RECUPERAÇÃO DA ARQUITETURA DE SOFTWARE PARA MANUTENÇÃO DE SISTEMAS

THIAGO HENRIQUE BRAGA

RECUPERAÇÃO DA ARQUITETURA DE SOFTWARE PARA MANUTENÇÃO DE SISTEMAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Roberto da Silva Bigonha
Coorientador: Marcelo de Almeida Maia

Belo Horizonte
Abril de 2013

Braga, Thiago Henrique

B813r Recuperação da Arquitetura de Software para Manutenção de Sistemas / Thiago Henrique Braga. — Belo Horizonte, 2013
xxiii, 122 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.
Orientador: Roberto da Silva Bigonha

Coorientador: Marcelo de Almeida Maia

1. Computação - Teses. 2. Engenharia de software - Teses. 3. Software - Arquitetura - Teses. I. Orientador II. Coorientador III. Título.

CDU 519.6*32 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Recuperação da arquitetura de software para manutenção de sistemas

THIAGO HENRIQUE BRAGA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCELO DE ALMEIDA MAIA - Coorientador
Departamento de Ciência da Computação - UFU

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

PROF. VLADIMIR OLIVEIRA DI IORIO
Departamento de Informática - UFV

Belo Horizonte, 19 de abril de 2013.

Dedico à minha família, fonte de amor sem fim.

Agradecimentos

Agradeço ao Colegiado do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais a oportunidade de participar do seu programa de mestrado.

Agradeço ao meu orientador e amigo, Roberto da Silva Bigonha, a atenção, disponibilidade, paciência, compreensão e ensinamentos que levarei para toda minha vida. Agradeço também ao meu co-orientador, Marcelo de Almeida Maia, o incentivo e confiança.

Agradeço aos colegas do Laboratório de Linguagens de Programação a troca de experiências e companheirismo.

A meus pais, Joaquim e Aparecida, e a meus irmãos, Guilherme e Vitor, agradeço o amor que vivemos a cada dia. A Ana Paula agradeço a motivação para terminar a dissertação.

Agradeço também a meus familiares e amigos que sempre me incentivaram questionando: e o mestrado?

Cada um de vocês contribuíram e contribuem na construção da minha vida. A todos vocês o meu sincero muito obrigado.

"De dia não se avisa. De noite pisca o olho."
(Joaquim Tadeu Braga)

Resumo

A arquitetura de *software* tem um papel importante tanto para o projeto quanto para o desenvolvimento e manutenção de *software*. Entretanto, vários programas são implementados sem possuírem uma arquitetura pré-definida, contrariando as boas práticas de desenvolvimento. Em outras situações, a arquitetura do sistema foi projetada, mas pode estar desatualizada em relação à implementação atual, devido às várias alterações do sistema. Como uma possível solução para esses problemas este trabalho propõe:

1. uma ferramenta para auxiliar na recuperação da arquitetura de *software* de programas escritos em Java, a partir do seu código-fonte e de análises feitas sobre dados coletados ao monitorar a execução do sistema;

2. uma metodologia para recuperação da arquitetura de *software* utilizando a ferramenta acima;

3. uma linguagem para especificar restrições arquiteturais entre componentes da descrição da arquitetura de *software*;

4. um mecanismo para verificar conformidade entre a arquitetura e a implementação do sistema.

Dentre as contribuições das ferramentas, destaca-se a possibilidade de verificar se alguma especialização de um *framework* obedece aos comportamentos esperados por sua especificação, definidos como restrições arquiteturais nos conectores da descrição da arquitetura do *framework*.

Abstract

The software architecture has an important role in the design and in the development and maintenance of software. However, several software has been implemented without having a previously defined architecture, as recommended by good practices for software development. In other situations, the architecture of the system has been designed in advance, but it may be out-of-date with respect to actual implementation, due to changes in the system. As a possible solution to these problems this work presents:

1. a tool to assist in the recovery of the software architecture of programs written in Java from its source code and analyses performed on data collected when the execution of the system is monitored;

2. a methodology for recovering the software architecture using the tool above;

3. a language to specify architectural restrictions among components of the software architecture description;

4. a mechanism to verify conformance between the architecture and implementation of the system.

Among the contributions of the tools, there is a possibility to check if some framework specialization meets the behaviour expected in its specification, defined as architectural restrictions on connectors of software architecture description of the framework.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Arquitetura de *Software*

Desde que o homem conseguiu computar e automatizar operações utilizando computadores digitais, muito trabalho foi feito para facilitar o desenvolvimento de programas. Antes de existirem as linguagens de programação de alto nível, os programas eram escritos, uma instrução por vez, utilizando código binário ou hexadecimal. Esse trabalho era tedioso e sujeito a muitos erros. A leitura e compreensão de programas eram bastante complicadas, e a sua modificação era uma tarefa extremamente difícil, pois todo programa era escrito utilizando endereço absoluto. A seguir procurou-se abstrair as operações de baixo nível que a máquina efetivamente realiza: operações de *hardware* para carregar registros de memória, resolver operações aritméticas e lógicas, deslocamento de *bits* foram codificadas em um conjunto de instruções, constituindo as primeiras abstrações de baixo nível, chamadas de linguagens de máquina ou *Assembly*. Os programas passaram a ser escritos como uma sequência de instruções, as quais eram traduzidas para linguagem de máquina por um tradutor, comumente chamado de *Assembler* [29].

Para aumentar a abstração das linguagens de máquina e facilitar a escrita de programas foram criadas as linguagens de programação: formalismos utilizados para descrever os procedimentos realizados pelo sistema projetado, seus algoritmos e estruturas de dados.

As primeiras linguagens de programação tiveram grande influência da arquitetura dos computadores, o que justifica a maneira de escrever programas dirigidos por processos. Seus principais elementos são:

- variáveis: posições de memória que podem ter seu valor alterado.
- comandos de atribuição: responsáveis pela alteração dos valores das variáveis.

- comandos de repetição: responsáveis pela iteração de um mesmo trecho de código do programa várias vezes.
- comandos de decisão: comandos que permitem decidir quando um trecho de código do programa será executado, em dado momento.
- procedimentos: são estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado.
- funções: semelhante aos procedimentos, exceto pelo fato de sempre retornar um valor ao ser chamada.

Em meados dos anos 70, o custo maior da computação mudou do *hardware* para *software*. Dijkstra caracterizou este período como a crise do *software* [12]. Esse termo expressava as dificuldades do desenvolvimento de *software* frente ao rápido crescimento da demanda, da complexidade dos problemas a serem resolvidos e da inexistência de técnicas estabelecidas para o desenvolvimento de sistemas que funcionassem adequadamente ou pudessem ser validados.

Como uma tentativa para contornar estes problemas e dar um tratamento de engenharia (sistemático e controlado) ao desenvolvimento de sistemas de *software* complexos, surgiu a área de Engenharia de *Software*. Apareceram então as primeiras metodologias de desenvolvimento de *software*: projeto top-down e refinamento passo a passo. Nessa técnica, a solução é especificada por um pseudo-programa de elevado nível, e em cada passo do refinamento, uma ou mais instruções do pseudo-programa dado são decompostas em instruções mais detalhadas. Essa sucessiva decomposição ou refinamento das especificações termina quando todas as instruções são expressas em termos de qualquer linguagem de programação. Logo percebeu-se que o código é a parte de um programa mais sujeito a variações: pequenas variações nos requisitos de um programa implicam em grande mudança no código e pouca mudança nos dados. Isto fez com que houvesse uma mudança radical no enfoque do desenvolvimento de programas. A programação, que antes dirigida por processos, seria dirigida por dados.

Seguindo esse novo enfoque, as linguagens de programação evoluíram em direção à programação dirigida por dados, originando no início dos anos 80 a metodologia orientada a objetos, que inclui:

- abstração de dados (encapsulamento);
- herança;
- polimorfismo;

- ligação dinâmica de métodos.

A principal ideia das linguagens orientadas a objetos é combinar em uma única entidade tanto os dados quanto as funções que operam sobre esses dados. Tal entidade é denominada objeto e suas funções são chamadas de métodos. Assim, a programação orientada a objetos (POO) apresenta-se como uma abordagem de programação que permite aos programadores raciocinar e solucionar problemas em termos de objetos, os quais estão diretamente associados às entidades reais dos problemas a serem resolvidos. Como resultado desse mapeamento natural, utilizando a POO, um programador pode concentrar-se nos objetos que compõem o sistema, em vez de tentar vislumbrar o sistema como um conjunto de procedimentos e dados.

A constante evolução das linguagens de programação possui algumas preocupações em comum:

- oferecer recursos mais elaborados que permitam ao desenvolvedor abstrair-se dos detalhes da máquina na qual o programa será executado;

- facilitar a leitura e escrita de programas;

- fornecer abstrações que estejam mais próximas da realidade dos problemas do que dos recursos das máquinas.

Com o aumento do tamanho e complexidade dos sistemas de *software*, o projeto do sistema vai além de recursos como algoritmos e estruturas de dados utilizados na construção do programa. Não que esses recursos deixem de ser importantes durante a implementação, mas do ponto de vista de uma visão macro do sistema, devem ser abstraídos. Um sistema de *software* complexo pode ser caracterizado por um conjunto de componentes abstratos de *software* (estruturas de dados e algoritmos) encapsulados na forma de procedimentos, funções, abstração de dados, tipos abstratos, objetos ou agentes e interconectados entre si, que deverão ser executados em sistemas computacionais. Nesse contexto, é interessante utilizar uma representação de alto nível de abstração, capaz de identificar as estruturas do sistema, suas relações e propriedades. Isso originou uma nova área da computação chamada Arquitetura de *Software*, como sendo o estudo da organização global dos sistemas de *software* bem como do relacionamento entre seus subsistemas e componentes [30]. A formalização da Arquitetura de *Software* como uma disciplina de engenharia para o desenvolvimento de *software* começou com Mary Shaw e David Garlan com a publicação do livro "*Software Architecture. Perspectives on an Emerging Discipline*", em 1996 [30].

Pode-se encontrar na literatura uma série de definições para arquitetura de *software*. A definição clássica apresentada por Shaw et al. diz que arquitetura de *software* define o que é o sistema em termos de componentes computacionais e os relacionamentos entre estes componentes.

Semelhante a esta definição, Bass et al. [6] diz que arquitetura de *software* são as estruturas que incluem componentes, suas propriedades externas e os relacionamentos entre eles, constituindo uma abstração do sistema. Esta abstração suprime detalhes de componentes que não afetam a forma como eles são usados ou como eles usam outros componentes, auxiliando o gerenciamento da complexidade.

Para Jazayeri et al. [19], a arquitetura de *software* é colocada como uma ferramenta para lidar com a complexidade do *software* e enfatizam que arquitetura deve satisfazer os requisitos funcionais e não funcionais do sistema, incrementando a definição de que arquitetura de *software* é o conjunto de componentes e seus relacionamentos. Portanto, é possível notar que a arquitetura de *software* é mais do que a descrição dos componentes que a compõem e do relacionamento entre eles. A arquitetura é a interface entre duas partes distintas: o problema de negócio e a solução técnica [5].

A descrição arquitetural de um sistema é extremamente importante, pois serve como um esqueleto sobre o qual propriedades podem ser provadas e, por isso, tem um papel vital em expor a capacidade do sistema em satisfazer seus requisitos principais. A seguir apresentam-se outros aspectos que justificam a importância e a necessidade da arquitetura de *software* no processo de desenvolvimento de um sistema:

- A arquitetura do *software* desempenha um papel chave como uma ponte entre requisitos e a implementação do sistema;

- A arquitetura de *software* deve atender às restrições e regras tanto do negócio quanto aquelas que influenciam os aspectos técnicos do sistema;

- A arquitetura constitui-se de um modelo simples e inteligente de como o sistema deve ser estruturado e como seus componentes se relacionam;

- A arquitetura permite antecipar as decisões de projeto, preocupando-se com tempo de desenvolvimento, custo e manutenção, definição das restrições de implementação e definição da estrutura organizacional, enfatizando os atributos de qualidade que o sistema requer e medindo através de avaliações a empregabilidade das qualidades necessárias [6].

1.1 Definição do Problema

A falta de abstrações intermediárias que conectam as características dos sistemas que os usuários necessitam às características dos sistemas efetivamente implementados é um tema importante em Engenharia de *Software*. Por muitos anos, diagramas de caixas e linhas apareceram e ainda são usados em descrições das implementações de sistemas. Estes diagramas geralmente são informais, raramente precisos e nunca completos. Visando melhores formas de desenvolvimento de *software*, descobriu-se a importância do projeto estrutural de um sistema. Esse nível de abstração é representado pela arquitetura de *software*, que permite o entendimento dos componentes de um sistema e seus inter-relacionamentos, e especialmente daquelas propriedades que são externamente visíveis ao longo do tempo e de implementações.

O projeto e especificação de um sistema é uma diretiva essencial para o sucesso do desenvolvimento. Metade do custo de um sistema é gasto com o projeto e como o sistema será organizado. Entretanto, gasta-se metade do custo total do sistema apenas para mantê-lo [11]. Isto reforça a necessidade de uma arquitetura documentada, para guiar os mantenedores durante a implementação e futura manutenção do sistema. A falta de uma arquitetura de software pode elevar o custo de manutenção do sistema pois uma parte do tempo será gasta para entender o *software* e a outra parte para descobrir onde a modificação deverá ser feita. Desta maneira, o principal papel da arquitetura de *software* de um sistema é informar aos interessados: (i) como o sistema inteiro foi particionado em componentes, (ii) como esses componentes se relacionam, (iii) como eles podem ser desenvolvidos cooperativamente e (iv) como os componentes podem ser agrupados para resolver o problema do sistema como um todo [11]. Outro ponto relevante é a utilização da arquitetura de *software* para reutilização, seja para incremento do sistema com funções que possuem características comuns ou seja na área de linhas de produtos [19].

Dessa maneira identificam-se alguns problemas importantes no desenvolvimento e manutenção de *software*:

- Vários programas são implementados sem uma arquitetura pré-definida, contrariando as boas técnicas de desenvolvimento. Descartando-se o projeto do sistema, pode-se acelerar a sua implementação, porém, a longo prazo, a manutenção do sistema será prejudicada, haja vista as dificuldades de entendimento do *software* a partir do código-fonte.

- Existem casos em que a arquitetura do sistema foi projetada, mas pode estar desatualizada em relação a implementação atual, devido às várias alterações na

implementação do sistema. Uma questão pertinente é como identificar a desatualização da arquitetura. Em outras palavras, como garantir que um sistema implementado esteja consistente com a sua arquitetura, projetada previamente [34].

- A utilização de *frameworks* que permitem a geração de sistemas completos de forma rápida por meio da reutilização (instanciação) e que podem ser distribuídos a diversos clientes. Os *frameworks* são aplicações semi-acabadas para um domínio, que geralmente definem uma série de pontos de extensão que podem ser especializados de acordo com o domínio do problema específico. Em algumas situações é interessante verificar se uma determinada especialização de um *framework* obedece a um comportamento esperado.

Esses problemas são pertinentes, pois um sistema implementado que não obedeça às especificações da sua arquitetura invalida todos os benefícios do projeto da arquitetura. Outro fator de grande importância é a possibilidade de verificar se a implementação do sistema segue as restrições arquiteturais definidas pela espeficicação da arquitetura do sistema. Isto vale tanto para aplicações que utilizam ou não *frameworks*, desde que a arquitetura do sistema determine um certo comportamento entre os componentes do sistema.

Assim sendo, é interessante que as ferramentas para manutenção de sistemas possuam as seguintes características:

- oferecer um ambiente para visualizar e editar a arquitetura de *software* dos sistemas em desenvolvimento;
- auxiliar no processo de recuperação da arquitetura de *software*;
- permitir a inserção de restrições entre os componentes da arquitetura de *software*, seja ela projetada ou recuperada;
- verificar se as restrições especificadas na descrição da arquitetura do sistema são satisfeitas pela implementação atual do mesmo.

1.2 Trabalho Proposto

Como uma possível solução para os problemas enunciados na Seção 1.1, este trabalho propõe:

(i) uma ferramenta denominada *ArchRecover* baseada em aspectos [20] para auxiliar na recuperação da arquitetura de *software* de programas escritos em Java, a partir do seu código-fonte e de análises feitas sobre dados coletados ao monitorar a execução do sistema;

(ii) uma linguagem para especificar restrições entre componentes da descrição da arquitetura de *software*;

(iii) um mecanismo baseado em aspectos chamado *ArchVerifier* para verificar conformidade entre a arquitetura e a implementação do sistema.

A recuperação da arquitetura de um *software* facilita o entendimento do sistema e pode auxiliar na comparação entre a arquitetura projetada e a recuperada, identificando inconsistências entre elas, bem como possibilitar que futuras manutenções preservem as propriedades arquiteturais.

As Figuras 1.1 e 1.2 apresentam uma visão geral do trabalho proposto, contendo os principais processos das ferramentas que foram implementadas para auxiliar na manutenção de sistemas:

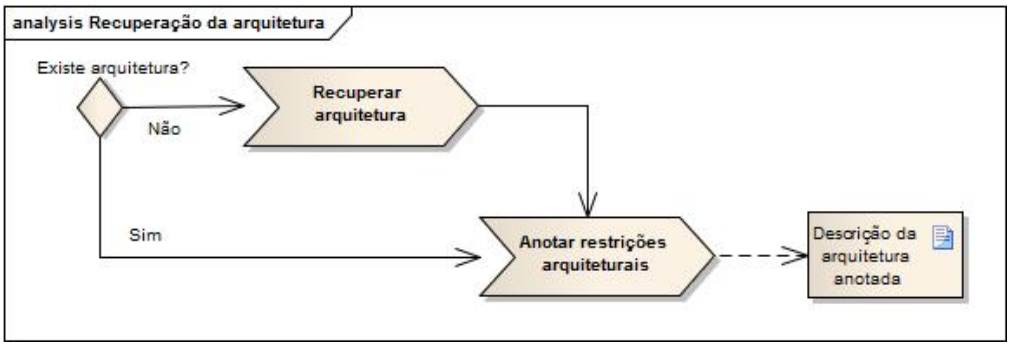

Figura 1.1. Processos de recuperação da arquitetura de software

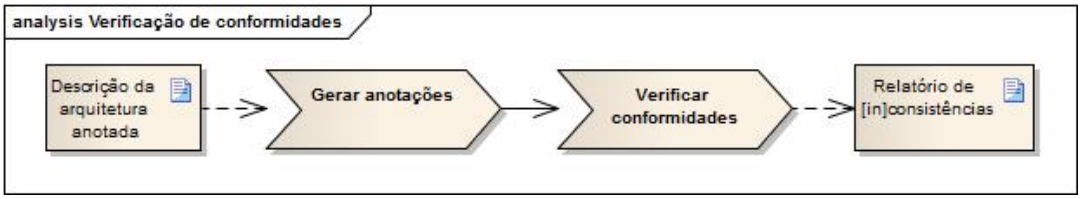

Figura 1.2. Processos de verificação de conformidades

1. O primeiro processo é a recuperação da arquitetura: se o sistema não possui uma arquitetura, ele será executado em conjunto com um aspecto que em tempo de

execução intercepta os eventos do programa (execução de métodos, execução de construtores e execução de blocos inicializadores estáticos de classe) gerando uma representação da execução do programa, chamada de **árvores de execução**. Por meio de consultas às árvores de execução o usuário pode construir de forma interativa a representação da arquitetura do programa adicionando os principais componentes e conectores. A metodologia de recuperação da arquitetura de *software* proposta por este trabalho será detalhada na Seção 5.3. Caso o sistema já tenha uma descrição da arquitetura, ou ela foi recuperada, o usuário anota as restrições arquiteturais aos conectores da descrição da arquitetura, as quais serão utilizadas nos próximos processos da ferramenta *ArchVerifier*.

2. O segundo processo é a anotação do código-fonte. De posse da representação da arquitetura com algumas restrições arquiteturais descrita em uma Linguagen para Descrição de Arquitetura (LDA), a ferramenta *ArchVerifier* gera um aspecto para anotar o código-fonte do programa nos pontos-chave das restrições arquiteturais. Este aspecto deverá ser utilizado pelo próximo processo para verificar conformidade entre a descrição da arquitetura e a implementação do sistema. A Seção 5.5 apresenta o gerador de anotações.

3. O terceiro processo é a verificação de conformidade: em resumo, o programa deverá ser executado em conjunto com dois aspectos, o que foi gerado pelo processo anterior que declara as anotações nos pontos-chave do programa e o aspecto de verificação que verifica conformidade entre arquitetura e a implementação atual do sistema. Assim como o aspecto de execução, o aspecto de verificação também constrói as árvores de execução do sistema e ao identificar os pontos-chave das restrições arquiteturais, o aspecto de verificação analisa se uma determinada restrição é satisfeita pelas árvores de execução do programa e produz um relatório de consistências ou inconsistências.

1.3 Motivação

Engenharia reversa[1] [10] é o processo de analisar um sistema para: (i) identificar os componentes do sistema e suas inter-relações e (ii) criar representações do sistema em outras formas ou em um nível mais elevado de abstração. Nesse contexto encontra-se a recuperação da arquitetura de *software*, que visa obter uma arquitetura documentada de um sistema existente [33]. O maior interesse em obter tal representação é a necessi-

[1] Do inglês, *Reverse engineering*

dade de entendimento do sistema com o intuito de facilitar a manutenção, reutilização, re-engenharia, etc. Na maioria dos casos, os sistemas não possuem uma documentação atualizada para suportar essas atividades, necessitando de uma maneira para obtê-la.

Nas últimas décadas, consideráveis pesquisas foram feitas para desenvolver notações, ferramentas e métodos para suportar o projeto da arquitetura de sistemas. Apesar dos avanços no desenvolvimento de uma engenharia base para arquitetura de *software*, um problema difícil ainda persiste: determinar se a arquitetura de um sistema implementado é consistente com a que foi projetada. A relevância deste problema se deve ao fato de que se um sistema for implementado e não seguir corretamente as especificações da sua arquitetura, todos os benefícios do projeto da arquitetura serão invalidados. Recuperando-se a arquitetura de um sistema, pode-se fazer comparações entre a arquitetura recuperada e a projetada para identificar incoerências, reforçando a confiabilidade da implementação do *software*.

E mais, em grande parte dos sistemas existentes ou suas descrições arquiteturais não existem ou não estão adequadamente descritas conforme a implementação atual do sistema. Uma questão importante é determinar se a arquitetura atual está ou não desatualizada. Por essas razões é necessário descrever a arquitetura de um sistema, utilizando técnicas ou processos para descobrir a arquitetura do sistema a partir das informações disponíveis, principalmente a partir do código-fonte.

O apoio de uma ferramenta que facilite a identificação de inconsistências entre a arquitetura do sistema e sua implementação é de extrema importância para os arquitetos do *software*, pois dessa maneira terão um maior grau de confiança de que a implementação incorpora as características e propriedades da arquitetura projetada.

As principais contribuições deste trabalho são:

- modelo para representar a execução de programas em Java, chamado de **árvores de execução**;
- uma linguagem de consulta às árvores de execução para auxiliar no processo de recuperação da arquitetura de *software*;
- uma linguagem para especificação de restrições entre componentes da descrição da arquitetura de *software* baseado em padrões de nomes e em hierarquia de tipos.
- um verificador de conformidade entre a descrição da arquitetura de *software* que possua restrições arquiteturais e a execução da implementação do sistema.

1.4 Organização do Texto

Este trabalho está organizado da seguinte maneira:

- No Capítulo 2 apresentam-se algumas das representações de arquitetura de *software* existentes, categorizando os tipos encontrados como representações informais, semi-formais e formais. O texto mostra os principais elementos de uma descrição de arquitetura, quais notações e linguagens são utilizadas para representá-los, bem como o significado de cada elemento arquitetural. No fim de cada seção são apresentados os comentários e críticas sobre cada categoria de representação.
- O Capítulo 3 apresenta uma técnica para recuperação da arquitetura de *software* de programas Java em tempo de execução. Avaliam-se as dificuldades dessa tarefa e as soluções apresentadas pela ferramenta DiscoTect [34]. No fim do capítulo são apresentadas algumas considerações sobre a omissão de detalhes importantes da metodologia utilizada pela ferramenta para recuperar a arquitetura de *software*.
- O Capítulo 4 é dedicado ao assunto verificação de conformidade entre arquitetura de *software* e implementação.
- Capítulo 5 detalha a solução proposta por este trabalho.
- No Capítulo 6 apresenta os detalhes de implementação das ferramentas que foram implementadas como *plug-ins* do Eclipse [ecl].
- A avaliação e validação das ferramentas são apresentadas no Capítulo 7 por meio de vários estudos de casos.
- Por último, o Capítulo 8 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Representações de Arquitetura de *Software*

Este capítulo apresenta algumas representações de arquitetura de *software* existentes, categorizando-as em três tipos: representações informais, semi-formais e formais. O principal objetivo deste capítulo é evidenciar os principais elementos de uma descrição de arquitetura, quais notações e linguagens são utilizadas para representá-los, bem como o significado de cada elemento arquitetural.

2.1 Representações Informais

A representação de arquitetura de *software* mais simples na documentação de sistemas de *software* é a informal. Em geral, cada equipe de desenvolvimento utiliza uma metodologia de desenvolvimento específica ou adaptada e utiliza diferentes formas para descrever a arquitetura de um sistema. Os tipos de representações informais encontradas são:

- arquiteturas descritas textualmente;
- diagramas de caixas e linhas.

A seguir, caracterizam-se esses dois tipos de representações informais, suas limitações e exemplos de cada uma.

2.1.1 Descrições de Arquitetura Textuais

As descrições textuais são textos em linguagem natural com informações sobre a arquitetura do sistema. Essa representação pode fornecer detalhes importantes dos princi-

pais componentes da arquitetura de um sistema. Entretanto, tende a ser uma descrição extensa e repleta de definições, o que exige muito esforço do leitor para compreendê-la. Além disso, existe a dificuldade de encontrar alguma informação específica dentro do texto da descrição da arquitetura.

Outro fator a ser levado em consideração é a capacidade de expressão do responsável pela arquitetura, o qual deve ter muito cuidado para não escrever descrições ambíguas e incompletas. Entretanto, o principal problema desse tipo de representação é que as relações e interações entre os componentes ficam espalhadas no texto e sujeitas à interpretação do leitor. Logo, pode-se inferir informações incorretas acerca da arquitetura, o que contraria o principal objetivo das descrições de arquitetura de *software*.

A seguir, apresenta-se parte da descrição da arquitetura do Apache Tomcat[1] [4], como um exemplo de descrição de arquitetura textual. Tomcat é um *container* de Servlets que é usada na implementação das tecnologias Java Servlets e Java Server Pages(JSP). A documentação da arquitetura disponível possui: (i) uma visão geral (ii) a inicialização do servidor; (iii) e o fluxo de processamento de requisições. No item (i), encontra-se uma série de definição de termos utilizados para descrever a arquitetura do Tomcat. Seguem alguns deles:

Termos	Descrição
`Servidor`	Um `Servidor` representa o *container* inteiro. Tomcat provê uma implementação padrão de `Server interface`, e esta é raramente modificada pelos usuários.
`Serviço`	Um `Serviço` é um componente intermediário que fica dentro de um `Servidor` e amarra um ou mais `Conectores` a exatamente um `Motor`. A implementação padrão é simples e suficiente: `Service interface`.
`Motor`	Um `Motor` representa o processamento de uma requisição para um `Serviço` específico. Como um `Serviço`, o `Motor` pode ter múltiplos `Conectores`. O `Motor` recebe e processa todas as requisições destes `Conectores`, entregando a resposta ao `Conector` apropriado para transmissão ao cliente.

A descrição da arquitetura do Tomcat primeiramente define os principais componentes do servidor. Encontram-se, na definição de cada componente, inúmeras referências ao Javadoc da implementação, uma documentação das classes do sistema. Apesar disso, a descrição da arquitetura é pobre, dando margens a várias perguntas em relação

[1]Disponível em: http://tomcat.apache.org/tomcat-5.5-doc/architecture/index.html

à interpretação da descrição, por exemplo:

- Quais são os componentes da arquitetura do Tomcat? São estes termos que foram definidos na seção visão geral da arquitetura?
- Um `Serviço` é uma relação existente entre `Conectores` e `Motor`?
- O responsável por transmitir as resposta para o cliente é o `Motor` ou o `Conector`?

Estas dúvidas são inerentes à falta de formalismo das descrições deste tipo e à ambiguidade do texto. Outra questão importante que este tipo de descrição deixa a desejar é o espalhamento das informações referente aos conectores da arquitetura do sistema.

2.1.2 Diagramas de Caixas e Linhas

Outra representação informal que é bastante utilizada para representar a arquitetura de sistemas são os diagramais de caixas e linhas [11]. Nestes diagramas as únicas considerações que pode-se fazer são:

- as caixas representam os componentes da arquitetura;
- as linhas representam as interações entre os componentes, ou seja, os conectores.

Em relação ao tipo de interação existente entre os componentes, nada se pode afirmar, exceto quando os diagramas possuem informações adicionais. Analisando esses diagramas, percebe-se que a maioria dos autores esforçam-se para representar mais detalhes do que esses diagramas realmente podem expressar, deixando algumas perguntas em aberto: as caixas representam programas em execução? Ou blocos de código-fonte? Ou computadores físicos? Ou meramente agrupamentos lógicos de funcionalidades? As setas e linhas representam dependências de compilação? Ou controle de fluxo? Na verdade, os diagramas contêm um pouco das respostas de cada uma destas questões [21].

Como caixas e linhas são insuficientes para representar a arquitetura, alguns autores adotam algumas convenções como legendas e comentários para completar a descrição, o que geralmente a sobrecarrega. Para diferenciar os diversos tipos de componentes de uma arquitetura, cria-se uma caixa para cada tipo. Esta diferenciação também é feita para os conectores, usando-se diferentes tipos de linhas ou de cores. Embora aparentem ser fáceis e intuitivos, estes diagramas são incompletos e imprecisos: podem fornecer múltiplas interpretações, dependendo da representação de cada elemento contido na descrição [11].

A Figura 2.1 é um exemplo deste tipo de representação informal: a descrição da arquitetura do *Duke's Bank J2EE Application*, foi obtida do Tutorial J2EE[2].

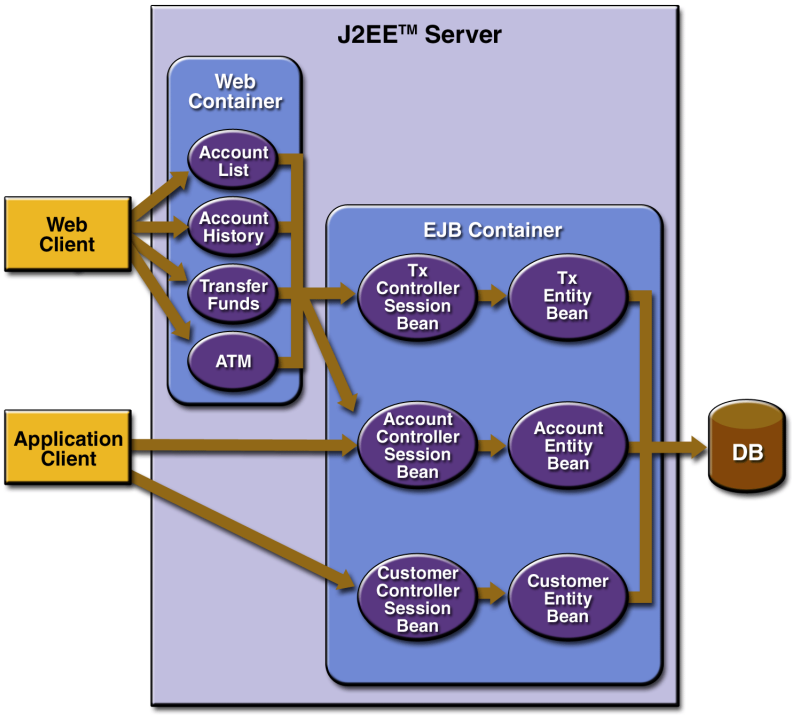

Figura 2.1. Arquitetura Informal do *Duke's Bank*

A descrição da arquitetura da Figura 2.1 fornece uma visão geral da implementação do sistema. Mesmo assim, pode-se identificar as seguintes inconsistências:

- existem diferentes tipos de componentes representados por caixas distintas. Porém, alguns deles foram representados com a mesma caixa e possuem funcionalidades completamente diferentes. É o caso dos *session beans*, dos *entity beans* [31] e das páginas web.

- A descrição utilizou somente uma representação para conectores, o que não é verdade para este sistema. Por exemplo, entre o banco de dados e os *entity beans* existe uma conexão JDBC, e a interação entre os *session beans* e os *entity beans* se dá por chamadas remota a *EJBs*.

Assim como nas descrições de arquitetura textuais, este tipo de representação não permite nenhum mecanismo de verificação e possibilita a inferência de dados incorretos sobre a arquitetura do sistema. A utilização de legendas pode auxiliar a compreensão

[2]Disponível em: http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html

da arquitetura, porém a falta de padrões permite que um mesmo elemento possua diferentes significados em diagramas distintos.

2.1.3 Comentários

A informalidade dessa categoria de representação prejudica a clareza do projeto da arquitetura do *software* e, em vez de esclarecer, este tipo de representação pode causar várias dúvidas. Diagramas informais não podem ser analisados formalmente para verificar consistência, completude e correção. Os requisitos arquiteturais assumidos no projeto inicial não podem ser verificados à medida que o sistema evolui. Além disto, existem poucas ferramentas para auxiliar os arquitetos nestas tarefas [15].

2.2 Representações Semi-Formais

Esta categoria de representação caracteriza-se por atribuir significados aos elementos de uma descrição de arquitetura dos sistemas, diminuindo a informalidade dos diagramas de caixas e linhas. Um exemplo são as extensões de UML, que possuem uma semântica mais precisa, porém com limitações para expressar aspectos dinâmicos da arquitetura: ou os diagramas ficam muito complexos e, portanto, menos úteis para análise, ou ficam muito simplificados, inexpressivos para análise, além de não se ter mecanismos automatizados ou formais de verificação de propriedades.

Um outro exemplo é o modelo de visões arquiteturais 4+1 (*The 4+1 View Model of Architecture*) [21], que destaca a importância de perspectivas estáticas e dinâmicas da arquitetura.

2.2.1 Extensões de UML

UML (*Unified Modeling Language*) [8] é uma notação de modelagem de propósito geral utilizada para especificar sistemas de *software* orientados a objetos. Por meio de seus diagramas é possível representar sistemas de *software* sob diversas perspectivas de visualização. Dentre seus benefícios destacam-se: familiaridade dos desenvolvedores, mapeamentos diretos para implementação e existência de ferramentas comerciais para suporte.

2.2.2 Modelo de Visões Arquiteturais 4+1

O Modelo de Visões Arquiteturais 4+1 [21] descreve a arquitetura de *software* usando quatro visões concorrentes, na qual cada uma está relacionada a um conjunto de pre-

ocupações específicas do interesse de diferentes participantes de um sistema, como ilustrado pela Figura 2.2, extraída de [21]. Os usuários finais, clientes e especialistas em dados trabalham com a visão lógica; os gerentes de projetos e programadores utilizam a visão de desenvolvimento; e os engenheiros de sistemas se preocupam com a visão física e a de processos. Esta abordagem possui uma quinta visão, chamada de cenários, que justifica o nome "4+1". A seguir, cada visão será detalhada, mostrando o objetivo de cada uma e a notação sugerida pelo autor.

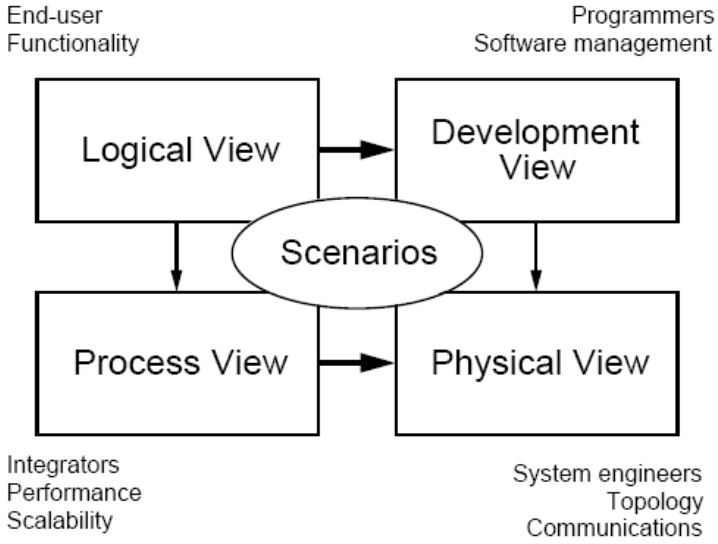

Figura 2.2. Modelo de Visões Arquiteturais 4+1

- *Visão lógica*: suporta os requisitos funcionais do sistema – o que ele deve prover em termos de serviços a seus usuários. O sistema é decomposto em um conjunto de abstrações, tomadas do domínio do problema, na forma de objetos e classes de objetos, que exploram os princípios de encapsulamento e herança. A notação utilizada para descrever o modelo de objetos do projeto são os diagramas de classes UML, com suas relações lógicas: associação, dependência, agregação, herança, etc.

- *Visão de processo*: captura os aspectos de concorrência e sincronização do projeto e também alguns requisitos não-funcionais como desempenho e disponibilidade. Trata assuntos como concorrência e distribuição, integridade do sistema, tolerância a falhas e como as principais abstrações da visão lógica preenchem a arquitetura de processos. Esta visão pode ser descrita em muitos níveis de abstração, cada nível cuidando de diferentes preocupações. Um processo é um grupo de tarefas que formam uma unidade executável, o qual pode ser iniciado,

reconfigurado e desligado pela arquitetura de processos. O *software* é particionado em um conjunto independente de tarefas, no qual uma tarefa é uma *thread* de controle separada, que pode ser agendada individualmente em um nodo de processamento. Para aumentar a distribuição ou a disponibilidade, um processo pode ser replicado. A notação utilizada para representar a visão de processos é uma notação estendida da originalmente proposta por Booch para tarefas em Ada [22].

- *Visão de desenvolvimento*: descreve a organização estática do *software* em seu ambiente de desenvolvimento. O *software* é empacotado em bibliotecas, ou subsistemas, que podem ser desenvolvidos por um ou alguns desenvolvedores. Os subsistemas são organizados em uma hierarquia de camadas, onde cada camada provê uma interface direta e bem-definida para as camadas acima dela. A representação utilizada é o diagrama de módulos e subsistemas, mostrando as relações de exportação e importação existentes entre os módulos.

- *Visão física*: descreve o mapeamento do *software* no *hardware* e reflete seu aspecto distribuído. Também trata de requisitos não-funcionais, tais como: disponibilidade, tolerância a falhas, desempenho e escalabilidade. Os vários elementos identificados – redes, processos, tarefas e objetos – precisam ser mapeados em vários nodos de processamento, e espera-se ter várias configurações físicas diferentes: algumas para desenvolvimento, outras para testes e implantação do sistema. O mapeamento do *software* para os nodos de processamento precisa ser altamente flexível e deve minimizar o impacto no código fonte.

- *Visão de cenário*: a quinta visão são alguns casos de uso e cenários escolhidos para agrupar as quatro visões anteriores. Os cenários são abstrações dos requisitos mais importantes do sistema. Seu projeto é representado utilizando diagramas de casos de uso e diagramas de interação de objetos. Esta visão serve para dois propósitos: (i) guia para descobrir os elementos arquiteturais durante o projeto da arquitetura do sistema e (ii) validação e ilustração depois que o projeto da arquitetura estiver completo.

Este modelo é um roteiro para documentação da arquitetura de um sistema, agrupando-a em cinco visões, cada uma com preocupações específicas. Para cada visão é apresentada a notação mais utilizada, geralmente UML. Porém o modelo é genérico o suficiente para utilizar a notação mais apropriada, de acordo com o domínio do problema. A Tabela 2.1 apresenta as principais características de cada visão.

	Visão				
Elemento	Lógica	Processo	Desenvolvimento	Física	Cenários
Componentes	Classes	Tarefas e Processos	Módulos e subsistemas	Nodos	Passos e roteiros
Conectores	associação, herança, agrupamentos	Mensagens, RPC, etc	dependência de compilação	meio de comunicação, LAN, WAN, etc	
Participantes	Usuários finais	Engenheiro de sistemas	Gerente de projetos e desenvolvedores	Engenheiro de sistemas	Clientes e desenvolvedores
Preocupações	Funcionalidade	Desempenho, disponibilidade, tolerância a falhas, integridade	Organização, reúso, portabilidade e linhas de produtos	Escalabilidade, desempenho e disponibilidade	Entendimento

Tabela 2.1. Principais Características das Visões do Modelo 4+1.

Não é todo sistema que necessita de todas as visões, podendo ser omitidas as visões menos úteis da descrição da arquitetura. Por exemplo, se o sistema utilizar somente um processador não faz sentido criar as visões de processo e física. Para sistemas pequenos é possível criar somente a visão lógica, pois a visão de desenvolvimento será similar, sendo desnecessário produzir duas descrições separadas. A visão considerada essencial é a de cenários e esta deve ser produzida em qualquer cincunstância.

Este modelo foi considerado como semi-formal pois apresenta uma forma de organizar a arquitetura de *software* por meio das cinco visões. Entretanto, o modelo permite a utilização de notações diferentes para uma mesma visão. Este grau de liberdade impossibilita uma padronização.

2.3 Representações Formais

Buscando melhores formas de representação, que capturassem os principais elementos da arquitetura, foram criadas várias Linguagens para Descrição de Arquitetura (LDA) seja para um domínio específico ou para uso de propósito geral [28]. Uma LDA pode ser definida como uma linguagem para especificar a estrutura de alto nível de uma aplicação, ao invés de se preocupar com detalhes de implementação de um módulo de código específico. Ela modela a arquitetura conceitual de um sistema de software, distinguindo-a da implementação do sistema. Assim, o papel principal de uma LDA é permitir a especificação de descrições da arquitetura de *software*, definindo seus componentes, como eles se comportam e os padrões e mecanismos para interações entre

esses componentes [30]. Estas linguagens possuem sintaxe bem formada, assim como seus elementos possuem significados bem definidos. Estes fatos as caracterizam como representações formais.

Shaw e Garlan [30] apresentam seis propriedades que caracterizam o que uma linguagem de descrição de arquitetura ideal deveria fornecer:

- Composição: possibilidade de descrever um sistema como uma composição de elementos independentes;
- Abstração: possibilidade de descrever os elementos de uma arquitetura de software de tal forma que seus papéis abstratos em um sistema sejam prescritos claramente e explicitamente;
- Reúso: possibilidade de reutilizar elementos em descrições arquiteturais diferentes, mesmo que fossem desenvolvidos fora do contexto de um sistema arquitetural;
- Configuração: possibilidade de descrever a estrutura de um sistema, independente dos elementos sendo estruturados e, se possível, dar suporte à reconfiguração dinâmica. Para isso, uma LDA deveria separar a descrição de estruturas compostas dos elementos constituintes, facilitando o entendimento da composição como um todo;
- Heterogeneidade: possibilidade de combinar descrições arquiteturais múltiplas e heterogêneas;
- Análise: possibilidade de realizar uma análise rica e variada de descrições arquiteturais.

Apesar de existirem várias LDAs, Medvidovic [28] mostrou que existia pouco consenso entre os pesquisadores do que é uma LDA, quais aspectos de uma arquitetura devem ser modelados por uma LDA e quais ferramentas uma LDA deve oferecer. Em seu trabalho, Medvidovic apresentou um conjunto de critérios para classificar e comparar LDAs. Algumas das linguagens avaliadas por Medvidovic foram:

- Aesop [14]: suporta o uso de estilos arquiteturais;
- C2 [32; 27]: oferece suporte a descrição de interfaces baseadas em eventos;
- Darwin [25; 26]: dá suporte para análise na troca de mensagens em sistemas distribuídos, utilizando o π-calculus para formalizar a semântica arquitetural;

- Meta-H [7]: oferece heurísticas para projetistas de controle de software de tempo real para aviação;
- Rapide [23; 24]: linguagem de propósito geral que permite simular e analisar projetos de arquiteturas;
- Wright [3]: LDA que formaliza a semântica da arquitetura de software em termos de componentes, conectores, portas e papéis tendo como base a álgebra CSP, "*Communicating Sequential Processes*";
- ACME: [16]: linguagem genérica para descrição de arquitetura.

Medvidovic caracterizou os requisitos mínimos de uma LDA como sendo a capacidade de modelar explicitamente componentes, conectores e suas configurações e que para uma LDA ser utilizada e útil ela deve disponibilizar ferramentas de suporte para desenvolvimento e evolução. Assim, os elementos principais de uma LDA são [28]:

- componentes: unidades de computação ou armazenamento de dados que podem ser tão pequenos quanto um único procedimento ou tão grandes como uma aplicação inteira;
- conectores: usados para modelar interações entre componentes e regras que governam essas interações. Diferente dos componentes, conectores podem não corresponder a unidades compiladas em um sistema implementado. Eles podem se manifestar como variáveis compartilhadas, chamadas de procedimentos, protocolos cliente-servidor, *pipes* e assim por diante;
- configurações arquiteturais: grafos conexos de componentes e conectores que descrevem a estrutura da arquitetura. Idealmente, a estrutura de um sistema deveria estar clara a partir de uma especificação de configuração somente, sem precisar estudar as especificações dos componentes e conectores. Composição hierárquica é desejável em LDAs, permitindo a descrição de um sistema em diferentes níveis de detalhe. Uma estrutura e um comportamento complexo poderiam ser explicitamente representados ou poderiam ser abstraídos em um componente ou conector único que faz parte de outra arquitetura ainda mais complexa.

Para especificar o comportamento dos componentes e os protocolos de interação entre eles, normalmente as LDAs fornecem um descrição semântica do sistema, além da descrição estrutural de sua arquitetura, que pode ser feita em CSP, π-calculus, redes de Petri, etc, ou mesmo via uma metalinguagem para especificar a semântica de uma

LDA. Contudo, o foco deste trabalho está nas definições estruturais de LDAs, sem considerar a especificação semântica. Assim, será apresentada a linguagem que será utilizada para representar a arquitetura de *software* de sistemas neste trabalho.

2.3.1 ACME

ACME [16] é uma linguagem de descrição de arquitetura que provê: (i) uma ontologia[3] arquitetural consistindo em sete elementos, (ii) um mecanismo de anotação flexível, responsável pela associação de informação não estrutural usando sub-linguagens definidas externamente e (iii) um mecanismo de tipos para abstrações comuns, reutilização de idiomas e estilos de arquitetura.

A linguagem possui sete tipos de elementos para representação de arquitetura: componentes, conectores, sistemas, portas, papéis, representações e mapas de representações. Os elementos mais básicos são os três primeiros e podem ser vistos na Figura 2.3, extraída de [16]. A seguir apresenta-se uma breve descrição de todos eles:

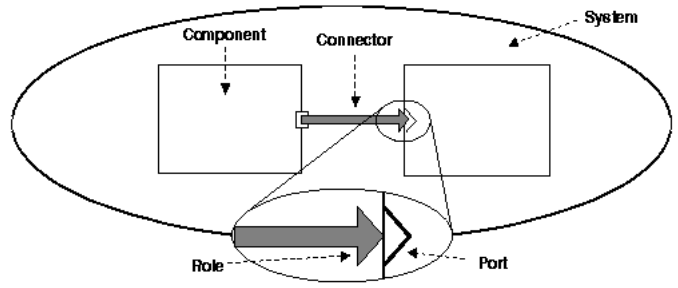

Figura 2.3. Principais Elementos de ACME

- *Componentes* (`Component`): representam os elementos primários de computação e armazenamento de dados de um sistema. Alguns exemplos são: clientes, servidores, filtros, objetos, arquivos e bancos de dados.

- *Conectores* (`Connector`): representam as interações entre os componentes do sistema. Computacionalmente falando, conectores realizam a comunicação e a coordenação de atividades entre componentes. Exemplos incluem formas de interação simples, tais como dutos (fluxo de dados) e chamadas de procedimentos. Conectores também podem representar interações mais complexas, tais como um

[3]Ontologia é um modelo de dados que representa um conjunto de conceitos dentro de um domínio e os relacionamentos entre estes.

protocolo cliente-servidor ou uma conexão SQL entre um banco de dados e uma aplicação.

- *Sistemas* (`System`): representam configurações de componentes e conectores, ou seja, como os componentes se relacionam e como estão conectados.

- *Portas* (`Port`): as interfaces dos componentes são definidas por um conjunto de portas. Cada porta identifica um ponto de interação entre um componente e seu ambiente. Um componente pode prover múltiplas interfaces usando diferentes tipos de portas. Uma porta pode representar uma interface simples como uma assinatura de procedimento ou interfaces mais complexas como uma coleção de chamadas de procedimentos que devem ser invocadas em uma ordem específica.

- *Papéis* (`Role`): conectores também têm interfaces que são definidas por um conjunto de papéis. Cada papel de um conector define um participante de uma interação representada pelo conector. Conectores binários têm dois papéis, por exemplo, os papéis de chamador e de chamado de um conector RPC ou os papéis de leitor e de escritor de um duto. Outros tipos de conectores podem ter mais de dois papéis. Por exemplo, um conector de transmissão de eventos deve ter um único papel de anunciador de evento e um número arbitrário de papéis de receptores de evento.

- *Representações* (`Representation`): cada descrição de arquitetura é chamada de representação em ACME. Pode-se criar uma hierarquia de descrições de arquitetura, detalhando um componente ou conector com uma ou mais descrições detalhadas de baixo nível. Porém, não existe nenhuma construção em ACME que suporte resolução de correspondências entre visões. As ligações entre as portas dos componentes e os papéis dos conectores são feitas por meio de uma lista de `Attachments`.

- *Mapa de Representações* (`Representation Map`): os mapas de representações são utilizados para criar outras visões de componentes ou conectores de um sistema, permitindo detalhar estes elementos internamente e indicar a correspondência entre a representação interna e a interface externa do componente ou conector. A representação interna é uma visão com mais detalhes do componente ou conector. Os mapas de representações permitem criar associações entre:

  – Portas internas (portas do componente na representação mais detalhada) e portas externas (portas do componente onde ele é utilizado), no caso de componentes;

– Papéis internos (papéis do conector na representação mais detalhada) e papéis externos (papéis do conector onde ele é utilizado), no caso de conectores.

Estas associações são feitas utilizando a palavra reservada `Bindings`, seguida da ligação da porta interna para a porta externa, ou da ligação do papel interno para o papel externo.

ACME provê anotações da estrutura da arquitetura com uma lista de propriedades (`Properties`). Cada propriedade (`Property`) possui um nome, um tipo opcional e um valor. Todos os sete tipos de entidades podem ser anotados. As propriedades aumentam o poder de expressão da linguagem, tornando-a extensível. As propriedades não são interpretadas por ACME e só são úteis quando uma ferramenta faz uso delas para análise, tradução ou manipulação.

Como um exemplo simples, a Figura 2.4 mostra o diagrama de uma arquitetura trivial contendo um cliente e um servidor, conectados por um conector RPC [17]. Para mostrar o uso de todos os elementos de ACME, o componente servidor foi detalhado em três sub-componentes.

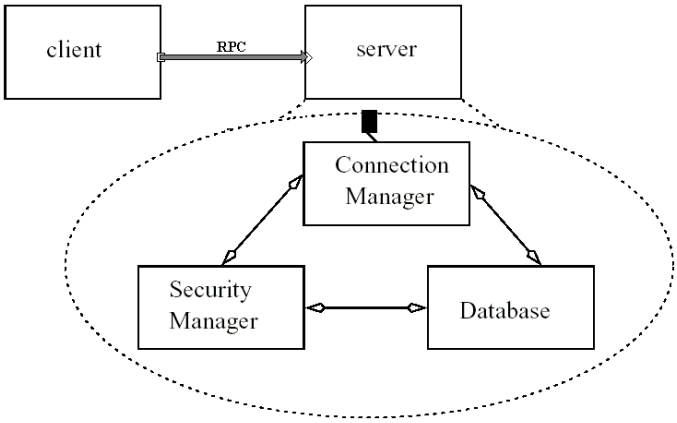

Figura 2.4. Diagrama Cliente-Servidor

A descrição em ACME é:

```
1 System simple_cs = {
2   Component client = {
3     Port send-request;
4     Property sourceCode : externalFile = "simplecs/Client.java"
5   }
6
7   Component server = {
8     Port receive-request;
9     Property sourceCode : externalFile = "simplecs/Server.java";
```

```
10      Representation serverDetails = {
11        Component connectionManager = {
12          Ports { externalSocket; securityCheckIntf; dbQueryIntf }
13        }
14        Component securityManager = {
15          Ports { securityAuthorization; credentialQuery; }
16        }
17        Component database = {
18          Ports { securityManagementIntf; queryIntf; }
19        }
20        Connector SQLQuery = { Roles { caller; callee } }
21        Connector clearanceRequest = { Roles { requestor; grantor } }
22        Connector securityQuery = { Roles { securityManager; requestor } }
23        Attachments {
24          connectionManager.securityCheckIntf to clearanceRequest.requestor;
25          securityManager.securityAuthorization to clearanceRequest.grantor;
26          connectionManager.dbQueryIntf to SQLQuery.caller;
27          database.queryIntf to SQLQuery.callee;
28          securityManager.credentialQuery to securityQuery.securityManager;
29          database.securityManagementIntf to securityQuery.requestor;
30        }
31        Bindings { connectionManager.externalSocket to server.receiveRequest }
32      }
33    }
34    Connector rpc = {
35      Roles {caller, callee}
36      Properties {
37        synchronous : boolean = true;
38        maxRoles : integer = 2
39      }
40    }
41    Attachments {
42      client.send-request to rpc.caller;
43      server.receive-request to rpc.callee
44    }
45  }
```

A descrição da arquitetura do sistema apresentado acima possui 2 componentes principais: o cliente e o servidor. O componente `client` possui:

- uma porta, `send-request`, que é utilizada para fazer as requisições ao servidor;
- e uma propriedade chamada `sourceCode`, cujo tipo é `externalCode`.

O componente `server` possui uma porta para receber as requisições chamada `receive-request` e também possui a propriedade `sourceCode`, indicando qual é o arquivo com o código-fonte deste componente. O componente servidor foi detalhado em uma representação interna mais expressiva, exemplificando o uso do mapa de representações. Ele possui:

- 3 componentes internos:
  - `connectionManager` com 3 portas;
  - `securityManager` com 2 portas;
  - `database` também com 2 portas.
- 3 conectores internos: `SQLQuery`, `clearanceRequest` e `securityQuery`.
- A ligação da linha 31 indica que o componente interno do servidor que recebe as requisições do cliente é o `connectionManager`.
- O componente `connectionManager` está conectado ao `securityManager` por meio do conector `clearanceRequest`, linhas 24 e 25 e está conectado ao `database` via o conector `SQLQuery`, linhas 26 e 27.
- As ligações das linhas 28 e 29 mostram que o `securityManager` está conectado ao componente `database` por meio do conector `securityQuery`.

2.3.2 Comentários

Embora ACME não seja considerada uma LDA por Medvidovic [28], a linguagem possui uma base comum a todas as LDAs. Em ACME qualquer outro aspecto de descrição arquitetural é representado como uma lista de propriedades. Estas propriedades não são processadas pelo núcleo da linguagem. Esta é a principal característica que a torna uma linguagem intermediária que possibilita o mapeamento de elementos arquiteturais de uma LDA para outra. Além disto, ACME é uma das principais LDAs utilizadas para especificação arquitetural, dada sua generalidade e flexibilidade, além do suporte de ferramentas textual e gráfica oferecido pela mesma.

2.4 Conclusão

Neste capítulo foram apresentadas diferentes representações de arquitetura de *software*, desde representações elementares como diagramas de caixas e linhas até as linguagens

para descrição de arquitetura (LDAs). Essas representações foram caracterizadas em três tipos: informais, semi-formais e formais. Mostrou-se também que as representações informais motivaram os pesquisadores a criarem as LDAs e que essas especificam a estrutura de alto nível de uma aplicação, definindo seus componentes, como eles se comportam e os padrões e mecanismos para interações entre esses componentes [30].

De acordo com Medvidovic, uma LDA deve modelar explicitamente componentes, conectores e configurações arquiteturais. A fim de inferir alguma informações sobre uma arquitetura, uma LDA deve modelar no mínimo interfaces de componentes. Sem esta informação, uma descrição de arquitetura torna-se uma coleção de identificadores conectados, similares aos diagramas de caixas e linhas sem nenhuma semântica explícita. Além disto, para serem verdadeiramente úteis, elas devem fornecer suporte a ferramentas para desenvolvimento e evolução baseado em arquitetura.

Os motivos que levaram à escolha de ACME como LDA para representar as descrições da arquitetura deste trabalho são:

- ACME possui os principais elementos para representar as descrições de arquitetura de software: componentes, conectores e propriedades arquiteturais;
- enfoque do trabalho está relacionado às definições estruturais das LDAs, sem considerar a especificação semântica;
- possibilidade de usar as propriedades arquiteturais para armazenar qualquer informação;
- ACME possui um editor gráfico, ACME Studio, para visualizar e especificar as descrições de arquitetura de software que pode ser usado no Eclipse [ecl].

Capítulo 3

Recuperação da Arquitetura de *Software*

Este capítulo apresenta o resultado do estudo de uma técnica para recuperação da arquitetura de *software*. O objetivo desse estudo é identificar quais são as dificuldades desse processo, quais os mecanismos utilizados pela técnica DiscoTect [34] e levantar quais melhorias ou soluções alternativas poderiam ser abordadas pelas ferramentas propostas neste trabalho. Na conclusão deste capítulo, são apresentadas algumas deficiências da técnica estudada que justificam a implementação da ferramenta proposta por este trabalho, *ArchRecover*, para recuperação da arquitetura de *software*.

3.1 DiscoTect

DiscoTect [34] é uma técnica de recuperação da arquitetura de *software* que, monitorando a execução de um sistema, captura os eventos produzidos, por exemplo, chamadas de métodos, inicialização de objetos, etc, e constrói uma visão da arquitetura do sistema. A ideia chave utilizada é que um sistema pode ser monitorado enquanto estiver executando e observações sobre seu comportamento podem ser usadas para inferir sua arquitetura dinâmica, ou seja, a arquitetura do sistema em tempo de execução.

Existem uma série de dificuldades para que essa técnica funcione, e a mais complicada delas é encontrar um mecanismo para tratar a lacuna de abstração, ou seja, o nível de detalhes que existe em uma especificação de alto-nível e a implementação de um sistema: geralmente, eventos de baixo-nível de um sistema não mapeiam diretamente em ações de arquitetura (criação de componentes, conectores). Para isso, o arcabouço permite mapear estilos de implementação para estilos de arquitetura. Esses mapeamentos são definidos como um conjunto de máquinas de estado conceitualmente

concorrente que são usadas em tempo de execução para trilhar o progresso de um sistema e produzir ações de arquitetura quando padrões de tempo de execução pré-definidos são reconhecidos.

Os principais elementos do projeto de DiscoTect podem ser visualizados na Figura 3.1, extraída de [34].

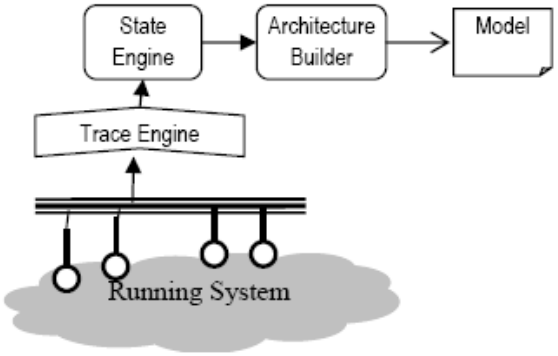

Figura 3.1. A Arquitetura de DiscoTect

Para monitorar o sistema em execução, o **Trace Engine** usa *Java Platform Debugger Architecture* (JPDA) para capturar eventos do sistema, que podem ser de três tipos: chamadas de métodos, inicialização de objetos e modificação de atributos. Os eventos capturados pelo **Trace Engine** são representados como eventos parametrizados:

- `Call (method, caller, callee)`: Um evento `Call` ocorre quando um método é invocado no sistema em execução. Cada evento deste tipo possui o nome do método (`method`), o objeto que invocou o método (`caller`) e o objeto o qual o método foi invocado (`callee`).

- `Init (constructor, creator, instance)`: Um evento `Init` ocorre quando um construtor é invocado, inicializando um objeto no sistema em execução. Este evento possui o nome do construtor (`constructor`), o objeto que invocou o construtor (`creator`) e o nome e tipo do novo elemento criado (`instance`).

- `Modify (owner, field, value)`: Um evento `Modify` ocorre quando algum campo de um objeto é alterado no sistema em execução. Este evento possui o nome do objeto proprietário do campo (`owner`), o campo que foi modificado (`field`) e o valor atribuído ao campo (`value`).

Quando um evento é identificado, ele é emitido para a **State Engine** (Figura 3.1), que interpreta a descrição de uma máquina de estados. A essência da solução para

o problema de mapeamentos entre operações de baixo-nível para ações de arquitetura estão nas máquinas de estados. Estas máquinas são usadas pela **State Engine** para fazer a recuperação da arquitetura. À medida que transições ocorrem, as ações de arquitetura especificadas nas transições são executadas pelo **Architecture Builder**. No fim do processo, o modelo produzido é a arquitetura do sistema descrita em ACME[16].

A máquina de estados de DiscoTect difere da definição padrão[1]. Ela é um grafo de estados, gatilhos, ações e transições. Os estados mantêm as informações da arquitetura descoberta até o momento. Cada estado é associado a um ou mais gatilhos, os quais definem o tipo de evento que deve ser observado e que condições específicas devem ser satisfeitas para que a transição ocorra. O gatilho também especifica qual o tipo de transição que ocorrerá entre dois estados. Quando uma transição acontece, ela pode produzir ações para construir a arquitetura do sistema observado.

Especificamente uma máquina de estados de DiscoTect requer a definição de estados, gatilhos, ações e transições:

1. **Estados.** Estados são etapas da recuperação de alguma arquitetura. Um estado pode representar um conhecimento parcial de uma arquitetura – por exemplo, um conector foi criado mas não se sabe quais componentes ele conecta – e permitir a construção de mapeamentos complexos que combinam pedaços de informações parciais para ações de arquitetura coerentes, que são armazenadas como variáveis de estado. A cada estado da máquina é associado um conjunto de variáveis.

2. **Gatilhos.** Um gatilho possui duas partes: especificação de evento e uma condição que deve ser satisfeita para que a transição ocorra.
   Quando um estado é ativado por um evento, os parâmetros do evento são gravados como variáveis de estado, as quais podem ser referenciadas em condições de gatilhos dos estados subsequentes ou ações. Dessa forma, uma ação de arquitetura pode usar informações dos estados anteriores. As condições dos gatilhos são escritas como expressões boleanas sobre os valores das variáveis de estado, que por sua vez são derivadas dos parâmetros do evento corrente, ou dos eventos dos estados anteriores. Condições também podem usar operadores para construir expressões mais complexas. Por exemplo, a expressão `v1 == v2` retorna verdadeiro se `v1` é igual a `v2`, e `v contains 'foo'` retorna verdadeiro se `v` contém a string "foo".

3. **Ações.** Uma ação especifica uma seqüência de operações de arquitetura para criar ou modificar a arquitetura que está sendo recuperada do sistema em execu-

[1] Definição padrão de máquina de estados: conjunto de estados e conjunto de função de transição.

ção. Ações estão diretamente ligadas ao estilo da arquitetura alvo, e são expressas usando operações apropriadas do estilo de arquitetura. Por exemplo, um estilo dutos e filtros deve incluir operações para criar *Dutos e Filtros*; um estilo *Cliente e servidor*, oferecer operações para criar e conectar clientes a servidores. Semelhante aos parâmetros dos eventos, operações podem definir valores explicitamente por meio da atribuição, e a variável é gravada no estado corrente para que possa ser usada em ações ou condições posteriores.

4. **Transições.** Ocorrem quando a condição de um gatilho é satisfeita. Ao ocorrer uma transição, ligam-se todas as variáveis do estado de destino com os valores parametrizados do evento observado e as variáveis definidas pela ação, caso exista. Dessa forma, uma variável v está presente em um estado s se, para cada transição de entrada em s, v é definida na transição ou está presente nos estados predecessores. Condições e ações de transições que saem de s podem referenciar variáveis presentes em s, bem como quaisquer novas variáveis definidas pela transição.

Para lidar com sequências de eventos intercalados, DiscoTect mantém concorrentemente mais de um estado ativo (chamado de ativação de estado) na máquina de estados. Cada ativação de estado é um par constituído de um estado e todas as variáveis ligadas naquele estado. DiscoTect provê três tipos de transições: *ordinary*, *fork* e *join*. Como em outras máquinas de estado, transições *ordinary* removem uma ativação de estado e adicionam outra, ligando todas as variáveis a partir da ativação de estado anterior. Para suportar concorrência, DiscoTect provê as transições *fork*, que mantém a ativação de estado original enquanto cria uma nova ativação de estado, em paralelo com a original, ligando as variáveis. O outro tipo de transição é *join*: funde duas ou mais ativações de estado em uma única ativação, ligando as variáveis.

O estado corrente da **State Engine** é um conjunto de ativações de estado. A **State Engine** começa com uma única ativação de estado que é o estado inicial da máquina de estados e nenhuma variável ligada. Toda vez que um evento é recebido do **Trace Engine**, ele é casado com todas transições de saída do estado corrente de cada ativação de estado. Se o evento casar com a especificação de evento do gatilho para uma ou mais transições, e a condição avaliar para verdadeiro, a transição associada ao gatilho é escolhida.

Para transições *ordinary*, a ativação fonte é removida e a nova ativação é adicionada para o estado de destino. As variáveis no novo estado são ligadas aos valores dos parâmetros definidos na transição, mantendo as variáveis da ativação anterior.

Se a transição é um *fork*, então a máquina mantém a ativação de estado origem, enquanto cria uma nova ativação de destino. Se a transição é um *join*, ela só pode ser

disparada se existe uma ativação de estado presente para cada estado origem da união. Neste caso, todas as ativações fontes são removidas, e a ativação de destino é criada, como de costume.

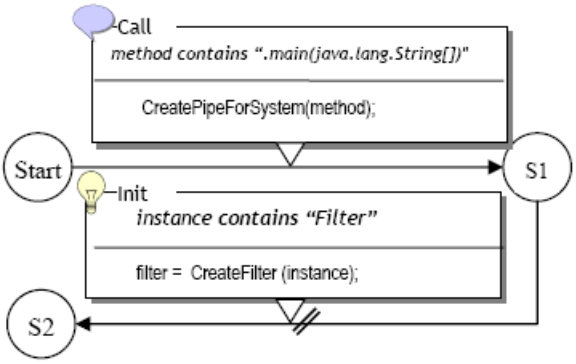

Figura 3.2. Parte da Máquina de Estados para descobrir os Filtros

Yan et al. [34] apresentam exemplos de máquinas de estados para recuperação da arquitetura de um sistema baseado no estilo *Dutos e Filtros*. Uma parte da máquina de estados pode ser visualizada na Figura 3.2, extraída de [34]. Essa parte identifica a criação de filtros. Basicamente, a ação para criar o sistema de Dutos e Filtros ocorre quando o **Trace Engine** envia o evento `Call(method="v1.RegSys.main(java.lang.String[])", caller=null, callee=null)` para a **State Engine**. Como a transição de *Start* para $S1$ é do tipo *ordinary*, o estado corrente passará a ser $S1$, e os parâmetros do evento `Call` serão ligados às variáveis do estado $S1$. Assim, o conjunto de ativações de estado corrente será:

{[$S1$, { ($S1.method$,`"v1.RegSys.main(java.lang.String[])"`),
($S1.caller$,`null`),
($S1.callee$,`null`) }]
}

Para mostrar um exemplo de transição *fork*, suponha que o **Trace Engine** produza o evento `Init(constructor="v1.SplitFilter", creator=null, instance="v1.SplitFilter(id=342)")`. Ocorrerá a transição de $S1$ para $S2$ (notação utilizada para transição *fork* é acrescentar um // na aresta do grafo) e o novo conjunto de ativações de estado será:

{[$S1$, { ($S1.method$,`"v1.RegSys.main(java.lang.String[])"`),
($S1.caller$,`null`),
($S1.callee$,`null`) }]

$[S2, \{ (S1.method,$"v1.RegSys.main(java.lang.String[])"$),$
$(S1.caller,$null$),$
$(S1.callee,$null$),$
$(S2.constructor,$"v1.SplitFilter"$),$
$(S2.creator,$null$),$
$(S2.instance,$"v1.SplitFilter(id=342)"$),$
$(S2.filter,$Filter("v1.SplitFilter(id=342)")$) \}]$
}

Note que a ação CreateFilter(name) acrescenta uma nova variável ao estado $S2$ chamada $filter$. E é assim que os componentes filtros serão descobertos.

O artigo de Yan et al. [34] também apresenta um outro exemplo, de um *software* real, no qual pode-se identificar inconsistências entre a arquitetura projetada e a recuperada.

3.2 Conclusão

Apesar de apresentar alguns resultados expressivos, DiscoTect não apresenta uma metodologia para descobrir quais são os principais componentes dos sistemas a serem recuperados. Durante o processo de construção da máquina de estados para recuperar a arquitetura do sistema, é proposta uma etapa comum como sendo a construção de uma máquina de estado que captura qualquer tipo de evento para meramente observar a criação de objetos e a interação entre eles via chamadas de métodos. Para sistemas simples e de pequeno porte, talvez seja possível obter indícios de quais sejam os principais componentes do sistema. Porém, para sistema de grande porte onde ocorrem milhares de eventos, a análise da descrição da arquitetura obtida será impraticável, se essa máquina de estados for utilizada.

Outro fator crítico dessa técnica é a construção da máquina de estado que recupera a arquitetura. É sugerida a construção incremental, sempre testando a execução do sistema e acrescentando novos gatilhos à definição da máquina de estados anterior. Dessa maneira, o resultado obtido depende das habilidades de quem está construindo a máquina de estados e da parte do código-fonte que foi instrumentado. Sobre a questão de analisar os eventos produzidos pela execução do sistema, essa técnica só é capaz de recuperar a arquitetura do sistema dos trechos de código que foram executados. Portanto, questões como se existe uma execução do sistema que produzirá uma arquitetura diferente não podem ser respondidas por esse método.

DiscoTect também não deixa claro qual o conhecimento acerca da implementação do sistema que quem escreve as máquinas de estado deve ter. Provavelmente quem implementou ou estudou a implementação do sistema poderá ser induzido a construir uma máquina de estado que recupere a arquitetura pré-definida. Caso a pessoa desconheça completamente a implementação, a etapa de construir uma máquina de estado que capture todo e qualquer evento do sistema em execução produzirá uma descrição de arquitetura tão repleta de detalhes quanto o código-fonte. Desta maneira, fica subentendido que a pessoa gastará um certo tempo tentando entender a implementação do sistema.

Essa abordagem tem a desvantagem de que os mapeamentos de padrões de implementação que produzem as ações de arquitetura estão fortemente ligados a uma convenção de codificação regular, ou seja, a uma nomenclatura de tipos padronizada. Por exemplo, a modificação do nome de uma classe pode invalidar a máquina de estados responsável por recuperar uma determinada arquitetura, caso o nome da classe tenha sido utilizado em alguma condição de um gatilho.

Essas conclusões indicam algumas possibilidades de melhorias para uma técnica de recuperação da arquitetura de *software*:

- Importância de uma metodologia para recuperação da arquitetura de software (Seção 5.3);
- Melhoria na fase de identificação dos principais componentes e conectores do sistema sem a necessidade de executar o programa diversas vezes.

Capítulo 4

Verificação de Conformidade

Este capítulo apresenta uma técnica para verificar consistência entre a arquitetura de *software* e implementação do sistema. ArchJava [2] é uma extensão de Java que unifica a arquitetura de *software* com a implementação de um sistema, utilizando um sistema de tipos para assegurar que a implementação está de acordo com as restrições arquiteturais, onde ferramentas para análise podem verificar por conformidade. O propósito da linguagem ArchJava é investigar os benefícios e as desvantagens de uma parte inexplorada das LDAs: a possibilidade de implementar a arquitetura descrita em uma linguagem de programação com o intuito de manter a integridade de comunicação entre os seus componentes.

4.1 ArchJava

ArchJava [2] é uma extensão de Java que unifica a arquitetura de *software* com a implementação de um sistema. Dessa maneira, um programa escrito nessa linguagem está seguro de que a implementação está de acordo com as restrições arquiteturais. O objetivo de ArchJava é garantir a integridade de comunicação entre os componentes da arquitetura, ou seja, garantir que os componentes implementados só se comuniquem diretamente com os componentes aos quais eles estão conectados na arquitetura. Em ArchJava, a integridade de comunicação define como os componentes da arquitetura se comunicarão em tempo de execução, através de uma das cinco categorias a seguir:

- **Unique Communication**: um objeto A invoca um método de um componente B que é especificado como único (*unique*);
- **Parent-Child Communication**: um objeto A invoca um método do seu sub-componente B;

- **Connection Communication**: um componente A invoca um método de um componente B, através de um padrão de conexão (*connect pattern*) na instância do componente, que diretamente possui A e B;
- **Lent Communication**: um objeto ou componente A invoca um método, especificado como *lent* ou como um parâmetro de um método de um objeto não-componente que foi temporariamente emprestado (*lent*) para A;
- **Shared Data Communication**: um objeto A acessa algum objeto B em um domínio arquitetural diferente, e A e B possuem seus domínios arquiteturais relacionados.

Para permitir que os programadores descrevam a arquitetura de *software* na implementação, ArchJava adicionou novas construções à linguagem Java para suportar componentes, conectores e portas.

- **Componente (*Component*)**: é um tipo especial de objeto que se comunica com outros componentes de forma estruturada. Componentes são instâncias de classes `Component`. Um componente só pode se comunicar com outros componentes no nível de arquitetura por meio de portas declaradas explicitamente: não é permitido chamadas de métodos entre componentes.
- **Porta (*Port*)**: uma porta representa um canal lógico de comunicação entre um componente e um ou mais componentes ao qual ele está conectado. Portas declaram três conjuntos de métodos, por meio das palavras-reservadas: `requires`, `provides` e `broadcasts`. Um método que o componente fornece (*provides*) será implementado por ele e estará disponível para ser chamado por outro componente que esteja conectado a esta porta. Inversamente, cada método necessário (*requires*) deverá ser implementado por outro componente, conectado a esta porta. Métodos `broadcasts` são semelhantes a métodos necessários, exceto que eles podem ser conectados a qualquer número de implementações e devem retornar `void`.
- **Conector *Connect***: para fazer a composição de componentes, ArchJava oferece a construção `connect`, que conecta duas ou mais portas, ligando cada método necessário a um método que o componente provê, com mesmo nome e assinatura.

A seguir será apresentado um exemplo de componente em ArchJava:

```
1 public component class Parser {
2   public port in {
```

```
3       provides void setInfo(Token symbol, SymTabEntry e);
4       requires Token nextToken() throws ScanException;
5     }
6     public port out {
7       provides SymTabEntry getInfo(Token t);
8       requires void compile(AST ast);
9     }
10    void parse(String file) {
11      Token tok = in.nextToken();
12      AST ast = parseFile(tok);
13      out.compile(ast);
14    }
15    AST parseFile(Token lookahead) { ... }
16    void setInfo(Token t, SymTabEntry e) {...}
17    SymTabEntry getInfo(Token t) { ... }
18    ...
19  }
```

No exemplo dado declara-se um componente chamado `Parser` composto de duas portas: `in` e `out`. O componente implementa os métodos `setInfo()` e `getInfo()` que suas portas provêem, requer que o método `nextToken()` seja implementado pelo componente que será conectado à porta `in` e que o componente que será conectado à porta `out` implemente o método `compile()`.

Em ArchJava a hierarquia de uma arquitetura é expressa por meio da composição de componentes. Esta composição é conseguida por um número de sub-componentes conectados. Um sub-componente é uma instância de um componente em um outro componente. Para mostrar um exemplo de composição de componentes, o exemplo abaixo mostrará a construção de um componente compilador em ArchJava, constituído de três sub-componentes:

```
1  public component class Compiler {
2    private final Scanner scanner = ...;
3    private final Parser parser = ...;
4    private final CodeGen codegen = ...;
5
6    connect scanner.out, parser.in;
7    connect parser.out, codegen.in;
8
9    public static void main(String args[]) {
10     new Compiler().compile(args);
11   }
12   public void compile(String args[]) {
13     // for each file in args do:
```

```
14      ...parser.parse(file);...
15    }
16  }
```

Esse exemplo mostra que o *parser* se comunica com o *scanner* usando um protocolo e com o *code generator* usando outro. A arquitetura implica que o *scanner* não se comunica diretamente com o *code generator*.

O compilador de ArchJava transforma os programas escritos em ArchJava em código Java. Durante esta transformação, ele garante a integridade de comunicação entre componentes. Existem dois tipos de violação de integridade que o compilador verifica:

1. Chamada de métodos direta entre componentes: um componente só pode invocar diretamente um método se este método estiver na classe do componente, ou pertencer imediatamente a um de seus subcomponentes.

2. Chamada de métodos por meio de portas: o sistema de tipos de ArchJava proibe chamadas de métodos que violam as restrições arquiteturais.

A abordagem de ArchJava apresenta os seguintes benefícios:

- argumentação confiável em relação à arquitetura;
- código fonte e descrição da arquitetura consistentes;
- encorajamento dos desenvolvedores para usufruir das vantagens da arquitetura de *software*.

4.2 Conclusão

A solução apresentada por ArchJava é uma forma de verificar consistência entre os componentes do sistema. Entretanto, esta abordagem possui as seguintes deficiências:

- A descrição da arquitetura do sistema fica espalhada nas classes dos componentes, juntamente com o código fonte, dificultando o entendimento global do programa.
- A solução é totalmente intrusiva: para um sistema existente, todas as classes que forem componentes deverão ser modificadas para se adequarem a ArchJava.
- As restrições arquiteturais de ArchJava se restrigem à integridade de comunicação e a linguagem não oferece elementos para verificar conformidade utilizando padrões de nome de tipos ou subtipos do sistema implementado.

Além destas deficiências, ArchJava possui as seguintes limitações, segundo Aldrich [2]:

- ArchJava só pode ser utilizada em aplicações *stand-alone*, em uma única *Java Virtual Machine* (JVM).
- Não permite a integridade de comunicação via dados compartilhados.
- como ArchJava está focada em tratar a integridade de comunicação, ainda não são suportadas outras características arquiteturais, tais como protocolos de conexão, estilos arquiteturais e multiplicidade de componentes.

Capítulo 5

ArchRecover e ArchVerifier

Este capítulo apresenta a solução proposta para auxiliar os engenheiros de *software* na manutenção de sistemas de *software* [9]:

(i) uma ferramenta denominada *ArchRecover* que, por meio de programação de aspectos [20] instrumenta o código-fonte de forma a auxiliar na recuperação da arquitetura de *software* de programas escritos em Java, durante as suas execuções;

(ii) uma linguagem para especificar restrições entre componentes da descrição da arquitetura de *software*;

(iii) uma metodologia para recuperação da arquitetura de software;

(iv) um mecanismo baseado em aspectos chamado *ArchVerifier* para verificar conformidade entre a arquitetura e a implementação do sistema.

A Figura 5.1 apresenta os principais processos das ferramentas que foram implementadas para auxiliar na manutenção de sistemas, a fim de fornecer uma visão geral do trabalho proposto:

1. O primeiro processo é a recuperação da arquitetura: o sistema é executado em conjunto com um aspecto que em tempo de execução intercepta os eventos do programa (execução de métodos, execução de construtores e execução de blocos inicializadores estáticos de classe) gerando uma representação da execução do programa, chamada de **árvores de execução**. Por meio de consultas às árvores de execução, o usuário pode construir de forma interativa a representação da arquitetura do programa, adicionando os principais componentes e conectores. A

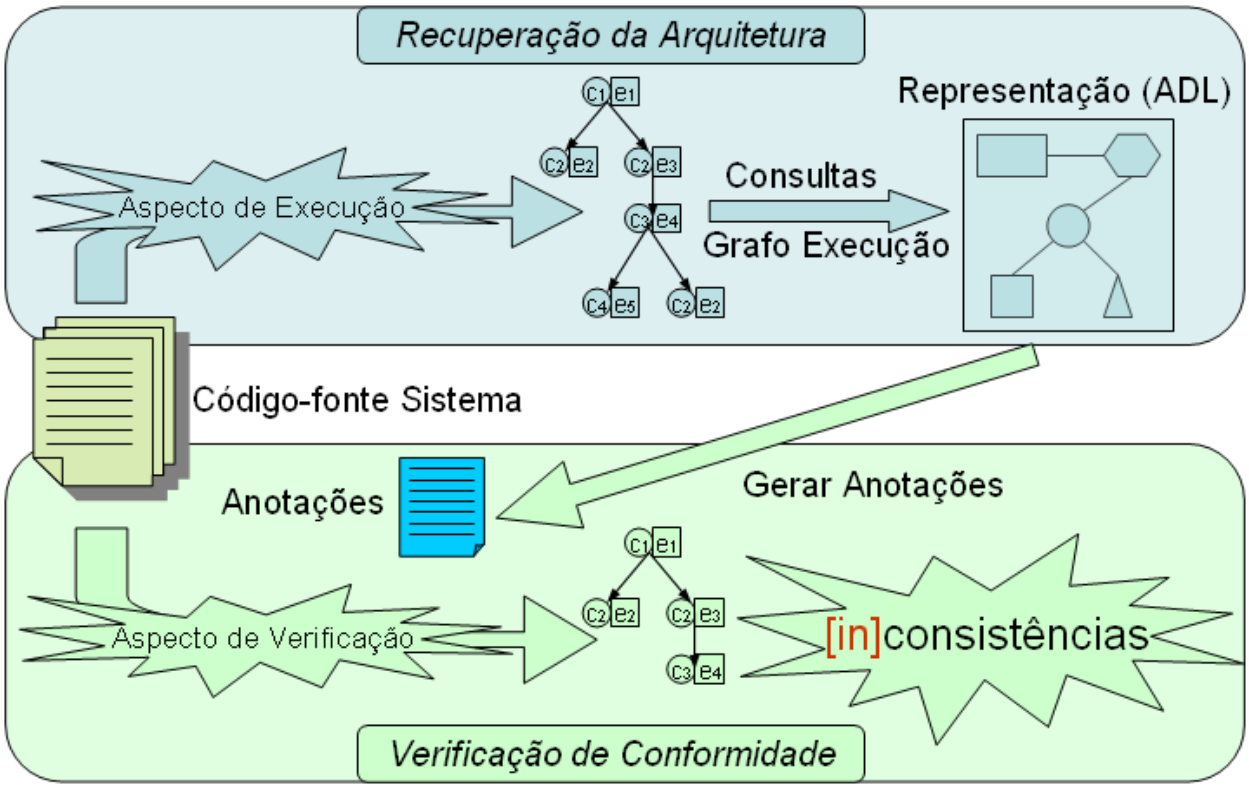

Figura 5.1. Processos das Ferramentas

metodologia de recuperação da arquitetura de *software* proposta está detalhada na Seção 5.3. Além disto, o usuário pode acrescentar restrições arquiteturais aos conectores da descrição da arquitetura, as quais serão utilizadas nos próximos processos da ferramenta *ArchVerifier*.

2. O segundo processo é a anotação do código-fonte. De posse da representação da arquitetura com algumas restrições arquiteturais descrita em uma LDA, a ferramenta *ArchVerifier* gera um aspecto para anotar o código-fonte do programa nos pontos-chave das restrições arquiteturais. Esse aspecto deverá ser utilizado pelo próximo processo para verificar conformidade entre a descrição da arquitetura e a implementação do sistema. A Seção 5.5 apresenta o gerador de anotações.

3. O terceiro processo é a verificação de conformidade: em resumo, o programa deverá ser executado em conjunto com dois aspectos, o que foi gerado pelo processo anterior que declara as anotações nos pontos-chave do programa e o aspecto de verificação que verifica conformidade entre arquitetura e a implementação atual do sistema. Assim como o aspecto de execução, o aspecto de verificação também constrói as árvores de execução do sistema e ao identificar os pontos-chave das

restrições arquiteturais, o aspecto de verificação analisa se uma determinada restrição é satisfeita pelas árvores de execução do programa e produz um relatório de consistências ou inconsistências.

Para isto, foram implementadas duas ferramentas:

- *ArchRecover*: ferramenta para auxiliar o engenheiro de *software* no processo de recuperação da arquitetura de *software*. A inspiração para construção dessa ferramenta surgiu ao identificar uma importante etapa do processo de recuperação da arquitetura de *software* da técnica DiscoTect [34]: criar uma máquina de estados para meramente observar a criação de objetos e a interação entre eles através das chamadas de métodos. A idéia principal utilizada pela ferramenta *ArchRecover* é criar uma representação da execução do sistema que será mostrada na Seção 5.1 e permitir que o usuário consulte informações sobre essa representação para ajudá-lo a identificar quais são os principais componentes da arquitetura de *software* e quais os relacionamentos desses componentes com os outros.
- *ArchVerifier*: ferramenta para verificar conformidade entre a descrição da arquitetura de um sistema que possua restrições arquiteturais e a implementação do sistema em tempo de execução. Isto é feito através de um passo intermediário que a partir da descrição da arquitetura do sistema gera anotações nos pontos-chave do sistema que pertencem às restrições arquiteturais. Executando-se o programa com o aspecto que contenha as anotações e o aspecto de verificação de conformidade, pode-se encontrar inconsistências entre a descrição da arquitetura e a implementação atual do sistema.

Para isso, foi modelada uma representação da execução de programas Java a fim de utilizá-la tanto na fase de recuperação da arquitetura de *software* quanto na fase de verificação de conformidade. Isto será mostrado na próxima seção.

5.1 Representação da Execução de Programas Java

Uma prática comum na metodologia de recuperação da arquitetura de *software* da técnica DiscoTect [34] é a construção da máquina de estados que capture qualquer criação de objetos e chamadas de métodos a fim de identificar os componentes e conectores da arquitetura do sistema. Embora a ideia pareça simples, o resultado produzido pode ser inviável de ser analisado, tendo em vista que durante a execução de um programa inúmeros objetos são criados e ocorrem várias chamadas de métodos. Seria necessário refinar o resultado, filtrando as informações obtidas, sendo que a cada passo do

refinamento seriam descartados os eventos desnecessários para a arquitetura do sistema. Em DiscoTect [34] isto é feito inicialmente criando-se uma máquina de estado para capturar qualquer tipo de evento. Em seguida, a cada passo de refinamento, acrescentam-se novos gatilhos à definição da máquina de estados anterior e executa-se o sistema novamente até que se tenha a máquina de estado final que recuperará a descrição da arquitetura do sistema. Outro empecilho que dificulta esse processo de refinamento é a dificuldade de reproduzir a mesma execução do programa, em cada passo do refinamento.

Com o objetivo de facilitar a etapa de refinamento dos eventos pertinentes à arquitetura do sistema, esse trabalho propõe uma representação da execução de programas Java. Essa representação permite que vários refinamentos sejam feitos para uma execução do programa em vez de executar o programa para cada novo refinamento. A Seção 5.1.1 apresenta detalhes da execução de programas Java.

5.1.1 Execução de Programa Java

Um programa em Java é um conjunto de classes compiladas em arquivos `.class`, os quais podem estar em diretórios locais do computador, em diretórios remotos ou empacotados em arquivos `.jar`.

A execução de programas Java é feita na Máquina Virtual Java (*Virtual Java Machine - JVM*), responsável por interpretar os arquivos `.class` na plataforma subjacente. O único contrato necessário para que um programa possa ser executado pela *JVM* é que possua uma classe que implemente o método `static void main( String[])`. Ao executar a *JVM* passando como parâmetro uma classe que possua o método principal, a Máquina Virtual Java inicia um novo ambiente de execução, carrega a classe especificada e invoca o método `static void main(String[])`.

A *JVM* permite que um programa tenha múltiplas linhas de execução concorrentes, por meio do mecanismo de *MultiThreading*. Quando uma *JVM* é inicializada, existe usualmente uma *Thread* principal, a qual executará o método `main(String[])` da classe designada. A *JVM* continuará a execução das *Threads* do programa até que ocorra uma das duas condições abaixo:

- O método `void exit(int)` da classe `Runtime` é invocado e o gerenciador de segurança permite a operação de término do programa;
- Todas *Threads* que não são *daemon threads* terminaram de executar, seja por ter retornado da chamada do método `void run()` ou por ter lançado uma exceção que propagou além do método `void run()`.

A seguir, serão apresentadas duas classes de um exemplo de programa em Java para mostrar a sua execução detalhada. O principal objetivo da explicação é identificar os diversos eventos de um programa Java, enfocando os pontos do programa onde eles são executados. Além disto, tem-se a expectativa de que esses pontos do código poderão contribuir para entender a arquitetura de *software* do programa. Então, sejam as seguintes classes:

```
1  public abstract class ExecutionPoints {
2    private static int instances = 0;
3    private int id;
4    private static int idNewInstance() { return ++instances; }
5    static { System.out.println("ExecutionPoints.staticinitialization"); }
6    public static void main(String[] args) {
7      System.out.println("ExecutionPoints.method");
8      ExecutionPoints sc = new SubExecutionPoints();
9      sc.m1();
10   }
11   { System.out.println("ExecutionPoints.initialization"); }
12   public ExecutionPoints() {
13     System.out.println("ExecutionPoints.constructor");
14     this.id = idNewInstance();
15     System.out.printf("ExecutionPoints.id = %d\n", this.id);
16   }
17   public abstract void m1();
18 }
19
20 class SubExecutionPoints extends ExecutionPoints {
21   static { System.out.println("SubExecutionPoints.staticinitialization"); }
22   { System.out.println("SubExecutionPoints.initialization"); }
23   public SubExecutionPoints() {
24     System.out.println("SubExecutionPoints.constructor");
25   }
26   public void m1() {
27     System.out.println("SubExecutionPoints.method");
28   }
29 }
```

Listing 5.1. Arquivo ExecutionPoints.java

Se esse arquivo for compilado e em seguida for executado pela máquina virtual Java por meio do comando `java ExecutionPoints`, ocorrerão os seguintes eventos durante a execução do programa:

1. A *JVM* criará um novo ambiente de execução e iniciará a *thread* principal, a qual executará o método `ExecutionPoints.main(String[])`;
2. A classe `ExecutionPoints` será então carregada pela *JVM*;
3. Antes de iniciar a execução do método `ExecutionPoints.main(String[])`, ocorre a inicialização estática da classe `ExecutionPoints`:

a) a variável estática `instances` na linha 2 será inicializada com o valor `0`;

b) o bloco de inicialização estática da classe na linha 5 será executado e invocará o método `java.io.PrintStream.println(String)`, que imprime a `String` "ExecutionPoints.staticinitialization" na saída padrão;

4. Em seguida, o método `ExecutionPoints.main(String[])` será executado:

a) na linha 7, o método `java.io.PrintStream.print(String)` será invocado e imprimirá a `String` "ExecutionPoints.method" na saída padrão;

b) na linha 8, será invocado o construtor da classe `SubExecutionPoints()`:

i. acontecerá a inicialização estática da classe `SubExecutionPoints`, na linha 21, onde o método `java.io.PrintStream.println(String)` será invocado, e imprimirá a `String` "SubExecutionPoints.staticinitialization" na saída padrão;

ii. em seguida, ocorre a pré-inicialização do objeto `SubExecutionPoints()`, ou seja, a chamada implícita do construtor `super()`:

A. acontecerá a inicialização do objeto `ExecutionPoints`, na linha 11, onde o método `java.io.PrintStream.println(String)` será invocado, e imprimirá a `String` "ExecutionPoints.initialization" na saída padrão;

B. então, a execução do construtor `ExecutionPoints()`, definido na linha 12:

- na linha 13, o método `java.io.PrintStream.print(String)` será invocado e imprimirá a `String` "ExecutionPoints.constructor" na saída padrão;
- na linha 14, o método estático `idNewInstance()` da classe é invocado. A variável estática `instances` será incrementada de uma unidade, na linha 4, e o método retornará o novo valor dessa variável. O valor `1` será então atribuído à variável de instância `id`.
- na linha 15, o método `java.io.PrintStream.printf(String, int)` será invocado e imprimirá a `String` "ExecutionPoints.id = 1" na saída padrão;

iii. ocorrerá a inicialização do objeto `SubExecutionPoints`, na linha 22, onde o método `java.io.PrintStream.println(String)` será invo-

cado, e imprimirá a `String` "SubExecutionPoints.initialization" na saída padrão;

iv. para então executar o corpo construtor `SubExecutionPoints()`, definido na linha 23:

A. o método `java.io.PrintStream.println(String)` será invocado na linha 24, e imprimirá a `String` "SubExecutionPoints.constructor" na saída padrão;

c) o método `m1()` do objeto `sc` é invocado na linha 9:

i. o método `java.io.PrintStream.println(String)` será invocado na linha 27, e imprimirá a `String` "SubExecutionPoints.method" na saída padrão;

d) Termina a execução do método `Test.main(String[])`;

5. Termina a execução da única *thread*;

6. O programa termina!

Após essa explicação detalhada da execução de um programa em Java, pode-se analisar os eventos observados segundo duas dimensões:

- A primeira dimensão se refere ao ponto do código onde esse evento foi executado. A essa dimensão será dado o nome de **Ponto de Execução**;

- A segunda dimensão se refere ao escopo no qual o trecho de código foi executado. A essa dimensão será dado o nome de **Contexto**.

Analisando a execução desse programa, identificam-se alguns pontos de execução importantes:

- execução de métodos: todas as operações que ocorrem dentro do bloco de um método de uma classe.

- execução de construtores: todas as operações que ocorrem dentro do bloco de um construtor de uma classe.

- execução de inicializador estático de classe: todas as operações que ocorrem dentro do bloco de inicialização estática de uma classe.

Foram escolhidos esses três pontos de execução pois são nos blocos dos métodos, construtores e inicializadores estático de classe que o código de um programa é efetivamente

executado. Em DiscoTect [34] existe um outro tipo de evento, modificação de atributos, que não será assim considerado nesse trabalho, pois os comandos de atribuição de um atributo de uma classe ocorrerão dentro de algum de seus construtores ou dentro de algum de seus métodos.

O escopo dos três pontos de execução citados acima define o contexto de execução, o qual pode ser de 2 tipos:

- contexto de classe: escopo de classe, referente aos blocos de inicialização estática de classe e aos métodos estáticos de uma classe.
- contexto de objeto: escopo de objeto ou das instâncias de uma classe, referentes aos contrutores e aos métodos não estáticos de uma classe.

Os contextos servem para identificar possíveis instâncias dos componentes da arquitetura de um sistema e os pontos de execução podem representar interações entre esses componentes. Os contextos são importantes, pois existe a necessidade de diferenciar as várias instâncias de um mesmo objeto, pois cada uma pode exercer um papel ou função específico, durante a execução do programa.

5.1.2 Pontos de Execução e Contextos

A linguagem AspectJ [20] introduz o conceito de pontos de junção (do inglês *join point*), que são pontos bem-definidos durante a execução de um programa. Existe uma série de tipos de pontos de junção definidos na linguagem, os quais podem ser divididos em três grupos:

- pontos de junção de chamada:
  - *Method call*: quando um método é invocado, não incluindo chamadas a métodos não estáticos da superclasse.
  - *Constructor call*: quando um objeto é construído e o construtor inicial do objeto é invocado, não incluindo chamadas a construtores da classe herdada ou chamadas a outros construtores da classe.
- pontos de junção de execução:
  - *Method execution*: quando o código do corpo de um método executa.
  - *Constructor execution*: quando o código do corpo de um construtor executa, depois da chamada de seu construtor `this` ou `super`.

  - *Static initializer execution*: quando o bloco de inicialização estática de uma classe executa.
  - *Object pre-initialization*: antes que o código de inicialização de um objeto de uma determinada classe execute. Compreende o tempo entre o início do primeiro construtor invocado e o início do construtor da classe herdada.
  - *Object initialization*: quando o código de inicialização de um objeto executa. Compreende o tempo entre o retorno do construtor da super classe e o retorno do primeiro construtor invocado.
  - *Handler execution*: quando um tratador de exceções executa.
  - *Advice execution*: quando o código do corpo de um adendo (do inglês *advice*) executa.

- pontos de junção de referências:
  - *Field reference*: quando um campo que não é uma constante é referenciado.
  - *Field set*: quando algum valor é atribuído a um campo.

A Tabela 5.1 apresenta exemplos de ocorrências dos pontos de junção definidos por AspectJ no programa de exemplo da Listagem 5.1.

Ponto de Junção	Ocorrências
Method call	Linhas 5, 7, 9, 11, 13, 14, 15, 21, 22, 24 e 27.
Constructor call	Linha 8.
Method execution	Corpo dos métodos das classes `ExecutionPoints` e `SubExecutionPoints`.
Constructor execution	Corpo dos construtores das classes `ExecutionPoints` e `SubExecutionPoints`.
Static initializer execution	Linhas 5 e 21.
Object pre-initialization	Avaliação dos argumentos para chamada do construtor da super classe de `SubExecutionPoints`.
Object initialization	Linhas 11 e 22.
Field reference	Linha 15.
Field set	Linha 14.

Tabela 5.1. Exemplos de pontos de Junção.

A próxima seção define um modelo para representação da execução de programas em Java chamado **árvores de execução**, o qual utiliza alguns dos pontos de execução existentes em AspectJ.

5.1.3 Definição das Árvores de Execução

As árvores de execução de um programa em Java são um grafo $G(V, E)$ no qual:

- $v \in V \Leftrightarrow v = (c, p)$:
  - c é um contexto para o qual o ponto de execução p foi executado;
  - p é um ponto de execução.

  Se o ponto de execução não é estático então o contexto é uma instância de uma classe, ou seja, um objeto. Caso o ponto de execução seja estático, o contexto será representado pelo objeto `Class` da classe.
- para cada `Thread` t de um programa, existe um vértice $v_t = (t,$ `java.lang.Thread.run()`$)$;
- existe uma aresta entre $v_0 = (c_0, p_0)$ e $v_1 = (c_1, p_1)$ se durante a execução de p_0 no contexto c_0, $c_1.p_1$ foi executado.

  Os pontos de execução podem ser de três tipos:

1. chamada de métodos;
2. chamada de construtores;
3. bloco de inicialização estática de classe.

A fim de exemplificar a construção de um grafo de execução, será utilizado a parte inicial do programa da Listagem 5.1 e a descrição detalhada de sua execução:

1. A *JVM* criará um novo ambiente de execução e iniciará a *thread* principal, a qual executará o método `ExecutionPoints.main(String[])`. Nesse momento, é criado o primeiro vértice do grafo de execução, referente ao método `run()` da *thread* principal, mostrado na Figura 5.2.

Figura 5.2. Grafo de execução inicial

2. A classe `ExecutionPoints` será então carregada pela *JVM*. Um novo vértice é inserido no grafo como filho do vértice que está executando nesse momento, ou seja, o método `run()` da *thread* principal. A Figura 5.3 apresenta o grafo de execução nessa situação.

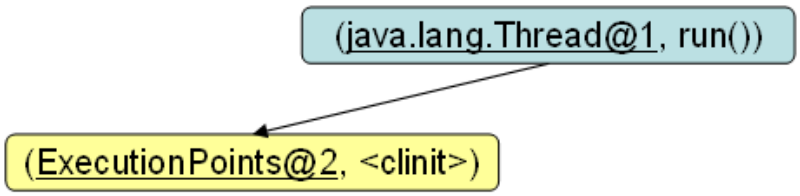

Figura 5.3. Grafo de execução ao carregar a classe ExecutionPoints

3. O bloco de inicialização estática da classe ExecutionPoints será executado e invocará o método `java.io.PrintStream.println(String)`. A representação do grafo de execução nesse momento pode ser visualizada na Figura 5.4.

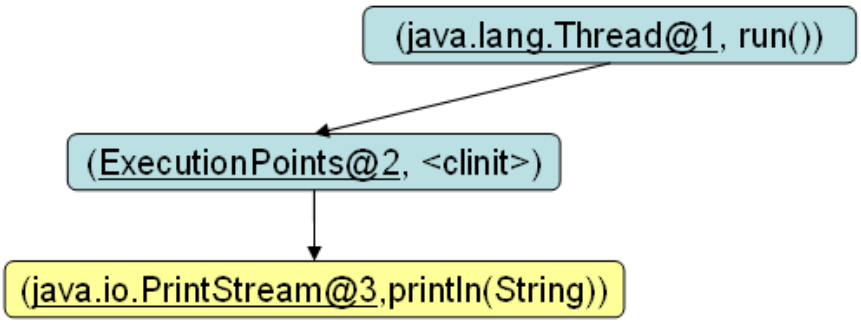

Figura 5.4. Grafo de execução ao executar o bloco de inicialização estática da classe ExecutionPoints

O grafo de execução do programa é construído pelo aspecto de execução. Esse aspecto intercepta os seguintes pontos de junção dos programas escritos em java:

- todas as inicializações estáticas de classe;
- todas as chamadas de métodos estáticos de classe;
- todas as chamadas de métodos de objetos;
- todas as chamadas de criação de objetos.

5.1.4 Outra abordagem: Hipergrafo de Execução

Uma outra abordagem para representar a execução de programas Java é definir o grafo de execução como hipergrafo. Primeiramente, será explicitado o conceito de um hipergrafo para em seguida apresentar a modelagem do hipergrafo de execução.

Um hipergrafo $H(V, E)$ é definido pelo par de conjuntos V e E, onde:

- V: conjunto de vértices, $V = \{v_1, \ldots, v_n\}$;

- E: conjunto de arestas, onde uma aresta $e \in E \Leftrightarrow e = \{v_1, \ldots, v_m\}$ e $m \geq 2$.

A principal diferença de um hipergrafo para um grafo é que uma aresta pode conectar mais de dois vértices, acrescentando a noção de multidimensão aos grafos simples.

Seja, por exemplo, o hipergrafo $H_1(V_1, E_1)$, representado na Figura 5.5, dado por:

- $V_1 = \{x_1, x_2, x_3, x_4\}$
- $E_1 = \{\{x_1, x_2, x_4\}, \{x_2, x_3, x_4\}, \{x_2, x_3\}\}$

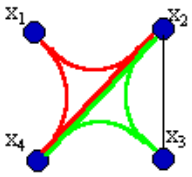

Figura 5.5. Hipergrafo H_1

A definição do hipergrafo de execução de um programa em Java é um grafo $H(V, E)$ no qual:

- $V = C \cup P$, onde:
  - C é o conjunto de todos os contextos de um programa;
  - P é o conjunto de pontos de execução de um programa.

  Se o ponto de execução pertence às instâncias de uma classe então o contexto é um objeto dessa classe. Caso o ponto de execução seja estático, ou seja, pertença à classe, o contexto será o objeto `Class` que representa essa classe.

- E é o conjunto de arestas, onde:
  - para cada *Thread* t de um programa existe uma aresta $e_t = \{$`java.lang.Thread` t, `run`$()\}$;
  - existe uma aresta $e = \{c_0, p_0, c_1, p_1\}$ onde $c_0 \in C$, $c_1 \in C$ e $\{p_0, p_1\} \subseteq P$ se durante a execução do ponto p_0 sobre o contexto c_0, $c_1.p_1$ é executado.

A vantagem do hipergrafo de execução é que ele pode ser visualizado em duas dimensões: (i) a dimensão de todos os contextos criados na execução do programa e (ii) a dimensão que contém todos os pontos de execução do programa, formando uma floresta de árvore de execução, sendo uma árvore para cada *Thread*.

O hipergrafo $H_e(V_e, E_e)$ referente ao início da execução do programa, até o passo 4a, é:

- $V_e = \{c_0, c_1, c_2, p_0, p_1, p_2, p_3\}$ onde:
  - c_0 = `java.lang.Thread@89ae9e`;
  - c_1 = `ExecutionPoints.class@15fea60`;
  - c_2 = `java.io.PrintStream@250c3a`;
  - p_0 = `run()`;
  - p_1 = `< clinit > ()`;
  - p_2 = `println(java.lang.String)`;
  - p_3 = `main(java.lang.String[])`;
- $E_e = \{\{c_0, p_0\}, \{c_0, p_0, c_1, p_1\}, \{c_1, p_1, c_2, p_2\}, \{c_0, p_0, c_1, p_3\}, \{c_1, p_3, c_2, p_2\}, \ldots\}$.

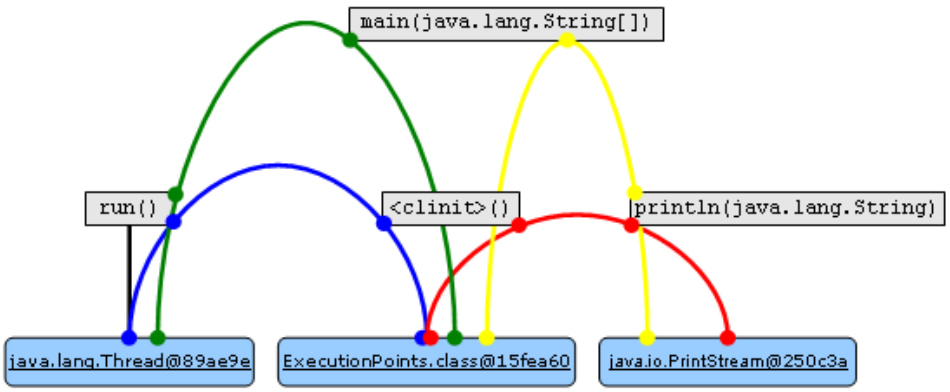

Figura 5.6. HiperGrafo de execução H_e do programa `ExecutionPoints`

A representação de programas Java utilizada neste trabalho será o grafo de execução. As justificativas para essa escolha são:

- O conceito de grafo é mais difundido que os hipergrafos;
- Modelo mais fácil de ser entendido.

5.2 Linguagem de Consulta sobre o Grafo de Execução

Esta seção apresentará a linguagem de consulta sobre o grafo de execução. O principal objetivo dessa linguagem é permitir que o usuário especifique consultas que filtrem

as informações sobre execução do programa para ajudá-lo a identificar quais são os principais componentes da arquitetura de *software* e quais os relacionamentos desses componentes com os outros. As informações sobre a execução do programa estão presentes no grafo de execução. A linguagem permite basicamente dois tipos de consultas:

- Consulta de Pontos de Execução Consecutivos: durante a execução do ponto *ExecutionPattern*$_r$ no contexto *ContextPattern*$_r$ o ponto de execução *ExecutionPattern*$_c$ no contexto *ContextPattern*$_c$ é imediatamente executado.
  `(`*ContextPattern*$_r$`,` *ExecutionPattern*$_r$`)`
  `->` `(`*ContextPattern*$_c$`,` *ExecutionPattern*$_c$`)`

- Consulta de Pontos de Execução Adjacentes: se existe um caminho na árvore de execução do programa, onde o ponto *ExecutionPattern*$_r$ no contexto *ContextPattern*$_r$ é a raiz e o ponto *ExecutionPattern*$_c$ no contexto *ContextPattern*$_c$ é um filho.
  `(`*ContextPattern*$_r$`,` *ExecutionPattern*$_r$`)`
  `-+>` `(`*ContextPattern*$_c$`,` *ExecutionPattern*$_c$`)`

  As produções iniciais da linguagem de consulta são:

$$
\begin{array}{lcl}
S & \rightarrow & V_r \texttt{ -> } V_c \\
 & | & V_r \texttt{ -+> } V_c
\end{array}
$$

$$
V \rightarrow \texttt{ (} \textit{ContextPattern} \texttt{ , } \textit{ExecutionPattern} \texttt{)}
$$

Um *ContextPattern* é um padrão de contexto: geralmente é uma expressão que define um padrão de um tipo. Essas expressões podem fazer uso do caractere curinga `"*"`. Alguns exemplos de padrões de contexto simples são:

- `com.my.Clazz`: padrão de contexto que casa com o tipo `Clazz` definido no pacote `com.my`;

- `com.my.*Interface`: padrão de contexto que casa com qualquer tipo cujo nome termine com `Interface` e foi definido no pacote `com.my`;

- `*`: padrão de contexto que casa com qualquer tipo;

- `com..*`: padrão de contexto que casa com qualquer tipo que esteja definido abaixo do pacote `com`.

Além do caractere curinga "*"pode-se utilizar o caractere "+"para selecionar tipos que são subtipos do padrão especificado. Alguns exemplos de padrões de contexto que utilizam esse outro operador são:

- `java.io.InputStream+`: padrão de contexto que casa com qualquer tipo que estende a classe `java.io.InputStream` e a própria classe inclusive;
- `java.sql.Statement+`: padrão de contexto que casa com qualquer tipo que implementa a interface `java.sql.Statement`.

As produções da linguagem de consulta para especificação de padrões de contexto encontram-se na Figura 5.7.

A linguagem de consulta também permite a especificação de padrões de contexto que utilizem os operadores lógicos `&&`, `||` ou `!`, além de poder agrupá-los entre parênteses. Alguns exemplos que utilizam esses operadores são:

- `java.io.*Stream && !java.io.Output*`: padrão de contexto que casa com qualquer tipo que esteja no pacote `java.io` e termine com a palavra `Stream` e não comece com a palavra `Output`;
- `java.io.InputStream+ && !(java.io.InputStream)`: padrão de contexto que casa com qualquer tipo que estende a classe `java.io.InputStream` e que não seja a própria classe;
- `* && !*`: padrão de contexto que não casa com nenhum tipo;
- `*..*Filter || *..*Filtro`: padrão de contexto que casa com qualquer tipo que termine com `Filter` ou `Filtro` e que esteja em qualquer pacote.

Um *ExecutionPattern* é um padrão de ponto de execução. Basicamente existem 4 tipos de padrão de ponto de execução:

- método: nesse padrão é opcional especificar os modificadores de visibilidade do método, é obrigatório especificar o tipo de retorno do método, o nome do método e os parâmetros formais do método.
- construtor: nesse padrão é opcional especificar os modificadores de visibilidade do construtor, é obrigatório utilizar a palavra reservada `new` e os parâmetros formais do construtor.

ContextPattern → *TypePatternExpr*

TypePatternExpr → *OrTypePatternExpr*
| *TypePatternExpr* `&&` *OrTypePatternExpr*

OrTypePatternExpr → *UnaryTypePatternExpr*
| *OrTypePatternExpr* `||` *UnaryTypePatternExpr*

UnaryTypePatternExpr → *BasicTypePatternExpr*
| `!` *UnaryTypePatternExpr*
| `(` *TypePatternExpr* `)`

BasicTypePatternExpr → `void`
| *BaseTypePatternExpr DimsOpt PlusOpt*

DimsOpt → *DimsOpt* `[]` | λ

PlusOpt → `+` | λ

BaseTypePatternExpr → *PrimitiveTypePatternExpr*
| *NamePatternExpr*

PrimitiveTypePatternExpr → `boolean` | `...` | `double`

NamePatternExpr → *SimpleNamePatternExpr*
| *NamePatternExpr PackageSep SimpleNamePatternExpr*

SimpleNamePatternExpr → `*`
| `identifier`
| `identifierpattern`

PackageSep → `.` | `..`

Figura 5.7. Gramática da linguagem para especificar padrões de contexto

- inicializador estático da classe: é obrigatório utilizar a palavra reservada `staticinitialization`.
- `?`: padrão de ponto de execução que casa com qualquer um dos 3 pontos de execução acima.

As produções da linguagem de consulta para especificação de padrões de ponto de execução encontram-se na Figura 5.8.

Os elementos da linguagem de consulta tiveram como base a linguagem AspectJ, que permite a especificação de pontos de junção. A principal diferença existente entre os elementos da linguagem de consulta e AspectJ é que a linguagem de consulta separa claramente a especificação de padrões de tipos da especificação de ponto de execução. Outra razão pela qual foi adotado o uso de elementos semelhantes à linguagem AspectJ é diminuir a curva de aprendizado da linguagem de consulta para interessados que já tenham costume com os elementos AspectJ.

5.3 Metodologia para Recuperação da Arquitetura de *Software*

Esta seção apresentará a metodologia para recuperação da arquitetura de *software* desenvolvida neste trabalho, utilizando diversos estudos de casos.

A primeira atividade dessa metodologia é a obtenção dos artefatos que ajudarão no entendimento dos principais componentes e conectores da arquitetura do sistema:

- Obter o código fonte do sistema: o código fonte será utilizado para executar o sistema em conjunto com o Aspecto de Execução.
- Obter documentação sobre a utilização do sistema, caso exista: essa documentação auxilia o engenheiro de *software* a conhecer os nomes de entidades do sistema e quais ações são realizadas sobre essas entidades.
- Obter especificações técnicas sobre o sistema, caso existam: fazem parte das especificações técnicas os diagramas UML a seguir:
  - Diagramas de classe
  - Diagramas de sequência

De posse desses artefatos, deve-se tentar identificar os principais componentes do sistema, analisando os diagramas existentes e a documentação obtida.

A próxima atividade é executar o sistema em conjunto com o Aspecto de Execução para que os eventos do programa sejam capturados e a ferramenta *ArchRecover* crie o grafo de execução. Aconselha-se a execução das funcionalidades do sistema por caso de uso.

Em seguida, deve-se explorar o grafo de execução:

1. Expandir a árvore de execução da *thread* principal e das demais *threads*, analisando o número de *threads*. A Figura 7.1 mostrará o grafo de execução do

programa `RegSys` e as 5 *threads* que foram criadas durante a sua execução. Uma análise completa da recuperação da arquitetura de *software* desse programa pode ser visto na Seção 7.1.

2. Elaborar consultas que filtrem as informações de interesse dos principais componentes da arquitetura do sistema:

   - Quando objetos desses componentes são criados:
     `(java.lang.Thread, public void run()) -+> (*.*MainComponent1 || *.*MainComponent2, new(..))`
     A Figura 7.2 apresentará o resultado da consulta que identifica a criação de todos os componentes Filtro, cujo nome termine com a palavra `Filter`, que ocorreram durante a execução do método `void run()` da *Thread* principal. Um observação importante é que existem dois objetos PassFilter diferentes: `v1.PassFilter@27` e `v1.PassFilter@31`.
   - Quais classes utilizam esses componentes:
     `(*, ?) -> (*.*MainComponent1, ?)`
     `(*, ?) -+> (*.*MainComponent2, ?)`
   - Quais classes esses componentes utilizam:
     `(*.*MainComponent1, ?) -> (*, ?)`
     `(*.*MainComponent2, ?) -+> (*, ?)`

3. Identificar as entradas e saídas de dados do programa (arquivos, banco de dados, etc):
   `(*, ?) -> (java.io.File, new(..))`

4. Identificar os principais conectores da arquitetura:
   `(*,?) -> (java.io.PipedWriter, void connect(java.io.PipedReader))`
   A Figura 7.7 apresentará o resultado dessa consulta.

5. Construir manualmente a descrição da arquitetura do sistema adicionando componentes e conectores descobertos. A Figura 7.12 mostrará a descrição da arquitetura recuperada do sistema `RegSys`.

5.4 Linguagem de Restrições Arquiteturais

Esta seção apresentará a linguagem para especificação de restrições arquiteturais. Os elementos dessa linguagem são um subconjunto das produções da linguagem de consultas. O principal objetivo dessa linguagem é permitir a especificação de restrições

arquiteturais que serão utilizadas no processo de verificação de conformidades. Assim como na linguagem de consultas, a linguagem de restrições arquiteturais permite dois tipos de restrições:

- Restrição de Pontos de Execução Consecutivos: durante a execução do ponto *ExecutionPattern*$_r$ no contexto *ContextPattern*$_r$ o ponto de execução *ExecutionPattern*$_c$ no contexto *ContextPattern*$_c$ é imediatamente executado.
  `(`*ContextPattern*$_r$`,` *ExecutionPattern*$_r$`)`
  `->` `(`*ContextPattern*$_c$`,` *ExecutionPattern*$_c$`)`

- Restrição de Pontos de Execução Adjacentes: se existe um caminho na árvore de execução do programa, onde o ponto *ExecutionPattern*$_r$ no contexto *ContextPattern*$_r$ é a raiz e o ponto *ExecutionPattern*$_c$ no contexto *ContextPattern*$_c$ é um filho.
  `(`*ContextPattern*$_r$`,` *ExecutionPattern*$_r$`)`
  `-+>` `(`*ContextPattern*$_c$`,` *ExecutionPattern*$_c$`)`

As produções iniciais da linguagem de restrições arquiteturais e as produções para especificação de padrões de contexto encontram-se na Figura 5.9.

As produções da linguagem de restrições arquiteturais para especificação de padrões de ponto de execução estão na Figura 5.10.

5.5 Gerador de Anotações

O gerador de anotações é o componente da ferramenta *ArchVerifier* responsável pela geração das anotações do código-fonte do programa nos pontos-chave que possuam restrições arquiteturais. Pontos-chave do programa são quaisquer pontos de execução que participe de uma determinada restrição arquitetural. Para se especificar as restrições, deve-se utilizar a linguagem de restrições arquiteturais. De acordo com essa linguagem, qualquer restrição arquitetural possui dois pontos-chave: origem e destino. Como um exemplo, seja a seguinte restrição arquitetural referente ao trecho de código da Figura 5.11:

```
(fw.web.Action+, execute(..))  -+> (fw.bs.Business+, new())
```

Essa restrição arquitetural possui como ponto-chave de origem o método `execute()`, com quaisquer parâmetros, de objetos cuja classe herde a superclasse `Action`. O ponto-chave de origem está selecionado no grafo de execução da Figura 5.11.

Já o ponto-chave de destino da restrição arquitetural é o construtor sem parâmetros de qualquer objeto cuja classe implemente a interface `Business`.

De forma sucinta, o funcionamento do gerador de anotações é:

1. Processar o arquivo com a descrição da arquitetura de *software* do programa extraindo todas as restrições arquiteturais;

2. Gerar o código do aspecto contendo a especificação e declaração das anotações dos pontos-chave do programa em AspectJ, da seguinte maneira:

   a) Gerar código para especificar as anotações do ponto-chave de origem;

   b) Gerar código para especificar as anotações do ponto-chave de destino;

   c) Gerar código para declarar as anotações que foram geradas nos pontos de execução de origem;

   d) Gerar código para declarar as anotações que foram geradas nos pontos de execução de destino.

Isto está mais bem detalhado na Seção 6.4.1, onde são mostrados os detalhes de implementação do gerador de anotações.

Com o intuito de exemplificar o código gerado pelas subetapas do processo acima, será utilizada a seguinte restrição arquitetural:

```
(fw.web.Action+, ?)  -+> (fw.bs.Business+, new(..))
```

Na subetapa 2a, como a especificação do ponto de execução do ponto-chave de origem utiliza o caractere curinga `?`, isso implicará na geração das seguintes anotações:

- Método:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface SourceMethodCall_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```

- Construtor:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface SourceConstructorCall_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```

- Inicializador estático da classe:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SourceStaticInitialization_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```

Isto ocorre pois o caractere curinga significa qualquer ponto de execução: chamada de método, chamada de construtor ou inicializador estático de classe.

Já a especificação do ponto de execução do ponto-chave de destino, subetapa 2b, gerará somente uma anotação, pois o ponto de execução é um contrutor:

- Construtor:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface TargetConstructorCall_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```

Na subetapa 2c, o gerador declarará as anotações de origem que foram geradas na subetapa 2a, produzindo o seguinte código:

- Método:

```
declare @method :
  * fw.web.Action+.*(..) :
  @SourceMethodCall_1();
```

- Construtor:

```
declare @constructor :
  fw.web.Action+.new(..) :
  @SourceConstructorCall_1();
```

- Inicializador estático da classe:

```
declare @type :
  fw.web.Action+ :
  @SourceStaticInicialization_1();
```

Por último, na subetapa 2d, produzirá o código abaixo para declarar a anotação de destino:

- Construtor:

```
declare @constructor :
  fw.bs.Business+.new(..) :
  @TargetConstructorCall_1();
```

O gerador de anotações produz o arquivo de saída com o nome especificado com a extensão `aj`, de AspectJ.

5.6 Verificação de Conformidade

A verificação de conformidade é realizada pela ferramenta *ArchVerifier*, que analisa se a descrição da arquitetura de um sistema que possua restrições arquiteturais está de acordo com a implementação do sistema em tempo de execução. Essa verificação ocorre quando o programa é executado em conjunto com dois aspectos:

- O aspecto que possui as anotações dos pontos-chave do programa e suas declarações, produzido pelo gerador de anotações;

- O aspecto de verificação de conformidade.

Assim como o aspecto de execução, o aspecto de verificação também constrói o grafo de execução do sistema e ao identificar os pontos-chave das restrições arquiteturais, o aspecto de verificação analisa se uma determinada restrição é satisfeita pelo grafo de execução do programa.

A ferramenta *ArchVerifier* apresenta durante a execução do programa quais restrições foram verificadas e quais não foram verificadas.

5.7 Conclusão

Este capítulo apresentou as ferramentas implementadas para auxiliar os engenheiros de *software* na manutenção de sistemas de *software*, bem como as linguagens para consulta e especificação de restrições arquiteturais e a metodologia para recuperação da arquitetura de *software* desenvolvida nesse trabalho. A principal vantagem da linguagem de consulta em relação à DiscoTect [34] é que o usuário pode para uma única execução do sistema consultar diversas informações para ajudá-lo a identificar quais são os principais componentes da arquitetura de *software* e quais os relacionamentos desses componentes com os outros, sempre refinando os resultados sem a necessidade de especificar outras máquinas de estado nem executar o programa várias vezes.

ExecutionPattern → *OrExecutionPattern*
| *ExecutionPattern* `&&` *OrTypePatternExpr*

OrExecutionPattern → *UnaryExecutionPattern*
| *OrExecutionPattern* `||` *UnaryExecutionPattern*

UnaryExecutionPattern → *BasicExecutionPattern*
| `!` *UnaryExecutionPattern*

BasicExecutionPattern → *BaseExecutionPattern*
| `(` *ExecutionPattern* `)`

BaseExecutionPattern → `?`
| `staticinitialization`
| *MethodPattern*
| *ConstructorPattern*

MethodPattern → *ModifiersOpt BasicTypePatternExpr*
MethodNamePattern `(` *FormalParametersOpt* `)`

ConstructorPattern → *ModifiersOpt* `new (` *FormalParametersOpt* `)`

MethodNamePattern → `*`
| `identifier`
| `identifierpattern`

ModifiersOpt → *ModifiersOpt Modifier* | λ

Modifier → `public` | `...` | `volatile`

FormalParametersOpt → *FormalParameters* | λ

FormalParameters → *FormalParameter*
| *FormalParameters* `,` *FormalParameter*

FormalParameter → `..`
| *BasicTypePatternExpr*

Figura 5.8. Gramática da linguagem para especificar padrões de ponto de execução

$$
\begin{array}{rcl}
S & \rightarrow & V_r \texttt{ -> } V_c \\
 & | & V_r \texttt{ -+> } V_c \\
 & & \\
V & \rightarrow & \texttt{(}\ \textit{ContextPattern}\ \texttt{,}\ \textit{ExecutionPattern}\ \texttt{)} \\
 & & \\
\textit{ContextPattern} & \rightarrow & \textit{BasicTypePatternExpr} \\
 & & \\
\textit{BasicTypePatternExpr} & \rightarrow & \texttt{void} \\
 & | & \textit{BaseTypePatternExpr}\ \textit{DimsOpt}\ \textit{PlusOpt} \\
 & & \\
\textit{DimsOpt} & \rightarrow & \textit{DimsOpt}\ \texttt{[]} \mid \lambda \\
 & & \\
\textit{PlusOpt} & \rightarrow & \texttt{+} \mid \lambda \\
 & & \\
\textit{BaseTypePatternExpr} & \rightarrow & \textit{PrimitiveTypePatternExpr} \\
 & | & \textit{NamePatternExpr} \\
 & & \\
\textit{PrimitiveTypePatternExpr} & \rightarrow & \texttt{boolean} \mid \ldots \mid \texttt{double} \\
 & & \\
\textit{NamePatternExpr} & \rightarrow & \textit{SimpleNamePatternExpr} \\
 & | & \textit{NamePatternExpr}\ \textit{PackageSep}\ \textit{SimpleNamePatternExpr} \\
 & & \\
\textit{SimpleNamePatternExpr} & \rightarrow & \texttt{*} \\
 & | & \texttt{identifier} \\
 & | & \texttt{identifierpattern} \\
 & & \\
\textit{PackageSep} & \rightarrow & \texttt{.} \mid \texttt{..}
\end{array}
$$

Figura 5.9. Gramática da linguagem de restrições para especificar padrões de contexto

ExecutionPattern → *BaseExecutionPattern*

BaseExecutionPattern → `?`
| `staticinitialization`
| *MethodPattern*
| *ConstructorPattern*

MethodPattern → *ModifiersOpt BasicTypePatternExpr*
MethodNamePattern `(` *FormalParametersOpt* `)`

ConstructorPattern → *ModifiersOpt* `new (` *FormalParametersOpt* `)`

MethodNamePattern → `*`
| `identifier`
| `identifierpattern`

ModifiersOpt → *ModifiersOpt Modifier* | λ

Modifier → `public` | `...` | `volatile`

FormalParametersOpt → *FormalParameters* | λ

FormalParameters → *FormalParameter*
| *FormalParameters* `,` *FormalParameter*

FormalParameter → `..`
| *BasicTypePatternExpr*

Figura 5.10. Gramática da linguagem de restrições para especificar padrões de ponto de execução

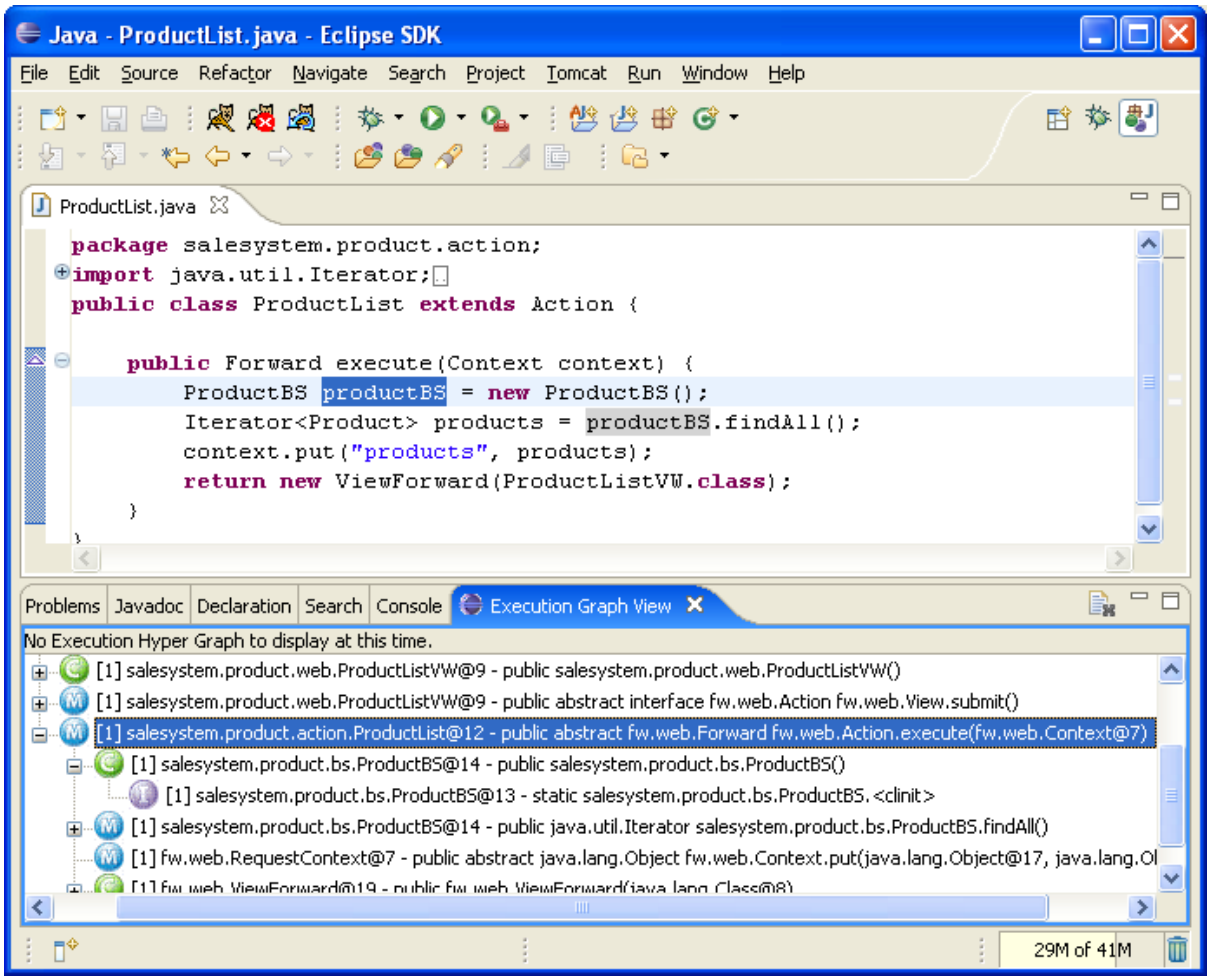

Figura 5.11. Código da Classe `ProductList` e Grafo de execução

Capítulo 6

Detalhes de Implementação

Este capítulo apresenta os detalhes de implementação das ferramentas *ArchRecover* e *ArchVerifier*. Devido ao fato das ferramentas terem vários pontos de interação com os usuários é interessante que elas possam ser utilizadas em um ambiente de desenvolvimento integrado, IDE[1]. Como o Eclipse [ecl] é uma IDE bastante utilizada no desenvolvimento de aplicações Java e permite extensões através de *plug-ins*, as ferramentas *ArchRecover* e *ArchVerifier* foram implementadas como *plug-ins* do Eclipse, para possibilitar a integração com um ambiente de desenvolvimento amplamente utilizado, aproveitando vários recursos já existentes. A Figura 6.1 mostra os *plug-ins* que foram desenvolvidos e suas dependências:

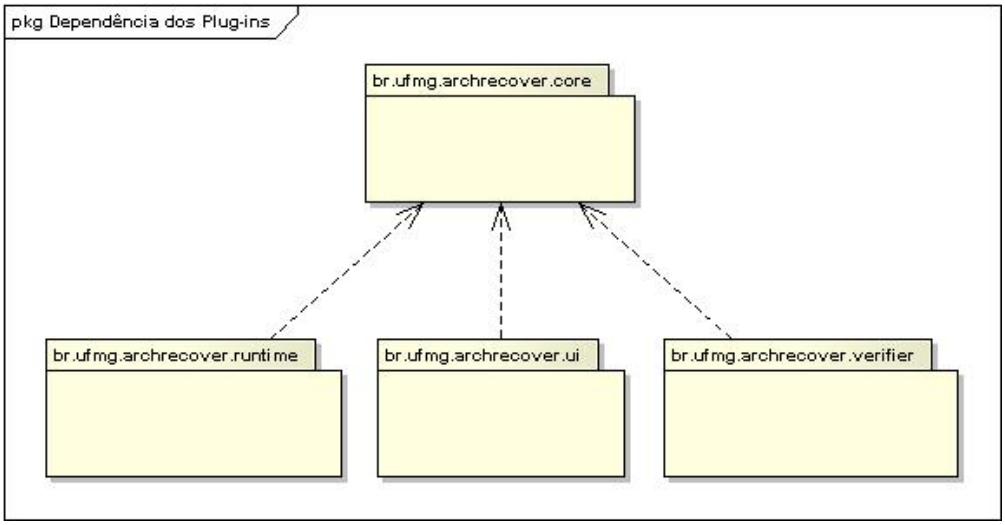

Figura 6.1. Dependência dos *Plug-ins*

[1]Do inglês, *Integrated Development Environment*

- `br.ufmg.archrecover.core`: *plug-in* que possui a definição do grafo de execução e as classes que são comuns aos demais *plug-ins*;
- `br.ufmg.archrecover.ui`: *plug-in* com as extensões de interface gráfica com o usuário das ferramentas *ArchRecover* e *ArchVerifier*;
- `br.ufmg.archrecover.runtime`: *plug-in* que possui o aspecto de execução, responsável pela construção do grafo de execução;
- `br.ufmg.archrecover.verifier`: *plug-in* que possui o gerador de anotações e o aspecto de verificação de conformidades.

A seguir, serão apresentados os detalhes de implementação de cada *plug-in* do Eclipse.

6.1 *Plug-in* `br.ufmg.archrecover.core`

Esse *plug-in* possui a definição do grafo de execução, o gerador de anotações e a implementação de algumas funções que são utilizadas em outros *plug-ins*:

- casamento de contexto;
- casamento de ponto de execução;

6.1.1 Grafo de Execução

O grafo de execução proposto na Seção 5.1.3 foi modelado em Java de acordo com o diagrama de classes da Figura 6.2.

O elemento básico do grafo de execução é um vértice, representado pela classe `br.ufmg.archrecover.core.graph.Vertex`. Um vértice possui basicamente:

- `context`: o objeto no qual o ponto de execução foi executado;
- `executionPoint`: ponto de junção (`org.aspectj.lang.JoinPoint`) que foi executado;
- `vertexs`: vetor de vértices filhos que foram executados durante a execução do ponto de execução;
- `root`: vértice pai.

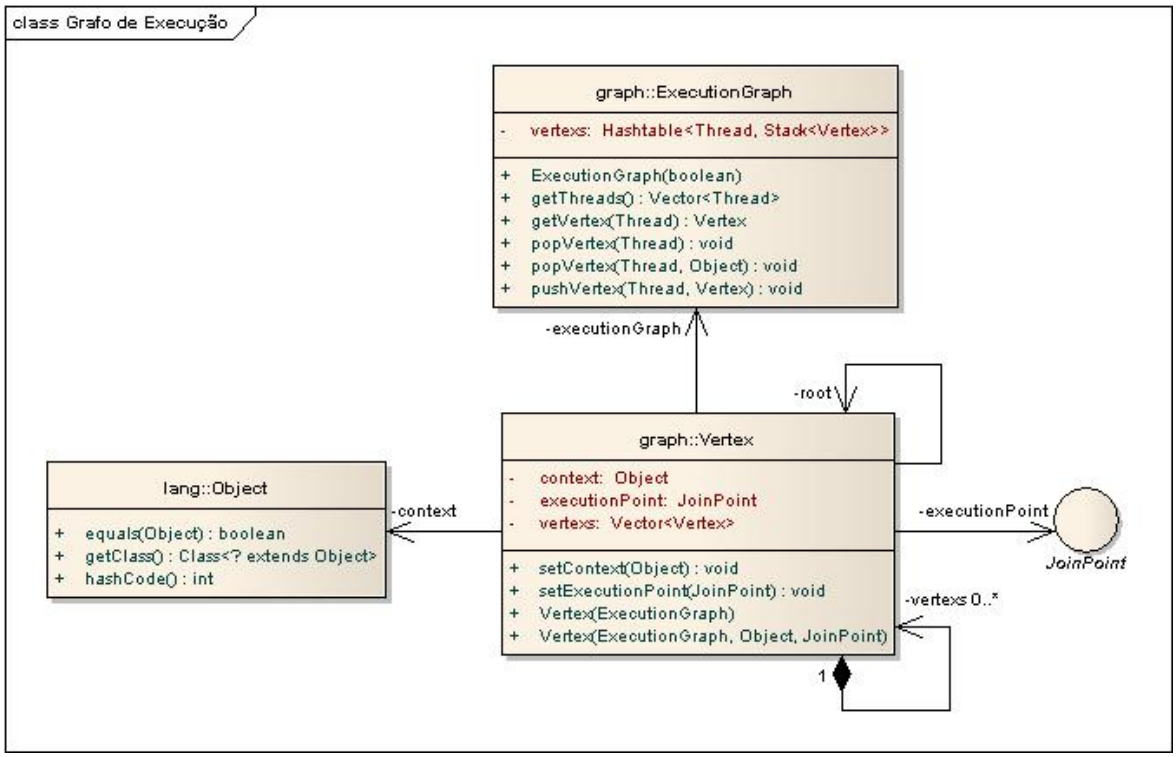

Figura 6.2. Diagrama de Classes do Grafo de Execução

O grafo de execução foi definido na classe `br.ufmg.archrecover.core.graph.ExecutionGraph`. Os objetos dessa classe possuem uma pilha de vértices para cada `Thread` do programa. O vértice raiz de cada `Thread` é o método `void run()` da Thread que estiver executando. Essa classe disponibiliza os seguintes métodos para o Aspecto de Execução para construção do grafo:

- `void pushVertex(Thread, Vertex)`: se a pilha de vértices da `Thread` está vazia, então o vértice raiz da `Thread` que está executando é empilhado. Caso contrário, recupera-se o vértice que está no topo da pilha, atribui-o como pai do vértice que está sendo empilhado e empilha o novo vértice. Esse método é utilizado antes da chamada de execução de algum evento monitorado pelo Aspecto de Execução.

- `void popVertex(Thread)`: desempilha o vértice que está no topo da pilha da `Thread`. Esse método é geralmente utilizado depois da chamada de execução de algum evento monitorado pelo Aspecto de Execução.

- `void popVertex(Thread, Object)`: desempilha o vértice que está no topo da pilha da `Thread` e atribui o contexto desse vértice como sendo o `Object`. Esse método é utilizado depois da chamada de execução de algum construtor monitorado pelo Aspecto de Execução.

6.1.2 Casamento de Contexto

As funções de casamento de contexto são utilizadas para verificar se o objeto do contexto de um vértice casa com a expressão de padrão de contexto que o usuário utilizou para fazer a consulta ao grafo de execução de um programa. A função de casamento de contexto foi implementada por meio da interface `br.ufmg.archrecover.core.matcher.ContextMatcher`. Essa interface define dois métodos:

- `String getDescription()`: retorna a expressão de padrão de contexto, equivalente à que o usuário informou na consulta;
- `boolean match(Object)`: retorna verdadeiro se o objeto casa com a expressão de contexto e falso, caso contrário.

A Figura 6.3 apresenta o diagrama de classes das classes que implementam as funções de casamento de contexto.

A classe `AnyParameterTypeContextMatcher` é utilizada pela expressão `..` da linguagem de consulta e seu método `boolean match(Object)` sempre retorna verdadeiro.

A classe `InParenthesesContextMatcher` é utilizada pela produção *UnaryTypePatternExpr* → `(` *TypePatternExpr*:*cp* `)` da linguagem de consulta. Essa regra da linguagem cria um novo objeto `InParenthesesContextMatcher` passando como parâmetro o objeto *cp*. Seu método `boolean match(Object)` invocará o método equivalente do objeto *cp*, exercendo assim um papel de *proxy*, segundo Gamma [13]. O objetivo desse padrão de projeto é fornecer um objeto representante para controlar o acesso ao mesmo.

A classe `NotContextMatcher` é utilizada pela produção *UnaryTypePatternExpr* → `!` *UnaryTypePatternExpr*:*cp* da linguagem de consulta. Essa regra da linguagem cria um novo objeto `NotContextMatcher` passando como parâmetro o objeto *cp*. Seu método `boolean match(Object)` negará o resultado da chamada de método equivalente do objeto *cp*, exercendo assim um papel de *decorator*, segundo Gamma [13], cujo objetivo desse padrão de projetos é: atribuir responsabilidades adicionais a um objeto dinamicamente, fornecendo uma alternativa flexível a subclasses para extensão de funcionalidade.

A classe `OrContextMatcher` é utilizada pela produção *OrTypePatternExpr* → *OrTypePatternExpr*:*cpl* `||` *UnaryTypePatternExpr*:*cpr* da linguagem de consulta. Essa regra da linguagem cria um novo objeto `OrContextMatcher` passando como parâmetro os objetos *cpl* e *cpr*. Seu método `boolean match(Object)` retornará o resultado

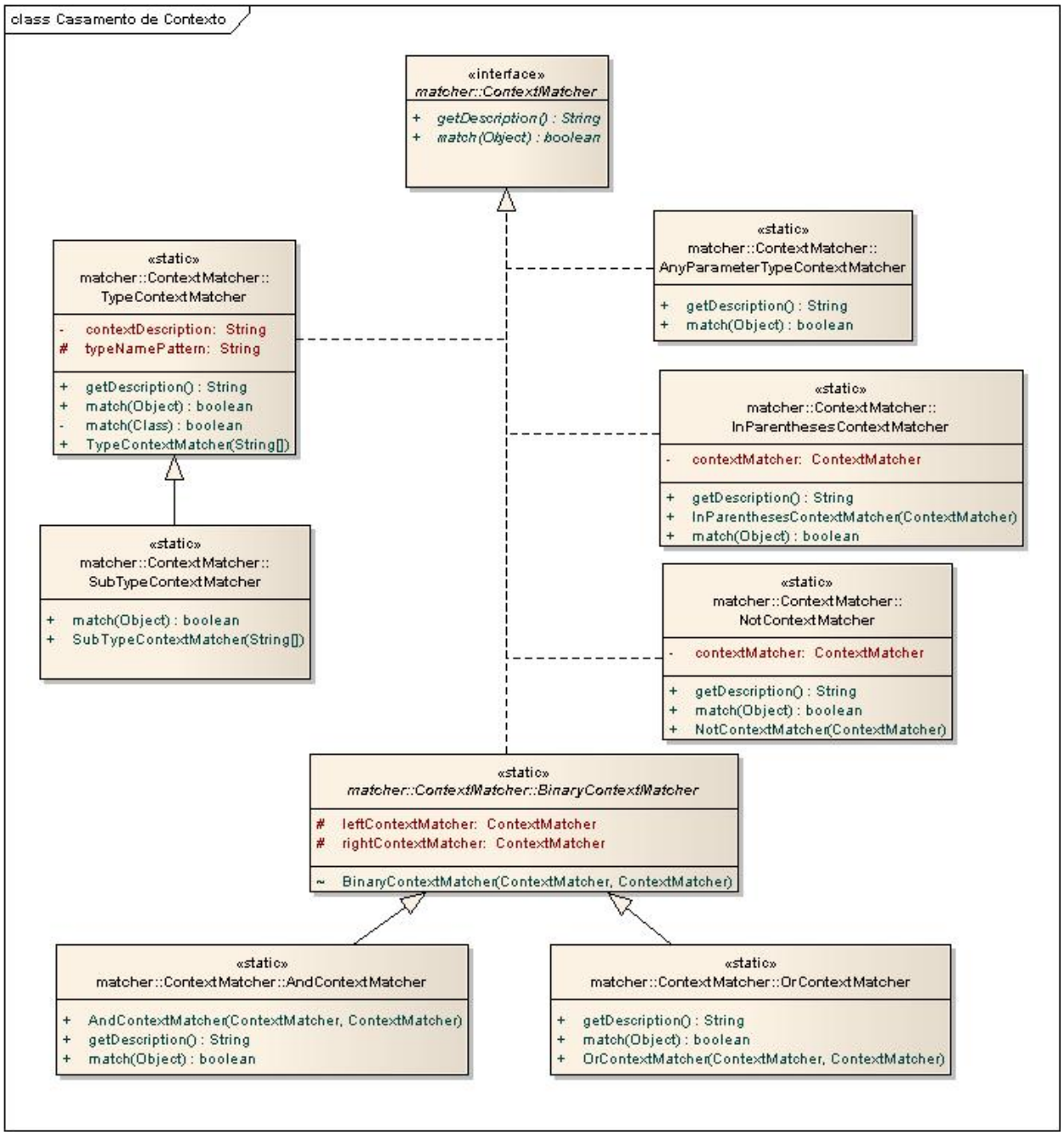

Figura 6.3. Diagrama de Classes do Casamento de Contexto

da operação lógica OU (`||`) entre o resultado do método equivalente do objeto *cpl* e o resultado do método equivalente do objeto *cpr*.

A classe `AndContextMatcher` é utilizada pela produção *TypePatternExpr* → *TypePatternExpr*:*cpl* `&&` *OrTypePatternExpr*:*cpr* da linguagem de consulta. Essa regra da linguagem cria um novo objeto `AndContextMatcher` passando como parâmetro os objetos *cpl* e *cpr*. Seu método `boolean match(Object)` retornará o resultado da ope-

ração lógica E (`&&`) entre o resultado do método equivalente do objeto *cpl* e o resultado do método equivalente do objeto *cpr*.

A classe `TypeContextMatcher` é utilizada pelas produções:

- *BasicTypePatternExpr* → `void`
- *BasicTypePatternExpr* → *BaseTypePatternExpr*:*p* *DimsOpt*:*d* *PlusOpt*:*s*

da linguagem de consulta. Essa regra da linguagem cria um novo objeto `TypeContextMatcher` passando como parâmetro um arranjo de `String`, onde a primeira `String` é a descrição do padrão de contexto e a segunda é uma expressão regular do padrão de contexto *p* concatenada com o padrão de arranjo *d*, quando *s* é falso. Seu método `boolean match(Object)` verificará se a expressão regular `typeNamePattern` casa com o nome da classe do objeto.

A classe `SubTypeContextMatcher` é utilizada pela produção *BasicTypePatternExpr* → *BaseTypePatternExpr*:*p* *DimsOpt*:*d* *PlusOpt*:*s* da linguagem de consulta, quando *s* é verdadeiro. Essa regra da linguagem cria um novo objeto `SubTypeContextMatcher` passando como parâmetro um arranjo de `String`, onde a primeira `String` é a descrição do padrão de contexto, concatenado com o caractere `+` e a segunda é uma expressão regular do padrão de contexto *p* concatenada com o padrão de arranjo *d*. Seu método `boolean match(Object)` verificará se a expressão regular `typeNamePattern` casa com o nome da classe do objeto, ou com o nome de qualquer uma de suas superclasses ou interfaces.

6.1.3 Casamento de Ponto de Execução

As funções de casamento de ponto de execução são utilizadas para verificar se a assinatura do ponto de execução de um vértice casa com a expressão de padrão de ponto de execução que o usuário utilizou para fazer a consulta ao grafo de execução de um programa. A função de casamento de ponto de execução foi implementada através da interface `br.ufmg.archrecover.core.matcher.ExecutionPointMatcher`. Essa interface define dois métodos:

- `String getDescription()`: retorna a expressão de padrão de ponto de execução, equivalente à que o usuário informou na consulta;
- `boolean match(Signature)`: retorna verdadeiro se a assinatura casa com a expressão de ponto de execução e falso, caso contrário.

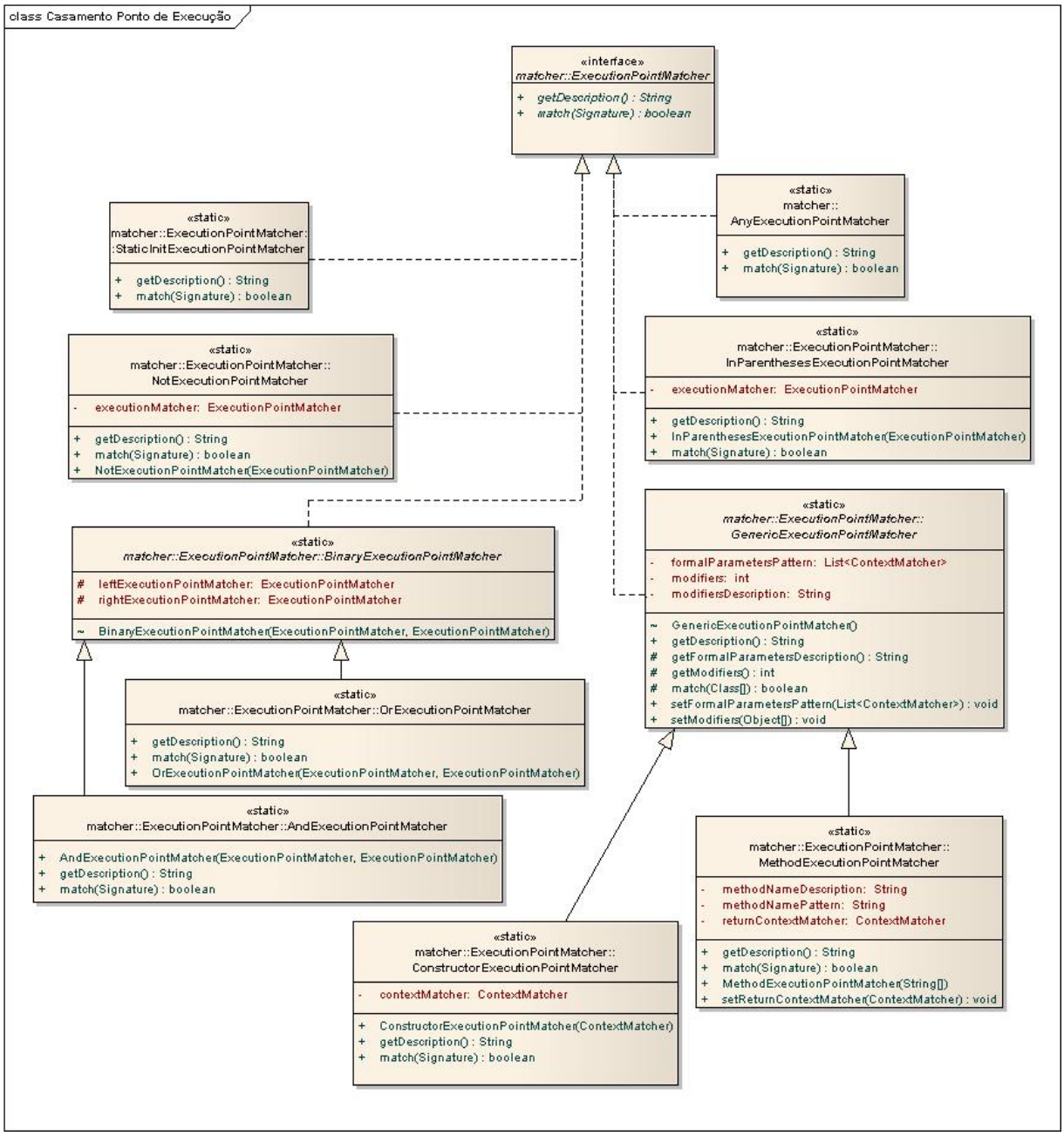

Figura 6.4. Diagrama de Classes do Casamento de Ponto de Execução

A Figura 6.4 apresenta o diagrama de classes das classes que implementam as funções de casamento de ponto de execução.

A classe `AnyExecutionPointMatcher` é utilizada pela produção *BaseExecutionPattern* → `?` da linguagem de consulta. Essa regra cria um novo objeto `AnyExecutionPointMatcher` e seu método `boolean match(Signature)` sempre retorna verdadeiro, casando dessa forma com qualquer ponto de execução, seja ele um método, um construtor ou um bloco de inicialização estática de uma classe.

A classe `InParenthesesExecutionPointMatcher` é utilizada pela produção *BasicExecutionPattern* → `(` *ExecutionPattern*:*ep* `)` da linguagem de consulta. Essa regra da linguagem cria um novo objeto `InParenthesesExecutionPointMatcher` passando como parâmetro o objeto *ep*. Seu método `boolean match(Signature)` invocará o método equivalente do objeto *ep*, exercendo assim um papel de *proxy*, segundo Gamma.

A classe `NotExecutionPointMatcher` é utilizada pela produção *UnaryExecutionPattern* → `!` *UnaryExecutionPattern*:*ep* da linguagem de consulta. Essa regra da linguagem cria um novo objeto `NotExecutionPointMatcher` passando como parâmetro o objeto *ep*. Seu método `boolean match(Signature)` negará o resultado do método equivalente do objeto *ep*, exercendo assim um papel de *decorator*, segundo Gamma.

A classe `OrExecutionPointMatcher` é utilizada pela produção *OrExecutionPattern* → *OrExecutionPattern*:*epl* `||` *UnaryExecutionPattern*:*epr* da linguagem de consulta. Essa regra da linguagem cria um novo objeto `OrExecutionPointMatcher` passando como parâmetro os objetos *epl* e *epr*. Seu método `boolean match(Signature)` retornará o resultado da operação lógica OU (`||`) entre o resultado do método equivalente do objeto *epl* e o resultado do método equivalente do objeto *epr*.

A classe `AndExecutionPointMatcher` é utilizada pela produção *ExecutionPattern* → *ExecutionPattern*:*epl* `&&` *OrTypePatternExpr*:*epr* da linguagem de consulta. Essa regra da linguagem cria um novo objeto `AndExecutionPointMatcher` passando como parâmetro os objetos *epl* e *epr*. Seu método `boolean match(Signature)` retornará o resultado da operação lógica E (`&&`) entre o resultado do método equivalente do objeto *epl* e o resultado do método equivalente do objeto *epr*.

A classe `StaticInitExecutionPointMatcher` é utilizada pela produção *BaseExecutionPattern* → `staticinitialization` da linguagem de consulta. Essa regra da linguagem cria um novo objeto `StaticInitExecutionPointMatcher`. Seu método `boolean match(Signature)` verificará se a assinatura é uma instância da interface `org.aspectj.lang.reflect.InitializerSignature`.

A classe `ConstructorExecutionPointMatcher` é utilizada pela produção *ConstructorPattern* → *ModifiersOpt* `new (` *FormalParametersOpt* `)` da linguagem de consulta. Essa regra da linguagem cria um novo objeto `ConstructorExecutionPointMatcher` passando como parâmetro um objeto `TypeContextMatcher` que casa com qualquer classe. Seu método `boolean match(Signature)` faz as seguintes verificações:

1. se a assinatura é uma instância da interface `org.aspectj.lang.reflect.ConstructorSignature`.

2. se a lista de modificadores de visibilidade casa com os modificadores de visibilidade do construtor;

3. se o nome da classe do construtor casa com o `TypeContextMatcher`;

4. se a lista de padrões de contexto de parâmetros formais casa com os tipos dos parâmetros formais do construtor.

A classe `MethodExecutionPointMatcher` é utilizada pela produção *MethodPattern* → *ModifiersOpt BasicTypePatternExpr MethodNamePattern*:*mn* (*FormalParametersOpt*) da linguagem de consulta. Essa regra da linguagem cria um novo objeto `MethodExecutionPointMatcher` passando como parâmetro um arranjo de `String`, onde a primeira `String` é a descrição do padrão do método e a segunda é uma expressão regular do nome do método *mn*. Seu método `boolean match(Signature)` faz as seguintes verificações:

1. se a assinatura é uma instância da interface `org.aspectj.lang.reflect.MethodSignature`.

2. se a lista de modificadores de visibilidade casa com os modificadores de visibilidade do método;

3. se o tipo de retorno do método casa com o padrão de retorno `ContextMatcher`;

4. se o padrão do nome do método casa com o nome do método;

5. se a lista de padrões de contexto de parâmetros formais casa com os tipos dos parâmetros formais do método.

6.2 *Plug-in* br.ufmg.archrecover.ui

Nesse *plug-in* foram implementadas as extensões para interface gráfica com o usuário das ferramentas *ArchRecover* e *ArchVerifier*:

- *View* para visualização do grafo de execução dos programas;
- Configuração de execução para *ArchRecover*;
- Configuração de execução para *ArchVerifier*.

A seguir serão apresentados os detalhes de implementação desses elementos.

6.2.1 *View* Grafo de Execução

A primeira extensão gráfica foi a criação de uma *view* para a visualização do grafo de execução no Eclipse. A especificação dela no arquivo plugin.xml é:

```
<extension point="org.eclipse.ui.views">
  <category
    name="ArchRecover"
    id="br.ufmg.archrecover">
  </category>
  <view
    category="br.ufmg.archrecover"
    class="br.ufmg.archrecover.ui.views.ExecutionGraphView"
    icon="icons/sample.gif"
    id="br.ufmg.archrecover.ui.views.ExecutionGraphView"
    name="Execution Graph View"/>
</extension>
```

O diagrama de classes da Figura 6.5 apresenta as classes que foram implementadas para a *view* `ExecutionGraphView`. Essa classe estende a classe `org.eclipse.ui.part.ViewPart` do Eclipse. O método `createPartControl()` cria um painel no qual as árvores de execução das *threads* são desenhadas. Cada nodo da árvore é um objeto `TreeObject`.

Os nodos da árvore possuem ícones diferentes para representar os três tipos de eventos:

- Ícone C: representa uma chamada de construtor;
- Ícone M: representa uma chamada de método;
- Ícone I: representa a inicialização estática de uma classe.

A Figura 6.6 abaixo mostra um exemplo da *view* para visualização do grafo de execução de um programa produtor consumidor, onde pode-se diferenciar os três tipos de eventos.

6.2.2 Configurações de execução para *ArchRecover* e *ArchVerifier*

Foram implementados dois pontos de extensão para criar duas novas configurações de execução de programas, uma para *ArchRecover* e outra para *ArchVerifier*. A Figura 6.7

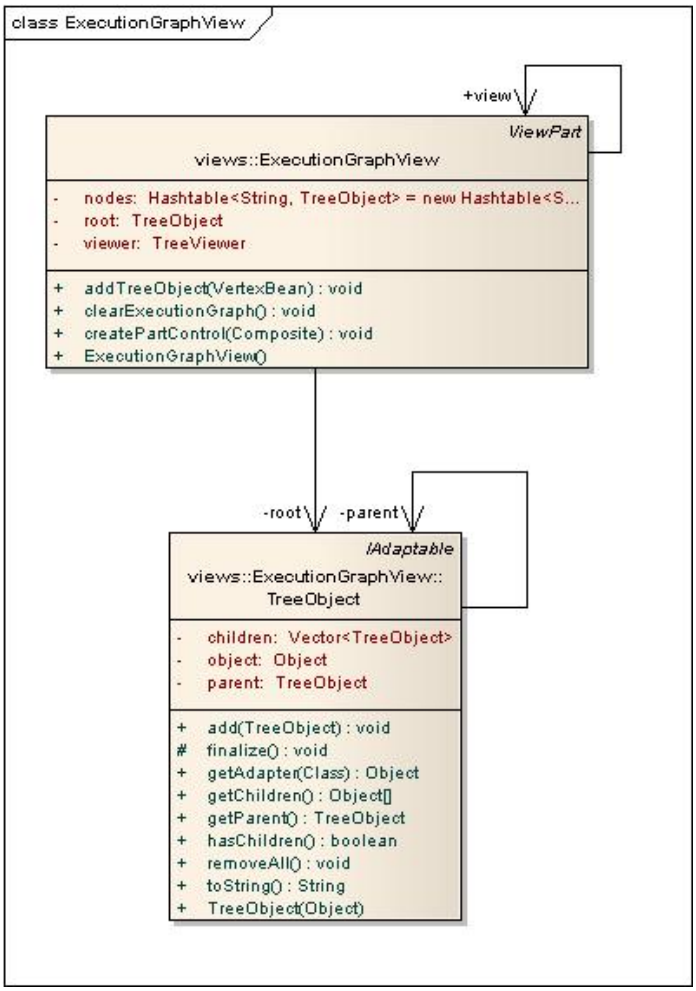

Figura 6.5. Diagrama de Classes da *view* do grafo de execução

mostra a tela onde essas extensões são utilizadas pelo Eclipse.

Ambas as configurações utilizam a função de costura de código em tempo de carga do *AJDT*[2] para executar o programa em conjunto com o aspecto de execução, no caso de *ArchRecover* e com o aspecto de verificação no caso de *ArchVerifier*. O conceito de *weaving*, ou costura, de bytecode Java é o processo de aplicar aspectos ao seu sistema. Esse processo pode ser efetuado em tempo de compilação ou em tempo de carga da classe na máquina virtual Java. Neste trabalho, utilizou-se a costura de código em tempo de carga[3], pois dessa maneira não existe a necessidade de ter o código-fonte do sistema. Basta obter o código do sistema compilado que a costura em tempo de carga se encarregará de interceptar os pontos de junção.

[2] *AspectJ Development Tools*
[3] Do inglês *Load Time Weaving* – LTW

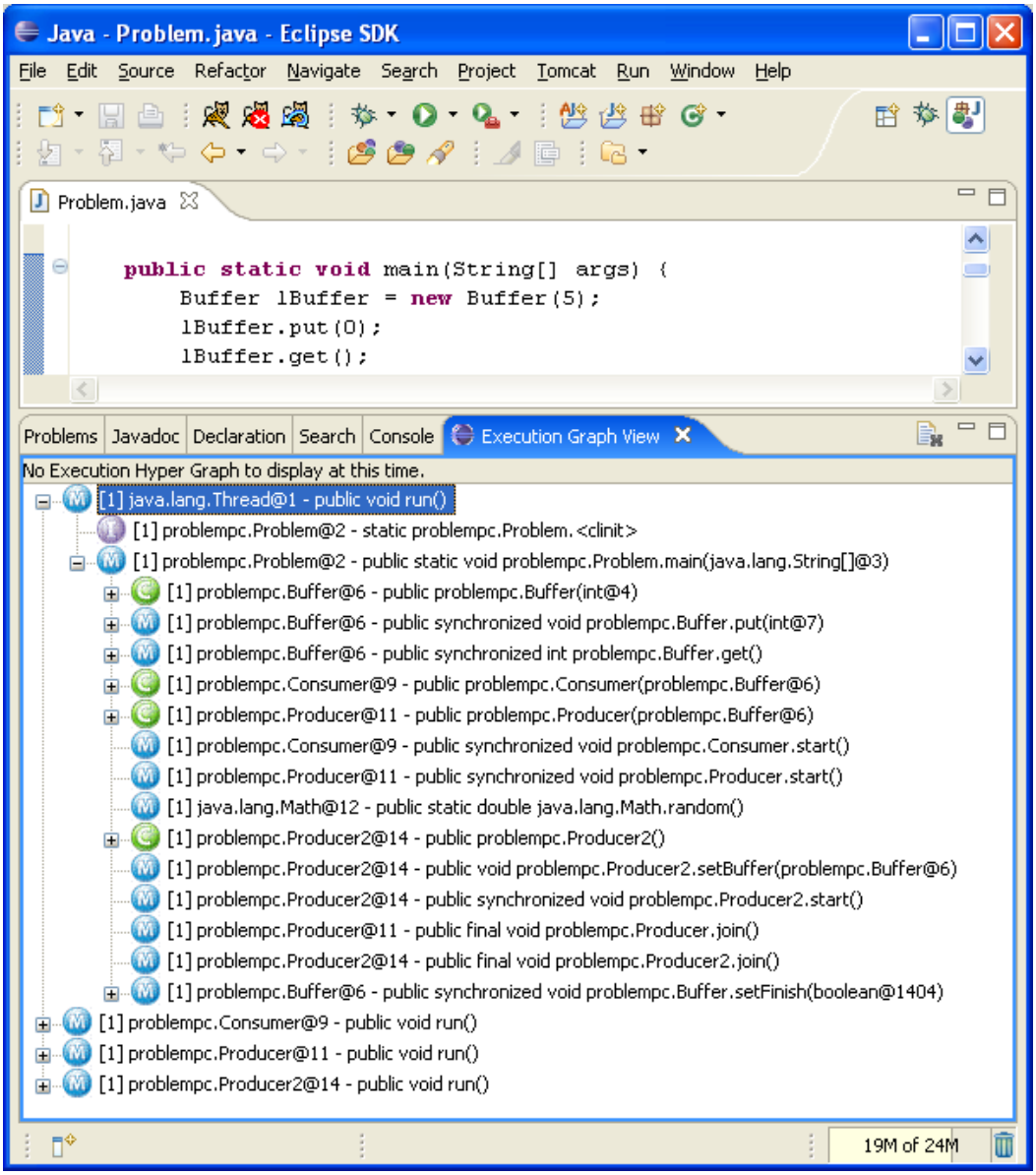

Figura 6.6. *View* do grafo de execução no Eclipse

6.3 *Plug-in* `br.ufmg.archrecover.runtime`

Esse *plug-in* possui as seguintes responsabilidades:

- geração do grafo de execução por meio do aspecto de execução que intercepta os eventos de interesse durante a execução de um programa e constrói o grafo de execução;
- linguagem para consultas ao grafo de execução;
- protótipo para executar a linguagem de consultas;

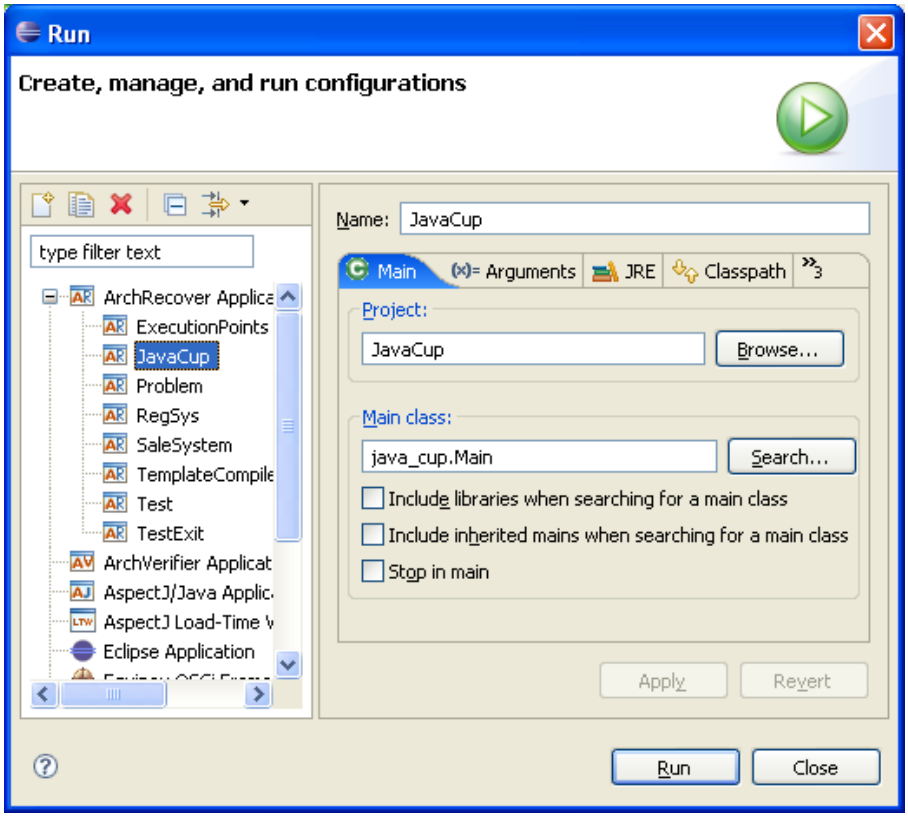

Figura 6.7. Configurações de execução para *ArchRecover* e *ArchVerifier* no Eclipse

6.3.1 Aspecto de Execução

O aspecto de execução é responsável pela interceptação dos eventos que ocorrem durante a execução de um programa Java para construir o grafo de execução. Sua implementação foi feita no aspecto `br.ufmg.archrecover.runtime.ExecutionAspect`.

6.3.2 Linguagem de Consulta sobre o Grafo de Execução

A especificação da linguagem de consulta definida na Seção 5.2 foi implementada em Java utilizando o gerador de analisadores sintáticos `CUP` [18]. `CUP` é um sistema para gerar *parsers* LALR de especificações simples. Ele posssui o mesmo papel que o YACC, programa amplamente utilizado e oferece a maioria das características do YACC. Porém, `CUP` é escrito em Java, usa especificações que incluem código Java embutido, e produz *parsers* que são implementados em Java.

O *parser* da linguagem de consulta foi então gerado na classe `br.ufmg.archrecover.runtime.graph.matcher.ExecutionGraphQueryParser`. O parser da linguagem de consulta processa as consultas e produz regras para casamento de con-

sulta da Seção 6.1.2 e para casamento de ponto de execução da Seção 6.1.3.

6.4 *Plug-in* `br.ufmg.archrecover.verifier`

Nesse *plug-in* foram implementados o gerador de anotações, elemento responsável pela geração das anotações dos pontos chaves que estejam presentes na descrição da arquitetura do sistema e o aspecto de verificação de conformidades, responsável por verificar se as restrições arquiteturais estão em conformidade com a implementação do sistema em tempo de execução. As próximas seções detalharão esses elementos da ferramenta *ArchVerifier*.

6.4.1 Gerador de Anotações

O gerador de anotações é o componente da ferramenta *ArchVerifier* responsável pela geração das anotações do código-fonte do programa nos pontos-chave que possuam restrições arquiteturais. Com o objetivo de permitir o uso de várias LDAs, o gerador de anotações foi projetado usando o padrão de projeto *AbstractFactory*, cujo objetivo é fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas [13]. O diagrama de classes da Figura 6.8 motrará o uso desse padrão. A interface `br.ufmg.archrecover.verifier.generator.factory.AnottationGenFactory` faz papel de *AbstractFactory*, cujo objetivo é definir as operações que deverão ser implementadas pelas fábricas concretas. No caso, a fábrica concreta é representada pela classe `AnottationGenFactoryImpl`. A interface `br.ufmg.archrecover.verifier.generator.AnottationGen` faz papel de *AbstractProduct* e define as operações que serão fornecidas pelos tipos de objetos produzidos pelas fábricas. A classe `AcmeAnnotationGen` faz papel de *ConcreteProduct*.

O programa do gerador de anotações recebe como entrada o caminho de um arquivo de descrição da arquitetura de um programa contendo as restrições arquiteturais e o nome do arquivo de saída. O programa então utiliza a fábrica `AnottationGenFactory` para criar o gerador de anotações da LDA do arquivo de entrada. A fábrica instancia um novo gerador de anotações correspondente à extensão do arquivo de entrada. Em seguida o programa utiliza o método `generate()` do gerador de anotações passando como padrâmetro o arquivo de saída. Por sua vez, esse método invoca o método `extractConstraints()`, o qual utiliza o parser responsável por processar o arquivo de entrada da LDA para extrair as restrições arquiteturais da descrição da arquitetura. Então, para cada restrição arquitetural, R_i, faz-se:

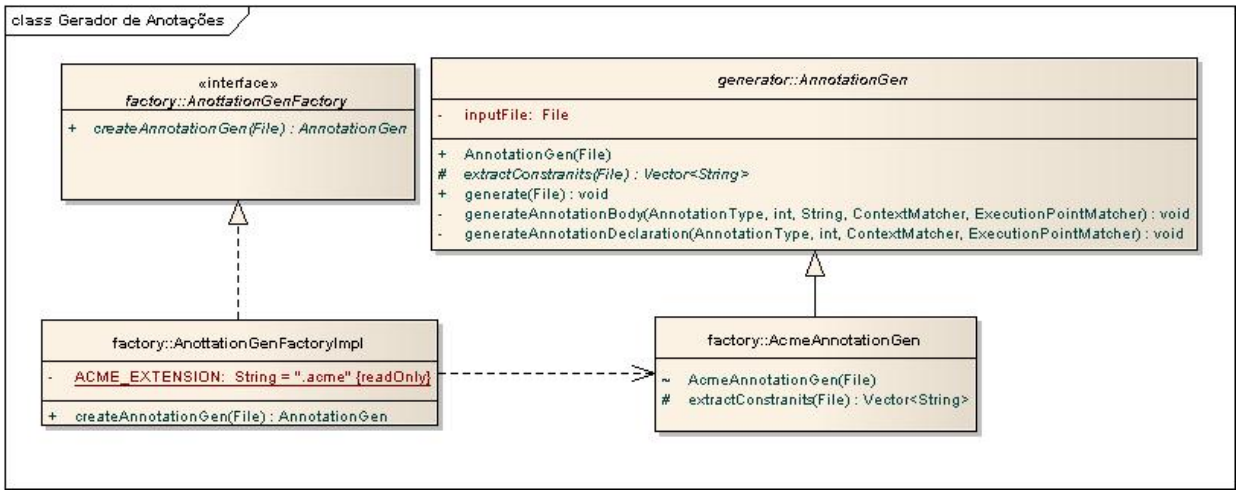

Figura 6.8. Diagrama de Classes do Gerador de Anotações

1. o gerador processa a restrição arquitetural utilizando o parser da linguagem de restrições arquiteturais;

2. o gerador produz as anotações do ponto-chave de origem;

3. o gerador produz as anotações do ponto-chave de destino;

4. o gerador declara as anotações que foram geradas nos pontos de execução de origem;

5. o gerador declara as anotações que foram geradas nos pontos de execução de destino.

A única responsabilidade que deve ser implementada pelos geradores de anotações é a extração das restrições arquiteturais, as quais dependeram da linguagem de descrição da arquitetura utilizada. A geração de código em AspectJ foi implementada na classe `AnottationGen`, na qual o método `generateAnnotationBody()` é responsável pelas Etapas 2 e 3 do processo acima e o método `generateAnnotationDeclaration()` é responsável pelas Etapas 4 e 5.

Para cada ponto chave de uma restrição arquitetural gera-se pelo menos uma anotação de origem e uma anotação de destino, sendo que o nome da anotação segue o padrão de nome da Tabela 6.1 para cada tipo de ponto de execução existente.

Foi preciso gerar código para cada anotação de um ponto chave utilizando os padrões de nome da Tabela 6.1 por que Java só permite que uma anotação de mesmo tipo por ponto chave, e como um mesmo ponto chave pode pertencer a mais de uma restrição arquitetural, isto não é possível.

Tipo de Ponto de Execução	Nomes das anotações de origem	Nomes das anotações de destino
Método	`SourceMethodCall_`i	`TargetMethodCall_`i
Construtor	`SourceConstructorCall_`i	`TargetConstructorCall_`i
Inicializador estático de classe	`SourceStaticInitialization_`i	`TargetStaticInitialization_`i
Curinga `"?"`	`SourceMethodCall_`i `SourceConstructorCall_`i `SourceStaticInitialization_`i	`TargetMethodCall_`i `TargetConstructorCall_`i `TargetStaticInitialization_`i

Tabela 6.1. Nomes das anotações geradas da i-ésima restrição arquitetural

6.4.2 Aspecto de Verificação de Conformidade

O principal papel desse *plug-in* é a verificação de conformidade entre a descrição da arquitetura e o código-fonte do sistema. O programa é executado em conjunto com um aspecto que intercepta os pontos de execução que foram anotados para fazer as verificações. Sua implementação está no aspecto `br.ufmg.archrecover.verifier.-ConformanceVerifier`.

6.4.3 Conclusão

Este capítulo apresentou os detalhes de implementação das ferramentas *ArchRecover* e *ArchVerifier* que foram implementadas como *plug-ins* do Eclipse para permitir interações com o usuário por meio de uma IDE de desenvolvimento amplamente utilizada.

Capítulo 7

Avaliação e Validação dos Resultados

Este capítulo apresenta alguns estudos de caso para avaliar as ferramentas *ArchRecover* e *ArchVerifier* bem como compará-las com outras ferramentas existentes. A Seção 7.1 apresenta dois estudos de caso sobre recuperação da arquitetura de *software*. No primeiro caso, recupera-se a arquitetura do sistema de exemplo apresentado em [34], baseado no estilo *Dutos e Filtros*. O segundo estudo de caso apresenta a recuperação da arquitetura do gerador de analisadores sintáticos `CUP` [18]. A Seção 7.2 apresenta um estudo de caso sobre verificação de conformidades.

7.1 Estudos de Caso: Recuperação da Arquitetura

O objetivo desses estudos de caso é recuperar a arquitetura de sistemas utilizando a metodologia de recuperação da arquitetura proposta por este trabalho na Seção 5.3 e as consultas sobre o grafo de execução após executar o programa em conjunto com o aspecto de execução. Utilizando uma abordagem experimental, cada estudo de caso terá:

- objetivo: o que espera-se demonstrar com o estudo de caso em questão.
- sistema alvo: uma breve descrição do sistema.
- motivo: motivo da escolha deste sistema alvo específico.
- passos do experimento: mostrar os passos para uma possível reprodução do experimento.

- resultados: quais resultados foram produzidos pelo experimento.
- riscos: quais os riscos para validade das conclusões.

7.1.1 Estilo *Dutos e Filtros*

O objetivo deste estudo de caso é a recuperação da arquitetura de software do sistema alvo `RegSys`[1] que foi apresentado como exemplo em [34]. Espera-se demonstrar que utilizando a metodologia de recuperação da arquitetura do sistema e as consultas sobre o grafo de execução seja possível recuperar a mesma descrição da arquitetura apresentada em DiscoTect. Este sistema foi escolhido para fins de comparação entre as ferramentas *ArchRecover* e DiscoTect.

O programa `RegSys` foi escrito seguindo o estilo de arquitetura *Dutos e Filtros*. Em resumo, este estilo define três tipos de componentes:

- um tipo fonte de dados, geralmente arquivos de entrada;
- um tipo para destino de dados, representados pelos arquivos de saída;
- filtros, cujas instâncias consomem a entrada de dados e produzem uma saída de dados.

O único tipo de conector definido por este estilo é o duto, utilizado para transportar dados entre as instâncias dos componentes deste estilo.

O programa `RegSys` cria uma configuração de filtros para verificar se os estudantes possuem os pré-requistos necessários para se matricularem em disciplinas de uma universidade da seguinte maneira:

1. cria-se um fluxo de dados dos alunos, lendo-os de um arquivo de entrada;
2. divide-se o fluxo de dados dos alunos em dois fluxos: um dos alunos de ciência da computação e um dos alunos de outros cursos;
3. verifica-se para cada tipo de aluno se os pré-requisitos foram satisfeitos, ou seja, se o alunos já cursou determinadas disciplinas;
4. e por fim agrupando os dois fluxos dos alunos que possuem os pré-requisitos e salvando-os em um arquivo de saída.

O programa é uma aplicação de pequeno porte, composta de 4 classes e 355 linhas de código. São elas:

[1] Disponível em: http://able.fluid.cs.cmu.edu:8080/Able/DiscoTect/PipeFilterExample.zip

- `SplitFilter`: este filtro lê os dados de um aluno por vez do arquivo de entrada e determina se o aluno é do curso de ciência da computação ou não. Caso afirmativo, os dados do aluno são enviados para um duto. Caso, contrário, os dados são enviados para outro duto.

- `PassFilter`: este filtro verifica se os alunos possuem os pré-requisitos necessários para matricularem nas disciplinas. Se sim, os dados do aluno é enviado para o próximo duto. Caso contrário os dados do aluno são descartados.

- `MergeFilter`: este filtro lê os dois fluxos de dados produzidos pelos PassFilter e grava-os em um arquivo de saída.

- `RegSys`: o método principal desta classe cria e inicia a execução dos filtros. Esta classe pode ser executada fornecendo o caminho dos arquivo de entrada e saída.

A seguir serão apresentados os passos do experimento, seguindo a metodologia de recuperação da arquitetura de software proposta neste trabalho.

O primeiro passo é tentar obter os artefatos sobre a documentação do sistema. Entretanto, a única documentação existente é a descrição do sistema presente no artigo[34], de DiscoTect.

O segundo passo é executar o programa em conjunto com o aspecto de execução para que os eventos do programa sejam capturados e a ferramenta *ArchRecover* crie a representação da execução do programa através do grafo de execução. Em seguida, deve-se explorar o grafo de execução, expandindo a árvore de execução da *Thread* principal e das demais *Thread* que por ventura tenham executado. A Figura 7.1 mostra o grafo de execução do programa `RegSys`. Expandindo os vértices do grafo de execução, percebe-se que durante a execução do programa foram criadas cinco *Threads*:

- principal: `[1] java.lang.Thread@1 - public void run()`

- `SplitFilter`: `[1] v1.SplitFilter@20 - public void run()`

- `PassFilter`: `[1] v1.PassFilter@27 - public void run()`

- `PassFilter`: `[1] v1.PassFilter@31 - public void run()`

- `MergeFilter`: `[1] v1.MergeFilter@33 - public void run()`

Cada uma das *Threads* forma uma árvore de execução. Um trecho da árvore de execução da *Thread* principal é mostrado na Figura 7.1.

O grafo mostra que os componentes do tipo filtro possuem nomes que terminam com a palavra `Filter`. Provavelmente estes componentes foram criados pela *Thread*

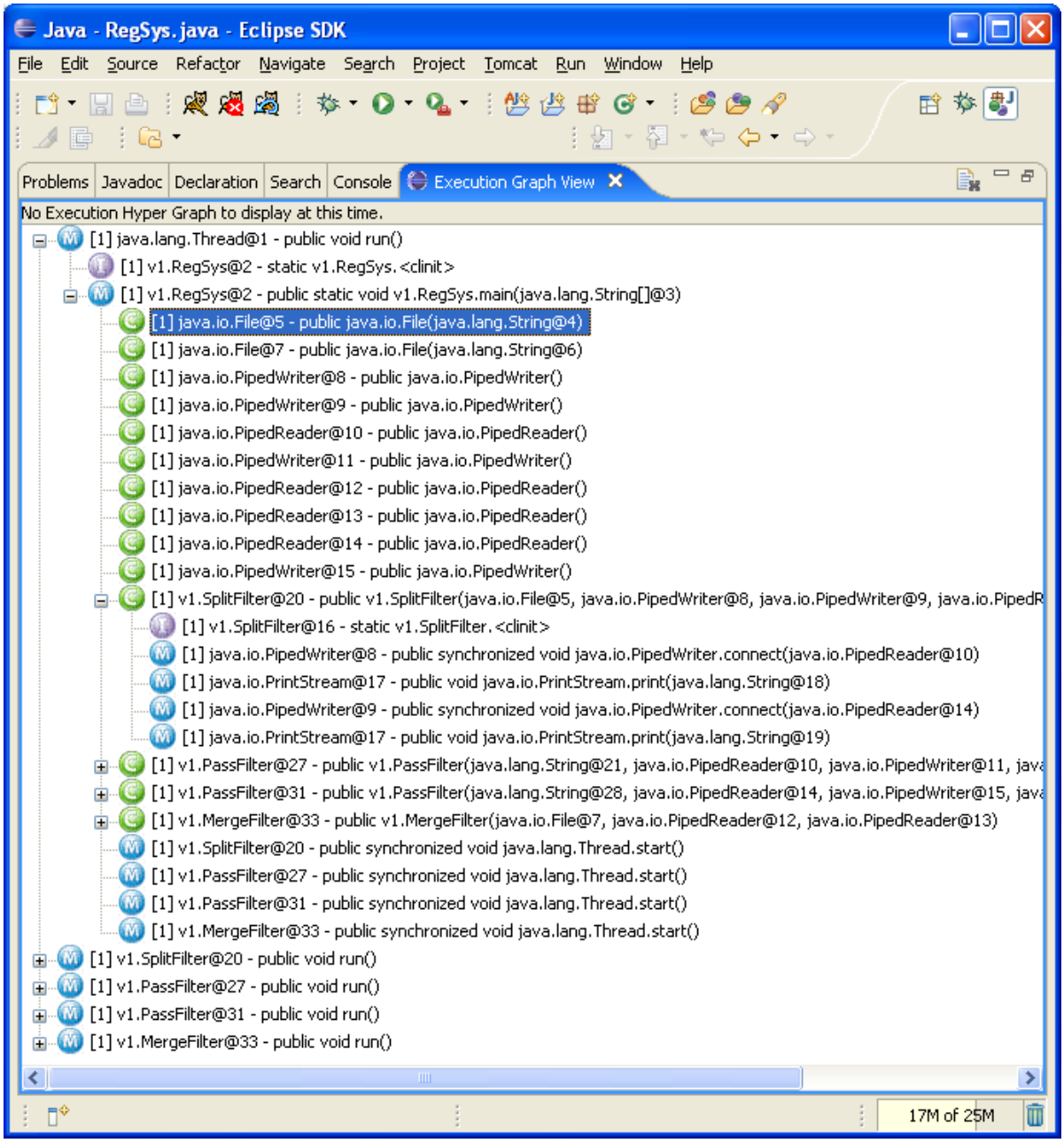

Figura 7.1. Grafo de Execução do programa `RegSys`

principal. A fim de comprovar esta suposição, pode-se fazer a seguinte consulta no grafo de execução:

```
(java.lang.Thread, public void run()) -+> (*.*Filter, new(..))
```

A consulta retorna a criação de todos os componentes Filtro, cujo nome termine com a palavra `Filter`, que ocorreram durante a execução do método `void run()` da *Thread* principal. O resultado pode ser visualizado na Figura 7.2.

Um observação importante é que existem dois objetos `PassFilter` diferentes:

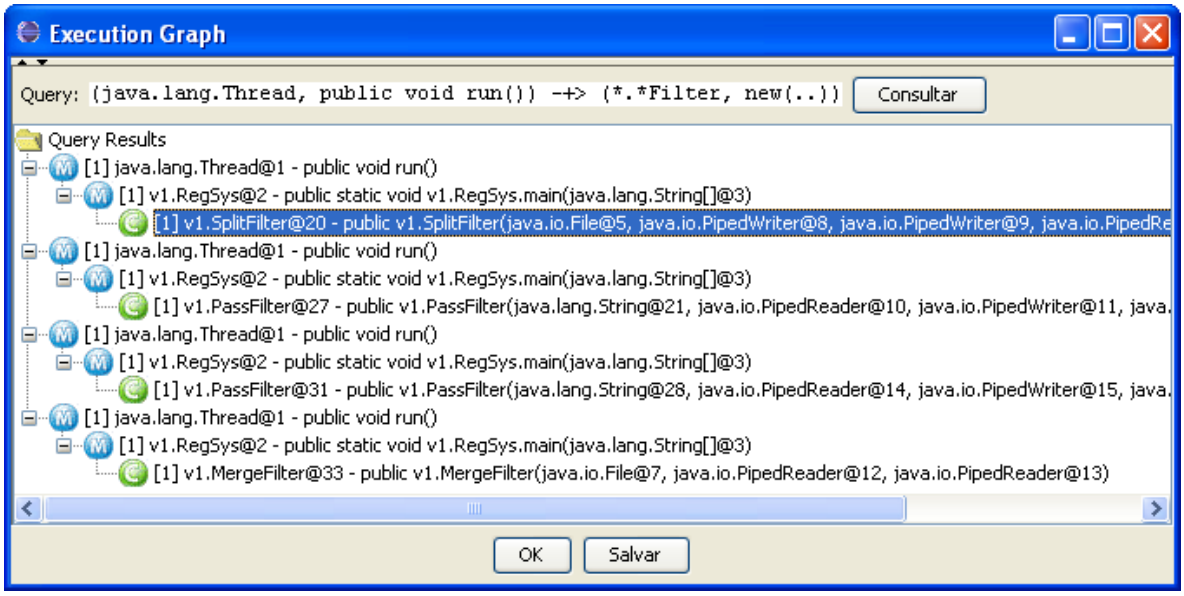

Figura 7.2. Resultado da consulta

`v1.PassFilter@27` e `v1.PassFilter@31`. Com as informações obtidas até o momento, pode-se começar a criar a descrição da arquitetura no AcmeStudio, criando os 4 componentes do tipo filtro, como mostra a Figura 7.3.

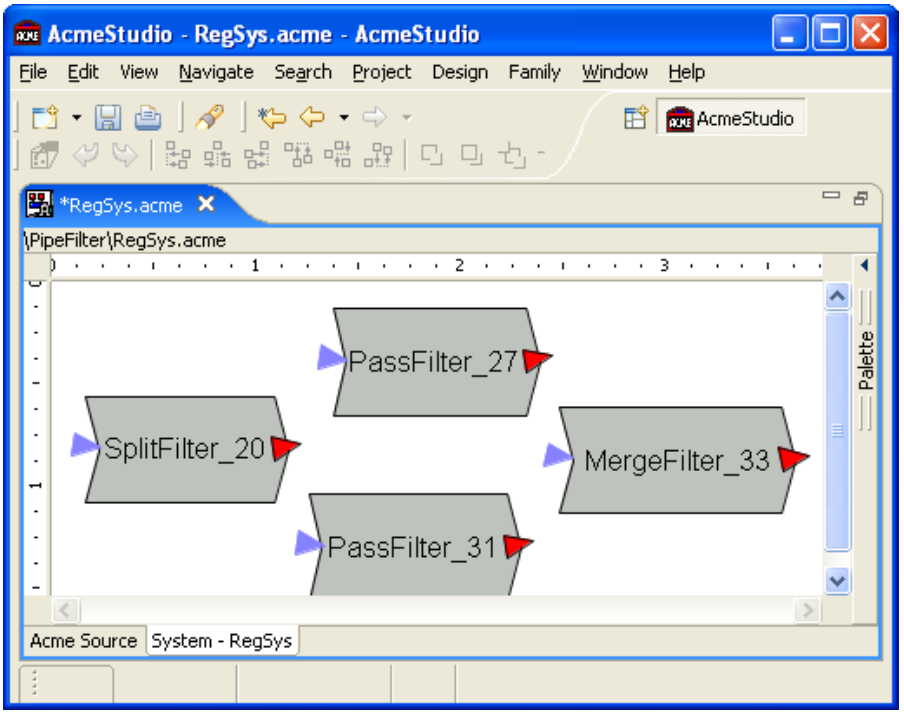

Figura 7.3. Início da Descrição da Arquitetura do programa `RegSys`

O próximo passo é identificar quais arquivos o programa manipula. A próxima

consulta abaixo identifica quais objetos do tipo `java.io.File` são criados durante a execução do programa:

```
(*, ?)  -> (java.io.File, new(..))
```

A consulta retorna 2 arquivos, que foram criados durante a execução do método `void main(String[])`:

- `java.io.File@5 - public java.io.File(java.lang.String@4)`
- `java.io.File@7 - public java.io.File(java.lang.String@6)`

Entretanto, não se sabe ainda se os arquivos exercem papel de entrada ou saída de dados. Uma consulta útil, que pode ser utilizada para identificar o papel de cada arquino no programa, é:

```
(*, ?)  -> (*, new(java.io.File))
```

A Figura 7.4 mostra o resultado da consulta, o qual fornece alguns indícios do fluxo de dados do programa:

1. o primeiro arquivo, `java.io.File@5`, é utilizado para entrada de dados, pois é utilizado para criar um objeto do tipo `java.io.FileReader`;
2. a criação do objeto `FileReader` ocorre durante a execução do método `void run()` do componente `v1.SplitFilter@20`. Isto é um forte indício de que este componente leia os dados do arquivo de entrada.
3. O segundo arquivo, `java.io.File@7`, é utilizado para saída de dados, pois é utilizado para criar um objeto do tipo `java.io.FileWriter`;
4. a criação do objeto `FileWriter` ocorre durante a execução do método `void run()` do componente `v1.MergeFilter@33`. Isto é um forte indício de que este componente grave os dados do arquivo de saída.

Para confirmar que o componente `v1.SplitFilter@20` lê os dados do arquivo de entrada, pode-se utilizar a seguinte consulta:

```
(v1.SplitFilter, public void run()) -+> (java.io.FileReader, * read*())
```

Porém, ao executar esta consulta nenhum resultado é obtido. Para tornar a consulta mais genérica, deve-se reescrevê-la da seguinte maneira:

```
(v1.SplitFilter, public void run()) -+> (java.io.Reader+, * read*())
```

Agora a consulta produz alguns resultados, que podem ser vistos na Figura 7.5. A penúltima consulta não produziu nenhum resultado pois o programa lê os dados do arquivo de entrada usando o objeto `java.io.BufferedReader@40` e não um objeto do tipo `java.io.FileReader`.

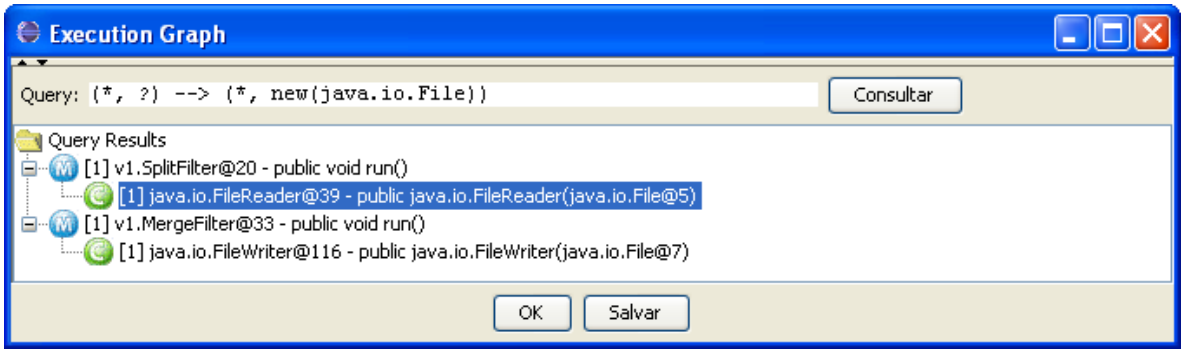

Figura 7.4. Resultado da consulta

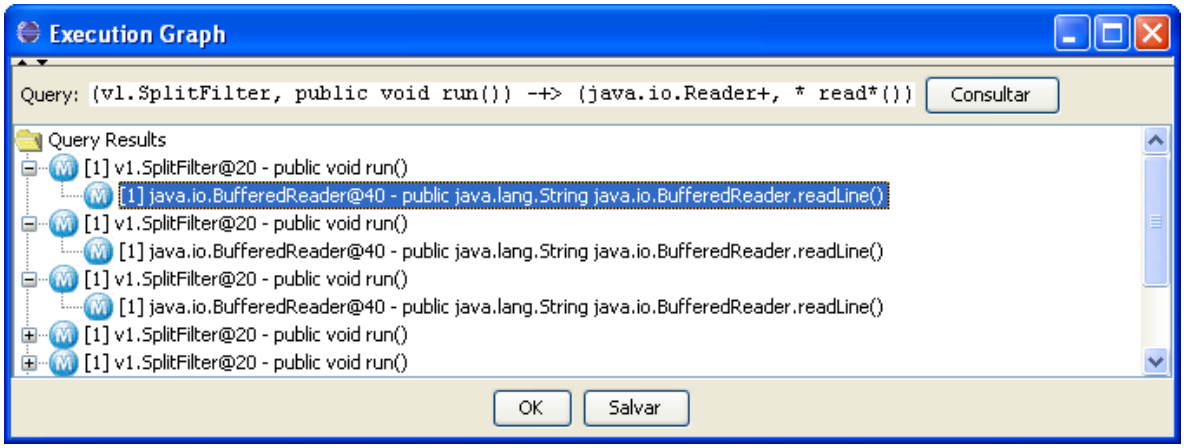

Figura 7.5. Resultado da consulta

Uma consulta semelhante pode ser utilizada para assegurar que o componente `v1.MergeFilter@33` grava os dados no arquivo de saída. A consulta é:

```
(v1.MergeFilter, public void run()) -+> (java.io.Writer+, void
write*(..))
```

A consulta produz alguns resultados, o que indica que o componente `v1.MergeFilter@33` realmente grava os dados no arquivo de saída.

Neste ponto, pode-se incrementar a descrição da arquitetura do programa `RegSys`, acrescentando a fonte dos dados e o destino, assim como os dutos que conectam o arquivo de entrada ao componente `SplitFilter` e o componente `MergeFilter` ao arquivo de saída. A Figura 7.6 ilustra esta situação.

Os próximos passos referem-se à descoberta dos demais conectores. Analisando árvore de execução da *Thread* principal na Figura 7.1 percebe-se que são criados alguns objetos `java.io.PipedWriter` e `java.io.PipedReader`, os quais provavelmente efetuam papel de dutos no programa. Para identificar a criação destes objetos, pode-se utilizar a seguinte consulta:

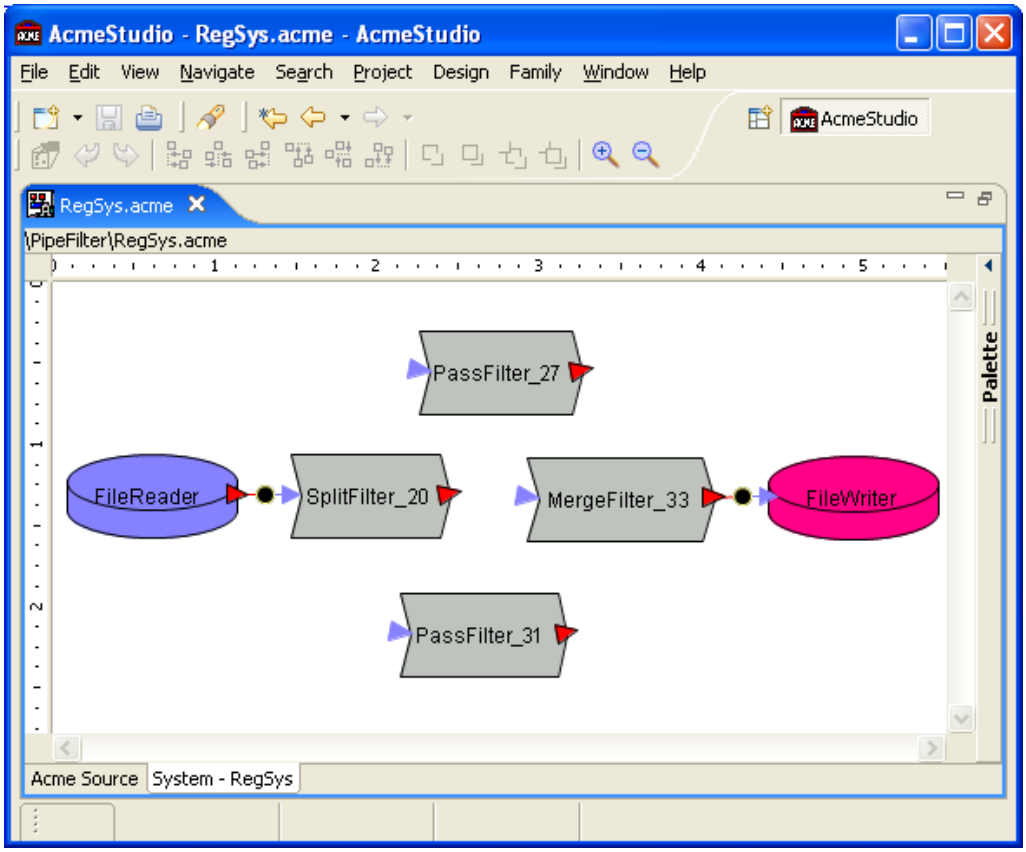

Figura 7.6. Descrição intermediária da Arquitetura do programa `RegSys`

```
(*, ?)  -> (java.io.PipedWriter || java.io.PipedReader, new())
```

A consulta retorna os seguintes objetos, que foram criados durante a execução do método `void main(String[])`:

- `java.io.PipedWriter@8 - public java.io.PipedWriter()`
- `java.io.PipedWriter@9 - public java.io.PipedWriter()`
- `java.io.PipedReader@10 - public java.io.PipedReader()`
- `java.io.PipedWriter@11 - public java.io.PipedWriter()`
- `java.io.PipedReader@12 - public java.io.PipedReader()`
- `java.io.PipedReader@13 - public java.io.PipedReader()`
- `java.io.PipedReader@14 - public java.io.PipedReader()`
- `java.io.PipedWriter@15 - public java.io.PipedWriter()`

A próxima consulta serve para identificar quais objetos do tipo `java.io.PipedReader` foram conectados a objetos do tipo `java.io.PipedWriter`:

```
(*,?)  -> (java.io.PipedWriter, void connect(java.io.PipedReader))
```

Os resultados desta consulta estão na Figura 7.7, os quais mostram que:

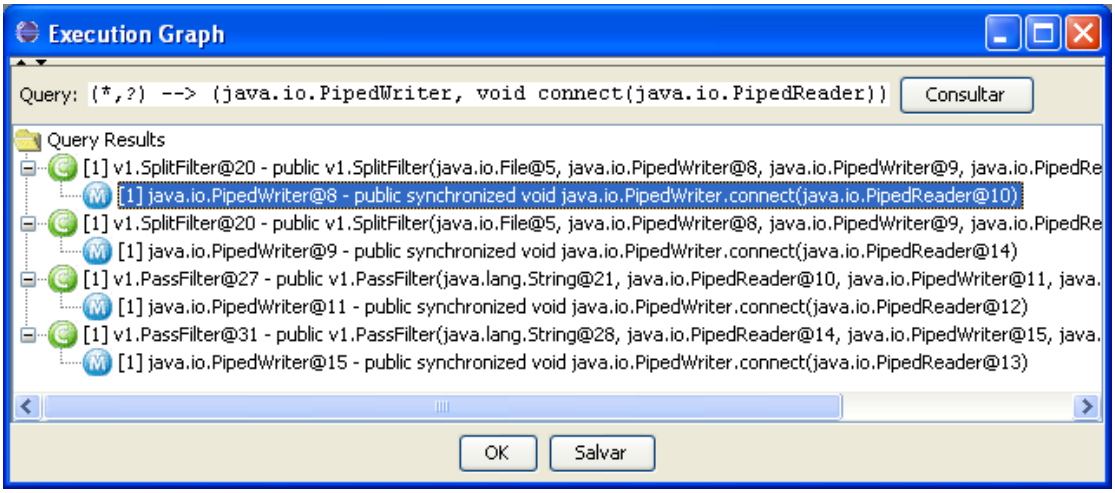

Figura 7.7. Resultado da consulta

- Durante a criação do componente `SplitFilter@20`, o objeto `java.io.PipedReader@10` é conectado ao objeto `java.io.PipedWriter@8` e o objeto `java.io.PipedReader@14` é conectado ao objeto `java.io.PipedWriter@9`. Isto são evidências que este componente possui duas portas de escrita.

- Durante a criação do componente `PassFilter@27`, o objeto `java.io.PipedReader@12` é conectado ao objeto `java.io.PipedWriter@11`. Isto mostra que este componete possui uma porta de escrita.

- Durante a criação do componente `PassFilter@31`, o objeto `java.io.PipedReader@13` é conectado ao objeto `java.io.PipedWriter@15`. Como era de se esperar, este compornete também possui uma porta de escrita.

A próxima consulta serve para mostrar que o compoente `SplitFilter@20` escreve nas duas portas de escrita:

```
(v1.SplitFilter, ?)  -+> (java.io.PipedWriter, * write(..))
```

O resultado da consulta pode ser visualizado na Figura7.8.

A questão agora é identificar de onde os componentes do tipo `v1.PassFilter` leêm seus dados. Para isto, pode-se utilizar a seguinte consulta:

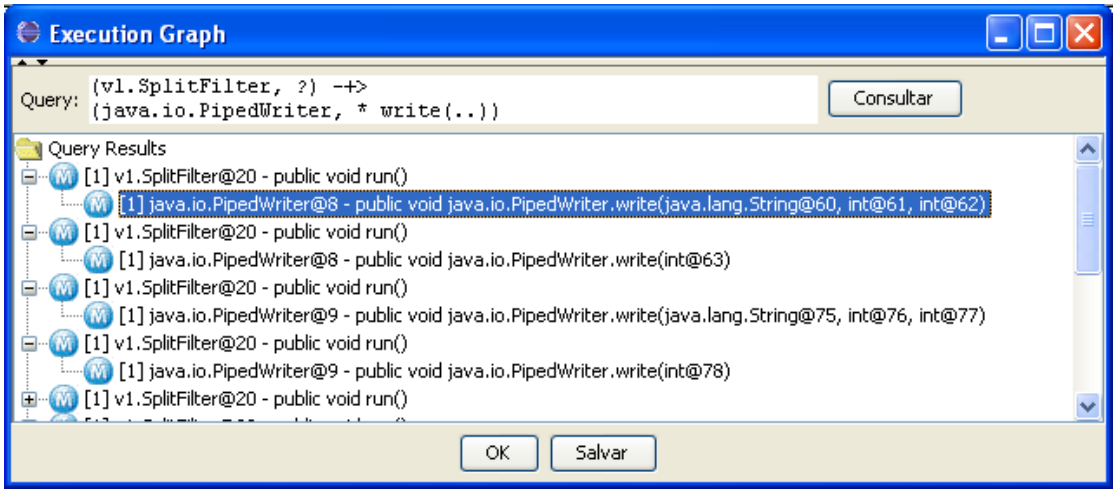

Figura 7.8. Resultado da consulta

`(v1.PassFilter, ?) -+> (java.io.PipedReader, * read(..))`
O resultado encontra-se na Figura 7.9.

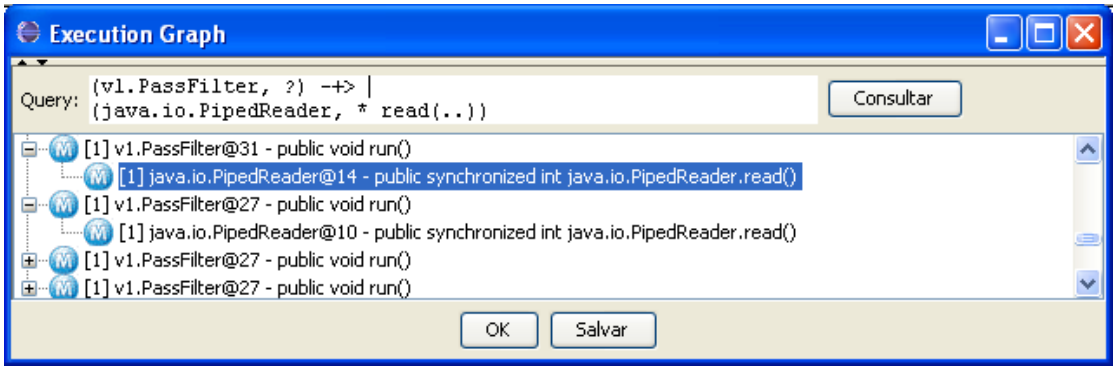

Figura 7.9. Resultado da consulta

Com este resultado e o resultado da consulta da Figura 7.7 conclui-se que:

- Existe um duto entre os componentes `SplitFilter@20` e `PassFilter@31`, pois o objeto `java.io.PipedReader@14` foi conectado ao objeto `java.io.PipedWriter@9` e o componente `PassFilter@31` lê seus dados através da porta `java.io.PipedReader@14`, como mostra a Figura 7.9.

- Existe um outro duto, porém entre os componentes `SplitFilter@20` e `PassFilter@27`, pois o objeto `java.io.PipedReader@10` foi conectado ao objeto `java.io.PipedWriter@8` e o componente `PassFilter@27` lê seus dados através da porta `java.io.PipedReader@10`, como mostra a Figura 7.9.

A próxima consulta serve para mostrar que os compoentes do tipo `PassFilter` escrevem em suas portas de escrita:

```
(v1.PassFilter, ?)  -+> (java.io.PipedWriter, * write(..))
```

O resultado da consulta pode ser visualizado na Figura 7.10.

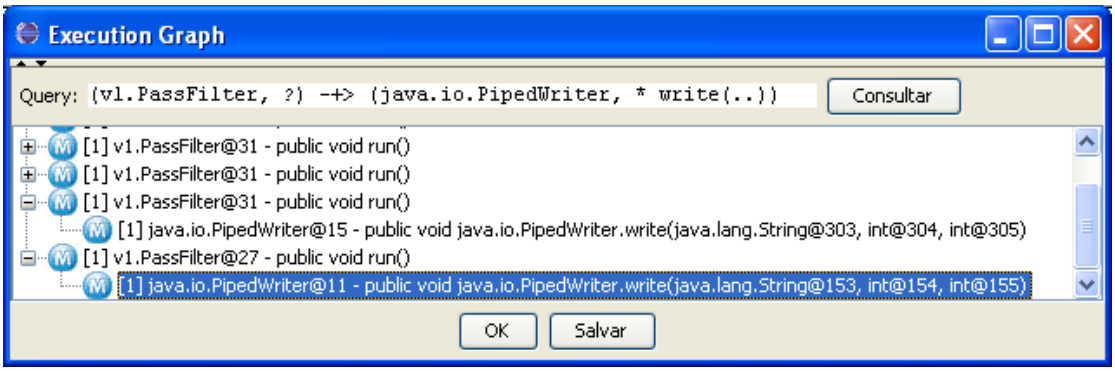

Figura 7.10. Resultado da consulta

Para terminar, basta identificar de onde o componente `v1.MergeFilter@33` lê seus dados. Para isto, pode-se utilizar a seguinte consulta:

```
(v1.MergeFilter, ?)  -+> (java.io.PipedReader, * read(..))
```

O resultado encontra-se na Figura 7.11.

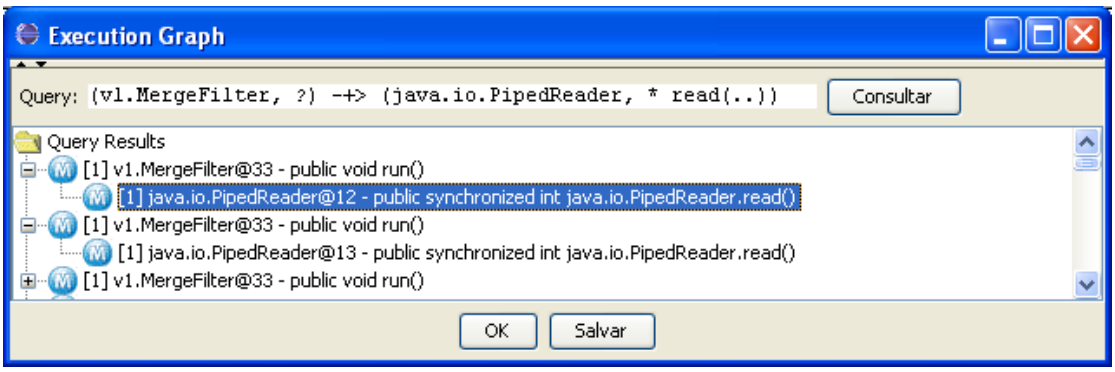

Figura 7.11. Resultado da consulta

Com este resultado e o resultado da consulta da Figura 7.7 conclui-se que:

- Existe um duto entre os componentes `PassFilter@27` e `MergeFilter@33`, pois o objeto `java.io.PipedReader@12` foi conectado ao objeto `java.io.PipedWriter@11` e o componente `MergeFilter@33` lê seus dados através da porta `java.io.PipedReader@12`, como mostra a Figura 7.11.
- Existe um duto entre os componentes `PassFilter@31` e `MergeFilter@33`, pois o objeto `java.io.PipedReader@13` foi conectado ao objeto

`java.io.PipedWriter@15` e o componente `MergeFilter@33` lê seus dados através da porta `java.io.PipedReader@13`, como mostra a Figura 7.11.

Assim, pode-se concluir a descrição da arquitetura do programa `RegSys` adicionando os dutos que foram descobertos. A descrição completa da arquitetura recuperada pode ser visualizada na Figura 7.12 e é equivalente à descrição apresentada em DiscoTect [34]. Entretanto, por se tratar de um sistema pequeno, com apenas 4 classes, não se pode garantir a validade da ferramenta *ArchRecover* para sistemas de grande porte. No Apêndice A encontra-se a descrição da arquitetura escrita em ACME.

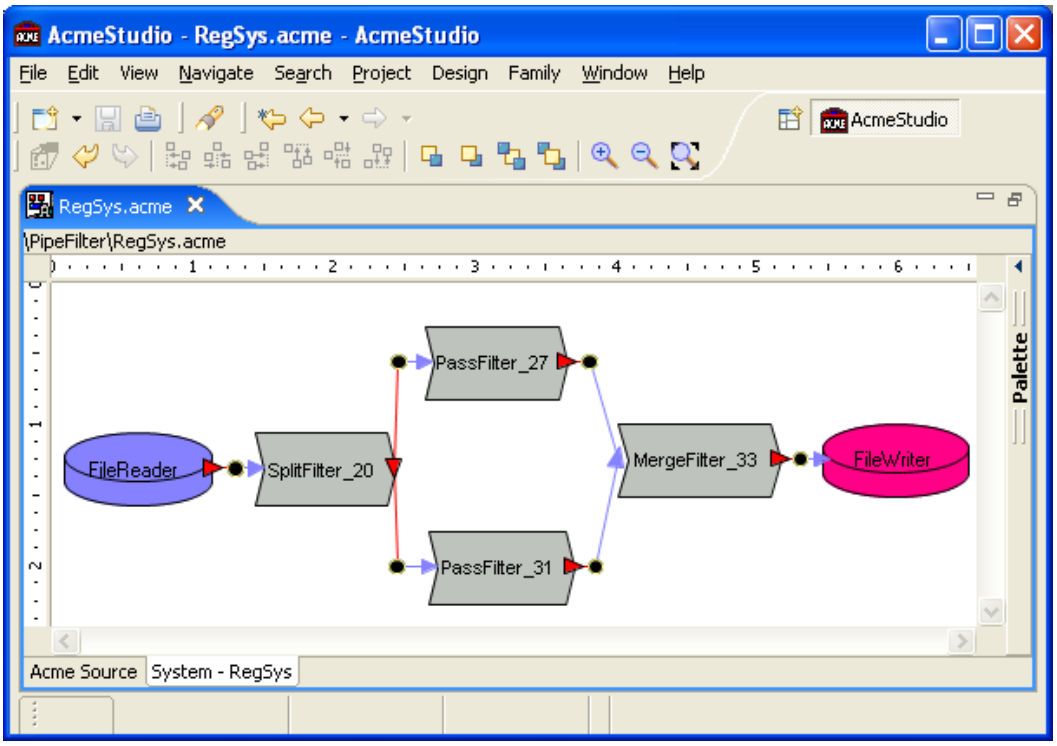

Figura 7.12. Descrição da Arquitetura Recuperada do programa `RegSys`

7.1.2 CUP

Esta seção apresenta um outro estudo de caso cujo objetivo é a recuperação da arquitetura do gerador de analisadores sintático LALR `CUP` [18]. Esta ferramenta possui uma linguagem para especificação de gramáticas livres do contexto cuja sintaxe é semelhante à BNF. A partir de uma especificação livre de contexto, o gerador `CUP` é capaz de construir um analisador sintático LALR em Java para a linguagem especificada. O `CUP` possui 40 classes e 5623 linhas de código.

A classe principal do `CUP` foi executada em conjunto com o Aspecto de Execução para que os eventos fossem capturados e a ferramenta *ArchRecover* criasse a representação da execução do programa por meio do grafo de execução – Figura 7.13. Durante a execução do programa, somente a *Thread* principal foi criada. Analisando o trecho da árvore de execução da *Thread* principal mostrado na Figura 7.13 pode-se imaginar uma seqüência de ações que o `CUP` faz para gerar o analisador sintático de uma linguagem. Em resumo, destaca-se os seguintes eventos:

1. `(java_cup.Main@2, void parser_args(java.lang.String[]@3))`: este método recebe o vetor de argumentos que o método principal do programa recebe como parâmetro. A idéia que pode-se ter deste método é que ele processa os argumentos do programa, configurando a execução de acordo com os argumentos recebidos.

2. `(java.io.BufferedInputStream@32, BufferedInputStream(java.io.InputStream@26))`: este evento é um forte indício de que o objeto `BufferedInputStream@32` seja o fluxo de dados do arquivo de entrada. A seguinte consulta é capaz de comprovar esta suspeita:
   ```
   (java_cup.Main, void main(java.lang.String[]))
   -+> (java.io.InputStream+, new(..))
   ```
   O resultado pode ser visualizado na Figura 7.14.

O próximo passo é identificar qual componente é responsável pela leitura do arquivo de entrada. Por se tratar de um gerador de analisador sintático, é provável que seja encontrado um analisador léxico. Utilizando evento em destaque Número 2, cujo resultado da consulta da Figura 7.14 comprova ser o fluxo de dados do arquivo de entrada, pode-se utilizar a seguinte consulta para identificar o analisador léxico:

```
(*, ?)  -> (java.io.InputStream+, * read*(..))
```

A Figura 7.15 mostra que o componente que lê os dados do arquivo de entrada é o componente `java_cup.lexer@3711`.

A fim de identificar uma restrição arquitetural de que somente o componente `java_cup.lexer@3711` lê o arquivo de entrada, pode-se utilizar a seguinte consulta para procurar por outros componentes que não sejam do tipo `java_cup.lexer` e que leiam algum fluxo de dados de entrada:

```
(!java_cup.lexer, ?)  -> (java.io.InputStream+, * read*(..))
```

Ao executar a consulta, nenhum resultado é encontrado. Sendo assim, o componente `java_cup.lexer@3711` é de fato o responsável pela leitura do arquivo de entrada.

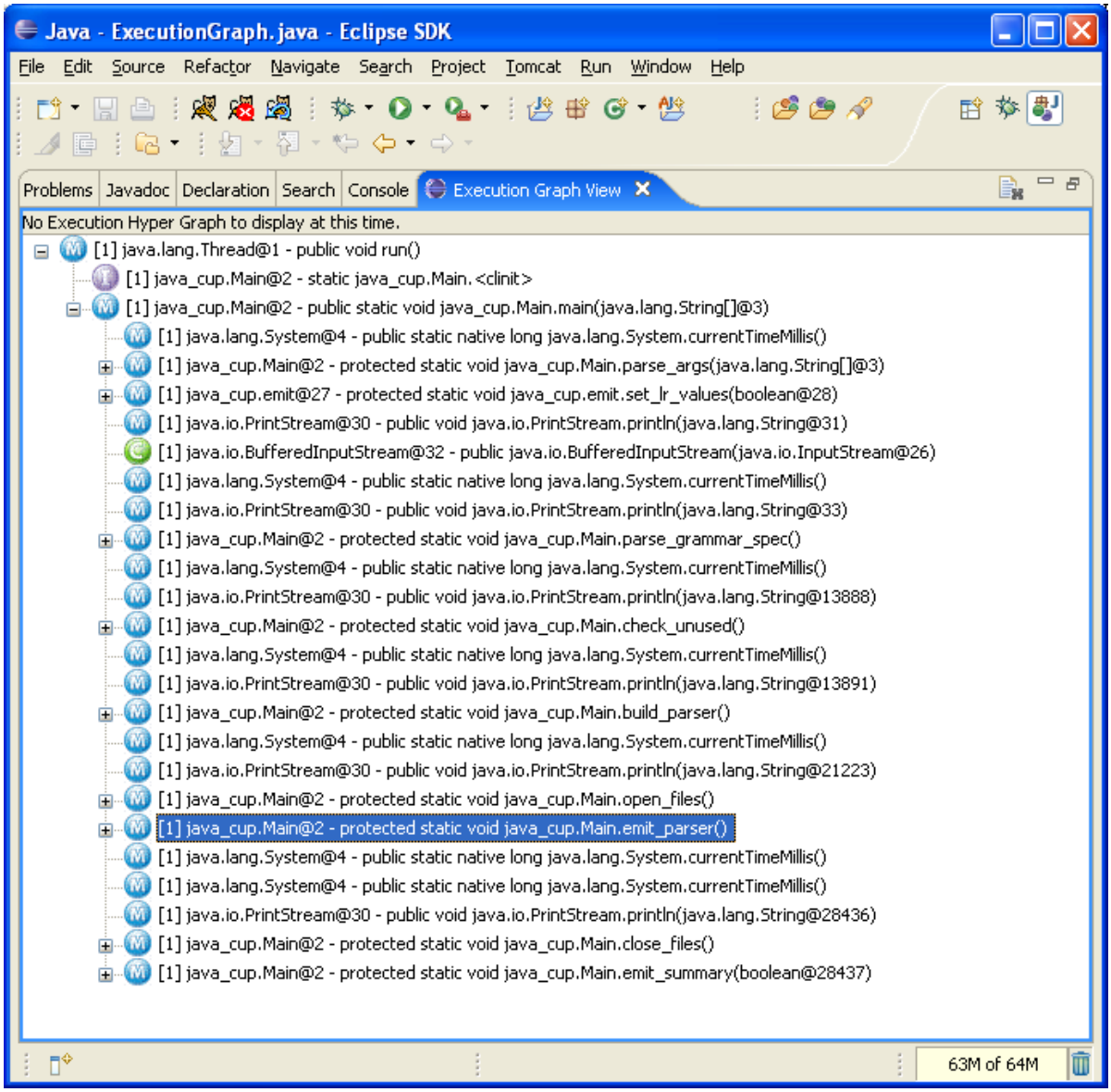

Figura 7.13. Grafo de Execução do programa `CUP`

O próximo passo é identificar o componente que utiliza o analisador léxico. Para isto, pode-se utilizar a consulta:

```
(*, ?)  -> (java_cup.lexer, ?)
```

A consulta produz como resultado inúmeras linhas, dentre as quais aparecem os seguintes objetos:

- `java_cup.parser@3705`;
- `java_cup.lexer@3711`.

Isto indica que o componente `java_cup.lexer@3711` invoca seus próprios métodos.

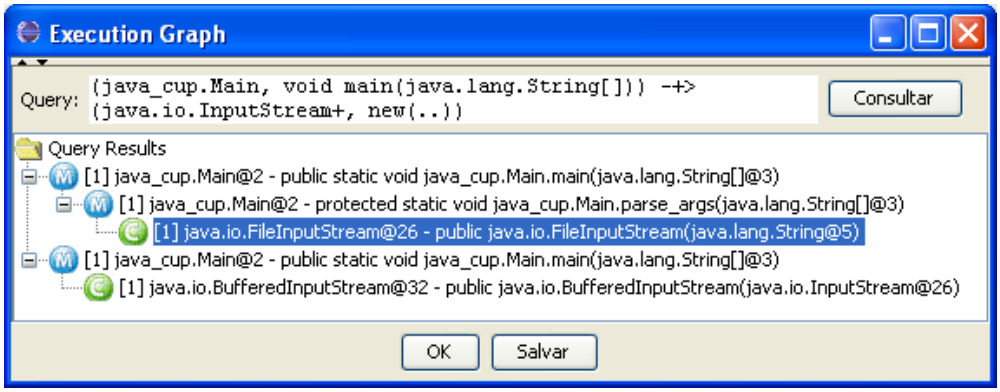

Figura 7.14. Resultado da consulta

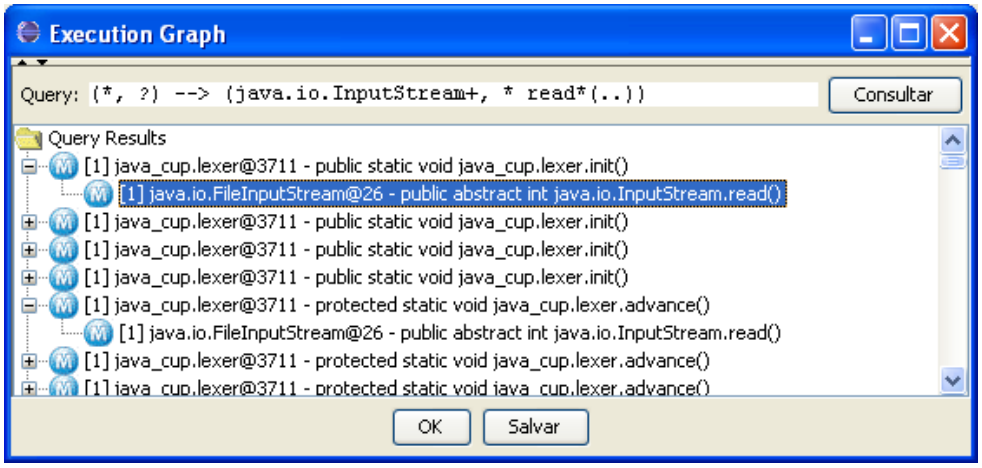

Figura 7.15. Resultado da consulta

Para filtrar o resultado, pode-se utilizar a seguinte consulta mais específica: `(!java_cup.lexer, ?) -> (java_cup.lexer, ?)` O resultado pode ser visualizado na Figura 7.16. Com este resultado, encontra-se um importante relacionamento entre os componentes `java_cup.parser@3705` e `java_cup.lexer@3711`: a invocação do método `java_cup.runtime.Symbol next_token()`. Assim, foi descoberto mais um componente importante da arquitetura do `CUP`, o analisador sintático de especificações cup.

Para saber qual componente utiliza o analisador sintático, utiliza-se a seguinte consulta:

```
(java_cup.Main, void main(java.lang.String[]))
-+> (java_cup.parser, java_cup.runtime.Symbol scan())
```

A Figura 7.17 apresenta o resultado desta consulta.

O próximo passo é identificar a geração dos arquivos produzidos pelo gerador de analisador sintático `CUP`. Para isto, pode-se utilizar a seguinte consulta:

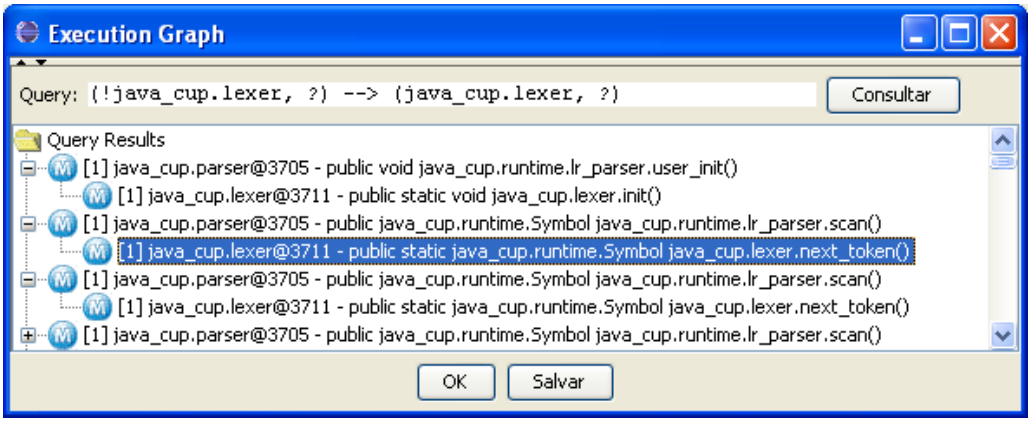

Figura 7.16. Resultado da consulta

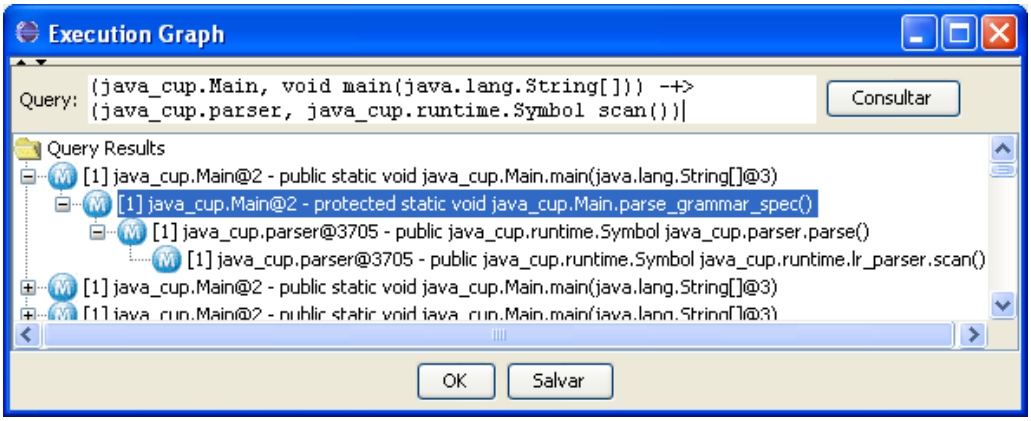

Figura 7.17. Resultado da consulta

```
(*, ?)  -> (java.io.OutputStream+, new(..))
```

A Figura 7.18 mostra o resultado desta consulta.

Os resultados mostram que foram criados fluxos de saída para 2 arquivos:

- `java.io.File@21227`;
- `java.io.File@21235`.

Supondo que os arquivos gerados serão gravados utilizando a API da classe abstrata `java.io.OutputStream` poderia ser utilizada a seguinte consulta:

```
(*, ?)  -> (java.io.OutputStream+, void write*(..))
```

Entretanto, nenhum resultado é encontrado. Em outra tentativa, pode-se utilizar a próxima consulta, supondo que foi utilizada a API da classe abstrata `java.io.Writer`:

```
(*, ?)  -> (java.io.Writer+, void write*(..))
```

Novamente, nenhum resultado é encontrado. Logo, tem-se a necessidade de descobrir

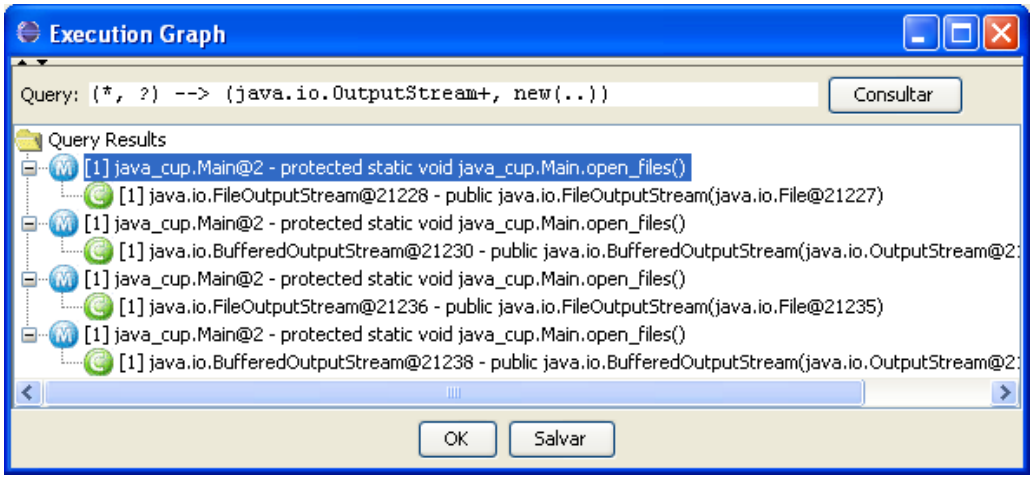

Figura 7.18. Resultado da consulta

qual API foi utilizada para gravar os arquivos de saída. Então, pode-se utilizar a seguinte consulta para descobrir quais objetos receberam como parâmetro as instâncias de `java.io.OutputStream`:

```
(*, ?)  -> (*, new(java.io.OutputStream+))
```

O resultado desta consulta pode ser visualizado na Figura 7.19 e mostra que a API utilizada para gravar os arquivos de saída foi a classe `java.io.PrintWriter`.

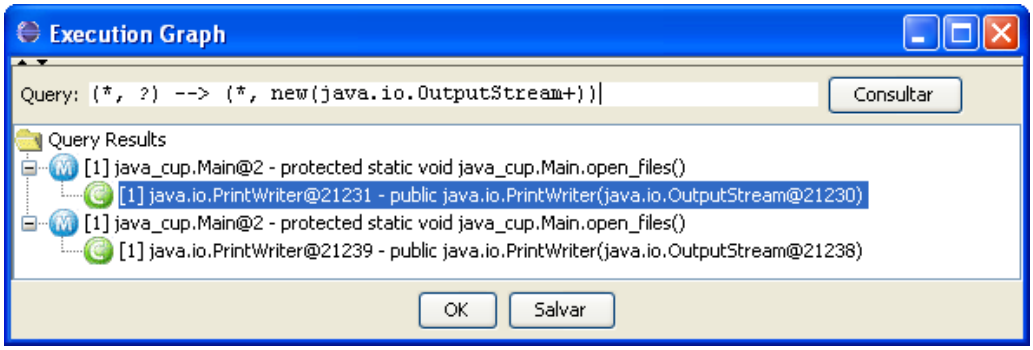

Figura 7.19. Resultado da consulta

Para identificar o componente responsável pela geração dos arquivos de saída, pode-se utilizar a seguinte consulta:

```
(*, ?)  -> (java.io.PrintWriter, * print*(..)  || * write*(..))
```

A Figura 7.19 apresenta o resultado da consulta: o componente que gera os arquivos de saída é o `java_cup.emit@27`.

A fim de identificar outra restrição arquitetural de que somente o componente `java_cup.emit@27` grava nos arquivo de saída, pode-se utilizar a seguinte consulta para procurar por outros componentes que não sejam do tipo `java_cup.emit` e que

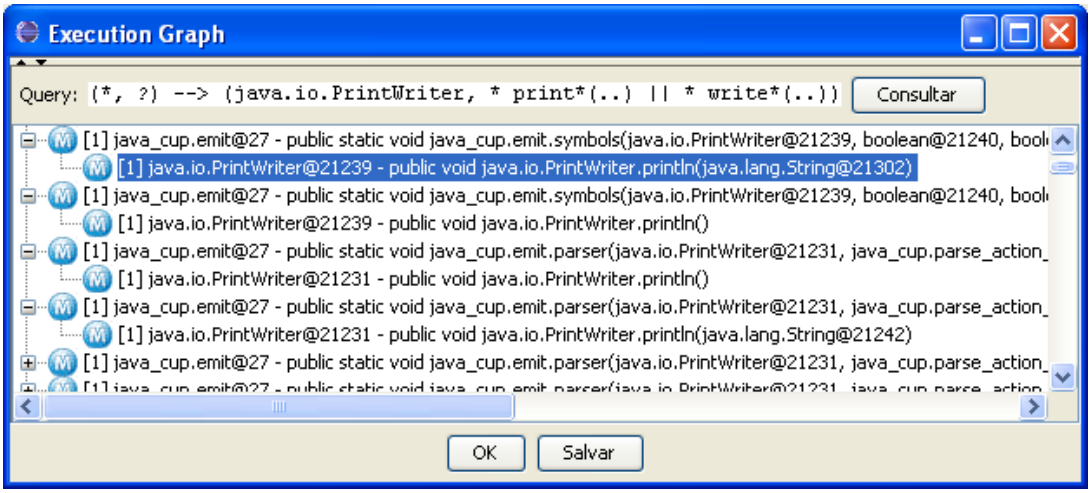

Figura 7.20. Resultado da consulta

escrevam em algum fluxo de dados de saída:

```
(!java_cup.emit, ?)  -> (java.io.PrintWriter, * print*(..)
|| * write*(..))
```

Ao executar a consulta, nenhum resultado é encontrado. Logo, o componente `java_cup.emit@27` é de fato o responsável pela geração dos arquivos de saída.

Com todos estes resultados pode-se construir a descrição da arquitetura do gerador de analisadores sintáticos `CUP` usando o AcmeStudio. A arquitetura recuperada pode ser visualizada na Figura7.21.

7.2 Estudos de Caso: Verificação de Conformidades

Para fazer a verificação de conformidades, foi escolhido um programa simples, problema do produtor-consumidor e que possui uma arquitetura relativamente pequena. O problema é composto de três módulos: Produtor, Consumidor e Buffer. Cada módulo possui sua funcionalidade intrínseca, requisitos funcionais, e interagem entre si da seguinte forma:

- o Produtor produz itens requerendo o serviço específico para armazenar itens no Buffer (método put());
- o Buffer armazena e fornece itens, por meio dos serviços (métodos put() e get());
- o Consumidor consome itens, requerendo o serviço específico para retirar itens do Buffer (método get()).

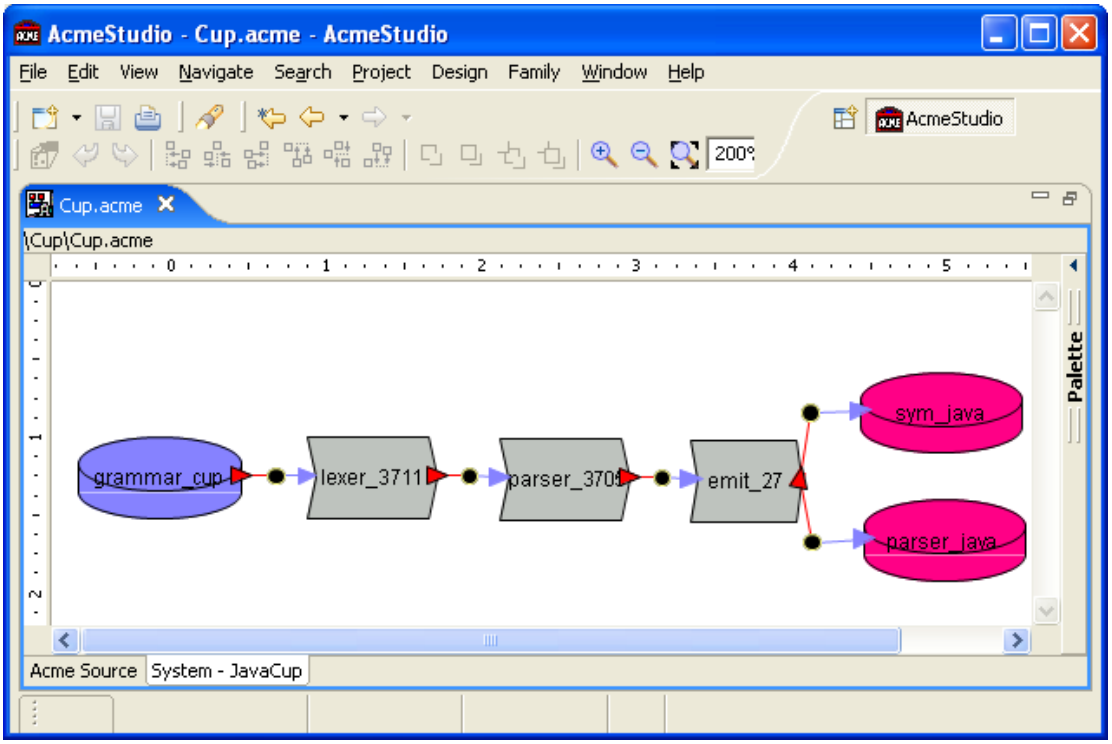

Figura 7.21. Descrição da Arquitetura do CUP

A seguir, apresenta-se a implementação em Java desse problema.

```
1  package problempc;
2  public class Buffer {
3      private int[] buffer;
4      private int quant;
5      public Buffer(int pQuant) {
6          buffer = new int[pQuant];
7          quant = 0;
8      }
9      public synchronized void put(int pItem) {
10         while (quant >= buffer.length)
11             try {
12                 wait();
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         buffer[quant++] = pItem;
17         notify();
18     }
19     public synchronized int get() {
20         while (quant <= 0)
21             try {
```

```
22                 wait();
23             } catch (InterruptedException e) {
24                 e.printStackTrace();
25             }
26         int lItem;
27         lItem = buffer[--quant];
28         notify();
29         return lItem;
30     }
31 }
```

```
1  package problempc;
2  public class Producer extends Thread {
3      private Buffer buffer;
4      public Producer(Buffer pBuffer) {
5          buffer = pBuffer;
6      }
7      public void run() {
8          for (int i = 0; i < 100; i++) {
9              buffer.put(i);
10             System.out.println("put " + i);
11         }
12     }
13 }
```

```
1  package problempc;
2  public class Consumer extends Thread {
3      private Buffer buffer;
4      public Consumer(Buffer pBuffer) {
5          buffer = pBuffer;
6      }
7      public void run() {
8          while (true) {
9              int j = buffer.get();
10             System.err.println("get " + j);
11         }
12     }
13 }
```

```
1  package problempc;
2  public class Problem {
3      public static void main(String[] args) {
4          Buffer lBuffer = new Buffer(2);
5          Consumer lConsumer = new Consumer(lBuffer);
6          Producer lProducer = new Producer(lBuffer);
```

```
7           lConsumer.start();
8           lProducer.start();
9           try {
10              lProducer.join();
11              lBuffer.setFinish(true);
12          } catch (InterruptedException e) {
13              e.printStackTrace();
14          }
15      }
16  }
```

A seguir apresenta-se a descrição da arquitetura desse problema em ACME, representada visualmente na Figura 7.22.

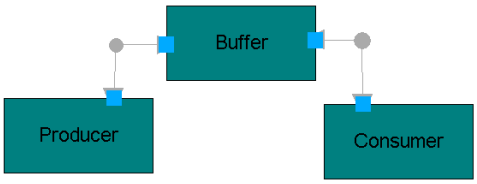

Figura 7.22. Descrição da arquitetura do Problema Produtor-Consumidor

```
1   System PCProblem = {
2       Component Producer = {
3           Port method_0 = {
4               Property returnType : string = "void";
5               Property methodName : string = "run";
6               Property parameters : string = "()";
7           };
8       };
9       Component Consumer = {
10          Port method_0 = {
11              Property returnType : string = "void";
12              Property methodName : string = "run";
13              Property parameters : string = "()";
14          };
15      };
16      Component Buffer = {
17          Port method_0 = {
18              Property returnType : string = "void";
19              Property methodName : string = "put";
20              Property parameters : string = "(int)";
21          };
22          Port method_1 = {
23              Property returnType : string = "int";
```

```
24              Property methodName : string = "get";
25              Property parameters : string = "()";
26          };
27      };
28      Connector methodCall_0 = {
29          Role target;
30          Role source;
31          Property method : string = "void problempc.Buffer.put(int)";
32          Property caller : string = "problempc.Producer";
33          Property callee : string = "problempc.Buffer";
34      };
35      Connector methodCall_1 = {
36          Role source;
37          Role target;
38          Property method : string = "int problempc.Buffer.get()";
39          Property caller : string = "problempc.Consumer";
40          Property callee : string = "problempc.Buffer";
41      };
42      Attachment Producer.method_0 to methodCall_0.source;
43      Attachment Buffer.method_0 to methodCall_0.target;
44      Attachment Consumer.method_0 to methodCall_1.source;
45      Attachment Buffer.method_1 to methodCall_1.target;
46  };
```

A descrição da arquitetura deverá fornecer as informações necessárias para anotar os pontos-chave do programa. Cada tipo de conector tem uma série de propriedades para armazenar os dados necessários para verificação de conformidade. Tomando como exemplo as chamadas de métodos, as propriedades relevantes são:

- method: a assinatura do método que foi invocado;
- caller: qual o tipo do objeto que invocou o método;
- callee: tipo do objeto cujo método foi invocado.

Outra informação contida na descrição da arquitetura é quais componentes o conector liga.

O gerador de pontos-chaves analisa a descrição da arquitetura identificando os conectores que forem do tipo methodCall_[0-9]+ . Esses conectores terão as propriedades method, caller e callee. Utilizando estas propriedades o gerador produz o seguinte arquivo, com as declarações de anotações nos pontos-chave.

```
1  public aspect ProblemPCKeyPoints {
2  declare @method :
```

```
3     int problempc.Buffer.get() :
4     @MethodCall(method = "int problempc.Buffer.get()",
5                 caller = "problempc.Consumer",
6                 callee = "problempc.Buffer");
7  declare @method :
8     void problempc.Buffer.put(int) :
9     @MethodCall(method = "void problempc.Buffer.put()",
10                caller = "problempc.Producer",
11                callee = "problempc.Buffer");
12 }
```

Executando o programa do problema consumidor com o aspecto que declara os pontos chaves de verificação no código e o aspecto de verificação de conformidades, a ferramenta *ArchVerifier* apresenta o resultado de quais pontos-chave do programa foram verificados ou não.

Capítulo 8

Conclusão

Este trabalho apresentou um estudo sobre o tema arquitetura de software e suas representações existentes, enfocando as linguagens para descrição de arquitetura. Em seguida, o texto apresentou a primeira técnica dinâmica, em tempo de execução, para recuperação da arquitetura de software, DiscoTect [34], e uma técnica para verificação de consistência entre a arquitetura de software e implementação de sistemas, ArchJava [2]. Com esses estudos, identificaram-se alguns problemas pertinentes à manutenção de sistemas de software relacionados à arquitetura de *software*:

- necessidade de obter a arquitetura de *software* de um sistema que foi construído sem uma arquitetura projetada previamente;
- necessidade de verificar se a implementação atual de um sistema está de acordo com sua arquitetura projetada previamente.

Para auxiliar os arquitetos de *software* na recuperação da arquitetura dos sistemas este trabalho propôs uma representação da execução de programas Java, chamado Grafo de Execução e uma linguagem de consultas sobre este grafo. Com uma notação semelhante aos elementos de AspectJ, a linguagem permite operações que são comuns na recuperação da arquitetura de *frameworks*, como o caso do operador `+` que verifica se o contexto de um ponto de execução é um subtipo do padrão utilizado na consulta e do caractere curinga `?`, que casa com qualquer ponto de execução. Tais operações não existem em DiscoTect [34].

Outra contribuição deste trabalho em relação à DiscoTect é que após ter executado o caso de uso para o qual deseja-se construir a arquitetura de *software*, o arquiteto poderá especificar várias consultas sobre a execução do programa, o que facilitará a etapa de refinamento dos eventos pertinentes à arquitetura do sistema. Essa contribuição é muito importante pois pode ser muito trabalhoso repetir a execução idêntica de

um caso de uso ou pode ser impraticável reproduzir a mesma execução do programa, em casos de concorrência. Em DiscoTect é necessário construir uma nova máquina de estado para cada etapa de refinamento e tentar reproduzir a mesma execução do caso de uso.

Diferente de DiscoTect, a metodologia para recuperação da arquitetura de *software* da Seção 5.3 parte do pressuposto que o arquiteto de *software* gastará um tempo inicial investigando indícios do nome dos principais componentes da arquitetura na documentação existente do sistema.

Assim como em DiscoTect, a solução proposta também é fortemente ligada aos nomes dos componentes e de suas operações. Por exemplo, a modificação do nome de uma classe pode invalidar a expressão da linguagem de consulta que identificava um componente da arquitetura de *software*.

A ferramenta *ArchVerifier* para verificação de conformidades entre a descrição da arquitetura de *software* e a implementação atual do sistema foi projetada de modo que não ficasse tão acoplada e intrusiva como ArchJava. *ArchVerifier* utiliza uma linguagem para especificar restrições arquiteturais cujos elementos são um subconjunto das produções da linguagem de consultas sobre o grafo de execução. Isto facilita a especificação das restrições arquiteturais.

Outro aspecto positivo dessa solução é que a descrição da arquitetura do sistema fica em um arquivo específico para este propósito, separado do código fonte do sistema, e não espalhada no código fonte, como é feito em ArchJava. Desta maneira, é possível entender a arquitetura do sistema de forma mais fácil e rápida, pois a especificação da arquitetura está em um documento de nível mais alto, contendo apenas os principais componentes e conectores da arquitetura. Além disto, esse desacoplamento permite que os desenvolvedores se preocupem com a implementação do sistema de acordo com o projeto da arquitetura do *software*. Já a responsabilidade de escrever as restrições arquiteturais, de gerar as anotações nos pontos-chave e de executar o sistema em conjunto com o aspecto de verificação fica a cargo do arquiteto de *software*.

Além dessas contribuições, destaca-se a possibilidade de verificar se alguma especialização de um *framework* obedece aos comportamentos esperados por sua especificação, definidos como restrições arquiteturais nos conectores da descrição da arquitetura do *framework*.

8.1 Trabalhos Futuros

Como trabalhos futuros, propõe-se:

- implementação de algumas melhorias no Eclipse [ecl]:
  - uma *view* para executar as consultas sobre o grafo de execução do programa;
  - uma *view* para mostrar o resultado da verificação de conformidade.
- especificação formal da linguagem de consultas sobre o grafo de execução;
- especificação formal da linguagem de restrições arquiteturais;
- implementação de um recuperador automático da arquitetura de *software* que utilize a linguagem de consultas sobre o grafo de execução para construir a arquitetura do sistema;
- validação dos resultados com sistemas de grande porte.

Referências Bibliográficas

[ecl] Eclipse Home Page. http://www.eclipse.org/.

[2] Aldrich, J.; Chambers, C. & Notkin, D. (2002). Archjava: connecting software architecture to implementation. Em *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pp. 187–197.

[3] Allen, R. (1997). *A Formal Approach to Software Architecture*. Tese de doutorado, Carnegie Mellon, School of Computer Science.

[4] Apache Tomcat Project (2006). Apache tomcat architecture. http://tomcat.apache.org/tomcat-5.5-doc/architecture/index.html. Last visited in March 2008.

[5] Astudillo, H. & Hammer, S. (1998). Understanding the architect's job. Em *Software Architecture Workshop of OOPSLA'98*.

[6] Bass, L.; Clements, P. & Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley, 1st edição.

[7] Binns, P.; Englehart, M.; Jackson, M. & Vestal, S. (1996). Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201--227.

[8] Booch, G.; Rumbaugh, J. & Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

[9] Braga, T.; Maia, M. & Bigonha, R. (2008). Recovering and checking software architectural properties based on execution tree analysis. Em *Anais do II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software*, pp. 1–14.

[10] Chikofsky, E. J. & Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

[11] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R. & Stafford, J. (2002). *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 1st edição.

[12] Dijkstra, E. W. (1972). The humble programmer. Em *Communications of the ACM*, volume 15, pp. 859–866. ACM Press.

[13] Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1st edição.

[14] Garlan, D. (1995). An introduction to the aesop system. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesopoverview.ps. Last visited in March 2008.

[15] Garlan, D. (2000). Software architecture: a roadmap. Em *The Future of Software Engineering.* ACM Press.

[16] Garlan, D.; Monroe, R. & Wile, D. (1997). Acme: an architecture description interchange language. Em *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, p. 7. IBM Press.

[17] Garlan, D.; Monroe, R. T. & Wile, D. (2000). Acme: Architectural description of component-based systems. Em *Foundations of Component-Based Systems*, pp. 47–68. Cambridge University Press.

[18] Hudson, S. E. (1999). Cup LALR parser generator for Java – User's Manual. *Available at http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html*.

[19] Jazayeri, M.; Ran, A.; Linden, F. V. D. & Linden, P. V. D. (2000). *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, 1st edição.

[20] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. & Griswold, W. G. (2001). An overview of aspectj. Em *ECOOP '01.*

[21] Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50.

[22] Kruchten, P. & Thompson, C. J. (1994). An object-oriented, distributed architecture for large scale ada systems. *Proceedings of the TRI-Ada '94 Conference*, pp. 262–271.

[23] Luckham, D. C.; Kenney, J. L.; Augustin, L. M.; Vera, J.; Bryan, D. & Mann, W. (1995). Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336--355.

[24] Luckham, D. C. & Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717--734.

[25] Magee, J.; Dulay, N.; Eisenbach, S. & Kramer, J. (1995). Specifying Distributed Software Architectures. Em *Fifth European Software Engineering Conference, ESEC '95 , Barcelona*.

[26] Magee, J. & Kramer, J. (1996). Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3--14.

[27] Medvidovic, N.; Oreizy, P.; Robbins, J. E. & Taylor, R. N. (1996). Using object-oriented typing to support architectural design in the c2 style. Em *Proceedings of ACM SIGSOFT '96: 4th Symposium on the Foundations of Software Engineering*, pp. 24--32.

[28] Medvidovick, N. & Taylor, R. (2000). A classification and comparison framework for software architecture. Em *IEEE Trans. Software Engineering*, volume 26(1).

[29] Sebesta, R. W. (1993). *Concepts of Programming Languages*. BENJAMIN CUMMINGS, 2nd edição.

[30] Shaw, M. & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1st edição.

[31] Sun Microsystems, Inc. (1999). Enterprise javabeans specification, v1.1. http://java.sun.com/products/ejb/docs.html. Last visited in March 2008.

[32] Taylor, R. N.; Medvidovic, N.; Anderson, K. M.; E. James Whitehead Jr., J. E. R.; Nies, K. A.; Oreizy, P. & Dubrow, D. L. (1996). A component and message-based architectural style for gui software. volume 22, pp. 390--406. IEEE Transactions on Software Engineering.

[33] van Deursen, A.; Hofmeister, C.; Koschke, R.; Moonen, L. & Riva, C. (2004). Symphony: View-driven software architecture reconstruction. Em *WICSA' 2004: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*, pp. 122–132.

[34] Yan, H.; Garlan, D.; Schmerl, B.; Aldrich, J. & Kazman, R. (2004). Discotect: A system for discovering architectures from running systems. Em *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pp. 470--479, Washington, DC, USA. IEEE Computer Society.

Apêndice A

Descrição da Arquitetura do programa RegSys em ACME

```
1  import $AS_GLOBAL_PATH\families\PipesAndFiltersFam.acme;
2  System RegSys : PipesAndFiltersFam =  new PipesAndFiltersFam extended with {
3      Component SplitFilter_20 : Filter =  new Filter extended with {
4          Property flowPaths : Set{flowpathRecT} = {[
5              fromPt : string = "input";
6              toPt : string = "output" ]};
7      };
8
9      Component PassFilter_27 : Filter =  new Filter extended with {
10         Property flowPaths : Set{flowpathRecT} = {[
11             fromPt : string = "input";
12             toPt : string = "output" ]};
13     };
14
15     Component PassFilter_31 : Filter =  new Filter extended with {
16         Property flowPaths : Set{flowpathRecT} = {[
17             fromPt : string = "input";
18             toPt : string = "output" ]};
19     };
20
21     Component MergeFilter_33 : Filter =  new Filter extended with {
22         Property flowPaths : Set{flowpathRecT} = {[
23             fromPt : string = "input";
24             toPt : string = "output" ]};
25     };
26
27     Component FileReader : DataSource =  new DataSource;
28
29     Component FileWriter : DataSink =  new DataSink;
30
31     Connector Pipe0 : Pipe =  new Pipe extended with {
32         Property flowPaths : Set{flowpathRecT} = {[
33             fromPt : string = "source";
```

```
34              toPt : string = "sink" ]};
35          Property bufferSize : int = 0;
36      };
37
38      Attachment FileReader.output to Pipe0.sink;
39      Attachment SplitFilter_20.input to Pipe0.source;
40      Connector Pipe1 : Pipe =  new Pipe extended with {
41          Property flowPaths : Set{flowpathRecT} = {[
42              fromPt : string = "source";
43              toPt : string = "sink" ]};
44          Property bufferSize : int = 0;
45      };
46
47      Attachment MergeFilter_33.output to Pipe1.sink;
48      Attachment FileWriter.input to Pipe1.source;
49      Connector Pipe2 : Pipe =  new Pipe extended with {
50          Property flowPaths : Set{flowpathRecT} = {[
51              fromPt : string = "source";
52              toPt : string = "sink" ]};
53          Property bufferSize : int = 0;
54      };
55
56      Attachment PassFilter_27.input to Pipe2.source;
57      Connector Pipe3 : Pipe =  new Pipe extended with {
58          Property flowPaths : Set{flowpathRecT} = {[
59              fromPt : string = "source";
60              toPt : string = "sink" ]};
61          Property bufferSize : int = 0;
62      };
63
64      Attachment PassFilter_31.input to Pipe3.source;
65      Attachment SplitFilter_20.output to Pipe2.sink;
66      Attachment SplitFilter_20.output to Pipe3.sink;
67      Connector Pipe4 : Pipe =  new Pipe extended with {
68          Property flowPaths : Set{flowpathRecT} = {[
69              fromPt : string = "source";
70              toPt : string = "sink" ]};
71          Property bufferSize : int = 0;
72      };
73
74      Attachment PassFilter_27.output to Pipe4.sink;
75      Attachment MergeFilter_33.input to Pipe4.source;
76      Connector Pipe5 : Pipe =  new Pipe extended with {
77          Property flowPaths : Set{flowpathRecT} = {[
78              fromPt : string = "source";
79              toPt : string = "sink" ]};
80          Property bufferSize : int = 0;
81      };
82
83      Attachment PassFilter_31.output to Pipe5.sink;
84      Attachment MergeFilter_33.input to Pipe5.source;
```

```
85  };
```

Listing A.1. Arquivo RegSys.acme

Apêndice B

Descrição da Arquitetura do Cup em ACME

```
1  import $AS_GLOBAL_PATH\families\PipesAndFiltersFam.acme;
2  System JavaCup : PipesAndFiltersFam =  new PipesAndFiltersFam extended with {
3      Component parser_java : DataSink =  new DataSink;
4
5      Component sym_java : DataSink =  new DataSink;
6
7      Component grammar_cup : DataSource =  new DataSource;
8
9      Component lexer_3711 : Filter =  new Filter extended with {
10         Property flowPaths : Set{flowpathRecT} = {[
11             fromPt : string = "input";
12             toPt : string = "output" ]};
13     };
14
15     Connector call_constraint_1 : Pipe =  new Pipe extended with {
16         Property flowPaths : Set{flowpathRecT} = {[
17             fromPt : string = "source";
18             toPt : string = "sink" ]};
19         Property bufferSize : int = 0;
20         Property constraint : string = "(java_cup.lexer, ?) --> (java.io.InputStream+, * re
21     };
22
23     Attachment grammar_cup.output to call_constraint_1.sink;
24     Attachment lexer_3711.input to call_constraint_1.source;
25     Component parser_3705 : Filter =  new Filter extended with {
26         Property flowPaths : Set{flowpathRecT} = {[
27             fromPt : string = "input";
28             toPt : string = "output" ]};
29     };
30
31     Connector call_constraint_2 : Pipe =  new Pipe extended with {
32         Property flowPaths : Set{flowpathRecT} = {[
33             fromPt : string = "source";
```

```
34              toPt : string = "sink" ]};
35          Property bufferSize : int = 0;
36      };
37
38      Attachment lexer_3711.output to call_constraint_2.sink;
39      Attachment parser_3705.input to call_constraint_2.source;
40      Component emit_27 : Filter =  new Filter extended with {
41          Property flowPaths : Set{flowpathRecT} = {[
42              fromPt : string = "input";
43              toPt : string = "output" ]};
44      };
45
46      Connector call_constraint_3 : Pipe =  new Pipe extended with {
47          Property flowPaths : Set{flowpathRecT} = {[
48              fromPt : string = "source";
49              toPt : string = "sink" ]};
50          Property bufferSize : int = 0;
51      };
52
53      Attachment parser_3705.output to call_constraint_3.sink;
54      Attachment emit_27.input to call_constraint_3.source;
55      Connector call_constraint_4 : Pipe =  new Pipe extended with {
56          Property flowPaths : Set{flowpathRecT} = {[
57              fromPt : string = "source";
58              toPt : string = "sink" ]};
59          Property bufferSize : int = 0;
60      };
61
62      Attachment emit_27.output to call_constraint_4.sink;
63      Attachment parser_java.input to call_constraint_4.source;
64      Connector call_constraint_5 : Pipe =  new Pipe extended with {
65          Property flowPaths : Set{flowpathRecT} = {[
66              fromPt : string = "source";
67              toPt : string = "sink" ]};
68          Property bufferSize : int = 0;
69      };
70
71      Attachment emit_27.output to call_constraint_5.sink;
72      Attachment sym_java.input to call_constraint_5.source;
73  };
```

Listing B.1. Arquivo JavaCup.acme