Fellipe Augusto Ugliara

Replicação Orientada a Metaprogramação

Sorocaba, SP

12 de Junho de 2018

Fellipe Augusto Ugliara

Replicação Orientada a Metaprogramação

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Computação Científica e Inteligência Computacional.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. José de Oliveira Guimarães

Coorientador: Prof. Dr. Gustavo Maciel Dias Vieira

Sorocaba, SP

12 de Junho de 2018

Ugliara, Fellipe Augusto

Replicação Orientada a Metaprogramação / Fellipe Augusto Ugliara. -- 2018.
110 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador: Prof. Dr. José de Oliveira Guimarães; Coorientador: Prof. Dr. Gustavo Maciel Dias Vieira
Banca examinadora: Prof. Dr. José de Oliveira Guimarães, Prof. Dr. João José Neto, Prof. Dr. Sahudy Montenegro González
Bibliografia

1. Replicação. 2. Metaprogramação. 3. Linguagem de Programação. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)
Bibliotecário(a) Responsável: Maria Aparecida de Lourdes Mariano – CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Fellipe Augusto Ugliara, realizada em 12/06/2018:

José de Oliveira Guimarães

Prof. Dr. José de Oliveira Guimarães
UFSCar

Prof. Dr. João José Neto
USP

Profa. Dra. Sahudy Montenegro González
UFSCar

Dedico essa pesquisa às traças que fizerem bom uso do papel,
e a cada leitor que a tirar da gaveta.

Agradecimentos

Agradeço,

a Katia Ruriko Muramatsu por ser compreensiva e ajudar com inúmeras revisões.

a Ruriko Tasaki Muramatsu pelos almoços de domingo que não ajudei a preparar.

ao Bruno Henrique Ugliara pelas revisões e pelo apoio positivo a cada etapa completada.

ao Oswaldo Toshio Muramatsu pelas conversas ao final dos domingos de trabalho.

ao Nahim Alves de Souza por explicar as etapas burocráticas e ajudar com o LaTeX.

ao Gustavo Marcello pela revisão técnica e pela paciência em escutar cada ideia.

a Ana Ugliara e ao Milton Ugliara por sempre me lembrarem de seguir aprendendo.

a Pretinha por cada latido que me ajudou a continuar acordado escrevendo.

"A linguagem fez-se para que nos sirvamos dela,
não para que a sirvamos a ela."
(A Língua Portuguesa, Fernando Pessoa)

Resumo

O objetivo dessa pesquisa é mostrar como aplicações distribuídas, coesas e pouco acopladas podem ser desenvolvidas. A proposta é usar técnicas de metaprogramação em compilação para automatizar parte do desenvolvimento do código-fonte das aplicações e inspecionar esse código-fonte à procura de inconsistências. Para isso foi usado Treplica e Cyan. Treplica fornece uma estrutura para o desenvolvimento de aplicações distribuídas, enquanto a linguagem de programação Cyan provê suporte à metaprogramação. Esses recursos permitiram a criação de componentes que podem ser usados para desenvolver aplicações replicadas. Nessa pesquisa foi possível aplicar metaprogramação para automatizar etapas do desenvolvimento dessas aplicações, tornando o código-fonte da replicação melhor isolado do restante da aplicação. A verificação desse código-fonte na busca de inconsistências também pode ser demonstrada. Essa pesquisa não tem a pretensão de tratar todas as possibilidades de automatização do desenvolvimento do código-fonte replicado, e também não pretende tratar todas as verificações necessárias à replicação. O objetivo é mostrar que, usando metaprogramação em compilação, é possível automatizar o desenvolvimento e a inspeção de código-fonte das aplicações de modo geral.

Palavras-chaves: Replicação. Metaprogramação. Linguagens de Programação.

Abstract

The objective of this research is to show how distributed, cohesive and coupled applications can be developed. The proposal is to use metaprogramming technics in compilation to automate part of the applications source code development, and to inspect the source code to find inconsistencies. For this purpose Treplica and Cyan were used. Treplica provides a structure to the distributed applications development, while Cyan provides the support to metaprogramming. These resources allowed the components creation, which can be used to develop replicated applications. In this research it was possible to apply metaprogramming to automate developmento steps of these applications, making the source code of replication better isolated from the rest of the application. The verification of this source code to find inconsistencies can also be made. This research does not intend to solve all automation possibilities of the replicated source code development, and it does not intend to solve all necessary verifications to the replication either. The objective of this research is to show that, using metaprogramming, it is possible to automate the development and the inspection of the applications source code in general.

Keywords: Replication. Metaprogramming. Programming Languages.

Lista de ilustrações

Lista de Códigos-Fonte

Sumário

Introdução

Quem, em computação, nunca ouviu a frase - "use essa linguagem que vai ser mais fácil de implementar esse algoritmo". Se comparada à visão teórica de computação, essa afirmação pode parecer sem fundamento, afinal todas as linguagens que são Turing-completo (SIPSER, 2006) permitem escrever os mesmos algoritmos e todas elas têm a mesma capacidade de representação. Também temos a tese de Church-Turing definindo que se uma função pode ser implementada por uma linguagem de programação, todas as demais linguagens de programação também podem implementar a mesma função (COPELAND, 2007). Então como pode ser melhor usar uma linguagem do que outra para implementar um determinado algoritmo, se as linguagens modernas são todas Turing-completas?

A escolha da linguagem é uma questão de estilo e da adaptação dela ao contexto do algoritmo implementado. Quanto mais próximas forem as abstrações fornecidas pelas linguagens dos algoritmos implementados com elas, menos complexas serão essas implementações (FOWLER, 2010). Isso é motivação suficiente para o projeto contínuo de novas linguagens e para o aperfeiçoamento das linguagens existentes. Para flexibilizar essas adaptações aos domínios de aplicação, as linguagens modernas têm incorporado conceitos que permitem definir nos programas não somente como será a sua execução, mas também como será sua compilação e interpretação. Isso possibilita adaptar a compilação e interpretação para atender melhor as necessidades de cada aplicação.

Os compiladores e interpretadores têm geralmente etapas de execução definidas e únicas para qualquer aplicação que será compilada ou interpretada. Essas etapas podem, em determinados casos, tornar o desenvolvimento de uma aplicação mais complexo por conta das características inerentes a elas. As linguagens que têm incorporado funcionalidades para permitir aos programas redefinir, remover e adicionar novos comportamentos às etapas de compilação e interpretação, acabam por tornar o desenvolvimento dos programas menos complexo. Isso acontece porque os algoritmos que eram amarrados a comportamentos fixos de compilação e interpretação, agora podem adaptar etapas para que sejam mais adequadas a eles.

Não é novidade permitir que etapas fixas sejam adaptadas, esse é o caminho convencional de muitos conceitos em computação. Um paralelo pode ser feito com as bibliotecas de computação gráfica (ROST et al., 2009), no início elas tinham um fluxo de execução fixo que passou a ser adaptável conforme as bibliotecas evoluíam (*Shading Language*). Com as linguagens de programação não é diferente, no início elas tinham compiladores e interpretadores com etapas fixas e com o tempo passaram a ser

mais flexíveis. Em ambos os casos foram criados mecanismos que permitiram usar as bibliotecas gráficas e as linguagens de programação para interagir com características vinculadas a elas próprias. No domínio das linguagens de programação, essa mecânica é chamada de metaprogramação e permite que as linguagens de programação definam funcionalidades a respeito de sua própria compilação, interpretação e execução.

O conceito de metaprogramação não é utilizado em um único contexto, por isso a metaprogramação não possui somente uma definição. Ela pode ser apresentada de modo diferente dependendo do contexto em que estiver sendo utilizada (DAMAŠEVIČIUS; ŠTUIKYS, 2015). Nessa pesquisa estamos interessados na definição de metaprogramação em tempo de compilação e não trataremos as demais formas de metaprogramação. A metaprogramação em compilação permite definir comportamentos não convencionais que o compilador pode executar dependendo do código-fonte que estiver compilando.

Por exemplo, em compilação usando metaprogramação, é possível adicionar uma funcionalidade ao compilador, para que toda vez que ele encontre uma função que retorna um valor de algum tipo específico, ele adicione antes desse retorno uma chamada de função que verifique se o valor retornado passa por uma validação respeitando um predicado. A chamada de função que foi adicionada pode de modo similar ser feita usando orientação a aspectos, que permite localizar pontos do código-fonte e realizar chamadas de funções nesses locais (KICZALES et al., 1997). Aplicações orientadas a aspectos usam desse tipo de implementação para separar os requisitos funcionais dos não funcionais e tornar o código-fonte melhor organizado.

Nessa pesquisa mostramos como a metaprogramação pode ser utilizada para separar requisitos não funcionais de modo diferente do que é feito usando orientação a aspectos. Ao contrário da orientação a aspectos, a metaprogramação possibilita explorar o código-fonte de forma mais abrangente, o que torna possível separar os requisitos não funcionais e verificar se o código-fonte respeita restrições impostas por esses requisitos. Essa verificação é uma tarefa que usando orientação a aspectos é bem mais difícil de realizar.

Essa pesquisa não usou orientação a aspectos em momento algum. A associação entre essa pesquisa e a orientação a aspectos está no problema que ambas resolvem. Esse problema é a separação do código-fonte dos requisitos funcionais do código-fonte dos requisitos não funcionais (GLINZ, 2007).

O requisito não funcional de redundância pode ser implementado em diferentes aplicações sem estar relacionado ao comportamento principal de cada uma delas. Uma aplicação que atenda a esse requisito se mantém disponível aos usuários mesmo que algum tipo de falha ocorra durante sua execução (CACHIN; GUERRAOUI; RODRIGUES, 2011). As aplicações não redundantes se comportam de forma diferente, uma falha

provavelmente fará com que essas aplicações se tornem indisponíveis para o usuário. Esse requisito não funcional foi selecionado para demonstrar como a metaprogramação permite separar dele o resto do código-fonte. Nesse caso prático, a metaprogramação também é usada para verificar o código-fonte procurando por restrições impostas pelo requisito de redundância da replicação.

Para implementar as aplicações redundantes foi usado o conceito de replicação. Cada aplicação replicada executa em mais de uma instância, assim se alguma instância falhar e parar de executar as outras podem manter a aplicação executando. Novas instâncias da mesma aplicação podem ser executadas a qualquer momento para recuperar uma instância que falhou. O mecanismo de replicação torna as aplicações redundantes. Ele também requer que as aplicações sejam implementadas de modo determinista. Essa restrição associada à replicação é verificada usando metaprogramação para evitar que as aplicações apresentem comportamentos não deterministas durante sua execução.

O comportamento determinista é aquele que, associado a uma condição inicial, sempre produzirá o mesmo resultado quando executado (WIESMANN et al., 2000). Caso as instâncias executem um comportamento não determinista, mesmo que elas iniciem a execução a partir da mesma condição inicial, elas poderão obter resultados diferentes, tornando-se assim inconsistentes.

A linguagem de programação foco dessa pesquisa é Cyan, por fornecer suporte à metaprogramação em tempo de compilação e permitir integrar código-fonte Java em código-fonte Cyan com facilidade (GUIMARÃES, 2018a). Em Cyan, a metaprogramação é feita em Java e consiste na implementação de classes que herdam classes do compilador de Cyan e que podem estender suas interfaces. Esses metaobjetos são compilados para serem usados pelo compilador de Cyan durante determinadas etapas da compilação das aplicações desenvolvidas em Cyan.

O *framework* Treplica foi utilizado para implementar o conceito de replicação (VIEIRA; BUZATO, 2008), ele é originalmente desenvolvido em Java e nessa pesquisa foi integrado em Cyan por meio de uma biblioteca desenvolvida em Cyan. Essa biblioteca cria uma casca para o código-fonte Java de Treplica, facilitando a implementação de aplicações de Cyan que usam Treplica. A metaprogramação foi utilizada para separar a biblioteca Treplica do restante da aplicação Cyan e para verificar se o código-fonte da aplicação possui comportamento não determinista. Caso o compilador encontre algum trecho de código-fonte não determinista, ele substituirá esse trecho por um código determinista e se ele não souber realizar essa substituição, ele retornará um erro de compilação.

O uso da metaprogramação no contexto da replicação acaba por reduzir os problemas de não determinismo inerentes a esse tipo de aplicação. Permitindo separar com maior transparência o código-fonte que implementa o mecanismo de replicação

do restante da aplicação, reduzindo assim a complexidade do desenvolvimento das aplicações replicadas.

Como resultado da pesquisa foi possível diminuir a complexidade das aplicações replicadas, devido ao uso de metaprogramação. Essa abordagem de utilizar metaprogramação para isolar o requisito não funcional de redundância e a verificação do código-fonte na busca de erros associados a esse requisito é uma solução que pode ser estendida a outros requisitos não funcionais. Isso permite que essa pesquisa possa ser generalizada para outros contextos onde seja necessário separar requisitos não funcionais e garantir que o código-fonte atenda a determinadas restrições de programação.

Nos Capítulos 1 e 2 são tratados os conceitos usados nessa pesquisa. A linguagem de programação Cyan e seus mecanismos de metaprogramação são abordados com base em exemplos práticos, e o *framework* Treplica é mostrado por meio de uma aplicação implementada em Java.

O Capítulo 3 mostra como o *framework* Treplica foi incorporado à linguagem de programação Cyan e como a metaprogramação foi utilizada para separar a replicação do restante do código-fonte da aplicação. Uma explicação sobre o problema de não determinismo também é apresentada neste capítulo. Ele termina explicando a implementação dos metaobjetos de Cyan que permitem o uso de Treplica de modo padronizado e transparente.

No Capítulo 4 são apresentados dois casos de uso de aplicações replicadas que foram implementadas usando metaprogramação. O primeiro é um jogo de tabuleiro para dois jogadores que usa replicação para permitir que cada jogador mantenha uma instância replicada do tabuleiro. O segundo exemplo trata de como os metaobjetos ajudam a evitar que os programas replicados criados sejam inconsistentes. No Apêndice A estão os códigos-fonte dos metaobjetos de Cyan desenvolvidos durante a realização dessa pesquisa.

1 Metaprogramação em Compilação

O objetivo desse capítulo e introduzir o conceito de metaprogramação. Ele inicia definindo uma linguagem de programação com elementos de metaprogramação em compilação. Essa linguagem é usada para mostrar como esse conceito funciona isoladamente de uma linguagem de programação real, que possui maior complexidade.

O capítulo segue detalhando a metaprogramação em compilação e mostrando a linguagem de programação Cyan, que é a linguagem principal dessa pesquisa. As etapas de compilação em Cyan são tratados junto com seus metaobjetos. A linguagem Xtend aparece no final do capítulo para mostrar um modelo de metaprogramação em compilação diferente do mostrado em Cyan.

1.1 Execução e Compilação de Programas

As linguagem de programação são um método padronizado para passar instruções a um computador (DERSHEM; JIPPING, 1995). Elas são regras sintáticas e semânticas usadas para definir programas de computador (FISCHER; GRODZINSKY, 1993). A semântica e a sintaxe são relacionadas à linguagem de programação por perspectivas diferentes. A semântica é relacionada ao significado da linguagem enquanto a sintaxe é associada a estruturas ou padrões formais de como a linguagem expressa esse significado (CHOMSKY, 1955).

Pela perspectiva da sintaxe uma linguagem é formada por tokens e regras. Os tokens são as palavras e os símbolos que pertencem à linguagem e as regras são a definição de como esses tokens podem ser agrupados de modo válido. Os possíveis agrupamento de tokens de uma linguagem de programação são chamados de sentenças. Dentre todas as sentenças que podem ser construídas com um conjunto de tokens, as que pertencem à linguagem são as que atendem a suas regras de agrupamento.

O conjunto de tokens da linguagem pode ser representado por expressões regulares, elas podem ser usadas para definir os tokens na maioria das linguagens de programação. As regras da sintaxe podem ser representadas usando o formalismo de Backus-Naur, que permite definir como os tokens podem ser agrupados para formar as sentenças válidas da linguagem de programação (AHO; ULLMAN, 1977).

Diferente da sintaxe, que está vinculada à forma como as sentenças são montadas, a semântica é associada ao significado das sentenças. Uma sentença pode ser sintaticamente adequada e apresentar erros de semântica. Na frase — *devido à greve dos motoristas de ônibus fiquei parado no tráfico ontem* — a sintaxe está correta, mas seu signifi-

cado lógico é inválido. O correto seria usar a palavra *tráfego* para se referir à circulação de veículos, e não a palavra *tráfico*, que se refere à venda ilícita de mercadorias.

As linguagens de programação podem conter erros de significado. Uma variável do tipo inteiro que recebe a atribuição de um texto é considerado um erro de semântica em algumas linguagens, mesmo a sintaxe da atribuição estando correta. A variável definida como inteira passa a ter uma restrição implícita de que somente as atribuições de variáveis do tipo inteiro são válidas.

Na Figura 1 é descrita uma linguagem que pode ser usada para representar operações matemáticas de multiplicação e de divisão entre números naturais. Operações matemáticas são instruções corriqueiras à maioria das linguagens de programação. Um exemplo é a soma de dois números naturais que pode ser representada de diversas formas em diferentes linguagens de programação — soma(2, 3), (2 + 3), (+ 2 3), (2 3 +) ou (2 soma 3). A notação de Backus-Naur é usada na descrição das regras dessa linguagem e os tokens são descritos usando palavras e expressões regulares.

As regras da linguagem são representadas pela construção ⟨nome da regra⟩ seguidas do símbolo ::= e de sua expansão. As demais palavras são tokens, com exceção à regra ⟨numero⟩ que se expande para um token representado por uma expressão regular. Essa expressão regular aceita qualquer número inteiro que não inicie com zero como uma sequência de caracteres válida para formar esse token.

O símbolo | separa as opções de expansão de uma regra, ⟨operação⟩ pode ser expandida tanto para o token **multiplica** como para o token **divide**. Uma dessas opções deve ser escolhida durante a expansão dessa regra, ela não pode se expandir simultaneamente para as duas opções na geração de uma mesma sentença.

⟨inicio⟩ ::= ⟨inverte⟩ **mostra** | ⟨valida⟩
⟨valida⟩ ::= **verifica** ⟨inverte⟩ **mostra**
⟨inverte⟩ ::= **troca** ⟨conta⟩ | ⟨conta⟩
⟨conta⟩ ::= ⟨numero⟩ ⟨operação⟩ ⟨subconta⟩
⟨subconta⟩ ::= (⟨conta⟩) | ⟨numero⟩
⟨operação⟩ ::= **multiplica** | **divide**
⟨numero⟩ ::= [1 - 9]?[0 - 9]+

Figura 1: Linguagem para multiplicar ou dividir e mostrar o resultado

Todas as sentenças criadas a partir dessa linguagem têm início na regra ⟨inicio⟩ e são formadas por expansões de suas regras. Vamos verificar se a sentença — **3 divide 9 mostra** — pode ser formada a partir das regras dessa linguagem. Começando

essa expansão pela regra ⟨inicio⟩ expandimos para a opção — ⟨inverte⟩ **mostra**. Essa segunda composição tem a regra ⟨inverte⟩, que é expandida para formar — ⟨conta⟩ **mostra**. Na sequência a regra ⟨conta⟩ é expandida para formar — ⟨numero⟩ ⟨operação⟩ ⟨subconta⟩ **mostra**. Esse procedimento termina quando uma expansão que só possua tokens for produzida. Se uma expansão igual a sentença — **3 divide 9 mostra** — puder ser produzida, significa que tal sentença pertence (é válida) a essa linguagem. Sentenças que não podem ser produzidas a partir da expansão dessas regras não pertencem (não são válidas) a essa linguagem.

A execução das sentenças dessa linguagem de programação pode ser realizada pelo computador *Alpha1* e o resultado dessa execução depende da semântica associada a cada sentença. O computador *Alpha1* executando uma sentença dessa linguagem que termine com a palavra-chave **mostra** irá exibir o resultado das contas realizadas em seu dispositivo de saída (monitor). Se a sentença tiver as palavras-chave **multiplica** e **divide** o computador *Alpha1* irá multiplicar ou dividir o número após as palavras-chave pelo número que precede a mesma palavra-chave. Na execução da sentença — **3 divide 9 mostra** — *Alpha1* dividiria 9 por 3 e mostraria o resultado 3 em um monitor.

O computador *Alpha1* executando uma sentença com a palavra-chave **troca** deve substituir as palavras-chave **multiplica** originais da sentença (não as trocadas) pela palavra-chave **divide** e deve substituir as palavras-chave **divide** originais da sentença (não as trocadas) pela palavra-chave **multiplica**. Isso produzirá uma nova sentença diferente da original sem a palavra-chave **troca**, a execução da sentença original deve ser encerrada e a nova sentença deve ser executada em seu lugar. No caso *Alpha1* executando a sentença — **troca 3 divide 9 mostra** — deve modificar a sentença original para — **3 multiplica 9 mostra** — e passar a executar essa nova sentença.

As sentenças com a palavra-chave **troca** possuem uma semântica não convencional se comparada às sentenças que somente possuem as palavras-chave **mostra**, **multiplica** e **divide**. As sentenças convencionais definem como um computador deve fazer para obter um resultado de sua execução, elas não se relacionam com a sintaxe da própria sentença. No caso das sentenças com a palavra-chave **troca**, a semântica associada a elas é referente à sintaxe da própria sentença, e não ao resultado que se obtém com sua execução.

As sentenças com a palavra-chave **troca** podem ser classificadas como sentenças de *metalinguagem* por se referir à própria linguagem na qual a sentença ganha um significado (DAMAŠEVIČIUS; ŠTUIKYS, 2015). O sufixo *meta* é usado para indicar uma reflexão sobre si, no caso o significado da sentença com a palavra-chave **troca** reflete sobre a própria sentença que lhe atribui um significado.

Programar em uma linguagem de programação, significa criar sentenças (programas) que pertençam a essa linguagem. Esses programas ganham significado quando

associados à semântica da linguagem a qual pertencem, fora do contexto da linguagem, ele não possui um significado associado. Então, por analogia, metaprogramação é a criação de um programa que reflete seu significado (semântica) sobre ele próprio e sobre outros programas que essa linguagem de programação permite criar.

Por razões desconhecidas o computador *Alpha1* precisou ser substituído pelo computador *Beta2*, e os programas criados usando a linguagem de programação (Figura 1) que *Alpha1* era capaz de executar não podem ser executados por *Beta2*. A linguagem de programação (Figura 2) que *Beta2* pode executar possui uma sintaxe diferente da linguagem de programação que *Alpha1* executa. Por sorte *Alpha1* pode ser adaptado para traduzir (compilar) seus programas para a linguagem que *Beta2* pode executar. Note que a semântica de uma linguagem não é uma característica do computar que a executa. A semântica é uma característica da própria linguagem, podem existir vários computadores que possam executar uma mesma linguagem.

Nessa nova linguagem de programação mostrada na Figura 2, os programas com a palavra-chave **artsom** executados por *Beta2* possui a mesma semântica dos programas com a palavra-chave **mostra** executados por *Alpha1*. O mesmo acontece nos programas com as palavras-chave **acort**, **acilpitlum** e **edivid** executados por *Beta2*, que possuem a mesma semântica dos programas com as palavras-chave **troca**, **multiplica** e **divide** executados por *Alpha1* nessa ordem.

⟨inicio⟩ ::= ⟨inverte⟩ **artsom**
⟨inverte⟩ ::= **acort** ⟨conta⟩ | ⟨conta⟩
⟨conta⟩ ::= ⟨numero⟩ ⟨operação⟩ ⟨subconta⟩
⟨subconta⟩ ::= (⟨conta⟩) | ⟨numero⟩
⟨operação⟩ ::= **acilpitlum** | **edivid**
⟨numero⟩ ::= [1 - 9]?[0 - 9]+

Figura 2: Linguagem executada por *Beta2* similar à linguagem da Figura 1

O programa — **3 divide 9 mostra**, se traduzido para a linguagem que *Beta2* executa, seria representado pelo programa — **3 edivid 9 artsom**. A compilação de programas da linguagem da Figura 1 para a linguagem da Figura 2 consiste em substituir as palavras-chave **mostra**, **troca**, **multiplica** e **divide** nessa ordem pelas palavras-chave **artsom**, **acort**, **acilpitlum** e **edivid** nessa ordem.

A palavra-chave **verifica** da linguagem executada por *Alpha1* não possui correspondência na linguagem executada por *Beta2*. A compilação no caso dessas linguagens ocorre somente da linguagem definida pela gramática da Figura 1 para a linguagem

definida pela gramática da Figura 2. Não definiremos a compilação da linguagem executada por *Beta2* para a linguagem executada por *Alpha1*.

Diferente de **troca**, que tem uma semântica associada à execução do programa, **verifica** tem uma semântica associada à compilação do programa. Durante a execução de programas com a palavra-chave **verifica** nada precisa ser feito. O computador *Alpha1* pode ignorar **verifica** e executar o restante do programa. Na execução do programa — **verifica 3 divide 9 mostra** — o resultado obtido pelo computador *Alpha1* é o mesmo resultado obtido do programa — **3 divide 9 mostra**.

Na compilação de programas com a palavra-chave **verifica** é validado se todos os números do programa são pares. Caso algum número ímpar seja encontrado, a compilação se torna inválida, um erro de compilação é produzido e o programa correspondente não é gerado. Se somente números pares forem encontrados, o restante do programa é compilado em um novo programa que pode ser executado pelo computador *Beta2*.

O programa — **verifica 2 divide 8 mostra**, após ser compilada, teria como resultado o programa — **2 edivid 8 artsom**. Em contrapartida o programa — **verifica 3 divide 9 mostra** — não seria compilado por conter números ímpares e um erro de compilação seria mostrado no monitor do computador *Alpha1*.

Os programas com as palavras-chave **verifica** e **troca** podem ser classificados como *metaprogramas*, porque sua execução ou compilação refletem sobre eles mesmos. No caso da palavras-chave **verifica** o comportamento de verificar o programa reflete sobre o próprio programa que definiu a verificação.

1.2 Linguagens Compiladas com Suporte à Metaprogramação

A escrita de programas usando metalinguagens como foi descrito é uma técnica conhecida como metaprogramação. Essa técnica permite que metaprogramas manipulem outros programas, e que programas manipulem a si próprios. Essa manipulação pode acontecer tanto durante a compilação, como durante a execução dos programas manipulados (DAMAŠEVIČIUS; ŠTUIKYS, 2015).

Metaprogramação tem um amplo significado, aqui usaremos metaprogramação para permitir, transformar e verificar um programa durante sua compilação. O programa que é modificado ou validado é chamado de programa base, e o programa que realiza a manipulação é chamado de metaprograma. O metaprograma é um conjunto de classes ou funções que permitem acessar a árvore sintática abstrata (AST) do programa base durante sua compilação. Ele funciona como uma extensão do compilador, os metaprogramas podem modificar o comportamento do compilador durante suas etapas de tradução. As etapas de análise léxica, de análise sintática, da verificação de tipos, da ge-

ração de código ou qualquer outra etapa da compilação podem ter seu comportamento personalizado para compilar o programa base de um modo diferente do convencional.

Xtend (RENTSCHLER et al., 2014), Groovy (KOENIG et al., 2007) e Cyan (GUIMARÃES, 2018a) são linguagens com suporte à metaprogramação em tempo de compilação. Essas linguagens permitem andar pela árvore sintática abstrata, obter informação a respeito das estruturas do programa base e modificar essas estruturas. Na Seção 1.3 exemplos de metaprogramação são explicados com detalhes para mostrar na prática como a aplicação dessa técnica pode funcionar.

Como a metaprogramação em Cyan é um dos pilares dessa pesquisa, os detalhes de como Cyan funciona são explicados de forma detalhada. Os tópicos a respeito de Cyan não estão limitados à metaprogramação, eles vão do básico a respeito da linguagem, passando pelo protocolo de metaobjetos, terminando por explicar como as aplicações e as bibliotecas de Cyan devem estar organizadas.

1.3 Uma Introdução à Linguagem de Programação Cyan

Cyan é uma linguagem de programação orientada a objetos com base em protótipos, tipagem estática e suporte à metaprogramação (GUIMARÃES, 2018a). Os programas escritos em Cyan são compilados para Java e depois para *bytecodes* que podem ser executados por uma máquina virtual Java. Em Cyan, diferente das linguagens tradicionais orientadas a objetos como C++ e Java, não são declaradas classes, são declarados protótipos usando a palavra reservada `object`. O protótipo de nome `Vehicle` mostrado no Código-Fonte 1 pode ser tomado como exemplo desse tipo de declaração.

```
1  package main
2
3  object Vehicle
4    var String type
5
6    func init: String type {
7      self.type = type;
8    }
9
10   // retorna a cadeia contida na variavel type
11   func getType -> String = type;
12 end
```

Código-Fonte 1: Definição de Protótipo em Cyan

Todo protótipo de Cyan deve ser declarado em um arquivo de mesmo nome e

deve definir na primeira linha desse arquivo a qual pacote o protótipo pertence. Para definir o pacote de um protótipo é usada a palavra reservada **`package`** seguida do nome do pacote. Os protótipos podem ter variáveis de instância, que são as variáveis declaradas dentro do protótipo, como a variável **`type`** no Código-Fonte 1. As variáveis de instância são declaradas usando a palavra reservada **`var`**. Se elas forem variáveis só de leitura devem ser declaradas usando a palavra reservada **`let`**.

Uma variável pode ter seu tipo indicado após as palavras reservadas **`var`** e **`let`**. No caso da variável **`type`**, **`String`** foi o tipo indicado. Dois tipos importantes em Cyan são **`Nil`** e **`Dyn`**. A palavra reservada **`Nil`** é um tipo usado para representar ausência de valor e o tipo **`Dyn`** é usado para representar qualquer valor. Variáveis do tipo **`Nil`** só podem receber o valor **`Nil`** enquanto variáveis do tipo **`Dyn`** podem receber valores de qualquer tipo e o valor **`Nil`**. As variáveis dos protótipos podem ainda ser compartilhadas entre suas instâncias, a palavra reservada **`shared`** deve ser adicionada antes de **`var`** para indicar que a variável terá uma única instância para todos os objetos do protótipo que a declarou. As variáveis que não são **`shared`** possuem uma instância para cada objeto do protótipo.

Os protótipos de Cyan também são compostos de métodos que são as construções formadas pela palavra reservada **`func`** seguida de seu nome e de seu código-fonte que fica cercado por chaves (**`{`** e **`}`**). Os métodos também podem ter parâmetros como no método **`setName:`** do Código-Fonte 2 (**`name`** é o parâmetro) e retornos como no método **`getType`** (**`-> String`** é o retorno) do Código-Fonte 1.

Os protótipos de Cyan podem ter suas variáveis de instância e métodos compartilhados através do conceito de herança. A palavra reservada **`extends`** permite a herança de um protótipo por outro. Usando herança é possível incluir os métodos e variáveis do protótipo herdado pelo protótipo que herdou e também possibilita que os métodos do protótipo herdado sejam especializados. No Código-Fonte 2 o protótipo **`Airplane`** herda o protótipo **`Vehicle`**, incluindo a variável **`type`** e o método **`getType`**. A herança de um protótipo funciona como se os métodos e variáveis do protótipo herdado tivessem sidos implementados no protótipo que definiu a herança.

```
1  package main
2
3  object Airplane extends Vehicle
4    var String name
5
6    func init {
7      super init: "flying";
8      self.name = "Legacy 650";
9    }
10   func getName -> String = name
```

```
11
12    func setName: String name {
13      self.name = name;
14    }
15  end
```

Código-Fonte 2: Herança de Protótipo em Cyan

No início da execução de uma aplicação em Cyan é criado uma instância para cada protótipo que compõe a aplicação. Essas instâncias não estão associadas a uma variável, elas são referenciadas pelo nome do protótipo. Isso permite que métodos e variáveis sejam acessados desde o início da execução, usando o nome do protótipo como mostrado na Linha 4 do Código-Fonte 3. O nome de um protótipo também aparece no código-fonte, assumindo a função de um tipo e não de um objeto, como acontece na declaração da Linha 1, que usa o nome do protótipo para definir de que tipo é a variável.

```
1    let plane = Airplane new;
2    plane setName: "Super Tucano";
3    var String str;
4    str = Airplane getName;
```

Código-Fonte 3: Nova instância de um protótipo

Para criar um novo protótipo em Cyan pode ser usado o método **`new`**. Na Linha 1 do Código-Fonte 3, é criada uma nova instância do protótipo **`Airplane`** usando esse método. A instância de **`Airplane`** referenciada pelo nome do protótipo recebe uma chamada do método **`new`** e retorna uma nova instância de **`Airplane`**. Essa nova instância é atribuída à variável **`plane`**, declarada com a palavra reservada **`let`**. A diferença entre as palavras reservadas **`var`** e **`let`** está no fato de que **`let`** só permite atribuição em sua inicialização e depois ela vira uma variável somente de leitura. Quando o tipo dessa variável não está explícito após as palavras reservadas **`var`** e **`let`**, o tipo é deduzido a partir da expressão atribuída à variável, no caso o tipo deduzido seria **`Airplane`**.

Chamadas de métodos em Cyan são realizadas por meio de mensagens similar ao que ocorre em Smalltalk. Uma mensagem é composta por palavras-chave, cada uma delas com zero ou mais parâmetros. Quando uma mensagem é enviada para um objeto, ela causa a execução de um método. Na Linha 2 do Código-Fonte 3 é realizado um envio de mensagem a um objeto do tipo **`Airplane`**, que recebe essa mensagem e executa o método **`setName:`** passando os parâmetros da mensagem.

A palavra reservada **`self`**, que aparece no método do Código-Fonte 2, é uma re-

ferência ao objeto que recebe a mensagem que dispara a execução desse método, similar a *this* usado nas linguagens C++ e Java. No método `setName:`, a palavra reservada `self` faz referência ao objeto que recebeu a mensagem que dispara sua execução.

Como resultado da execução de um método, um valor é retornado por sua chamada. Se o método não possuir um retorno definido, `Nil` é retornado. O método que possui retorno definido é declarado usando o símbolo `->` seguido pelo tipo do retorno, essa declaração deve ser feita após o nome do método ou depois de seus parâmetros, caso eles existam. No Código-Fonte 2, o método `getName` define um retorno do tipo `String`. Esse método é chamado na Linha 4 do Código-Fonte 3 e seu retorno é atribuído à variável `str`.

Em Cyan também é possível definir funções anônimas. Elas são chamadas de anônimas, pois diferentemente dos métodos, elas não tem um nome que as identifique. O Código-Fonte 4 mostra como uma função anônima é definida e usada. Na primeira linha desse código, a variável `f` é definida com o tipo `Function<String, Nil>` e recebe uma função anônima de mesmo tipo. O comportamento da função anônima atribuída a variável `f` é imprimir a mensagem que for passada como parâmetro à função. Na chamada realizada na segunda linha do exemplo será impressa a *string* `"fine"` que foi passada como parâmetro.

O tipo `Function` usado serve para representar uma função, no caso anterior o tipo `Function<String, Nil>` representa uma função que tem um parâmetro do tipo `String` e um retorno do tipo `Nil`. As funções anônimas podem ter mais parâmetros, mas o último tipo da lista será sempre o tipo de retorno da função anônima. Caso só exista um tipo na lista ele será referente ao retorno, essa lista não pode ser vazia.

```
1  var Function<String, Nil> f = { (: String msg :) msg println };
2  f eval: "fine"; // imprime "fine"
```

Código-Fonte 4: Definição e Uso de uma Função Anônima

1.3.1 Compilador de Cyan e sua Interface com os Metaobjetos

A compilação de um programa em Cyan acontece em etapas e a metaprogramação é suportada por meio de metaobjetos (*metaobjects*) usados durante essas etapas. Esses metaobjetos (GUIMARÃES, 2018b) permitem alterar o código-fonte original dos programas, eles produzem novas versões do código-fonte que são usadas no lugar da versão original. As versões de código-fonte que foram produzidas são descartadas ao final da compilação do programa.

Os metaobjetos usados durante a compilação são associados ao código-fonte por meio de anotações de Cyan. As anotações são palavras-chave precedidas pelo símbolo arroba (`@`) e podem receber parâmetros. As anotações que recebem parâmetros são declaradas no seguinte formato — `@`⟨nome da anotação⟩`(` ⟨parâmetros⟩ `)`. Já as anotações sem parâmetros seguem esse formato — `@`⟨nome da anotação⟩.

O Código-Fonte 5 é um exemplo de metaprogramação em Cyan. Nesse exemplo, o metaobjeto associado ao protótipo permite ao compilador produzir uma nova versão de **`Person`** que implementa uma versão adaptada do padrão *Observer* (GAMMA et al., 1995). O metaobjeto é associado ao protótipo por meio da anotação **`@observable`**. Durante a compilação esse metaobjeto produz uma nova versão do código-fonte de **`Person`**, que é mostrada no Código-Fonte 7. A diferença dessa nova versão em relação à anterior é que, durante sua execução as atualizações da variável **`name`** são seguidas por uma chamada à função anônima que foi atribuída à variável **`notify`** na linha 3 do Código-Fonte 6.

```
1 @observable
2 object Person
3   var String name
4 end
```

Código-Fonte 5: Protótipo marcado com a anotação **@observable**

No método **`run`** do Código-Fonte 6, a variável **`p`** recebe uma instância de **`Person`**, que é inicializada com a função anônima **`notify`**. A chamada dessa função recebe um parâmetro do tipo **`String`** e imprime seu valor. O método **`run`** é o primeiro a ser chamado quando um programa Cyan é executado. Essa versão adaptada do padrão *Observer* não notifica outros objetos, ela dispara uma função anônima que implementa o que deve ser executado quando essas variáveis de instância são atualizadas.

```
1 object Program
2   func run {
3     let notify = { (: String msg :) msg println };
4     let p = Person new: notify;
5     p setName: "Ana";
6   }
7 end
```

Código-Fonte 6: Exemplo de uso do prótotipo **Person**

Na compilação do Código-Fonte 5, o metaobjeto associado adiciona ao pro-

tótipo `Person` a variável `notify`, o construtor `init` e os métodos `set` para cada variável. A versão modificada do protótipo `Person` durante a compilação é mostrada no Código-Fonte 7.

Em Cyan os metaobjetos são implementados em Java, e não na própria linguagem Cyan. Os metaobjetos geram versões modificadas do código Cyan compilado. Como já foi mencionado, o código fonte original não é modificado, novas versões desse código-fonte são criadas temporariamente durante a compilação do programa.

```
1  object Person
2    var String name
3    Function<String, Nil> notify
4
5    func init: Function<String, Nil> f {
6      self.name = "nil";
7      self.notify = f;
8    }
9
10   func setName: String s {
11     self.name = s;
12     notify eval: s;
13   }
14 end
```

Código-Fonte 7: **Person** gerado durante a compilação

A implementação do compilador de Cyan foi realizada usando a linguagem de programação Java. Essa linguagem permite usar a programação reflexiva como uma extensão à programação orientada a objetos. A principal função reflexiva utilizada na implementação do suporte à metaprogramação foi a função que realiza a carga de arquivos do tipo *class* (classes Java compiladas) durante a execução de um programa em Java. Essa função foi usada para permitir que metaobjetos possam ser compilados separadamente da compilação do compilador de Cyan.

A compilação dos metaobjetos desenvolvidos em Java gera como resultado arquivos do tipo *class* que podem ser carregados pelo compilador de Cyan, como mostra a Figura 3. Nela, os retângulos de borda contínua representam o modo como os metaobjetos são carregados pelo compilador de Cyan para serem usados na compilação. As etapas desse carregamento têm início no retângulo **Etapa 1** e são ligadas por setas que indicam a próxima etapa do carregamento. Ao final dessas etapas o metaobjeto estará preparado para ser usado pelo compilador. Caso o metaobjeto não seja encontrado, o compilador gera um erro de compilação. O retângulo **Memória** representa os metaobjetos mantidos em memória pelo compilador durante a compilação.

Para que os metaobjetos possam ser usados pelo compilador, suas classes devem ser implementadas na linguagem de programação Java usando as interfaces definidas pelo protocolo de metaobjetos de Cyan (*PMO*). O compilador de Cyan usa objetos instanciados de classes que implementam essas interfaces em diferentes etapas da compilação. As classes e interfaces do *PMO* estão localizadas no pacote `meta` do compilador de Cyan.

Os metaobjetos são usados quando o compilador encontra alguma anotação de Cyan (exemplo `@observable`, Código-Fonte 5) no programa que está sendo compilado. As anotações são uma referência direta a qual metaobjeto (arquivo do tipo *class*) o compilador precisa usar durante a compilação do código fonte anotado.

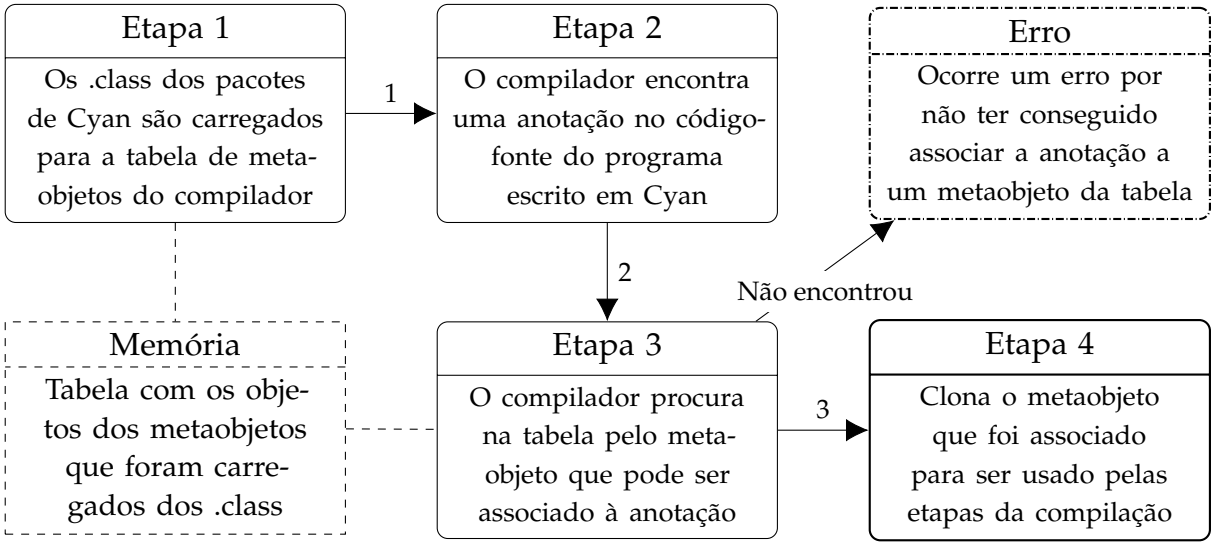

Figura 3: Carregando metaobjeto de uma anotação de Cyan pelo compilador

As etapas da compilação de Cyan e as interfaces do *PMO* são mostradas nas Figuras 4, 5 e 6. As linhas tracejadas são as etapas da compilação, elas indicam o sentido da execução por meio de setas. A compilação inicia na etapa **Parse (1)** da Figura 4 e termina na etapa **Generate (10)** da Figura 6. A etapa **Parse (4)** liga a Figura 4 com a Figura 5 e a etapa **Parse (7)** liga a Figura 5 com a Figura 6. As etapas que utilizam interfaces Java do *PMO* são ligadas a caixas de borda tracejada que listam os nomes dessas interfaces.

As entradas e saídas de cada etapa da compilação são representadas por retângulos de borda contínua. O retângulo de onde a seta de uma etapa inicia é a entrada que a etapa recebe e o retângulo para onde a seta de uma etapa aponta são as saídas que a etapa gera. No caso da etapa **Parse (1)**, a entrada é o código-fonte **Code-A** e a saída (resultado da etapa) são o código-fonte **Code-B** e uma **AST** do compilador de Cyan.

Em algumas etapas da compilação os tipos associados às expressões da AST podem não estar resolvidos. Isso significa que os tipos dos objetos da AST só estão

disponíveis para consulta pelos metaobjetos em determinadas etapas da compilação. Nessas figuras, o termo **AST** se refere à árvore sintática abstrata sem informações a respeito dos tipos de seus objetos. Se o termo **AST** for seguido de **/TI** significa que os tipos dos objetos foram resolvidos parcialmente. E caso seja seguido de **/TI/In** significa que os tipos restantes também foram resolvidos. A verificação dos tipos associados a objetos da **AST** acontece diversas vezes durante a compilação.

Os tipos que podem ser consultados nas etapas que recebem **/TI** são todos os encontrados fora do corpo dos métodos. Como os tipos das variáveis de instância, os tipos dos protótipos herdados, os tipos dos parâmetros e retornos dos métodos, entre outros. Os tipos internos aos métodos só podem ser consultados nas etapas que receberam **/TI/In** como entrada.

Ao decorrer da compilação novas versões do código-fonte podem ser geradas. Nessas figuras o termo **Code-A** se refere ao código-fonte original, e cada nova versão gerada é representada por **Code-**⟨próxima letra do alfabeto⟩. A nova versão de **Code-A** é **Code-B** e a próxima **Code-C**. Na etapa **Parse (1)** uma nova versão do código-fonte pode ser gerada, então antes dessa etapa temos **Code-A**, e depois dela **Code-B**. Enquanto na etapa **Type Interfaces (5)**, o código-fonte se mantém na versão **Code-D**.

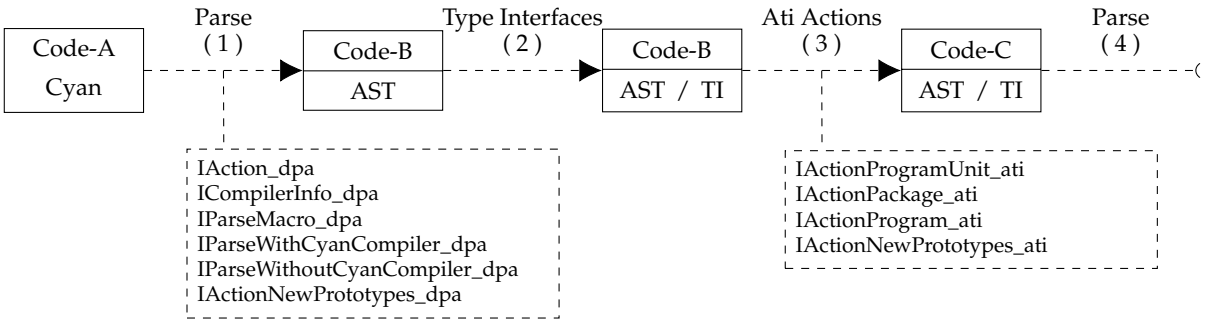

Figura 4: Etapas 1 a 3 da compilação em Cyan

As etapas **Parse (1)**, **(4)** e **(7)** recebem de entrada uma versão do código-fonte em Cyan e realizam a análise sintática desse código-fonte. Ao final dessas etapas é retornado uma **AST**, e nas etapas **Parse (1)** e **(4)** é gerada uma nova versão do código-fonte. Nas etapas **Type Interfaces (2)**, **(5)** e **(8)** são resolvidos parcialmente os tipos dos objetos da **AST** e o código-fonte pode ser verificado. Os tipos de tudo que se encontra fora do corpo dos métodos é resolvido, como os tipos das variáveis de instância, os tipos dos protótipos herdados, os tipos dos parâmetros e retornos dos métodos, entre outros.

A etapa **Ati Actions (3)** só acontece uma vez durante a compilação. Nessa etapa é permitido verificar e inserir código-fonte nos protótipos existentes. Também é possível renomear métodos e criar novas variáveis de instância, métodos e protótipos. Essas alterações acontecem ao final dessa etapa quando uma nova versão do código-fonte

é gerada.

Os tipos internos dos métodos são resolvidos nas etapas **Calc. Internal Types (6)** e **(9)**. Na etapa **Calc. Internal Types (6)** é possível inserir código-fonte dentro dos métodos e criar novos protótipos. A vantagem em usar essa etapa e não a etapa anterior para realizar essas tarefas, está no fato de que aqui existem informações a respeito do código-fonte que não estavam disponíveis na etapa anterior. Como os tipos das variáveis e dos tipos das expressões locais ao método.

A etapa final da compilação é a **Generate (10)**. Ela é responsável por transformar a **AST/TI/In** da última versão do código-fonte Cyan em código-fonte Java. Essa etapa e as etapas **Type Interfaces (2)**, **Type Interfaces (5)**, **Parse (7)**, **Type Interfaces (8)** e **Calc. Internal Types (9)** não geram novas versões do código-fonte.

Em cada etapa o compilador verifica se o metaobjeto associado a uma anotação implementa uma interface associada à essa etapa. Caso implemente, ele chama os métodos dessas interfaces referentes à etapa. Nas etapas **Parse (1)** e **Parse (4)**, se um método for inserir código-fonte, ele insere esse código-fonte após a anotação e prossegue com a compilação a partir do código-fonte inserido. Já nas etapas **Ati Actions (3)** e **Calc. Internal Types (6)** o código-fonte é inserido após a etapa terminar. A etapa **Type Interfaces (8)** é similar à etapa **Ati Actions (3)**, que como as etapas **Calc. Internal Types (6)** e **Calc. Internal Types (9)**, só permitem realizar conferências do código-fonte, não permitem inserções como as etapas anteriores.

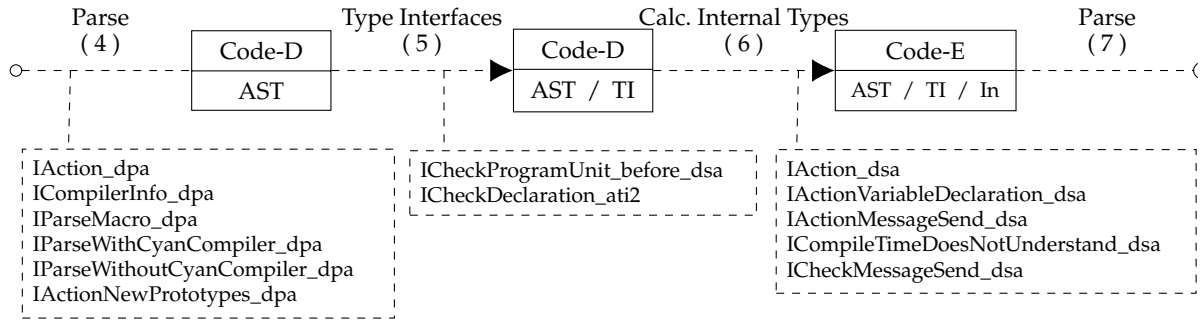

Figura 5: Etapas 4 a 6 da compilação em Cyan

A interface `IAction_dpa` está associada às etapas **Parse (1)** e **(4)**, conforme mostrado no retângulo pontilhado a esquerda da Figura 4. Essa interface permite adicionar código-fonte após a anotação relacionada ao metaobjeto que implementou essa interface. Quando o compilador encontra uma anotação dessas, ele chama o método `dpa_codeToAdd` da interface `IAction_dpa` e adiciona o código-fonte retornado por esse método após a anotação encontrada. A compilação prossegue a partir do código-fonte inserido por esse método.

Usando metaobjetos de Cyan é possível definir Linguagens Específicas de Do-

mínio (DSL). O metaobjeto `@concepts` e tem o seguinte formato `@concepts{*` ⟨código-fonte da DSL⟩ `*}` e permite que uma DSL seja acoplada a essa anotação. Ainda nas etapas **Parse (1)** e **(4)** é possível realizar a análise (*parse*) das DSLs por meio das interfaces `IParseWithCyanCompiler_dpa` e `IParseWithoutCyanCompiler_dpa`.

Nas etapas **Parse (1)** e **(4)** ainda temos a interface `IActionNewPrototypes_dpa`, que permite criar novos protótipos de Cyan. Outra interface que permite criar novos protótipos de Cyan é a interface `IActionNewPrototypes_ati`, associada à etapa **Ati Actions (3)** da compilação.

Passando à etapa **Ati Actions (3)**, temos a interface `IActionProgramUnit_ati`, que permite renomear métodos, criar novos protótipos, criar variáveis de instância, criar variáveis compartilhadas e criar novos métodos. Nessa mesma etapa as interfaces `IActionPackage_ati` e `IActionProgram_ati` permitem adicionar código-fonte a protótipos de um pacote e a todos os protótipos do programa respectivamente.

A interface `IAction_dsa` da etapa **Calc. Internal Types (6)** permite adicionar código-fonte após as anotações associadas aos metaobjetos e criar novos protótipos. Nessa etapa temos também a interface `IActionVariableDeclaration_dsa`, que permite adicionar código-fonte após as declarações de variáveis locais. E temos a interface `IActionMessageSend_dsa` usada para interceptar mensagens para o método anotado. Nessa linha de interceptar mensagens para métodos é possível usar a interface `ICheckMessageSend_dsa` para validar essas mensagens.

Nas etapas **Type Interfaces (5)**, **(8)** e **Calc. Internal Types (9)** o *PMO* fornece diversas interfaces que permitem verificar a AST na procura de inconsistências. Caso alguma inconsistência seja encontrada, poderá ser gerado um erro de compilação. Para percorrer a AST é possível usar o padrão de projeto *Visitor* suportado pelo compilador.

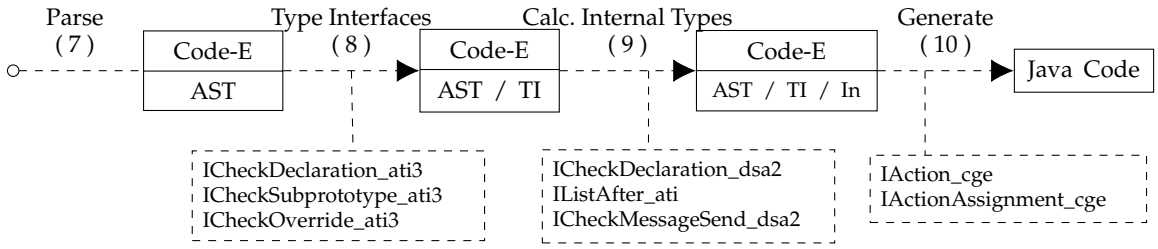

Figura 6: Etapas 7 a 10 da compilação em Cyan

Não se pode deixar de comentar sobre os parâmetros do tipo `ICompiler_ati` de métodos das interfaces e classes do *PMO*, como no método `ati_renameMethod` da interface `IActionProgramUnit_ati`. As variáveis desse tipo fornecem ao metaobjeto um acesso restrito aos métodos do compilador de Cyan. Esse acesso aos objetos do compilador permite que os métodos implementados procurem por estruturas do código-

fonte, emitam erros de compilação e recuperem informações a respeito do processo de compilação. De modo simplificado, os parâmetros do tipo `ICompiler_ati` permitem que os métodos do *PMO* acessem o compilador de Cyan.

1.3.2 Criação de Programas e Pacotes em Cyan

Programas e pacote em Cyan são organizados a partir de uma pasta raiz que leva o nome do programa ou do pacote. Nos programas dentro da pasta raiz deve existir o código-fonte do programa e o arquivo *proj.pyan*, que indica quais pacotes esse programa utiliza. Nos pacotes o código-fonte deve ser colocado direto na pasta raiz e o arquivo *proj.pyan* não deve ser adicionado, pois ele é exclusivo dos programas de Cyan.

O arquivo *proj.pyan* define quais pacotes um programa está usando. O estudo de caso do Capítulo 4 usa os pacotes `treplica` e `cyan.math` como mostrado no Código-Fonte 8, que mostra o arquivo *proj.pyan* do estudo de caso. O pacote padrão de Cyan (`cyan.lang`) não precisa ser adicionada, ele é incluído por definição pelo compilador.

```
1 program
2
3   package treplica at "lib/treplica"
4   package cyan.math at "lib/cyan"
```

Código-Fonte 8: Arquivo *proj.pyan* do estudo de caso

No desenvolvimento de programas e pacotes em Cyan é possível usar trechos de código fonte em Java diretamente. Para evitar misturar código-fonte Java em aplicações de Cyan é possível desenvolver pacotes que funcionam como casca para o código-fonte Java, que será incorporado em Cyan. Essa pesquisa utiliza uma versão de Cyan que ainda não permite importar pacotes e classes de Java na mesma sintaxe usadas para Cyan.

O uso de código Java em Cyan é feito por um metaobjeto que está associado à anotação `@javacode`. O código Java implementado dessa forma não será traduzido pelo compilador. Na fase **Generate (10)** de geração de código em Java, o conteúdo associado a cada anotação será gerada sem modificações. Isto é, a anotação do Código-Fonte 9 irá gerar na fase **Generate (10)** exatamente o que está entre os símbolos <<< e >>>. Cyan permite que outras sequências de símbolos sejam usadas, nessa pesquisa usamos a sequência de símbolos <<< e >>>. O Código-Fonte 9 mostra como o código-fonte Java pode ser incorporado em Cyan usando esse metaobjeto.

```
1 @javacode<<< import java.io.Serializable; >>>
```

Código-Fonte 9: Uso de Código Java em Cyan

1.4 Xtend e suas Anotações Ativas

Esta seção apresenta a linguagem de programação Xtend e mostra como suas anotações ativas funcionam. Essa linguagem não foi usada no desenvolvimento prático desse trabalho, mas ela é uma forma alternativa de demonstrar o conceito de metaprogramação em compilação.

Como não existe uma definição única de como deveria ser a metaprogramação em compilação, cada linguagem que a suporta apresenta uma definição adaptada a sua implementação. O conceito de metaprogramação em compilação entre essas linguagens é similar, mas a aplicação do conceito é característica de cada uma delas.

Xtend (RENTSCHLER et al., 2014) é uma linguagem de programação similar à linguagem Java, com melhorias em diversos aspectos. A compilação de código-fonte em Xtend produz código-fonte em Java e a metaprogramação é suportada por meio de suas anotações ativas (*active annotations*), que são uma evolução das anotações (*annotations*) do Java. As anotações ativas permitem que sejam desenvolvidos programas que acrescentam funcionalidades à compilação de aplicações em Xtend. Essas anotações ativas são equivalentes às anotações de Cyan.

A implementação de características e comportamentos das aplicações que usam anotações ativas passa a ser realizada também pelo compilador e não somente pelos desenvolvedores (RENTSCHLER et al., 2014). Essas anotações ativas interferem na compilação da aplicação realizando transformações que alteram o código-fonte produzido pelo compilador.

As anotações ativas aparecem no código-fonte de aplicações em Xtend como as anotações em Java. Elas são formadas pelo nome da anotação precedidas pelo símbolo arroba (`@`). No intuito de facilitar a comparação entre os exemplos de Java e Xtend, vamos iniciar o exemplo em Xtend por um trecho de código que utiliza o padrão *Observer*.

O compilador de Xtend cria uma estrutura intermediária para representar em memória o código-fonte Java. Para preservar essa associação ao longo do texto, vamos chamar essa estrutura de Estrutura Java. A Figura 7 mostra as quatro etapas da compilação considerando a Estrutura Java. As etapas na figura são representadas pelas setas e as entradas e saídas de cada etapa são representadas pelas caixas.

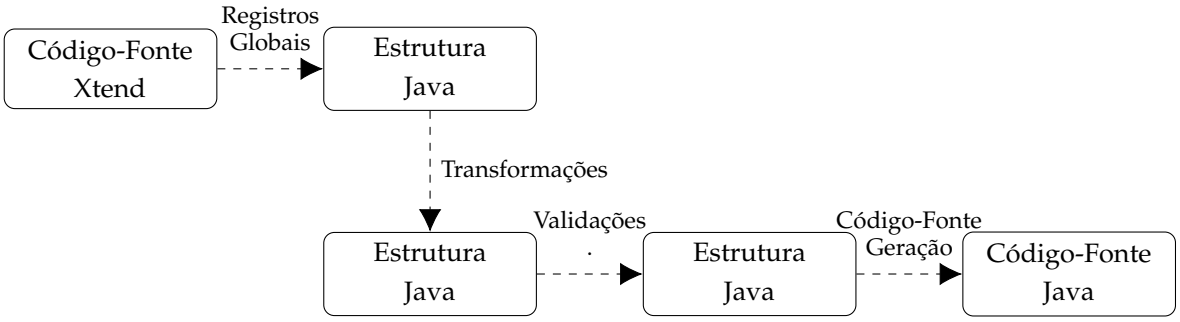

Figura 7: Etapas da compilação em Xtend

Na primeira etapa (*Registros Globais*) é possível criar as classes e interfaces Java diretamente na Estrutura Java. Em seguida, durante a etapa de transformação é possível adicionar, remover e alterar o conteúdo interno das classes e interfaces pertencentes à Estrutura Java.

A terceira etapa da compilação consiste em validar as definições e os usos dos tipos Java presentes na Estrutura Java. A quarta e última etapa permite intervir na forma como código-fonte Java será gerado a partir da Estrutura Java. Todas as etapas de compilação podem utilizar de anotações ativas para manipular a Estrutura Java.

Os Códigos-Fonte 10 a 13 mostram o programa em Java que implementa o padrão de projeto *Observer*. Esse programa será comparado a um programa implementado em Xtend (BLEWITT; BUNDY; STARK, 2005) (MIAO; SIEK, 2014) para mostrar como as anotações ativas funcionam. Os Códigos-Fonte 14 e 15 mostram a aplicação implementada em Xtend. O padrão *Observer* utiliza objetos associados a duas classes, uma dependente e outra mantenedora. Ao decorrer da execução dessas aplicações, os objetos da classe dependente são alertados quando os objetos das classe mantenedora são atualizados.

```
1 public interface Element {
2   public void addObserver(Observer obs);
3   public void notifiesObservers();
4 }
5
6 public interface Observer {
7   public void update(Object obj);
8 }
```

Código-Fonte 10: Interfaces usadas para implementar o padrão *Observer* em Java

O Código-Fonte 10 mostra duas interfaces: **`Element`**, que corresponde à interface que deve ser implementada pela classe mantenedora e **`Observer`**, que define a implementação que deve ser atendida pelas classes dependentes. A interface

`Element` tem o método `addObserver` usado para registrar dependentes e o método `notifiesObservers` usado para informar os dependentes que o objeto da classe mantenedora foi atualizado. Na interface `Observer` existe apenas o método `update`, que implementa o comportamento associado ao objeto da classe dependente após uma atualização do objeto da classe mantenedora. Esse método é chamado na execução do método `notifiesObservers`.

```
1  public class Reader implements Observer {
2    private int id;
3
4    public Reader(int id) {
5      this.id = id;
6    }
7
8    public void update(Object obj) {
9      System.out.println(
10     String.format("%d -> %s", this.id, (String)obj));
11   }
12 }
```

Código-Fonte 11: Exemplo de implementação da interface **Observer**

O Código-Fonte 11 mostra a implementação de uma classe dependente, ela implementa a interface `Observer`. Nessa classe, o método `update` é reescrito para exibir uma mensagem toda vez que for chamado. O objeto da classe dependente é registrado pelo objeto de uma classe mantenedora, como mostrado no Código-Fonte 12. O objeto da classe mantenedora só pode registrar objetos de classes que implementem a interface `Observer`, pois ela espera que a classe registrada possua o método `update` descrito pela interface `Observer`.

```
1  public class Main {
2    public static void main(String args []){
3      Board board = new Board();
4      Reader reader = new Reader(1);
5      board.addObserver(reader);
6      board.setText("Text 1");
7    }
8  }
```

Código-Fonte 12: Exemplo de uso das classes implementadas em Java

No Código-Fonte 13 temos a implementação de uma classe mantenedora, ela im-

plementa a interface `Element`, respeitando o comportamento dos métodos `addObserver` e `notifiesObservers`. A desvantagem do uso de uma interface é a necessidade de implementar esses métodos sempre que `Element` for usada, uma vez que essa implementação será similar na maioria das vezes que for realizada.

O Código-Fonte 12 mostra como os objetos dos tipos `Reader` e `Board` interagem. No método `main` é construído um objeto do tipo `Reader` e um objeto do tipo `Board`, que recebe `reader` como seu dependente. Quando o objeto `board` é atualizado, o objeto `reader` é notificado.

```
1  public class Board implements Element {
2    private List <Observer> observers;
3    private String text;
4
5    public void setText(String text) {
6      this.text = text;
7      notifiesObservers();
8    }
9
10   public void addObserver(Observer obs) {
11     this.observers.add(obs);
12   }
13
14   public void notifiesObservers() {
15     for(Observer obs : this.observers){
16       obs.update(this.text);
17     }
18   }
19 }
```

Código-Fonte 13: Exemplo de implementação da interface **Element**

O Código-Fonte 14 mostra a classe `ObservableBean`, que é equivalente à classe mantenedora do exemplo em Java. Uma classe dependente não é necessária em Xtend, pois podemos usar uma função anônima para substituir o método `update` dessa classe. Essa função anônima é passada como parâmetro a `addPropertyChangeListener` para que ela seja atribuída ao objeto `ObservableBean`. Na segunda etapa da compilação (Figura 7), o método `addPropertyChangeListener` é adicionado à classe `ObservableBean` pela anotação ativa `@Observable`.

As funções anônimas em Xtend são declaradas entre colchetes (`[` e `]`) e seus parâmetros são separados do corpo da função por uma barra vertical (`|`). O retorno de uma função anônima é a última expressão que ela define. Uma função anônima tem o seguinte formato, `[` ⟨parâmetros⟩ `|` ⟨corpo⟩ ⟨retorno⟩`]`.

Durante a compilação, as estruturas marcadas com anotações podem ser alteradas, como `ObservableBean`, que está marcada com a anotação `@Observable`. A vantagem dessa metaprogramação é reduzir o código fonte e possibilitar o reuso. Estando implementada a anotação ativa que define o padrão *Observer*, diferentes classes podem ser anotadas, não sendo mais necessário escrever os métodos `addObserver` e `notifiesObservers` inúmeras vezes.

Em Xtend o operador `=>` pode ser usado após a criação de um objeto para chamar métodos da nova instância, como no Código-Fonte 14, em que o método `addPropertyChangeListener` é chamado passando como parâmetro a função anônima — `[println("'property «propertyName» equal «newValue»"')]` — que Xtend compila como sendo uma chamada à `addPropertyChangeListener([...])`. Ele também permite a construção — ⟨nome da variável⟩ `=` ⟨valor atribuído⟩ — que Xtend modifica para a uma chamada ao método `setName("Max")` dessa nova instância durante a compilação.

Na classe `ObservableExample` um objeto de `ObservableBean` é criado na Linha 8 do Código-Fonte 14. E após a criação do objeto é enviada uma mensagem `addPropertyChangeListener` para ele. Na sequência, o atributo `name` é atualizado (uma mensagem `setName` é enviada para o objeto) disparando a função anônima que foi passada para `addPropertyChangeListener`. Nesse exemplo usamos a função anônima passada como parâmetro a `addPropertyChangeListener`, ao invés de declarar o método `update`. Isso permite definir o comportamento de cada instância da classe `ObservableBean` em sua inicialização.

```
1  @Observable
2  class ObservableBean {
3    String name
4  }
5
6  class ObservableExample {
7    def static void main(String[] args) {
8      new ObservableBean => [
9        addPropertyChangeListener [
10         println('''property <<propertyName>> equal <<newValue>>''')
11       ]
12       name = "Max"
13     ]
14   }
15 }
```

Código-Fonte 14: Exemplo de uso em Xtend

Para que possamos analisar o comportamento desse programa diretamente, vamos entender como a anotação ativa `@Observable` modifica a classe `ObservableBean` e qual código Java é obtido como resultado dessa compilação. No Código-Fonte 15 temos o resultado do código modificado de `ObservableBean`. Esse resultado é uma classe em Java que mantém o atributo `name` e recebe a adição de três métodos e de um atributo privado. Os métodos adicionados `getName` e `setName` permitem o acesso ao atributo privado `name` e o método `setName` também alertam os dependentes de que esse atributo foi atualizado. Esses métodos são seguidos do atributo `change` e do método `addPropertyChangeListener`, que é responsável por registrar a função anônima no objeto `ObservableBean`.

No Código-Fonte 15, a variável de instância do tipo `PropertyChangeSupport` é usada para implementar o padrão *Observer*. Os objetos dessa classe recebem uma função anônima (representado por um objeto da classe `PropertyChangeListener`) e associam sua chamada a mudanças de outro objeto. Nessa implementação o método `addPropertyChangeListener` registra essa função anônima. A chamada dessa função anônima ocorre quando o método `firePropertyChange` for chamado. Durante a execução quando o método `setName` é chamado, ele chama o método `firePropertyChange`, que chama a função anônima que foi registrada por `addPropertyChangeListener`.

No Código-Fonte 14 na Linha 10, os parâmetros recebidos pela chamada do método `firePropertyChange` são usados na função anônima registrada pelo método `addPropertyChangeListener`. O primeiro parâmetro (`"name"`) é representado na função anônima por `«propertyName»` e o parâmetro (`name`) é representado na mesma função anônima por `«newValue»`.

```
1  public class ObservableBean {
2    private String name;
3
4    public String getName() {
5      return this.name;
6    }
7
8    public void setName(final String name) {
9      change.firePropertyChange("name", this.name, name);
10     this.name = name;
11   }
12
13   private PropertyChangeSupport change =
14     new PropertyChangeSupport(this);
15
16   public void addPropertyChangeListener(
17   final PropertyChangeListener listener) {
18     this.change.addPropertyChangeListener(listener);
```

```
19     }
20 }
```

Código-Fonte 15: Classe Java **ObservableBean** gerada pelo compilador Xtend

A anotação ativa `@Observable` que gera a classe do Código-Fonte 15 está dividida em duas partes. A primeira é responsável por escrever os métodos de acesso a todos os atributos da classe anotada com `@Observable`, e a segunda parte é responsável por gerar o atributo `change` e o método `addPropertyChangeListener`.

2 Replicação de Processos

Replicação é mais um conceito chave dessa pesquisa. Esse capítulo inicia mostrando como a replicação funciona, ela é explicada de modo simplificado por meio de um exemplo prático. O capítulo segue apresentando o *framework* Treplica e suas classes, usados ao decorrer do restante da pesquisa. O *framework* OpenReplica é mostrado na sequência, ele é comparado ao *framework* Treplica. O capítulo encerra explicando coesão e acoplamento, que são conceitos que motivam o uso de metaprogramação para desenvolver programas replicados.

2.1 Ações e Estados das Replicas

Alguns programas podem ter sua execução representada por um conjunto de estados e um conjunto de transições. Esses estados representam a situação do programa em um determinado instante de sua execução e as transições representam as ações realizadas pelo programa para ir de um estado para outro.

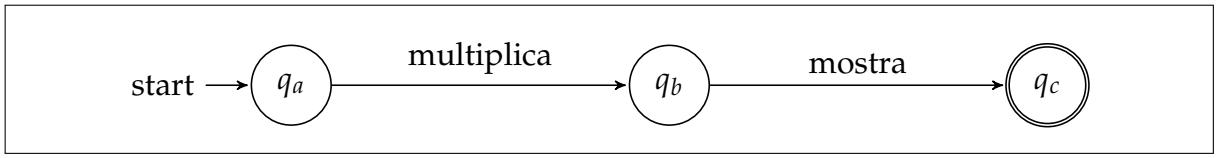

Figura 8: Execução do programa que multiplica seis por dois

O programa — **6 multiplica 2 mostra** — criado usando a linguagem da Figura 1 pode ter sua execução representada por três estados e duas transições, como mostrado na Figura 8. A execução desse programa tem início no estado q_a. Nesse estado o programa tem os valores seis e dois armazenados em uma memória reservada fornecida pelo computador. Na transição para o estado q_b os valores são multiplicados e o resultado doze também é armazenado nessa memória. Então quando o programa está no estado q_b ele tem os números seis, dois e doze armazenados na memória reservada. Na transição para q_c o resultado é mostrado e todos os números mantidos pelo programa são liberados.

Durante a execução desse programa, o resultado só é mostrado na transição do estado q_b para q_c. Caso o computador interrompa a execução no momento em que o programa está no estado q_b o resultado não será mostrado. Em situações críticas em que não se pode admitir falhas, não é possível depender da execução de uma única instância do programa para obter um resultado. Nessas situações é mais adequado executar várias instâncias de um mesmo programa, de modo que essas instâncias colaborem

umas com as outras para garantir que pelo menos uma delas irá terminar a execução, mesmo que alguma falha ocorra (CACHIN; GUERRAOUI; RODRIGUES, 2011). Essa técnica de executar um programa usando várias instâncias é chamada de replicação.

A replicação garante que cada vez que uma instância ou réplica realiza uma transição, as outras réplicas também realizam a mesma transição, de modo que em todas as instâncias a ordem das transições realizadas seja a mesma. Uma réplica que parou de funcionar pode ser recuperada pois as demais instâncias sabem quais transições foram executadas e qual a ordem dessas execuções. Então, para recuperar a réplica que falhou, basta criar uma nova instância e fazer com que ela execute todas as transições que as demais réplicas já tenham realizado.

O programa de multiplicação se torna capaz de tolerar falhas ao ser executado com várias réplicas. Isso acontece porque mesmo que uma réplica pare de executar no estado q_b, uma nova réplica pode ser criada e atualizada com as transições de uma outra instância que esteja executando normalmente. Essa nova réplica substitui a que falhou, e a execução prossegue até que as réplicas cheguem ao estado q_c e o programa termine. A replicação não altera o comportamento do programa de multiplicação, ele continua recebendo dois números e mostrando o resultado da multiplicação desses valores. Por isso é desejável que a escrita do programa — **6 multiplica 2 mostra** — seja modificada o menos possível (VIEIRA; BUZATO, 2010), visto que a funcionalidade original do programa não sofreu alterações.

Esse tipo de replicação pode ser definido de duas formas: replicação passiva ou replicação ativa. No modelo passivo existe uma réplica (mestre) que realiza a escrita dos dados e a propagação dessa mudança para as demais cópias (escravos), enquanto que a leitura dos dados pode ser feita a partir de qualquer réplica. No modelo ativo todas as réplicas modificam (escrevem) os dados de modo simultâneo e a leitura pode ser feita a partir de qualquer réplica (SCHNEIDER, 1990). Nessa arquitetura é necessário um protocolo para garantir a ordem na realização das operações.

Em ambos os modelos, as operações aplicadas às réplicas devem ser deterministas (VIEIRA; BUZATO, 2010). Não importa quando e onde elas sejam aplicadas às réplicas, as mudanças devem fornecer sempre o mesmo resultado. Um exemplo de operação não determinista é determinar a hora atual de modo convencional. Implementada de forma ingênua, essa operação retorna um valor diferente todas as vezes que for executada por uma réplica da aplicação.

2.2 Introdução ao Framework Treplica em Linguagem Java

Treplica é um *framework* que permite desenvolver aplicações replicadas em Java, implementando replicação ativa baseada em máquina de estados similar ao exemplo da

Figura 8. O desenvolvimento de aplicações com Treplica deve seguir algumas etapas. Primeiro, é definida a classe que contém os dados que são replicados. O Código-Fonte 16 mostra a classe `Info` que contém os dados a serem replicados, eles são dois atributos privados, um `int` e uma `String`. Esses atributos guardam os valores modificados pela ação que será discutida em breve. A classe `Info` também possui dois métodos que são usados para atribuir valores aos atributos privados de `Info`.

```
1  public class Info implements Serializable {
2
3    private int number;
4    private String text;
5
6    void setNumber(int number) {
7      this.number = number;
8    }
9
10   void setText(String text) {
11     this.text = text;
12   }
13
14   String getText() {
15     return this.text;
16   }
17
18   int getNumber() {
19     return this.number;
20   }
21 }
```

Código-Fonte 16: Exemplo de classe que contém os dados que são replicados

Após a definição da classe que contém os dados replicados é necessário que uma ação seja definida. Uma ação em Treplica é um termo usado para se referir a transição que modifica um estado replicado. A interface `Action` define a interface de uma ação responsável por alterar os valores da classe sendo replicada. Essa alteração de valores é feita pelo método `executeOn` da interface `Action` que deve ser implementado. O Código-Fonte 17 mostra a classe `UpdateAction`, que implementa `Action` e funciona como uma casca para a chamada das funções `setNumber` e `setText`, contendo os parâmetros necessários para efetuar a chamada. Quando um objeto de `UpdateAction` é passado para a máquina de estados do Treplica, ela envia uma cópia desse objeto às demais réplicas e todas chamam o método `executeOn`, passando como parâmetro a cópia local do objeto `Info`. Assim todas as cópias ficam com objetos `Info` contendo os mesmos valores.

```
protected static class UpdateAction implements Action, Serializable {
  private int updateNumber;
  private String updateText;

  public UpdateAction(int number, String text) {
    this.updateNumber = number;
    this.updateText = text;
  }

  public Object executeOn(Object states) {
    Info info = (Info) states;
    info.setNumber(updateNumber);
    info.setText(updateText);
    return null;
  }
}
```

Código-Fonte 17: Exemplo de classe que altera os valores replicados

Treplica deve ser inicializado antes que as ações possam ser executadas, isso pode ser feito pelo próprio método **`main`** da aplicação. O Código-Fonte 18 mostra a inicialização de Treplica e mostra como uma ação é executada. O estado a ser replicado é mantido sob guarda de Treplica, em um objeto **`StateMachine`**. Esse objeto é criado e inicializado pelo método **`createPaxosSM`** que recebe, entre outros argumentos, o estado inicial do objeto a ser replicado. A chamada ao método **`execute`** passando como parâmetro um objeto **`UpdateAction`** é equivalente a uma chamada aos métodos **`setNumber`** e **`setText`** do objeto replicado (Código-Fonte 17). Esse método de Treplica ordena, encaminha e aplica a ação na instância local e nas demais réplicas para que sejam atualizadas.

```
public class App {
  public static StateMachine states;

  public static void main(String[] args) throws Exception {
    Info info = new Info();
    states = StateMachine.createPaxosSM(
      (Serializable)info, 200, 2, false, "/var/tmp/app" + args[0]);
    states.execute(new UpdateAction(2, "text"));
  }
}
```

Código-Fonte 18: Exemplo de classe que prepara o Treplica e executa uma ação

O diagrama de sequência da Figura 9 mostra o fluxo de execução dessa aplicação.

As réplicas **A** e **B** iniciam a execução pelo método `main` da classe `App`. O contexto da aplicação (`Info`) e a máquina de estados de Treplica (`StateMachine`) têm seus objetos instanciados por esse método. Em seguida a **Réplica A** chama o método `execute` de Treplica que irá executar a ação `UpdateAction`. Treplica replica essa chamada para a **Réplica B** e executa a ação replicada em ambas as réplicas (`executeOn`).

A execução das ações em uma aplicação replicada pode ocorrer de duas formas: a réplica que chama uma ação é a mesma réplica que a executa, ou a réplica que executa a ação a recebeu de outra réplica. A ação executada pela **Réplica A** foi chamada pela própria réplica **Réplica A**. Enquanto a ação executada pela **Réplica B** foi chamada pela réplica **Réplica A**. Replicar uma ação significa serializar o objeto que a representa e enviar esse objeto às outras réplicas. Nesse exemplo o objeto do tipo `UpdateAction` é serializado com os parâmetros que recebeu quando foi instanciado, nesse caso o número `2` e a *string* `"text"`. Ele é enviado (replicado) às outras réplicas e cada uma chama o método `executeOn` de sua cópia desse objeto.

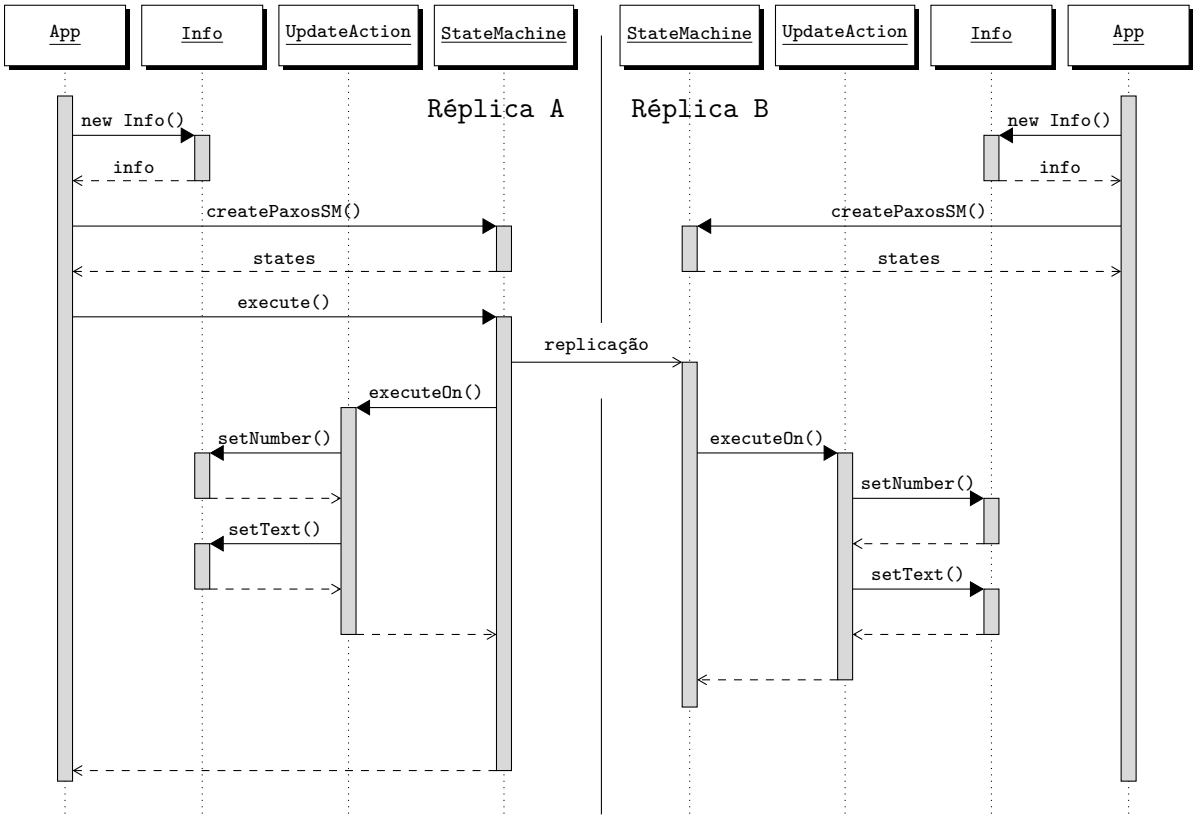

Figura 9: Execução do exemplo de Treplica em Java

2.3 Replicação Usando OpenReplica

Esta seção mostra como aplicações replicadas podem ser implementadas usando o *framework* OpenReplica. Ele não foi usado no desenvolvimento prático desse trabalho,

mas OpenReplica é uma forma alternativa de demonstrar o conceito de replicação ativa. A comparação da replicação em Treplica e a replicação em OpenReplica é feita quase que diretamente, devido à grande similaridade entre as interfaces desses *frameworks*.

OpenReplica é um *framework* implementado na linguagem de programação Python, que permite o desenvolvimento de aplicações distribuídas (ALTINBÜKEN; SIRER, 2012). Esse *framework* foi projetado com base no conceito da replicação ativa. As aplicações replicadas que usam OpenReplica são separadas em duas partes; uma que gerencia o conjunto de réplicas e contém os dados que serão replicados, e as requisições que as réplicas poderão atender. E outra que acessa essas réplicas no intuito de alterar e recuperar esses dados por meio dessas requisições.

Para iniciar um sistema de réplicas é necessário definir uma classe Python que contém os dados a serem replicados e a implementação de suas requisições, e em seguida iniciar uma réplica raiz que recebe conexões das demais réplicas que irão compor o conjunto de réplicas. O OpenReplica usa o algoritmo Paxos para garantir que esse conjunto se mantenha consistente e capaz de atender as requisições que virão de seus clientes.

```
1  class Counter:
2    def __init__(self, value=0):
3      self.value = value
4
5    def decrement(self):
6      self.value -= 1
7
8    def increment(self):
9      self.value += 1
10
11   def getvalue(self):
12     return self.value
13
14   def __str__(self):
15     return "The counter value is %d" % self.value
```

Código-Fonte 19: Classe com dados que serão replicados e suas requisições

O Código-Fonte 19 implementa a classe **`Counter`** que será replicada. Ela possui a variável **`value`**, que é o dado replicado, e os métodos **`decrement`**, **`increment`** e **`getvalue`**, que serão as requisições aceitas pelas réplicas. Para iniciar a réplica raiz de um conjunto de réplicas, basta executar o comando mostrado no Código-Fonte 20, nesse comando a classe **`Counter`** está implementada em um arquivo chamado *counter*.

```
concoord replica -o counter.Counter -a 127.0.0.1 -p 14000
```

Código-Fonte 20: Comando para iniciar a primeira réplica de um conjunto de réplicas

O comando do Código-Fonte 20 é formado pelos parâmetros: **`objectname (-o)`**, que define qual classe será replicada; **`addr (-a)`**, que define o endereço de execução da réplica; e **`port (-p)`**, que define a porta em que a réplica receberá as requisições. Depois que a réplica raiz foi criada, as demais réplicas serão criadas usando o comando do Código-Fonte 21.

```
concoord replica
  -o counter.Counter -b 127.0.0.1:14000 -a 127.0.0.1 -p 14001
```

Código-Fonte 21: Comando para criar novas réplicas e associar a uma outra réplica raiz

O comando que cria as réplicas que estão associadas a uma réplica raiz possui um parâmetro adicional que indica qual o endereço da réplica raiz. No Código-Fonte 21, o parâmetro **`bootstrap (-b)`** indica qual o endereço da réplica raiz ao qual a nova réplica criada por esse comando será associada. Um conjunto de réplicas pode ter quantas réplicas forem necessárias, quanto mais réplicas existirem, maior será sua tolerância a falhas e sua capacidade de atender a um número maior de requisições.

Os clientes que pretendem realizar requisições a esse conjunto de réplicas devem usar uma classe **`ClientProxy`** para enviar requisições que atendam a interface esperada pelas réplicas. O Código-Fonte 22 mostra a implementação da classe **`Counter`**, que deve ser usada no lugar da classe Counter original. Usando essa interface, programas em Python podem se conectar ao conjunto de réplicas criado e requisitar que métodos sejam executados. O método **`invoke_command`** de ClientProxy é quem requisita às réplicas que executem um método do objeto do tipo **`Counter`** mantido por elas.

O método **`decrement`** da interface faz uma chamada ao método **`invoke_command`**. Esse método do objeto **`proxy`** faz uma requisição ao conjunto de réplicas para que elas executem o método **`decrement`** do objeto **`Counter`** que elas mantêm.

As classes adaptadoras que usam **`ClientProxy`** são fortes candidatas a serem programadas com o auxílio da metaprogramação, elas são padronizadas e podem ser inferidas com base nas classes usadas para criar as réplicas. Similar às classes **`Action`** de Treplica, essas classes não possuem lógica referente à aplicação, elas são somente uma casca para a lógica implementada em outras classes da aplicação.

```
from concoord.clientproxy import ClientProxy

class Counter:
  def __init__(self,bootstrap,timeout=60,debug=False,token=None):
    self.proxy = ClientProxy(bootstrap, timeout, debug, token)

  def __concoordinit__(self, value=0):
    return self.proxy.invoke_command('__init__', value)

  def decrement(self):
    return self.proxy.invoke_command('decrement')

  def increment(self):
    return self.proxy.invoke_command('increment')

  def getvalue(self):
    return self.proxy.invoke_command('getvalue')

  def __str__(self):
    return self.proxy.invoke_command('__str__')
```

Código-Fonte 22: Classe **proxy** para a classe **Counter**

A implementação de uma aplicação cliente é mostrada no Código-Fonte 23. Ela importa a classe `proxy.Counter`, cria um objeto do tipo `Counter` que aponta para as réplicas do conjunto criado e realiza requisições para essas réplicas. Quando o conjunto de réplicas recebe essa requisição, ele se organiza para eleger qual das réplicas irá atender a ela.

```
>>> from concoord.proxy.counter import Counter
>>> c = Counter("127.0.0.1:14000, 127.0.0.1:14001")
>>> c.increment()
>>> c.increment()
>>> c.getvalue()
```

Código-Fonte 23: Implementação do cliente que acessa as réplicas de **Counter**

2.4 Coesão e Acoplamento de Requisitos não Funcionais

Os requisitos funcionais de um programa descrevem o que deve ser realizado por ele (GLINZ, 2007). Esses requisitos podem ser definidos como o que o programa é capaz de fazer (IEEE, 1990); ou como os aspectos comportamentais do sistema (ANTON,

1997). No caso de uma calculadora, esses requisitos podem ser as operações matemáticas de adição e subtração. Os requisitos funcionais são normalmente associados ao domínio para qual o programa foi desenvolvido.

Os requisitos não funcionais, ao contrário dos anteriores, tratam de características como desempenho, confiabilidade e segurança, entre outras (GLINZ, 2007). Eles podem ser definidos como os aspectos sobre o qual o sistema opera (ANTON, 1997). Essas características são mais genéricas e, uma vez implementadas, podem ser reutilizadas em diversos programas. Normalmente, a replicação é enquadrada junto aos requisitos não funcionais de um programa. Não importa qual domínio use o conceito de replicação, ele permanecerá o mesmo. A implementação de uma solução de replicação pouco acoplada e coesa pode ser usada por diversos programas.

As partes que compõem um programa de computador podem ser reutilizadas com menor dificuldade se respeitarem determinadas características. Essas partes devem ser coesas, elas não devem ser responsáveis por mais do que uma tarefa (EDER; KAPPEL; SCHREFL, 1994). Caso uma parte do programa cuide da conexão com algum dispositivo e do desenho da interface de usuário em um monitor, ela não é uma parte coesa. O reuso da função de conexão exige que ela seja separada da função de desenho no monitor. Partes coesas são responsáveis por somente uma tarefa e podem ser reutilizadas sem adaptações.

O *framework* OpenReplica, mostrado na pesquisa de Deniz e Sirer (ALTINBÜKEN; SIRER, 2012), é usado para implementar serviços confiáveis. Ele é responsável por garantir a replicação dos serviços e pode ser considerado um programa coeso, uma vez que não detêm responsabilidades diferentes da replicação. A implementação do OpenReplica é baseada na orientação a objetos e, para usar o *framework* em um programa, é preciso que uma camada de interface seja desenvolvida para encapsular os métodos que contêm o comportamento replicado (Seção 2.3). Acoplar o OpenReplica ao programa usando essas interfaces é uma característica indesejada, pois caso ele passe por alguma modificação também pode ser necessário modificar o programa que utiliza o OpenReplica.

No *framework* Treplica não é diferente, o acoplamento dele com a aplicação é uma característica indesejada igual ao que ocorre com as aplicações que usam OpenReplica. Os *frameworks* que resolvem algum tipo de requisito não funcional provavelmente apresentaram essa mesma característica de acoplamento indesejado (EDER; KAPPEL; SCHREFL, 1994).

Quanto menos acopladas forem as partes do programa, melhor será para manter e reusar essas partes. O desejado é projetar programas com boa coesão e pouco acoplamento (HITZ; MONTAZERI, 1995). As técnicas de metaprogramação podem ser uma alternativa para alcançar esse objetivo. Programas que tenham requisitos não

funcionais podem ser desenvolvidos e usados com transparência se suas partes não funcionais forem projetadas usando metaprogramação. O acoplamento de um programa é considerado alto quando, para se modificar partes referentes a determinado requisito, se torna necessário alterar outras partes não relacionadas ao mesmo requisito.

2.4.1 Requisitos de Validação e Perda de Desempenho

As validações realizadas pelas aplicações também podem entrar na lista de seus requisitos não funcionais. Os programas que executam alguma forma de validação podem usar de metaprogramação para realizar essa atividade de verificação. Chlipala (CHLIPALA, 2013) mostra como é possível verificar o código fonte em tempo de compilação usando macros, mantendo suas partes pouco acopladas a essa verificação. Ele verifica o código fonte somente na compilação, mantendo o desempenho do programa durante a execução, uma vez que a execução não está sobrecarregada com verificações.

Na pesquisa realizada por Mekruksavanich et al. (MEKRUKSAVANICH; YUPAPIN; MUENCHAISRI, 2012) também foi usada metaprogramação para detectar defeitos em programas. Para isso foram criados componentes que são capazes de descrever e identificar defeitos nesses programas. Não é diferente o trabalho de Blewitt et al. (BLEWITT; BUNDY; STARK, 2005), que faz uso de técnicas de metaprogramação para validar programas de computador buscando inconsistências.

A execução de uma aplicação pode ter sua velocidade prejudicada por estar sobrecarregada de validações. Quanto mais etapas de validação acontecerem antes da execução da aplicação, menos sua performance será impactada. Em linguagens compiladas, determinadas validações podem ser movidas para dentro da compilação reduzindo o tempo consumido durante a execução dessas aplicações. Validações a respeito das entradas e saídas de um programa podem ser feitas somente sobre as características das entradas e saídas, não é possível em compilação inferir validações a respeito de seu conteúdo, uma vez que ele só existe durante a execução da aplicação.

Refraseando a introdução, essa pesquisa mostra como os requisitos não funcionais de replicação podem ser implementados usando metaprogramação para produzir programas coesos e pouco acoplados. Também é discutida a validação desses programas, no intuito de garantir que o desenvolvimento destes requisitos respeitem as regras necessárias à replicação.

3 Replicação Usando Metaprogramação

Essa pesquisa propõe a escrita de componentes usando metaprogramação para auxiliar no desenvolvimento de programas distribuídos que usam replicação ativa. Esses componentes serão usados para encapsular os trechos de código referentes ao comportamento distribuído (TOURWÉ; MENS, 2003) e para verificar se o código produzido respeita as restrições necessárias a esses programas (FILMAN; HAVELUND, 2002). São usados o *framework* Treplica e a linguagem de programação Cyan para demonstrar essa abordagem. Cyan possui estruturas de metaprogramação que permitem inspecionar e automatizar a escrita de código fonte.

3.1 Framework Treplica Orientado a Protótipos

Treplica é um *framework* (VIEIRA; BUZATO, 2010) usado para desenvolver aplicações distribuídas. Ele implementa um ambiente de replicação ativa baseado em máquinas de estado. Os estados são contextos de Treplica e guardam os valores da aplicação que serão replicados. As transições entre os estados são ações de Treplica e manipulam os valores mantidos pelo contexto. O histórico de transições realizadas durante a execução da aplicação também é mantido para que novas instâncias criadas possam ser atualizadas para o estado atual da aplicação. Para isso, a nova instância executa todas as transições do histórico, e ao final dessas execuções, a instância terá o mesmo contexto que as outras réplicas. Isso acontece porque a nova réplica partiu do mesmo estado inicial e realizou todas as transições executadas até o momento. Treplica usa o algoritmo Paxos (LAMPORT, 2006) para realizar a replicação e fornece ao programador uma abstração de fila persistente ordenada.

Replicação não faz parte dos requisitos funcionais das aplicações, por esse motivo Treplica tenta fornecer interfaces que tornam o mecanismo de replicação transparente aos desenvolvedores (VIEIRA; BUZATO, 2010). A tarefa de criar novas aplicações pode ser separada em duas etapas: uma etapa de definição do contexto da aplicação que determina quais dados serão replicados, e uma etapa para definir quais métodos desse contexto devem funcionar como ações de treplica.

Para desenvolver aplicações em Cyan usando o *framework* Treplica, foi implementado um pacote Cyan que incorpora a biblioteca Java de Treplica. Essa seção mostra como esse pacote deve ser usado para desenvolver uma aplicação replicada em Cyan. A implementação dos protótipos `Context`, `Action` e `Treplica` que pertencem a esse pacote são mostradas na Seção 3.1.2. Esses protótipos encapsulam as classes Java de Treplica que foram mostradas na Seção 2.2. A classe `StateMachine` é encapsulada pelo

protótipo **Treplica**, a classe **Action** pelo protótipo **Action** e a classe **Serializable** pelo protótipo **Context**.

O protótipo que representa o contexto da aplicação deve estender **Context**, isso garante que esse protótipo seja seriável (*serializable*). Essa propriedade permite que os métodos definidos como ações possam converter os dados em texto, para que esses dados possam ser compartilhados com as demais réplicas. Essa abordagem evita problemas de compatibilidade entre sistemas, pois o formato texto é padronizado.

Os protótipos que representam as ações devem estender **Action** e são responsáveis pela transição entre os estados da aplicação. Eles manipulam o contexto da aplicação por meio do método **executeOn:**. Esse método funciona como uma casca para a chamada do método que deve ser replicado. Esses protótipos também são seriáveis para que Treplica possa enviar cópias dessa ação às demais réplicas. Quando uma ação é executada, todas as réplicas recebem uma cópia dessa ação e cada uma delas executa sua cópia. Desse modo todas as réplicas mantêm seu estado atualizado a cada ação executada pela aplicação.

```
1  package main
2
3  import treplica
4
5  object Info extends Context {
6    var String text
7
8    func init {
9      self.text = "";
10   }
11
12   func setText: String text {
13     self.text = text;
14   }
15
16   func getText -> String {
17     return self.text;
18   }
19 }
```

Código-Fonte 24: Exemplo de protótipo que estende **Context** de Treplica

A aplicação da Seção 2.2 será reimplementada aqui usando o pacote Treplica de Cyan. Os Códigos-Fonte 16, 17 e 18 em Java são implementados em Cyan nos Códigos-Fonte 24, 25 e 26 respectivamente. No Códigos-Fonte 24 a variável de instância **number** não foi considerada. No Código-Fonte 24 é definido o contexto dessa aplicação, esse

protótipo recebe o nome de `Info` e estende `Context`. Ele contém a variável `text` do tipo `String`. Podemos considerar que o estado dessa aplicação é composto por essa variável, esse estado pode ser alterado usando o método `setText` desse protótipo.

A ação que manipula o valor da variável `text` está definida no Código-Fonte 25. O protótipo desse código recebe o nome de `UpdateAction`, e ele é uma casca para a chamada do método `setText` de `Info`. O protótipo `UpdateAction` estende `Action` para representar uma ação de Treplica e implementa o método `executeOn` chamando `setText` para atualizar `text`. A ação também é seriável, por isso ela pode ser repassada por Treplica a todas as réplicas. Isso não seria possível de ser realizado com uma chamada direta ao método `setText`. Os parâmetros do método chamado pela ação são representados como variáveis de `UpdateAction` para que sejam serializados junto com a ação. Assim, as réplicas que receberem essa ação podem chamar o mesmo método com os mesmos parâmetros.

```
1  package main
2  import treplica
3
4  object UpdateAction extends Action {
5    var String updateText
6
7    func init: String text { self.updateText = text; }
8
9    func executeOn: Context context {
10     var info = Info cast: context;
11     info setText: self.updateText;
12   }
13 }
```

Código-Fonte 25: Protótipo que implementa uma transição

O Código-Fonte 26 define o método `run` do protótipo `Program`. Esse é o primeiro método a ser chamado quando a aplicação for executada. Na Linha 8 um objeto do tipo `Treplica` é instanciado e na Linha 9 a máquina de estados recebe o contexto `info`. Depois da inicialização, para executar uma `Action`, o método `execute` do protótipo `Treplica` deve ser chamado. O método `execute:` recebe um objeto do tipo `Action` como parâmetro, por meio da variável `action`. Esse método serializa a ação e envia uma cópia para cada réplica. Quando a ação for executada por cada réplica, ela chamará o método `executeOn:` que modifica o valor da variável `text` do objeto `info` daquela réplica.

```
1  package main
2  import treplica
```

```
3
4  object Program {
5    func run: Array<String> args {
6      let info = Info new;
7      let treplica = Treplica new;
8      treplica runMachine: info numberProcess: 3
9               rtt: 200 path: "/var/tmp/magic" ++ args[1];
10     let action = UpdateAction new: "text";
11     if args[1] == "1" {
12       treplica execute: action;
13     }
14   }
15 }
```

Código-Fonte 26: Configuração e execução de Treplica

O diagrama de sequência da Figura 10 mostra o fluxo de execução dessa aplicação. As réplicas **A** e **B** iniciam a execução pelo método `run` da classe `Program`. O contexto da aplicação (`Info`) e a máquina de estados de Treplica (`Treplica`) têm seus objetos instanciados por esse método. Em seguida, a **Réplica A** chama o método `execute:` de Treplica que irá executar a ação `UpdateAction`. Treplica replica essa chamada para a **Réplica B** e executa a ação replicada em ambas as réplicas (`executeOn:`).

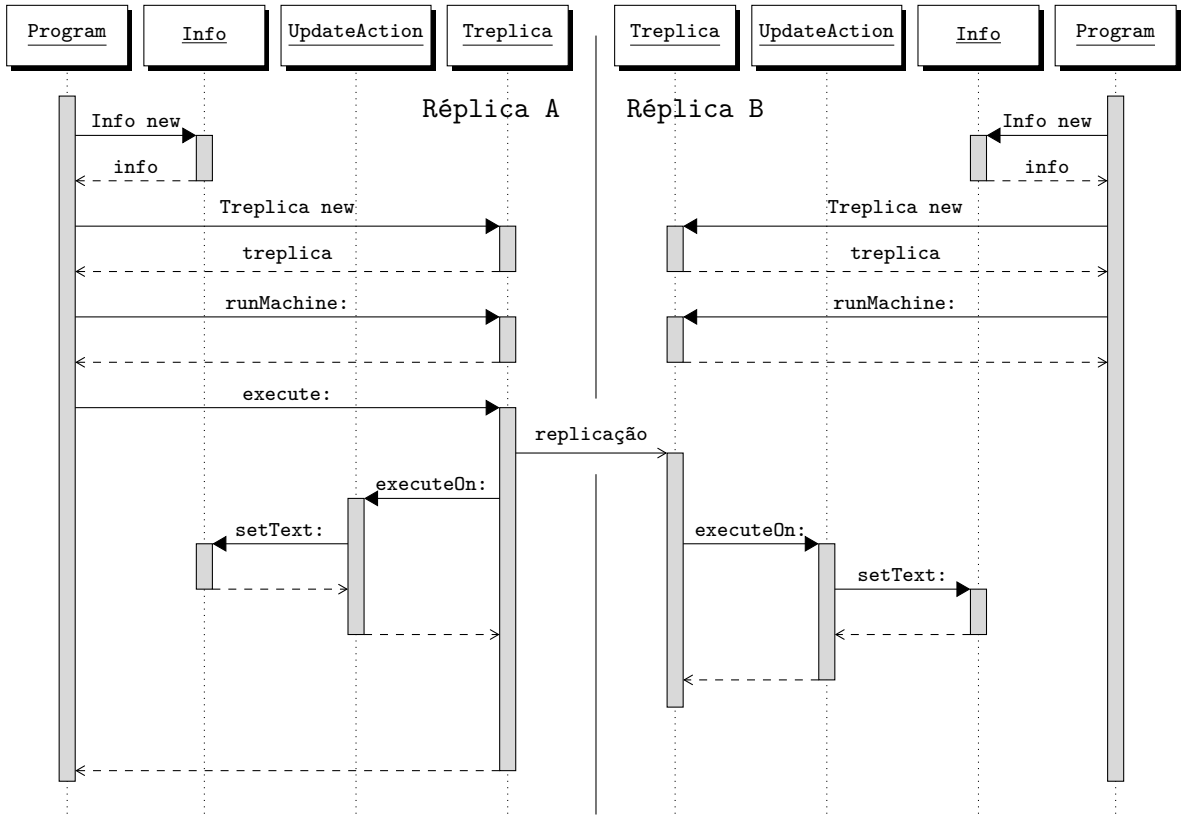

Figura 10: Execução do exemplo de Treplica em Cyan

Na execução do programa, cada réplica receberá um número como argumento de modo incremental. Se forem executadas duas réplicas, a primeira réplica executada irá receber o número `1` e a segunda réplica o número `2`. De modo que a réplica que receber o número `1` como argumento irá executar a ação de Treplica que será copiada às outras réplicas. As réplicas iniciam a execução pelo método **`run:`** e são associadas à máquina de estado de Treplica pelo método **`runMachine:`**. A máquina de estados de Treplica é responsável por copiar as ações entre as réplicas associadas.

3.1.1 Não Determinismo de Data e Hora

O Treplica requer adequada atenção à forma como as aplicações são implementadas, devido às inconsistências que são resultado de um código que não respeite as restrições impostas pelo modelo de replicação adotado. Um exemplo é a chamada de métodos não deterministas dentro das transições. O protótipo **`UpdateAction`** chama o método **`setText:`**, caso essa função faça uso de um método não determinista como **`currentTime`**, teremos um problema de consistência de valores entre as réplicas de **`Info`**. Esse protótipo será alterado para servir de exemplo. O Código-Fonte 27 mostra o método **`setText:`**, que ao usar o método **`currentTime`**, se torna não determinista.

```
1 ...
2 object Info extends Context {
3   ...
4   func setText: String text {
5     var date = System currentTime asString;
6     self.text = text ++ date;
7   }
8   ...
9 }
```

Código-Fonte 27: Exemplo de método **setText:** não determinista

Essa alteração causa um problema de consistência, pois **`info`** poderá assumir um valor diferente cada vez que o método **`executeOn:`** de uma instância de **`UpdateAction`** for chamado. Essa propriedade não determinista do método **`currentTime`** pode levar a uma divergência entre os valores atribuídos a **`text`** em cada réplica, pois cada réplica vai chamar a ação uma vez e o resultado obtido pode ser diferente em cada chamada.

Uma solução para esse problema é realizar a operação não determinista antes de criar a instância de **`UpdateAction`**. O resultado dessa operação será passado à instância em sua inicialização. Desse modo o resultado da operação se torna um valor fixo que será copiado para as outras réplicas. Ao invés da ação realizar a operação não

determinista, ela só recebe o resultado dessa ação e o repassa às outras réplicas. O Código-Fonte 28 mostra a implementação determinista dessa ação.

```
1  object UpdateAction extends Action {
2    var String date
3    var String updateText
4
5    func init: String text, String date {
6      self.date = date;
7      self.updateText = text;
8    }
9
10   func executeOn: Context context {
11     var info = Info cast: context;
12     info setText: self.updateText ++ self.date;
13   }
14 }
```

Código-Fonte 28: Exemplo ação determinista

O Código-Fonte 29 cria um objeto **`date`** que recebe a data atual do sistema. Em seguida a variável **`action`** recebe uma instância do protótipo **`UpdateAction`** que foi inicializada com o valor da variável **`date`**. Esse valor fixo será atribuído às outras réplicas pela ação **`UpdateAction`** quando ela for replicada.

```
1 var date = System currentTime asString;
2 var action = UpdateAction new: "text", date;
```

Código-Fonte 29: Exemplo de construção da ação determinista

3.1.2 Incorporação de Treplica em um Pacote Cyan

O *framework* Treplica foi desenvolvido em Java e integrado a um pacote de Cyan para que ele possa ser usado por aplicações escritas em Cyan. Essa implementação consistiu de protótipos em Cyan que fazem chamadas diretas ao *framework* Treplica desenvolvido em Java. Essa implementação foi realizada usando a construção de linguagem **`@javacode`**, como mostrado no Código-Fonte 9 da Seção 1.3.2.

O código-fonte dentro da construção **`@javacode`** está em Java e o restante do código-fonte em Cyan. Uma variável `rtt` declarada no código-fonte em Cyan pode ser referenciada no trecho de Java (dentro do escopo) como `_rtt`.

O Código-Fonte 30 define o protótipo **Context** que representa um contexto de Treplica. Todo contexto de Treplica deve implementar a interface **Serializable** de Java. Esse protótipo também contém uma variável que recebe uma instância do tipo **Treplica**. A atribuição dessa variável ocorre de forma cruzada, a instância de **Context** é referenciada pela instância de **Treplica**, e o mesmo ocorre com a instância de **Treplica** que é referenciada pela instância de **Context**. Essa referência cruzada irá permitir que os metaobjetos transformem um método tradicional em um método replicado, como será mostrado na Seção 3.3.

```
1  package treplica
2
3  @javacode<<< import java.io.Serializable; >>>
4
5  @javaImplements(Serializable)
6  abstract object Context
7    var Treplica treplica;
8
9    func init {
10     @javacode<<< _treplica = null; >>>
11   }
12
13   func setTreplica: (Treplica tr) { treplica = tr; }
14
15   func getTreplica -> Treplica { return treplica; }
16 end
```

Código-Fonte 30: Código Fonte de **Context**

O protótipo **Treplica** mostrado no Código-Fonte 31 expõe a máquina de estados de Treplica implementada em Java para as aplicações em Cyan. Ele contém a variável **stateMachine** que é iniciada com uma instância da máquina de estados de Treplica. Essa atribuição é feita pelo método **runMachine** que recebe um contexto de Treplica e cria a máquina de estados usando esse contexto e alguns parâmetros de configuração.

Depois que a instância de Treplica for inicializada é possível usar o método **getState** para acessar o contexto que foi passado à função **runMachine**. O último método desse protótipo é usado para executar as ações de Treplica, ele deve receber uma ação que possa ser associada ao contexto com o qual a instância desse protótipo foi inicializada. Na chamada do método **execute:**, deve ser passada a ele uma instância da ação que se deseja executar. Todos os métodos do protótipo **Treplica** foram implementados para que chamem diretamente os métodos da máquina de estados de Treplica em Java.

```
1  package treplica
2
3  @javacode<<<
4  import java.io.Serializable;
5  import br.unicamp.treplica.StateMachine;
6  >>>
7
8  object Treplica
9    @javacode<<< public static StateMachine stateMachine; >>>
10
11   func init { ... }
12
13   func runMachine: (Context context) numberProcess: (Int nProcesses)
14       rtt: (Int rtt) path: (String stableMedia) {
15     @javacode<<<
16     try {
17       stateMachine = StateMachine.createPaxosSM(
18         (Serializable) _context,((CyInt ) _rtt).n, ((CyInt )
19         _nProcesses).n, false, ((CyString ) _stableMedia).s);
20     } catch (Exception p) {
21       ...
22     }
23     >>>
24     return Nil
25   }
26
27   func getState -> Context {
28     @javacode<<<
29     _Context result;
30     result = (_Context)stateMachine.getState();
31     return result;
32     >>>
33   }
34
35   func execute: (Action action) -> Dyn {
36     @javacode<<<
37     try {
38       return stateMachine.execute(_action);
39     } catch (Exception p) {
40       ...
41     }
42     >>>
43     return Nil
44   }
45 end
```

Código-Fonte 31: Código Fonte de **Treplica**

As ações de Treplica herdam o protótipo **Action** mostrado no Código-Fonte 32. Esse protótipo implementa as interfaces **Serializable** e **Action** de Java. A classe **Action** pertencente à Treplica possui um método **executeOn** que em Java recebe um contexto como parâmetro. O método **executeOn** em Java que é uma redefinição de um método da interface **Action** em Java foi redefinido no protótipo **Action** de Cyan, de modo que quando chamado, o fluxo de execução passe ao método **_executeOn_1** de Java. O método **executeOn:** de Cyan depois de compilado recebe o nome de **_executeOn_1**, portanto o método abstrato **executeOn:** de Cyan será chamado quando o método **executeOn** de Java for chamado. O protótipo que herda **Action** de Cyan é responsável por implementar o método abstrato **executeOn:**. Assim essa implementação do método abstrato será chamada pelo método **executeOn** de Java.

```
1  package treplica
2
3  @javacode<<<
4  import java.io.Serializable;
5  import br.unicamp.treplica.Action;
6  >>>
7
8  @javaImplements(Action)
9  @javaImplements(Serializable)
10 abstract object Action
11   func init {
12   }
13
14   @javacode<<<
15   @Override
16   public Object executeOn(Object stateMachine) {
17     return _executeOn_1( (_Context)stateMachine );
18   }
19   >>>
20
21   abstract func executeOn: Context context -> Dyn
22 end
```

Código-Fonte 32: Código Fonte de **Action**

3.2 Usando os Metaobjetos de Treplica

Aplicações replicadas desenvolvidas usando Treplica precisam da criação de um protótipo que represente o contexto que será replicado, e de ações que representam as chamadas de métodos deste contexto. Em Treplica, para cada nova ação é neces-

sário criar um novo protótipo que estenda o protótipo `Action`. Portanto, o aumento do número de ações leva a um aumento do número de protótipos a serem criados. A complexidade desses protótipos também muda em função do número de parâmetros necessários à ação; quanto mais parâmetros, maior será o código e a complexidade do protótipo. Esse processo repetitivo de criação de ações pode tornar o código-fonte propenso a falhas.

Ao criar protótipos que estendem `Action`, podemos convencionar que cada protótipo irá replicar um método único do contexto da aplicação. O protótipo `UpdateAction` do Código-Fonte 25 é um modelo que atende a essa convenção. Esse protótipo replica a chamada do método `setText:` do contexto do Código-Fonte 24.

Para que os protótipos respeitem essa convenção, eles devem ter um construtor que irá receber os parâmetros do método replicado, variáveis de instância que irão armazenar os valores recebidos pelo construtor e o método `executeOn:`, que chamará o método replicado passando as variáveis de instância atribuídas pelo construtor. Seguindo essa convenção foi projetado o metaobjeto `treplicaAction` para criar, durante a compilação, os protótipos que herdam de `Action`. Se esse código-fonte gerado pelo metaobjeto fosse implementado por um desenvolvedor poderia acabar contendo falhas devido à repetição envolvida nessa implementação.

O metaobjeto `treplicaAction` é usado no Código-Fonte 33 para anotar o método `setText:`. Durante a compilação do protótipo `Info` uma nova ação de Treplica é gerada com base no método `setText:`. Esse metaobjeto cria um protótipo similar ao mostrado no Código-Fonte 25 e adiciona métodos ao protótipo `Info`, que permitem replicar as chamadas do método `setText:` de modo direto, como mostrado na Linha 12 do Código-Fonte 34. O protótipo `Info` modificado é mostrado no Código-Fonte 36 da Seção 3.3.

Essa chamada direta a um método que será replicado consiste em remover a necessidade de chamar o método `execute` (Linha 13 do Código-Fonte 26) passando a ação que representa esse método `setText:`. A chamada ao método `execute` é incorporada ao contexto que possui o método replicado. O código-fonte gerado pelo metaobjeto que permite chamar um método replicado diretamente é mostrado na Seção 3.3.

A verificação de não determinismo em métodos replicados é uma característica secundária do metaobjeto `treplicaAction`. Durante a compilação, um método do metaobjeto percorre as instruções do método anotado em busca de chamadas a métodos não deterministas. Isto é feito de forma recursiva.

Os métodos considerados não deterministas são definidos pelo desenvolvedor por meio de arquivos. Nesses arquivos é possível definir quais chamadas de métodos não deterministas devem ser substituídos durante a compilação por outras chamadas de métodos deterministas. Os detalhes são explicados na Seção 3.3.2.

```
1  ...
2  object Info extends Context {
3    var String text
4    ...
5    @treplicaAction
6    func setText: String text {
7      self.text = text;
8    }
9    ...
10 }
```

Código-Fonte 33: **Action** usando Metaobjeto

A criação e a configuração de Treplica também são realizadas de forma padronizada: um contexto deve ser instanciado e passado à Treplica em seu construtor. Sua inicialização é feita pela chamada do método `runMachine` do objeto `Treplica`, como mostrado nas Linhas 9 e 10 do Código-Fonte 26. O metaobjeto `treplicaInit` foi projetado para iniciar o *framework* e vinculá-lo a uma instância de objeto criada localmente. Esse metaobjeto é usado para anotar variáveis que estendam `Context`. Na Linha 9 do Código-Fonte 34, `treplicaInit` anota a declaração da variável `info`. Esse metaobjeto produz uma nova versão da declaração do método `run`, que cria uma instância de `Treplica` e atribui o objeto de `info` a essa instância, similar ao método `run` do Código-Fonte 26. A versão do método `run` produzida pelo metaobjeto será mostrada no Código-Fonte 43 na Seção 3.3.3.

Usar os metaobjetos `treplicaInit` e `treplicaAction` reduz a quantidade e a complexidade do código produzido durante o desenvolvimento de uma aplicação replicada. Isso torna a aplicação menos propensa a falhas, o oposto do que ocorre na Seção 3.1, que mostra o desenvolvimento de uma aplicação replicada que não usa metaprogramação. O código-fonte implementado usando esses metaobjetos fica parecido com a implementação de um código-fonte não replicado se as anotações forem removidas. Isso permite implementar um código-fonte mais transparente por ocultar a replicação do desenvolvedor.

```
1  package main
2
3  import treplica
4
5  object Program {
6    func run: Array<String> args {
7      var local = "/var/tmp/magic" ++ args[1];
8
```

```
9       @treplicaInit( 3, 200, local )
10      var info = Info new;
11
12      if args[1] == "1" {
13        info setText: "text";
14      }
15    }
16  }
```

Código-Fonte 34: Configuração de Treplica usando metaobjetos

3.3 Implementação dos Metaobjetos de Treplica

Esta seção descreve como os metaobjetos `treplicaAction` e `treplicaInit` foram implementados e mostra os códigos-fonte produzidos por eles durante a compilação. O código-fonte implementado é mostrado por completo no Apêndice A, e no decorrer dessa seção, trechos desse código-fonte são utilizados. Os trechos de códigos-fonte que foram extraídos do Apêndice A são marcados com uma borda dupla.

As classes dos metaobjetos `treplicaAction` e `treplicaInit` herdam da classe `CyanMetaobjectWithAt` e implementam interfaces do *PMO* de Cyan. A classe herdada e essas interfaces permitem que esses metaobjetos participem de etapas da compilação. Além dessa classe existem outras que podem ser herdadas, como explicado na Seção 1.3.1.

Toda classe de metaobjeto que herda de `CyanMetaobjectWithAt` deve redefinir o método `getName` para que ele retorne o nome da anotação com que esse metaobjeto deve ser associado. Quando o compilador encontra uma anotação no programa que está sendo compilado, ele procura por um metaobjeto que possa ser associado. Se não existir nenhum metaobjeto que possa ser associado à anotação, o compilador mostrará um erro de compilação. A implementação do método `mayBeAttachedList` herdado de `CyanMetaobjectWithAt` pelas classes dos metaobjetos define onde os metaobjetos podem ser acoplados. No caso do metaobjeto `treplicaAction` implementado pela classe `CyanMetaobjectTreplicaAction`, o método `mayBeAttachedList` retorna `AttachedDeclarationKind.METHOD_DEC` para indicar que ele deve ser atrelado somente a declarações de métodos. O trecho de Código-Fonte 35 mostra a implementação dos métodos `getName` e `mayBeAttachedList` no metaobjeto `treplicaAction`.

Os exemplos e códigos-fonte dessa seção foram desenvolvidos para uma versão antiga do compilador de Cyan. Para que esse código-fonte possa ser usado com versões recentes do compilador, algumas alterações são necessárias. O endereço eletrônico com o compilador de Cyan e os arquivos usados nessa pesquisa são mostrados na Conclusão.

```
1  @Override
2  public String getName() {
3    return "treplicaAction";
4  }
5
6  @Override
7  public DeclarationKind[] mayBeAttachedList() {
8    return decKindList;
9  }
10
11 private static DeclarationKind[] decKindList =
12     new DeclarationKind[] { DeclarationKind.METHOD_DEC };
```

Código-Fonte 35: Métodos de **CyanMetaobjectTreplicaAction**

Dessa forma, durante a compilação do Código-Fonte 33 quando o compilador de Cyan encontra a anotação **`treplicaAction`**, ele procura por um metaobjeto com nome **`"treplicaAction"`** e verifica se a anotação encontrada está atrelada a uma definição de método de Cyan. Somente depois um ou mais métodos desse metaobjeto associados à anotação são chamados pelo compilador para gerar uma nova versão do código-fonte que está sendo compilado.

```
1  package main
2
3  import treplica
4
5  object Info extends Context {
6    ...
7    func setText: String text {
8      var action = InfosetText new: text;
9      self getTreplica execute: action;
10   }
11
12   func setTextTreplicaAction: String text {
13     self.text = text;
14   }
15   ...
16 }
```

Código-Fonte 36: Protótipo **Info** modificado

3.3.1 Ações e o metaobjeto **treplicaAction**

Para que o metaobjeto `treplicaAction` funcione como explicado na Seção 3.2 é necessário que a classe `CyanMetaobjectTreplicaAction` estenda do *PMO* as interfaces `IActionProgramUnit_ati` e `IAction_dsa`. Elas participam das etapas **Ati Actions (3)** e **Calc. Internal Types (6)** respectivamente. Esta classe deve redefinir métodos das interfaces para criar novos métodos em Cyan, renomear métodos existentes e criar novos protótipos.

A execução desse metaobjeto acontece em etapas e envolve um conjunto de funções das interfaces do metaobjeto. Essa execução ocorre do seguinte modo: primeiro o método `ati_renameMethod` do metaobjeto `treplicaAction` renomeia o método `setText:` do protótipo `Info` para `setTextTreplicaAction:`. Para evitar que um outro método tenha o mesmo nome é adicionado ao final de `setTextTreplicaAction:` um complemento único gerado pelo metaobjeto. Depois, o novo método `setText:` é criado no mesmo protótipo do método original que foi renomeado. O Código-Fonte 36 mostra tanto o método `setTextTreplicaAction:` quanto o novo método `setText:`. Por último, ele cria o novo protótipo `InfosetText` mostrado no Código-Fonte 37 para representar a chamada replicada do novo método `setTextTreplicaAction:`. A nova versão de `setText:` chama o método `execute:` de Treplica passando um objeto do tipo `InfosetText`.

```
1  object InfosetText extends Action {
2    var String textVar
3    func init:  String text {
4      textVar = text;
5    }
6
7    override
8    func executeOn: Context context {
9      var obj = Info cast: context;
10     obj setTextTreplicaAction:  textVar;
11   }
12 }
```

Código-Fonte 37: Protótipo criado por **treplicaAction**

Para implementar o comportamento descrito do metaobjeto `treplicaAction`, foi necessário definir três métodos da interface `IActionProgramUnit_ati`. Primeiro o método `ati_renameMethod`, que renomeia `setText:` para `setTextTreplicaAction:`. Em seguida, o método `ati_NewPrototypeList`, que cria o protótipo de `InfosetText`. Finalmente, o método `ati_codeToAddToPrototypes`, que cria o novo método `setText:`.

O método **ati_codeToAddToPrototypes**, como os demais métodos, gera uma nova versão do código-fonte e a retorna na forma de uma *String* para o compilador de Cyan. O compilador Cyan chama os métodos do metaobjeto respectivamente na ordem em que foram mencionados.

```
1  public ArrayList<Tuple3<String, String, String[]>> ati_renameMethod(
       ICompiler_ati compiler_ati) {
2    String prototypeName = compiler_ati.getCurrentPrototypeName();
3    MethodDec md = (MethodDec) this.getAttachedDeclaration();
4    String oldMethodName = md.getName();
5    String newMethodNameWithPar[] = new String[1 + md.
       getMethodSignature().getParameterList().size()];
6    String auxName = md.getNameWithoutParamNumber();
7    auxName = auxName.replaceAll(":", "");
8    newMethodNameWithPar[0] = auxName +"TreplicaAction"+ this.fixName;
9    ...
10   ArrayList<Tuple3<String, String, String[]>> tupleList = new
       ArrayList<>();
11   tupleList.add(new Tuple3<String, String, String[]>(prototypeName,
       oldMethodName, newMethodNameWithPar));
12   return tupleList;
13 }
```

Código-Fonte 38: Método que renomeia os métodos anotados com **treplicaAction**

O método **ati_renameMethod**, mostrado no Código-Fonte 38, pega o nome atual do método e adiciona **"TreplicaAction"** ao final do nome. Para evitar que um outro método tenha o mesmo nome, é adicionado ao final de **"TreplicaAction"** um complemento único gerado pelo método Java **NameServer.nextLocalVariableName()**. Se o método renomeado tiver parâmetros, eles são adicionados na mesma lista que foi adicionado o novo nome. Uma lista de tuplas formadas pelo nome antigo, o novo nome e os parâmetros é criada e retornada para o compilador. Uma nova versão do código-fonte é gerada onde o nome antigo contido em cada tupla dessa lista é substituído pelo nome novo da mesma tupla. Essa versão do metaobjeto **treplicaAction** não trata os métodos com mais de um seletor, remover essa limitação é uma sugestão de melhoria futura.

```
1  public ArrayList<Tuple2<String, StringBuffer>> ati_NewPrototypeList(
       ICompiler_ati compiler_ati) {
2    MethodDec md = (MethodDec) this.getAttachedDeclaration();
3    String methodName = md.getNameWithoutParamNumber();
4    ...
5    for (ParameterDec par : md.getMethodSignature().getParameterList()) {
```

```
6       varDef += "var " + par.getType().getName() +" "+ par.getName() + "Var\n";
7       funcPar += " " + par.getType().getName() + " " + par.getName() + ",";
8       funcAssign += par.getName() + "Var = " + par.getName() + ";\n";
9       callPar += " " + par.getName() + "Var,";
10    }
11    ...
12    String objectName = compiler_ati.getCurrentPrototypeName();
13
14    ArrayList<Tuple2<String, StringBuffer>> protoCodeList = new ArrayList<>();
15    String prototypeName = NameServer.removeQuotes(objectName + methodName);
16
17    this.fixName = NameServer.nextLocalVariableName();
18    StringBuffer code = new StringBuffer(
19      "package main\n" + "\n" +
20      "import treplica\n" + "\n" +
21      "object " + prototypeName + " extends Action\n" + "\n" +
22        varDef + "\n" +
23        "func init" + hasFuncPar + funcPar + " {\n" +
24          funcAssign +
25        "}\n" + "\n" +
26        "override\n" +
27        "func executeOn: Context context -> Dyn {\n" +
28          "var obj = Cast<" + objectName + "> asReceiver: context;\n" +
29          retStatement + "obj " + methodName + "TreplicaAction" + this.fixName
                +
30            hasCallPar + callPar + ";\n" +
31          retNullState +
32        "}\n" +
33      "end\n"
34    );
35
36    String unescape = Lexer.unescapeJavaString(code.toString());
37    protoCodeList.add(new Tuple2<String, StringBuffer>(prototypeName, new
          StringBuffer(unescape)));
38
39    return protoCodeList;
40  }
```

Código-Fonte 39: Método que cria o protótipo da ação associada ao método anotado

O método **ati_NewPrototypeList** é mostrado no Código-Fonte 39, ele cria o protótipo da ação que será associada ao método anotado (no exemplo, é o protótipo **InfosetText** do Código-Fonte 37). O nome do método anotado é atribuído à variável **methodName**. Em seguida os parâmetros do método anotado são percorridos e seus tipos e nomes são atribuídos a um grupo de variáveis (**varDef**, **funcPar**, **funcAssign**, **callPar**), como a variável **callPar**, que lista os nomes dos parâmetros

do método separados por vírgula. Essa lista é usada como entrada para o construtor de um protótipo na Linha 18 do Código-Fonte 39. Esse grupo de variáveis do método **ati_NewPrototypeList** é usado para gerar as variáveis de instância e o construtor do novo protótipo.

As variáveis de instância criadas no novo protótipo serão usadas em seu método **executeOn:**. Esse método é implementado em todas as ações e é usado para chamar o método anotado que essa ação representa. O método **ati_NewPrototypeList** termina construindo uma lista de tuplas na qual cada tupla contém o nome e o código-fonte do protótipo e retornando essa lista apara o compilador. Uma nova verão do código-fonte que contém o novo protótipo é gerada pelo compilador.

```
1  public ArrayList<Tuple2<String, StringBuffer>> ati_codeToAddToPrototypes(
       ICompiler_ati compiler) {
2    MethodDec md = (MethodDec) this.getAttachedDeclaration();
3    String methodName = md.getNameWithoutParamNumber();
4    ...
5    String objectName = compiler.getCurrentPrototypeName();
6    String prototypeName = NameServer.removeQuotes(objectName + methodName);
7
8    StringBuffer code = new StringBuffer("\n\n" +
9      "func " + methodName + hasFunPar + funcPar + hasCallRet +
10       funcRet + " {\n" +
11       "var action = " + prototypeName + " new" + hasCallPar +
12         callPar + ";\n" +
13       "var Dyn ret = getTreplica execute: action;\n" +
14       retStatement +
15     "}\n"
16   );
17
18   ArrayList<Tuple2<String, StringBuffer>> result = new ArrayList<>();
19   result.add(new Tuple2<String, StringBuffer>(objectName, code));
20   return result;
21 }
```

Código-Fonte 40: Método que adiciona o novo método do contexto

O método **ati_codeToAddToPrototypes**, que é mostrado no Código-Fonte 40, cria um novo método no protótipo do método anotado com **treplicaAction**. O método criado tem o mesmo nome, parâmetros e retorno do método que foi renomeado pelo método **ati_renameMethod**. O comportamento atribuído a esse novo método consiste em criar uma instância do novo protótipo gerado pelo método **ati_NewPrototypeList** e passar essa instância como parâmetro de uma chamada ao método **execute** de Treplica.

O Código-Fonte 33 mostra a versão original do protótipo `Info` que tem o método `setText:` marcado com a anotação `treplicaAction`. Durante a compilação esse metaobjeto gera uma nova versão desse código-fonte, como mostrado no Código-Fonte 36, e gera o protótipo `InfosetText`, mostrado no Código-Fonte 37. O método renomeado mudou de `setText:` para `setTextTreplicaAction:`, e um novo método criado recebe o nome de `setText:`.

3.3.2 Metaobjeto **treplicaAction** e não determinismo

O metaobjeto `treplicaAction` também verifica a ocorrência de não determinismo nos métodos anotados por ele e retorna um erro de compilação caso alguma ocorrência seja encontrada. Esse comportamento é implementado no método que redefine o método `dsa_codeToAdd` pertencente à interface `IAction_dsa`, que também é implementada pela classe do metaobjeto `treplicaAction`. Essa verificação é feita por meio de uma busca em profundidade que tem início no método anotado com esse metaobjeto.

Essa busca em profundidade utiliza o padrão de projeto *Visitor* para percorrer os métodos chamados na implementação do método anotado de modo recursivo. Percorrer um método significa procurar por chamadas de métodos dentro do método que está sendo percorrido e percorrer esse método também, recursivamente. A busca termina quando o método que estiver sendo percorrido não chamar outros métodos ou somente chamar métodos já analisados. Chamadas do método por ele mesmo e chamadas aos métodos do pacote padrão de Cyan (`cyan.lang`) são ignoradas.

Um método é não determinista se ele usa em sua implementação algum outro método não determinista ou se ele é explicitamente indicado como não determinista pelo desenvolvedor. No pacote treplica de Cyan ou no programa que está sendo compilado, o desenvolvedor pode adicionar regras que associam quais métodos não deterministas devem ser trocados por métodos deterministas. O metaobjeto `treplicaAction` irá procurar nesses dois locais por regras que associam métodos não deterministas que devem ser trocados por métodos deterministas. As regras podem ser adicionadas no pacote treplica, mas manter as regras junto ao pacote que implemente os métodos deterministas usados pelas regras é mais adequado.

Essas regras são definidas no arquivo de nome *deterministic* que fica na pasta *--data* e pode ser colocada no diretório do pacote e no diretório da aplicação. O metaobjeto não realiza verificações entre as regras e os protótipos e métodos do pacote onde o arquivo foi definido, as regras podem fazer referência a protótipos de qualquer pacote de Cyan. O arquivo *deterministic* não é limitado a ser definido em um único diretório, cada pacote pode definir seu arquivo de regras, e eles podem coexistir.

Cada linha desse arquivo contém uma regra que define quais são os métodos

explicitamente indicados como não deterministas e por qual método cada um deles deve ser trocado. O Código-Fonte 41 mostra como uma regra deve ser representada. O `PacoteA`, o `PrototipoA` e o `metodoA` à esquerda são a parte não determinista que será trocada pela parte determinista, que são o `PacoteB`, o `PrototipoB` e o `metodoB` à direita. Os tipos de parâmetros, o retorno e as palavras-chave de ambos os métodos devem ser os mesmos para que as expressões contidas na chamada possam ser mantidas. Caso duas regras façam referência ao mesmo `PacoteA`, `PrototipoA` e `metodoA` será usada a primeira regra que o metaobjeto encontrar.

Caso os métodos possuam parâmetros em quantidades diferentes ou de tipos diferentes, a substituição será realizada e em seguida o compilador irá indicar um erro de compilação informando que o novo método contém mais ou menos parâmetros que o esperado, ou que os tipos dos parâmetros não são compatíveis.

O método `metodoB` não deve usar variáveis de instância, uma vez que na substituição do `metodoA` não é criada uma instância do `PrototipoB`. O `metodoB` é chamado direto do `PrototipoB`, ele pode ser comparado a métodos estáticos de linguagens de programação orientadas a objetos.

```
1 PacoteA,PrototipoA,metodoA-PacoteB,PrototipoB,metodoB
```

Código-Fonte 41: Modelo de regra para indicar não determinismo

Um método `random` do protótipo `Math` contido no pacote `cyan.math` é um exemplo de método explicitamente não determinista que pode ser adicionado ao arquivo de regras pelo desenvolvedor. Caso ele queira substituir o método `random` pelo método `randomStatic` que ele desenvolveu em seu protótipo `MyMath`, ele adicionaria no arquivo *deterministic* a regra mostrada no Código-Fonte 42.

```
1 cyan.math,Math,random-main,MyMath,randomStatic
```

Código-Fonte 42: Exemplo de regra criada pelo desenvolvedor

Com relação a não-determinismo, `treplicaAction` não tem a pretensão de cobrir todos os casos possíveis. A intenção é mostrar a possibilidade de utilizar os metaobjetos para impor validações do código-fonte e para gerar novas versões desse código-fonte removendo os problemas encontrados.

Casos como o de interfaces que podem ser implementadas por diferentes protótipos são um problema uma vez que não são verificadas todas as implementações dessas

interfaces. Os métodos não deterministas que modificam variáveis de instância também são um problema, pois se substituídos por métodos deterministas que não modificam as variáveis do mesmo modo, podem causar falhas ao programa. Outro caso não tratado é o dos métodos não deterministas sobrecarregados com métodos deterministas que continuam sendo considerados não deterministas. Entre outras possíveis situações problemáticas envolvidas na substituição de uma chamada por outra.

3.3.3 Inicialização de Treplica e o metaobjeto **treplicaInit**

O metaobjeto `treplicaInit`, diferentemente de `treplicaAction`, implementa a interface `IActionVariableDeclaration_dsa` do *PMO* usada na etapa **Calc. Internal Types (6)** da compilação. O metaobjeto implementa o método `dsa_codeToAddAfter` dessa interface para gerar uma nova versão do código-fonte da declaração de variável à qual estiver atrelado. Esse metaobjeto substitui a declaração das Linhas 9 e 10 mostrada no Código-Fonte 34 pelo trecho entre as Linhas 5 e 9 do Código-Fonte 43. Esse novo código-fonte, além de declarar a variável `info`, também declara a variável `treplicainfo`. Em seguida ele atribui o objeto `info` a uma variável do objeto `treplicainfo` por meio do método `runMachine`. Finalmente, o objeto `treplicainfo` é atribuído a uma variável do objeto `info` pelo método `setTreplica`.

```
1  object Program {
2    func run: Array<String> args {
3      var local = "/var/tmp/magic" ++ args[1];
4
5      var info = Info new;
6      var treplicainfo = Treplica new;
7      treplicainfo runMachine: info numberProcess: 3
8                   rtt: 200 path: local;
9      info setTreplica: treplicainfo;
10
11     info setText: "text";
12   }
13 }
```

Código-Fonte 43: Protótipo **Program** modificado

O método `dsa_codeToAddAfter` mostrado no Código-Fonte 44 é o responsável pelo comportamento do metaobjeto `treplicaInit`. Ele gera o código-fonte necessário à inicialização da máquina de estados de Treplica e passa a variável anotada como o contexto inicial dessa máquina de estados. O código-fonte gerado por esse método é retornado ao compilador que produz uma nova versão do programa com esse código-fonte.

```
1  public StringBuffer dsa_codeToAddAfter() {
2    ...
3    StatementLocalVariableDecList varList =
4      (StatementLocalVariableDecList) dec;
5    ArrayList<Object> parameterList = withAt.getJavaParameterList();
6    ...
7
8    String nameTreplicaVar = "treplica"
9      + varList.getLocalVariableDecList().get(0).getName();
10   StringBuffer code = new StringBuffer("var "
11     + nameTreplicaVar + " = Treplica new;\n");
12   code.append(nameTreplicaVar + " runMachine: ");
13   code.append(varList.getLocalVariableDecList().get(0).getName());
14   code.append(" numberProcess: ");
15   code.append(strList[0]);
16   code.append(" rtt: ");
17   code.append(strList[1]);
18   code.append(" path: ");
19   code.append(strList[2]);
20   code.append(";\n");
21   code.append(varList.getLocalVariableDecList().get(0).getName()
22     + " setTreplica: " + nameTreplicaVar + ";\n");
23
24   return code;
25 }
```

Código-Fonte 44: Método principal do metaobjeto **treplicaInit**

3.4 Comportamento Genérico de **treplicaAction** e **treplicaInit**

Os metaobjetos **treplicaAction** e **treplicaInit** são mostrados nessa seção de forma genérica. Assim, pode-se mostrar quais trechos do código-fonte original são usados para gerar a nova versão desse código-fonte. Essas versões de código-fonte são analisadas do ponto de vista sintático, os trechos de código-fonte usados na nova versão que são obtidos da versão anterior estão representados pela sintaxe $\langle nome\ do\ trecho\ de\ código\text{-}fonte \rangle$ em ambas as versões desse código-fonte.

Como exemplo dessa notação, no Código-Fonte 45 é mostrado trecho de código que será utilizado por um metaobjeto fictício (`metaobj`) para gerar uma nova verão de código-fonte. Os trechos utilizados estão entre $\langle$ e $\rangle$, esses trechos são omitidos e recebem um nome que os identifica. Todos os códigos-fonte dessa seção estão escritos na linguagem Cyan. Trechos concatenados são indicados por $\langle\langle trecho\ 1\rangle\langle trecho\ 2\rangle\rangle$, mostrando que o trecho $\langle trecho\ 1\rangle$ está concatenado ao $\langle trecho\ 2\rangle$.

```
1 @metaobj
2 func ⟨nome do método exemplo⟩: ⟨parâmetros exemplo⟩ {
3    ...
4 }
```

Código-Fonte 45: Exemplo de código-fonte original

O `metaobj` foi implementado para que durante a compilação, ele crie dois novos método com os mesmos parâmetros e mesmo retorno do método anotado, mas com seus nomes seguidos das letras `ABC` e `DFG`. Durante a compilação esse metaobjeto gera uma nova versão do Código-Fonte 45, que é mostrada no Código-Fonte 46. Os trechos de código ⟨*nome do método exemplo*⟩ e ⟨*parâmetros exemplo*⟩ são usados na nova versão desse código-fonte.

```
1  func ⟨nome do método exemplo⟩: ⟨parâmetros exemplo⟩ {
2     ...
3  }
4
5  func ⟨nome do método exemplo⟩ABC: ⟨parâmetros exemplo⟩ {
6     ...
7  }
8
9  func ⟨nome do método exemplo⟩DFG: ⟨parâmetros exemplo⟩ {
10    ...
11 }
```

Código-Fonte 46: Exemplo do código-fonte gerado por **metaobj**

3.4.1 Sintaxe Genérica de **treplicaAction**

O metaobjeto **treplicaAction** é usado para anotar métodos de Cyan que pertencem a protótipos que estendem `Context`:. Esses métodos são anotados para que representem uma `Action` de Treplica. O Código-Fonte 47 mostra um método anotado com **treplicaAction**, durante a compilação esse método será renomeado, e uma nova versão desse método será criada. Um protótipo que representa esse método como uma `Action` de Treplica também será gerado. O método anotado deve ter uma única palavra-chave (seletor), pois **treplicaAction** não está preparado para tratar métodos com mais palavras-chave.

```
1 object ⟨nome do protótipo⟩ extends Context {
```

```
2    ...
3    @treplicaAction
4    func ⟨nome do método⟩: ⟨tipo do parâmetro 1⟩ ⟨nome do parâmetro 1⟩,
5         ⟨tipo do parâmetro 2⟩ ⟨nome do parâmetro 2⟩,
6         ⟨tipo do parâmetro n⟩ ⟨nome do parâmetro n⟩ {
7      ...
8    }
9  }
```

Código-Fonte 47: Versão original do código-fonte que usa **treplicaAction**

O Código-Fonte 48 mostra o protótipo gerado por **treplicaAction**. O nome desse novo protótipo é gerado com base no nome do contexto do método anotado concatenado com o nome do próprio método. As variáveis de instância desse protótipo são definidas com base nos parâmetros do método anotado concatenadas com `Var`, qualquer número de parâmetros é aceito. O método `init` tem como parâmetro os mesmos parâmetros do método anotado, esse método atribui às variáveis de instância seus parâmetros. O método `executeOn` recebe a instância de `Context` e chama o método ⟨*nome do método*⟩`TreplicaAction`⟨*id único*⟩, passando as variáveis de instância como parâmetros. Esse método é o método anotado que foi renomeado.

```
1  object ⟨⟨nome do protótipo⟩⟨nome do método⟩⟩ extends Action {
2    var ⟨tipo do parâmetro 1⟩ ⟨nome do parâmetro 1⟩Var
3    var ⟨tipo do parâmetro 2⟩ ⟨nome do parâmetro 2⟩Var
4    var ⟨tipo do parâmetro n⟩ ⟨nome do parâmetro n⟩Var
5
6    func init: ⟨tipo do parâmetro 1⟩ ⟨nome do parâmetro 1⟩,
7         ⟨tipo do parâmetro 2⟩ ⟨nome do parâmetro 2⟩,
8         ⟨tipo do parâmetro n⟩ ⟨nome do parâmetro n⟩ {
9      ⟨nome do parâmetro 1⟩Var = ⟨nome do parâmetro 1⟩;
10     ⟨nome do parâmetro 2⟩Var = ⟨nome do parâmetro 2⟩;
11     ⟨nome do parâmetro n⟩Var = ⟨nome do parâmetro n⟩;
12   }
13
14   override
15   func executeOn: Context context {
16     var obj = ⟨nome do protótipo⟩ cast: context;
17     obj ⟨nome do método⟩TreplicaAction⟨id único⟩: ⟨nome do parâmetro 1⟩Var,
           ⟨nome do parâmetro 2⟩Var, ⟨nome do parâmetro n⟩Var;
18   }
19 }
```

Código-Fonte 48: Protótipo gerado por **treplicaAction**

A nova versão do protótipo que estende `Context` é mostrada no Código-Fonte 49. O método ⟨*nome do método*⟩ é renomeado para ⟨*nome do método*⟩`TreplicaAction` ⟨*id único*⟩, o compilador também adiciona ao final do novo nome um complemento (⟨*id único*⟩) para tornar esse nome único dentro do protótipo. Um novo método ⟨*nome do método*⟩ é criado, ele tem os mesmo parâmetros do método original. Esse novo método (⟨*nome do método*⟩) instancia o novo protótipo que foi gerado e chama o método `execute` de treplica passando essa instância que representa uma `Action` de Treplica.

Tomando o Código-Fonte 33 como exemplo concreto dessa abstração de sintaxe, os trechos de código-fonte usados pelo metaobjeto **treplicaAction** seriam: ⟨*nome do protótipo*⟩ representando `Info`, ⟨*nome do método*⟩ representando `setText`, ⟨*tipo do parâmetro* 1⟩ representando `String` e ⟨*nome do parâmetro* 1⟩ representando `text`. Aplicando esses trechos de código-fonte sobre o Código-Fonte 48, é gerado um protótipo igual ao do Código-Fonte 37, e aplicados ao Código-Fonte 49, é gerado uma versão nova do contexto com o método anotado igual a do Código-Fonte 36.

```
1  object ⟨nome do protótipo⟩ extends Context {
2    ...
3    func ⟨nome do método⟩: ⟨tipo do parâmetro 1⟩ ⟨nome do parâmetro 1⟩,
4           ⟨tipo do parâmetro 2⟩ ⟨nome do parâmetro 2⟩,
5           ⟨tipo do parâmetro n⟩ ⟨nome do parâmetro n⟩ {
6      var action = ⟨⟨nome do protótipo⟩⟨nome do método⟩⟩ new:
7        ⟨nome do parâmetro 1⟩, ⟨nome do parâmetro 2⟩, ⟨nome do parâmetro n⟩;
8      self getTreplica execute: action;
9    }
10
11   func ⟨nome do método⟩TreplicaAction⟨id único⟩:
12        ⟨tipo do parâmetro 1⟩ ⟨nome do parâmetro 1⟩,
13        ⟨tipo do parâmetro 2⟩ ⟨nome do parâmetro 2⟩,
14        ⟨tipo do parâmetro n⟩ ⟨nome do parâmetro n⟩ {
15      ...
16   }
17 }
```

Código-Fonte 49: Nova versão do protótipo que contém o método anotado

3.4.2 Sintaxe Genérica de **treplicaInit**

O metaobjeto **treplicaInit** é usado para anotar declarações de variáveis que recebem instâncias de protótipos que estendem `Context`. No Código-Fonte 50 esse metaobjeto anota a declaração da variável ⟨*nome da variável*⟩ que recebe uma instância do tipo ⟨*tipo do contexto*⟩. Essa anotação também recebe os parâmetros ⟨*processos*⟩,

⟨*rtt*⟩ e ⟨*local*⟩ usados para inicializar a máquina da estados de Treplica.

```
1 @treplicaInit( ⟨processos⟩, ⟨rtt⟩, ⟨local⟩ )
2 var ⟨nome da variável⟩ = ⟨tipo do contexto⟩ new;
```

Código-Fonte 50: Versão original do código-fonte que usa **treplicaInit**

O Código-Fonte 51 mostra a nova versão dessa declaração gerada por **treplicaInit**. Uma nova variável é criada para receber a instância do protótipo `Treplica`. Essa instância é inicializada chamando o método `runMachine:`, que recebe os parâmetros associados à anotação. Por último, a variável com a instância do contexto chama o método **setTreplica:** para receber a instância de Treplica associada à variável `treplica`⟨*nome da variável*⟩.

```
1 var ⟨nome da variável⟩ = ⟨tipo do contexto⟩ new;
2 var treplica⟨nome da variável⟩ = Treplica new;
3 treplica⟨nome da variável⟩ runMachine: ⟨nome da variável⟩
4       numberProcess: ⟨processos⟩ rtt: ⟨rtt⟩ path: ⟨local⟩;
5 ⟨nome da variável⟩ setTreplica: treplica⟨nome da variável⟩;
```

Código-Fonte 51: Nova versão do código-fonte de inicialização de Treplica

Tomando o Código-Fonte 34 como exemplo concreto dessa abstração de sintaxe, os trechos de código-fonte usados pelo metaobjeto **treplicaInit** seriam: ⟨*processos*⟩ representando `3`, ⟨*rtt*⟩ representando `200`, ⟨*local*⟩ representando `local`, ⟨*nome da variável*⟩ representando `info` e ⟨*tipo do contexto*⟩ representando `Info`. Aplicando esses trechos de código-fonte sobre o Código-Fonte 51 é gerada uma versão dessa declaração anotada igual ao Código-Fonte 43.

4 Estudos de Caso

A replicação ativa permite a construção de aplicações que compartilham dados entre suas instâncias, sem que seja necessário centralizar essas informações. Em uma aplicação replicada, cada instância possui uma cópia de todos os dados do sistema. Cada instância é uma réplica das demais, e funciona como um histórico das ações realizadas pelo sistema.

Jogos de tabuleiro são exemplos de aplicações que podem ser implementadas usando replicação. Esses jogos possuem diversos elementos e informações que precisam ser compartilhadas entre seus jogadores, como o tabuleiro, as peças, a posição de cada peça, a situação atual da partida. Nesse caso a replicação permite que jogadores usando instâncias diferentes da aplicação compartilhem esses elementos e informações. Esse estudo de caso mostra a implementação de um jogo de tabuleiro, usando Treplica, metaobjetos, e Cyan, no intuito de demonstrar como o uso da metaprogramação permite implementar aplicações replicadas de modo transparente.

Em seguida é mostrada uma aplicação exemplo que, durante a compilação, tem seus métodos não deterministas trocados por métodos deterministas. Para facilitar a demonstração da mecânica de substituição dos métodos foram usados métodos deterministas para representar os métodos não deterministas. Assim, foi possível definir nomes significativos para os métodos do exemplo. Trechos do código-fonte e o arquivo *deterministic* dessa aplicação são mostrados na Sessão 4.2.

4.1 Disputa pela Rota do Leste

O jogo de tabuleiro Disputa pela Rota do Leste é uma batalha entre guerreiros e magos que estão disputando uma rota comercial com os reinos do leste. Ele é uma aplicação desenvolvida em Cyan que deve ser usada por dois jogadores, cada um com uma instância diferente do programa. As instâncias de cada jogador podem ser executadas no mesmo computador, para simplificar a demonstração.

Cada jogador controla um grupo três personagens (dois guerreiros e um mago) e o jogo é baseado em rodadas e ações. Em cada rodada um determinado jogador pode realizar três ações de ataque e movimento. A ação de selecionar personagens pode ser realizada sem limitação durante a rodada. A ação *START* reinicia a partida. Os jogadores não podem executar ações durante a rodada do oponente, e após realizar três ações, a vez é passada para o oponente. Todas as ações são realizadas via um terminal de texto, que recebe os comandos dos jogadores. O jogo termina quando todos os personagens

de um jogador estiverem fora de combate.

A Figura 11 mostra uma instância da aplicação sendo executada. À esquerda é representado o tabuleiro, que é definido pelos símbolos: `.`, `?`, `@`, `#`. O ponto (`.`) representa os espaços por onde os personagens podem se mover. O ponto de interrogação (`?`) representa o mago, e o guerreiro é representado pela arroba (`@`). Os obstáculos são representadas no tabuleiro pelo cardinal (`#`), os personagens não podem se mover passando por eles.

À direita são representadas as informações referentes à partida. São mostrados os jogadores (`Player`), os personagens e suas vidas (`Mage 1`), os pontos de ação disponíveis para a rodada (`Action Points`), e qual o personagem selecionado (`Selected`). As informações correspondentes ao jogador associado à instância da aplicação estão em - `Player 1` - `YOU`, as informações em - `Player 2` - `OPONENT` - são referentes ao jogador que está usando a outra instância da aplicação.

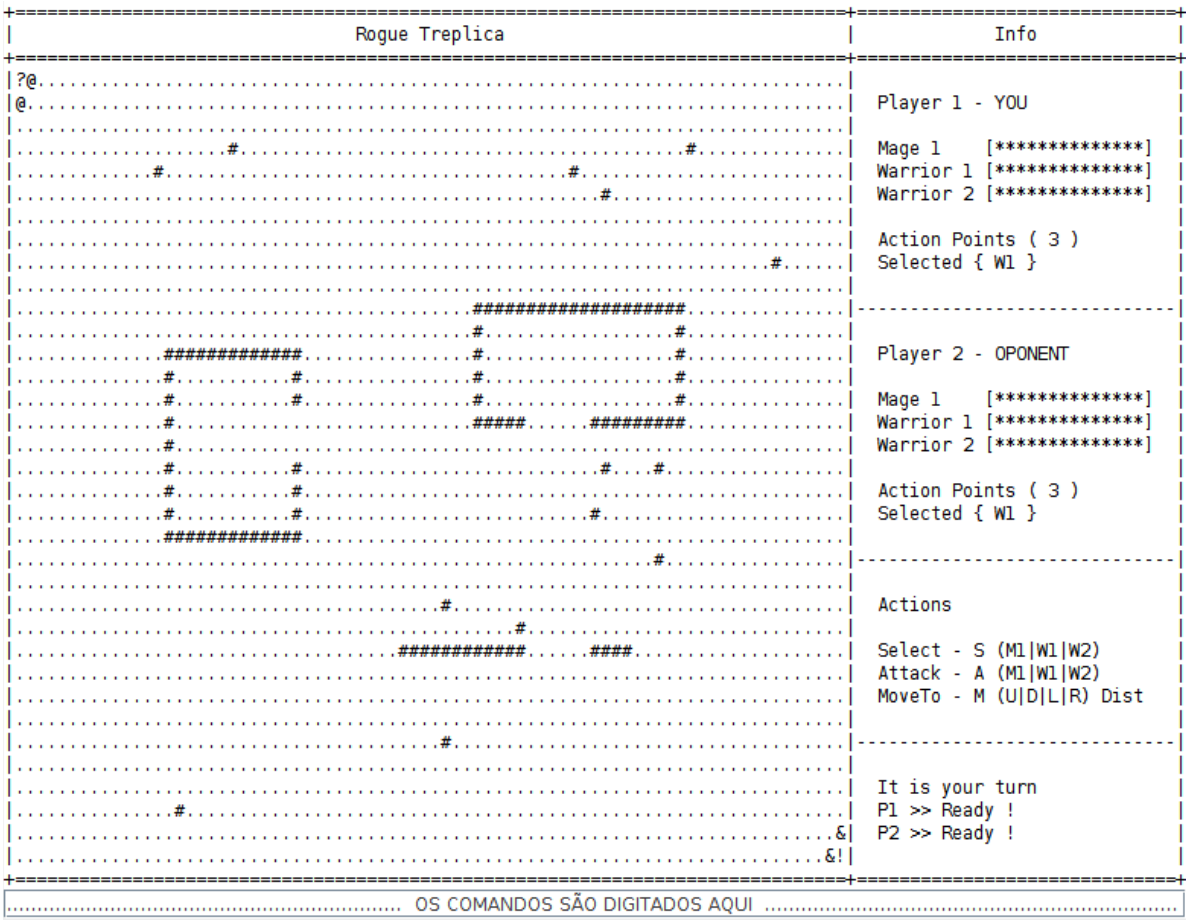

Figura 11: Uma instância da aplicação sendo executada

Ainda à direita da figura, a seção `Actions` mostra como os jogadores devem construir os comandos. A ação de selecionar personagens , `Select`; é formada pela letra `S` seguida do personagem a ser selecionado (`M1`: mago, `W1` e `W2`: guerreiros). A ação

`Attack` é usada para atacar um personagem do oponente, ela é formada pela letra `A` seguida do personagem pertencente ao oponente que será atacado, caso o atacante esteja distante do seu alvo o ataque não ocorrerá.

A última ação disponível é a `MoveTo`, usada para mover um personagem pelo tabuleiro, ela é formada pela letra `M` seguida da direção que o personagem irá se mover (`U`: cima, `D`: baixo, `L`: esquerda, `R`: direita), e da distância que ele irá se deslocar (número de espaços). Caso o movimento não possa ocorrer por causa de um obstáculo ou por causa da distância, uma mensagem será exibida. As ações são realizadas com base no personagem selecionado, é ele quem se movimenta e ataca.

A última seção à direita mostra se a rodada atual pertence ao jogador dessa instância da aplicação (`It is your turn`). Essa seção também exibe a última atualização associada a cada jogador (`P1 » Ready!`), pode mostrar mensagens de alerta, e indicar o resultado das ações desse jogador. Os comandos são digitados pelos jogadores no espaço abaixo do tabuleiro e das informações, no local indicado na Figura 11.

4.1.1 Implementação do Protótipo de um Tabuleiro Compartilhado

Para implementar esse jogo de tabuleiro foram desenvolvidos quatorze protótipos. O protótipo `Board` é de maior importância. Ele é quem define o contexto (`Context`) e as ações (`Action`) de Treplica. Os outros protótipos são usados para separar os elementos da aplicação e tornar o código melhor organizado. Podemos considerar o protótipo `Board` como o centro da aplicação, pois ele é quem acaba integrando os outros protótipos.

O protótipo `Entity` (Código-Fonte 52) define dois métodos: o método `getType`, que retorna o tipo da entidade, e o método `draw` usado para desenhar a entidade no mapa. Esse protótipo é herdado pelos protótipos `Figure` e `Tile` que sobrescrevem seus métodos. Desse modo o protótipo `Map` pode usar os métodos de `Entity` para acessar tanto as instâncias de `Figure`, como de `Tile`. Isso permite que o protótipo `Map` mantenha uma única matriz do tipo `Entity` contendo instâncias de ambos os protótipos.

```
1 package main
2
3 object Entity
4   func getType -> Int { return Const ENTITY; }
5   func draw { }
6 end
```

Código-Fonte 52: Protótipo **Entity** herdado por *Figure* e *Tile*

O protótipo **Figure** representa uma peça do tabuleiro, um mago ou um guerreiro. Ele possui métodos como **setPosition** para marcar a posição da peça no mapa, **setLife** para modificar os pontos de vida da peça quando ela é atacada, **getDistanceMax** para verificar a distância máxima que a peça pode ser deslocada com um movimento. Os elementos que compõem o cenário como o chão e os obstáculos por sua vez são representados pelo protótipo **Tile**. Métodos como **draw** e **getType** são implementados por ambos os protótipos, **Figure** e **Tile**, uma vez que eles herdam de **Entity**.

O mapa com todas as peças e partes do cenário é mantido pelo protótipo **Board** como parte do contexto de Treplica, definido o estado a ser replicado. O protótipo **Board** mantém um vetor de instâncias do tipo **Player** e a variável **currentPlayer**, que é uma referência para o **Player** que tem direito a jogar na rodada atual. O protótipo **Board** define quatro ações de Treplica: **startAction**, chamada para reiniciar o jogo; **moveAction**, usada pelos jogadores para mover as peças pelo mapa; **attackAction**, chamada para que uma peça ataque um mago ou guerreiro inimigo; e **selectAction**, que permite aos jogadores selecionarem qual peça eles querem que se mova ou ataque. O Código-Fonte 53 mostra a implementação parcial do protótipo **Board** que usa a anotação do metaobjeto **@treplicaAction** para marcar os métodos que serão as ações de Treplica.

```
1  package main
2
3  import treplica
4  import cyan.math
5
6  object Board extends Context
7    var Map map
8    var Array< Player > players
9    var String currentPlayer
10   ...
11   @treplicaAction
12   func selectAction: String target { ... }
13
14   @treplicaAction
15   func attackAction: String target { ... }
16
17   @treplicaAction
18   func moveAction: String direction, String value { ... }
19   ...
20   @treplicaAction
21   func startAction { ... }
22   ...
23 end
```

Código-Fonte 53: Protótipo **Board** que integra a aplicação

Não se pode deixar de mencionar o protótipo **Program** mostrado no Código-Fonte 54. Ele usa a anotação do metaobjeto **@treplicaInit** para iniciar a máquina de estados de Treplica e implementa o método **event**, que recebe os comandos dos jogadores e chama os métodos replicados do contexto.

O *loop* principal da aplicação está implementado no protótipo **Window**. Esse protótipo aguarda comandos do jogador na forma de entrada via teclado (comando) e chama o método **event** de **Program** passando essa entrada. Note que os métodos de **local** que representam ações de Treplica também são chamados pela sua máquina de estados. Em ambos os casos a tela do jogo é atualizada após cada chamada de uma ação. Alguns protótipos usados para implementar essa aplicação não foram detalhados por não serem relevantes para a compreensão da implementação desse estudo de caso.

```
1  package main
2  import treplica
3
4  object Program extends Input
5    ...
6    override func event: String text {
7      ...
8      local selectAction: temp1;
9      ...
10     local attackAction: temp1;
11     ...
12     local moveAction: temp1, temp2;
13     ...
14   }
15
16   func run: Array<String> args {
17     Window build: self;
18
19     var local = ("/var/tmp/magic" ++ args[1]);
20     @treplicaInit( 2, 200, local )
21     var data = Board new: args[1];
22     data startBoard;
23     ...
24   }
25 end
```

Código-Fonte 54: Protótipo **Program** do Jogo de Tabuleiro

4.2 Somente os Métodos Deterministas Compilaram

As aplicações replicadas baseadas em máquinas de estado têm como uma de suas principais características o determinismo de suas ações. Uma ação que será replicada deve ser determinista, caso seu comportamento fuja a essa regra ela pode tornar a aplicação inconsistente.

O metaobjeto **`treplicaAction`** é responsável por transformar métodos tradicionais em ações de Treplica e por verificar se esses métodos apresentam algum caso de não determinismo cadastrado. Nesse estudo de caso é definida uma ação de Treplica, que possui um método cadastrado como não determinista, para demonstrar como a compilação dessa aplicação localiza e trata essa inconsistência.

Essa aplicação é composta de dois protótipos. Um é similar ao protótipo definido no Código-Fonte 34 que inicializa a máquina de estados de Treplica. O outro é o protótipo do Código-Fonte 55, que contém o contexto associado a essa máquina de estados e à ação não determinista implementada pelo método **`nondeterministic`**.

```
1  package main
2
3  import treplica
4  import cyan.math
5
6  object Tested extends Context
7    var String result
8    func init { result = "hello"; }
9
10   @treplicaAction
11   func perform {
12     nondeterministic: (setResult: "bye");
13     Out println: "perform: " ++ self.result;
14   }
15
16   func setResult: String value -> String {
17     self.result = value;
18     return value;
19   }
20
21   func nondeterministic: String value {
22     Out println: "nondeterministic: " ++ value ++ " random: " ++ Math
           random;
23   }
24 end
```

Código-Fonte 55: Contexto com ação não determinista

Para que o método **nondeterministic** seja reconhecido como não determinista, deve ser cadastrado no arquivo *deterministic* da biblioteca Treplica de Cyan ou no arquivo *deterministic* da própria aplicação. Nessa caso o cadastro do método está no pacote de Treplica, como mostrado no Código-Fonte 56. Essa regra define que o método **nondeterministic** do protótipo **Tested** deve ser substituído pelo método **testok** do protótipo **Deterministic**. O método que substitui o caso não determinista deve ter os mesmos parâmetros do método substituído.

```
1 main,Tested,nondeterministic-treplica,Deterministic,testok
```

Código-Fonte 56: Regra de substituição cadastrada

Dessa forma, durante a compilação o método **nondeterministic** será substituído pelo método **testok**, e a ocorrência de não determinismo será removida pelo metaobjeto **treplicaAction**. A compilação também emite um alerta ao desenvolvedor a respeito do não determinismo que foi substituído. Caso o protótipo ou o método que remove o não determinismo não seja encontrado, a compilação falhará. Nesses casos um erro de compilação é mostrado como resultado.

O protótipo **Deterministic** é mostrado no Código-Fonte 57. O método **testok** implementado possui os mesmos parâmetros do método **nondeterministic** e a chamada ao método **random** do protótipo **Math** foi substituída pela *string* **23**. O método realmente não determinista é o **random**, ele poderia ter sido substituído diretamente, como no exemplo da sessão 3.3.2. Para mostrar a substituição de um método com parâmetros foi necessário encapsular o método não determinista dentro do método **nondeterministic**.

```
1  package treplica
2
3  object Deterministic
4
5    func random -> Double {
6      return 23.0;
7    }
8
9    func testok: String value {
10     Out println: "testok: " ++ value ++ " random: " ++ "23";
11   }
12 end
```

Código-Fonte 57: Protótipo com método determinista

Se o método **random** do protótipo **Math** fosse substituído diretamente pelo método **random** do protótipo **Deterministic**, a regra cadastrada seria igual à mostrada no Código-Fonte 58, e nesse caso não existem parâmetros a serem passados ao novo método. Os protótipos implementados para remover o não determinismo devem pertencer aos pacotes **treplica** ou **main**. O metaobjeto **treplicaAction** pode ser melhorado para que esses protótipos possam pertencer a qualquer pacote.

```
1 cyan.math,Math,random-treplica,Deterministic,random
```

Código-Fonte 58: Regra de substituição cadastrada

Repare que, no caso da primeira regra, a busca por métodos não deterministas inicia no método **perform** e termina após substituir o método **nondeterministic**. No caso da segunda regra, a busca se propaga recursivamente aos métodos internos do método inicial. Essa segunda busca percorre o método **perform** onde encontra o método **nondeterministic**, esse método encontrado também é percorrido na procura do método **random** do protótipo **Math**. Se a chamada de método que for substituída possuir parâmetros, esses serão mantidos na chamada do novo método.

Conclusão

A metaprogramação foi usada nessa pesquisa para transformar aplicações não distribuídas em aplicações replicadas de modo transparente. O *framework* Treplica forneceu a implementação da replicação para essas aplicações e a metaprogramação foi suportada pela linguagem Cyan através de seus metaobjetos.

Essa abordagem mostrou como implementar aplicações que sejam menos acopladas à replicação, e como manter seu código mais coeso. Essa afirmação pode ser verificada com base na comparação das implementações de aplicações replicadas que usaram da metaprogramação, em relação às aplicações replicadas que não usam de metaprogramação. É possível comparar os exemplos da Seção 3.1 com os da Seção 3.2 e verificar que o segundo exemplo sofreu uma redução do código fonte em comparação ao primeiro. No segundo exemplo a replicação não exigiu que novos protótipos fossem criados explicitamente, ou que a máquina de Treplica fosse inicializada diretamente. Sua implementação ficou restrita a marcar algumas estruturas do programa que foi replicado. O código-fonte mostrado na Seção 3.2 é mais coeso aos requisitos funcionais do programa do que o código fonte mostrado na Seção 3.1. O acoplamento da replicação com o restante do programa é menor na Seção 3.2 do que o acoplamento presente no código fonte da Seção 3.1.

A metaprogramação também foi usada nessa pesquisa para validar o código-fonte das aplicações replicadas durante sua compilação, com o objetivo de encontrar trechos de código não deterministas. As Seções 3.3 e 4.2 mostram como a metaprogramação foi usada para realizar essa validação. O problema de não determinismo apresentado na Seção 3.1.1 é um exemplo de inconsistência que não deve estar presente nas aplicações replicadas, pois isso levaria a um comportamento inconsistente da aplicação.

O uso da metaprogramação permitiu que as tarefas de desenvolver e verificar o código-fonte fossem automatizadas. Os códigos-fonte que implementam requisitos não funcionais facilitam essa automatização por serem mais padronizados do que dos demais códigos-fonte. Isso não impede que requisitos funcionais possam ser implementados usando metaprogramação, só é menos comum encontrar um requisito funcional que possui um código-fonte padronizado em diferentes implementações.

Abraçando uma visão menos conservadora é possível comparar os metaprogramas a um programador especializado em trabalhar com determinado requisito não funcional, que irá auxiliar as equipes de desenvolvimento a implementar esse requisito e a verificar se suas limitações e exigências são atendidas pelo restante do programa.

A construção de uma base de metaprogramas que resolva a implementação

dos requisitos não funcionais mais corriqueiros pode reduzir o tempo de desenvolvimento, padronizar a implementação desses requisitos e reduzir o tempo de criação das aplicações. Indiretamente ela também permite que o nível de conhecimento específico da equipe a respeito de determinados requisitos não seja tão elevado, o que diminui a necessidade de treinamento da equipe e estudos preliminares à implementação das aplicações.

Construir metaobjetos para automatizar o desenvolvimento e inspecionar requisitos não funcionais diferentes da replicação, não é a única extensão possível a essa pesquisa. Ela pode ser estendida para medir de modo formal como a metaprogramação melhora a coesão e reduz o acoplamento do código-fonte. O trabalho de Hitz e Montazeri (HITZ; MONTAZERI, 1995) fornece uma alternativa de como medir a coesão e o acoplamento do código-fonte de modo quantitativo. Sendo assim, ele pode ser usado como suporte a um possível trabalho futuro.

Uma linha de pesquisa mais teórica é a classificação de quais mecanismos de atualização e verificação são comuns entre diferentes requisitos não funcionais. Isso ajudaria a desenvolver metaobjetos que possam ser reutilizados com diferentes requisitos não funcionais. O modo como esses requisitos influenciam o código-fonte também poderia ser analisado.

Nessa pesquisa ficou em aberto a modificação do metaobjeto **treplicaAction** para que a verificação do não determinismo atenda a todos os códigos-fonte sem as limitações deixadas por essa pesquisa. O arquivo de regras usado nessa validação também poderia ser substituído por uma anotação de Cyan.

A linguagem Xtend aparece nesse trabalho para servir de comparação com a linguagem Cyan. Em novas pesquisas, outras linguagens que permitem metaprogramação em compilação podem ser estudadas para identificar quais características fornecidas por elas permitem desenvolver código-fonte mais coeso e menos acoplado. Uma comparação detalhada entre essas linguagens seria importante para classificar quais características são mais versáteis na automação do desenvolvimento e na verificação do código-fonte independente de requisitos não funcionais específicos.

Do ponto de vista teórico visando compreender, classificar e propor melhorias à metaprogramação em compilação, ou do ponto de vista prático de aplicar a metaprogramação em compilação a requisitos não funcionais como no caso da replicação, essa pesquisa pode ser usada como uma referência para futuros trabalhos que pretendem tornar o desenvolvimento do código-fonte mais automático e correto por meio do uso da metaprogramação em compilação.

Publicações

Durante a realização dessa pesquisa foi submetido um artigo ao Simpósio Brasileiro de Linguagens de Programação (SBLP), que aconteceu em Fortaleza (Ceará, Brasil). Nesse artigo é mostrada a automatização do desenvolvimento do código-fonte. Ele não trata da verificação desse código-fonte. A automatização foi aplicada em programas distribuídos que foram desenvolvidos em Cyan usando Treplica.

- UGLIARA. F. A.; VIEIRA G. M.; GUIMARÃES J. O.. "Transparent Replication Using Metaprogramming in Cyan". Simpósio Brasileiro de Linguagens de Programação (*SBLP 2017*). Fortaleza, Ceará, Brasil. Setembro, 2017.

Endereços Eletrônicos

O código-fonte desenvolvido nessa pesquisa está disponível na internet em um repositório de projetos. Caso seja necessário reproduzir essa pesquisa, o ponto de partida é esse repositório. Para os entusiastas do LaTeX, também estão disponíveis: o código-fonte dessa dissertação, e o código-fonte da apresentação usada na defesa desse trabalho. Os arquivos README.md do repositório contêm detalhes de como executar esse projeto.

- Dissertação, apresentação, compilador e códigos-fonte:
  https://bitbucket.org/fellipe-ugliara/mestrado/src/master/

Referências

AHO, A. V.; ULLMAN, J. D. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1977. Citado na página 5.

ALTINBÜKEN, D.; SIRER, E. G. *Commodifying replicated state machines with OpenReplica*. [S.l.], 2012. Citado 2 vezes nas páginas 34 e 37.

ANTON, A. I. Goal identification and refinement in the specification of software-based information systems. Georgia Institute of Technology, 1997. Citado na página 37.

BLEWITT, A.; BUNDY, A.; STARK, I. Automatic verification of design patterns in Java. In: ACM. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2005. p. 224–232. Citado 2 vezes nas páginas 22 e 38.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. *Introduction to Reliable and Secure Distributed Programming*. [S.l.]: Springer, 2011. ISBN 3642152597. Citado 2 vezes nas páginas 2 e 30.

CHLIPALA, A. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2013. v. 48, n. 9, p. 391–402. Citado na página 38.

CHOMSKY, N. Logical syntax and semantics: Their linguistic relevance. *Language*, JSTOR, v. 31, n. 1, p. 36–45, 1955. Citado na página 5.

COPELAND, J. The church-turing thesis. *NeuroQuantology*, v. 2, n. 2, 2007. Citado na página 1.

DAMAŠEVIČIUS, R.; ŠTUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, v. 37, n. 2, 2015. Citado 3 vezes nas páginas 2, 7 e 9.

DERSHEM, H. L.; JIPPING, M. J. *Programming languages: structures and models*. [S.l.]: ITP, 1995. Citado na página 5.

EDER, J.; KAPPEL, G.; SCHREFL, M. *Coupling and cohesion in object-oriented systems*. [S.l.], 1994. Citado na página 37.

FILMAN, R. E.; HAVELUND, K. Source-code instrumentation and quantification of events. *NASA Form*, NASA Ames Research Center, n. 1676, 2002. Citado na página 39.

FISCHER, A. E.; GRODZINSKY, F. S. *The anatomy of programming languages*. [S.l.]: Prentice Hall, 1993. Citado na página 5.

FOWLER, M. *Domain-specific languages*. [S.l.]: Pearson Education, 2010. Citado na página 1.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2. Citado na página 14.

GLINZ, M. On non-functional requirements. In: IEEE. *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*. [S.l.], 2007. p. 21–26. Citado 3 vezes nas páginas 2, 36 e 37.

GUIMARÃES, J. d. O. *The Cyan Language*. [S.l.], 2018. Disponível em: <http://cyan-lang.org/wp-content/uploads/2018/02/The-Cyan-Language.pdf>. Citado 2 vezes nas páginas 3 e 10.

GUIMARÃES, J. d. O. *The Cyan Language Metaobject Protocol*. [S.l.], 2018. Disponível em: <http://cyan-lang.org/the-cyan-mop/>. Citado na página 13.

HITZ, M.; MONTAZERI, B. Measuring coupling and cohesion in object-oriented systems. na, 1995. Citado 2 vezes nas páginas 37 e 74.

IEEE. Standard glossary of software engineering terminology. In: . [S.l.], 1990. Citado na página 36.

KICZALES, G. et al. Aspect-oriented programming. In: SPRINGER. *European conference on object-oriented programming*. [S.l.], 1997. p. 220–242. Citado na página 2.

KOENIG, D. et al. *Groovy in action*. [S.l.]: Manning Publications Co., 2007. Citado na página 10.

LAMPORT, L. Fast Paxos. *Distributed Computing*, Springer, v. 19, n. 2, p. 79–103, 2006. Citado na página 39.

MEKRUKSAVANICH, S.; YUPAPIN, P. P.; MUENCHAISRI, P. Analytical learning based on a meta-programming approach for the detection of object-oriented design defects. *Information Technology Journal*, Asian Network for Scientific Information (ANSINET), v. 11, n. 12, p. 1677, 2012. Citado na página 38.

MIAO, W.; SIEK, J. Compile-time reflection and metaprogramming for Java. In: ACM. *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. [S.l.], 2014. p. 27–37. Citado na página 22.

RENTSCHLER, A. et al. Designing information hiding modularity for model transformation languages. In: ACM. *Proceedings of the 13th International Conference on Modularity*. [S.l.], 2014. p. 217–228. Citado 2 vezes nas páginas 10 e 21.

ROST, R. J. et al. *OpenGL shading language*. [S.l.]: Pearson Education, 2009. Citado na página 1.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, ACM, v. 22, n. 4, p. 299–319, 1990. Citado na página 30.

SIPSER, M. *Introduction to the Theory of Computation*. [S.l.]: Thomson Course Technology Boston, 2006. Citado na página 1.

TOURWÉ, T.; MENS, T. Identifying refactoring opportunities using logic meta programming. In: IEEE. *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. [S.l.], 2003. p. 91–100. Citado na página 39.

VIEIRA, G. M. D.; BUZATO, L. E. Treplica: Ubiquitous replication. In: *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*. Rio de Janeiro, Brasil: [s.n.], 2008. Disponível em: <http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=7450>. Citado na página 3.

VIEIRA, G. M. D.; BUZATO, L. E. *Implementation of an Object-Oriented Specification for Active Replication Using Consensus*. [S.l.], 2010. Disponível em: <http://www.ic.unicamp.br/~reltech/2010/10-26.pdf>. Citado 2 vezes nas páginas 30 e 39.

WIESMANN, M. et al. Understanding replication in databases and distributed systems. In: IEEE. *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. [S.l.], 2000. p. 464–474. Citado na página 3.

APÊNDICE A – Código Fonte dos Metaobjetos

```
1  package meta.treplica;
2  ...
3
4  public class CyanMetaobjectTreplicaAction extends CyanMetaobjectWithAt implements
       IActionProgramUnit_ati, IAction_dsa {
5    ICompiler_dsa compiler;
6    Vector<Pair<Triple<String, String, String>, Triple<String, String, String>>> methodsND;
7      Program program;
8      String fixName;
9    Vector<MethodDec> visited;
10   HashMap<String, Set<ProgramUnit>> mapProtoSubProto;
11   Vector<String> loadedPack;
12
13   public CyanMetaobjectTreplicaAction() {
14     super(MetaobjectArgumentKind.ZeroParameter);
15     visited = new Vector<MethodDec>();
16     methodsND = new Vector<Pair<Triple<String, String, String>, Pair<String, String>>>();
17     loadedPack = new Vector<String>();
18   }
19
20     public static String substringBeforeLast(String str, String separator) {
21         int pos = str.lastIndexOf(separator);
22         if (pos == -1) {
23           return str;
24         }
25         return str.substring(0, pos);
26     }
27
28   public void actionNonDeterminism(ExprMessageSend node, MethodDec method, String pack,
       String func) {
29         loadTreplicaNd(method.getDeclaringObject().getCompilationUnit().getPackageName());
30     func = func.replaceAll(":", "");
31     String parameters = "";
32     if (node != null) {
33       String [] splitParam = node.asString().split(":", 2);
34       if(splitParam.length == 2) {
35         parameters = ":" + splitParam[1];
36       }
37     }
38     Pair<String, String> left = new Pair<String, String>(pack, func);
39     Triple<String, String, String> right = null;
40     for (Pair<Triple<String, String, String>, Triple<String, String, String>> pairFunc :
         methodsND) {
41       if(!pairFunc.getKey().getLeft().equals(substringBeforeLast(method.getDeclaringObject
           ().getFullName(), "."))) ) {
42         break;
43       }
44       if (pairFunc.getKey().getMiddle().equals(left.getKey())
45           && pairFunc.getKey().getRight().equals(left.getValue())) {
```

```
46          right = pairFunc.getValue();
47          break;
48        }
49      }
50      if (right != null) {
51
52        StringBuffer codeToAdd = new StringBuffer(right.getLeft() + "." + right.getMiddle() +
              " " + right.getRight().trim() + parameters );
53        if (node != null) {
54          compiler.removeAddCodeExprMessageSend(node, this.getMetaobjectAnnotation(),
                codeToAdd, node.getType());
55        }
56      }
57    }
58
59    public boolean visitedContain(MethodDec item) {
60      String packItem = item.getDeclaringObject().getCompilationUnit().getNamePublicPrototype
            ();
61      String funcItem = item.getNameWithoutParamNumber();
62
63          for (MethodDec method : visited) {
64        String packTemp = method.getDeclaringObject().getCompilationUnit().
              getNamePublicPrototype();
65        String funcTemp = method.getNameWithoutParamNumber();
66
67        if (packTemp.equals(packItem) && funcTemp.equals(funcItem)) {
68          return true;
69        }
70      }
71
72      return false;
73    }
74
75    public void visitorWalk(ExprMessageSend node, String pack, String func) {
76      Pair<String, String> methodPair = new Pair<String, String>(pack, func);
77      for (MethodDec method : findMethod(methodPair)) {
78        if (!visitedContain(method)) {
79          actionNonDeterminism(node, method, pack, func);
80          visitorFunctionsTree(method);
81        }
82      }
83    }
84
85    public void visitorFunctionsTree(MethodDec method) {
86      visited.addElement(method);
87      method.accept(new ASTVisitor() {
88        @Override
89        public void visit(ExprMessageSendWithSelectorsToExpr node) {
90          if(node.getReceiverExpr() != null){
91            if(node.getReceiverExpr().getType() != null){
92              String pack = node.getReceiverExpr().getType().getName();
93              String func = node.getMessage().getMethodName();
94              visitorWalk(node, pack, func);
95            }
96          }
97        }
98
99        @Override
100       public void visit(ExprIdentStar node) {
101         if (node.getIdentStarKind() == IdentStarKind.unaryMethod_t) {
```

```
102           String pack = ((CompilationUnit) node.getFirstSymbol().getCompilationUnit())
103               .getNamePublicPrototype();
104           String func = node.getName();
105           visitorWalk(null, pack, func);
106         }
107       }
108
109       @Override
110       public void visit(ExprMessageSendUnaryChainToExpr node) {
111         if( node.getReceiver().getType() != null){
112         String pack = node.getReceiver().getType().getName();
113         String func = node.getMessageName();
114         visitorWalk(node, pack, func);
115         }
116       }
117
118       @Override
119       public void visit(ExprMessageSendWithSelectorsToSuper node) {
120         String pack = node.getSuperobject().getType().getName();
121         String func = node.getMessage().getMethodName();
122         visitorWalk(node, pack, func);
123       }
124
125       @Override
126       public void visit(ExprMessageSendUnaryChainToSuper node) {
127         String pack = node.getReceiver().getType().getName();
128         String func = node.getMessageName();
129         visitorWalk(node, pack, func);
130       }
131     });
132   }
133
134   public ArrayList<MethodDec> findMethod(Pair<String, String> name) {
135     ArrayList<MethodDec> list = new ArrayList<MethodDec>();
136     for (CyanPackage pack : program.getPackageList()) {
137       if (!pack.getPackageName().equals("cyan.lang")) {
138         for (CompilationUnit compilerUnit : pack.getCompilationUnitList()) {
139           for (ProgramUnit programUnit : compilerUnit.getProgramUnitList()) {
140             Set<ProgramUnit> subProtoList = this.mapProtoSubProto
141                 .get(pack.getPackageName() + " " + programUnit.getName());
142
143                       for (ProgramUnit subProt : subProtoList) {
144               ObjectDec dec = (ObjectDec) subProt;
145               for (MethodDec method : dec.getMethodDecList()) {
146                 if (method.getNameWithoutParamNumber().equals(name.getValue())) {
147                   list.add(method);
148                 }
149               }
150             }
151
152             ObjectDec dec = (ObjectDec) programUnit;
153             for (MethodDec method : dec.getMethodDecList()) {
154               if (method.getNameWithoutParamNumber().equals(name.getValue())) {
155                 list.add(method);
156               }
157             }
158           }
159         }
160       }
161     }
```

```
162     return list;
163   }
164
165   @Override
166   public String getName() {
167     return "treplicaAction";
168   }
169
170   @Override
171   public DeclarationKind[] mayBeAttachedList() {
172     return decKindList;
173   }
174
175   private static DeclarationKind[] decKindList = new DeclarationKind[] { DeclarationKind.
          METHOD_DEC };
176
177   @Override
178   public ArrayList<Tuple2<String, StringBuffer>> ati_codeToAddToPrototypes(ICompiler_ati
          compiler) {
179     MethodDec md = (MethodDec) this.getAttachedDeclaration();
180     String methodName = md.getNameWithoutParamNumber();
181     methodName = methodName.replaceAll(":", "");
182     String funcPar = "";
183     String hasFunPar = " ";
184     String callPar = "";
185     String hasCallPar = "";
186     String funcRet = "";
187     String hasCallRet = "";
188     String retStatement = "";
189
190         for (ParameterDec par : md.getMethodSignature().getParameterList()) {
191       funcPar += " " + par.getType().getName() + " " + par.getName() + ",";
192       callPar += " " + par.getName() + ",";
193     }
194
195     if (md.getMethodSignature().getReturnTypeExpr() != null) {
196       hasCallRet = " -> ";
197       funcRet = md.getMethodSignature().getReturnTypeExpr().getType().getName();
198       retStatement = "@javacode<<<\n" + "return ("
199           + md.getMethodSignature().getReturnTypeExpr().getType().getJavaName() + ")_ret;\n
                " + ">>>\n";
200     }
201
202     if (funcPar.length() > 0) {
203       funcPar = funcPar.substring(0, funcPar.length() - 1);
204       hasFunPar = ": ";
205     }
206
207     if (callPar.length() > 0) {
208       callPar = callPar.substring(0, callPar.length() - 1);
209       hasCallPar = ": ";
210     }
211
212     String objectName = compiler.getCurrentPrototypeName();
213     String prototypeName = NameServer.removeQuotes(objectName + methodName);
214     StringBuffer code = new StringBuffer("\n\n" + "func " + methodName + hasFunPar +
          funcPar + hasCallRet + funcRet
215         + " {\n" + "var action = " + prototypeName + " new" + hasCallPar + callPar + ";\n"
216         + "var Dyn ret = getTreplica execute: action;\n" + retStatement + "}\n");
217     ArrayList<Tuple2<String, StringBuffer>> result = new ArrayList<>();
```

```
218     result.add(new Tuple2<String, StringBuffer>(objectName, code));
219     return result;
220   }
221
222   public ArrayList<Tuple3<String, String, String[]>> ati_renameMethod(ICompiler_ati
          compiler_ati) {
223     String prototypeName = compiler_ati.getCurrentPrototypeName();
224     MethodDec md = (MethodDec) this.getAttachedDeclaration();
225     String oldMethodName = md.getName();
226     String newMethodNameWithPar[] = new String[1 + md.getMethodSignature().getParameterList
            ().size()];
227     String auxName = md.getNameWithoutParamNumber();
228     auxName = auxName.replaceAll(":", "");
229     newMethodNameWithPar[0] = auxName + "TreplicaAction" + this.fixName;
230         int indexArrayName = 1;
231
232         for (ParameterDec par : md.getMethodSignature().getParameterList()) {
233             newMethodNameWithPar[indexArrayName] = par.getName();
234             indexArrayName++;
235         }
236
237     ArrayList<Tuple3<String, String, String[]>> tupleList = new ArrayList<>();
238     tupleList.add(new Tuple3<String, String, String[]>(prototypeName, oldMethodName,
            newMethodNameWithPar));
239     return tupleList;
240   }
241
242   public ArrayList<Tuple2<String, StringBuffer>> ati_NewPrototypeList(ICompiler_ati
          compiler_ati) {
243     MethodDec md = (MethodDec) this.getAttachedDeclaration();
244     String methodName = md.getNameWithoutParamNumber();
245     methodName = methodName.replaceAll(":", "");
246     String varDef = "";
247     String funcPar = "";
248     String hasFuncPar = " ";
249     String funcAssign = "";
250     String callPar = "";
251     String hasCallPar = "";
252     String retStatement = "";
253     String retNullState = "";
254     for (ParameterDec par : md.getMethodSignature().getParameterList()) {
255       varDef += "var " + par.getType().getName() + " " + par.getName() + "Var\n";
256       funcPar += " " + par.getType().getName() + " " + par.getName() + ",";
257       funcAssign += par.getName() + "Var = " + par.getName() + ";\n";
258       callPar += " " + par.getName() + "Var,";
259     }
260
261     if (md.getMethodSignature().getReturnTypeExpr() != null) {
262       retStatement = "return ";
263     } else {
264       retNullState = "return Nil;\n";
265     }
266
267     if (funcPar.length() > 0) {
268       funcPar = funcPar.substring(0, funcPar.length() - 1);
269       hasFuncPar = ": ";
270     }
271
272     if (callPar.length() > 0) {
273       callPar = callPar.substring(0, callPar.length() - 1);
```

```
274       hasCallPar = ": ";
275     }
276
277     String objectName = compiler_ati.getCurrentPrototypeName();
278     ArrayList<Tuple2<String, StringBuffer>> protoCodeList = new ArrayList<>();
279     String prototypeName = NameServer.removeQuotes(objectName + methodName);
280
281         this.fixName = NameServer.nextLocalVariableName();
282     StringBuffer code = new StringBuffer(
283         "package main\n" + "\n" + "import treplica\n" + "\n" + "object " + prototypeName +
                " extends Action\n"
284             + "\n" + varDef + "\n" + "func init" + hasFuncPar + funcPar + " {\n" +
                    funcAssign + "}\n" + "\n"
285             + "override\n" + "func executeOn: Context context -> Dyn {\n" + "var obj = Cast
                    <" + objectName
286             + "> asReceiver: context;\n" + retStatement + "obj " + methodName + "
                    TreplicaAction" + this.fixName
287                     + hasCallPar + callPar + ";\n" + retNullState + "}\n" + "end\n"
288
289     );
290     String unescape = Lexer.unescapeJavaString(code.toString());
291     protoCodeList.add(new Tuple2<String, StringBuffer>(prototypeName, new StringBuffer(
            unescape)));
292     return protoCodeList;
293   }
294
295   private void loadTreplicaNd(String packName) {
296     if (!loadedPack.contains(packName)) {
297         loadedPack.add(packName);
298         Tuple2<FileError, char[]> fileList = this.compiler.getEnv().getProject().
                getCompilerManager()
299             .readTextDataFileFromPackage("deterministic", packName);
300         if (fileList.f2 != null) {
301           String rawText = new String(fileList.f2);
302           String lines[] = rawText.split("\n");
303           for (String line : lines) {
304             String parts[] = line.split("-");
305             String left[] = parts[0].split(",");
306             String right[] = parts[1].split(",");
307             Triple<String, String, String> pLeft = Triple.of(left[0], left[1], left[2]);
308             Triple<String, String, String> pRight = Triple.of(right[0], right[1], right[2])
                    ;
309             methodsND.add(0, new Pair<Triple<String, String, String>, Triple<String, String
                    , String>>(pLeft, pRight));
310           }
311         }
312       }
313   }
314
315   @Override
316   public StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa) {
317     this.compiler = compiler_dsa;
318     this.loadTreplicaNd("treplica");
319     program = compiler.getEnv().getProject().getProgram();
320     Declaration dec = this.getMetaobjectAnnotation().getDeclaration();
321     if (!(dec instanceof MethodDec)) {
322       this.addError("Internal error: metaobject '" + getName() + "' should be attached to a
              method");
323       return null;
324     }
```

```
325         mapProtoSubProto = compiler.getMapPrototypeSubtypeList();
326         MethodDec method = (MethodDec) dec;
327         visitorFunctionsTree(method);
328         return null;
329     }
330 }
```

Código-Fonte 59: Código Fonte de **CyanMetaobjectTreplicaAction**

```
1  package meta.treplica;
2  ...
3
4  public class CyanMetaobjectTreplicaInit extends CyanMetaobjectWithAt
5      implements IActionVariableDeclaration_dsa {
6    ...
7    @Override
8    public String getName() {
9      return "treplicaInit";
10   }
11
12   private static DeclarationKind[] decKindList = new DeclarationKind[] { DeclarationKind.
         LOCAL_VAR_DEC };
13
14   @Override
15   public StringBuffer dsa_codeToAddAfter() {
16     CyanMetaobjectWithAtAnnotation withAt = (CyanMetaobjectWithAtAnnotation) this.
           getMetaobjectAnnotation();
17     Declaration dec = withAt.getDeclaration();
18     StatementLocalVariableDecList varList = (StatementLocalVariableDecList) dec;
19
20     ArrayList<Object> parameterList = withAt.getJavaParameterList();
21     String []strList = new String[parameterList.size()];
22     int i = 0;
23     for ( Object elem : parameterList ) {
24       if ( (elem instanceof Integer) ) {
25         strList[i] = Integer.toString(((Integer) elem).intValue());
26       }else if ( (elem instanceof String) ) {
27         strList[i] = (String) elem;
28       }
29       ++i;
30     }
31
32     String nameTreplicaVar = "treplica" + varList.getLocalVariableDecList().get(0).getName
           ();
33     StringBuffer code = new StringBuffer("var " + nameTreplicaVar + " = Treplica new;\n");
34     code.append(nameTreplicaVar + " runMachine: ");
35     code.append(varList.getLocalVariableDecList().get(0).getName());
36     code.append(" numberProcess: ");
37     code.append(strList[0]);
38     code.append(" rtt: ");
39     code.append(strList[1]);
40     code.append(" path: ");
41     code.append(strList[2]);
42     code.append(";\n");
43     code.append(varList.getLocalVariableDecList().get(0).getName() + " setTreplica: " +
           nameTreplicaVar + ";\n");
44
45     return code;
46   }
47 }
```

Código-Fonte 60: Código Fonte de **CyanMetaobjectTreplicaInit**