Especificação Formal Orientada por Aspectos

Wagner S. Pires, Roberto S. Bigonha

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerias(UFMG)

{salazar,bigonha}@dcc.ufmg.br

Abstract. *This paper describes, Aspect$\mathcal{M}$, an aspect oriented formal specification language. This language unifies the well known benefits of formal specification, such as rigorous requirement description, from which verification and validation can be carried out and the facilities provide by aspect oriented programming, such as separation of concerns.*

Resumo. *Este artigo descreve, Aspect$\mathcal{M}$, uma linguagem de especificação formal orientada por aspectos. Esta linguagem unifica as vantagens existentes em uma especificação formal, como, a descrição precisa dos requisitos do sistema, a partir da qual podem ser realizadas verificações, e as vantagens da programação orientada por aspecto, como modularização de interesses transversais.*

1. Introdução

Especificações e métodos formais têm sido aplicados com objetivo de dar uma base matemática ao desenvolvimento de sistemas. Esses métodos utilizam modelos matemáticos que fazem uso de sintaxe e semântica bem definidas. Assim, técnicas de verificação formal podem ser aplicadas para garantir a correção de determinadas propriedades do software no processo de desenvolvimento.

Métodos formais podem ser aplicados em qualquer etapa do ciclo de desenvolvimento do software, tanto no processo de especificação, para verificar se o software atende aos seus requisitos, como no processo de implementação, para concluir acerca da correção da implementação em relação à especificação. Desta forma, a solução de um problema pode ser representada por uma especificação formal e esta pode ser validada, ainda que parcialmente por meio de sua execução (simulação), ou ainda ser analisada e verificada formalmente, provando-se propriedades do sistema antes mesmo de sua implementação.

Com o aumento crescente da complexidade dos sistemas constata-se que certos interesses (do inglês, *concerns*) não se encaixam em um único módulo de programa, ou pelo menos em um conjunto de módulos altamente relacionados. Desta forma, segundo Tzilla Elrad et al. [Elrad et al. 2001], qualquer decomposição de sistemas (procedimental ou orientada por objetos) revelará que alguns interesses estão localizados dentro de módulos específicos, enquanto outros estão espalhados e entrelaçados em vários módulos. Esses últimos são chamados de interesse transversais. Pode-se ter, por exemplo, o entrelaçamento de código de negócio com código de apresentação, e o espalhamento de código de acesso a banco de dados em vários módulos.

Em busca de uma solução para o problema da separação e composição de interesses transversais, nos últimos anos houve um investimento na separação avançada de

interesses. A Programação Orientada por Aspectos (AOP, do inglês, *Aspect-Oriented Programming*) propõe mecanismos e abstrações inovadores para melhorar a separação de interesses oferecida pelos paradigmas atuais.

AOP tem por objetivo tornar mais fácil a manutenibilidade e evolução de softwares. Entretanto, segundo Silva e Prado Leite [Silva and Leite 2005a], AOP trata de aspectos ou interesses transversais de baixo nível de abstração, quando os requisitos já foram congelados e muitas decisões de gerência e desenho já foram tomadas.

Com o intuito de possibilitar o tratamento de aspectos em todas as etapas do desenvolvimento, têm sido propostos métodos e linguagens de modelagem considerando estes novos elementos e relacionamentos de desenho de maneira a permitir a separação e composição de interesses transversais no nível de desenho. Como exemplo, pode-se citar [Silva and Leite 2005b, Ramos and Castro 2005]. Estas abordagens têm se estendido também à fase de definição de requisitos, denominada por alguns de *early-aspects* [Rashid et al. 2002b].

Dentro deste contexto, este trabalho tem como objetivo geral unificar as vantagens existentes em uma especificação formal, como, a descrição precisa dos requisitos do sistema, a partir da qual podem ser realizadas verificações, e as vantagens da programação orientada por aspecto, como modularização de interesses transversais. Como principal resultado, obteve-se uma linguagem de especificação formal, a saber, Aspect$\mathcal{M}$, que possui recursos necessários para especificar interesses transversais de forma modular.

Uma das vantagens do uso do modelo ASM para a especificação da semântica de um algoritmo é que esta especificação pode ser executada. Para isso, além de especificar as construções da linguagem Aspect$\mathcal{M}$, foi implementado um compilador para a linguagem. Assim, as construções desta linguagem são compiladas para código C++ e AspectC++. Deste modo, tendo a linguagem C++ como linguagem alvo, torna-se possível uma execução rápida e eficiente.

Este trabalho encontra-se dividido em 7 seções. Na Seção 2, são apresentados os principais conceitos relacionados a separação avançada de interesses. Na Seção 3, descrevem-se as principais características e construções da linguagem Aspect$\mathcal{M}$. Na Seção 4, descreve-se a implementação do compilador que possibilita a execução das especificações em Aspect$\mathcal{M}$. A Seção 5 contém um exemplo de uso da linguagem. Por fim, na Seção 7, apresentam-se as conclusões do artigo.

2. Separação Avançada de Interesses

A separação de interesses é um princípio fundamental da Engenharia de Software que tenta resolver as limitações da cognição humana ao lidar com a complexidade do software [Garcia 2004]. Um interesse é uma parte do problema que queremos tratar como uma unidade conceitual única na solução do software.

O princípio da separação de interesses defende que, para superar a complexidade, deve-se resolver um interesse por vez. Na engenharia de software, esse princípio está relacionado à modularização e à decomposição de sistemas. Os sistemas complexos devem ser decompostos em unidades modulares menores e claramente separadas, cada uma lidando com um único interesse. Um módulo é um artefato de programação que pode ser desenvolvido e compilado separadamente de outras partes que compõem o pro-

grama [Dijkstra 1976, Parnas 2002]. Como principais benefícios ou vantagens dessa decomposição, pode-se destacar:

- Facilidade de manutenção - como o código fica mais bem modularizado e com menos redundâncias, o engenheiro de software não necessita procurar trechos de códigos por todas as classes da aplicação, pois ele está localizado em poucos módulos;
- Facilidade de compreensão do código - da mesma forma que o tópico anterior, a modularidade faz com que o engenheiro de software se concentre em poucos módulos para entender determinada funcionalidade;
- Facilidade de evolução - como o código encontra-se bem modularizado, adicionar novos aspectos que tratam de outros interesses se torna mais fácil.

A separação dos interesses depende muito da adeqüabilidade das abstrações e métodos usados ao longo do processo de desenvolvimento de software. Ela também depende dos mecanismos de composição suportados pelas linguagens utilizadas.

Há várias abordagens recentes à separação avançada de interesses, dentre elas, pode-se citar a Programação Orientada por Aspectos (POA, do inglês *Aspect-Oriented Programming*), a Programação Orientada por Assunto (*Subject-Oriented Programming*), a Programação Adaptativa (*Adaptive Programming*) e a Separação Multidimensional de Interesses (*Multi-Dimensional Separation of Concerns*). Essas abordagens, brevemente descritas por Chavez [Chavez 2004], contribuíram para melhorar a programação orientada por objetos, uma vez que proporcionam a separação de interesses em outras dimensões, além das classes e dos objetos. Cada uma dessas abordagens citadas descreve seu próprio modelo de programação para oferecer suporte à modularização dos interesses transversais.

Com objetivo de melhorar o entendimento e, consequentemente, a qualidade do documento de requisitos, desta maneira antecipando a correção de possíveis erros, alguns trabalhos propõem a utilização da separação avançada de interesses na fase de requisitos [Rashid et al. 2002a], cuja a idéia principal é que a especificação dos interesses são mais bem compreendidas quando os interesses principais e transversais estiverem especificados separadamente. Além dessa separação de interesse, as pesquisas nessa área fornecem algumas técnicas para modelagem e ainda como e onde posteriormente combinar o que foi separado.

Rashid et al. [Rashid et al. 2002a] propuseram um modelo de processo genérico, denominado AORE (*Aspect-Oriented Requirements Engineering*), para separar interesses transversais no nível de requisitos. No modelo AORE, o termo "interesse" é utilizado num sentido mais restrito, correspondendo a um interesse transversal de alto nível de abstração, como, por exemplo, segurança ou auditoria.

Diante das vantagens propostas pela aplicação de técnicas orientadas por aspecto na fase de requisitos, Ramos e Castro [Ramos and Castro 2005] propõem uma metodologia para avaliação da qualidade desta nova maneira de especificar os requisitos. Como principal conclusão, os autores afirmam que "a busca de informações sobre atributos foi facilitada, pois apenas foi necessário identificar no diagrama de casos de uso onde se encontrava o atributo procurado e posteriormente localizar a tabela que continha sua descrição. Diferente de quando o documento de requisitos não é orientado por aspectos,

pois neste caso o avaliador teria de fazer uma busca em todo o documento de requisitos a fim de encontrar trechos que tratam do atributo".

Desta forma, a aplicação de técnicas orientadas por aspecto na especificação de um sistema pode facilitar o seu entendimento e compreensão, uma vez que cada interesse está especificado em apenas um lugar. Além disso, torna-se mais fácil provar determinadas propriedades relacionadas ao sistema, como será ilustrado na Seção 5.

3. A Linguagem Aspect$\mathcal{M}$

Aspect$\mathcal{M}$ é uma linguagem orientada por aspectos e constitui-se de uma nova versão da linguagem de especificação formal Machĭna [Bigonha et al. 2007]. Esta versão acrescenta novas construções à linguagem, permitindo a implementação de interesses transversais de forma modular. Por meio de Aspect$\mathcal{M}$, o usuário pode definir pontos específicos no fluxo de execução das regras de transição, como a chamada de ações ou o acesso a funções, e a partir destes pontos, pode-se definir regras de transição. Além disso, pode-se também alterar a álgebra dos módulos de forma não intrusiva e transparente para os módulos afetados.

Baseado em Eos, Aspect$\mathcal{M}$ não possui uma nova unidade de modularização para implementação dos interesses transversais. Utilizando a idéia de *Classpects* proposta por Rajan e Sullivan, em [Rajan and Sullivan 2005], as novas construções desta versão são definidas no próprio módulo, em uma seção denominada **aspect**. Assim, os módulos são compilados por um compilador especial, que une os elementos e gera um código em C++ e AspectC++.

Análogo a AspectJ, esta versão acrescenta a linguagem Machĭna dois tipos de implementação de interesses transversais: transversalidade dinâmica e transversalidade estática. A transversalidade dinâmica é restrita a um conjunto pré-definido de pontos especificados por um modelo de pontos de junção. Esse modelo concentra-se na execução das regras de transição de uma especificação em Machĭna. O modelo de ponto de junção é o principal componente de uma linguagem orientada por aspectos e representa um arcabouço conceitual para descrever os tipos de pontos de junção de interesse e as restrições associadas a seu uso. Ele é altamente dependente da linguagem de componentes adotada, neste caso, a linguagem Machĭna.

A transversalidade dinâmica, em Aspect$\mathcal{M}$, permite definir regras adicionais em pontos específicos no fluxo de execução das regras de transição. Por exemplo, pode-se especificar que uma certa regra seja executada paralelamente à execução de outras regras ou ações de um conjunto de módulos. Para isso, basta especificar separadamente qual a regra deverá ser utilizada e os pontos de junção onde a mesma será inserida.

A transversalidade estática afeta as estruturas dos módulos de uma especificação em Machĭna. Tem-se, por exemplo, a introdução de algumas funções ou tipos em um conjunto de módulos previamente especificados.

Desta forma, Aspect$\mathcal{M}$ permite a modularização adequada de uma grande variedade de interesses transversais como verificação e tratamento de erros, sincronização, distribuição, monitoramento e auditoria, etc. A Listagem 1 ilustra a estrutura de um módulo em Aspect$\mathcal{M}$. Esse módulo possui todas as construções de um módulo em Machĭna, além de um conjunto de construções usuais de uma linguagem orientada por

aspectos, como AspectJ. Assim, um módulo pode entrecortar outro módulo naturalmente.

```
1  module module-name
2      import elementos importados
3      include elementos incluídos
4      algebra:
5          elementos declarados(funções e tipos)
6      abstractions:
7          declaração de abstrações(ações)
8      initial state:
9          inicializações de funções dinâmicas
10     transition:
11         regras de transição de estado
12     aspect:
13         declaração de intertipos, conjuntos de junção e adendos
14     invariant:
15         invariante de execução
16 end module-name
```

Listagem 1. Estrutura de um aspecto em Aspect$\mathcal{M}$

A parte **import** coloca no escopo do módulo a interface dos agentes com os quais o agente do módulo deseja se comunicar. A interface de um agente contém as assinaturas de abstrações de regras que o módulo do agente torna público para uso de outros agentes. A parte **include** define os módulos secundários e seus elementos que devem ser incorporados ao módulo. Esta cláusula tem importante papel na formação do vocabulário dos agentes. A seção **algebra** define os elementos da álgebra subjacente ao modelo, contendo os *sorts* ou tipos e as funções do módulo. A seção **abstrations** define abstrações de regras de transição, que podem ser usadas localmente ou exportadas. Já a seção **initial state** serve para inicializar essas funções dinâmicas.

A seção **transition** define a regra de transição de estado do agente, a qual é executada repetidamente quando o agente é disparado. Esta seção representa o corpo do programa dos agentes associados ao módulo. A seção **aspect** permite especificar regras que entrecortam outros módulos, possibilitando assim a implementação de interesses transversais de forma modular. Por fim, a seção **invariant** define a condição envolvendo os elementos de um módulo que deve ser invariante durante sua execução. Entre duas iterações da regra de transição do módulo, o invariante é verificado pelo sistema de execução Machĭna. Caso seja violado, uma mensagem de erro é emitida e a execução, interrompida.

O restante desta seção apresenta as construções da linguagem Aspect$\mathcal{M}$referentes a especificação de requisitos transversais, sendo que uma descrição completa pode ser encontrada em [Pires 2007].

A capacidade de uma linguagem de aspectos suportar a modularização de interesses transversais é determinada pelo MPJ. Um MPJ consiste, basicamente, em três elementos: (i) pontos de junção, (ii) conjunto de junção, meios que identifiquem um ponto de junção e (iii) adendos, meios que especifiquem semântica em um ponto de junção. O modelo de pontos de junção (MPJ) de Aspect$\mathcal{M}$ é léxico e que identifica entidades e configurações da especificação em Machĭna por seus nomes. Entidades, como

módulos e ações, são artefatos estáticos de uma especificação. Configurações são artefatos dinâmicos da simulação desta especificação, como a chamada de ações.

Aspect$\mathcal{M}$ adota um modelo no qual um ponto de junção é um ponto bem definido no fluxo de execução das regras de transição de uma especificação em Machĭna. Os pontos de junção de Aspect$\mathcal{M}$ incluem a chamada ou execução de ações, inicialização de funções, acesso a funções, escopo de execução, entre outros.

Um conjunto de junção captura ou identifica pontos de junção no fluxo de execução das regras de transição. Uma vez capturado um ponto de junção, pode-se especificar adendos (regras adicionais) envolvendo este ponto, por exemplo, executar uma outra regra ao mesmo tempo. Além disso, alguns conjuntos de junção expõem dados do contexto de execução do ponto de junção capturado, como, seu tipo ou assinatura. Assim, os adendos podem utilizar este contexto para implementar novas funcionalidades.

Aspect$\mathcal{M}$ provê uma coleção de designadores para categorizar um conjunto de pontos de junção. Um designador de conjunto de junção é uma fórmula que especifica um conjunto de pontos de junção ao qual um adendo pode ser aplicado. Na Tabela 1 estão relacionados os principais designadores implementados em Aspect$\mathcal{M}$, com suas respectivas características.

Designador	**Características**
execution(action-pattern)	Execução de ações
get(functio-pattern)	Referência a funções
initialization(module-pattern)	Inicialização de estado
transition(module-pattern)	Transição entre os estados
target(identificador ou module-pattern)	O agente receptor(alvo) é do tipo do módulo especificado ou associado ao identificador
args(identificador ou tipo)	Os argumentos são instância do tipo informado ou são associados ao identificador
within(module-pattern)	O código em execução está definido no(s) módulo(s) especificado(s)
withincode(action-pattern)	O código em execução está definido na(s) ação(es) especificada(s)

Tabela 1. Designadores em Aspect$\mathcal{M}$

O designador de execução de ações captura a execução das ações. Para definir este tipo de designador, utiliza-se o operador **execution**, parametrizado com a assinatura da ação. O designador referente ao acesso as funções captura a leitura de qualquer tipo de função de um módulo. Para definir este tipo de designador, utiliza-se o operador **get**, parametrizado com a assinatura da função. Já designador referente a inicialização de funções captura a execução do código especificado no bloco **initial state** dentro das definições dos módulos. A identificação é realizada por meio do operador **initialization**, parametrizado pelo nome do módulo.

O designador referente a transição entre os estados de um módulo, **transition**, descreve o ponto de execução antes, durante ou depois da atualização de todas as funções de um estado, ou seja, captura a transição de um estado S_n para um estado S_{n+1}. A

definição de antes ou depois fica especificada no adendo.

Contextos podem ser utilizados na definição de conjuntos de junção. Em Aspect$\mathcal{M}$, pode-se utilizar designadores para definir conjuntos de junção nos seguintes elementos de contexto: tipo do agente receptor de uma chamada a uma função e argumentos de funções ou ações.

O designador **target**, parametrizado com a assinatura de módulo, é utilizado para definir pontos de junção que ocorrem durante a execução de uma ação cujo agente receptor é do tipo do módulo passado como parâmetro, enquanto que o designador **args** é utilizado para definir pontos de junção que utilizam os valores dos argumentos de uma função ou chamada de ação. Além disso, estes designadores podem ser utilizados para associar o parâmetro formal ao parâmetro real. Como realizado nos exemplo da Seção 5.

Por fim, a estrutura do módulo pode ser utilizada na definição dos conjuntos de junção. Em Aspect$\mathcal{M}$, pode-se utilizar designadores para especificar pontos de junção que ocorram dentro de módulos ou dentro de ações. O designador **within** é utilizado para definir pontos de junção que ocorrem no escopo do módulo cujo nome é dado como parâmetro e o designador **withincode** permite capturar pontos de junção no escopo de ações de módulos, passando no lugar de um nome de módulo a assinatura de uma ação.

O adendo compreende as ações que devem ser executadas em cada ponto de junção. Em Aspect$\mathcal{M}$ existem quatro tipos de adendo: (i) anterior, (ii) posterior, (iii) de adição e (iv) de contorno (do inglês *before, after, addition* e *around*).

O adendo anterior é executado antes do ponto de junção, enquanto que o adendo posterior é executado depois do ponto de junção. Diferente de linguagens como AspectJ, esses adendos, anterior e posterior, só podem estar associados a um designador, a saber **transition**, que permite captura a transição de um estado S_n para um estado S_{n+1}. Por exemplo, pode-se especificar pré e pós condições para executar uma determinada regra de transição. Já o adendo de adição, criado para o contexto de ASM, executa paralelamente ao momento em que o ponto de junção é alcançado. Por fim, o adendo de contorno executa quando o ponto de junção é alcançado e tem controle explícito se o próprio ponto afetado deve ou não ser executado.

Essas declarações associam uma regra de transição a um conjunto de junção, indicando como uma regra entrecortará uma especificação em Machĭna. Basicamente, quando o ponto de junção é capturado pelo conjunto de junção, a regra de transição é inserida para que posteriormente seja executada.

A definição de um adendo pode ser dividida em três partes: (i) a declaração do adendo; (ii) a especificação do conjunto de junção; (iii) e a declaração da regra de transição. A Listagem 2 apresenta a declaração de um adendo de contorno. A linha 2 representa a declaração do adendo e especifica quando o adendo vai executar a ação nos pontos capturados. Já a linha 3, na parte entre os dois pontos e a palavra **do**, especifica os conjuntos de junção responsáveis pela coleta dos pontos de junção, ou seja, define os pontos onde a regra de transição, especificada pelo adendo, vai ser inserida para que posteriormente seja executada. Por fim, na linha 4, é especifica a regra de transição que será inserida nos pontos coletados pelos conjuntos de junção especificados.

A semântica deste adendo é captura a execução das regras de transição dos

módulos terminados com `Unit`. Além disso, captura o agente responsável pela execução e o associa a variável `agente`.

```
1  aspect:
2    around(agente:Agent):
3      transition(*Unit) and target(agente)  do
4          ...  Regra de Transição de Estado ...
5    end
```

Listagem 2. Exemplo da declaração de um adendo

A transversalidade dinâmica, obtida a partir do modelo de ponto de junção, permite modificar o comportamento da execução do programa, ao passo que a transversalidade estática permite redefinir a estrutura estática dos tipos.

Em Aspect$\mathcal{M}$ é possível implementar a introdução de novos elementos à álgebra de um módulo, como, tipos e funções. Além disso, é possível adicionar novas ações, ou seja, abstrações de regras de transição. Este mecanismo de introdução é denominado declaração de intertipos (do inglês, *inter-type declaration*). Desta forma, diferente dos adendos, que operam primariamente de forma dinâmica, as declarações de intertipos operam de forma estática, em tempo de compilação.

A Listagem 3 ilustra alguns exemplos. Basicamente, basta definir o tipo, função ou ação precedido do nome do módulo que receberá o elemento. Neste exemplo, entre as linhas 2 e 5, define-se que os tipos `Undefined, Id, Value` e `Memory` serão introduzidos ao módulo `Globals`. Observe que na definição de `Memory`, considerou-se que os tipos `Undefined, Id` e `Value` já pertencem ao módulo `Globals`.

```
1  aspect:
2    type Globals.Undefined = public enum {UNDEFINED};
3    type Globals.Id = String;
4    type Globals.Value = Bool | Int;
5    public type Globals.Memory = Id -> Value | Undefined;
```

Listagem 3. Exemplo de introdução de tipos

4. Implementação

Considerando-se que o compilador implementado produz um programa em C++ e AspectC++ para refletir a semântica de uma especificação em Aspect$\mathcal{M}$, é preciso deixar explícito a abordagem utilizada para o mapeamento dos elementos da linguagem Aspect$\mathcal{M}$ para OOP e AOP. A Figura 1 ilustra os possíveis elementos gerados. Cabe ressaltar que a parte relacionada a AspectC++, arquivo de extensão .ah, está destacada apenas para indicar que sua geração é condicional.

Todo processo de tradução é descrito em [Pires 2007], sendo que os principais elementos a se considerar são os módulos e os agentes. Assim, no processo de tradução para C++, a compilação de um módulo dá origem a uma classe. Os componentes do módulo, ou seja, as funções, tipos e abstrações de regras são traduzidos nos membros desta classe.

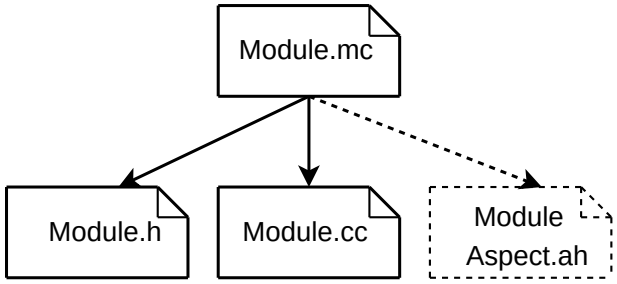

Figura 1. Geração de código C++ e AspectC++ a paritr de um arquivo Machĭna

Toda classe que representa um módulo deve herdar de uma classe denominada `Module`. Esta classe define o método `executeTransitionRule` como um método puramente virtual. Desta forma, os agentes podem ser uniformemente tratados como objetos da classe `Module`, e cada agente executará sua regra de transição corretamente.

Um agente, por sua vez, é um objeto do módulo do seu tipo. O disparo de um agente é a execução de seu método `start` por meio de uma *thread*. A compilação de um módulo de definição Machĭna, o qual cria e dispara os principais agentes, dá origem a uma classe que cria as instâncias dos agentes especificados de acordo com seu tipo e então dispara estes agentes.

Com relação as construções orientadas por aspecto, declaradas na seção **aspect**, a parte referente a transversalidade dinâmica, a saber, conjuntos de junção e adendos, são convertidas pra AspectC++. Os designadores de conjuntos de junção de Aspect$\mathcal{M}$ são compilados para designadores em AspectC++, realizando algumas adaptações. Por exemplo, **transition** `(moduleName)` de Aspect$\mathcal{M}$ é compilado para **execution** `(moduleName::executeTransitionRule)` de AspectC++, ou seja, captura o método que executa a regra de transição do módulo. Já com os adendos, primeiramente, é realizada uma inversão de controle do aspecto para o módulo. Convém mencionar que todo adendo declarado na seção **aspect** possui um método na classe correspondente ao módulo de sua declaração. Então, basicamente, o código do aspecto criada uma instância do módulo que possui o código do adendo e, em seguida, é chamado o método que corresponde ao adendo.

Basicamente, na compilação de uma especificação Machĭna, o usuário apenas indica o nome do arquivo que contém o módulo de definição Machĭna. Então, a partir deste arquivo são geradas todas as classes e aspectos necessários. Durante a etapa de compilação, como os aspectos possuem a propriedade de transparência (*obliviousness*), ou seja, os módulos não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos, não se sabe antecipadamente todos os módulos que serão envolvidos no processo de combinação. Desta forma, similar ao combinador de AspectC++, todos os módulos existentes no diretório corrente são lidos e a sua respectiva AST é armazenada na memória.

Assim, para implementar a transversalidade estática, percorre-se cada AST identificando módulos que possuem uma seção aspect com declarações de inter-tipos, ou seja, introdução de funções, tipos ou ações. Uma vez encontrado, os nodos referentes à declaração de inter-tipos contém informações necessários para criar um novo nodo,

semelhante a uma declaração normal, e em qual AST este novo nodo deve ser inserido. Desta forma, um novo nodo é criado e inserido na AST determinada pela sua declaração.

5. Modularização de Especificações

O problema da caldeira a vapor abordado nesta seção foi descrito por Jean-Raymond Abrial [Abrial 1994], como sugestão de trabalho para os participantes do *Dagstul Meeting Methods for Semantics and Specification*, realizado em Junho de 1995.

Segundo Abrial, a caldeira deve trabalhar com o nível de água dentro de limites considerados normais de funcionamento e nunca ultrapassar os níveis de risco definidos. Assim, é importante que o sistema trabalhe corretamente porque a quantidade de água presente, quando a caldeira estiver funcionando, deve sempre estar entre esses limites; caso contrário, a caldeira ficará danificada.

Inicialmente, Abrial descreve o ambiente físico no qual o programa deve interagir. Cada unidade física é apresentada, assim como suas restrições e condições de funcionamento. Além disto, são definidas algumas constantes e variáveis que devem ser verificadas e controladas ao longo do funcionamento do sistema.

Logo após, é apresentado como o programa deve se comunicar com as unidades físicas e quais são as fases de comunicação. Segundo Abrial, o programa comunica-se com os dispositivos físicos através de mensagens, sendo que o tempo para transmissão pode ser negligenciado. O programa segue um ciclo que, a princípio, não termina. Este ciclo acontece a cada cinco segundos e consiste nas seguintes ações: (a) receber as mensagens dos dispositivos físicos, (b) analisar a informação recebida e (c) transmitir uma mensagem aos dispositivos. Por fim, são detalhados os modos de operação inicialização, normal, recuperação, degradado e emergência, que o programa pode assumir e, finalmente, é descrito a detecção de erros nos equipamentos.

Em [Abrial et al. 1995], foram apresentadas 23 soluções abordando diferentes métodos de especificação. Uma delas foi proposta por Börger[Beierle et al. 1996]. Um fato importante na solução de Börger é a descrição clara de alguns interesses transversais, como :

> [*Note that we assume the condition* ¬EmergencyStop *to be global precondition extending the guards of all rules - except for global rules*]
>
> [*Although this is not explicitly stated in the informal description, one can reasonably argue that the operations of adjusting the water level to a default value or of maintaining its value within an admissible range are essentially the same for any* mode $\in \{normal, degraded, rescue\}$]

O tratamento de erro na caldeira a vapor representa um interesse transversal, como também o controle do nível de água e a sincronização da comunicação. Quando se identifica um erro grave, através da função `EmergencyStop`, o sistema deve ser imediatamente transferido para o modo de emergência, e aguardar que o ambiente externo tome as providências necessárias. Desta maneira, antes de executar qualquer regra, seja das unidades físicas ou dos modos de operação, deve ser verificado que não existe um erro grave.

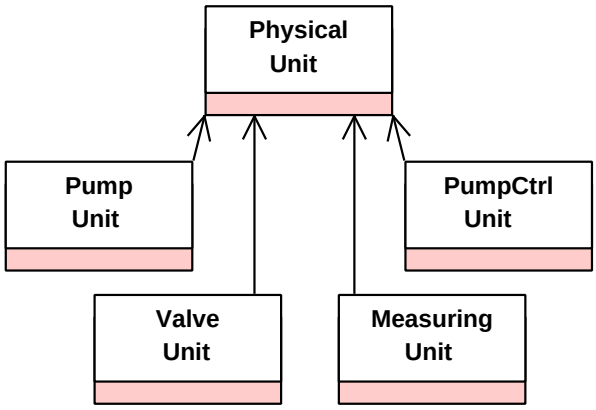

Figura 2. Módulos para as unidades físicas especificados em Machĭna

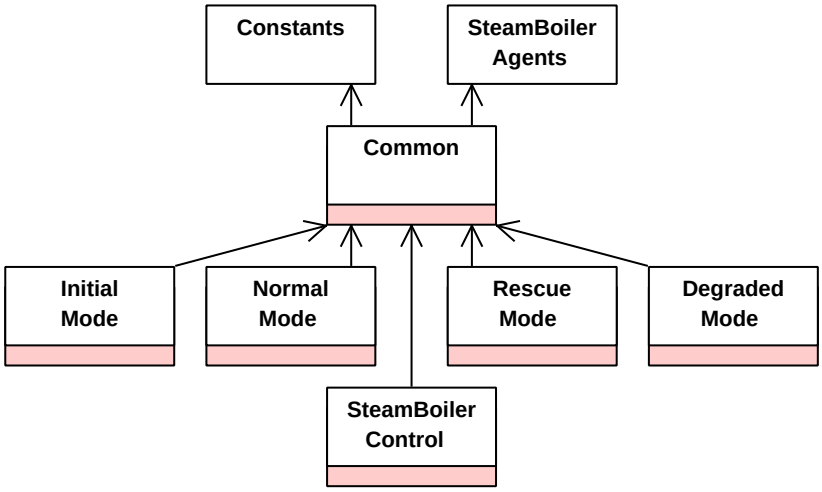

Figura 3. Módulos para o sistema de controle especificados em Machĭna

Assim, com o objetivo de validação, as linguagens Machĭna e Aspect$\mathcal{M}$ foram usadas na especificação da caldeira a vapor. Utilizando a linguagem Machĭna, a especificação é composta por 13 módulos. As Figuras 2 e 3 ilustram a estrutura dos módulos desta especificação. Nestas figuras os módulos são representados por retângulos e as operações de *include* correspondem às setas. Cabe ressaltar que as regras relacionadas aos interesses transversais, representadas em cinza, ficam espalhadas e entrelaçadas com, praticamente, toda especificação.

Os problemas de espalhamento e entrelaçamento na especificação trazem sérias dificuldades para o entendimento de cada interesse, para manipulação da especificação, para a reutilização de partes da especificação e para realizar modificações nos próprios interesses. Estes problemas trazem dificuldades ainda maiores se consideramos o processo de desenvolvimento de software inteiro, pois este depende da boa elaboração dos interesses para o efetivo desenvolvimento e aceitação do produto [Sommerville 2000].

Assim, o principal problema destacado nesta trabalho é o espalhamento e entrelaçamento de interesses transversais na especificação formal de sistemas. Consi-

deramos que eles são problemas naturais, decorrentes da complexidade dos sistemas e, portanto, uma linguagem de especificação formal deve possuir recursos apropriados para especificar estes interesses transversais. Desta forma, como a linguagem Machĭna, na versão atual, não possui tais recursos, pode-se observar os seguintes problemas em uma especificação:

- o entendimento do código da especificação fica mais difícil, uma vez que temos códigos com propósitos distintos misturados;
- como o código da especificação referente a um interesse transversal não é apropriadamente encapsulado, ele encontra-se espalhado por todo o sistema, desta forma, uma alteração neste código provocará modificações em vários lugares, tornando com isso difícil a manutenção/evolução do sistema;
- visto que o código do interesse principal está misturado com o código do interesse transversal, a reusabilidade do componente fica prejudicada caso desejarmos fazer uso da sua funcionalidade básica num contexto diferente;
- para provar determinadas propriedades, como as regras das unidades físicas não serão executadas na presença de algum erro grave, é necessário analisar toda especificação, umas vez que estas regras estão espalhadas e entrelaçadas com as demais.

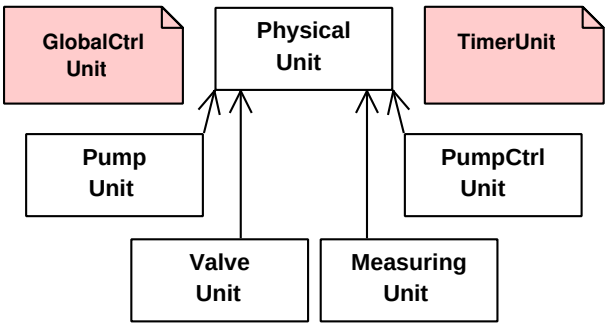

Figura 4. Módulos para as unidades físicas especificados em Aspect$\mathcal{M}$

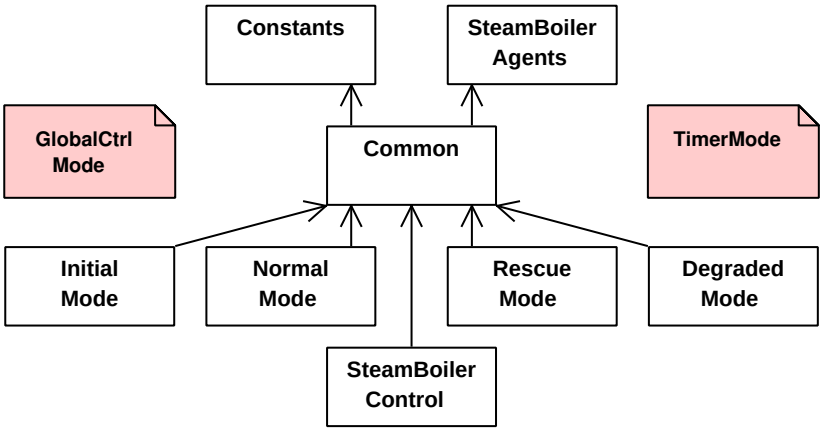

Figura 5. Módulos para o sistema de controle especificados em Aspect$\mathcal{M}$

Agora, utilizando a linguagem Aspect$\mathcal{M}$, é possível modularizar as regras relacionadas aos interesses transversais, que antes se encontravam espalhadas e entrelaçadas com toda a especificação. Assim, estas regras foram agrupadas em quatro módulos. As Figuras 4 e 5 ilustram a estrutura dos módulos desta nova especificação. Convém mencionar que os módulos destacados em cinza especificam os interesses transversais e as operações de *include* destes módulos foram omitidas.

Como exemplo de especificação de um módulo em Aspect$\mathcal{M}$, a Listagem 4 apresenta a codificação do módulo `PumpUnit`. Caso fosse utilizada a linguagem Machĭna, ainda seria necessário especificar as regras relacionadas ao tratamento de erro, ajuste do nível de água e demais interesses transversais, dificultando assim a sua compreensão e entendimento.

```
1  module PumpUnit
2    include PhysicalUnit;
3    algebra:
4      dynamic turnOn: Bool := false;
5    abstractions:
6      public action setTurnOn (in turnOnPump:Bool) is
7        turnOn := turnOnPump
8      end
9    initial state:
10     steamBoiler := SteamBoilerControl.theAgent;
11   transition:
12     if programReady then
13       steamBoiler.recievePumpInfo(self, ready, failure, turnOn);
14     end;
15 end PumpUnit
```

Listagem 4. Módulo para representar as bombas em Aspect$\mathcal{M}$

Utilizando as construções de Aspect$\mathcal{M}$ pode-se incluir o tratamento de erro, como outros interesses transversais, sem causa o entrelaçamento de regras na especificação. A Listagem 5 apresenta um tratamento de erro para os dispositivos físicos. Antes de executar qualquer regra é verificado a condição $\neg$`EmergencyStop`. Cabe ressaltar que a função `EmergencyStop` é definida por outras 3 funções, a saber `externalStop`, `reachingLimitLevel` e `transmissionFailure`. Como objetivo é apenas demonstrar como este requisito poder ser adicionado as regras da especificação sem causar um entrelaçamento, optou-se por declarar essas funções como externas. Um outro requisito especificado é o ajuste do nível de água, apresentado na Listagem 6. Antes de executar qualquer regra nos modos normal, recuperação e degradado é verificado o nível de água, e caso esteja foram dos limites permitidos, é realizado um ajuste.

Basicamente, as regras das listagens capturam o ponto em que o agente irá executar sua regra de transição, realiza a conversão (*casting*) para o tipo do agente corrente e verifica o valor de algumas funções.

A linguagem Aspect$\mathcal{M}$ permite especificar cada interesse separadamente com o mínimo de acoplamento. O resultado é uma especificação modularizada mesmo na presença de interesses transversais. Em alguns casos, é possível reduzir a redundância das regras, uma vez que cada interesse encontra-se separado. Além disso, a especificação

```
1  module GlobalCtrlUnit
2    include PhysicalUnit,PumpUnit,PumpCtrlUnit,ValveUnit,MeasuringUnit;
3    aspect:
4      external PhysicalUnit.externalStop: Bool;
5      external PhysicalUnit.reachingLimitLevel: Bool;
6      external PhysicalUnit.transmissionFailure: Bool;
7      derived function PhysicalUnit.emergencyStop : Bool :=
8          reachingLimitLevel or externalStop or transmissionFailure;
9
10     around(agente:Agent): transition(*Unit) and target(agente) do
11       with agente
12         as p:PumpUnit      => if not p.emergencyStop then proceed end;
13         as pc:PumpCtrlUnit => if not pc.emergencyStop then proceed end;
14         as v:ValveUnit     => if not v.emergencyStop then proceed end;
15         as m:MeasuringUnit => if not m.emergencyStop then proceed end;
16       end;
17     end
18 end
```

Listagem 5. Verificação de erros graves nas unidades físicas

bem modularizada facilita a compreensão e o entendimento.

Um outro ponto importante é a prova de propriedades. Por exemplo, para provar que as regras das unidades físicas não serão executadas na presença de algum erro grave, representado pelo função `EmergencyStop`, basta analisar as 18 linhas da Listagem 5, ao invés de quase toda a especificação.

6. Conclusão

As tecnologias de programação pós-objeto propõem mecanismos e abstrações inovadores para melhorar a separação de interesses oferecida pelos paradigmas atuais. A POA se concentra em mecanismos para a simplificação da implementação dos interesses transversais. Enquanto a tendência na POO, por exemplo, é encontrar pontos comuns entre as classes e empurrá-las para cima na árvore de herança (generalização), a POA tenta encontrar interesses espalhados, agrupá-los em abstrações de primeira classe, chamados de aspectos, e lança-los horizontalmente para fora da estrutura dos objetos. Além disso, a POA também oferece novos mecanismos que permitem a composição transparente e flexível destes aspectos.

Este artigo apresentou uma linguagem de especificação formal orientada por aspectos, a saber Aspect$\mathcal{M}$. Esta linguagem unifica as vantagens de uma especificação formal e as vantagens da programação orientada por aspectos. Como relação a linguagens com objetivos semelhantes, Aspect$\mathcal{M}$ destaca-se pela: (i) diminuição das responsabilidades dos módulos, (ii) modularização e (iii) reusabilidade. Com isso, obtém-se uma especificação mais coesa e clara. Além de auxiliar na prova de algumas propriedades, conforme mencionado na Seção 5.

Como trabalhos futuros, pretende-se implementar uma ferramenta visual para a linguagem Aspect$\mathcal{M}$. Os ambientes de programação modernos oferecem diversas facilidades para o desenvolvedor. Cabe citar, a título de exemplo, o *auto-complete* de comandos e expressões, a marcação de pontos de parada para depuração, editores para componentes

```
module GlobalCtrlMode
  include NormalMode, RescueMode, DegradedMode;
  aspect:
    before(agente:Agent): transition(*Mode) and target(agente) do
      with agente
        as n:NormalMode   =>
          if n.waterLevelAdjusted then n.retainWaterLevel
          else n.adjustWaterLevel end;
        as r:RescueMode   =>
          if r.waterLevelAdjusted then r.retainWaterLevel
          else r.adjustWaterLevel end;
        as d:DegradedMode =>
          if d.waterLevelAdjusted then d.retainWaterLevel
          else d.adjustWaterLevel end;
      end
    end
end
```

Listagem 6. Ajuste do nível de água para o sistema de controle

visuais, destaque de sintaxe, verificação de erros de sintaxe em tempo real etc. O desenvolvimento de ferramentas visuais para a linguagem Aspect$\mathcal{M}$ seria de grande valia para a utilização da linguagem de forma mais produtiva.

Além disso, para verificar um sistema especificado na linguagem Machĭna, Stefani [Stefani 2007] propõem a conversão de Machĭna em NuSMV [NuSMV 2007]. Assim, um trabalho futuro seria o estudo da conversão das construções de Aspect$\mathcal{M}$ em NuSMV ou em outro método de verificação, com o objetivo de verificar se as propriedades adicionadas pelos aspectos estão respeitando o modelo, ou seja, é preciso garantir que a implementação esteja de acordo com a especificação do problema. Por fim, pretende-se realizar uma validação exaustivamente do compilador implementado e do uso de Aspect$\mathcal{M}$ na especificação de sistemas de grande porte e complexidade.

Referências

[Abrial 1994] Abrial, J.-R. (1994). A steam-boiler control specification problem.

[Abrial et al. 1995] Abrial, J.-R., Börger, E., and Langmaack, H. (1995). The stream boiler case study: Competition of formal program specification and development methods. In *Formal Methods for Industrial Applications*, pages 1–12.

[Beierle et al. 1996] Beierle, C., Börger, E., Durdanovic, I., Glässer, U., and Riccobene, E. (1996). Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In Abrial, J.-R., Börger, E., and Langmaack, H., editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer.

[Bigonha et al. 2007] Bigonha, R., Tirelo, F., Iorio, V., Bigonha, M., and Lobato, M. (2007). A linguagem de especificação formal machĭna 2.0. Technical report, Departamento de Ciência da Computação - UFMG.

[Chavez 2004] Chavez, C. V. F. (2004). *Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos*. Tese de doutorado, PUC-Rio, Departamento de Informática.

[Dijkstra 1976] Dijkstra, E. W. (1976). A discipline of programming.

[Elrad et al. 2001] Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32.

[Garcia 2004] Garcia, A. F. (2004). *Objetos e Agentes: Uma Abordagem Orientada a Aspectos*. Tese de doutorado, PUC-Rio, Departamento de Informática.

[NuSMV 2007] NuSMV (2007). http://nusmv.irst.itc.it, último acesso: 24 de abril de 2007.

[Parnas 2002] Parnas, D. L. (2002). On the criteria to be used in decomposing systems into modules. pages 411–427.

[Pires 2007] Pires, W. S. (2007). Uma linguagem de especificação formal orientada por aspectos. Dissertação de mestrado, UFMG.

[Rajan and Sullivan 2005] Rajan, H. and Sullivan, K. J. (2005). Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York. ACM Press.

[Ramos and Castro 2005] Ramos, R. A. and Castro, J. B. (2005). Avaliação de uma metodologia de medição da qualidade em um documento de requisitos orientado a aspectos. In *WER*, pages 161–172.

[Rashid et al. 2002a] Rashid, A., Sawyer, P., Moreira, A., and Araújo, J. (2002a). Early aspects: A model for aspect-oriented requirements engineering. In *Joint Int'l Conf. Requirements Engineering*, pages 199–202. IEEE.

[Rashid et al. 2002b] Rashid, A., Sawyer, P., Moreira, A. M. D., and Araújo, J. (2002b). Early aspects: A model for aspect-oriented requirements engineerin. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 199–202, Washington, DC, USA. IEEE Computer Society.

[Silva and Leite 2005a] Silva, L. F. and Leite, J. C. S. P. (2005a). Integração de características transversais durante a modelagem de requisitos. In *SBES '05: Simpósio Brasileiro de Engenharia de Software*.

[Silva and Leite 2005b] Silva, L. F. and Leite, J. C. S. P. (2005b). Uma linguagem de modelagem de requisitos orientada a aspectos. In *WER*, pages 13–25.

[Sommerville 2000] Sommerville, I. (2000). *Software Engineering (6th Edition)*. Addison Wesley.

[Stefani 2007] Stefani, I. G. A. (2007). Método de refinamento machina. Dissertação de mestrado, UFMG.