Um Estudo de Caso Real em Refinamento de Especificações Formais Orientadas a Objetos

Virgínia A. O. Cordeiro, Augusto Sampaio, Silvio L. Meira

Universidade Federal de Pernambuco
Departamento de Informática
50.732-970, Recife - PE - Brasil
{vaoc,acas,srlm}@di.ufpe.br

Sumário

Este trabalho apresenta um método de refinamento para especificações em MooZ *através de sua aplicação a um estudo de caso.* MooZ *é uma linguagem de especificação que é uma extensão orientada a objetos de* Z. *O código final obtido como produto do refinamento utiliza* Eiffel, *uma linguagem de programação orientada a objetos.*

Abstract

This work presents a refinement method for MooZ *specifications by applying it to a case study.* MooZ *is a specification language that is an object oriented extension of* Z. *The final code obtained as the product of the refinement process uses* Eiffel, *an object oriented programming language.*

Keywords: Engenharia de Software, Métodos Formais, Desenvolvimento de Sistemas, Refinamento, Cálculo de Refinamento

1 Introdução

Em [CSM94a], apresentamos um método para refinar especificações em MooZ para programas em Eiffel, o qual é abordado em mais detalhes em [CSM94b]. MooZ [MC90] é uma linguagem de especificação formal baseada em modelos que estende Z [Spi89] através do paradigma de orientação a objetos. Eiffel [Mey92] é uma linguagem de programação orientada a objetos que incorpora características para dar suporte a um desenvolvimento mais rigoroso de sistemas. Neste trabalho, a ênfase é na aplicação do método de refinamento a um estudo de caso mais detalhado. O método de refinamento e as notações específicas das duas linguagens serão introduzidos no decorrer da apresentação do estudo de caso.

O estudo de caso consiste de parte de um sistema de manutenção industrial que está sendo desenvolvido pela In Forma Software Ltda., através do programa *Softex 2000*, núcleo Recife. Aspectos como modularidade da especificação e como isto se reflete no refinamento (não abordados em [CSM94b, CSM94a]) são considerados aqui. Na próxima seção descrevemos brevemente o sistema de manutenção industrial e sua especificação formal inicial, aproveitando para introduzir a notação de MooZ. Em seguida, na Seção 3, apresentamos a aplicação do método de refinamento à especificação, discutindo cada um dos passos utilizados até obtermos o código final em Eiffel. Finalmente, na última seção, discutimos algumas das conclusões decorrentes deste trabalho. No apêndice, apresentamos algumas das regras de refinamento utilizadas no desenvolvimento do estudo de caso.

2 O Sistema de Manutenção Industrial

Por *manutenção industrial* entendemos o conjunto de procedimentos, métodos e técnicas utilizados em uma planta industrial para reparar falhas e defeitos e implementar melhorias na infra-estrutura material que suporta a produção. O órgão da indústria que realiza esta atividade é chamado de Setor de Manutenção Industrial, ou simplesmente, Setor de Manutenção.

Diariamente chegam ao setor de manutenção diversas **Requisições de Serviços**, que chamaremos de **RS**, vindas de diversos setores da indústria, que chamaremos de **órgãos solicitantes**. Estas requisições são solicitações que pedem que algum serviço seja executado, como o conserto de um equipamento, ou instalação de um aparelho, entre outros, sem, no entanto, especificar como este serviço deve ser realizado. Ao chegar ao setor de Manutenção Industrial, técnicos em planejamento tratam de transformar estas **RS** em **Ordens de Trabalho**, que abreviaremos como **OT**, detalhando como o serviço será realizado. Estes técnicos dividem o serviço em **tarefas** que serão alocadas para especialistas dos diversos subsetores da manutenção, como eletricistas, mecânicos, carpinteiros, etc. Os subsetores da manutenção responsáveis pela execução das tarefas são chamados de **órgãos executores**. Assim, uma **OT** contém as seguintes informações, entre outras: órgão que solicitou o serviço, tarefas que serão executadas, especialidade necessária para a execução de cada tarefa, duração de cada tarefa, *dependências temporais* entre as tarefas (quais tarefas devem ser executadas antes de uma determinada tarefa) e quando pode/deve dar-se o *início* e/ou *término* da execução.

O Sistema de Manutenção, portanto, deve lidar com **OTs** que estabelecem os serviços de manutenção que devem ser executados, e estas **OTs**, por sua vez, contêm informações sobre as tarefas a serem executadas e quais especialistas disponíveis na manutenção devem executá-la. A seguir, apresentamos a especificação inicial do sistema em MooZ.

2.1 A Especificação em MooZ

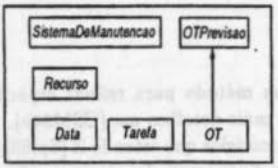

Figura 1: Hierarquia das Classes da Especificação

Na figura 1, apresentamos a hierarquia das classes da especificação. A classe *SistemaDeManutencao* representa o Sistema de Manutenção Industrial. As tarefas são modeladas pela classe *Tarefa*. A classe *Recurso* contém as informações sobre a alocação de recursos para a execução das tarefas. As **OTs** são modeladas pela classe *OT*, porém as informações que dizem respeito ao planejamento das **OTs** estão na classe *OTPrevisao*. Pelo diagrama apresentado na figura 1, é possível perceber que a classe *OT* é subclasse de *OTPrevisao*. A classe *Data* modela as operações de manipulação de datas. Não apresentamos a especificação desta classe, assumindo que ela possui operações para comparar e manipular datas, como $=_{data}$, $>_{data}$, $<_{data}$, $\leq_{data}$, $\geq_{data}$, $+_{data}$, $+_{dias}$, entre outras, além de uma constante *DataNula* definida para representar uma data não inicializada. Além disso, assumimos a existência em Eiffel de uma classe **Date** que implementa a classe *Data*, com todas as suas operações e constantes.

Uma especificação em MooZ é formada por uma ou mais classes. Cada classe possui um estado, introduzido através de um esquema anônimo na cláusula **state**, o qual possui uma parte onde são declarados os seus componentes e outra parte que contém um predicado lógico que estabelece o invariante da classe. Dentro de uma classe, é possível especificar o seu estado inicial, através da cláusula **initialstates**, e suas operações, através da cláusula **operations**. A classe *Recurso*, modelada abaixo, representa os recursos necessários à execução de uma determinada tarefa.

Class *Recurso*

O *givenset CodFunc* modela o código das funções/especialidades disponíveis no Setor de Manutenção. Em MooZ, os *givensets* introduzidos através da cláusula givensets podem ser utilizados para parametrizar uma classe a qual pode ser posteriormente instanciada. O componente *funcao* descreve o código da função/especialidade necessária à execução de uma tarefa e o componente *previsao* descreve a quantidade prevista (um natural não nulo) de especialistas na função que serão necessários para executar uma tarefa. As descrições em MooZ são intercaladas com textos que descrevem informalmente o que está sendo especificado.

givensets *CodFunc*

state

$funcao : CodFunc$
$previsao : \mathbb{N}_1$

EndClass *Recurso*.

Class *Tarefa*

Nesta classe estão representadas as informações previstas para cada tarefa. O componente *tarefaId* representa sua identificação, *desc* representa sua descrição, *ezec* modela o local onde a tarefa será executada, *recs* representa o conjunto de recursos necessários à execução da tarefa, *conc* é a data de conclusão da tarefa, *duracaoPrev* modela a sua duração, *dependencias* representa as tarefas das quais ela é dependente e *inicioPrev* representa o seu início previsto. O *givenset TarId* representa o conjunto dos identificadores de tarefas e *CodEzec* modela o código dos órgãos executores do Setor de Manutenção.

givensets *TarId*, *CodEzec*

state

$tarefaId : TarId$
$desc : String$
$ezec : CodEzec$
$recs : \mathbb{P}\ Recurso$
$conc : Data$
$duracaoPrev : \mathbb{N}$
$dependencias : \mathbb{P}\ Tarefa$
$inicioPrev : Data$

Observe que o componente *dependencias* pode assumir valores que são conjuntos de tarefas. Este tipo de recursão é natural em orientação a objetos e não apresenta problemas desde que haja um valor base para que a recursão não seja infinita; neste caso, o valor base é o conjunto vazio de tarefas.

A inicialização dos componentes desta classe é modelada pelo esquema a seguir, onde todos os componentes têm valores iniciais fornecidos como entrada, exceto os componentes *conc* e *inicioPrev* que inicialmente possuem como valor a constante *DataNula*, indicando que a tarefa ainda não foi concluída nem tem uma data prevista para ser executada. Em MooZ, os componentes de estado modificados são indicados usando-se o símbolo Δ. Componentes utilizados mas não alterados na definição são introduzidos pelo símbolo Ξ. As operações introduzidas através de esquemas são divididas em duas partes: uma contendo declarações e outra contendo um predicado que descreve o comportamento da operação. As variáveis decoradas com "?" indicam as entradas da operação e as decoradas com "!" representam a saída. Componentes de estado sem decoração dizem respeito ao estado anterior da operação e decorados com "′" correspondem ao estado posterior.

initialstates

Init

$\Delta(tarefaId, desc, ezec, recs, conc, duracaoPrev, dependencias, inicioPrev)$

$tarefaId? : TarId$

$desc? : String$

$ezec? : CodExec$

$recs? : \mathbb{P}\ Recurso$

$duracaoPrev? : \mathbb{N}$

$dependencias? : \mathbb{P}\ Tarefa$

$tarefaId' = tarefaId? \land desc' = desc?$

$ezec' = exec? \land recs' = recs?$

$conc' = DataNula \land duracaoPrev' = duracaoPrev?$

$dependencias' = dependencias? \land inicioPrev' = DataNula$

EndClass *Tarefa*.

Class *OTPrevisao*

Esta classe representa as características estabelecidas inicialmente para uma **OT**. Assim, temos como componentes de estado o conjunto de tarefas de uma **OT**, representado por *tarefas* e a data prevista para o início da execução das tarefas, *inicioPrev*. Nesta classe, estão modeladas as relações de dependências entre as tarefas que devem ser executadas em uma **OT**. A operação *IncluiTarefasOK* representa o caso em que a operação de inclusão de tarefas ocorre com sucesso e *TarefasJaExistem* corresponde ao caso de erro. Estas definições são locais à classe e são usadas apenas para modularizar a descrição da operação final de inclusão, *IncluiTarefas*, que trata os casos de sucesso e de erro.

private *IncluiTarefasOK*, *TarefasJaExistem*

Na definição do estado, o primeiro predicado do invariante da classe estabelece que todas as tarefas das quais uma tarefa depende estão na mesma **OT**, ou seja, estão contidas no conjunto *tarefas* de uma **OT**. O último predicado diz que existe pelo menos uma tarefa inicial — que não depende de nenhuma outra tarefa — no conjunto de tarefas.

state

$tarefas : \mathbb{P}\ Tarefa$

$inicioPrev : Data$

$\forall tar : tarefas \bullet tar.dependencias \subset tarefas$

$\exists tar : tarefas \bullet tar.dependencias = \emptyset$

initialstates

O estado inicial dos objetos desta classe é definido pelo esquema abaixo. A operação de inicialização exige, como pré-condição, que o conjunto de tarefas *tarefas*? fornecido como entrada tenha pelo menos uma tarefa inicial e que todas as tarefas do conjunto dependam apenas de tarefas que também estão em *tarefas*?. Com isso, o invariante do estado é satisfeito pela operação de inicialização.

Init

$\Delta(tarefas, inicioPrev)$

$tarefas? : \mathbb{P}\ Tarefa$

$tarefas' = tarefas? \land inicioPrev' = DataNula$

$\forall tar : tarefas? \bullet tar.dependencias \subset tarefas?$

$\exists tar : tarefas? \bullet tar.dependencias = \emptyset$

operations

O tipo enumerado (*free type*) *Mensagem* representa as mensagens que podem ser retornadas pela execução das operações.

$Mensagem ::= ok \mid TarefasJaExistentes$

A operação *IncluiTarefasOK* acrescenta novas tarefas ao conjunto *tarefas*, desde que nenhuma das tarefas novas existam em *tarefas*.

IncluiTarefasOK

$\Delta(tarefas)$
$tarefas? : \mathbb{P}\ Tarefa$
$resposta! : Mensagem$

$tarefas? \cap tarefas = \emptyset$
$tarefas' = tarefas \cup tarefas?$
$resposta! = ok$

A operação abaixo descreve o caso em que a pré-condição da operação acima não é satisfeita:

TarefasJaExistem

$\Xi(tarefas)$
$tarefas? : \mathbb{P}\ Tarefa$
$resposta! : Mensagem$

$tarefas? \cap tarefas \neq \emptyset$
$resposta! = TarefasJaExistentes$

A operação total para inclusão de novas tarefas é definida como:

$IncluiTarefas \mathrel{\widehat=} IncluiTarefasOK \lor TarefasJaExistem$

onde a disjunção de dois esquemas S_1 e S_2 retorna um novo esquema cuja parte declarativa inclui as variáveis introduzidas em S_1 e S_2 e o predicado é a disjunção dos predicados de S_1 e S_2.

EndClass *OTPrevisao*.

Class *OT*

Uma classe pode herdar de várias outras classes, seguindo o conceito de herança múltipla, e isto é estabelecido pela cláusula superclasses. A subclasse herda todas as definições introduzidas nas superclasses e o seu estado é formado pelos componentes de estado definidos nas superclasses e os definidos na própria classe e o invariante é a conjunção dos invariantes das superclasses e o invariante definido localmente. Caso os esquemas que definem o estado inicial de uma subclasse sejam homônimos a esquemas introduzidos na cláusula initialstates nas superclasses, o estado inicial da subclasse corresponde à conjunção destes esquemas homônimos.

Esta classe modela o estado de uma única **OT**. Além dos componentes herdados da classe *OTPrevisao*, ela inclui componentes para representar a descrição da **OT** (*desc*), a data do início da execução da **OT** (*inicio*) e a data de conclusão da **OT** (*conc*).

superclasses *OTPrevisao*

Os esquemas *ConcluiOK* e *TarefasEmAberto*, que representam, respectivamente, os caso de sucesso e de erro da operação de conclusão de uma **OT**, são locais à classe.

private *ConcluiOK*, *TarefasEmAberto*

state

$desc : String$
$inicio : Data$
$conc : Data$

initialstates

O estado inicial desta classe consiste basicamente da atribuição de valores iniciais aos componentes de estado. A variável *desc* é inicializada a partir de *desc?* passado como entrada e as variáveis *inicio* e *conc* são inicializadas como datas nulas. O esquema de inicialização *Init* definido na superclasse *OTPrevisao* é implicitamente incluído no esquema *Init* definido abaixo.

Init

$\Delta(desc, inicio, conc)$

$desc? : String$

$desc' = desc? \land inicio' = DataNula \land conc' = DataNula$

operations

O tipo enumerado *Mensagem* representa as mensagens retornadas pela execução das operações.

$Mensagem ::= ok \mid TarefasNaoConcluidas$

A operação *ConcluiOK* estabelece uma data de conclusão da **OT**, desde que todas as suas tarefas já tenham sido concluídas. O caso de erro é descrito em seguida.

ConcluiOK

$\Delta(conc)$

$\Xi(tarefas)$

$dataConc? : Data$

$resposta! : Mensagem$

$\forall tar : tarefas \bullet tar.conc \neq DataNula \land tar.conc \leq_{data} dataConc?$

$conc' = dataConc?$

$resposta! = ok$

TarefasEmAberto

$\Xi(tarefas)$

$dataConc? : Data$

$resposta! : Mensagem$

$\exists tar : tarefas \bullet tar.conc = DataNula \lor tar.conc >_{data} dataConc?$

$resposta! = TarefasNaoConcluidas$

A operação final é definida como:

$Conclui \mathrel{\widehat=} ConcluiOK \lor TarefasEmAberto$

EndClass *OT*.

Class *SistemaDeManutencao*

Esta classe representa o sistema de manutenção como um todo. Especificamos o sistema de manutenção como um mapeamento de identificadores de **OT's**, representado pelo givenset *OTId*, para **OT's**, representadas pela classe *OT*.

givensets *OTId*

private *OTNaoExiste*, *AtualizaOT*, *ConcluiOTOK*

state

$OTs : OTId \pfun OT$

As únicas definições privativas da classe são *OTNaoExiste*, que representa o caso de erro das operações, *AtualizaOT* que, como veremos em seguida, corresponde ao esquema que *promove* as operações da classe *OT* para esta classe. e *ConcluiOTOK* que corresponde ao casos de sucesso na conclusão de uma **OT**.

operations

Mais uma vez, *Mensagem* representa a mensagem retornada pela execução das operações.

Mensagem ::= *ok* | *OTNaoExistente*

O componente *OTs* desta classe representa todas as **OT**'s existentes no Sistema de Manutenção e é fácil perceber que, muitas vezes, será necessário efetuar uma operação sobre uma única **OT** deste conjunto. Podemos, então, utilizar a técnica de *Promoção* [Woo90] para reutilizar as operações definidas na classe **OT**. O esquema *AtualizaOT* definido abaixo corresponde ao *framing schema* utilizado na promoção. Este esquema, através do identificador, seleciona no estado global *OTs* uma única **OT** sobre a qual a operação será efetuada. O novo estado global será o estado anterior, adicionando ao mesmo o novo estado local da **OT**, *ot'*, relacionado ao identificador *otId*?. Em todas as operações, a **OT** escolhida deve existir no estado global.

AtualizaOT

$\Delta(OTs)$
$otId? : OTId$
$ot,$
$ot' : OT$

$otId? \in OTs\ dom$
$ot = OTs(otId?)$
$OTs' = OTs \oplus \{otid? \mapsto ot'\}$

A operação *ConcluiOTOK* promove a operação *Conclui* definida na classe *OT*. Após definir a operação a partir do *framing schema* e da operação local, devemos esconder os componentes locais *ot* e *ot'* que não são de interesse na versão final da operação.

ConcluiOTOK

AtualizaOT
ot Conclui

$ConcluiOT \mathrel{\widehat=} ConcluiOTOK \backslash (ot, ot')$

EndClass *SistemaDeManutencao*.

3 O Método de Refinamento

Nesta seção, apresentamos o método de refinamento através de sua aplicação à especificação apresentada na seção anterior até obtermos o código em Eiffel correspondente.

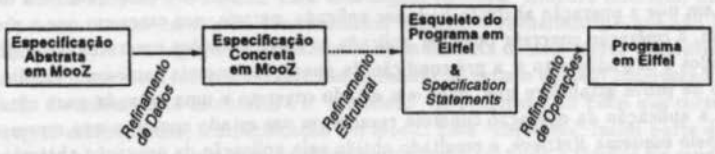

Figura 2: O Refinamento em MooZ

O método de refinamento considerado envolve o *refinamento de dados* e *de operações*, após os quais obtém-se, respectivamente, uma representação de dados mais adequada às estruturas de dados em Eiffel e operações expressas por algoritmos que utilizam os construtores de Eiffel. Em algum ponto do método de refinamento, porém, é necessário iniciar a tradução da notação das especificações em MooZ para a notação de programas em Eiffel. Isto implica na conversão das estruturas e construções de uma notação para outra, a qual é realizada em uma etapa adicional, chamada *Refinamento Estrutural*. Assim, o método considera o refinamento das especificações

em três aspectos: o refinamento de dados, o refinamento estrutural e o refinamento de operações. O processo de refinamento está esquematizado na Figura 1. Vale ressaltar que este método de refinamento é iterativo e composicional, ou seja, podemos refinar partes da especificação separadamente que a especificação como um todo estará sendo refinada e o refinamento pode ser feito em vários passos onde, a cada passo, a versão obtida é um refinamento da especificação original correspondente. Todas as etapas do processo serão discutidas a partir de sua aplicação ao estudo de caso apresentado na Seção 2.1. Uma apresentação mais completa e detalhada do método pode ser encontrada em [Cor94].

3.1 O Refinamento de Dados

Nesta seção apresentamos um exemplo do processo de refinamento de dados da especificacão. Esta é a primeira etapa do método pois ocorre totalmente dentro do contexto de MooZ. Nesta etapa, são escolhidas representações mais adequadas para os dados, ou por ser possível implementá-las diretamente ou por serem mais eficientes. Esta etapa segue uma abordagem *rigorosa*, semelhante à apresentada em [Wor92]. A relação que existe entre a representação abstrata dos dados e a representação concreta é estabelecida através de um esquema que inclui os componentes do estado abstrato e do estado concreto em sua parte declarativa e cujo predicado estabelece a relação entre eles. Este refinamento implica em algumas obrigações de provas. Mais especificamente, considerando S o estado abstrato do sistema, S_c o estado concreto, *Retrieve* o esquema que relaciona a representação abstrata dos dados com a concreta, *Init* o estado inicial abstrato, $Init_c$ o estado inicial concreto, Op uma operação abstrata e Op_c a operação concreta correspondente, é necessário provar as seguintes propriedades:

$$\vdash \exists S_c' \bullet Init_c \tag{1}$$

$$Init_c \vdash \exists S' \bullet (Init \wedge Retrieve') \tag{2}$$

$$(pre\ Op) \wedge Retrieve \vdash pre\ Op_c \tag{3}$$

$$(pre\ Op) \wedge Retrieve \wedge Op_c \vdash \exists S' \bullet (Op \wedge Retrieve') \tag{4}$$

onde S', $Retrieve'$ e S_c' dizem respeito ao estado posterior. A prova de (1) estabelece que é possível achar um estado inicial para a nova representação e que este estado é consistente. A segunda obrigação de prova serve para mostrar que qualquer estado inicial concreto é correto, ou seja, de acordo com a relação expressa pelo esquema *Retrieve*, qualquer estado inicial concreto representa um estado inicial abstrato. A obrigação de prova (3) estabelece que, para os estados e entradas em que a operação abstrata pode ser aplicada, ou seja, nos casos em que a pré-condição é satisfeita, a operação concreta pode ser aplicada às representações concretas correspondentes a estes estados e entradas, isto é, a pré-condição da operação concreta também é válida. A última obrigação de prova estabelece que, dado um estado concreto e uma entrada para uma operação concreta, a aplicação da operação concreta resulta em um estado concreto que representa, como expresso pelo esquema *Retrieve*, o resultado obtido pela aplicação da operação abstrata ao estado e à entrada correspondentes ao estado e à entrada concretos.

Como o processo de refinamento baseia-se no apresentado em [Wor92, Woo93], ilustramos esta etapa do método aplicando-a apenas a uma parte da especificação. Inicialmente, o componente *tarefas* da classe *OTPrevisao*, que está definido como um conjunto de tarefas ($\mathbb{P}$ *Tarefa*), é refinado para uma seqüência ($Seq(X \setminus Tarefa)$)[1], que pode ser implementada pelo tipo **List** de Eiffel. Assim, o novo estado da classe, que chamamos de estado concreto, passa a ser[2]:

[1] A notação $(x \setminus y)$ representa a substituição de todas as ocorrências de x por y e, neste caso, corresponde á instanciação do *givenset* X da classe *Seq* por *Tarefa*.

[2] Os componentes do estado concreto estão subscritos com $_c$ para diferenciar dos componentes do estado abstrato. Nas etapas posteriores do processo de refinamento, onde assumimos que os dados já foram refinados e toda a especificação já está em um estado concreto, a subscrição é suprimida novamente.

$$tarefas_c : Seq(X \setminus Tarefa)$$
$$inicioPrev_c : Data$$
$$\forall tar : ran(tarefas_c) \bullet tar.dependencias \subset ran(tarefas_c)$$
$$\exists tar : ran(tarefas_c) \bullet tar.dependencias = \varnothing$$

O seguinte esquema relaciona os componentes do estado abstrato com os do estado concreto:

Retrieve

$$\Xi(tarefas, tarefas_c, inicioPrev, inicioPrev_c)$$
$$tarefas = ran(tarefas_c)$$
$$inicioPrev = inicioPrev_c$$

A seguir, apresentamos a nova definição da operação de inicialização, *Init* e da operação *IncluiTarefasOK*, considerando o novo estado da classe *OTPrevisao*. As obrigações de prova relacionadas ao esquema de inicialização e à operação *IncluiTarefasOK* geradas podem ser facilmente verificadas.

Init$_c$

$$\Delta(tarefas_c, inicioPrev)$$
$$tarefas_c? : \mathbb{P}\, Tarefa$$
$$tarefas'_c = tarefas_c?$$
$$inicioPrev' = DataNula$$
$$\forall tar : tarefas_c? \bullet tar.dependencias \subset tarefas_c?$$
$$\exists tar : tarefas_c? \bullet tar.dependencias = \varnothing$$

IncluiTarefasOK$_c$

$$\Delta(tarefas_c)$$
$$tarefas_c? : \mathbb{P}\, Tarefa$$
$$resposta! : Mensagem$$
$$tarefas? \cap (tarefas\ ran) = \varnothing$$
$$tarefas'_c\ ran = (tarefas_c\ ran) \cup tarefas_c?$$
$$resposta! = ok$$

3.2 O Refinamento Estrutural

Uma vez refinados os dados, podemos começar a converter a notação de MooZ para Eiffel, através do refinamento estrutural. Para que este processo também seja rigoroso o refinamento das estruturas deve levar em consideração a semântica formal das duas linguagens. Porém, até o nosso conhecimento, nem MooZ nem Eiffel possuem uma semântica formal completamente definida. Com isso, o processo de refinamento das estruturas é sistemático, onde definimos regras sintáticas para converter de uma notação para outra. Consideramos, por exemplo, que uma classe em MooZ deve ser implementada por uma classe em Eiffel, porém, não podemos verificar se a semântica da classe em MooZ é preservada após o seu mapeamento para Eiffel. Esta é a razão para a existência de uma etapa adicional, de forma que é possível isolar o processo que segue uma abordagem sistemática das outras etapas, que seguem uma abordagem rigorosa. Embora informal, utilizamos regras que documentam as características assumidas durante o refinamento e que são bastante intuitivas. Um exemplo é a regra 5 apresentada no apêndice. As demais regras são apresentadas e discutidas informalmente à medida em que são aplicadas ao estudo de caso considerado.

O resultado do refinamento estrutural é o "esqueleto" do programa em Eiffel que corresponde, segundo as regras estabelecidas, à especificação em MooZ. Este "esqueleto" inclui parte do código final que se deseja obter com o processo de refinamento como um todo e trechos de especificação expressos como *specification statements* [Mor90] que devem ser refinados na etapa de refinamento de operações. Um *specification statement* é definido como um par de predicados $w : [P, Q]$ que representam uma operação da especificação, onde w, chamado de *frame*, é uma lista das variáveis que são alteradas pela operação, P é a pré-condição da operação e Q corresponde à pós-condição. Os *specification statements* expressam as propriedades de uma operação através da lógica de predicados e devem ser refinados para serem expressos como comandos da linguagem de programação. Isto implica que o refinamento de um *specification statement* pode ser bastante complexo caso o seu nível de abstração seja distante do nível de abstração que pode ser expresso pela linguagem de implementação.

No refinamento estrutural, a estrutura hierárquica da especificação é preservada. Em Eiffel a unidade básica de programação é uma classe, logo, toda classe especificada em MooZ é implementada por uma classe em Eiffel. Eiffel também suporta o conceito de herança múltipla, portanto, a relação de herança em MooZ, estabelecida através da cláusula superclasses, é mapeada diretamente para Eiffel, na cláusula **inherit**. Eiffel permite, ainda, a definição de classes genéricas, parametrizadas, que devem ser instanciadas quando utilizadas. Desta forma, os *givensets* são transformados em parâmetros das classes em Eiffel. Em alguns casos, isto implica que durante a tradução de uma classe de MooZ, a classe correspondente em Eiffel tem que ser parametrizada pelo menos pelos mesmos parâmetros definidos nas classes que ela utiliza, além dos parâmetros decorrentes dos *givensets* que não foram refinados na etapa anterior.

As propriedades e o comportamento dos objetos de uma classe em Eiffel são descritos por *features*, que podem ser atributos ou rotinas. Os atributos representam dados associados a instâncias de uma classe e as rotinas representam procedimentos ou funções que podem ser aplicados aos objetos. A notação para referenciar um atributo ou rotina é a mesma: $o.f$, onde o é um objeto e f é um *feature* (atributo ou rotina). Os features são introduzidos em cláusulas **feature** e a cada uma destas cláusulas é possível associar a visibilidade dos *features* introduzidos nela. Os *features* correspondentes às definições públicas introduzidas na especificação devem ser implementados em uma cláusula **feature** que possui como cliente a classe pré-definida de Eiffel **ANY** (ou a declaração de clientes é omitida), indicando que estes *features* são visíveis a qualquer classe. Caso contrário, é indicado na cláusula **feature** a classe **NONE** como cliente, que também é uma classe especial de Eiffel que indica que nenhuma classe pode ter acesso aos *features* introduzidos nesta cláusula. Eiffel permite, ainda, que se defina uma lista de classes clientes, que são as únicas a terem acesso ao conjunto de *features* introduzido em uma determinada cláusula **feature**. Em MooZ, porém, o escopo estabelecido para as definições é o mesmo para todas as classes, sendo as classes clientes sempre estabelecidas por **NONE** ou **ANY**.

As definições dentro de cada classe em MooZ são traduzidas para atributos ou rotinas da classe correspondente em Eiffel que são introduzidos nas cláusulas **feature** e, durante o refinamento, definimos uma cláusula **feature** diferente para cada cláusula mapeada da especificação, comentando-as.

Eiffel possui um mecanismo de asserções que possibilita a expressão de propriedades formais de uma classe e de suas operações. É possível estabelecer, através da cláusula **invariant**, um invariante para a classe e, para cada operação, é possível explicitar sua pré e pós-condição, através das cláusulas **require** e **ensure**, respectivamente. Estas cláusulas, no entanto, servem basicamente para documentação e depuração de programas. A rigor, estas cláusulas podem ser omitidas pois a pré e a pós-condição das operações são documentadas formalmente nos *specification statements* correspondentes e são preservadas durante o refinamento. É interessante, contudo, manter a cláusula **invariant**, já que o invariante da classe não é explicitado diretamente pelos *specification statements*.

Assumimos a existência de algumas classes e operações em Eiffel como **SET**, que representa o construtor de tipo $\mathbb{P}$ de MooZ, **LIST**, que representa o tipo *Seq*, **Map**, que representa o tipo mapeamento, e **Date**, possuindo operações adequadas à manipulação de datas. Consideramos, ainda, que cada uma destas classes possui operações equivalentes às operações definidas para os tipos correspondentes em MooZ. Eiffel não tem como definir diretamente tipos enumerados e, portanto, decidimos representar cada constante do tipo *Mensagem* como um valor do tipo **STRING**.

Ilustramos esta etapa do método de refinamento através da sua aplicação a uma parte da especificação apresentada na Seção 2.1. O refinamento das partes da especificação que foram suprimidos neste trabalho, por questão de espaço, é trivial e pode ser feito de maneira análoga à apresentada nesta seção.

A seguir, mostramos o mapeamento da classe *OT* para Eiffel. Na especificação esta classe

é subclasse de *OTPrevisao* e esta relação é preservada na implementação. Os componentes de estado são transformados em atributos da classe e seus tipos já estão convertidos para Eiffel. A operação de inicialização *init* é indicada na cláusula **creation** abaixo.

class *OT*[*TarId, CodFunc, CodExec*]
  inherit
    OTPrevisao[*TarId, CodExec, CodFunc*]
    rename *init* **as** *initSuper*
    creation
    init
  feature - - *Componentes de Estado*
    desc: **String**;
    inicio: **Date**;
    conc: **Date**

Para utilizarmos a operação *init* definida na superclasse *OTPrevisao* para definir incrementalmente a operação de inicialização da classe *OT*, como ocorre na especificação, é necessário herdá-la renomeando-a para *initSuper*, como indicado na cláusula **rename**, evitando o conflito dos nomes. Na definição da operação *init* de *OT*, incluímos uma chamada ao método *initSuper* que inicializa todos os componentes herdados de *OTPrevisao*.

As operações definidas na classe são implementadas como rotinas (métodos) em Eiffel. As variáveis decoradas com "?" são implementadas como parâmetros da rotina correspondente e as variáveis decoradas com "!" são implementadas como saída do método e são representadas pela variável *Result*, pré-definida em Eiffel. As variáveis que definem o estado já estão incluídas implicitamente no método e as demais variáveis introduzidas em um esquema são implementadas como variáveis locais na rotina correspondente. O código do método, que aparece na cláusula **do**, é representado por um *specification statement* e é obtido formalmente na próxima etapa do refinamento: o refinamento de operações. Para converter a notação das partes da especificação que serão refinadas formalmente para *specification statements*, adaptamos o método proposto em [Kin90] que converte esquemas em Z para *specification statements*. Neste método, são calculadas a pré e a pós-condição dos esquemas. A pré-condição é obtida através da fórmula padrão para esquemas: $\exists S', x! : T \mid inv \bullet pred$, onde S' representa a lista de variáveis do estado posterior, $x!$ representa as variáveis de saída, T é o tipo respectivo de cada uma delas, inv é o invariante da classe definido pelo predicado do esquema anônimo e $pred$ é o predicado do esquema. A pós-condição consiste do predicado original do esquema, omitindo-se os predicados já expressos na pré-condição, eliminando-se a decoração "'" e decorando as referências aos componentes do estado anterior com "$_o$". No *frame* do *specificatioon statement* são incluídos os componentes de estado introduzidos em uma lista Δ e a variável *Result*, que representa as variáveis de saída.

feature - - *Inicialização*
  init(*tarefas?*:**List**[*Tarefa*[*TarId, CodExec, CodFunc*]],*desc?*:**STRING**) **is**
    do
      initSuper(*tarefas?*);
      $desc, inicio, conc : \left[true, \left(\begin{array}{l} desc = desc? \\ inicio = DataNula \\ conc = DataNula \end{array} \right) \right]$
  end; - - *init*

Através das regras de atribuição (regra 2) e de composição sequencial (regra 4) apresentadas no apêndice, é fácil perceber que o *specification statement* acima pode ser implementado com uma seqüência de atribuições.

Na próxima cláusula **feature** são introduzidas as definições privativas da classe *OT* declaradas na cláusula private da especificação. O string **"tarefasNaoConcluídas"** retornado como resultado do método *tarefasEmAberto* que implementa a operação *TarefasEmAberto* representa o valor *TarefasNaoConcluídas* do *free type Mensagem* da especificação. O mesmo ocorre com o

valor *ok*, que é implementado pelo string "ok".

```
feature {NONE} - - operacoes privativas
  tarefasEmAberto(dataConc?:Date): STRING is
    do
      Result : [ ( ∃ tar : tarefas •
                   tar.conc = DataNula ∨
                   tar.conc >data dataConc? ) , result = "tarefasNaoConcluidas" ]
    end; - - tarefasEmAberto
  concluiOK(dataConc?: Date):STRING is
    do
      conc : [ ( ∀ tar : tarefas • tar.conc ≠ DataNula ∧
                                   tar.conc ≤data dataConc? ) , ( conc = dataConc?
                                                                  Result = "ok" ) ]
    end; - - concluiOK
```

Na cláusula **feature** a seguir estão os métodos que correspondem às operações públicas da classe. Mostramos apenas o método correspondente à versão total da operação *Conclui*. Como esta definição é feita em termos de disjunção, utilizamos a regra 5[3], apresentada no apêndice, para lidar com estas operações. Neste caso, a pré-condição do esquema correspondente ao primeiro termo da disjunção não pode ser expressa diretamente em Eiffel. Portanto, utilizamos uma variável local b do tipo **BOOLEAN** para calcular o valor da pré-condição. O valor de b é obtido através do refinamento do *specification statement* remanescente no método correspondente. Este refinamento será mostrado na próxima seção.

```
  conclui(dataConc?:Date):STRING is
    local
      b: BOOLEAN
    do
      b : [ true, ( b ⇔ (∀ tar : (tarefas ran) • tar.conc ≠ DataNula ∧
                                 tar.conc ≤data dataConc?) ) ] (†)
      if b then
        Result := concluiOK(dataConc?);
      else
        Result := tarefasEmAberto(dataConc?)
    end; - - conclui
end - - Class OT
```

A tradução da classe *SistemaDeManutencao* é apresentada a seguir.

```
class SistemaDeManutencao [TarId,CodFunc,CodExec,OTId]
  feature - - Componentes de Estado
    OTs: MAP[OTId,OT[TarId,CodFunc,CodExec]]
```

Para esta classe, apresentamos apenas o refinamento estrutural dos esquemas envolvidos com o mecanismo de promoção utilizado para modularizar a especificação. O processo de refinamento estrutural das demais operações é similar ao que já foi mostrado e, portanto, será suprimido.

O método *atualizaOT* definido abaixo implementa o *framing schema AtualizaOT* utilizado na promoção de operações nesta classe e retorna a **OT** selecionada do componente global *OTs*.

```
feature {NONE} - - operacoes privativas
    atualizaOT(otId?:OTId):OT[TarId,CodFunc,CodExec] is
      do
        Result : [otId? ∈ OTs dom, Result = OTs get(otId?)]
    end; - - atualizaOT
  concluiOTOK. mostrado abaixo.
    concluiOTOK(ot?:OT[TarId,CodFunc,CodExec]; dataConc?: Date): INTEGER is
      local
        ot: OT[TarId.CodFunc.CodExec]
      do
        ot := atualizaOT(ot?);
```

[3]Esta regra, embora possa ser formalizada, só se aplica a esta etapa do refinamento, pois a notação considerada ainda é a de esquemas.

```
        Result := ot.conclui(dataConc?)
    end; -- concluiOTOK
end -- Class SistemaDeManutencao
```

O esquema promovido *Conclui* é implementado como um método da classe *OT* e é passado como mensagem ao objeto *ot*. Vale ressaltar que a assinatura do método *atualizaOT* não corresponde à assinatura do esquema original, pois, na implementação está sendo retornado um resultado. Porém, a tradução de um *framing schema* é um caso específico e pode sempre ser tratado desta maneira, ou seja, o componente local selecionado no estado global deve ser retornado como resultado da rotina que implementa este esquema. Além disso, linguagens de programação orientada a objetos possuem efeito colateral, de maneira que, ao alterar o estado de um objeto, todas as referências para este objeto são atualizadas automaticamente para referenciar o novo estado, logo, o predicado que estabelece o novo valor do componente local no *framing schema* pode ser ignorado e basta uma única variável local *ot* para representar os estados anterior e posterior deste componente. [Woo92] discute como o mecanismo de promoção de Z pode ser implementado como chamadas de procedimentos em linguagens de programação procedimentais, utilizando regras do cálculo de refinamento apresentadas em [Mor90]. O mesmo raciocínio pode ser aplicado para derivar regras que formalizem o refinamento de operações promovidas como passagem de mensagem, em linguagens orientadas a objetos.

Nesta seção obtemos o "esqueleto" de uma parte da implementação em Eiffel de maneira sistemática utilizando alguma regras informais discutidas no decorrer desta etapa do refinamento. Na próxima seção apresentamos mais uma etapa formal do refinamento onde, a partir dos *specification statements* que restaram no código obtido até este ponto, utilizamos regras formais para derivar trechos de programa em Eiffel.

3.3 O Refinamento de Operações

specification statements em código em Eiffel, usando uma abordagem rigorosa baseada no cálculo de refinamento de Morgan [Mor90]. Nesta etapa, aplicamos regras de transformação que foram adaptadas de regras bem estabelecidas utilizadas por cálculos de esquema considerando a notação das construções de Eiffel. Após o refinamento de operações, todos os *specification statements* que existiam inicialmente já estão completamente refinados para Eiffel, completando o processo de refinamento da especificação inicial. Para ilustrar esta etapa do processo, mostramos a derivação do código correspondente ao cálculo da guarda do método *conclui* da classe *OT* que é expresso pelo *specification statement* (†), utilizando algumas das regras introduzidas no apêndice. Por concisão, apresentamos apenas alguns passos do refinamento, mas o refinamento passo a passo pode ser encontrado em [Cor94].

A estrutura do predicado da pós-condição de (†) sugere que o mesmo seja implementado como um laço que percorre todo o conjunto de tarefas para verificar se todos os elementos *tar* satisfazem à condição ($tar.conc \neq DataNula \land tar.conc \leq_{data} dataConc?$). Caso à condição seja satisfeita, *b* assume o valor *true*, senão, *b* tem valor *false*. Para percorrer todos os elementos do conjunto (*tarefas ran*), utilizaremos um conjunto auxiliar *ct* do mesmo tipo no qual incluíremos todos os elementos já testados de (*tarefas ran*), controlando quais já foram verificados. Ao final da execução do laço, o conjunto *ct* possuirá os mesmos elementos do conjunto (*tarefas ran*) original.

Para expressar esta estratégia de implementação, declaramos *ct* como uma variável local e fortalecemos a pós-condição, obtendo:

(†) $\sqsubseteq$ Introdução de variável local (regra 3) e fortalecimento da pós-condição (regra 1)

$$
\begin{array}{l}
|[\, VAR\ ct : \mathbb{P}\ Tarefa \bullet \\
ct, b : \left[\begin{array}{l} true, \\ \left(\begin{array}{rcl} b & \Leftrightarrow & (\forall\, tar : (tarefas\ ran) \bullet tar.conc \neq DataNula \land \\ & & \quad tar.conc \leq_{data} dataConc?) \\ ct = (tarefas\ ran) & & \end{array} \right) \end{array} \right] \\
\,]|
\end{array}
$$

Utilizando algumas regras de refinamento, podemos encontrar um *specification statement* da forma: $w : [pre, inv \wedge \neg G]$, onde inv corresponde ao predicado:

$$((((\forall tar : ct \bullet tar.conc \neq DataNula \wedge tar.conc \leq_{data} dataConc?) \wedge b) \vee$$
$$((\exists tar : ct \bullet tar.conc = DataNula \vee tar.conc >_{data} dataConc?) \wedge \neg b))) \wedge$$
$$ct \subseteq (tarefas\ ran))$$

e G é: $ct \neq (tarefas\ ran)$. Assim, podemos introduzir o comando de iteração, obtendo:

```
⊑  Introdução de Iteração (regra 6)
   from ct, b : [true, inv]
   invariant ct.isSubsetOf(tarefas.elems)
     -- and forall tar : ct, (tar.conc <> DataNula and ...
   variant (tarefas.elems).card - ct.card -- (#(tarefas ran) − #ct)
   until ct = tarefas.elems -- ct = (tarefas ran)
   loop
   ct, b : [( inv                ), ( inv                                                  )]
           ( ct ≠ (tarefas ran)  )  ( 0 ≤ (#(tarefas ran) − #ct) < (#(tarefas ran) − #ct0) )
   end
```

Após mais alguns passos, derivamos o código que implementa o cálculo de b:

```
local
  ct: SET[Tarefa];
  t: Tarefa
 do
 from
   ct := emptyset;
   b := true
  invariant ct.isSubsetOf(tarefas.elems) -- and forall tar : ct, ...
  variant (tarefas.elems).card - ct.card -- (#(tarefas ran) − #ct)
  until ct = tarefas.elems -- ct = (tarefas ran)
  loop
   t := ((tarefas.elems).diff(ct)).choose;
   ct := ct.add(t);
   if (t.conc = dataNula or t.conc > dataConc?) then
    b := false
   else;
   end -- if
  end -- loop
```

Para o refinamento completo do *specification statement* visto como exemplo nesta seção, foram necessários 15 passos de refinamento. Por este exemplo é possível perceber quão grande pode ser a diferença entre o nível de abstração de uma especificação e de um programa. Neste caso, o *specification statement* a ser refinado, embora aparentemente simples, é muito abstrato. Para derivar formalmente o código correspondente a ele, foram necessários muitos passos e a adição de muita informação à especificação inicial. Foi necessário, por exemplo, a introdução de duas novas variáveis locais: ct e t. A complexidade envolvida no processo de refinamento rigoroso pode ser justificada pela certeza de que o código final realmente satisfaz à especificação.

4 Conclusões

Neste trabalho, apresentamos um método de refinamento através de sua aplicação a um estudo de caso. O método consiste de três etapas distintas: refinamento de dados, refinamento estrutural e refinamento de operações. A primeira etapa segue uma abordagem rigorosa baseada no refinamento de dados considerado em [Wor92]. A segunda etapa segue uma abordagem sistemática, pois as

regras utilizadas nesta etapa são puramente sintáticas. Para tornar esta abordagem rigorosa ou formal, seria necessário provar que as regras aplicadas preservam a semântica da especificação, o que depende da semântica formal das duas linguagens envolvidas no processo — MooZ e Eiffel — que, até nosso conhecimento, ainda não foram completamente definidas. A última etapa também segue uma abordagem rigorosa, baseada no cálculo de refinamento apresentado em [Mor90].

O refinamento de dados é ilustrado por um único exemplo, por ser bastante simples e por Eiffel permitir que se definam novas classes que implementem os tipos de dados utilizados na especificação. No refinamento estrutural, mostramos como converter as estruturas de uma especificação em MooZ para Eiffel, obtendo o "esqueleto" das classes e métodos correspondentes às definições especificadas. Nesta etapa, convertemos a notação das operações a serem refinadas formalmente na etapa seguinte para *specification statements*. O refinamento formal de operações é ilustrado através de um exemplo que mostra a dificuldade do processo.

Vale ressaltar que, para a *proposição* do método, a etapa de refinamento estrutural foi a que exigiu mais trabalho. Isto deve-se ao fato de que foi preciso estudar minunciosamente as construções de MooZ e de Eiffel para propor um mapeamento entre elas. No entanto, para efeito de refinamento, esta etapa é bastante simples, envolvendo apenas um passo para converter as estruturas de MooZ para Eiffel. Para as etapas formais/rigorosas, adaptamos técnicas de refinamento já existentes para considerar a notação de MooZ e de Eiffel. Em particular, o refinamento de dados baseia-se no refinamento de dados apresentado em [Wor92] e o refinamento de operações baseia-se nas idéias do cálculo de refinamento de Morgan [Mor90]. Estas duas etapas podem envolver vários passos de refinamento e a maior parte do código final em Eiffel é obtido através do refinamento de operações, que é justamente a etapa mais árdua e trabalhosa do refinamento.

A principal contribuição deste trabalho é a apresentação de um estudo de caso não trivial de refinamento que corresponde a uma aplicação real, complementando o trabalho proposto inicialmente em [CSM94a, CSM94b]. O mecanismo de *promoção*, muito utilizado em especificações em Z e que foi adaptado para MooZ, é considerado no estudo de caso, onde é adotada uma estratégia de refinamento que conserva a modularidade obtida por meio deste mecanismo. Nossa intenção é completar, em um futuro próximo, o desenvolvimento de todo o sistema de manutenção industrial parcialmente apresentado neste trabalho utilizando este método de refinamento.

Agradecimentos

Agradecemos a In Forma Software Ltda. por permitir o uso do sistema de manutenção e, em especial, a Ismar Neumann Kaufman pelas valiosas discussões relacionadas a este trabalho.

Referências

[Cor94] Virgínia Adélia de Oliveira Cordeiro. De MooZ para Eiffel – Uma Abordagem Rigorosa para o Desenvolvimento de Sistemas. Tese de Mestrado, Universidade Federal de Pernambuco, Departamento de Informática, março 1994.

[CSM94a] Virgínia A. O. Cordeiro, Augusto Sampaio, e Silvio L. Meira. De MooZ para Eiffel. Em *XXI SEMISH*. Caxambu, MG, agosto 1994.

[CSM94b] Virgínia A. O. Cordeiro, Augusto Sampaio, e Silvio L. Meira. From MooZ to Eiffel — A Rigorous Approach to System Development. Em *Formal Methods Europe*, Barcelona, outubro 1994. A ser publicado em LNCS, Springer-Verlag, Heidelberg, DE.

[Kin90] S. King. Z and the Refinement Calculus. Relatório Técnico PRG-79, Oxford University Computing Laboratory, Programming Researh Group, fevereiro 1990.

[MC90] S. R. L. Meira e A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. Em Prof. C. J. van Rijsbergen, editor. *Workshop on Computing Series*, páginas 173 – 192, Oxford - UK, dezembro 1990. Springer-Verlag.

[Mey92] Bertrand Meyer. *EIFFEL : The Language.* Object-Oriented Series. Prentice Hall, 1992.

[Mor90] Caroll Morgan. *Programming From Specifications.* Prentice-Hall. 1990.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice hall, C.A.R. Hoare Series Editor, 1989.

[Woo90] J.C.P. Woodcock. Mathematics as a Management Tool: Proof Rules for Promotion. Em B. A. Kitchenham, editor, *Software Engineering for Large Software Systems.* Elsevier Applied Science, 1990.

[Woo92] J. C. P. Woodcock. Implementing Promoted Operations in Z. Em C. B. Jones, R. C. Shaw, e T. Denvir, editores, *5th Refinement Workshop.* BCS/Springer-Verlag Workshops in Computing, 1992.

[Woo93] J. C. P. Woodcock. Using Standard Z: Specification, Refinement & Proof. Book to be published, janeiro 1993.

[Wor92] J. B. Wordsworth. *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley, 1992.

A Algumas Regras de Refinamento

As regras introduzidas abaixo são adaptações de regras apresentadas em [Mor90, Kin90].

1. *Fortalecimento da pós condição*: Se $pre[w\backslash w_0] \wedge post_2 \vdash post_1$, então:
   $$w : [pre, post_1] \sqsubseteq w : [pre, post_2]$$
2. *Introdução de Atribuição Simples*: Se $(w = w_0) \wedge pre \vdash post[w\backslash E]$, então:
   $$w, x : [pre, post] \sqsubseteq w := E,$$ onde E é uma expressão.
3. *Introdução de Variáveis Locais*: Seja l uma variável nova de tipo T que não aparece livre em pre e $post$, então: $w : [pre, post] \sqsubseteq |[\ \mathbf{var}\ l : T \bullet w, l : [pre, post]\ ]|$
4. *Composição Sequencial*: Dado um predicado mid qualquer,
   $$w, x : [pre, post] \sqsubseteq x : [pre, mid]; w, x : [mid, post]$$
5. *Introdução de Condicional*: Se $Op \mathrel{\widehat{=}} Op_1 \vee \ldots \vee Op_n$, então:
   $$Op \sqsubseteq |[\mathbf{VAR}\ b_1, \ldots, b_n : \mathbf{BOOLEAN} \bullet b_1, \ldots, b_n : \left[true, \begin{pmatrix} b_1 \Leftrightarrow pre\ Op_1 \\ \ldots \\ b_n \Leftrightarrow pre\ Op_n \end{pmatrix} \right];$$
   if b_1 **then** Op_1^* ... **elsif** b_n **then** Op_n^* **end**]|

   onde $Op_i^*, i = 1, \ldots, n$, representam os *specification statements* correspondentes a cada Op_i. Cada variável local b_i é utilizada para calcular a pré-condição de cada operação Op_i, caso esta não possa ser expressa diretamente em Eiffel. Se $Op \mathrel{\widehat{=}} Op_1 \vee Op_2$ e se $pre\ Op_1 = \neg pre\ Op_2$, então podemos colocar no código que refina Op a segunda guarda como $\neg b$ ou, simplesmente, utilizamos a cláusula **else**.
6. *Iteração* : $w : [pre, inv \wedge \neg G] \sqsubseteq$
   from $w : [pre, inv]$
   invariant inv
   variant V
   until $\neg G$
   loop $w : [inv \wedge G, inv \wedge (0 \leq V < V_0)]$
   end

   Esta regra introduz o comando de iteração, onde inv é uma propriedade que é sempre verdade durante a execução do laço, V é o variante do laço, representado por uma função que retorna um um número inteiro positivo que decresce a cada execução, garantindo que o laço não é infinito, V_0 corresponde a $V[w\backslash w_0]$, $\neg G$ é a condição de parada e, portanto, o código da cláusula **loop** é executado até que a condição G não seja mais válida.