod  :  String
```

AXIOMS

```
include-client(clients,client)
= if not(exist-client(clients,ICS(Client,get-clientcode,client)))
       then add(client,clients)
       else clients

exclude-client(add(client,clients),clientcod)
= if clientcod == ICS(Client,get-clientcode,client)
       then clients
       else add(client,exclude-client(clients,clientcod))

exclude-client(emptyset,_) = emptyset

exist-client(add(client,clients),clientcod)
= if clientcod == ICS(Client,get-clientcode,client)
        then true
        else exist-client(clients,clientcod)

exist-client(emptyset,_) = false
```

Código OBJ resultante da tradução:

```
obj ATOClients is
 pr (SET *(sort Set to Clients))[ATOClient] .
 op include-client : Clients Client -> Clients .
 op exclude-client : Clients String -> Clients .
 op exist-client : Clients String -> Bool .
 var clients : Clients .
 var client : Client .
 var clientcod : String .
 var !String : String .
 eq include-client(clients,client) = if not(exist-client(clients,
get-clientcode(client))) then add(client,clients) else clients fi .
 eq exclude-client(add(client,clients),clientcod) = if (clientcod
== get-clientcode(client)) then clients else add(client,
exclude-client(clients,clientcod)) fi .
 eq exclude-client(emptyset,!String) = emptyset .
 eq exist-client(emptyset,!String) = false .
```

```
 eq exist-client(add(client,clients),clientcod) = if clientcod ==
get-clientcode(client) then true else exist-client(clients,
clientcod) fi .
endo
```

8.1.4 ATOClient

Este ATO possui somente uma operação definida pelo usuário que retorna o código de cadastro de um cliente.

CLASS

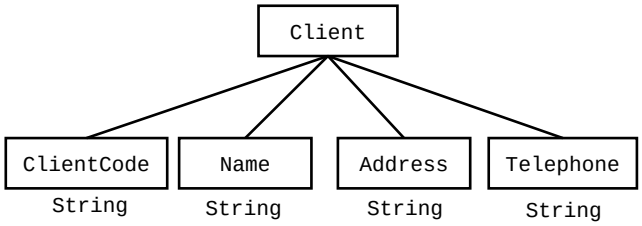


Figura 8.3: Definição da classe do ATOClient

INCLUDE

INTERFACES

```
  get-clientcode  :  Client  ->  String
```

FORMAL VARIABLES

```
  cli  :  Client
```

AXIOMS

```
get-clientcode(cli) = select-ClientCode(cli)
```

Código OBJ resultante da tradução:

```
obj ATOClient is
 pr (REGISTER4 *(sort Register to Client,op reg to reg-Client,
     op select1 to select-ClientCode,op select2 to select-Name,
     op select3 to select-Telephone))[STRING,STRING,STRING,STRING] .
 op get-clientcode : Client -> String .
 var cli : Client .
 eq get-clientcode(cli) = select-ClientCode(cli) .
endo
```

8.1.5 ATODiscs

CLASS

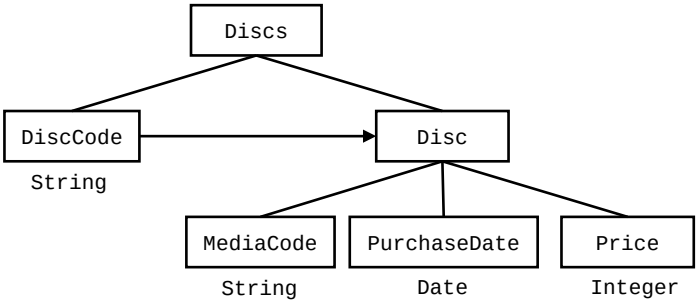


Figura 8.4: Definição da classe do ATODiscs

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **include-disc**: insere o cadastro de um disco;
- **exclude-disc**: exclui o cadastro de um disco;
- **exist-disc-media**: verifica se existe pelo menos um disco de um determinado CD/DVD;
- **get-price**: retorna o valor da diária do disco.

INCLUDE

```
Boolean
```

INTERFACES

```
include-disc      :  Discs String String Date Integer  ->  Discs
exclude-disc      :  Discs String                      ->  Discs
exist-disc-media  :  Discs String                      ->  Boolean
get-price         :  Discs String                      ->  Integer
```

FORMAL VARIABLES

```
ds                                 :  Discs
codnewdisc mediacod disccod mcod   :  String
pdate                              :  Date
pr                                 :  Integer
```

AXIOMS

```
include-disc(ds,codnewdisc,mediacod,pdate,pr)
= if not(exist-disc-media(ds,codnewdisc))
        then modify(codnewdisc,reg-Disc(mediacod,pdate,pr),ds)
        else ds

exclude-disc(ds,disccod)
= if exist-disc-media(ds,disccod)
       then restrictwith(ds,add(disccod,emptyset))
       else ds
```

```
exist-disc-media(modify(disccod,reg-Disc(mcod,_,_),ds),mediacod)
= if mcod == mediacod
     then true
     else exist-disc-media(ds,mediacod)

exist-disc-media(emptymap,_) = false

get-price(ds,disccod)
= select-Price(imageof(disccod,restrictto(ds,add(disccod,emptyset))))
```

Código OBJ resultante da tradução:

```
obj ATODiscs is
 pr (MAP *(sort Map to Discs))[STRING,(REGISTER3 *(sort Register
     to Disc,op reg to reg-Disc,op select1 to select-MediaCode,
     op select2 to select-PurchaseDate,op select3 to select-Price))
     [STRING,DATE,INT]] .
 op include-disc : Discs String String Date Int -> Discs .
 op exclude-disc : Discs String -> Discs .
 op exist-disc-media : Discs String -> Bool .
 op get-price : Discs String -> Int .
 var !ds : Discs .
 var !codnewdisc : String .
 var !mediacod : String .
 var !pdate : Date .
 var !pr : Int .
 var !disccod : String .
 var !mcod : String .
 var !PurchaseDate : Date .
 var !Price : Int .
 var !String : String .
 eq include-disc(!ds,!codnewdisc,!mediacod,!pdate,!pr) = if not(
exist-disc-media(!ds,!codnewdisc)) then modify(!codnewdisc,
reg-Disc(!mediacod,!pdate,!pr),!ds) else !ds fi .
 eq exclude-disc(!ds,!disccod) = if exist-disc-media(!ds,!disccod)
then restrictwith(!ds,add(!disccod,emptyset)) else !ds fi .
 eq exist-disc-media(modify(!disccod,reg-Disc(!mcod,!PurchaseDate,
!Price),!ds),!mediacod) = if (!mcod == !mediacod) then true else
exist-disc-media(!ds,!mediacod) fi .
 eq exist-disc-media(emptymap,!String) = false .
 eq get-price(!ds,!disccod) = select-Price(imageof(!disccod,
restrictto(!ds,add(!disccod,emptyset)))) .
endo
```

8.1.6 ATORentals

CLASS

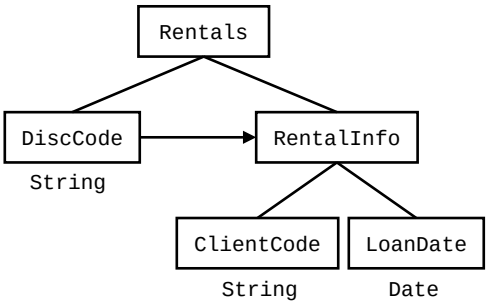


Figura 8.5: Definição da classe do ATORentals

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **lend-disc**: efetua o empréstimo de um disco (CD ou DVD);
- **give-disc-back**: efetua o retorno do disco à locadora;
- **is-lended**: verifica se um disco está emprestado;
- **client-has-disc**: verifica se um determinado cliente tem algum disco locado;
- **get-loandate**: retorna a data que um disco foi locado.

INCLUDE

```
Boolean
```

INTERFACES

```
lend-disc        :  Rentals String String Date  ->  Rentals
give-disc-back   :  Rentals String              ->  Rentals
is-lended        :  Rentals String              ->  Boolean
client-has-disc  :  Rentals String              ->  Boolean
get-loandate     :  Rentals String              ->  Date
```

FORMAL VARIABLES

```
rentals                      :  Rentals
disccod clientcod ccod dcod  :  String
d1 today                     :  Date
```

AXIOMS

```
lend-disc(rentals,disccod,clientcod,today)
= if not(is-lended(rentals,disccod))
    then modify(disccod,reg-RentalInfo(clientcod,today),rentals)
    else rentals

give-disc-back(rentals,disccod)
= if is-lended(rentals,disccod)
    then restrictwith(rentals,add(disccod,emptyset))
```

```
    else rentals

is-lended(rentals,disccod)
= disccod belongsto domain(rentals)

client-has-disc(modify(disccod,reg-RentalInfo(ccod,_),rentals),
  clientcod)
= if ccod == clientcod
    then   true
    else   client-has-disc(rentals,clientcod)

client-has-disc(emptymap,_) = false

get-loandate(modify(disccod,reg-RentalInfo(_,d1),rentals),dcod)
= if dcod == disccod
    then d1
    else get-loandate(rentals,dcod)
```

Código OBJ resultante da tradução:

```
obj ATORentals is
 pr (MAP *(sort Map to Rentals))[STRING,(REGISTER2 *(sort Register
      to RentalInfo,op reg to reg-RentalInfo,op select1 to
      select-ClientCode,op select2 to select-LoanDate))
      [STRING,DATE]] .
 op lend-disc : Rentals String String Date -> Rentals .
 op give-disc-back : Rentals String -> Rentals .
 op is-lended : Rentals String -> Bool .
 op client-has-disc : Rentals String -> Bool .
 op get-loandate : Rentals String -> Date .
 var !rentals : Rentals .
 var !disccod : String .
 var !clientcod : String .
 var !d1 : Date .
 var !today : Date .
 var !ccod : String .
 var !dcod : String .
 var !LoanDate : Date .
 var !String : String .
 var !ClientCode : String .
 eq lend-disc(!rentals,!disccod,!clientcod,!today) = if not(is-lended(
!rentals,!disccod)) then modify(!disccod,reg-RentalInfo(!clientcod,
!today),!rentals) else !rentals fi .
 eq give-disc-back(!rentals,!disccod) = if is-lended(!rentals,!disccod)
then restrictwith(!rentals,add(!disccod,emptyset)) else !rentals fi .
 eq is-lended(!rentals,!disccod) = !disccod belongsto domain(!rentals) .
 eq client-has-disc(modify(!disccod,reg-RentalInfo(!ccod,!LoanDate),
!rentals),!clientcod) = if (!ccod == !clientcod) then true else
client-has-disc(!rentals,!clientcod) fi .
 eq client-has-disc(emptymap,!String) = false .
```

```
 eq get-loandate(modify(!disccod,reg-RentalInfo(!ClientCode,!d1),
!rentals),!dcod) = if (!dcod == !disccod) then !d1 else get-loandate(
!rentals,!dcod) fi .
endo
```

8.1.7 ATOMediaSet

CLASS

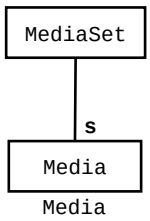


Figura 8.6: Definição da classe do ATOMediaSet

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **include-media**: insere o cadastro com os dados de um CD (ou DVD);
- **exclude-media**: exclui o cadastro de um CD (ou DVD);
- **exist-media**: verifica se existe cadastro co CD/DVD;
- **add-track**: insere uma música no cadastro do CD;
- **add-actor**: insere um ator no cadastro do DVD.

INCLUDE

```
Boolean
```

INTERFACES

```
include-media  :  MediaSet Media          ->  MediaSet
exclude-media  :  MediaSet String         ->  MediaSet
exist-media    :  MediaSet String         ->  Boolean
add-track      :  MediaSet String String  ->  MediaSet
add-actor      :  MediaSet String String  ->  MediaSet
```

FORMAL VARIABLES

```
mediaset                 :  MediaSet
media newmedia           :  Media
tr-name ac-name mediacod :  String
```

AXIOMS

```
include-media(mediaset,newmedia)
= if not(exist-media(mediaset,ICS(ATOMedia,get-mediacode,newmedia)))
    then add(newmedia,mediaset)
    else mediaset
```

```
exclude-media(add(media,mediaset),mediacod)
= if ICS(ATOMedia,get-mediacode,media) == mediacod
    then mediaset
    else add(media,exclude-media(mediaset,mediacod))

exclude-media(emptyset,_) = emptyset

exist-media(add(media,mediaset),mediacod)
= if ICS(ATOMedia,get-mediacode,media) = mediacod
    then true
    else exist-media(mediaset,mediacod)

exist-media(emptyset,_) = false

add-track(add(media,mediaset),mediacod,tr-name)
= if ICS(ATOMedia,get-mediacode,media) == mediacod
    then add(ICS(ATOMedia,include-track,media,tr-name),mediaset)
    else add(media,add-track(mediaset,mediacod,tr-name))

add-actor(add(media,mediaset),mediacod,ac-name)
= if ICS(ATOMedia,get-mediacode,media) == mediacod
    then add(ICS(ATOMedia,include-actor,media,ac-name),mediaset)
    else add(media,add-actor(mediaset,mediacod,ac-name))
```

Código OBJ resultante da tradução:

```
obj ATOMediaSet is
 pr (SET *(sort Set to MediaSet))[ATOMedia] .
 op include-media : MediaSet Media -> MediaSet .
 op exclude-media : MediaSet String -> MediaSet .
 op exist-media : MediaSet String -> Bool .
 op add-track : MediaSet String String -> MediaSet .
 op add-actor : MediaSet String String -> MediaSet .
 var !mediaset : MediaSet .
 var !newmedia : Media .
 var !mediacod : String .
 var !tr-name : String .
 var !ac-name : String .
 var !media : Media .
 var !String : String .
 eq include-media(!mediaset,!newmedia) = if not(exist-media(!mediaset,
get-mediacode(!newmedia))) then add(!newmedia,!mediaset) else
!mediaset fi .
 eq exclude-media(add(!media,!mediaset),!mediacod) = if (get-mediacode(
!media) == !mediacod) then !mediaset else add(!media,exclude-media(
!mediaset,!mediacod)) fi .
 eq exclude-media(emptyset,!String) = emptyset .
 eq exist-media(add(!media,!mediaset),!mediacod) = if (get-mediacode(
!media) == !mediacod) then true else exist-media(!mediaset,!mediacod) fi .
 eq exist-media(emptyset,!String) = false .
 eq add-track(add(!media,!mediaset),!mediacod,!tr-name)
= if (get-mediacode(!media) == !mediacod) then add(include-track(!media,
```

```
!tr-name),!mediaset) else add(!media,add-track(!mediaset,!mediacod,
!tr-name)) fi .
 eq add-actor(add(!media,!mediaset),!mediacod,!ac-name)
= if ((get-mediacode(!media)).ATOMedia == !mediacod) then add(
include-actor(!media,!ac-name),!mediaset) else add(!media,add-actor(
!mediaset,!mediacod,!ac-name)) fi .
endo
```

8.1.8 ATOMedia

CLASS

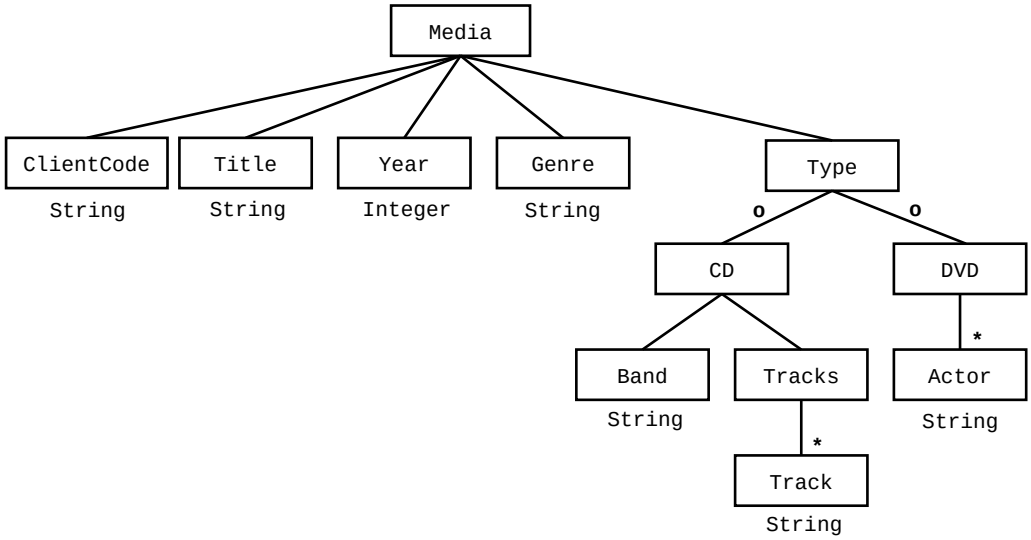


Figura 8.7: Definição da classe do ATOMedia

Abaixo é apresentada uma descrição de alto nível para cada operação deste ATO:

- **include-track**: insere uma música no cadastro de um CD;
- **include-actor**: insere um ator no cadastro de um DVD;
- **get-mediatitle**: retorna o título do CD/DVD;
- **get-mediacode**: retorna o código de um CD/DVD.

INCLUDE

INTERFACES

```
  include-track   :  Media String  ->  Media
  include-actor   :  Media String  ->  Media
  get-mediatitle  :  Media         ->  String
  get-mediacode   :  Media         ->  String
```

FORMAL VARIABLES

```
  c ac-name ti tr-name  :  String
  y                     :  Integer
  tr                    :  Tracks
  ac                    :  DVD
```

AXIOMS

```
include-track(reg-Media(_,_,_,_,Type-CD(reg-CD(b,tr))),tr-name)
= reg-Media(_,_,_,_,Type-CD(reg-CD(b,cons(tr-name,tr))))

include-actor(reg-Media(c,ti,y,g,Type-DVD(ac)),ac-name)
= reg-Media(c,ti,y,g,Type-DVD(cons(ac-name,ac)))

get-mediacode(reg-Media(c,_,_,_,_)) = c

get-mediatitle(reg-Media(_,ti,_,_,_)) = ti
```

Código OBJ resultante da tradução:

```
obj ATOMedia is
 pr (REGISTER5 *(sort Register to Media,op reg to reg-Media,
      op select1 to select-MediaCode,op select2 to select-Title,
      op select3 to select-Year,op select4 to select-Genre,
      op select5 to select-Type))[STRING,STRING,INT,STRING,
      (DISJOINTUNION2 *(sort Disjointunion to Type,op apply1
      to Type-CD,op is1 to is-CD,op get1 to get-CD,op apply2
      to Type-DVD,op is2 to is-DVD,op get2 to get-DVD))
      [(REGISTER2 *(sort Register to CD,op reg to reg-CD,
      op select1 to select-Band,op select2 to select-Tracks))
      [STRING,(LIST *(sort List to Tracks))[STRING]],
      (LIST *(sort List to DVD))[STRING]]] .
 op include-track : Media String -> Media .
 op include-actor : Media String -> Media .
 op get-mediatitle : Media -> String .
 op get-mediacode : Media -> String .
 var !c : String .
 var !ti : String .
 var !y : Int .
 var !g : String .
 var !b : String .
 var !tr : Tracks .
 var !tr-name : String .
 var !ac : DVD .
 var !ac-name : String .
 var !Title : String .
 var !Year : Int .
 var !Genre : String .
 var !Type : Type .
 var !MediaCode : String .
 eq include-track(reg-Media(!c,!ti,!y,!g,Type-CD(reg-CD(!b,
!tr))),!tr-name) = reg-Media(!c,!ti,!y,!g,Type-CD(reg-CD(!b,
cons(!tr-name,!tr)))) .
 eq include-actor(reg-Media(!c,!ti,!y,!g,Type-DVD(!ac)),
!ac-name) = reg-Media(!c,!ti,!y,!g,Type-DVD(cons(!ac-name,!ac))) .
 eq get-mediacode(reg-Media(!c,!Title,!Year,!Genre,!Type)) = !c .
 eq get-mediatitle(reg-Media(!MediaCode,!ti,!Year,!Genre,!Type))
```

```
= !ti .
endo
```

8.1.9 Exemplo de Reduções de Termos

O exemplo de termo a ser reduzido consiste de uma instância da especificação da locadora. O objetivo é mostrar que as operações que computam os empréstimos apresentam a funcionalidade desejada. Primeiramente, é preciso criar um termo que represente um estado válido da classe VCDRShop. Portanto, o termo:

```
reg-VCDRShop(add(reg-Client("xy01","John","Avenue 15th n34",
"5551234"),emptyset),modify("d08",reg-Disc("m02",
date(04,07,2002),4),emptymap),emptymap,add(reg-Media("m02",
"Signs",2002,"Fiction",Type-DVD(add("actor1",add("actor2",
emptyset)))),emptyset))
```

representa uma locadora que tem os seguintes dados cadastrados:

- Cliente chamado “John”;
- DVD cujo título é “Signs”;
- Uma cópia (disco) do DVD Signs.

Considere `t` o termo mostrado acima. Abaixo é mostrado o empréstimo do DVD “Signs” ao cliente “John”.

```
lend-disc(t,"d08","xy01",date(10,10,2002))
=
reg-VCDRShop(add(reg-Client("xy01","John","Avenue 15th n34",
"5551234"),emptyset),modify("d08",reg-Disc("m02",date(04,07,
2002),4),emptymap),modify("d08",reg-RentalInfo("xy01",date(
10,10,2002)),emptymap),add(reg-Media("m02","Signs",2002,
"Fiction",Type-DVD(add("actor1",add("actor2",emptyset)))),
emptyset))
```

Considere como `t'` o termo da locadora que contém as informações do empréstimo para simular a devolução deste disco à locadora. O primeiro passo para efetuar a devolução de um disco é calcular o pagamento das diárias do mesmo para então computar a baixa deste empréstimo. Ambas operações são mostradas a seguir.

```
calculate-bill(t',"d08",date(12,10,2002)) = 8

give-disc-back(t',"d08")
=
reg-VCDRShop(add(reg-Client("xy01","John","Avenue 15th n34",
"5551234"),emptyset),modify("d08",reg-Disc("m02",
date(04,07,2002),4),emptymap),emptymap,add(reg-Media("m02",
"Signs",2002,"Fiction",Type-DVD(add("actor1",add("actor2",
emptyset)))),emptyset))
```

Como pode ser observado acima, o valor a ser pago por John é oito reais e, após a baixa do disco, a locadora retorna ao estado inicial antes do empréstimo.

8.2 Estudo de Caso 2: Especificações do ProTool

Nesta seção são apresentadas as traduções para OBJ dos ATOs que envolvem o ProTool, especificados no capítulo 6. Todos os ATOs foram testados no ambiente OBJ3 para validar tanto as operações de edição de ATO algébricos quanto as operações de tradução.

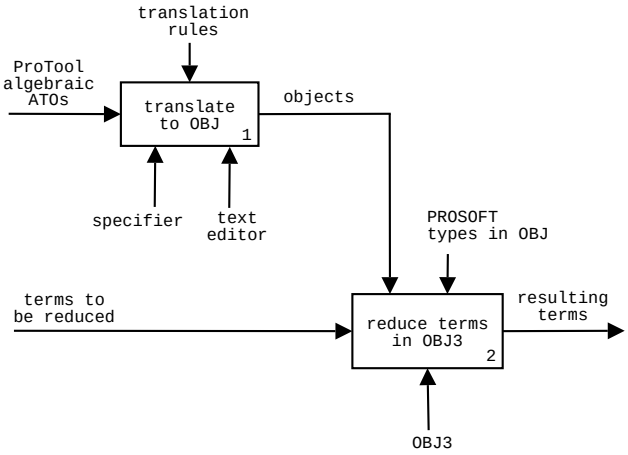


Figura 8.8: Validação das especificações do ProTool utilizando somente as regras de tradução

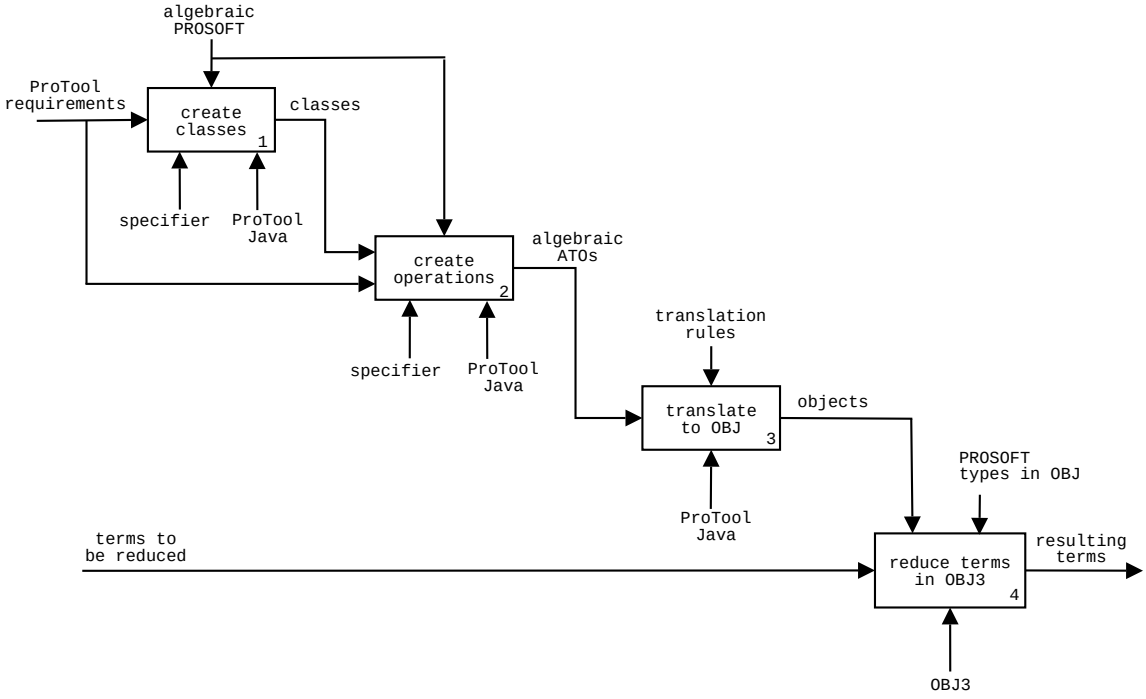


Figura 8.9: Validação das especificações do ProTool utilizando o ProTool-Java

O diagrama SADT da figura 8.8 ilustra o processo de validação dos ATOs algébricos do ProTool em OBJ. Este tipo de validação dos ATOs algébricos implica na validação do mapeamento proposto antes da implementação do ProTool no ambiente PROSOFT. A atividade 1 do diagrama mostra que os ATOs algébricos foram traduzidos manualmente para OBJ seguindo as regras de tradução estabelecidas neste trabalho. A atividade 2 ilustra as reduções de termos que foram realizadas para validar os ATOs dentro do ambiente

OBJ3. Nesta atividade quem executa a redução de termos é o ambiente OBJ3 de acordo com os objetos (ATOs traduzidos para OBJ) e os tipos de dados do PROSOFT em OBJ. Este processo de validação foi muito importante porque propiciou o autor identificar algumas falhas na especificação assim como simplificar algumas regras de tradução.

O diagrama SADT da figura 8.9 ilustra o processo de validação das especificações do ProTool utilizando a ferramenta ProTool-Java. As duas primeiras atividades mostram a edição dos ATOs algébricos do ProTool utilizando a ferramenta ProTool-Java. A saída da segunda atividade é composta dos ATOs algébricos que compõem a especificação do ProTool. Na sequência, a ferramenta ProTool-Java traduz automaticamente os ATOs algébricos para objetos em OBJ. A utilização destes objetos resultantes da tradução junto com os termos a serem reduzidos, permitem que sejam computadas as reduções, viabilizando a validação por execução das especificações. Os termos a serem reduzidos na atividade 4 foram os mesmo utilizados na atividade 2 da figura 8.8. As reduções resultaram nas mesmas formas normais obtidas anteriormente, o que significa que a implementação satisfez a especificação.

Cabe ressaltar que as reduções sobre as especificações do ProTool não são mostradas aqui, pois ocuparia muito espaço de texto para as detalhar.

A seguir são mostradas as traduções dos ATOs algébricos do ProTool para OBJ.

8.2.1 ATOMetaATO

O resultado da tradução deste ATO para OBJ é:

```
obj ATOMetaATO is
 pr (REGISTER6 *(sort Register to MetaATO,op reg to reg-MetaATO,
     op select1 to select-Name,op select2 to select-Class,
     op select3 to select-Extref,op select4 to select-Interfaces,
     op select5 to select-Variables, op select6 to select-Axioms))
     [STRING,ATOClass,(ATOOtherATOs *(sort OtherATOs to Extref)),
     ATOInterfaces,ATOVariables,ATOAxioms] .
 op createMetaATO : String -> MetaATO .
 op addprimitive : String String MetaATO -> MetaATO .
 op addlist : String MetaATO -> MetaATO .
 op addset : String MetaATO -> MetaATO .
 op addmap : String MetaATO -> MetaATO .
 op addregister : String Integer MetaATO -> MetaATO .
 op adddu : String Integer MetaATO -> MetaATO .
 op ins-extref : String MetaATO -> MetaATO .
 op getextref : MetaATO -> String .
 op delextref : String MetaATO -> MetaATO .
 op isemptyextref : MetaATO -> Bool .
 op addinterface : Interface MetaATO -> MetaATO .
 op createopint : String String -> Interface .
 op includeoparity : String Interface -> Interface .
 op opname : Interface -> String .
 op isemptyarity : Interface -> Bool .
 op valuesort : Interface -> String .
 op sortarity : Interface -> String .
 op delarity : String Interface -> Interface .
```

```
op isemptyints : Interfaces -> Bool .
op getlint : Interfaces -> Interface .
op delint : Interface Interfaces -> Interfaces .
op addvariable : String String MetaATO -> MetaATO .
op getvname : MetaATO -> String .
op getvsort : String MetaATO -> String .
op delvar : String MetaATO -> MetaATO .
op isemptyvars : MetaATO -> Bool .
op addaxiom : Term Term MetaATO -> MetaATO .
op delaxiom : Term MetaATO -> MetaATO .
op get-t-lhs : MetaATO -> Term .
op get-t-rhs : Term MetaATO -> Term .
op isemptyaxs : MetaATO -> Bool .
op addop-in-term : String -> Term .
op addterm-in-term : Term Term -> Term .
op getopname : Term -> String .
op getsubterms : Term -> Terms .
op hassubterms : Term -> Bool .
op termstail : Terms -> Terms .
op termshead : Terms -> Term .
op makeobj : String -> OBJ .
op impmod : String Module OBJ -> OBJ .
op ins-opdecl : Opdecl OBJ -> Signature .
op makemod : String -> Module .
op opdecl : String String -> Opdecl .
op ins-arity : String Opdecl -> Opdecl .
op ins-var : String String OBJ -> OBJ .
op ins-equation : Term Term OBJ -> OBJ .
op mk-op-in-term : String -> Term .
op ins-term-in-term : Term Term -> Term .
op ins-terms-in-term : String Terms -> Term .
op t-to-terms : Term -> Terms .
op link-terms : Terms Terms -> Terms .
op isrecclass : Class String -> Bool .
op get-classname : Class -> String .
op phi-class : Class -> Module .
op phi-sorts : Class OBJ String -> OBJ .
op phi-rclass : Class Class OBJ -> OBJ .
op phi : MetaATO -> OBJ .
op phi-extref : MetaATO MetaATO OBJ -> OBJ .
op phi-ints : MetaATO Interfaces OBJ -> OBJ .
op fillarity : Interface Opdecl -> Opdecl .
op phi-vars : MetaATO MetaATO OBJ -> OBJ .
op phi-axioms : MetaATO MetaATO OBJ -> OBJ .
op phi-term : Term -> Term .
op phi-subterms : Terms -> Terms .
op extract-text : OBJ -> Text .
var name : String .
```

```
 var prim : String .
 var sort : String .
 var n : Int .
 var extref : Extref .
 var ints    : Interfaces .
 var int : Interface .
 var axioms  : Axioms .
 var t1 : Term .
 var t2 : Term .
 var obj : OBJ .
 var op : Opdecl .
 var opdecl : Opdecl .
 var oldsort : String .
 var newsort : String .
 var oldop newop : String .
 var vars : Variables .
 var class : Class .
 var ma : MetaATO .
 var terms : Terms .
 var mname : String .
 eq createMetaATO(name) = reg-MetaATO(name,createclass("un"),
emptyset,emptylist,emptymap,emptymap) .
 eq addprimitive(name,prim,reg-MetaATO(mname,class,extref,
ints,vars,axioms)) = reg-MetaATO(mname,addPrimitive(name,prim,
class),extref,ints,vars,axioms) .
 eq addlist(name,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = reg-MetaATO(mname,addList(name,class),extref,ints,
vars,axioms) .
 eq addset(name,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = reg-MetaATO(mname,addSet(name,class),extref,ints,
vars,axioms) .
 eq addmap(name,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = reg-MetaATO(mname,addMap(name,class),extref,ints,
vars,axioms) .
 eq addregister(name,n,reg-MetaATO(mname,class,extref,ints,
vars,axioms)) = reg-MetaATO(mname,addRegister(name,n,class),
extref,ints,vars,axioms) .
 eq adddu(name,n,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = reg-MetaATO(mname,addDisjointUnion(name,n,class),
extref,ints,vars,axioms) .
 eq ins-extref(name,reg-MetaATO(mname,class,extref,ints,
vars,axioms)) = reg-MetaATO(mname,class,addato(name,extref),
ints,vars,axioms) .
 eq getextref(reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = getatoname(extref) .
 eq delextref(name,reg-MetaATO(mname,class,extref,ints,
vars,axioms)) = reg-MetaATO(mname,class,deleteato(name,
extref),ints,vars,axioms) .
```

```
 eq isemptyextref(reg-MetaATO(mname,class,extref,ints,
vars,axioms)) = isemptyotheratos(extref) .
 eq addinterface(int,reg-MetaATO(mname,class,extref,ints,
vars,axioms)) = reg-MetaATO(mname,class,extref,
addinterface(int,ints),vars,axioms) .
 eq createopint(name,sort) = createint(name,sort) .
 eq includeoparity(sort,int) = addarity(sort,int) .
 eq opname(int) = getop(int) .
 eq isemptyarity(int) = isemptyaritylist(int) .
 eq valuesort(int) = getvsort(int) .
 eq sortarity(int) = getsarity(int) .
 eq delarity(sort,int) = deletesarity(sort,int) .
 eq isemptyints(ints) = isemptyinterfaces(ints) .
 eq getint(ints) = getinterface(ints) .
 eq delint(int,ints) = deleteinterface(int,ints) .
 eq addvariable(name,sort,reg-MetaATO(mname,class,extref,ints,
vars,axioms)) = reg-MetaATO(mname,class,extref,ints,addvar(
name,sort,vars),axioms) .
 eq getvname(reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = getvarname(vars) .
 eq getvsort(name,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = getvarsort(name,vars) .
 eq delvar(name,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = reg-MetaATO(mname,class,extref,ints,deletevar(
name,vars),axioms) .
 eq isemptyvars(reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = isemptyvariables(vars) .
 eq addaxiom(reg-MetaATO(mname,class,extref,ints,vars,
axioms),t1,t2) = reg-MetaATO(mname,class,extref,ints,vars,
addaxiom(t1,t2,axioms)) .
 eq delaxiom(t,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = reg-MetaATO(mname,class,extref,ints,vars,
deleteaxiom(t,axioms)) .
 eq get-t-lhs(reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = get-term-lhs(axioms) .
 eq get-t-rhs(t,reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = get-term-rhs(t,axioms) .
 eq isemptyaxs(reg-MetaATO(mname,class,extref,ints,vars,
axioms)) = isemptyaxioms(axioms) .
 eq addop-in-term(name) = addop(name) .
 eq addterm-in-term(t1,t2) = addterm(t1,t2) .
 eq getopname(t) = getopname(t) .
 eq getsubterms(t) = getterms(t) .
 eq hassubterms(t) = hasparameters(t) .
 eq termstail(terms) = parameterstail(terms) .
 eq termshead(terms) = parametershead(terms) .
 eq makeobj(name) = createOBJ(name) .
 eq impmod(mode,mod,obj) = addimp(mode,mod,obj) .
```

```
 eq ins-opdecl(op,obj) = insertop(op,obj) .
 eq makemod(name) = createmodule(name) .
 eq opdecl(name,sort) = declareop(name,sort) .
 eq ins-arity(sort,opdecl) = addarity(sort,opdecl) .
 eq ins-var(name,sort,obj) = addvar(name,sort,obj) .
 eq ins-equation(t1,t2,obj) = addeq(t1,t2,obj) .
 eq mk-op-in-term(name) = add-op-in-term(name) .
 eq ins-term-in-term(t1,t2) = addterm-in-term(t1,t2) .
 eq ins-terms-in-term(name,terms) = add-terms(name,terms) .
 eq t-to-terms(t) = termtoterms(t) .
 eq link-terms(t1,t2) = concatterms(t,tt) .
 eq isrecclass(class,name) = isrec(class,name) .
 eq get-classname(class) = getname(class) .
 eq phi-class(class) = phic(class) .
 eq phi-sorts(class,obj,name) = sorts-cr(class,obj,name) .
 eq phi-rclass(class,class,obj) = phicr(class,class,obj) .

 eq phi(ma) = if isrecclass(select-Class(ma),get-classname(
select-Class(ma))) then phi-extref(ma,ma,phi-rclass(
select-Class(ma),select-Class(ma),phi-sorts(select-Class(ma),
makeobj("ATO"++select-Name(ma)),get-classname(
select-Class(ma))))) else phi-extref(ma,ma,impmod("pr",
phi-class(select-Class(ma)),makeobj("ATO"++select-Name(ma))))
fi .

 eq phi-extref(ma,reg-MetaATO(mname,class,extref,ints,vars,
axioms),obj) = if isemptyextref(extref) then phi-ints(ma,
select-Interfaces(ma),obj) else if getextref(extref) ==
"Boolean" then phi-extref(ma,reg-MetaATO(mname,class,
delextref("Boolean",extref),ints,vars,axioms),impmod("pr",
makemod("BOOL"),obj)) else if getextref(extref) == "Integer"
then phi-extref(ma,reg-MetaATO(mname,class,delextref("Integer",
extref),ints,vars,axioms),impmod("pr",makemod("INT"),obj)) else
phi-extref(ma,reg-MetaATO(mname,class,delextref(getextref(extref),
extref),ints,vars,axioms),impmod("pr",makemod(ucase(getextref(
extref))),obj)) fi fi fi .


eq phi-ints(ma,ints,obj) = if isemptyints(ints) then phi-vars(ma,
ma,obj) else if valuesort(getint(ints)) == "Boolean" then
phi-ints(ma,delint(getint(ints),ints),ins-opdecl(fillarity(
getint(ints),opdecl(opname(getint(ints)),"Bool"),obj) else
if valuesort(getint(ints)) == "Integer" then phi-ints(ma,
delint(getint(ints),ints),ins-opdecl(fillarity(getint(ints),
opdecl(opname(getint(ints)),"Int"),obj) else phi-ints(ma,
delint(getint(ints),ints),ins-opdecl(fillarity(getint(ints),
opdecl(opname(getint(ints)),valuesort(getint(ints))),obj)
fi fi fi .
```

```
eq fillarity(int,opdecl) = if isemptyarity(int) then opdecl
else if sortarity(int) == "Boolean" then fillarity(delarity(
"Boolean",int),ins-arity("Bool",opdecl)) else if sortarity(
int) == "Integer" then fillarity(delarity("Integer",int),
ins-arity("Int",opdecl)) else fillarity(delarity(sortarity(
int),int),ins-arity(sortarity(int),opdecl)) fi fi fi .

eq phi-vars(ma,reg-MetaATO(mname,class,extref,ints,vars,
axioms),obj) = if isemptyvars(vars) then phi-axioms(ma,ma,obj)
else phi-vars(ma,reg-MetaATO(mname,class,extref,ints,delvar(
getvname(reg-MetaATO(mname,class,extref,ints,vars,axioms)),
reg-MetaATO(mname,class,extref,ints,vars,axioms)),axioms),
ins-var(getvname(reg-MetaATO(mname,class,extref,ints,vars,
axioms)),getvsort(getvname(reg-MetaATO(mname,class,extref,
ints,vars,axioms)),reg-MetaATO(mname,class,extref,ints,vars,
axioms)),obj)) fi .

eq phi-axioms(ma,reg-MetaATO(mname,class,extref,ints,vars,
axioms),obj) = if isemptyaxs(axioms) then obj else phi-axioms(ma,
delaxiom(t,reg-MetaATO(mname,class,extref,ints,vars,axioms)),
ins-equation(phi-term(get-t-lhs(reg-MetaATO(mname,class,extref,
ints,vars,axioms)),phi-term(get-t-rhs(get-t-lhs(reg-MetaATO(
mname,class,extref,ints,vars,axioms)),reg-MetaATO(mname,class,
extref,ints,vars,axioms))),obj)) fi .

eq phi-term(t) = if isin(ucase(getopname(t)),"ICS") then
ins-terms-in-term(termshead(getsubterms(t)),phi-subterms(
termstail(getsubterms(t)))) else if hassubterms(t) then
ins-terms-in-term(getopname(t),phi-subterms(getsubterms(t)))
else mk-op-in-term(getopname(t)) fi fi .

 eq phi-subterms(terms) = if terms == emptylist then terms
else link-terms(t-to-terms(phi-term(termshead(terms))),
phi-subterms(termstail(terms))) fi .

 eq extract-text(obj) = phi-txt(obj) .
endo
```

8.2.2 ATOClass

O resultado da tradução deste ATO para OBJ é:

```
obj ATOClass is
 sort Class .
 sort Type .
 sort Map .
 sort Register .
 sort DisjointUnion .
```

```
op reg-Class : String Type -> Class .
op select-Name : Class -> String .
op select-Type : Class -> Type .
var !1r2 : String .
var !2r2 : Type .
eq select-Name(reg-Class(!1r2,!2r2)) = !1r2 .
eq select-Type(reg-Class(!1r2,!2r2)) = !2r2 .

op Type-Primitive : String -> Type .
op Type-List : Class -> Type .
op Type-Set : Class -> Type .
op Type-Map : Map -> Type .
op Type-Register : Register -> Type .
op Type-DisjointUnion : DisjointUnion -> Type .
op is-Primitive : Type -> Bool .
op is-List : Type -> Bool .
op is-Set : Type -> Bool .
op is-Map : Type -> Bool .
op is-Register : Type -> Bool .
op is-DisjointUnion : Type -> Bool .
op get-Primitive : Type -> String .
op get-List : Type -> Class .
op get-Set : Type -> Class .
op get-Map : Type -> Map .
op get-Register : Type -> Register .
op get-DisjointUnion : Type -> DisjointUnion .
var !1d6 : String .
var !2d6 : Class .
var !3d6 : Class .
var !4d6 : Map .
var !5d6 : Register .
var !6d6 : DisjointUnion .
eq is-Primitive(Type-Primitive(!1d6)) = true .
eq is-Primitive(Type-List(!2d6)) = false .
eq is-Primitive(Type-Set(!3d6)) = false .
eq is-Primitive(Type-Map(!4d6)) = false .
eq is-Primitive(Type-Register(!5d6)) = false .
eq is-Primitive(Type-DisjointUnion(!6d6)) = false .
eq is-List(Type-Primitive(!1d6)) = false .
eq is-List(Type-List(!2d6)) = true .
eq is-List(Type-Set(!3d6)) = false .
eq is-List(Type-Map(!4d6)) = false .
eq is-List(Type-Register(!5d6)) = false .
eq is-List(Type-DisjointUnion(!6d6)) = false .
eq is-Set(Type-Primitive(!1d6)) = false .
eq is-Set(Type-List(!2d6)) = false .
eq is-Set(Type-Set(!3d6)) = true .
```

```
eq is-Set(Type-Map(!4d6)) = false .
eq is-Set(Type-Register(!5d6)) = false .
eq is-Set(Type-DisjointUnion(!6d6)) = false .
eq is-Map(Type-Primitive(!1d6)) = false .
eq is-Map(Type-List(!2d6)) = false .
eq is-Map(Type-Set(!3d6)) = false .
eq is-Map(Type-Map(!4d6)) = true .
eq is-Map(Type-Register(!5d6)) = false .
eq is-Map(Type-DisjointUnion(!6d6)) = false .
eq is-Register(Type-Primitive(!1d6)) = false .
eq is-Register(Type-List(!2d6)) = false .
eq is-Register(Type-Set(!3d6)) = false .
eq is-Register(Type-Map(!4d6)) = false .
eq is-Register(Type-Register(!5d6)) = true .
eq is-Register(Type-DisjointUnion(!6d6)) = false .
eq is-DisjointUnion(Type-Primitive(!1d6)) = false .
eq is-DisjointUnion(Type-List(!2d6)) = false .
eq is-DisjointUnion(Type-Set(!3d6)) = false .
eq is-DisjointUnion(Type-Map(!4d6)) = false .
eq is-DisjointUnion(Type-Register(!5d6)) = false .
eq is-DisjointUnion(Type-DisjointUnion(!6d6)) = true .
eq get-Primitive(Type-Primitive(!1d6)) = !1d6 .
eq get-List(Type-List(!2d6)) = !2d6 .
eq get-Set(Type-Set(!3d6)) = !3d6 .
eq get-Map(Type-Map(!4d6)) = !4d6 .
eq get-Register(Type-Register(!5d6)) = !5d6 .
eq get-DisjointUnion(Type-DisjointUnion(!6d6)) = !6d6 .

op reg-Map : Class Class -> Map .
op select-Domain : Map -> Class .
op select-Range : Map -> Class .
var !1r22 : Class .
var !2r22 : Class .
eq select-Domain(reg-Map(!1r22,!2r22)) = !1r22 .
eq select-Range(reg-Map(!1r22,!2r22)) = !2r22 .

pr INT .
pr SET[Class] .
op emptylist : -> Register .
op cons : Class Register -> Register .
op elements : Register -> Set .
op head : Register -> Class .
op last : Register -> Class .
op length : Register -> Int .
op projection : Register Int -> Class .
op replace : Register Int Class -> Register .
op tail : Register -> Register .
op concat : Register Register -> Register .
```

```
 op isin : Class Register -> Bool .
 op delete : Class Register -> Register .
 op invert : Register -> Register .
 var c : Class .
 var c1 : Class .
 var l : Register .
 var l1 : Register .
 var n : Int .
 eq elements(emptylist) = emptyset .
 eq elements(cons(c,l)) = add(c,elements(l)) .
 eq last(cons(c,l)) = if l == emptylist then c else last(l) fi .
 eq head(cons(c,l)) = c .
 eq length(emptylist) = 0 .
 eq length(cons(c,l)) = length(l) + 1 .
 eq tail(cons(c,l)) = l .
 eq projection(cons(c,l),n) = if n == 1 then c else projection(
tail(cons(c,l)),n - 1) fi .
 eq replace(emptylist,n,c1) = emptylist .
 eq replace(cons(c,l),n,c1) = if n == 1 then cons(c1,l) else
cons(c,replace(l,n - 1,c1)) fi .
 eq concat(l,emptylist) = l .
 eq concat(emptylist,l) = l .
 eq concat(cons(c,l),l1) = cons(c,concat(l,l1)) .
 eq isin(c,emptylist) = false .
 eq isin(c1,cons(c,l)) = if (c == c1) then true else isin(c1,l) fi .
 eq delete(c,emptylist) = emptylist .
 eq delete(c1,cons(c,l)) = if (c == c1) then l else cons(c,delete(c1,
l)) fi .
 eq invert(emptylist) = emptylist .
 eq invert(cons(c,l)) = concat(invert(l),cons(c,emptylist)) .

 pr INT .
 pr SET[Class] .
 op emptylist : -> DisjointUnion .
 op cons : Class DisjointUnion -> DisjointUnion .
 op elements : DisjointUnion -> Set .
 op head : DisjointUnion -> Class .
 op last : DisjointUnion -> Class .
 op length : DisjointUnion -> Int .
 op projection : DisjointUnion Int -> Class .
 op replace : DisjointUnion Int Class -> DisjointUnion .
 op tail : DisjointUnion -> DisjointUnion .
 op concat : DisjointUnion DisjointUnion -> DisjointUnion .
 op isin : Class DisjointUnion -> Bool .
 op delete : Class DisjointUnion -> DisjointUnion .
 op invert : DisjointUnion -> DisjointUnion .
 var c2 : Class .
 var c12 : Class .
```

```
 var l2 : DisjointUnion .
 var l12 : DisjointUnion .
 var n : Int .
 eq elements(emptylist) = emptyset .
 eq elements(cons(c2,l2)) = add(c2,elements(l2)) .
 eq last(cons(c2,l2)) = if l2 == emptylist then c2 else
last(l2) fi .
 eq head(cons(c2,l2)) = c2 .
 eq length(emptylist) = 0 .
 eq length(cons(c2,l2)) = length(l2) + 1 .
 eq tail(cons(c2,l2)) = l2 .
 eq projection(cons(c2,l2),n) = if n == 1 then c2 else
projection(tail(cons(c2,l2)),n - 1) fi .
 eq replace(emptylist,n,c12) = emptylist .
 eq replace(cons(c2,l2),n,c12) = if n == 1 then cons(c12,l2)
else cons(c2,replace(l2,n - 1,c12)) fi .
 eq concat(l2,emptylist) = l2 .
 eq concat(emptylist,l2) = l2 .
 eq concat(cons(c2,l2),l12) = cons(c2,concat(l2,l12)) .
 eq isin(c2,emptylist) = false .
 eq isin(c12,cons(c2,l2)) = if (c2 == c12) then true else
isin(c12,l2) fi .
 eq delete(c2,emptylist) = emptylist .
 eq delete(c12,cons(c2,l2)) = if (c2 == c12) then l2 else
cons(c2,delete(c12,l2)) fi .
 eq invert(emptylist) = emptylist .
 eq invert(cons(c2,l2)) = concat(invert(l2),cons(c2,emptylist)) .

 pr BOOL .
 pr INT .
 pr ATOOBJ .
 op createclass : String -> Class .
 op addPrimitive : String String Class -> Class .
 op addList : String Class -> Class .
 op addSet : String Class -> Class .
 op addMap : String Class -> Class .
 op addRegister : String Int Class -> Class .
 op addDisjointUnion : String Int Class -> Class .
 op createMembers : Int -> Register .
 op createAlternatives : Int -> DisjointUnion .
 op path : Class Class -> Class .
 op teta : Type Class -> Type .
 op lambda : DisjointUnion Class -> DisjointUnion .
 op lambda : Register Class -> Register .
 op is-closed : Class -> Bool .
 op isrec : Class String -> Bool .
 op isrec : Register String -> Bool .
 op isrec : DisjointUnion String -> Bool .
```

```
op phic : Class -> Module .
op count : Register -> Int .
op count : DisjointUnion -> Int .
op inttostr : Int -> String .
op ren-reg-ops : Module Register Int -> Module .
op ren-du-ops : Module DisjointUnion Int String -> Module .
op mk-mod : String -> Module .
op rnm-op : String String Module -> Module .
op rnmsort : String String Module -> Module .
op inst-mod : Module Module -> Module .
op decl-sort : String OBJ -> OBJ .
op imp-mod : String Module OBJ -> OBJ .
op rnm-du-ops : String String String Module -> Module .
op sorts-cr : Class OBJ String -> OBJ .
op sorts-cr : Register OBJ String -> OBJ .
op sorts-cr : DisjointUnion OBJ String -> OBJ .
op isPrimitive : Class -> Bool .
op get-ref : Class -> String .
op phicr : Class Class OBJ -> OBJ .
op getdomain : Class -> String .
op getrange : Class -> String .
op isList : Class -> Bool .
op isSet : Class -> Bool .
op isMap : Class -> Bool .
op isRegister : Class -> Bool .
op isDisjointUnion : Class -> Bool .
op getname : Class -> String .
op getsort : Class -> String .
op reg-list : Register Module -> Module .
op du-list : DisjointUnion Module -> Module .
op aux-reg : Register Class OBJ -> OBJ .
op aux-du : DisjointUnion Class OBJ -> OBJ .
var newclass : Class .
var class : Class .
var m : Class .
var a : Class .
var range : Class .
var domain : Class .
var list : Class .
var set : Class .
var oldclass : Class .
var type : Type .
var type2 : Type .
var type3 : Type .
var name : String .
var name2 : String .
var name3 : String .
var cname : String .
```

```
 var prim : String .
 var mode : String .
 var namelist : String .
 var n : Int .
 var mlist : Register .
 var alist : DisjointUnion .
 var mod : Module .
 var mod1 : Module .
 var obj : OBJ .


 eq createclass(name) = reg-Class(name,Type-Primitive("ut") .
 eq addPrimitive(name,prim,oldclass) = path(oldclass,
reg-Class(name,Type-Primitive(prim))) .
 eq addList(name,oldclass) = path(oldclass,reg-Class(name,
Type-List(reg-Class("un",Type-Primitive("ut"))))) .
 eq addSet(name,oldclass) = path(oldclass,reg-Class(name,
Type-Set(reg-Class("un",Type-Primitive("ut"))))) .
 eq addMap(name,oldclass) = path(oldclass,reg-Class(name,
Type-Map(reg-Map(reg-Class("un",Type-Primitive("ut")),
reg-Class("un",Type-Primitive("ut")))))) .
 eq addRegister(name,n,oldclass) = path(oldclass,
reg-Class(name,Type-Register(createMembers(n)))) .
 eq addDisjointUnion(name,n,oldclass) = path(oldclass,
reg-Class(name,Type-DisjointUnion(createAlternatives(n)))) .
 eq createMembers(n) = if n > 0 then concat(cons(
reg-Class("un",Type-Primitive("ut")),emptylist),
createMembers(n-1)) else emptylist fi .
 eq createAlternatives(n) = if n > 0 then concat(cons(
reg-Class("un",Type-Primitive("ut")),emptylist),
createAlternatives(n-1)) else emptylist fi .
 eq path(reg-Class(name,type),newclass) = if (name == "un")
then newclass else reg-Class(name,teta(type,newclass)) fi .
 eq teta(Type-List(reg-Class(name,type)),reg-Class(
name2,type2)) = if (name == "un") then Type-List(reg-Class(
name2,type2)) else Type-List(reg-Class(name,teta(type,
reg-Class(name2,type2)))) fi .
 eq teta(Type-Set(reg-Class(name,type)),reg-Class(name2,
type2)) = if (name == "un") then Type-Set(reg-Class(name2,
type2)) else Type-Set(reg-Class(name,teta(type,reg-Class(
name2,type2)))) fi .
 eq teta(Type-DisjointUnion(reg-Class(cons(reg-Class(name,
type),alist))),reg-Class(name2,type2)) = if name == "un"
then Type-DisjointUnion(reg-Class(cons(reg-Class(name2,
type2),alist))) else Type-DisjointUnion (reg-Class(lambda
(cons(reg-Class(name,type),alist),reg-Class(name2,
type2)))) fi .
 eq lamda(cons(reg-Class(name,type),alist),newclass) =
```

```
if is-closed(reg-Class(name,type)) then concat(cons(
reg-Class(name,type),emptylist),lambda(alist,newclass))
else cons(reg-Class(name,teta(type,newclass)),alist) fi .
 eq teta(Type-Register(reg-Class(cons(reg-Class(
name,type),mlist))),reg-Class(name2,type2)) = if name ==
 "un" then Type-Register(reg-Class(cons(reg-Class(name2,
type2),mlist))) else Type-Register(reg-Class(lambda(cons(
reg-Class(name,type),mlist),reg-Class(name2,type2)))) fi .
 eq lamda(cons(reg-Class(name,type),mlist),newclass) =
if is-closed(reg-Class(name,type)) then concat(cons(
reg-Class(name,type),emptylist),lambda(mlist,newclass))
else cons(reg-Class(name,teta(type,newclass)),mlist) fi .

eq teta(Type-Map(reg-Map(reg-Class(name,type),
reg-Class(name2,type2))),reg-Class(name3,type3)) =
if name == "un" then Type-Map(reg-Map(reg-Class(
name3,type3),reg-Class(name2,type2))) else if not(
is-closed(reg-Class(name,type))) then Type-Map(reg-Map(
reg-Class(name,teta(type,reg-Class(name3,type3))),
reg-Class(name2,type2))) else if (name2 == "un")
then Type-Map(reg-Map(reg-Class(name,type),
reg-Class(name3,type3))) else Type-Map(reg-Map(
reg-Class(name,type),reg-Class(name2,teta(type2,
reg-Class(name3,type3))))) fi fi fi .


 eq is-closed(reg-Class(name,Type-Primitive(prim))) =
if prim == "ut" then false else true fi .
 eq is-closed(reg-Class(name,Type-List(list))) =
is-closed(list) .
 eq is-closed(reg-Class(name,Type-Set(set))) =
is-closed(set) .
 eq is-closed(reg-Class(name,Type-Map(reg-Map(domain,
range)))) = is-closed(domain) and is-closed(range) .
 eq is-closed(reg-Class(name,Type-Register(emptylist)))
= false .
 eq is-closed(reg-Class(name,Type-Register(cons(m,
mlist)))) = is-closed(m) and is-closed(reg-Class(name,
Type-Register(mlist))) .
 eq is-closed(reg-Class(name,Type-DisjointUnion(
emptylist))) = false .
 eq is-closed(reg-Class(name,Type-DisjointUnion(cons(a,
alist)))) = is-closed(a) and is-closed(reg-Class(name,
Type-DisjointUnion(alist))) .
 eq isrec(reg-Class(name,Type-List(list)),name)
= isrec(list,name) .
 eq isrec(reg-Class(name,Type-Set(set)),name)
= isrec(set,name) .
```

```
 eq isrec(reg-Class(name,Type-Map(reg-Map(domain,
range))),name) = isrec(domain,name) or isrec(range,name) .
 eq isrec(reg-Class(name,Type-Primitive(prim)),name)
= if prim == name then true else false fi .
 eq isrec(cons(m,mlist),name) = if mlist == emptylist then
isrec(m,name) else isrec(m,name) or isrec(mlist,name) fi .
 eq isrec(cons(a,alist),name) = if alist == emptylist then
isrec(a,name) else isrec(a,name) or isrec(alist,name) fi .
 eq isrec(reg-Class(name,Type-Register(mlist)),name) =
isrec(mlist,name) .
 eq isrec(reg-Class(name,Type-DisjointUnion(alist)),name)
= isrec(alist,name) .
 eq mk-mod(name) = createmodule(name) .
 eq rnm-op(oldop,newop,mod) = renameop(oldop,newop,mod) .
 eq rnmsort(oldsort,newsort,mod) = renamesort(oldsort,
newsort,mod) .
 eq inst-mod(mod,mod1) = instmodule(mod,mod1) .
 eq decl-sort(name,obj) = declaresort(name,obj) .
 eq imp-mod(mode,mod,obj) = addimp(mode,mod,obj) .
 eq count(emptylist) = 0 count(cons(m,members)) = 1 +
count(members) .
 eq count(cons(a,alternatives)) = 1 + count(alternatives) .
 eq inttostr((1).Integer) = (1).String .
 eq inttostr((2).Integer) = (2).String .
 eq inttostr((3).Integer) = (3).String .
 eq inttostr((4).Integer) = (4).String .
 eq inttostr((5).Integer) = (5).String .
 eq inttostr((6).Integer) = (6).String .
 eq inttostr((7).Integer) = (7).String .
 eq inttostr((8).Integer) = (8).String .
 eq inttostr((9).Integer) = (9).String .
 eq inttostr((10).Integer) = (10).String .
 eq ren-reg-ops(mod,cons(m,mlist),n) = if mlist =/= emptylist
then ren-reg-ops(rnm-op("select"++inttostr(n),"select-"++
getname(m),mod),mlist,n - 1) else rnm-op("select"++inttostr
(n),"select-"++getname(m),mod) fi .
 eq ren-du-ops(mod,cons(a,alist),n,name) = if alist =/=
emptylist then ren-du-ops(rnm-op("get"++ ++inttostr(n),
"get\-"++getname(a),rnm-op("is"++inttostr(n),"is\-"++
getname(a),rnm-op("apply"++inttostr(n),name++getname(a),
mod))),alist,n - 1,name) else rnm-op("get"++ ++inttostr(n),
"get-"++getname(a),rnm-op("is"++inttostr(n),"is\-"++getname(a),
rnm-op("apply"++inttostr(n),name++getname(a),mod))) fi .
 eq rnm-du-ops(name,prim,namelist,mod) = if not(isin("\#",
namelist)) and length(namelist) > 1 then mod else rnm-du-ops(
name,prim,extract(locate("\#",namelist) + 1,length(namelist),
namelist),rnm-op(prim++"\-"++extract(1,locate("\#",namelist)
- 1,namelist),name++"-++extract(1,locate("\#",namelist) - 1,
```

```
namelist),mod)) fi .
 eq phic(reg-Class(name,Type-List(list))) =
inst-mod(phic(list),rnmsort("List",name,mk-mod("LIST"))) .
 eq phic(reg-Class(name,Type-Set(set)))
= inst-mod(phic(set),rnmsort("Set",name,mk-mod("SET"))) .
 eq phic(reg-Class(name,Type-Map(reg-Map(domain,range))))
= inst-mod(phic(range),inst-mod(phic(domain),sortrnm("Map",
name,mk-mod("MAP")))) .
 eq phic(reg-Class(name,Type-Register(cons(m,mlist)))) =
inst-mod(phic(reg-Class(name,Type-Register(mlist))),inst-mod(
phic(m),ren-reg-ops(rnm-op("reg","reg\-"++name,rnmsort("Register"
++inttostr(count(cons(m,mlist))),name,mk-mod("REGISTER"++
inttostr(count(cons(m,mlist)))))),cons(m,mlist),
count(cons(m,mlist))))) .
 eq phic(reg-Class(name,Type-DisjointUnion(cons(a,alist)))) =
inst-mod(phic(reg-Class(name,Type-DisjointUnion(alist))),
inst-mod(phic(a),ren-du-ops(rnmsort("DisjointUnion"++
inttostr(count(cons(a,alist))),name,mk-mod("DISJOINTUNION"++
inttostr(count(cons(a,alist)))),cons(a,alist),count(cons(a,
alist)),name))) .
 eq phic(reg-Class(name,Type-Primitive(prim))) = if prim ==
"Boolean" then mk-mod("BOOL") else if prim == "Integer" then
mk-mod("INT") else if prim == "String" or prim == "Date" or
prim == "Time" then mk-mod(ucase(prim)) else rnmsort(prim,
name,mk-mod("ATO"++prim)) fi fi fi .
 eq sorts-cr(reg-Class(name,Type-List(list)),obj,cname) =
sorts-cr(list,decl-sort(name,obj),cname) .
 eq sorts-cr(reg-Class(name,Type-Set(set)),obj,cname) =
sorts-cr(set,decl-sort(name,obj),cname) .
 eq sorts-cr(reg-Class(name,Type-Map(reg-Map(domain,range)),
obj,cname) = sorts-cr(range,sorts-cr(domain,decl-sort(name,
obj),cname),cname) .
 eq sorts-cr(reg-Class(name,Type-Register(mlist)),obj,cname)
= sorts-cr(mlist,decl-sort(name,obj),cname) .
 eq sorts-cr(cons(m,mlist),obj,cname) = if mlist == emptylist
then sorts-cr(m,obj,cname) else sorts-cr(m,sorts-cr(mlist,obj,
cname),cname) fi .
 eq sorts-cr(reg-Class(name,Type-DisjointUnion(alist)),obj,
 cname) = sorts-cr(alist,decl-sort(name,obj),cname) .
 eq sorts-cr(cons(a,alist),obj,cname) = if alist == emptylist
then sorts-cr(a,obj,cname) else sorts-cr(a,sorts-cr(alist,obj,
cname),cname) fi .
 eq sorts-cr(reg-Class(name,Type-Primitive(prim)),obj,cname) =
if prim == "Boolean" then imp-mod("pr",mk-mod("BOOL"),obj)
else if prim == "Integer" then imp-mod("pr",mk-mod("INT"),obj)
else if prim == "String" or prim == "Date" or prim == "Time"
then imp-mod("pr",mk-mod(ucase(prim)),obj) else if prim == cname
then obj else imp-mod("pr",rnmsort(prim,name,mk-mod("ATO"++prim)),
```

```
obj) fi fi fi fi .
 eq isPrimitive(reg-Class(name,type)) = is-Primitive(type) .
 eq get-ref(reg-Class(name,Type-Primitive(prim))) = prim .
 eq getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
Type-Primitive(prim)),range)))) = if prim == "Boolean" then "Bool"
else if prim == "Integer" then "Int" else if prim == "String" or
prim == "Date" or prim == "Time" then prim else name2 fi fi fi .
 eq getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
Type-List(list)),range)))) = name2 .
 eq getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
Type-Set(set)),range)))) = name2 .
 eq getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
Type-Map(map)),range)))) = name2 .
 eq getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
Type-Register(mlist)),range)))) = name2 .
 eq getdomain(reg-Class(name,Type-Map(reg-Map(reg-Class(name2,
Type-DisjointUnion(alist)),range)))) = name2 .
 eq getrange(reg-Class(name,Type-Map(reg-Map(name3,reg-Class(
name2,Type-Primitive(prim)))))) = if prim == "Boolean" then
"Bool" else if prim == "Integer" then "Int" else if prim ==
"String" or prim == "Date" or prim == "Time" then prim else
name2 fi fi fi .
 eq getrange(reg-Class(name,Type-Map(reg-Map(domain,reg-Class(
name2,Type-List(list)))))) = name2 .
 eq getrange(reg-Class(name,Type-Map(reg-Map(domain,reg-Class(
name2,Type-Set(set)))))) = name2 .
 eq getrange(reg-Class(name,Type-Map(reg-Map(domain,reg-Class(
name2,Type-Map(map)))))) = name2 .
 eq getrange(reg-Class(name,Type-Map(reg-Map(domain,reg-Class(
name2,Type-Register(mlist)))))) = name2 .
 eq getrange(reg-Class(name,Type-Map(reg-Map(domain,reg-Class(
name2,Type-DisjointUnion(alist)))))) = name2 .
 eq isList(reg-Class(name,type)) = if is-List(type) == true
then true else false fi .
 eq isSet(reg-Class(name,type)) = if is-Set(type) == true
then true else false fi .
 eq isMap(reg-Class(name,type)) = if is-Map(type) == true
then true else false fi .
 eq isRegister(reg-Class(name,type)) = if is-Register(type)
== true then true else false fi .
 eq isDisjointUnion(reg-Class(name,type))
= if is-DisjointUnion(type) == true then true else false fi .
 eq getname(class) = select-Name(class) .
 eq getsort(reg-Class(name,Type-Primitive(prim)))
= if prim == "Boolean" then "Bool" else if prim == "Integer"
then "Int" else if prim == "String" or prim == "Date" or
prim == "Time" then prim else name fi fi fi .
 eq getsort(reg-Class(name,Type-List(list))) = name .
```

```
 eq getsort(reg-Class(name,Type-Set(set))) = name .
 eq getsort(reg-Class(name,Type-Map(map))) = name .
 eq getsort(reg-Class(name,Type-Register(mlist))) = name .
 eq getsort(reg-Class(name,Type-DisjointUnion(alist))) = name .
 eq reg-list(cons(m,mlist),mod) = if mlist == emptylist
then mod else reg-list(mlist,rnm-op(getname(m),getsort(m),
mod)) fi .
 eq du-list(cons(a,alist),mod) = if alist == emptylist
then mod else du-list(alist,rnm-op(getname(a),getsort(a),
mod)) fi .


eq phicr(reg-Class(name,Type-List(list)),class,obj)
= if not(isPrimitive(list)) then phicr(list,class,
imp-mod("create\-spec",rnmsort(name,getname(list),
mk-mod("LIST")),obj)) else if get-ref(list) ==
getname(class) then imp-mod("create\-spec",
rnmsort(name,get-ref(list),mk-mod("LIST")),obj)
else if get-ref(list) == "Boolean" then
imp-mod("create\-spec",rnmsort(name,"Bool",
mk-mod("LIST")),obj) else if get-ref(list) ==
"Integer" then imp-mod("create\-spec",rnmsort(name,
"Int",mk-mod("LIST")),obj) else if get-ref(list) ==
"String" or get-ref(list) == "Date" or get-ref(list)
== "Time" then imp-mod("create\-spec",rnmsort(name,
get-ref(list),mk-mod("LIST")),obj) else imp-mod(
"create\-spec",rnmsort(name,getname(list),mk-mod(
"LIST")),obj) fi fi fi fi fi .



eq phicr(reg-Class(name,Type-Set(set)),class,obj) =
if not(isPrimitive(set)) then phicr(set,class,imp-mod(
"create\-spec",rnmsort(name,getname(set),mk-mod("SET")),
obj)) else if get-ref(set) == getname(class) then
imp-mod("create\-spec",rnmsort(name,get-ref(set),mk-mod(
"SET")),obj) else if get-ref(set) == "Boolean" then
imp-mod("create\-spec",rnmsort(name,"Bool",mk-mod("SET"))
,obj) else if get-ref(set) == "Integer" then imp-mod(
"create\-spec",rnmsort(name,"Int",mk-mod("SET")),obj)
else if get-ref(set) == "String" or get-ref(set) ==
"Date" or get-ref(set) == "Time" then imp-mod(
"create\-spec",rnmsort(name,get-ref(set),mk-mod("SET"
)),obj) else imp-mod("create\-spec",rnmsort(name,
getname(set),mk-mod("SET")),obj) fi fi fi fi fi .


eq phicr(reg-Class(name,Type-Map(reg-Map(domain,range))),
```

```
class,obj) = if not(isPrimitive(domain)) and not(
isPrimitive(range)) then phicr(domain,class,phicr(range,
class,imp-mod("create\-spec",rnm-op(getname(range),
"\empty",rnmsort(name,getname(domain),mk-mod("MAP"))),
obj))) else if not(isPrimitive(domain)) and isPrimitive(
range) then if get-ref(range) == getname(class) then
phicr(domain,class,imp-mod("create\-spec",rnm-op(get-ref(
range),"\empty",rnmsort(name,getname(domain),mk-mod("MAP"
))),obj)) else if get-ref(range) == "Boolean" then phicr(
domain,class,imp-mod("create\-spec",rnm-op("Bool","\empty",
rnmsort(name,getname(domain),mk-mod("MAP"))),obj)) else
if get-ref(range) == "Integer" then phicr(domain,class,
imp-mod("create\-spec",rnm-op("Int","\empty",rnmsort(name,
getname(domain),mk-mod("MAP"))),obj)) else if get-ref(range)
== "String" or get-ref(range) == "Date" or get-ref(range)
== "Time" then phicr(domain,class,imp-mod("create\-spec",
rnm-op(get-ref(range),"\empty",rnmsort(name,getname(domain),
mk-mod("MAP"))),obj)) else phicr(domain,class,imp-mod(
"create\-spec",rnm-op(getname(range),"\empty",rnmsort(name,
getname(domain),mk-mod("MAP"))),obj)) fi fi fi fi else
if isPrimitive(domain) and not(isPrimitive(range))
then if get-ref(domain) == getname(class) then phicr(range,
class,imp-mod("create\-spec",rnm-op(getname(range),"\empty"
,rnmsort(name,get-ref(domain),mk-mod("MAP"))),obj)) else
if get-ref(domain) == "Boolean" then phicr(range,class,
imp-mod("create\-spec",rnm-op(getname(range),"\empty",
rnmsort(name,"Bool",mk-mod("MAP"))),obj)) else if
get-ref(domain) == "Integer" then phicr(range,class,
imp-mod("create\-spec",rnm-op(getname(range),"\empty",
rnmsort(name,"Int",mk-mod("MAP"))),obj)) else if
get-ref(domain) == "String" or get-ref(domain) ==
"Date" or get-ref(domain) == "Time" then phicr(range,
class,imp-mod("create\-spec",rnm-op(getname(range),
"\empty",rnmsort(name,get-ref(domain),mk-mod("MAP"))),obj))
else phicr(range,class,imp-mod("create\-spec",rnm-op(getname
(range),"\empty",rnmsort(name,getname(domain),mk-mod("MAP"))),
obj)) fi fi fi fi else imp-mod("create\-spec",rnm-op(get-ref(
range),"\empty",rnmsort(name,get-ref(domain),mk-mod("MAP"))),
obj)) fi fi fi .

 eq phicr(reg-Class(name,Type-Register(mlist)),class,obj) =
aux-reg(mlist,class,imp-mod("create\-spec",reg-list(mlist,
rnmsort(name,"\empty",mk-mod("REGISTER"))),obj)) .
 eq aux-reg(cons(m,mlist),class,obj) = if mlist == emptylist
then if isPrimitive(m) then obj else phicr(m,class,obj) fi
else if isPrimitive(m) then obj else phicr(m,class,aux-reg(
mlist,class,obj)) fi fi .
 eq phicr(reg-Class(name,Type-DisjointUnion(alist)),class,
```

```
obj) = aux-du(alist,class,imp-mod("create\-spec",du-list(
alist,rnmsort(name,"\empty",mk-mod("DISJOINTUNION"))),obj)) .
 eq aux-du(cons(a,alist),class,obj) = if alist == emptylist
then if isPrimitive(a) then obj else phicr(a,class,obj) fi
else if isPrimitive(a) then obj else phicr(a,class,aux-du(
alist,class,obj)) fi fi .
endo
```

8.2.3 ATOOtherATOs

O resultado da tradução deste ATO para OBJ é:

```
obj ATOOtherATOs is
 pr (SET *(sort Set to OtherATOs))[STRING] .
 pr BOOL .
 op addato : String OtherATOs -> OtherATOs .
 op getatoname : OtherATOs -> String .
 op deletaato : String OtherATOs -> OtherATOs .
 op isemptyotheratos : OtherATOs -> Bool .
 var name : String .
 var name2 : String .
 var oatos : OtherATOs .
 eq addato(name,otaos) = cons(name,oatos) .
 eq getatoname(add(name,oatos)) = name .
 eq deleteato(name,oatos) = delete(name,oatos) .
 eq isemptyotheratos(oatos) = isempty(oatos) .
endo
```

8.2.4 ATOVariables

O resultado da tradução deste ATO para OBJ é:

```
obj ATOVariables is
 pr (MAP *(sort Map to Variables))[STRING,STRING] .
 pr BOOL .
 op addvar : String String Variables -> Variables .
 op getvarname : Variables -> String .
 op getvarsort : String Variables -> String .
 op deletevar : String Variables -> Variables .
 op isemptyvariables : Variables -> Bool .
 var vname   : String .
 var vsort   : String .
 var vars    : Variables .
 eq addvar(vname,vsort,vars) = modify(vname,vsort,vars) .
 eq getvarname(modify(vname,vsort,vars)) = vname .
 eq getvarsort(vname,vars) = imageof(vname,vars) .
 eq deletevar(vname,vars)
= restrictwith(vars,add(vname,emptyset)) .
 eq isemptyvariables(vars) = isempty(vars) .
endo
```

8.2.5 ATOInterfaces

O resultado da tradução deste ATO para OBJ é:

```
obj ATOInterfaces is
 pr (LIST *(sort List to Interfaces))[(ATOOpdecl *
(sort Opdecl to Interface))] .
 op addinterface : Interface Interfaces -> Interfaces .
 op createint : String String -> Interface .
 op addarity : String Interface -> Interface .
 op getop : Interface -> String .
 op isemptyaritylist : Interface -> Bool .
 op getvsort : Interface -> String .
 op getsarity : Interface -> String .
 op deletesarity : String Interface -> Opdecl .
 op isemptyinterfaces : Interfaces -> Bool .
 op getinterface : Interfaces -> Interface .
 op deleteinterface : Interface Interfaces -> Interfaces .
 var ints : Interfaces .
 var int : Interface .
 var opname : String .
 var sort : String .
 eq addinterface(int,ints) = cons(int,ints) .
 eq createint(opname,sort) = createopdecl(opname,sort) .
 eq addarity(sort,int) = addsortarity(sort,int) .
 eq getop(int) = getopname(int) .
 eq isemptyaritylist(int) = isemptyarity(int) .
 eq getvsort(int) = getvaluesort(int) .
 eq getsarity(int) = getsortarity(int) .
 eq deletesarity(sort,int) = deletesortarity(sort,int) .
 eq isemptyinterfaces(ints) = isempty(ints) .
 eq getinterface(ints) = head(int) .
 eq deleteinterface(int,ints) = delete(int,ints) .
endo
```

8.2.6 ATOOpdecl

O resultado da tradução deste ATO para OBJ é:

```
obj ATOOpdecl is
 pr (REGISTER3 *(sort Register to Opdecl,op reg to reg-Opdecl,
op select1 to select-Opname,op select2 to select-Arity,
op select3 to select-Valuesort))[STRING,(LIST *(sort List to
Arity))[STRING],STRING] .
 pr BOOL .
 op createopdecl : String String -> Opdecl .
 op addsortarity : String Opdecl -> Opdecl .
 op getopname : Opdecl -> String .
 op isemptyarity : Opdecl -> Bool .
 op getvaluesort : Opdecl -> String .
```

```
 op getsortarity : Opdecl -> String .
 op deletesortarity : String Opdecl -> Opdecl .
 var aritylist   : Arity .
 var sort : String .
 var opname : String .
 var opdecl  : Opdecl .
 var vsort : String .
 eq createopdecl(opname,sort) = reg-Opdecl(opname,emptylist,
sort) .
 eq addsortarity(sort,reg-Opdecl(_,aritylist,_)) = reg-Opdecl
(_,cons(sort,aritylist),_) .
 eq getopname(opdecl) = select-Opname(opdecl) .
 eq isemptyarity(reg-Opdecl(_,aritylist,_)) = isempty(
aritylist) .
 eq getvaluesort(opdecl) = select-Valuesort(opdecl) .
 eq getsortarity(opdecl) = head(select-Arity(opdecl)) .
 eq deletesortarity(sort,reg-Opdecl(opname,aritylist,vsort)) =
reg-Opdecl(opname,delete(sort,aritylist),vsort) .
endo
```

8.2.7 ATOAxioms

O resultado da tradução deste ATO para OBJ é:

```
obj ATOAxioms is
 pr (MAP *(sort Map to Axioms))[ATOTerm,ATOTerm] .
 op addaxiom : Term Term Axioms -> Axioms .
 op deleteaxiom : Term Axioms -> Axioms .
 op get-term-lhs : Axioms -> Term .
 op get-term-rhs : Term Axioms -> Term .
 op isemptyaxioms : Axioms -> Bool .
 var axioms  : Axioms .
 var t : Term .
 var tt : Term .
 var terms : Terms .
 eq addaxiom(t,tt,axioms) = modify(t,tt,axioms) .
 eq deleteaxiom(t,axioms) = restrictwith(axioms,
add(t,emptyset)) .
 eq get-term-lhs(modify(t,tt,axioms)) = t .
 eq get-term-rhs(t,axioms) = imageof(t,axioms) .
 eq isemptyaxioms(axioms) = isempty(axioms) .
endo
```

8.2.8 ATOTerm

O resultado da tradução deste ATO para OBJ é:

```
obj ATOTerm is
 sort Term .
 sort Terms .
```

```
 op reg-Term : String Terms -> Term .
 op select-Opname : Term -> String .
 op select-Terms : Term -> Terms .
 var !1r2 : String .
 var !2r2 : Terms .
 eq select-Opname(reg-Term(!1r2,!2r2)) = !1r2 .
 eq select-Terms(reg-Term(!1r2,!2r2)) = !2r2 .
 pr INT .
 pr SET[Term] .
 op emptylist : -> Terms .
 op cons : Term Terms -> Terms .
 op elements : Terms -> Set .
 op head : Terms -> Term .
 op last : Terms -> Term .
 op length : Terms -> Int .
 op projection : Terms Int -> Term .
 op replace : Terms Int Term -> Terms .
 op tail : Terms -> Terms .
 op concat : Terms Terms -> Terms .
 op isin : Term Terms -> Bool .
 op delete : Term Terms -> Terms .
 op invert : Terms -> Terms .
 var c : Term .
 var c1 : Term .
 var l : Terms .
 var l1 : Terms .
 var n : Int .
 eq elements(emptylist) = emptyset .
 eq elements(cons(c,l)) = add(c,elements(l)) .
 eq last(cons(c,l)) = if l == emptylist then c else
last(l) fi .
 eq head(cons(c,l)) = c .
 eq length(emptylist) = 0 .
 eq length(cons(c,l)) = length(l) + 1 .
 eq tail(cons(c,l)) = l .
 eq projection(cons(c,l),n) = if n == 1 then c else
projection(tail(cons(c,l)),n - 1) fi .
 eq replace(emptylist,n,c1) = emptylist .
 eq replace(cons(c,l),n,c1) = if n == 1 then cons(c1,l)
else  cons(c,replace(l,n - 1,c1)) fi .
 eq concat(l,emptylist) = l .
 eq concat(emptylist,l) = l .
 eq concat(cons(c,l),l1) = cons(c,concat(l,l1)) .
 eq isin(c,emptylist) = false .
 eq isin(c1,cons(c,l)) = if (c == c1) then true else
isin(c1,l) fi .
 eq delete(c,emptylist) = emptylist .
 eq delete(c1,cons(c,l)) = if (c == c1) then l else
```

```
cons(c,delete(c1,l)) fi .
 eq invert(emptylist) = emptylist .
 eq invert(cons(c,l)) = concat(invert(l),cons(c,
emptylist)) .
 pr BOOL .
 pr INT .
 op addop : String -> Term .
 op addterm : Term Term -> Term .
 op getopname: Term -> String .
 op addterms : String Terms -> Term .
 op getterms : Term -> Terms .
 op hasparameters : Term -> Bool .
 op parameterstail : Terms -> Terms .
 op parametershead : Terms -> Term .
 op termslength : Terms -> Int .
 op termtoterms : Term -> Terms .
 op concatterms : Terms Terms -> Terms .
 var opname : String .
 var terms : Terms .
 var newterm : Term .
 var t : Term .
 var tt : Term .
 eq addop(opname) = reg-Term(opname,emptylist) .
 eq addterm(newterm,reg-Term(opname,terms)) = reg-Term(opname,
cons(newterm,terms)) .
 eq getopname(t) = select-Opname(t) .
 eq addterms(opname,terms) = reg-Term(opname,terms) .
 eq getterms(t) = select-Terms(t) .
 eq hasparameters(reg-Term(_,terms)) = isempty(terms) .
 eq parameterstail(terms) = tail(terms) .
 eq parametershead(terms) = head(terms) .
 eq termslength(terms) = length(terms) .
 eq termtoterms(t) = cons(t,emptylist) .
 eq concatterms(t,tt) = concat(t,tt) .
endo
```

8.2.9 ATOOBJ

O resultado da tradução deste ATO para OBJ é:

```
obj ATOOBJ is
 pr (REGISTER4 *(sort Register to OBJ,op reg to reg-OBJ,
op select1 to select-Name,op select2 to select-Signature,
op select3 to select-Variables,op select4 to
select-Equations))[STRING,ATOSignature,ATOVariables,
(ATOAxioms *(sort Axioms to Equations))] .
 pr ATOText .
 op createOBJ : String -> OBJ .
 op declaresort : String OBJ -> OBJ .
```

```
op delsortdecl : String OBJ -> OBJ .
op isemptysortdecl : OBJ -> Boolean .
op firstsortdecl : OBJ -> String .
op addimp : String Module OBJ -> OBJ .
op del-imp : String Module Importations -> Importations .
op putimps : Importations OBJ -> OBJ .
op getimps : OBJ -> Importations .
op isemptyimps : OBJ -> Bool .
op lastimpmode : OBJ -> String .
op lastimpmod : OBJ -> Module .
op insopdecl : Opdecl OBJ -> Signature .
op isemptyopdecl : OBJ -> Bool .
op firstopdecl : OBJ -> Opdecl .
op delopdecl : Opdecl OBJ -> Signature .
op mod-name : Module -> String .
op createmodule : String -> Module .
op renamesort : String String Module -> Module .
op renameop : String String Module -> Module .
op instmodule : Module Module -> Module .
op isempty-sortrenm : Module -> Bool .
op get-oldsort : Module -> String .
op get-newsort : String Module -> String .
op del-sortrenm : String Module -> Module .
op isempty-oprenm : Module -> Bool .
op get-oldop : Module -> String .
op get-newop : String Module -> String .
op del-oprenm : String Module -> Module .
op isempty-inst : Module -> Bool .
op last-module : Module -> Module .
op del-instmod : Module Module -> Module .
op declareop : String String -> Opdecl .
op addarity : String Opdecl -> Opdecl .
op get-opname : Opdecl -> String .
op isempty-arity : Opdecl -> Bool .
op get-valuesort : Opdecl -> String .
op get-sortarity : Opdecl -> String .
op del-sortarity : String Opdecl -> Opdecl .
op addvar : String String OBJ -> OBJ .
op getvarname : OBJ -> String .
op getvarsort : String OBJ -> String .
op delvar : String OBJ -> Variables .
op isempty-vars : OBJ -> Bool .
op addeq : Term Term OBJ -> OBJ .
op del-eq : Term OBJ -> OBJ .
op get-lhs : OBJ -> Term .
op get-rhs : Term OBJ -> Term .
op isempty-eqs : OBJ -> Bool .
op add-op-in-term : String -> Term .
```

```
op add-term-in-term : Term Term -> Term .
op add-terms : String Terms -> Term .
op get-opname : Term -> String .
op getparams : Term -> Terms .
op hasparams : Term -> Bool .
op paramstail : Terms -> Terms .
op paramshead : Terms -> Term .
op terms-length : Terms -> Int .
op phi-txt : OBJ -> Text .
op ins-line : String Text -> Text .
op ins-text : Text Text -> Text .
op invert-text : Text -> Text .
op phi-sortdecl : OBJ Text -> Text .
op phi-imp : OBJ Text -> Text .
op isprimtypes : String -> Bool .
op phi-mod : Module String -> String .
op rnmsort : Module -> String .
op rnmops : Module String -> String .
op aux-mod : Module String -> String .
op phi-spec : Module Text -> Text .
op create-spec-set : Module Text -> Text .
op create-spec-list : Module Text -> Text .
op create-spec-map : Module Text -> Text .
op create-spec-reg : Module Text -> Text .
op create-spec-du : Module Text -> Text .
op phi-opdecl : OBJ Text -> Text .
op str-opdecl : Opdecl -> String .
op str-arity : Opdecl String -> String .
op phi-vars : OBJ Text -> Text .
op phi-eqs : OBJ Text -> Text .
op phi-t : Term String -> String .
op sub-ts : Terms String -> String .
op mixfix : String Terms -> String .
var name : String .
var opname : String .
var sort : String .
var mode : String .
var str : String .
var sig : Signature .
var opdecl : Opdecl .
var op : Opdecl .
var vars : Variables .
var eqs : Equations .
var t : Term .
var t1 : Term .
var t2 : Term .
var terms : Terms .
var mod : Module .
```

```
 var mod1 : Module .
 var prmode : String .
 var oldsort : String .
 var newsort : String .
 var oldop : String .
 var newop : String .
 var imps : Importations .
 var obj : OBJ .
 var txt : Text .
 var txt1 : Text .
 var line : String .
 eq createOBJ(name) = reg-OBJ(name,reg-Signature(emptylist,
emptylist,emptylist),emptymap,emptymap) .
 eq declaresort(sort,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(
name,addsortdecl(sort,sig),vars,eqs) .
 eq delsortdecl(sort,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(
name,delsortdecl(sort,sig),vars,eqs) .
 eq isemptysortdecl(reg-OBJ(name,sig,vars,eqs))
= isemptysortdecl(sig) .
 eq firstsortdecl(reg-OBJ(name,sig,vars,eqs)) =
firstsortdecl(sig).
 eq addimp(mode,mod,reg-OBJ(name,sig,vars,eqs)) =
reg-OBJ(name,addimp(mode,mod,sig),vars,eqs) .
 eq del-imp(mode,mod,imps) = delimp(mode,mod,imps) .
 eq putimps(imps,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(name,
putimps(imps,sig),vars,eqs) .
 eq getimps(reg-OBJ(name,sig,vars,eqs)) = getimps(sig) .
 eq isemptyimps(reg-OBJ(name,sig,vars,eqs)) = isemptyimp(sig) .
 eq lastimpmode(reg-OBJ(name,sig,vars,eqs)) = lastimpmode(sig) .
 eq lastimpmod(reg-OBJ(name,sig,vars,eqs)) = lastimpmod(sig) .
 eq insopdecl(opdecl,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(name,
addopdecl(opdecl,sig),vars,eqs) .
 eq isemptyopdecl(reg-OBJ(name,sig,vars,eqs)) =
isemptyopdecl(sig) .
 eq firstopdecl(reg-OBJ(name,sig,vars,eqs)) = firstopdecl(sig) .
 eq delopdecl(opdecl,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(name,
delopdecl(opdecl,sig),vars,eqs) .
 eq mod-name(mod) = getmodname(mod) .
 eq createmodule(name) = createmod(name) .
 eq renamesort(oldsort,newsort,mod) = rnmsort(oldsort,newsort,
mod) .
 eq renameop(oldop,newop,mod) = rnmop(oldop,newop,mod) .
 eq instmodule(mod,mod1) = instmod(mod,mod1) .
 eq isempty-sortrenm(mod) = isemptysortrenm(mod) .
 eq get-oldsort(mod) = getoldsort(mod) .
 eq get-newsort(sort,mod) = getnewsort(sort,mod) .
 eq del-sortrenm(sort,mod) = delsortrenm(sort,mod) .
 eq isempty-oprenm(mod) = isemptyoprenm(mod) .
```

```
 eq get-oldop(mod) = getoldop(mod) .
 eq get-newop(opname,mod) = getnewop(opname,mod) .
 eq del-oprenm(opname,mod) = deloprenm(opname,mod) .
 eq isempty-inst(mod) = isemptyinstantiate(mod) .
 eq last-module(mod) = lastmodule(mod) .
 eq del-instmod(mod,mod1) = delinstmod(mod,mod1) .
 eq declareop(opname,sort) = createopdecl(opname,sort) .
 eq addarity(sort,opdecl) = addsortarity(sort,opdecl) .
 eq get-opname(opdecl) = getopname(opdecl) .
 eq isempty-arity(opdecl) = isemptyarity(opdecl) .
 eq get-valuesort(opdecl) = getvaluesort(opdecl) .
 eq get-sortarity(opdecl) = getsortarity(opdecl) .
 eq del-sortarity(sort,opdecl) = deletesortarity(opdecl,sort) .
 eq addvar(name,sort,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(name,
sig,addvar(name,sort,vars),eqs) .
 eq getvarname(reg-OBJ(name,sig,vars,eqs)) = getvarname(vars) .
 eq getvarsort(name,reg-OBJ(name,sig,vars,eqs))
= getvarsort(name,vars) .
 eq delvar(name,reg-OBJ(name,sig,vars,eqs))
= deletevar(name,vars) .
 eq isempty-vars(reg-OBJ(name,sig,vars,eqs))
= isemptyvariables(vars) .
 eq addeq(t,t1,reg-OBJ(name,sig,vars,eqs)) = reg-OBJ(name,
sig,vars,addaxiom(t,t1,eqs)) .
 eq del-eq(reg-OBJ(name,sig,vars,eqs),t)
= deleteaxiom(eqs,t) .
 eq get-lhs(reg-OBJ(name,sig,vars,eqs))
= get-term-lhs(eqs) .
 eq get-rhs(t,reg-OBJ(name,sig,vars,eqs))
= get-term-rhs(t,eqs) .
 eq isempty-eqs(reg-OBJ(name,sig,vars,eqs))
= isemptyaxioms(eqs) .
 eq add-op-in-term(name) = addop(name) .
 eq add-term-in-term(t1,t2) = addterm(t1,t2) .
 eq add-terms(name,terms) = addterms(name,terms) .
 eq get-opname(t) = getopname(t) .
 eq getparams(t) = getterms(t) .
 eq hasparams(t) = hasparameters(t) .
 eq paramstail(terms) = parameterstail(terms) .
 eq paramshead(terms) = parametershead(terms) .
 eq terms-length(terms) = termslength(terms) .
 eq ins-line(line,txt) = addline(line,txt) .
 eq ins-text(txt,txt1) = addtext(txt,txt1) .
 eq invert-text(txt) = inverttext(txt) .
 eq phi-txt(obj,txt) =  phi-sortdecl(obj,ins-line("obj
\space"++select-Name(obj)++"\space is",ICS(ATOText,
emptylist))) .
 eq phi-sortdecl(obj,txt) = if isemptysortdecl(obj)
```

```
then phi-imp(obj,txt) else phi-sortdecl(delsortdecl(
obj,lastsortdecl(obj)),ins-line("sort \space "++
lastsortdecl(obj)++"\space .",txt)) fi .
 eq phi-imp(obj,txt) = if isemptyimps(obj) then
phi-opdecl(obj,txt) else if lastimpmode(obj) ==
"create\-spec" then phi-imp(putimps(del-imp(
"create\-spec",lastimpmod(obj),getimps(obj)),
obj),add-text(txt,phi-spec(lastimpmod(obj),
emptylist))) else phi-imp(putimps(del-imp("pr",
lastimpmod(obj),getimps(obj)),obj),ins-line("\space
pr \space "++phi-mod(lastimpmod(obj),"\empty")++
"\space .",txt)) fi fi .
 eq isprimtypes(name) = if name == "BOOL" or name
== "INT" or name == "STRING" or name == "DATE" or
name == "TIME" then true else false fi .

eq phi-mod(mod,str) = if isprimtypes(mod-name(mod))
then str++mod-name(mod) else if isin("ATO",
mod-name(mod)) and isempty-sortrenm(mod) then str++
mod-name(mod) else if isin("ATO",mod-name(mod))
then str++"\paropen"++mod-name(mod)++"\space \*
\paropen sort \space"++get-oldsort(mod)++"\space
to \space"++get-newsort(get-oldsort(mod),mod)++
"\parclose\parclose" else if (mod-name(mod) ==
"LIST") or (mod-name(mod) == "SET") then str++
"\paropen"++mod-name(mod)++"\space \*\paropen
sort \space"++get-oldsort(mod)++"\space to \space"
++get-newsort(get-oldsort(mod),mod)++"\parclose
\parclose\["++phi-mod(last-module(mod),"\empty")
++"\]" else if mod-name(mod) == "MAP" then str++
"\paropen"++mod-name(mod)++"\space \*\paropen
sort \space"++get-oldsort(mod)++"\space to \space"
++get-newsort(get-oldsort(mod),mod)++"\parclose\
parclose\["++phi-mod(last-module(mod))++"\comma"
++phimod(last-module(del-instmod(last-module(mod)
,mod)),"\empty")++"\]" else if isin("REGISTER",
mod-name(mod)) or (isin("DISJOINTUNION",mod-name(
mod))) then str++"\paropen"++mod-name(mod)++
"\space \*\paropen sort \space"++rnmsort(mod)
++rnmops(mod,"\empty")++"\parclose\parclose\["
++aux-mod(mod,"\empty")++"\]" else str fi fi
fi fi fi fi .

 eq rnmsort(mod) = get-oldsort(mod)++"\space to
\space"++get-newsort(get-oldsort(mod),mod) .
 eq rnmops(mod,str) = if isempty-oprenm(mod)
then str else "\comma op \space "++get-oldop(mod)
++"\space to \space"++get-newop(get-oldop(mod),
```

```
mod)++rnmops(del-oprenm(get-oldop(mod),mod),
"\empty") fi .
 eq aux-mod(mod,str) = if isempty-inst(mod)
then str else phi-mod(last-module(mod),"\empty")
++aux-mod(del-instmod(last-module(mod),mod),
"\empty")) fi .

eq phi-spec(mod,txt) = if mod-name(mod) == "SET" then
create-spec-set(mod,txt) else if mod-name(mod)
== "LIST" then create-spec-list(mod,txt) else
if mod-name(mod) == "MAP" then create-spec-map(mod,txt)
else if mod-name(mod) == "REGISTER" then
create-spec-reg(mod,txt) else if mod-name(mod) ==
"DISJOINTUNION" then create-spec-du(mod,txt) else txt
fi fi fi fi fi .


 eq create-spec-list(mod,txt) = ...
 eq create-spec-set(mod,txt) = ...
 eq create-spec-map(mod,txt) = ...
 eq create-spec-reg(mod,txt) = ...
 eq create-spec-du(mod,txt) = ...
 eq phi-opdecl(obj,txt) = if isemptyopdecl(obj) then
phi-vars(obj,txt) else phi-opdecl(delopdecl(lastopdecl(
obj),obj),ins-line(str-opdecl(last-opdecl(obj)),txt)) fi .
 eq str-opdecl(op) = "\space op \space "++get-opname(op)
++"\space \: \space"++str-arity(op,"\empty")++"\space
\-\> \space"++get-valuesort(op)++"\space\." .
 eq str-arity(op,str) = if isempty-arity(op) then str
else str++"\space"++get-sortarity(op)++str-arity(
del-sortarity(get-sortarity(op),op),"\empty") fi .
 eq phi-vars(obj,txt) = if isempty-vars(obj) then phi-eqs(
obj,txt) else phi-vars(delvar(getvarname(obj),obj),ins-line(
"\space var \space"++getvarname(obj)++"\space \: \space "++
getvarsort(getvarname(obj),obj)++"\.",txt)) fi .
 eq phi-eqs(obj,txt) = if isempty-eqs(obj) then ins-line(
"endo",txt) else phi-eqs(del-eq(get-lhs(obj),obj),ins-line(
"\space eq \space "++phi-t(get-lhs(obj),"\empty")++"\space
\= \space"++phi-t(get-rhs(get-lhs(obj),obj),"\empty")++
"\space\.",txt)) fi .
 eq phi-t(t,str) = if not(hasparams(t)) then get-opname(t)
else if not(isin("\underline",get-opname(t))) then
get-opname(t)++"\paropen"++sub-ts(getparams(t),"\empty")
++"\parclose" else mixfix(get-opname(t),getparams(t)) fi fi .
 eq sub-ts(terms,str) = if terms == ICS(ATOTerm,emptylist)
then str else if terms-length(terms) == 1 then str++phi-t(
paramshead(terms),"\empty") else sub-ts(paramstail(terms),
str++phi-t(paramshead(terms),"\empty")++"\comma") fi fi .
```

```
 eq mixfix(opname,terms) = if terms == emptylist then opname
else mixfix(rep1("\underline","\space"++phi-t(paramshead(terms
)),"\empty")++"\space",opname),paramstail(terms)) fi .
endo
```

8.2.10 ATOSignature

O resultado da tradução deste ATO para OBJ é:

```
obj ATOSignature is
 pr (REGISTER3 *(sort Register to Signature,op reg to
reg-Signature,op select1 to select-Sortdecls,op select2
to select-Importations,op select3 to select-Opdecls))
[(LIST *(sort List to Sortdecls))[STRING],(LIST *(sort
List to Importations))[(REGISTER2 *(sort Register to
Description,op reg to reg-Description,op select1 to
select-Mode,op select2 to select-Module))[STRING,
ATOModule],(LIST *(sort List to Opdecls))[ATOOpdecl]] .
 op addsortdecl : String Signature -> Signature .
 op delsortdecl : String Signature -> Signature .
 op isemptysortdecl : Signature -> Bool .
 op firstsortdecl : Signature -> String .
 op addimp : String Module Signature -> Signature .
 op delimp : String Module Importations -> Importations .
 op putimps : Importations Signature -> Signature .
 op getimps : Signature -> Importations .
 op isemptyimp : Signature -> Bool .
 op lastimpmode : Signature -> String .
 op lastimpmod : Signature -> Module .
 op addopdecl : Opdecl Signature -> Signature .
 op isemptyopdecl : Signature -> Bool .
 op firstopdecl : Signature -> Opdecl .
 op delopdecl : Opdecl Signature -> Signature .
 var sig : Signature .
 var newsort : String .
 var sort : String .
 var s : Sortdecls .
 var ops : Opdecls .
 var newop : Opdecl .
 var op : Opdecl .
 var imps : Importations .
 var impmode : String .
 var mode : String .
 var mod : Module .
 var mod1 : Module .
 var imps2 : Importations .
 eq addsortdecl(newsort,reg-Signature(s,imps,ops))
= reg-Signature(cons(newsort,s),imps,ops) .
 eq delsortdecl(sort,reg-Signature(s,imps,ops))
```

```
= reg-Signature(delete(sort,s),imps,ops) .
 eq isemptysortdecl(reg-Signature(s,imps,ops)) = isempty(s) .
 eq firstsortdecl(reg-Signature(s,imps,ops)) = last(s) .
 eq addimp(impmode,mod,reg-Signature(s,imps,ops)) =
reg-Signature(s,cons(reg-Description(impmode,mod),imps),ops) .
 eq delimp(mode,mod1,cons(reg-Description(impmode,mod),imps))
= if (impmode == mode) and (mod == mod1) then imps else cons(
reg-Description(impmode,mod),delimp(imps,mode,mod1)) .
 eq putimps(imps,reg-Signature(s,imps2,ops))
= reg-Signature(s,imps,ops) .
 eq getimps(sig) = select-Importations(sig) .
 eq isemptyimp(reg-Signature(s,imps,ops)) = isempty(imps) .
 eq lastimpmode(reg-Signature(s,imps,ops)) = select-Mode(
last(imps)) .
 eq lastimpmod(reg-Signature(s,imps,ops)) = select-Module(
last(imps)) .
 eq addopdecl(newop,reg-Signature(s,imps,ops))
= reg-Signature(s,imps,cons(newop,ops)) .
 eq isemptyopdecl(reg-Signature(s,imps,ops)) = isempty(ops) .
 eq firstopdecl(reg-Signature(s,imps,ops)) = head(opdecl) .
 eq delopdecl(op,reg-Signature(s,imps,ops)) = delete(op,ops) .
endo
```

8.2.11 ATOModule

O resultado da tradução deste ATO para OBJ é:

```
obj ATOModule is
 sort Module .
 sort Sortrenm .
 sort Oprenm .
 sort Instantiate .

 op reg-Module : String Sortrenm Oprenm Instantiate -> Module .
 op select-Modname : Module -> String .
 op select-Sortrenm : Module -> Sortrenm .
 op select-Oprenm : Module -> Oprenm .
 op select-Instantiate : Module -> Instantiate .
 var !1r4 : String .
 var !2r4 : Sortrenm .
 var !3r4 : Oprenm .
 var !4r4 : Instantiate .
 eq select-Modname(reg-Module(!1r4,!2r4,!3r4,!4r4)) = !1r4 .
 eq select-Sortrenm(reg-Module(!1r4,!2r4,!3r4,!4r4)) = !2r4 .
 eq select-Oprenm(reg-Module(!1r4,!2r4,!3r4,!4r4)) = !3r4 .
 eq select-Instantiate(reg-Module(!1r4,!2r4,!3r4,!4r4)) = !4r4 .

 pr (SET *(sort Set to SetD))[STRING] .
 pr (SET *(sort Set to SetR))[STRING] .
```

```
 op emptymap : -> Sortrenm .
 op modify : String String Sortrenm -> Sortrenm .
 op imageof : String Sortrenm -> String .
 op domain : Sortrenm -> SetD .
 op range : Sortrenm -> SetR .
 op merge : Sortrenm Sortrenm -> Sortrenm [comm].
 op restrictto : Sortrenm SetD -> Sortrenm .
 op restrictwith : Sortrenm SetD -> Sortrenm .
 op override : Sortrenm Sortrenm -> Sortrenm .
 op composition : Sortrenm Sortrenm -> Sortrenm .
 var d d1 : String .
 var r r1 : String .
 var m m1 : Sortrenm .
 var sd : SetD .
 eq imageof(d,modify(d1,r,m)) = if d == d1 then r else imageof(
d,m) fi .
 eq domain(modify(d,r,m)) = add(d,emptyset) U domain(m) .
 eq domain(emptymap) = emptyset .
 eq range(emptymap) = emptyset .
 eq range(modify(d,r,m)) = add(r,emptyset) U range(m) .
 eq merge(m,emptymap) = m .
 eq merge(modify(d,r,m),m1) = modify(d,r,merge(m,m1)) .
 eq restrictwith(modify(d,r,m),sd) = if (d belongsto sd) then
restrictwith(m,delete(d,sd)) else modify(d,r,restrictwith(m,
sd)) fi .
 eq restrictwith(emptymap,sd) = emptymap .
 eq restrictwith(m,emptyset) = m .
 eq restrictto(emptymap,sd) = emptymap .
 eq restrictto(m,emptyset) = emptymap .
 eq restrictto(modify(d,r,m),add(d1,sd)) = if (d1 belongsto
domain(modify(d,r,m))) then modify(d1,imageof(d1,modify(d,r,
m)),restrictto(restrictwith(modify(d,r,m),add(d1,emptyset)),
sd)) else restrictto(modify(d,r,m),sd) fi .
 eq override(m,emptymap) = m .
 eq override(m,modify(d1,r1,m1)) = override(merge(
restrictwith(m,add(d1,emptyset)),modify(d1,r1,emptymap)),m1) .
 eq composition(modify(d,r,m),modify(d1,r1,m1)) = modify(d,
r1,composition(m,m1)) .
 eq composition(m,emptymap) = m .

 pr (SET *(sort Set to SetD))[STRING] .
 pr (SET *(sort Set to SetR))[STRING] .
 op emptymap : -> Oprenm .
 op modify : String String Oprenm -> Oprenm .
 op imageof : String Oprenm -> String .
 op domain : Oprenm -> SetD .
 op range : Oprenm -> SetR .
 op merge : Oprenm Oprenm -> Oprenm [comm].
```

```
 op restrictto : Oprenm SetD -> Oprenm .
 op restrictwith : Oprenm SetD -> Oprenm .
 op override : Oprenm Oprenm -> Oprenm .
 op composition : Oprenm Oprenm -> Oprenm .
 var d2 d12 : String .
 var r2 r12 : String .
 var m2 m12 : Oprenm .
 var sd2 : SetD .
 eq imageof(d2,modify(d12,r2,m2)) = if d2 == d12 then r2 else
imageof(d2,m2) fi .
 eq domain(modify(d2,r2,m2)) = add(d2,emptyset) U domain(m2) .
 eq domain(emptymap) = emptyset .
 eq range(emptymap) = emptyset .
 eq range(modify(d2,r2,m2)) = add(r2,emptyset) U range(m2) .
 eq merge(m2,emptymap) = m2 .
 eq merge(modify(d2,r2,m2),m12) = modify(d2,r2,merge(m2,m12)) .
 eq restrictwith(modify(d2,r2,m2),sd2) = if (d2 belongsto sd2)
then restrictwith(m2,delete(d2,sd2)) else modify(d2,r2,
restrictwith(m2,sd2)) fi .
 eq restrictwith(emptymap,sd2) = emptymap .
 eq restrictwith(m2,emptyset) = m2 .
 eq restrictto(emptymap,sd2) = emptymap .
 eq restrictto(m2,emptyset) = emptymap .
 eq restrictto(modify(d2,r2,m2),add(d12,sd2)) = if (d12
belongsto domain(modify(d2,r2,m2))) then modify(d12,imageof(d12,
modify(d2,r2,m2)),restrictto(restrictwith(modify(d2,r2,m2),
add(d12,emptyset)),sd2)) else restrictto(modify(d2,r2,m2),
sd2) fi .
 eq override(m2,emptymap) = m2 .
 eq override(m2,modify(d12,r12,m12)) = override(merge(
restrictwith(m2,add(d12,emptyset)),modify(d12,r12,emptymap)),
m12) .
 eq composition(modify(d2,r2,m2),modify(d12,r12,m12))
= modify(d2,r12,composition(m2,m12)) .
 eq composition(m2,emptymap) = m2 .

 pr INT .
 pr SET[Module] .
 op emptylist : -> Instantiate .
 op cons : Module Instantiate -> Instantiate .
 op elements : Instantiate -> Set .
 op head : Instantiate -> Module .
 op last : Instantiate -> Module .
 op length : Instantiate -> Int .
 op projection : Instantiate Int -> Module .
 op replace : Instantiate Int Module -> Instantiate .
 op tail : Instantiate -> Instantiate .
 op concat : Instantiate Instantiate -> Instantiate .
```

```
 op isin : Module Instantiate -> Bool .
 op delete : Module Instantiate -> Instantiate .
 op invert : Instantiate -> Instantiate .
 var c : Module .
 var c1 : Module .
 var l : Instantiate .
 var l1 : Instantiate .
 var n : Int .
 eq elements(emptylist) = emptyset .
 eq elements(cons(c,l)) = add(c,elements(l)) .
 eq last(cons(c,l)) = if l == emptylist then c else last(l) fi .
 eq head(cons(c,l)) = c .
 eq length(emptylist) = 0 .
 eq length(cons(c,l)) = length(l) + 1 .
 eq tail(cons(c,l)) = l .
 eq projection(cons(c,l),n) = if n == 1 then c else projection(
tail(cons(c,l)),n - 1) fi .
 eq replace(emptylist,n,c1) = emptylist .
 eq replace(cons(c,l),n,c1) = if n == 1 then cons(c1,l) else
cons(c,replace(l,n - 1,c1)) fi .
 eq concat(l,emptylist) = l .
 eq concat(emptylist,l) = l .
 eq concat(cons(c,l),l1) = cons(c,concat(l,l1)) .
 eq isin(c,emptylist) = false .
 eq isin(c1,cons(c,l)) = if (c == c1) then true else isin(c1,
l) fi .
 eq delete(c,emptylist) = emptylist .
 eq delete(c1,cons(c,l)) = if (c == c1) then l else cons(c,
delete(c1,l)) fi .
 eq invert(emptylist) = emptylist .
 eq invert(cons(c,l)) = concat(invert(l),cons(c,emptylist)) .

 pr BOOL .
 op createmod : String -> Module .
 op getmodname : Module -> String .
 op rnmsort : String String Module -> Module .
 op rnmop : String String Module -> Module .
 op instmod : Module Module -> Module .
 op isemptysortrenm : Module -> Bool .
 op getoldsort : Module -> String .
 op getnewsort : String Module -> String .
 op delsortrenm : String Module -> Module .
 op isemptyoprenm : Module -> Bool .
 op getoldop : Module -> String .
 op getnewop : String Module -> String .
 op deloprenm : String Module -> Module .
 op isemptyinstantiate : Module -> Bool .
 op lastmodule : Module -> Module .
```

```
 op delinstmod : Module Module -> Module .
 var name : String .
 var sort : String .
 var opname : String .
 var oldsort : String .
 var newsort : String .
 var sortmap : Sortrenm .
 var opmap : Oprenm .
 var oldop : String .
 var newop : String .
 var mod : Module .
 var modlist : Instantiate .
 eq createmod(name) = reg-Module(name,emptymap,emptymap,
emptylist) .
 eq getmodname(mod) = select-Modname(mod) .
 eq rnmsort(oldsort,newsort,reg-Module(name,sortmap,opmap,
modlist)) = reg-Module(name,modify(oldsort,newsort,sortmap)
,opmap,modlist) .
 eq rnmop(oldop,newop,reg-Module(name,sortmap,opmap,modlist
)) = reg-Module(name,sortmap,modify(oldop,newop,opmap),modlist) .
 eq instmod(mod,reg-Module(name,sortmap,opmap,modlist))
= reg-Module(name,sortmap,opmap,cons(mod,modlist)) .
 eq isemptysortrenm(reg-Module(name,sortmap,opmap,modlist))
= isempty(sortmap) .
 eq getoldsort(reg-Module(name,modify(oldsort,newsort,sortmap),
opmap,modlist) = oldsort .
 eq getnewsort(sort,reg-Module(name,sortmap,opmap,modlist)) =
imageof(sort,sortmap) .
 eq delsortrenm(sort,reg-Module(name,sortmap,opmap,modlist))
= reg-Module(name,restrictwith(sortmap,add(sort,emptyset)),
opmap,modlist) .
 eq isemptyoprenm(reg-Module(name,sortmap,opmap,modlist)) =
isempty(opmap) .
 eq getoldop(reg-Module(name,sortmap,modify(oldop,newop,
opmap),modlist) = oldop .
 eq getnewop(opname,reg-Module(name,sortmap,opmap,modlist))
= imageof(opname,opmap) .
 eq deloprenm(opname,reg-Module(name,sortmap,opmap,modlist))
= reg-Module(name,sortmap,restrictwith(opmap,add(opname,
emptyset)),modlist) .
 eq isemptyinstantiate(reg-Module(name,sortmap,opmap,
modlist)) = isempty(modlist) .
 eq lastmodule(reg-Module(name,sortmap,opmap,modlist)) =
last(modlist) .
 eq delinstmod(mod,reg-Module(name,sortmap,opmap,modlist))
= reg-Module(name,sortmap,opmap,delete(mod,modlist)) .
endo
```

8.2.12 ATOText

O resultado da tradução deste ATO para OBJ é:

```
obj ATOText is
 pr (LIST *(sort List to Text))[STRING] .
 op addline : String Text -> Text
 op addtext : Text Text -> Text .
 op inverttext : Text -> Text .
 var txt : Text .
 var txt1 : Text .
 var line : String .
 eq addline(line,txt) = cons(line,txt) .
 eq addtext(txt,txt1) = concat(txt,txt1) .
 eq inverttext(txt) = invert(txt) .
endo
```

8.3 Considerações Finais

Neste capítulo foram apresentados os estudos de caso que guiaram a validação do ProTool. O estudo de caso da locadora é bem simples e portanto é fácil de entender a estratégia de tradução e como devem serem realizadas as reduções de termos algébricos no ambiente OBJ3.

É importante destacar que os ATOs do ProTool foram validados no OBJ3 antes de passar para a fase de implementação dos mesmos no ambiente PROSOFT-Java. A utilização da tradução proposta possibilitou que erros encontrados nas especificações do ProTool fossem corrigidos antes de passar para a fase de implementação. Consequentemente, no decorrer da implementação do modelo em PROSOFT-Java, o autor não precisou retornar às especificações, economizando tempo na atividade de programação.

No próximo capítulo são apresentadas as conclusões resultantes desta dissertação de mestrado, assim como os trabalhos que podem serem realizados a partir desta.

9 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi apresentada a proposta de um mecanismo para prototipação de especificações de tipos abstratos de dados do PROSOFT-algébrico em OBJ. A geração de protótipos fornece recursos para validar, por execução, os requisitos funcionais do software em desenvolvimento.

Validar por execução é relevante pois uma especificação pode capturar erroneamente os requisitos dos usuários. Em outras palavras, nem sempre a especificação formal não corresponde a uma solução do problema proposto pelo usuário do software. Assim sendo, somente através da experimentação de um sistema executável é possível detectar tais falhas.

O modelo proposto é um recurso muito valioso para auxiliar a validação dos requisitos funcionais, e portanto, evitar que erros sejam propagados para a fase de implementação do ATO algébrico. Como mostrado em (BOEHM, 1981; DAVIS, 1993), uma validação mais cedo diminui os custos e o tempo de desenvolvimento, além de aumentar a qualidade do software.

Em (WYK, 2000; RUS et al., 1997; WYK, 2003) é mostrado de maneira bem abstrata a possibilidade de mapear diferentes linguagens através do conceito de *compilação algébrica*. Uma das contribuições deste trabalho é a definição concreta do mapeamento entre duas linguagens de especificação (PROSOFT-algébrico e OBJ) o que demonstra a viabilidade de estabelecer tal mapeamento.

Este trabalho não traz nenhuma contribuição direta para as abordagens e técnicas de construção de protótipos de software, mas disponibiliza uma ferramenta de prototipação que emprega uma notação formal ergonômica devido a representação gráfica das classes dos ATOs.

Para estabelecer o mapeamento, foi necessário um estudo aprofundado da linguagem algébrica do PROSOFT, resultando na definição semântica dos tipos de dados do PROSOFT e suas respectivas operações algébricas em termos das especificações OBJ. Este estudo também proporcionou a documentação de características do PROSOFT que não haviam sido documentadas até então. Esta documentação servirá de instrumento para os futuros e atuais membros do grupo de pesquisa PROSOFT compreenderem melhor os recursos da linguagem.

Foi proposto um editor de ATOs algébricos e a correspondente tradução para OBJ, sendo que todas estruturas envolvidas foram especificadas formalmente. A partir da especificação foi derivada a implementação de um protótipo, o qual permitiu tanto a experimentação de estudos de caso, quanto a validação do mapeamento proposto. A implementação do ProTool em PROSOFT-Java é uma ferramenta que fornece recursos para o usuário editar seus ATOs e gerar código OBJ que prototipa estes ATOs.

Para garantir matematicamente que o mapeamento entre as duas linguagens está cor-

reto, relaciona-se as sintaxes e as álgebras das linguagens, estabelecendo a prova formal da tradução. Uma vez que a semântica formal do PROSOFT-algébrico não existe, não foi possível estabelecer tal prova.

Como trabalho futuro, propõe-se a extensão da linguagem algébrica. Possíveis recursos a serem adicionados são: importação de ATOs mais avançada, definição de ATOs parametrizados, estratégias para avaliação de operações, precedência e atributos (comutatividade, associatividade, etc.) de operações. Além destas extensões, propõe-se a definição da semântica operacional para o PROSOFT-algébrico.

Outro ponto interessante que poderia ser agregado à notação PROSOFT-algébrico é o conceito de termos visuais. Em (BARDOHL; CLASEN, 1994) é apresentado um método que torna possível representar graficamente as operações algébricas. Combinando o conceito de instanciação gráfica das classes dos ATOs algébricos com uma linguagem gráfica para representação de operações sobre estas classes, torna a especificação algébrica mais fácil de ser lida e interpretada tanto pelos desenvolvedores quanto pelos usuários do software em construção.

Outros trabalhos futuros consistem no aprimoramento do gerador de código PROSOFT-Java a partir dos ATOs algébricos. A versão atual deste gerador considera somente a classe do ATO algébrico, enquanto que também deveria levar em conta as operações definidas no ATO. Mesmo que este aprimoramento seja proposto e implementado, o especificador ainda teria que codificar manualmente toda interface gráfica do usuário. Para superar este obstáculo, poderia também ser desenvolvido um gerador (ou assistente) de interfaces para os ATOs implementados. Deste modo, o usuário do ambiente PROSOFT contaria com uma ferramenta extremamente poderosa na construção de software, e conseqüentemente poderia alocar muito mais tempo para a fase de análise do problema a ser solucionado.

REFERÊNCIAS

ALAVI, M. An Assessment of the Prototyping Approach to Systems. **Communications of the ACM**, New York, v.27, n.6, p.556–564, June 1984.

ALVES, R. **Uma Proposta de Apoio para Decisões de Grupo no Ambiente PROSOFT**. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

ARMSTRONG, J.; VIRDING, S.; WILLIAMS, M. Use of Prolog for Developing a New Programming Language. In: CONF. ON THE PRACTICAL APPLICATION OF PROLOG, 1., 1992, London, England. **Proceedings...** [S.l.] Association for Logic Programming, 1992.

ASTESIANO, E.; BIDOIT, M.; KIRCHNER, H.; KRIEG-BRÜCKNER, B.; MOSSES, P.; SANNELLA, D.; TARLECKI, A. CASL: the common algebraic specification language. **Theoretical Computer Science**, Amsterdam, v.286, n.2, p.153–196, Sept. 2002.

BALZERT, H. Vom singulären Werkzeug zur integrierten Software-Entwicklungsumgebung. **Angewandte Informatik**, Berlin, v.29, n.5, p.175–184, 1987.

BARDOHL, R.; CLASEN, I. Graphical Support for Prototyping of Algebraic Specifications. In: GI JAHRESTAGUNG, 1994, Hamburg. **Proceedings...** [S.l.]: Springer-Verlag, 1994. p.19–26.

BASILI, V.; TURNER, A. Iterative Enhancement: a pratical technique for software development. **IEEE Transactions on Software Engineering**, New York, v.1, n.4, p.390–396, 1975.

BASILI, V.; WEISS, D. Evaluation of a Software Requirements Documents by Analysis of Change Data. In: ICSE, 5., 1981, San Diego. **Proceedings...** [S.l.: s.n.], 1981.

BOLOGNESI, T.; BFINKSMA, E. Introduction to the ISO Specification Language LOTOS. **Computer Networks ISDN Systems**, Amsterdam, v.14, p.25–59, 1987.

BOEHM, B. **Software Engineering Economics**. Englewood Cliffs: Prentice Hall, 1981.

BOEHM, B. A Spiral Model of Software Development and Enhancement. **IEEE Computer**, New York, v.21, n.5, p.61–72, May 1988.

BOEHM, B.; GRAY, T.; SEEWALDT, T. Prototyping Versus Specifying: a multiproject experiment. **IEEE Transactions on Software Engineering**, New York, v.10, n.3, p.290–402, 1984.

BOEHM, B. W.; MCCLEAN, R. K.; URFRIG, D. B. Some Experience with Automated Aids to the Design of Large-Scale Reliable Software. **IEEE Transactions on Software Engineering**, New York, v.1, n.1, p.125–133, 1975.

BOROVANSKY, P. et al. **ELAN**: user manual. Disponível em: <citeseer.nj.nec.com/article/borovansky00elan.html>. Acesso em: 04 fev. 2003.

BRAND, M. et al. The ASF+SDF Meta-environment: a component-based language development environment. In: COMPUTATIONAL COMPLEXITY, 2001, New York. **Proceedings...** [S.l.: s.n.], 2001. p.365–370.

BUDDE, R.; KAUTZ, K.; KUHLENKAMP, K.; ZULLIGHOVEN, H. What is Prototyping? **Information Technology and People**, Northwind, v.6, n.2+3, 1992.

CLARKE, E. et al. Formal methods: state of the art and future directions. **ACM Computing Surveys**, [S.l.], v.28, n.4, p.626–643, 1996.

COHEN, B. A Justification of Formal Methods for System Specification. **Software Engineering Journal**, [S.l.], v.4, n.1, p.26–35, Jan. 1989.

CROW, J. et al. **NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems**. Washington, DC: NASA Office of Safety and Mission Assurance, 1995. v.1.

CURTIS, B.; KRASNER, H.; ISCOE, N. A field study of the software design process for large systems. **Communications of The ACM**, New York, v.31, n.11, p.1268–1287, 1988.

DAVIS, A.; BERSOFF, E.; COMER, E. A Strategy for Comparing Alternative Software Development Life Cycle Models. **IEEE Transactions on Software Engineering**, New York, v.14, n.10, p.1453–1461, 1988.

DAVIS, A. M. **Software Requirements**: objects, functions and states. Englewood Cliffs: Prentice Hall, 1993.

EHRIG, H.; ENGELS, G.; KREOWSKI, H.; ROZENBERG, G. **Handbook of Graph Grammars and Computing by Graph Transformation**. [S.l.]: World Scientific, 1999. v.2.

EHRIG, H.; MAHR, B. **Fundamentals of Algebraic Specification I**. Berlin: Springer-Verlag, 1985.

FLOYD, C. A Systematic Look at Prototyping. In: APPROACHES TO PROTOTYPING, 1984, Berlin. **Proceedings...** [S.l.]: Springer-Verlag, 1984. p.1–18.

GEORGE, C. The RAISE Specification Language: a tutorial. In: VDM, 1991. **Proceedings...** [S.l.: s.n.], 1991. p.238–319. (Lecture Notes in Computer Science, v.551).

GOGUEN, J. A.; COLEMAN, D.; GALLIMORE, R. **Application of Algebraic Specification Using OBJ**. [S.l.]: Cambridge University Press, 1992.

GOGUEN, J. A.; MESEGUER, J. Order-Sorted Algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. **Theoretical Computer Science**, [S.l.], v.105, n.2, p.217–273, 1992.

GOGUEN, J. **Introducing TRIM**. [S.l.]: Dept. of Computer Science and Statistics, University of Rhode Island, 2001. (TR01-283).

GOGUEN, J. et al. Introducing OBJ. In: GOGUEN, J. (Ed.). **Applications of Algebraic Specification using OBJ**. [S.l.]: Kluwer, 2000.

GOGUEN, J.; LIN, K.; ROSU, G.; MORI, A.; WARINSCHI, B. An overview of the Tatami project. In: FUTATSUGI, K.; TAMAI, T.; NAKAGAWA, A. (Ed.). **Cafe**: an industrial-strength algebraic formal method. [S.l.]: Elsevier, 2000. p.61–78.

GOMAA, H. The impact of rapid Prototyping on Specifying User Requirements. **ACM SIGSOFT Software Engineering Notes**, [S.l.], v.8, n.2, p.17–28, 1983.

GORDON, V.; BIEMAN, J. Rapid Prototyping: lessons learned. **IEEE Software**, [S.l.], v.12, n.1, p.85–95, 1994.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. **The Java Language Specification. 2nd ed.** Boston, Mass.: Addison-Wesley, 2000.

GUTTAG, J.; HORNING, J. **Larch**: languages and tools for formal specification. New York: Springer-Verlag, 1993.

HANSEN, H. The ACT-System: experiences and future enhancements. In: RECENT TRENDS IN DATA TYPE SPECIFICATIONS, WADT, 1987, Berlin. **Proceedings...** [S.l.]: Springer-Verlag, 1987. p.111–130. (Lecture Notes in Computer Science, v.332).

HENNESSY, M. **Algebraic Theory of Processes**. Cambridge: MIT Press, 1988.

HOARE, C. An Overview of Some Methods for Program Design. **IEEE Computer**, [S.l.], v.20, n.9, p.85–91, Aug. 1987.

HUDAK, P.; PETERSON, J.; FASEL, J. **A Gentle Introduction to Haskell 98**. Disponível em: <www.haskell.org/tutorial/>. Acesso em: 17 ago. 2001.

INCE, D.; HEKMATPOUR, S. Software Prototyping: progress and prospects. **Information and Software Technology**, [S.l.], v.29, n.1, p.9–14, 1987.

JACKSON, M. **System Design**. [S.l.]: Prentice-Hall International, 1985.

JONES, C. B. **Systematic Software Development using VDM**. Englewood Cliffs: Prentice-Hall, 1986. 300p.

JONES, M. P.; PETERSON, J. C. **Hugs 98**: a functional programming system based on haskell 98. Disponível em: <citeseer.nj.nec.com/jones99hugs.html>. Acesso em: 20 ago. 2001.

KAUSHAAR, J.; SHIRLAND, L. A Prototyping Method for Applications Development End Users and Information Systems Specialists. **MIS Quartely**, [S.l.], v.9, n.4, p.189–197, 1985.

KEUS, H. Prototyping: a more reasonable approach to system development. **ACM SIGSOFT Software Engineering Notes**, [S.l.], v.7, n.5, 1982.

KIEBACK, A.; LICHTER, H.; SCHNEIDER-HUFSCHMIDT, M.; ZüLLIGHOVEN, H. Prototypen in industriellen Software Projekten –Erfahrungen und Analysen. **Informatik-Spektrum**, [S.l.], v.15, n.2, p.65–77, 1992.

LEEUWEN, J. van. **Handbook of Theoretical Computer Science**. Cambridge: MIT Press, 1994. v.b.

LUQI; GOGUEN, J. A. Formal methods: promises and problems. **IEEE Software**, [S.l.], v.14, n.14, p.73–85, Jan. 1997.

LUQI, W. R. Status report: computer-aided prototyping. **IEEE Software**, [S.l.], v.9, n.6, p.77–81, 1992.

LUTZ, R. R. Targeting safety-related errors during software requirements analysis. **Proceedings of ACM SIGSOFT Software Engineering Notes**, New York, v.18, n.5, p.99–106, 1993.

MALCOLM, G.; GOGUEN, J. An Executable Course in the Algebraic Semantics of Imperative Programs. In: DEAN, C. N.; HINCHEY, M. G. (Ed.). **Educational Issues of Formal Methods**. [S.l.]: Academic Press, 1996.

MAYHEW, P.; DEARNLEY, P. An Alternative Prototyping Classification. **The Computer Journal**, [S.l.], v.40, n.6, p.481–484, 1987.

MILNER, R. **Communication and Concurrency**. [S.l.]: Prentice Hall, 1989.

MILNER, R.; PARROW, J. A calculus for mobile process I. **Information and Computation**, [S.l.], v.100, p.1–40, 1992.

MILNER, R.; PARROW, J.; WALKER, D. A calculus for mobile process II. **Information and Computation**, [S.l.], v.100, p.41–77, 1992.

MONTANARI, U.; PISTORE, M. **History-Dependent Automata**. [S.l.: s.n.], 1998. (TR-98-11).

MORAES, S. **Um Ambiente Expert para Apoio ao Desenvolvimento de Software**. 1997. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

NAUMAN, J.; JENKINS, M. Prototyping: the new paradigm for systems development. **MIS Quately**, [S.l.], v.6, n.3, p.29–44, 1982.

NUNES, D. **Projeto PROSOFT**. 2003. Em desenvolvimento. PPGC-UFRGS, Porto Alegre.

NUNES, D. J. Estratégia Data-Driven no Desenvolvimento de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 6., 1992, Gramado. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1992. v.1, p.81–85.

NUNES, D. J. **PROSOFT**: descrição do ambiente. 1994. Relatório de Pesquisa. PPGC-UFRGS, Porto Alegre.

NUNES, I. **Componentes de percepção para o ambiente PROSOFT cooperativo**. 2001. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

PALVIA, P.; NOSEK, J. An Empirical Evaluation of System Development Methodologies. **Information Resources Management Journal**, [S.l.], v.3, p.23–32, 1990.

PAUL, J.; SIMON, G. **Bugs in the Program**: problems in federal government computer software development and regulation. Staff Study by the Subcommittee on Investigations and Oversight. Sept. 1989. Disponível em: <http://citeseer.nj.nec.com/ncontextsummary/1417564/0>. Acesso em: 09 fev. 2002.

PAULK, M.; CURTIS, B.; CHRISSIS, M.; WEBER, C. **Capability Maturity Model for Software, Version 1.1**. [S.l.]: Software Engineering Institute, 1993. (CMU/SEI-93-TR-24, DTIC Number ADA263403).

PETERSON, J. **Petri Net Theory and the Modeling of Systems**. Englewood Cliffs, NJ: Prentice-Hall, 1981.

PIMENTA, A. **Especificação Formal de uma Ferramenta de Reutilização de Especificações de Requisitos**. 1998. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

RABELO, A. **APSEE-Monitor**: um mecanismo de apoio a visualização de processos de software. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

RANGEL, G. S. **Estudo de Abordagens e Técnicas para Prototipação de Software**. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

REIS, C. **Um Gerenciador de Processos de Software para o Ambiente PROSOFT**. 1998. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REIS, C. **Uma Abordagem Flexível para Execução de Processos de Software Evolutivos**. 2003. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REIS, R. **Uma Proposta de Suporte ao Desenvolvimento Cooperativo de Software no Ambiente Prosoft**. 1998. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REIS, R. **APSEE-Reuse**: um meta-modelo para apoiar a reutilização de processos de software. 2003. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REISIG, W. **Petri Nets**: an introduction. Berlin: Springer-Verlag, 1985. (EATCS Monographs on Theoretical Computer Science, v.4).

ROYCE, W. Managing the Development of Large Software Systems. **IEEE WESCON**, [S.l.], p.1–9, Aug. 1970.

ROZENBERG, G. **Handbook of Graph Grammars and Computing by Graph Transformation**. [S.l.]: World Scientific, 1997. v.1.

RUS, T.; HALVERSON, T.; WYK, E. V.; KOOIMA, R. An Algebraic Language Processing Environment. In: PROC. INTERNATIONAL CONFERENCE ON ALGEBRAIC METHODOLOGY AND SOFTWARE TECHNOLOGY, AMAST, 6., 1997, Sydney. **Algebraic Methodology and Software Technology**: proceedings**.** Berlin: Springer-Verlag, 1997. p.581–585.

SALLIS, P.; TATE, G.; MCDONELL, S. **Software Engineering**. Boston: Addison Wesley, 1995.

SCHLEBBE, H. **Java-PROSOFT Manual**. [S.l.]: Universität Stuttgart, 2002.

SCHLEBBE, H.; SCHIMPF, S. **Reenginering of PROSOFT in Java**. Stuttgart: Universität Stuttgart, Fakultät Informatik, 1997.

SOMMERVILLE, I. **Software Engineering**. Harlow: Addison-Wesley, 2001.

SPIVEY, J. **The Z notation**: a reference manual. [S.l.]: Prentice Hall, 1992.

TATE, G. Prototyping: helping to build the right software. **Information and Software Technology**, [S.l.], v.32, n.4, p.237–244, 1990.

TATE, G.; VERNER, J. Case Study of Rsk Management, Incremental Development and Evolutionary Prototyping. **Information and Software Technology**, [S.l.], v.42, n.4, p.207–214, 1990.

TAVOLATO, P.; VINCENA, K. A Prototyping Method and Its Tool. In: APPROACHES TO PROTOTYPING, 1984. **Proceedings...** Berlin: Springer-Verlag, 1984. p.434–446.

WATT, D. **Programming Language Syntax and Semantics**. New York: Prentice-Hall, 1991.

WOOD, D. P.; KANG, K. C. **A Classification and Bibliography of Software Prototyping**. [S.l.]: Software Engineering Institute - Carnegie Mellon University, 1992.

WYK, E. V. Meta Languages in Algebraic Compilers. In: ALGEBRAIC METHODOLOGY AND SOFTWARE TECHNOLOGY, 2000, Iowa City. **Proceedings...** Heidelberg: Springer-Verlag, 2000. p.119–134. (Lecture Notes in Computer Science, v.1816).

WYK, E. V. Specification Languages in Algebraic Compilers. **Theoretical Computer Science**, [S.l.], v.231, n.3, p.351–385, Jan. 2003.

APÊNDICE A OPERAÇÕES DOS TIPOS DO PROSOFT-ALGÉBRICO

Este anexo é responsável por mostrar as operações algébricas dos tipos de dados (primitivos e compostos) que fazem parte da notação PROSOFT-algébrico.

As operações apresentam-se na forma: nome da operação e sua respectiva funcionalidade (rank). Com intuito de facilitar a interpretação semântica das operações, são mostrados exemplos informais de uso de cada operação algébrica. O leitor interessado pode encontrar a semântica formalmente definida para cada operação no anexo 2.

O símbolo "_" nas operações abaixo indica o *place-holder* (posição ou lugar) onde cada argumento está localizado dentro da operação sob a sintaxe mixfix. Quando não aparecer este símbolo, então a operação tem sintaxe pré-fixada, caso não seja um construtor.

A.1 Tipo Conjunto (Set)

```
emptyset        -> Set
add             Element Set -> Set
_U_             Set Set -> Set
_belongsto_     Element Set -> Boolean
cardinality     Set -> Integer
complement      Set Set -> Set
_contain_       Set Set -> Boolean
delete          Element Set -> Set
equal           Set Set -> Boolean
_intersection_  Set Set -> Set
isempty         Set -> Boolean
```

Exemplos:

```
{}
add(x1,{x2,x3})={x1,x2,x3}
{x1,x2}U{x3}={x1,x2,x3}
x1 belongsto {x1,x2,x3} = true
cardinality({x1,x2,x3}) =3
complement({x1,x2,x3},{x2}) ={x1,x3}
{x1,x2,x3}contain{x1,x3} =true
delete(x2,{x1,x2,x3}) ={x1,x3}
equal({x1,x2},{x3}) =false
{x1,x2,x3} intersection {x1,x3} ={x1,x3}
isempty({x1}) =false
```

A.2 Tipo Composto Lista (List)

```
emptylist      -> List
cons           Element List -> List
concat         List List -> List
elements       List -> Set
head           List -> Element
last           List -> Element
length         List -> Integer
projection     List Integer -> Element
replace        List Integer Element -> List
tail           List -> List
isin           Element Element -> Boolean
delete         Element List -> List
invert         List -> List
isempty        List -> Boolean
```

Exemplos:

```
<>
cons(x1,<>) =<x1>
concat(<x1,x2>,<x1>) =<x1,x2,x1>
elements(<x1,x2,x3>) ={x1,x2,x3}
head(<x1,x2,x3>) =<x1>
last(<x1,x2,x3>) =x3
length(<x1,x2,x3>) =3
projection(<x1,x2,x3>,2) =x2
replace(<x1,x2,x3>,2,x5) =<x1,x5,x3>
tail(<x1,x2,x3>) =<x2,x3>
isin(x2,<x1,x2,x3>) =true
delete(x2,<x1,x2,x3>) =<x1,x3>
invert(<x1,x2,x3>) =<x3,x2,x1>
isempty(<>) =true
```

A.3 Tipo Composto Mapeamento (Map)

```
emptymap     -> Map
modify       Domain Range Map -> Map
composition  Map Map -> Map
domain       Map -> Set
imageof      Domain Map -> Range
merge        Map Map -> Map
override     Map Map -> Map
range        Map -> Set
restrictto   Map Set -> Map
restricwith  Map Set -> Map
isempty      Map -> Boolean
```

Exemplos:

```
[]
modify(x1,y1,[]) =[x1->y1]
```

```
composition([x1->y1],[x2->y2]) =[x1->y2]
domain([x1->y1,x2->y2]) ={x1,x2}
imageof(x1,[x1->y1,x2->y2]) =y1
merge([x1->y1],[x2->y2]) =[x1->y1,x2->y2]
override([x1->y1,x2->y2],[x1->y3]) =[x1->y3,x2->y2]
range([x1->y1,x2->y2]) ={y1,y2}
restrictto([x1->y1,x2->y2,x3->y3],{x2}) =[x2->y2]
restrictwith([x1->y1,x2->y2,x3->y3],{x2}) =[x1->y1,x3->y3]
isempty([]) =true
```

A.4 Tipo Composto Registro (Register)

```
reg         D1 D2 D3 Dn -> Register
select-D1   Register -> D1
select-D2   Register -> D2
select-Dn   Register -> Dn
```

Exemplos:

```
reg-Type(2,true)
select-Integer(reg-Type(2,true))=2
select-Boolean(reg-Type(2,true))=true
```

A.5 Tipo Composto União Disjunta (DisjointUnion)

```
DisjointUnion-D1    D1 -> DisjointUnion
DisjointUnion-D2    D2 -> DisjointUnion
DisjointUnion-Dn    Dn -> DisjointUnion
get-D1              DisjointUnion -> D1
get-D2              DisjointUnion -> D2
get-Dn              DisjointUnion -> Dn
is-D1               DisjointUnion -> Boolean
is-D2               DisjointUnion -> Boolean
is-Dn               DisjointUnion -> Boolean
```

Exemplos:

```
Type-Number(2)
Type-Truth(false)
get-Number(Type-Number(2)) =2
get-Truth(Type-Truth(false)) =false
is-Number(Type-Number(2)) =true
is-Truth(Type-Truth(false)) =true
```

A.6 Tipo Primitivo Inteiro (Integer)

```
-_   Integer -> Integer            -4
_+_  Integer Integer -> Integer     3 + 4 =7
_-_  Integer Integer -> Integer     5 - 3 =2
_*_  Integer Integer -> Integer     3 * 3 =9
_<_  Integer Integer -> Boolean     2 < 3 =true
```

```
_<=_ Integer Integer -> Boolean     0 <= 0 =true
_>_  Integer Integer -> Boolean     9 > 6 =false
_>=_ Integer Integer -> Boolean     7 >= 8 =false
```

A.7 Tipo Primitivo String

```
"_"     String String -> String
length  String -> Integer
_++_    String String -> String
get     String Integer -> String
isin    String String -> Boolean
lcase   String -> String
ucase   String -> String
locate  String String -> Integer
rep1    String String String -> String
inpos   Integer String -> String
extract Integer Integer String -> String
```

Exemplos:

```
"PROSOFT"
length("PROSOFT") =7
"PRO" ++ "SOFT" ="PROSOFT"
get("PROSOFT",3) ="O"
isin("abc", "fg54abcty") = true
lcase("aBc") ="abc"
ucase("AbC") ="ABC"
locate("_","if_then_else_fi") =3
rep1("_"," var ","if_then_else_fi") ="if var then_else_fi"
inpos(3,"guilherme") ="i"
extract(3,5,"guilherme") ="ilh"
```

A.8 Tipo Primitivo Data (Date)

```
date        Integer Integer Integer -> Date
get-day     Date -> Integer
get-month   Date -> Integer
get-year    Date -> Integer
toint       Date -> Integer
isvalid     Date-> Boolean
```

Exemplos:

```
date(4,6,1977)
get-day(date(4,6,1977)) =4
get-month(date(4,6,1977)) =6
get-year(date(4,6,1977)) =1977
toint(date(4,6,1977)) = 721789
isvalid(date(4,6,1977)) =true
```

A.9 Tipo Primitivo Hora (Time)

```
time        Integer Integer Integer -> Time
get-hour    Time -> Integer
get-min     Time -> Integer
get-sec     Time -> Integer
toint       Time -> Integer
isvalid     Time -> Boolean
```

Exemplos:

```
time(16,04,55)
get-hour(time(16,04,55)) =16
get-min(time(16,04,55)) =04
get-sec(time(16,04,55)) =55
toint(time(16,04,55)) =57895
isvalid(time(16,04,55)) =true
```

A.10 Tipo Primitivo Boolean

```
true              -> Boolean
false             -> Boolean
_and_             Boolean Boolean -> Boolean
_or_              Boolean Boolean -> Boolean
_xor_             Boolean Boolean -> Boolean
not_              Boolean -> Boolean
_implies_         Boolean Boolean -> Boolean
_==_              Universal Universal -> Boolean
_=/=_             Universal Universal -> Boolean
if_then_else_fi   Boolean Universal Universal -> Universal
```

Exemplos:

```
true
false
true and false = false
true or false = true
true xor false = true
not true = false
true implies false = false
5 == 4  = false
5 =/= 4 = true
if true then 5 else 4 fi = 5
```

APÊNDICE B ESPECIFICAÇÕES EM OBJ

Neste anexo são mostradas as especificações em OBJ que dão semântica aos tipos de dados do PROSOFT-algébrico. Nas próximas seções o leitor pode encontrar as especificações de listas, conjuntos, mapeamentos, registros, uniões disjuntas, string, data e hora. Os tipos Boolean e Integer da notação PROSOFT foram reutilizados identicamente como foram definidos para OBJ, e podem ser encontrados em (GOGUEN et al., 2000).

Ressalta-se que as especificações dos tipos primitivos data e hora são consideradas uma simplificação, uma vez que para definição dos conceitos de hora, minuto, segundo, dia, mês e ano, deveriam ser utilizados subconjuntos dos números inteiros para limitar tais valores.

B.1 Especificação Parametrizada para Conjuntos

```
obj SET[X :: TRIV] is
  sort Set .
  pr INT .
  op emptyset : -> Set .
  op add : Elt Set -> Set .
  op _U_ : Set Set -> Set .
  op _belongsto_ : Elt Set -> Bool .
  op cardinality : Set -> Int .
  op _intersection_ : Set Set -> Set .
  op delete : Elt Set -> Set .
  op equal : Set Set -> Bool .
  op _contain_ : Set Set -> Bool .
  op complement : Set Set -> Set .
  op isempty : Set -> Bool .
  var S : Set .
  vars S1 S2 : Set .
  var X : Elt .
  var E : Elt .
  var E1 E2 : Elt .

  eq isempty(S) = if S == emptyset then true else false fi .
  eq complement(S,emptyset) = S .
  eq complement(S,add(E1,S1)) = if (E1 belongsto S) then
complement(delete(E1,S),S1) else complement(S,S1) fi .
  eq emptyset contain emptyset = false .
  eq emptyset contain add(E,S) = false .
```

```
  eq add(E,S) contain emptyset = true .
  eq add(E,S) contain add(E1,S1) = if ((E1 == E) or (E1
belongsto S)) then (true and (add(E,S) contain S1)) else false fi
.
  eq equal(emptyset,emptyset) = true .
  eq equal(add(E,S),emptyset) = false .
  eq equal(emptyset,add(E,S)) = false .
  eq equal(add(E,S),add(E1,S1)) = if ((E == E1) or (E
belongsto S1)) then (true and equal(S,add(E1,S1))) else false fi .
  eq X belongsto emptyset = false .
  eq X belongsto add(E,emptyset) = (X == E) .
  eq X belongsto add(E,S) = if X == E then true else (X
belongsto S) fi .
  eq cardinality(emptyset) = 0 .
  eq cardinality(add(E,emptyset)) = 1 .
  eq cardinality(add(E,S)) = 1 + cardinality(S) .
  eq emptyset U S = S .
  eq S U emptyset = S .
  eq add(E,S1) U S2 = if (E belongsto S2) then (S1 U S2)
else (S1 U add(E,S2)) fi .
  eq emptyset intersection S = emptyset .
  eq S intersection emptyset = emptyset .
  eq add(E,S1) intersection S2 = if (E belongsto S2) then
(add(E,emptyset) U (S1 intersection S2)) else (S1 intersection S2)
fi .
  eq delete(E,emptyset) = emptyset .
  eq delete(E2,add(E1,S)) = if (E1 == E2) then S else
(delete(E2,S) U add(E1,emptyset)) fi . endo
```

B.2 Especificação Parametrizada para Listas

```
obj LIST[C :: TRIV] is
  sort List .
  pr INT .
  pr SET[C] .
  op emptylist : -> List .
  op cons : Elt List -> List .
  op elements : List -> Set .
  op head : List -> Elt .
  op last : List -> Elt .
  op length : List -> Int .
  op projection : List Int -> Elt .
  op replace : List Int Elt -> List .
  op tail : List -> List .
  op concat : List List -> List .
  op isin : Elt List -> Bool .
  op delete : Elt List -> List .
  op invert : List -> List .
  op isempty : List -> Bool .
  var c : Elt .
  var c1 : Elt .
```

```
  var l : List .
  var l1 : List .
  var n : Int .
  eq isempty(l) = if l == emptylist then true else false fi .
  eq elements(emptylist) = emptyset .
  eq elements(cons(c,l)) = add(c,elements(l)) .
  eq last(cons(c,l)) = if l == emptylist then c else last(l) fi .
  eq head(cons(c,l)) = c .
  eq length(emptylist) = 0 .
  eq length(cons(c,l)) = length(l) + 1 .
  eq tail(cons(c,l)) = l .
  eq tail(emptylist) = emptylist .
  eq projection(cons(c,l),n) = if n == 1 then c else projection(
tail(cons(c,l)),n - 1) fi .
  eq replace(emptylist,n,c1) = emptylist .
  eq replace(cons(c,l),n,c1) = if n == 1 then cons(c1,l) else
cons(c,replace(l,n - 1,c1)) fi .
  eq concat(l,emptylist) = l .
  eq concat(emptylist,l) = l .
  eq concat(cons(c,l),l1) = cons(c,concat(l,l1)) .
  eq isin(c,emptylist) = false .
  eq isin(c1,cons(c,l)) = if (c == c1) then true else isin(c1,
l) fi .
  eq delete(c,emptylist) = emptylist .
  eq delete(c1,cons(c,l)) = if (c == c1) then l else cons(c,
delete(c1,l)) fi .
  eq invert(emptylist) = emptylist .
  eq invert(cons(c,l)) = concat(invert(l),cons(c,emptylist)) .
endo
```

B.3 Especificação Parametrizada para Mapeamento

```
obj MAP[C1 :: TRIV,C2 :: TRIV] is
    sort Map .
    pr (SET *(sort Set to SetD))[C1] .
    pr (SET *(sort Set to SetR))[C2] .
    op emptymap : -> Map .
    op modify : Elt.C1 Elt.C2 Map -> Map .
    op imageof : Elt.C1 Map -> Elt.C2 .
    op domain : Map -> SetD .
    op range : Map -> SetR .
    op merge : Map Map -> Map [comm].
    op restrictto : Map SetD -> Map .
    op restrictwith : Map SetD -> Map .
    op override : Map Map -> Map .
    op composition : Map Map -> Map .
    op isempty : Map -> Bool .
    var d d1 : Elt.C1 .
    var r r1 : Elt.C2 .
    var m m1 : Map .
    var sd : SetD .
```

```
    eq isempty(m) = if m == emptymap then true else false fi .
    eq imageof(d,modify(d1,r,m)) = if d == d1 then r else
imageof(d,m) fi .
    eq domain(modify(d,r,m)) = add(d,emptyset) U domain(m) .
    eq domain(emptymap) = emptyset .
    eq range(emptymap) = emptyset .
    eq range(modify(d,r,m)) = add(r,emptyset) U range(m) .
    eq merge(m,emptymap) = m .
    eq merge(modify(d,r,m),m1) = modify(d,r,merge(m,m1)) .
    eq restrictwith(modify(d,r,m),sd) = if (d belongsto sd)
then restrictwith(m,delete(d,sd)) else modify(d,r,restrictwith(
m,sd)) fi .
    eq restrictwith(emptymap,sd) = emptymap .
    eq restrictwith(m,emptyset) = m .
    eq restrictto(emptymap,sd) = emptymap .
    eq restrictto(m,emptyset) = emptymap .
    eq restrictto(modify(d,r,m),add(d1,sd)) = if (d1
belongsto domain(modify(d,r,m))) then modify(d1,imageof(d1,
modify(d,r,m)),restrictto(restrictwith(modify(d,r,m),add(d1,
emptyset)),sd)) else restrictto(modify(d,r,m),sd) fi .
    eq override(m,emptymap) = m .
    eq override(m,modify(d1,r1,m1)) = override(merge(restrictwith(
m,add(d1,emptyset)),modify(d1,r1,emptymap)),m1) .
    eq composition(modify(d,r,m),modify(d1,r1,m1)) = modify(d,
r1,composition(m,m1)) .
    eq composition(m,emptymap) = m .
endo
```

B.4 Especificação Parametrizada para Registros

São mostradas somente as especificações parametrizadas para registros de 2 e 3 domínios. As especificações de registros contendo um número de número maior de domínios são criadas seguindo a lógica apresentada.

```
obj REGISTER2[FS1 :: TRIV,FS2 :: TRIV] is
 sort Register2 .
 op reg : Elt.FS1 Elt.FS2 -> Register2 .
 op select1 : Register2 -> Elt.FS1 .
 op select2 : Register2 -> Elt.FS2 .
 var !1r2 : Elt.FS1 .
 var !2r2 : Elt.FS2 .
 eq select1(reg(!1r2,!2r2)) = !1r2 .
 eq select2(reg(!1r2,!2r2)) = !2r2 .
endo

obj REGISTER3[FS1 :: TRIV,FS2 :: TRIV,FS3 :: TRIV] is
 sort Register3 .
 op reg : Elt.FS1 Elt.FS2 Elt.FS3 -> Register3 .
 op select1 : Register3 -> Elt.FS1 .
 op select2 : Register3 -> Elt.FS2 .
 op select3 : Register3 -> Elt.FS3 .
```

```
 var !1r3 : Elt.FS1 .
 var !2r3 : Elt.FS2 .
 var !3r3 : Elt.FS3 .
 eq select1(reg(!1r3,!2r3,!3r3)) = !1r3 .
 eq select2(reg(!1r3,!2r3,!3r3)) = !2r3 .
 eq select3(reg(!1r3,!2r3,!3r3)) = !3r3 .
endo
```

B.5 Especificação Parametrizada para União Disjunta

São mostradas somente as especificações parametrizadas para uniões disjuntas de 2 e 3 domínios. As especificações de uniões disjuntas contendo um número de número maior de domínios são criadas seguindo a lógica apresentada.

```
obj DISJOINTUNION2[FS1 :: TRIV,FS2 :: TRIV] is
 sort Disjointunion2 .
 op apply1 : Elt.FS1 -> Disjointunion2 .
 op apply2 : Elt.FS2 -> Disjointunion2 .
 op is1 : Disjointunion2 -> Bool .
 op is2 : Disjointunion2 -> Bool .
 op get1 : Disjointunion2 -> Elt.FS1 .
 op get2 : Disjointunion2 -> Elt.FS2 .
 var !1d2 : Elt.FS1 .
 var !2d2 : Elt.FS2 .
 eq is1(apply1(!1d2)) = true .
 eq is1(apply2(!2d2)) = false .
 eq is2(apply1(!1d2)) = false .
 eq is2(apply2(!2d2)) = true .
 eq get1(apply1(!1d2)) = !1d2 .
 eq get2(apply2(!2d2)) = !2d2 .
endo
```

```
obj DISJOINTUNION3[FS1 :: TRIV,FS2 :: TRIV,FS3 :: TRIV] is
 sort Disjointunion3 .
 op apply1 : Elt.FS1 -> Disjointunion3 .
 op apply2 : Elt.FS2 -> Disjointunion3 .
 op apply3 : Elt.FS3 -> Disjointunion3 .
 op is1 : Disjointunion3 -> Bool .
 op is2 : Disjointunion3 -> Bool .
 op is3 : Disjointunion3 -> Bool .
 op get1 : Disjointunion3 -> Elt.FS1 .
 op get2 : Disjointunion3 -> Elt.FS2 .
 op get3 : Disjointunion3 -> Elt.FS3 .
 var !1d3 : Elt.FS1 .
 var !2d3 : Elt.FS2 .
 var !3d3 : Elt.FS3 .
 eq is1(apply1(!1d3)) = true .
 eq is1(apply2(!2d3)) = false .
 eq is1(apply3(!3d3)) = false .
```

```
 eq is2(apply1(!1d3)) = false .
 eq is2(apply2(!2d3)) = true .
 eq is2(apply3(!3d3)) = false .

 eq get1(apply1(!1d3)) = !1d3 .
 eq get2(apply2(!2d3)) = !2d3 .
 eq get3(apply3(!3d3)) = !3d3 .
endo
```

B.6 Especificação Parametrizada para Datas

```
obj DATE is
 sort Date .
 pr INT .
 op date : Int Int Int -> Date .
 op get-day : Date -> Int .
 op get-month : Date -> Int .
 op get-year : Date -> Int .
 op toint : Date -> Int .
 op isvalid : Date -> Bool .
 var !d !m !y : Int .
 eq get-day(date(!d,!m,!y)) = !d .
 eq get-month(date(!d,!m,!y)) = !m .
 eq get-year(date(!d,!m,!y)) = !y .
 eq toint(date(!d,!m,!y)) = (365 * !y) +
(30 * !m) + !d .
 eq isvalid(date(!d,!m,!y)) = (1 <= !d) and
(!d < 31) and (1 <= !m) and (!m < 12) .

endo
```

B.7 Especificação Parametrizada para Horas

```
obj TIME is
 sort Time .
 pr INT .
 op time : Int Int Int -> Time .
 op get-hour : Time -> Int .
 op get-min : Time -> Int .
 op get-sec : Time -> Int .
 op toint : Time -> Int .
 op isvalid : Time -> Bool .
 var !h !m !s : Int .
 eq get-hour(time(!h,!m,!s)) = !h .
 eq get-min(time(!h,!m,!s)) = !m .
 eq get-sec(time(!h,!m,!s)) = !s .
 eq toint(time(!h,!m,!s)) = (3600 * !h) +
(60 * !m) + !s .
 eq isvalid(time(!h,!m,!s)) = (0 <= !h) and
(!h < 24) and (0 <= !m) and (!m < 60) and
```

```
(0 <= !s) and (!s < 60) .
endo
```

B.8 Especificação Parametrizada para String

```
obj STRING is
 sort String .
 sort Symbol .
 pr INT .
 subsort Symbol < String .
 ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Symbol .
 ops A B C D E F G H I J K L M N O P Q R S T U V X Y Z W : -> Symbol .
 ops 0 1 2 3 4 5 6 7 8 9 : -> Symbol .
 ops \! \@ \# \$ \% \^ \& \* \paropen \parclose \- \underline \=
\+ \{{ \}} \[ \] : -> Symbol .
 ops \| \\ \: \; \" \’ \comma \< \> \. \? \/ \‘ \~ \space
\empty : -> Symbol .
 op "_" : String -> String .
 op __ : String String -> String [assoc] .
 op _++_ : String String -> String .
 op length_ : String -> Int .
 op get : String Int -> String .
 op isin : String String -> Bool .
 op aux : String String -> Bool .
 op ucase : String -> String .
 op lcase : String -> String .
 op locate : String String -> Int .
 op rep1 : String String String -> String .
 op inpos : Int String -> String .
 op extract : Int Int String -> String .
 vars !S !S1 !SS !S2 : String .
 vars !A !A1 !A2 : Symbol .
 var !n !n1 !n2 : Int .
 eq inpos(!n,"(!A)") = if !n == (1).Int then "(!A)" else \empty fi .
 eq inpos(!n,"(!A !S)") = if !n == (1).Int then "(!A)" else inpos(!n
- (1).Int,"(!S)") fi .
 cq extract(!n1,!n2,"(!A !S)") = "(!A)" ++ extract(!n1,!n2 - (1).Int,
"(!S)") if !n1 == (1).Int and !n2 > (1).Int .
 cq extract(!n1,!n2,"(!A !S)") = extract(!n1 - (1).Int,!n2 - (1).Int,
"(!S)") if !n1 > (1).Int .
 cq extract(!n1,!n2,"(!A !S)") = "(!A)" if !n1 == (1).Int and
!n2 == (1).Int .
 cq extract(!n1,!n2,"(!A)") = "(!A)" if !n1 == (1).Int and !n2 ==
(1).Int .
 eq rep1("(!A)","(!A1)","(!A2 !S2)") = if !A2 == !A then "(!A1 !S2)"
else "(!A2)" ++ rep1("(!A)","(!A1)","(!S2)") fi .
 eq rep1("(!A)","(!A1 !S1)","(!A2 !S2)") = if !A2 == !A then
"(!A1 !S1 !S2)" else "(!A2)" ++ rep1("(!A)","(!A1 !S1)","(!S2)") fi .
 eq locate("(!A)","(!A1)") = if !A == !A1 then (1).Int else (0).Int fi .
 eq locate("(!A)","(!A1 !S1)") = if !A == !A1 then (1).Int else (1).Int
+ locate("(!A)","(!S1)") fi .
```

```
 cq ucase("(!A)") = "(A)" if !A == a .
 cq ucase("(!A)") = "(B)" if !A == b .
 cq ucase("(!A)") = "(C)" if !A == c .
*** define ucase for all symbols
 eq ucase("(!A !S)") = ucase("(!A)") ++ ucase("(!S)") .
 eq lcase("(!A !S)") = lcase("(!A)") ++ lcase("(!S)") .
 cq lcase("(!A)") = "(a)" if !A == A .
 cq lcase("(!A)") = "(b)" if !A == B .
 cq lcase("(!A)") = "(c)" if !A == c .
*** define lcase for all symbols
 eq "(!S)" ++ "(!S1)" = "(!S !S1)" .
 eq length("(!A)") = 1 .
 eq length("(!A !SS)") = 1 + length("(!SS)") .
 eq !S ++ !SS = !S !SS .
 eq get("(!A !SS)",!n) = if !n == (1).Int then "(!A)" else get("(!SS)",
!n - (1).Int) fi .
 eq isin("(!A !S)","(!A1 !S1)") = if length("(!A !S)") >
length("(!A1 !S1)") then false else if !A == !A1 then
aux("(!S)","(!S1)") else isin("(!A !S)","(!S1)") fi fi .
 eq isin("(!A !S)","(!A1)") = false .
 eq aux("(!A !S)","(!A1 !S1)") = if !A == !A1 then true and
aux("(!S)","(!S1)") else false fi .
 eq aux("(!A)","(!A1 !S)") = if !A1 == !A then true else false fi .
 eq aux("(!A)","(!A1)") = if !A1 == !A then true else false fi .
 eq isin("(!A1)","(!A)") = if !A1 == !A then true else false fi .
 eq isin("(!A1)","(!A !S)") = if !A1 == !A then true else
isin("(!A1)","(!S)") fi .
endo
```

APÊNDICE C OPERAÇÕES PARA CRIAR OS TIPOS DE DADOS DAS CLASSES RECURSIVAS

Neste anexo são apresentados os esboços das operações responsáveis pela criação dos tipos compostos, do PROSOFT-algébrico em OBJ, quando a classe do ATO é recursiva.

Cada operação `create-spec` cria as operações construtoras, modificadoras e observadoras para cada tipo composto do PROSOFT.

C.1 Operação para Conjunto

A operação `create-spec-set` tem como argumentos o sort do conjunto e o sort dos elementos do conjunto. O resultado da computação desta operação é um texto (lista de strings) que contém as operações do tipo conjunto.

Exemplo: dada uma classe recursiva, em determinado ramo da hierarquia da classe deve-se criar um conjunto de strings. O sort deste conjunto é *SetofStrings* e a chamada à operação seria:

```
create-spec-set(SetofStrings,String)=
 pr INT .
 op emptyset : -> SetofStrings .
 op add : String SetofStrings -> SetofStrings .
  ...
```

enquanto que a operação genérica é:

```
create-spec-set(SORT,ELT)=
 pr INT .
 op emptyset : -> SORT .
 op add : ELT SORT -> SORT .
 op _U_ : SORT SORT -> SORT .
 op _belongsto_ : ELT SORT -> Bool .
 op cardinality : SORT -> Int .
 op _intersection_ : SORT SORT -> SORT .
 op delete : ELT SORT -> SORT .
 op equal : SORT SORT -> Bool .
 op _contain_ : SORT SORT -> Bool .
 op complement : SORT SORT -> SORT .
 var S : SORT .
 vars S1 S2 : SORT .
 var X : ELT .
```

```
 var E : ELT .
 var E1 E2 : ELT .
 eq complement(S,emptyset) = S .
 eq complement(S,add(E1,S1)) = if (E1 belongsto S) then
complement(delete(E1,S),S1) else complement(S,S1) fi .
 eq emptyset contain emptyset = false .
 eq emptyset contain add(E,S) = false .
 eq add(E,S) contain emptyset = true .
 eq add(E,S) contain add(E1,S1) = if ((E1 == E) or (E1
belongsto S)) then (true and (add(E,S) contain S1)) else
false fi .
 eq equal(emptyset,emptyset) = true .
 eq equal(add(E,S),emptyset) = false .
 eq equal(emptyset,add(E,S)) = false .
 eq equal(add(E,S),add(E1,S1)) = if ((E == E1) or (E
belongsto S1)) then (true and equal(S,add(E1,S1))) else
false fi .
 eq X belongsto emptyset = false .
 eq X belongsto add(E,emptyset) = (X == E) .
 eq X belongsto add(E,S) = if X == E then true else (X
belongsto S) fi .
 eq cardinality(emptyset) = 0 .
 eq cardinality(add(E,emptyset)) = 1 .
 eq cardinality(add(E,S)) = 1 + cardinality(S) .
 eq emptyset U S = S .
 eq S U emptyset = S .
 eq add(E,S1) U S2 = if (E belongsto S2) then
(S1 U S2) else (S1 U add(E,S2)) fi .
 eq emptyset intersection S = emptyset .
 eq S intersection emptyset = emptyset .
 eq add(E,S1) intersection S2 = if (E belongsto S2) then
(add(E,emptyset) U (S1 intersection S2)) else (S1
intersection S2) fi .
 eq delete(E,emptyset) = emptyset .
 eq delete(E2,add(E1,S)) = if (E1 == E2) then
S else (delete(E2,S) U add(E1,emptyset)) fi .
```

C.2 Operação para Lista

A operação `create-spec-list` é semelhante à operação `create-spec-set` mostrada na seção anterior, e sua definição informal é:

```
create-spec-list(SORT,ELT)=
 pr INT .
 pr SET[ELT] .
 op emptylist : -> SORT .
 op cons : ELT SORT -> SORT .
 op elements : SORT -> Set .
 op head : SORT -> ELT .
 op last : SORT -> ELT .
 op length : SORT -> Int .
```

```
 op projection : SORT Int -> ELT .
 op replace : SORT Int ELT -> SORT .
 op tail : SORT -> SORT .
 op concat : SORT SORT -> SORT .
 op isin : ELT SORT -> Bool .
 op delete : ELT SORT -> SORT .
 op invert : SORT -> SORT .
 var c : ELT .
 var c1 : ELT .
 var l : SORT .
 var l1 : SORT .
 var n : Int .
 eq elements(emptylist) = emptyset .
 eq elements(cons(c,l)) = add(c,elements(l)) .
 eq last(cons(c,l)) = if l == emptylist then c else
last(l) fi .
 eq head(cons(c,l)) = c .
 eq length(emptylist) = 0 .
 eq length(cons(c,l)) = length(l) + 1 .
 eq tail(cons(c,l)) = l .
 eq projection(cons(c,l),n) = if n == 1 then c else
projection(tail(cons(c,l)),n - 1) fi .
 eq replace(emptylist,n,c1) = emptylist .
 eq replace(cons(c,l),n,c1) = if n == 1 then cons(c1,l)
else cons(c,replace(l,n - 1,c1)) fi .
 eq concat(l,emptylist) = l .
 eq concat(emptylist,l) = l .
 eq concat(cons(c,l),l1) = cons(c,concat(l,l1)) .
 eq isin(c,emptylist) = false .
 eq isin(c1,cons(c,l)) = if (c == c1) then true else
isin(c1,l) fi .
 eq delete(c,emptylist) = emptylist .
 eq delete(c1,cons(c,l)) = if (c == c1) then l else
cons(c,delete(c1,l)) fi .
 eq invert(emptylist) = emptylist .
 eq invert(cons(c,l)) = concat(invert(l),cons(c,emptylist)) .
```

C.3 Operação para Mapeamento

A operação `create-spec-map` tem como argumentos o sort do mapeamento e os sorts do domínio e imagem do mapeamento. O resultado da computação desta operação é um texto (lista de strings) que contém as operações do tipo mapeamento.

Exemplo: dada uma classe recursiva, em determinado ramo da hierarquia da classe deve-se criar um mapeamento entre strings. O sort deste mapeamento é *Variables* e a chamada à operação seria:

```
create-spec-map(Variables,String,String)=
 pr (SET *(sort Set to SetD))[String] .
 pr (SET *(sort Set to SetR))[String] .
 op emptymap : -> Variables .
 op modify : String String Variables -> Variables .
```

```
 op imageof : String Variables -> String .
 op domain : Variables -> SetD .
 op range : Variables -> SetR .
 ...
```

enquanto que a operação genérica é:

```
create-spec-map(SORT,DSORT,RSORT)=
 pr (SET *(sort Set to SetD))[DSORT] .
 pr (SET *(sort Set to SetR))[RSORT] .
 op emptymap : -> SORT .
 op modify : DSORT RSORT SORT -> SORT .
 op imageof : DSORT SORT -> RSORT .
 op domain : SORT -> SetD .
 op range : SORT -> SetR .
 op merge : SORT SORT -> SORT [comm].
 op restrictto : SORT SetD -> SORT .
 op restrictwith : SORT SetD -> SORT .
 op override : SORT SORT -> SORT .
 op composition : SORT SORT -> SORT .
 var d d1 : DSORT .
 var r r1 : RSORT .
 var m m1 : SORT .
 var sd : SetD .
 eq imageof(d,modify(d1,r,m)) = if d == d1 then r else
imageof(d,m) fi .
 eq domain(modify(d,r,m)) = add(d,emptyset) U domain(m) .
 eq domain(emptymap) = emptyset .
 eq range(emptymap) = emptyset .
 eq range(modify(d,r,m)) = add(r,emptyset) U range(m) .
 eq merge(m,emptymap) = m .
 eq merge(modify(d,r,m),m1) = modify(d,r,merge(m,m1)) .
 eq restrictwith(modify(d,r,m),sd) = if (d belongsto sd)
then restrictwith(m,delete(d,sd)) else modify(d,r,
restrictwith(m,sd)) fi .
 eq restrictwith(emptymap,sd) = emptymap .
 eq restrictwith(m,emptyset) = m .
 eq restrictto(emptymap,sd) = emptymap .
 eq restrictto(m,emptyset) = emptymap .
 eq restrictto(modify(d,r,m),add(d1,sd)) = if (d1
belongsto domain(modify(d,r,m))) then modify(d1,imageof(d1,
modify(d,r,m)),restrictto(restrictwith(modify(d,r,m),add(d1,
emptyset)),sd)) else restrictto(modify(d,r,m),sd) fi .
 eq override(m,emptymap) = m .
 eq override(m,modify(d1,r1,m1)) = override(merge(
restrictwith(m,add(d1,emptyset)),modify(d1,r1,emptymap)),m1) .
 eq composition(modify(d,r,m),modify(d1,r1,m1)) = modify(d,
r1,composition(m,m1)) .
 eq composition(m,emptymap) = m .
```

C.4 Operação para Registro

A operação `create-spec-reg` tem como argumentos: o sort do registro e duas listas (a primeira com os sorts dos domínios do registro e a segunda com os nomes usados na classe gráfica do ATO. O resultado da computação desta operação é um texto (lista de strings) que contém as operações do tipo registro.

São mostradas somente as definições da operação para registros de 2 e 3 domínios. Registros contendo um número de número maior de domínios são criadas seguindo a lógica apresentada.

```
create-spec-reg(SORT,ALIST,NLIST)=
 op reg-SORT : ALIST.1 ALIST.2 -> SORT .
 op select-NLIST.1 : SORT -> ALIST.1 .
 op select-NLIST.2 : SORT -> ALIST.2 .
 var X : ALIST.1 .
 var Y : ALIST.2 .
 eq select-NLIST.1(reg-SORT(X,Y)) = X .
 eq select-NLIST.2(reg-SORT(X,Y)) = Y .

create-spec-reg(SORT,ALIST,NLIST)=
 op reg-SORT : ALIST.1 ALIST.2 ALIST.3 -> SORT .
 op select-NLIST.1 : SORT -> ALIST.1 .
 op select-NLIST.2 : SORT -> ALIST.2 .
 op select-NLIST.3 : SORT -> ALIST.3 .
 var X : ALIST.1 .
 var Y : ALIST.2 .
 var Z : ALIST.3 .
 eq select-NLIST.1(reg-SORT(X,Y,Z)) = X .
 eq select-NLIST.2(reg-SORT(X,Y,Z)) = Y .
 eq select-NLIST.3(reg-SORT(X,Y,Z)) = Z .
```

C.5 Operação para União Disjunta

A operação `create-spec-du` tem como argumentos: o sort da união disjunta e duas listas (a primeira com os sorts dos domínios disjuntos e a segunda com os nomes usados na classe gráfica do ATO. O resultado da computação desta operação é um texto (lista de strings) que contém as operações do tipo união disjunta.

São mostradas somente as definições da operação para união disjuntas de 2 e 3 domínios. Uniões contendo um número de número maior de domínios disjuntos são criadas seguindo a lógica apresentada.

```
create-spec-du(SORT,ALIST,NLIST)=
 op SORT-NLIST.1 : ALIST.1 -> SORT .
 op SORT-NLIST.2 : ALIST.2 -> SORT .
 op is-NLIST.1 : SORT -> Bool .
 op is-NLIST.2 : SORT -> Bool .
 op get-NLIST.1 : SORT -> ALIST.1 .
 op get-NLIST.2 : SORT -> ALIST.2 .
 var X : ALIST.1 .
 var Y : ALIST.2 .
 eq is-NLIST.1(SORT-NLIST.1(X)) = true .
```

```
 eq is-NLIST.1(SORT-NLIST.2(Y)) = false .
 eq is-NLIST.2(SORT-NLIST.1(X)) = false .
 eq is-NLIST.2(SORT-NLIST.2(Y)) = true .
 eq get-NLIST.1(SORT-NLIST.1(X)) = X .
 eq get-NLIST.2(SORT-NLIST.2(Y)) = Y .

create-spec-du(SORT,ALIST,NLIST)=
 op SORT-NLIST.1 : ALIST.1 -> SORT .
 op SORT-NLIST.2 : ALIST.2 -> SORT .
 op SORT-NLIST.3 : ALIST.3 -> SORT .
 op is-NLIST.1 : SORT -> Bool .
 op is-NLIST.2 : SORT -> Bool .
 op is-NLIST.3 : SORT -> Bool .
 op get-NLIST.1 : SORT -> ALIST.1 .
 op get-NLIST.2 : SORT -> ALIST.2 .
 op get-NLIST.3 : SORT -> ALIST.3 .
 var X : ALIST.1 .
 var Y : ALIST.2 .
 var Z : ALIST.3 .
 eq is-NLIST.1(SORT-NLIST.1(X)) = true .
 eq is-NLIST.1(SORT-NLIST.2(Y)) = false .
 eq is-NLIST.1(SORT-NLIST.3(Z)) = false .
 eq is-NLIST.2(SORT-NLIST.1(X)) = false .
 eq is-NLIST.2(SORT-NLIST.2(Y)) = true .
 eq is-NLIST.2(SORT-NLIST.3(Z)) = false .
 eq is-NLIST.3(SORT-NLIST.1(X)) = false .
 eq is-NLIST.3(SORT-NLIST.2(Y)) = false .
 eq is-NLIST.3(SORT-NLIST.3(Z)) = true .
 eq get-NLIST.1(SORT-NLIST.1(X)) = X .
 eq get-NLIST.2(SORT-NLIST.2(Y)) = Y .
 eq get-NLIST.3(SORT-NLIST.3(Z)) = Z .
```