UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

Tratamento de Exceções Concorrentes na Linguagem Java

por

Wellington de Oliveira Júnior

Trabalho de Graduação

Centro de Informática

Universidade Federal de Pernambuco (UFPE) – Recife, PE – Brazil

Orientador: Fernando Castor

Recife, 10 de Setembro de 2013

Resumo

Abstract. *This article presents studies on the development of a framework to handle exceptions that occur during an application developed in Java that is using multiple threads, for this it is simulated the use of the try handler but in a concurrent context.*

Resumo. *Este artigo apresenta estudos sobre o desenvolvimento de um mecanismo para tratar exceções ocorridas no decorrer de uma aplicação desenvolvida na linguagem de programação Java onde sejam utilizadas várias threads, para isso é simulando o uso do tratamento try mas em um contexto concorrente.*

Agradecimentos

Gostaria de agradecer primeiramente à minha família, especialmente meus pais, pela forma como fui criado, pelos valores que me foram passados, pela confiança em mim depositada e pela paciência.

Queria agradecer aos meus colegas de univerisade, pelas horas dedicadas aos projetos, as viradas de noites, as paradas no meio do projeto para jogar alguma coisa e por tudo que aprendi com eles durante esses anos de universidade.

Devo agradecer a minha namorada, Bianca Helena, pela paciência com as noites que tinha que passar no CIn, os finais de semana que não tivemos juntos e a como ela conseguiu aguentar a minha irritação nos prazos proximos de entregua. Apesar do que, ela que saiu ganhando por estar comigo, então estamos quites :D

Agradeço também ao meu orientador, Fernando Castor, pela paciência de ensinar e o fazer de uma forma que inspira os alunos a querer aprender, pela confiança em mim me orientando tanto na iniciação científica quanto neste projeto e pela palavras sábias, tanto para minha vida profissional quanto para minha vida pessoal.

Por último, gostaria de agradecer à Universidade Federal de Pernambuco e especialmente ao Centro de Informática e todos os seus professores, por me prover com os meios para poder aprender e assim tornar meus objetivos possíveis.

Obrigado a todos.

Índice

1. Introdução

Threads são fluxos de execução de um código derivados de um fluxo principal, tem como objetivo paralelizar a execução de determinado código e com isso dividir tarefas para fazer melhor uso dos recursos, tanto no caso onde haja mais de um núcleo de processamento quanto no caso em que há uma determinada ação que apresenta um grande tempo de resposta. Ao chavear a execução de threads se obtém um uso ótimo dos recursos o que resulta numa melhor percepção de velocidade e desempenho.

Java fornece suporte extensivo à construção de programas concorrentes e paralelos. A linguagem permite a criação de threads e o gerenciamento de threads, além de vários mecanismos para controlar o acesso de threads a recursos compartilhados, como de variáveis atómicas, monitores e locks. A linguagem também implementa, através de bibliotecas, construções de mais alto nível, como executors, barreiras de sincronização e pools de threads. Outra característica importante de Java é o seu mecanismo de tratamento de exceções. Este permite que erros que aconteçam durante a execução de um método possam ser tratados dentro do código do próprio método ou lançados para qualquer método que venha a chamá-lo.

Embora Java forneça suporte de alto nível tanto à programação concorrente quanto ao tratamento de erros, a linguagem não integra essas funcionalidades de maneira satisfatória. Em particular, quando uma thread executa um método e este produz uma exceção esta pode fazer com que a thread seja finalizada sem notificar nenhuma outra thread que possa, porventura, depender da que falhou. Essa situação pode levar a diversos problemas, como deadlocks, corrupção do estado de variáveis e não-liberação de recursos previamente alocados. Esta situação está em conflito com os objetivos dos projetistas de Java, que ambicionavam desenvolver uma linguagem ao mesmo tempo concorrente e altamente robusta. Apesar disso, não é surpreendente, que ainda haja muito poucos trabalhos apresentando propostas para integrar tratamento de erros e construções para programação concorrente/paralela de forma prática. Algumas das propostas existentes focam em modelos puramente teóricos enquanto outras baseiam-se principalmente em aplicações de tempo real, muito diferentes das aplicações que são e serão executadas em máquinas multi-núcleo.

Buscando uma união entre o tratamento de exceções em Java e o seu suporte a programação concorrente, este trabalho propõe um mecanismo para solucionar a falta de suporte de Java, de uma forma simples, eficiênte e tentando respeitar os objetivos de propostos para que um mecanismo de exceção seja funcional e prático, para que possa um dia ser utilizado pelos desenvolvedores.

O bloco safe tem a sintaxe bastante aproximada com a já familiar solução de tratamento de exceções em Java, o bloco try, e permite ao desenvolvedor solucionar grande parte dos problemas de desenvolvimento concorrente sem precisar aprender uma sintaxe nova ou formas incomuns de programar. Basta colocar o trecho de código dentro do bloco safe, indicar quais são as exceções que podem ocorrer na sua execução e como o programa deverá reagir a elas e toda computação que ocorrer decorrente deste bloco será segura. Ainda havendo a opção do programador determinar que tipo de tratamento de limpeza para garantir que variáveis não vão ficar num estado instável.

2. Contexto

2.1 Tratamento de exceções

O funcionamento de um programa consiste de um fluxo de informações e procedimentos. Em um fluxo normal, o sistema vai de um estado estável para outro estádo estável, mantendo o bom funcionamento do sistema como um todo. Caso aconteça algum tipo de problema no fluxo normal, o programa poderia falhar ou o sistema poderia entrar num estádo não estável. Para que isso não aconteça, desviasse o fluxo para um fluxo alternativo onde esperasse que seja encontrada uma solução para o problema que fez o programa ocorrer em um erro. Este desvio de fluxo é uma exceção. Cada exceção deve ser tratada de acordo com o tratamento de exceção definido na linguagem a qual ela pertence.

O fluxo possível para uma exceção é demonstrado na abaixo, originária de [13].

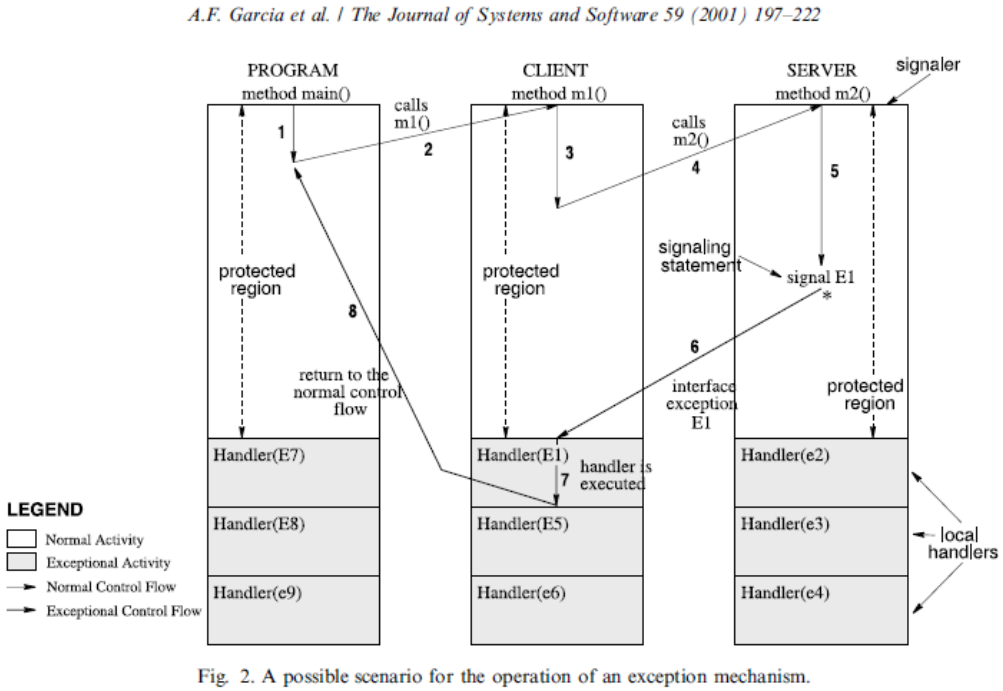

Fig. 2. A possible scenario for the operation of an exception mechanism.

Figura 1. Esquema da ocorrência de uma exceção e seu tratamento.

A figura ilustra a execução de um método dentro de uma região protegida de código. O programa chama o médoto **m1** do **Client** e este chama o método **m2** do

Server. Ao executar o método **m2**, o **Server** dispara uma exceção, que é sinalizada por meio do **signal E1** ao tratador de exceções **Handler E1** do **Client**, este tem os recursos necessários para tornar o programa estável novamente. Ao acabar de tratar a exceção, o **Handler E1** retornar o controle para o fluxo normal e o programa continua a sua execuçao.

Vamos analisar à seguir a classificação de tratamento de exceções sugerida por [13] onde os tratamentos de exceções são classificados de acordo com as seguintes características:

A1-*Representação da exceção*: considerando que exceções podem ser representadas como símbolos, objetos de dados ou objetos completos.

A2-*Exceções externas na assinatura*: o fato de a assinatura dos métodos poder conter dados sobre exceções.

A3-*Separação entre as exceções internas e externas*: diferenciação entre os as exceções internas do programa e as externas.

A4-*Diferentes manipuladores de exceção:* há diferentes tipos de manipuladores de exceções, como uma declaração, um bloco, um método, um objeto, uma classe e uma exceção.

A5-*Acoplamento dos manipuladores de exceção*: medida de nível de acoplamento em que os manipuladores de exceção estão, divididos entre os métodos estático, dinâmico ou semi-dinâmico.

A6-*Propagação das exceções*: referindo-se ao fato das exceções serem propadas para fora dos métodos que as fizeram ocorrer, podendo ser uma propagação explícita ou automática (implícita).

A7-*Continuação do controle de fluxo*: depois de levantada a exceção, há duas maneiras de continuar a execução do código, o modelo de retomada e o modelo de terminio. Dentro do modelo de terminio ainda há uma divisão entre o retorno, a terminação estrita e o tentar novamente.

A8-*Ações de limpeza*: Para manter o programa em um estado consistente é necessário um tratamento de limpeza, estes podem fazer uso de propagação explicita, construtor específico ou limpeza automática.

A9-*Checagem de seguraça*: Testa por possiveis erros inseridos pelo tratamento de exceção, podem ser checagens estáticas ou dinâmicas.

A10-*Tratamento de exceções concorrentes*: Suporte para tratamento de exceções que ocorrem ao utilizar diversos fluxos de execução num programa.

Estas sendo as características da linguagem, ainda a os objetivos que procura-se atingir ao implementar um tratamento de exceções numa linguagem[13], tendo em vista que o tratamento de exceções pode causar certos problemas em alguns aspectos enquanto melhora outros, é importante ter em mente estes objetivos para não sacrificar muito um deles em busca de melhorar algum dos outros. Os objetivos ao implementar tratamento de exceções são: Legibilidade, Modularidade, Manutenabilidade, Reusabilidade, Testabilidade, Facilidade para Escrever, Consistência, Segurança, Simplicidade, Uniformidade, Rastreabilidade e Desempenho.

2.2 Tratamento de Exceções em Java

Dentro as linguagens orientadas a objeto avaliadas quanto as características de seu tratamento de exceção [13] sendo elas: CLU, Ada 95, Lore, SmallTalk, Eifel, Modula-3, C++, Java, Delphi, Guide, Beta e Arche.

“Java suporta a representação de exceções como objetos de dados , a ligação semi-dinâmica , e o modelo de terminio, com retorno. As regiões protegidas em Java são declaradas usando o bloco try. Permite uma melhor verificação estática e fornece suporte específico para a programação de ações de limpeza. No entanto, não há qualquer distinção entre as exceções internas e externas.

Java adota uma solução híbrida para a interface de exceção. Todas as exceções devem ser throwable , ou seja, eles devem ser herdado (direta ou indiretamente) da classe Throwable . Essa classe tem duas subclasses predefinido Error and Exception. O primeiro grupo inclui exceções após o qual não se espera programas comuns para se recuperar (por exemplo, erros de carregamento e e de conexão ou erros relacionados a Virtual Machine) e o segundo, relacionado a exceções que ocorrem dentro do sistema em tempo de execução (por exemplo, aritméticas ,ponteiro, problemas de indexação). O

compilador não requer que os programadores lidem com as exceções não verificadas ou mesmo especificá-los na assinatura do método.

Java fornece programadores com construção try ... finally , que define as ações de limpeza. O bloco finally é sempre executado no final do bloco try, caso ocorra uma exceção ou não, fora nos casos em que o bloco try gera uma exceção que não é capturada por seus manipuladores, caso em que a exceção é propagada.

Embora Java descreve claramente a semântica de tratamento de exceções em programas concorrentes , não oferece um apoio mais geral para o tratamento de exceção concorrente. O mecanismo de exceção Java é integrado com o modelo de thread / sincronização de Java: todos os bloqueios do objeto receptor são liberados quando um método sincronizado sinais de uma exceção para o chamador. Uma exceção assíncrona pode ser criado em uma thread de programa concorrente, invocando o método stop de classe Thread.”[13]

Na pontuação, se o a escolha do tratamento tivesse em vista os objetivos de tratamento de exceções a escolha poderia receber uma pontuação que variava de -1 até 1. O tratamento de exceções de Java foi pontuado de acordo com as características anteriormente descritas da seguinte forma. A1, por utiliza exceções no formato de objetos de dados, Java conseguiu 1 ponto neste quesito. A2, a solução hibrida de Java é considerada ideal e concede mais 1 ponto. A3, nenhuma das linguagens selecionadas apresenta qualquer diferenciação, então todas as linguagens, incluindo Java, receberam -1. A4, O tratamento de Java é atráves de blocos, isso é ruim para a modularização e logo Java recebeu -1 ponto. A5, o tratamento semi-dinâmico de Java garantiu mais 1 ponto. A6, Java contem tanto uma propagação de exceções automática quanto uma explícita, a propagação automática pode levar a erros e é considerada ruim (-1) entretato a explícita ajuda na modularização e na legibilidade (+1), logo Java recebeu 0 pontos neste quesito. A7, Ao terminar a execução mas permitir o retorno ao fluxo normal, Java recebeu mais 1 ponto. A8, Java recebeu mais um ponto pela limpeza explícita. A9, Java contém tanto checagem dinâmica quanto estática, cada uma delas concede 1 ponto, somando 2 pontos. A10, por ter um tratamento de exceções concorrentes limitado, Java recebe 0 pontos neste quesito.

No total, Java acabou em segundo lugar com 6 pontos, dos 15 pontos que eram possíveis de serem conseguidos.Levando em conta a pontuação de Java, o mecanismo proposto vem com objetivo de solucionar o último quesito, tratamento de exceções concorrentes, mantendo em mente os 12 objetivos dos tratamentos de exceção levantados.

2.3 Concorrência em Java

Em programação concorrente há dois conceitos básicos, processos e threads. Processos são ambientes de execução auto-contidos, normalmente ele tem um conjunto privado e completo de recursos de tempo de execução e, em particular, tem seu próprio espaço de memória. Normalmente processos são vistos como sinônimos entre programas.

Threads são fluxos de execução de um código derivados de um fluxo principal, criados dentro da execução do programa, partilham a memória com as outras threads criadas pelo mesmo programa. Threads existem dentro de um processo, considerando que cada processo contem pelo menos uma thread. As threads tem como objetivo paralelizar a execução de determinado código e com isso dividir tarefas para fazer melhor uso dos recursos, tanto no caso onde haja mais de um núcleo de processamento, ou no caso em que há uma determinada ação que apresenta um grande tempo de resposta e pode ser repartida em partes menores e executada de forma paralela ou até o tempo de espera para uma das respostas é muito demorado, como quando é necessesária entrada de dados pelo usuário. Ao chavear a execução de threads se obtém um uso ótimo dos recursos o que resulta numa melhor percepção de velocidade e desempenho.

Java fornece suporte extensivo à construção de programas concorrentes e paralelos. A linguagem permite a criação de threads e o gerenciamento de threads, além de vários mecanismos para controlar o acesso de threads a recursos compartilhados, como de variáveis atómicas, monitores e locks. A linguagem também implementa, através de bibliotecas, construções de mais alto nível, como executors, barreiras de sincronização e pools de threads.

3. Abordagem desenvolvida

O mecanismo desenvolvido da ênfase aos vários problemas inerentes à construção de aplicações paralelas confiáveis e leva em consideração também as particularidades de Java. Por exemplo, threads em Java trabalham com objetos compartilhados, o que aumenta a probabilidade do estado do sistema ser corrompido devido a uma exceção. Além disso, considera questões recentes que surgiram tanto na literatura acadêmica [10, 11] quanto na indústria [12] sobre o projeto de mecanismos de tratamento de exceções, como o uso ou não de exceções verificadas e o uso de especificações locais ou globais para exceções e seus tratadores.

Como decisão de projeto, foi optado por desconsiderar a recuperação do estado estável da aplicação automaticamente no caso de ocorrer uma exceção. Em busca de uma maneira mais intuitiva e mais parecidas com linguagens que foram desenvolvidas pensando nos erros que podem acontecer, como Erlang, foi decidido elimiar a execução de uma thread que venha a levantar uma exceção e sincronizar todas as execuções ao final de um bloco safe, de modo que caso uma thread acabe, naturalmente ou por causa de uma exceção, ele deve esperar todas as threads do bloco acabarem para que o programa retorne a execução normal da thread principal, como ilustra as figuras abaixo. O fato de o programador poder tratar uma exceção que possa vir a ocorrer da a ele o poder de evitar problemas que ocorrem quando uma thread acaba inesperadamente, como deadlocks, corrupção do estado de variáveis e não-liberação de recursos previamente alocados e caso seja necessário, ainda há a opção de realizar ações de limpeza fazendo uso do finally.

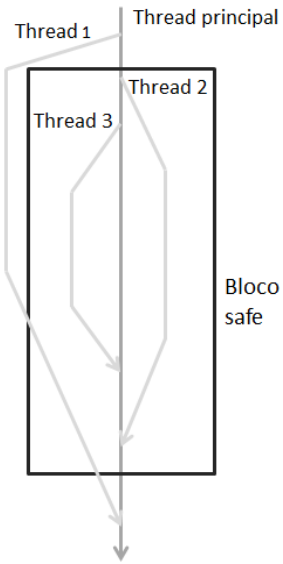

Figura 2: Criação de threads dentro da Thread principal

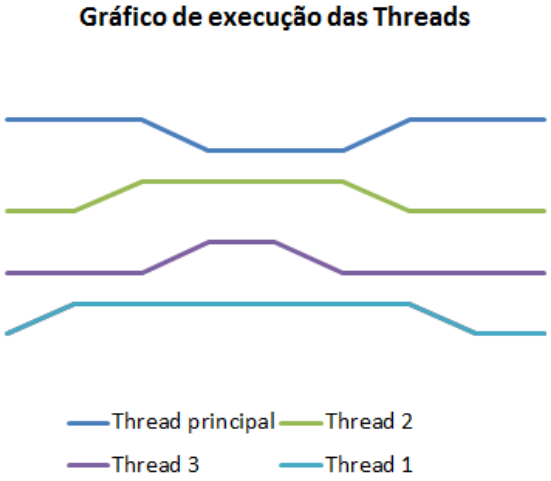

Figura 3: Execução das Threads da figura anterior.

Com isso entendende-se que não só criou-se um mecanismo para tratar exceções concorrentes em Java mas também desenvolveu-se também algo que possa de fato ser utilizado no desenvolvimento de aplicações reais, tentando unir a teoria a prática de desenvolvimento e sanar problemas que afligem os programadores que resolvem desenvolver aplicações concorrentes.

As modificações necessárias incluiram adicionar a palavra reservada safe na linguagem para demarcar um bloco seguro para execução de código concorrente, assim como a palavra reservada try para execução sequencial, e caso não sejam inicializadas

threads dentro do bloco safe então o mesmo funciona da mesma forma que o bloco de tratamento de exceções dentro de um try. O bloco safe funcionaria garantindo a integridade e consistência do código da seguinte forma:

(I) Qualquer thread criada dentro do bloco safe seria sincronizada ao final do bloco, enquanto a thread pai, a thread que deu origem a thread que está dentro do bloco safe, será mantida parada e sua execução só será retomada ao final da execução do bloco, com isso exceções lançadas por uma thread não devem lançar exceções não tratadas em outra parte da execução. Foi optado por não sincronizar a criação das threads no começo do bloco pois isso não é uma pratica comum no desenvolvimento de aplicações, sendo algo não utilizado por programadores e ferindo o objetivo de simplicidade da implementação do tratamento de exceções.

(II) Ao inicializar o bloco safe, assim como no caso de um bloco try, são definidas as exceções que serão tratadas caso sejam lançadas exceções nas threads criadas. Caso sejam criadas threads pelas threads (threads aninhadas) que estão em um determinado bloco safe e caso estas threads acabem por lanaçr uma exceção, esta excepção seria propagada até este ponto e, caso houvesse um tratador para a exceção levantada, ela seria devidamente tratada.

(III) Exceções lançadas dentro de threads criadas dentro de outras threads, o caso das threads aninhadas, também são suportadas, bem como o caso de safes aninhados, e caso exceções sejam lançadas estas serão tratadas com o tratador de exceções do bloco safe imediatamente acima da thread, no caso de haver vários blocos safes que tratem a mesma exceção, o tratamento mais próximo será considerado para tratar a exceção.

(IV) Para dar suporte ao programador na hora de depurar o código, seria exibido no stack trace o caminho pelo qual o código passou dando destaque aos blocos safes.

(V) Assim como o bloco try, é permitido ao programador definir ações de limpeza fazendo uso do finally.

4. Uma Extensão para a Linguagem Java

A princípio, o mecanismo proposto foi de implementar como uma extensão de um dos compiladores existentes para a linguagem Java . Foi efetuada uma busca para encontrar um interpretador e compilador de Java onde fosse possível efetuar as alterações necessárias na gramática da linguagem, uma vez que para fazer as alterações necessárias era preciso modifica-la, para poder adicionar palavras chaves e se fazia necessário também ter acesso ao código antes dele ser compilado pela JVM. Como possibilidades foram levantadas o JDT do Eclipse [7] e o OpenJDK [8], ambos foram descartados pela dificuldade em realizar as modificações necessárias. Para a realização do projeto foi escolhido o JastAddJ [9] , que é um compilador e interpretador de Java para Java, escrito em Java e de fácil modificação. Apesar das complicações causadas em relação ao tempo de compilação da solução adotada, este foi mantida visando a facilidade para implementação da solução.

Foram realizadas modificações dentro da gramática sintatica contida no Jastadd através do Beaver[14] para adicionar uma nova declaração chamada “SafeStmt”, que é uma cópia exatada o “TryStmt”. Todas as diferenças entre as execuções entre eles será feito depois, com injeção de código.

Diferente das abordagens demostradas anteriormente, buscou-se tentar manter a sintaxe o mais próxima possível da original, buscando manter os objetivos de implementação de tratamento de exceções em mente, neste caso principalmente a uniformidade, simplicidade e legibilidade, modificando somente a palavra reservada **try** pela nova palavra reservada **safe**, para tentar eliminara a resistencia por parte dos desenvolvedores em aceitar trabalhar com uma solução muito diferente do que eles já estão acostumados a trabalhar. Ao invés de retornar as threads para o estado estável, como é o caso da solução do atomic box, optamos por eliminar a thread, aumentando assim o desempenho da solução. A estrutura do safe é a seguinte:

`safe``{`*bloco safe*`}` **`catch`**`(Exception e1){`*bloco catch*`}` `[`**`finally`**`{`*bloco finally*`}]`

O que estiver dentro do safe será executado normalmente, e caso não sejam criadas threads dentro deste bloco, então funcionará exatamente como um try. O que

estiver dentro do bloco catch será executado em caso ocorra a Exception e1, podendo esta exceção ser qualquer uma que herde da classe Exception e o bloco safe oferece suporte a multiplas exceções, ocorrendo um desvio de fluxo. A palavra reservada finally, assim como no caso de um bloco try, marca um bloco opcional, este bloco será sempre executado ao final do bloco safe ou do bloco catch. Ele pode ser usado para ações de limpeza uma vez que ele é executado mesmo depois de um bloco catch e mesmo que ocorra uma exceção que não foi prevista pelo programador. O diagrama de atividades em UML abaixo, feito utilizando o programa Astah Community [15], ilustra o funcionamento do mecanismo:

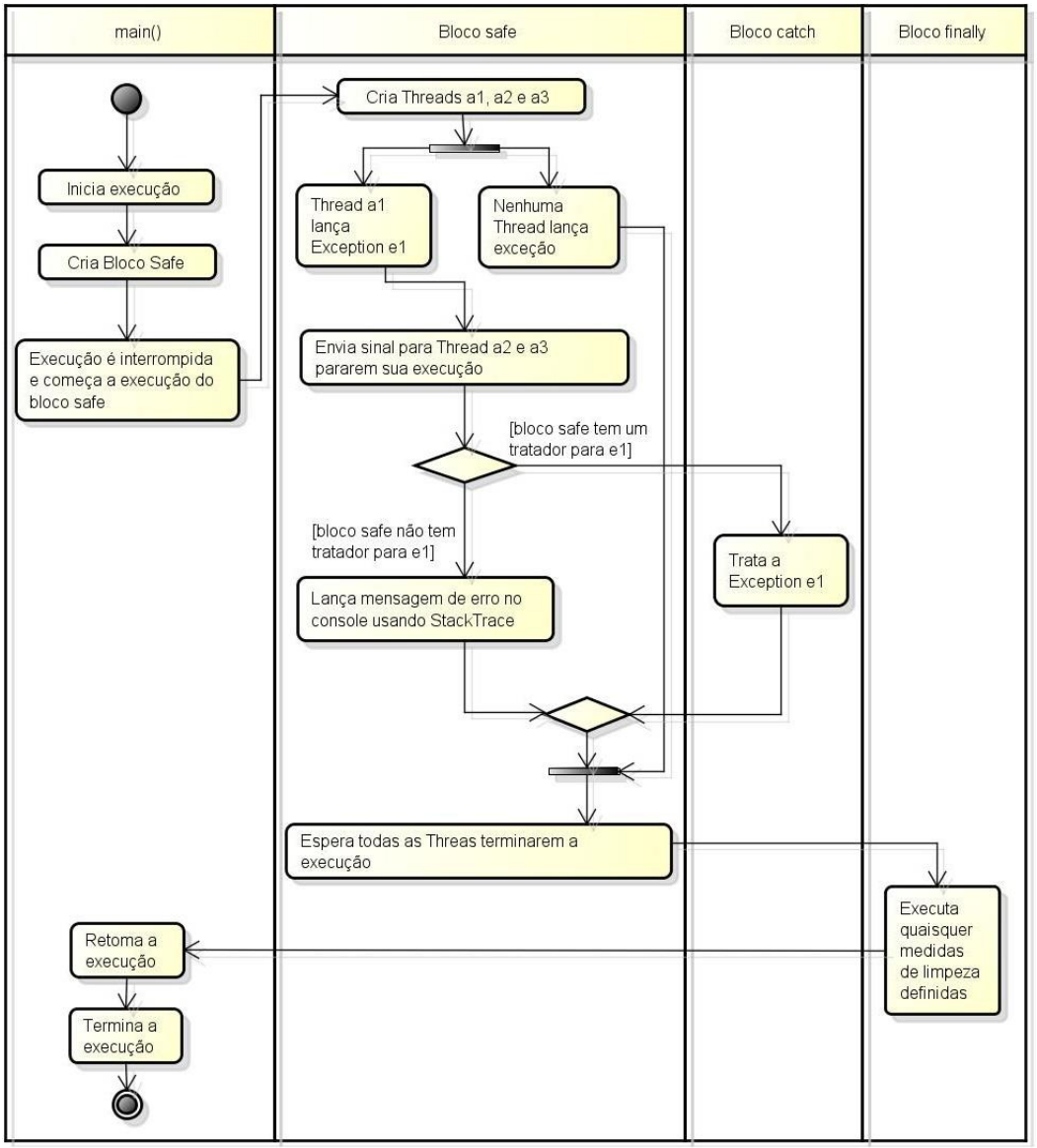

Figura 4 : Diagrama de Atividades ilustrando o funcionamento do mecanismo.

Para controlar o fluxo foram utilizadas duas classes novas. Elas tem como objetivo servir como base para as linhas que serão inseridas no código original e seus métodos serão invocados sempre que necessário efetuar algum tipo de verificação em tempo de execução. Essas classes são identificadas como SafeManager e o SafeNode, ambos buscam representar um grafo, onde o SafeManager funciona como um gerenciar dos nós e o SafeNode representa tanto um bloco safe no código quanto um dos nós do grafo. Atráves do grafo foi possível verificar a profundidade dos blocos safes, nos casos onde havia mais de um safe dentro da mesma linha de execução.

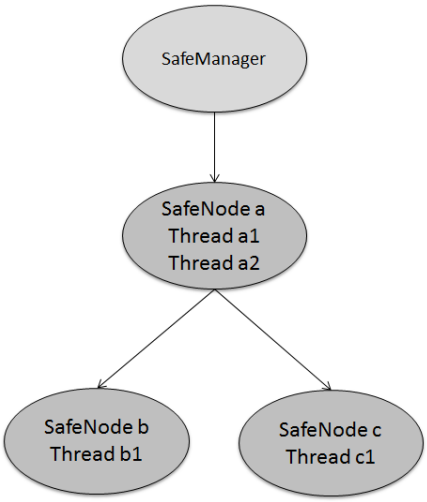

Figura 5: Ilustração do grafo SafeManager/SafeNode

Na fígura acima, temos a representação do SafeManager, e este está ligado ao "SafeNode a",onde duas threads foram inicializadas, a1 e a2 e este SafeNode tem ainda tem mais 2 SafeNodes ligados a ele, "SafeNode b", que inicializou um Thread b1 e "SafeNode c", que inicializou uma Thread c1, isto quer dizer que o "SafeNode a" é a raiz da estrutura e os acessos aos outros safes partirá dele. O SafeNode representa um bloco safe no código, então o fato dele estar vinculado ao SafeManager significa que existe um bloco safe e que cada uma das suas threads dentro do "SafeNode a" criou um novo bloco safe, ou até mesmo que uma delas, durante a sua execução, criou dois blocos safes. Ao utilizar a palavra reservada safe, o interpretador vai notar que foi criado um novo bloco safe e ira adiciona-lo ao SafeManager, ao faze-lo, ira verificar se a thread que está criando este novo bloco safe já não está contida em um bloco safe, direta ou indiretamente. Se a thread estiver, então é criado uma estrutura como no gráfico acima de safes aninhados, caso não esteja, então é adicionado ao mesmo nível da estrutura um

novo SafeNode, como no caso dos SafeNodes b e c. Seria ainda possível conter inumerais threads em cada um dos SafeNodes ou até mesmo um bloco safe onde não houvessem threads inicializadas dentro do mesmo, neste caso o bloco safe seria executado exatamente como se fosse um bloco try.

Para as interações com o JastAdd, foi criada uma classe chamada Recursos, esta classe tem como objetivo modularizar o código e unificar todas os trechos de código que foram utilizados pelas classes nativas do JastAdd em um único ponto, permitindo assim o fácil acesso ao código acrescentado. A classe Recursos contem somente métodos estáticos que são chamados por diversas outras classes do sistema, com a intenção de injetar o código necessário. Há também as classes SafeManagerShell e SafeNodeShell, porém estas diferem das classes concretras, SafeManager e SafeNode, explicadas acima. Estas duas classes contem os trechos de código necessários para injetar as classes concretas no código fonte do programa que será criado, utilizando as classes do JastAdd que representação o interpretador de java para poder criar as classes, juntamente com os seus atributos e métodos. Foi criado ainda uma classe nova no JastAdd para representar o bloco safe, chamada SafeStmt. Esta classe representa para o interpretador um bloco safe e é criada toda vez que a palavra reserva safe é utilizada, respeitado a sintaxe adequada. A única diferença entre o SafeStmt e o TryStmt, classe nativa do JastAdd, é que no método createBCode, método utilizado pelo JastAdd em todas as suas classes para gerar o código fonte que será enviado a JVM, a classe SafeStmt invoca métodos da classe Recursos para garantir o funcionamento do mecanismo. O diagrama de classes em UML, feito utilizando o Astah Community [15], a seguir representa como foram implementadas as 4 classes: Recursos, SafeManagerShell, SafeNodeShell e SafeStmt, desta última, todos os métodos e atributos foram omitidos com exceção do createBCode, uma vez que eles não sofrerema modificações e não interagem com as outras 3 classes.

Figura 6: Diagrama de classes das classes acrescentadas.

Pela solução adotada, foi realizada uma injeção de código, e este código seria inserido caso a palavra reservada safe fosse usada. A injeção de código troca o bloco safe por um bloco try e modifica certas estruturas para garantir o bom funcionamento do código, como adicionar um caso redundante em todas as execuções do método “run()” que forem executadas no código fonte. Esta redundância tem como objetivo desviar o fluxo. Caso a thread que esta executado o método run esteja contida dentro de um bloco safe, ou tenha sido inicializada por uma thread que esteja contida num bloco safe, ela executará o trecho de código modificado e seguro, caso não, então ela será executada

como foi escrita originalmente. Para mais detalhes sobre as modificações, conferir o apéndice.

A seguir, há uma descrição de como funcionam os métodos das classes SafeNode e SafeManager. O código fonte de ambas as classes encontra-se no apéndice para consulta.

A1 - Classe SafeNode: Os métodos na classe SafeNode, classe que representa um bloco safe no código fonte, tem as seguintes utilidades:

void addThread(Thread thread){...}: Adiciona a Thread do parâmetro a este nó do grafo.

void tryWakeUp(Thread thread){...}: Remove a Thread do parâmetro deste grafo e notifica a todas as threads que esperam por esse monitor que uma thread foi removida, caso não existam mais threads inacabadas neste grafo, então a thread ira acordar, junto com todas as outras threads, simbolizando que aquele bloco safe chegou ao final e o fluxo pode continuar normalmente.

void addException(Exception excep){...}: Adiciona as exceções que devem ser tratadas por este safe ao nó.

void syncUp(){...}: É invocado ao final da execução, para que a thread que criou o bloco espere que todas as threads que estejam no dentro do bloco safe terminem a sua execução antes dela poder continuar.

void breakUp(){...}: Caso uma exceção seja lançada durante a execução de uma das threads que pertence a este nó então, todas as threads devem ter a sua execução interrompida, como em Java não há como terminar a execução de uma thread no meio dela, é invocado interrupt de cada uma das threads que pertencem ao bloco safe.

boolean existsThread(Thread thread){...}: Verifica se a thread passada no parâmetro está presente neste grafo.

boolean hasException(){...}: Verifica se o bloco safe foi inicializado com alguma exceção.

A2 – Classe SafeManager: Os métodos na classe SafeManager, classe responsável por gerenciar os nós do grafo gerado pela criação de blocos safes aninhados, tem as seguintes utilidades:

void addSafe(SafeNode newChild,SafeNode parent){…}: Utilizando esse método é possível adicionar um novo nó na árvore, isso ocorre na criação de um bloco safe, onde também é criado um SafeNode para representa-lo. É necessário saber quem é o bloco safe pai daquele safe, contudo caso não exista, o bloco safe inserido será considerado a nova raiz da árvore.

SafeNode getRoot(){…}: Retorna a raiz da árvore.

DefaultTreeModel getTree(){…}: Retorna a árvore com todos os seus nós.

void removeSafe(SafeNode safeNode){…}: Remove o SafeNode passado como parâmetro da árvore.

boolean isSafe(Thread thread){…}: Tem como objetivo verificar se a thread passada no parâmetro está contida em um bloco safe, caso esteja então a execução do run() será tratada em caso de exceção, em caso contrário então o método deverá ter sua execução inalterada.

SafeNode getSafe(SafeNode originalNode,Thread thread){…}: Retorna o SafeNode que contenha o parâmetro passado. Com o SafeNode referente a thread, é possível saber que tratamento as exceções deverão receber

Para a modificação do código e para assegurar que todas as threads fossem devidamente tratadas, foram modificadas certas partes do código original e diversas linhas de código foram inseridas para modificar o fluxo da programação. Essa modificação nas linhas de código se fez necessária uma vez que não era desejável modificar a JVM e ainda assim era necessário modificar o fluxo de controle do programa. Para consultar exatamente quais são as modificações contidas no código, verificar o apéndice.

5. Trabalhos Relacionados

Há exemplos de novos mecanismos de tratamento de exceções que buscam sanar os problemas das abordagens clássicas, como o o atomic box [4] ou o failbox [5] ou até mesmo a abordagem utilizada pela Microsoft na linguagem C# [9] mas em Java ainda há uma ausência de qualquer tipo de solução apresentada buscando solucionar este problema.

Temos no failbox uma tentativa de assegurar que problemas decorrentes da execução de threads não sai do bloco protegido, neste caso chamado de Failbox. O programador criaria um bloco onde caso não houvesse nenhum problema, o código seria executado normalmente mas caso alguma exceção fosse lançada, então todos os trechos de código que passasem por esta área do código seriam finalizados, eliminando assim falhas na segurança de dependência. Para que o mecanismo funciona, é necessário que o programador garanta que uma operação B depende de uma operação A e então ambas são executadas num mesmo failbox.

Quanto ao atomic box nos verificamos que é uma abordagem já mais madura do que o Failbox e com uma proposta de substituição completa, podendo rodar até duas vezes mais rápido do que um failbox. A idéia é uma extensão de java para cordenação de exceções levantadas durante o uso de threads. Com uma sintaxe muito mais agradável, funcionando parecido com um bloco try, usando a palavra reservada abox, o atomic box faz com que as threads que lançarem exceção sejam devolvidas a seu estado estável para sua utilização posterior.

O resultado de ambas as abordagens agrega muito a literatura mas verifica-se na prática que não trazem grandes avanços para os desenvolvedores. Buscando uma solução que balanceie os objetivos do tratamento de exceções, principalmente a simplicidade, uniformidade e legibilidade, mantendo a sintaxe o mais próxima possível da atual de Java é fundamental para o bom uso de qualquer mecanismo para solucionar o problema de exceções concorrentes em Java.

6. Trabalhos Futuros e Conclusão

Apesar dos avanços realizados em relação ao tratamento de exceção concorrente em Java ainda há pontos que podem ser melhorados no mecanismo desenvolvido.

Threads que forem criadas fora do source do projeto não recebem tratamento. Uma vez que os tratamentos são feitos com base em injeção de código, sem acesso ao código fonte, não há como ser feita a modicação necessária e portanto não há como assegurar o bom funcionamento. A maneira mais eficiente de garantir isso seria uma modificação diretamente na JVM.

A implementação do suporte ao *stack trace*, está parte ainda estava sob uma análise de qual seria a melhor maneira de lidar com o problema e encontra-se ainda na fase de análise de viabilidade. Apesar de extremamente importante, há necessidade da experimentação com desenvolvedores para poder utilizar a maneira mais adequada de exibir o *stack trace*. Atualmente é utilizada a solução padrão para exbir erros, que provavelmente não é a melhor opção.

O mecanismo funcional e dentro do esperado foi realizado. Threads criadas dentro de um bloco safe, são seguras e sincronizadas ao final do bloco graças a mecanismos que garatem a segurança da sua execução. Mas detalhes sobre as modificações presentes no código são encontradas no apêndice.

O desenvolvimento do mecanismo, utilizando uma estrutura de grafos para cordenação dos safes e o tratamento das exceções concorrentes foi realizada com sucesso entretanto o desenvolvimento incremental dependia da utilização por parte dos desenvolvedores na construção de projetos.

O problema de tratamento de exceções concorrentes em linguagens com paradigma orientado a objeto é um problema real e há diversas tentativas de solucionar este problema como [4], [9] e [11] mas elas falham em priorizar os objetivos de simplicidade e a uniformidade e por conta disto se tornam não práticos de utilizar. A solução apresentada vem como uma forma de mostrar que é possível solucionar parte dos problemas de uma forma simples e de forma uniforme com o tratamento de exceções já utilizado em Java.

Apesar da utilidade funcional da solução proposta, provavelmente a solução no seu formato atual não teria um desempenho bom o suficiente para ser utilizada no desenvolvimento de sofwares por profissionais mas esperasse que com os resultados alcançados o desenvolvimento de Java venha a notar uma necessidade de solucionar este problema e incorporar, dentro da JVM, código que consiga tratar exceções concorrentes, mantendo ainda em mente a simplicidade e uniformidade e priorizando também o desempenho, objetivo que só é possível de ser conseguido com o uso de uma solução nátiva.

Apêndice 1: Código fonte das Classes SafeNode e SafeManager

```
class SafeNode extends DefaultMutableTreeNode {
    private ArrayList<Thread> threads;
    private ArrayList<Exception> catchExceptions;
    private AtomicInteger index = new AtomicInteger(0);
    public SafeNode() {
        super();
        this.threads = new ArrayList<Thread>();
    }
    public synchronized void addThread(Thread thread) {
        this.threads.add(thread);
        this.index.incrementAndGet();
    }
    public synchronized void tryWakeUp(Thread thread) {
        this.threads.remove(thread);
        this.index.decrementAndGet();
        notifyAll();
    }
    private synchronized ArrayList<Exception> getExceptions(){
        if(this.catchExceptions == null)
            this.catchExceptions = new ArrayList();
        return catchExceptions;
    }
    public synchronized void addException(Exception excep) {
        this.getExceptions().add(excep);
    }
    public synchronized boolean existsException(Exception excep) {
        return this.getExceptions().contains(excep);
    }
    public synchronized void syncUp() {
        while(index.get() > 0){
            try {
                wait();
            } catch (InterruptedException e) {
            e.printStackTrace();
            }
        }
    }
    public synchronized void breakUp() {
        for(int i = 0; i < threads.size(); i++) {
            threads.get(i).interrupt();
        }
    }
    public synchronized boolean existsThread(Thread thread) {
        if(threads.contains(thread))
            return true;
        return false;
    }
}
```

Figura 7: Classe SafeNode

```
class SafeManager {
    private static SafeManager instance;
    private static DefaultTreeModel tree;
    private SafeManager() {
        super();
        tree = new DefaultTreeModel(null, false);
    }
    public static SafeManager getInstance() {
        synchronized(SafeManager.class) {
            if(instance == null)
                instance = new SafeManager();
        } return instance;
    }
    public synchronized void addSafe(SafeNode newChild, SafeNode parent) {
        if(tree.getRoot() == null) {
            tree.setRoot(newChild);
        } else {
            tree.insertNodeInto(newChild, (MutableTreeNode)parent, 0);
        }
    }
    public synchronized SafeNode getRoot() { return (SafeNode)tree.getRoot();}
    public synchronized DefaultTreeModel getTree() { return tree; }
    public synchronized void removeSafe(SafeNode safeNode) {
        if(tree.getRoot() != null) {
            tree.removeNodeFromParent(safeNode);
        } else {
            if(tree.getRoot().equals(safeNode))
                tree.setRoot(null);
        }
    }
    public synchronized boolean isSafe(Thread thread) {
        if(getSafe((SafeNode)tree.getRoot(), thread) != null)
            return true;
        return false;
    }
    public SafeNode getSafe(SafeNode originalNode, Thread thread) {
        if(originalNode == null) { return null;}
        else if(originalNode.existsThread(thread)) {
            return (SafeNode)originalNode;
        }
        SafeNode node = null;
        if(!originalNode.isLeaf()){
            node = (SafeNode)tree.getChild(originalNode, 0);
            for(int i = 0; i < tree.getChildCount(originalNode); i++) {
                node = (SafeNode)tree.getChild(originalNode, i);
                if(node.existsThread(thread)) { return node; }
            }
        }
        return getSafe(node, thread);
    }
}
```

Figura 8: Classe SafeManager.

Apêndice 2: Modificações no código fonte.

Segue um exemplo de como se dá a modificação do código buscando efetuar as mudanças necessárias para garantir que o bloco safe funcione. Todas as modificações no código fazem uso das classes explicadas anteriormente. O primeiro grupo de imagens equivale as classes sem alterações enquanto o segundo grupo equivale as classes já alteradas pelo funcionamento do mecanismo.

```
public class NewTest {
    public NewTest() {
        safe{
            new Thread(new TestRun2()).start();
            Thread thread2 = new Thread(new TestRun());
            thread2.start();
        } catch (Exception e) {
            System.out.println("Thread Interrompida");
        }
    System.out.println("teste");
    }

    public static void main(String args[]) {
        new NewTest();
    }
}
```

Figura 9. Classe NewTest antes de ser modificada.

```
class TestRun implements Runnable{
    public void run() {
        String s = "run";
        System.out.println(s);
    }
}
class TestRun2 implements Runnable{
    public void run() {
        String s = "run2";
        System.out.println(s);
    }
}
```

Figura 10: Classes TestRun e TesteRun2 antes de serem modificadas.

```
public class NewTest {
  public NewTest() {
    super();
    SafeNode safeNode = new SafeNode();
    SafeManager.getInstance().addSafe(safeNode, SafeManager.getInstance().getSafe(
        SafeManager.getInstance().getRoot(), Thread.currentThread()));
    try {
      Thread variavelNova0 = new Thread(new TestRun2());
    if(SafeManager.getInstance().isSafe(Thread.currentThread()))
        SafeManager.getInstance().getSafe(SafeManager.getInstance().getRoot(),
            Thread.currentThread()).addThread(variavelNova0);

    safeNode.addThread(variavelNova0);
      variavelNova0.start();
      Thread thread2 = new Thread(new TestRun());
     if(SafeManager.getInstance().isSafe(Thread.currentThread()))
        SafeManager.getInstance().getSafe(SafeManager.getInstance().getRoot(),
            Thread.currentThread()).addThread(thread2);

     thread2.start();
      safeNode.addThread(thread2);
      try {
        variavelNova0.join();
      }
      catch (InterruptedException e) {
        System.out.println("Thread Interrompida");
      }
      try {
        thread2.join();
      }
      catch (InterruptedException e) {
        System.out.println("Thread Interrompida");
      }
    }
    catch (Exception e) {
      System.out.println("Thread Interrompida");
    }
    System.out.println("teste");
  }
  public static void main(String[] args) {
    new NewTest();
  }
}
```

Figura 11: Classe NewTest depois de ser modificada.

```
class TestRun implements Runnable {
  public void run() {
    if(SafeManager.getInstance().isSafe(Thread.currentThread())) {
      try {
        String s = "run";
        System.out.println(s);
      }
      catch (Exception e) {
        SafeManager.getInstance().getSafe(SafeManager.getInstance().getRoot(),
        Thread.currentThread()).breakUp();
        System.out.println("Thread Interrompida");
      }
      finally {
        SafeManager.getInstance().getSafe(SafeManager.getInstance().getRoot(),
        Thread.currentThread()).tryWakeUp(Thread.currentThread());
      }
    }
    else {
        String s = "run";
        System.out.println(s);
    }
  }
}
```

Figura 12: Classe TestRun depois de ser modificada.

Referências

[1] James Gosling. The java language environment. Section 1.2: Design Goals of Java. Technical report, Sun Microsystems, 1996. Adress: http://java.sun.com/docs/white/langenv/Intro.doc2.html. Last visit: July 1st 2012

[2] Avelino Zorzo, Brian Randell, and Alexander Romanovsky. Tla specification of a mechanism for concurrent exception handling. In Concurrency in Dependable Computing, chapter 3, pages 41–59. Kluwer Academic Plublishers, 2002.

[3] Fernando Castor, Alexander Romanovsky, and Cecília M. F. Rubira. Improving reliability of cooperative concurrent systems with exception flow analysis. Journal of Systems and Software, 82(5):874–890, 2009.

[4] Derin Harmanci, Vincent Gramoli and Pascal Felber. Atomic Boxes: Coordinated Exception Handling with Transactional Memory. Adress: http://lpd.epfl.ch/gramoli/doc/pubs/ECOOP2011-preprint.pdf. Last visit: July 1st 2012

[5] Jie Xu, Brian Randell, Alexander Romanovsky. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. Adress: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.3758&rep=rep1&type=pdf Last visit: July 1st 2012

[5] Bart Jacobs and Frank Piessens. Failboxes: Provably Safe Exception Handling. Adress: https://lirias.kuleuven.be/bitstream/123456789/221909/2/failboxes.pdf Last visit: July 1st 2012

[6] Eclipse Java development tools (JDT). Adress: http://www.eclipse.org/jdt/ Last visit: July 1st 2012

[7] OpenJDK. Address: http://openjdk.java.net/. Last visit: July 1st 2012

[8] JastAddJ: The JastAdd Extensible Java Compiler. Adress: http://jastadd.org/web/jastaddj/ Last visit: July 1st 2012

[9] Parallel Tasks. Adress: http://msdn.microsoft.com/en-us/library/FF963549.aspx Last visit: July 1st 2012

[10] Nélio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. Ejflow: taming exceptional control flows in aspect-oriented programming. In Proceedings of the 7th ACM Conference on Aspect-Oriented Software Development, pages 72–83, Brussels, Belgium, March 2008.

[11] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In Proceedings of the 23rd European Conference on Object-Oriented Programming, volume 5653 of LNCS, pages 470–494, Genoa, Italy, 2009. Springer.

[12] Brian Goetz. Java theory and practice: The exceptions debate, May 2004. Address: http://www.ibm.com/developerworks/java/library/j-jtp05254.html. Last visit: July 1st 2012.

[13] Alessandro F. Garcia , Cecília M. F. Rubira , Alexander Romanovsky , Jie Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software, 2001. Address: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.251. Last visit: 10 September 2013.

[14] Beaver - a LALR Parser Generator. Address: http://beaver.sourceforge.net/spec.html. Last visit: 10 September 2013.

[15] Astah Community. Address: http://astah.net/editions/community. Last visit: 10 September 2013