Tratamento de Exceções Orientado a Contextos

Fabiane Cristine Dillenburg, Jorge Luis Victória Barbosa
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA)
Universidade do Vale do Rio dos Sinos (UNISINOS)
93.022-000 – São Leopoldo – RS – Brasil
fdillenburg@turing.unisinos.br, jbarbosa@unisinos.br

Resumo

A popularização de dispositivos computacionais cada vez menores e com maior poder computacional tem tornado a computação móvel mais comum na vida cotidiana. O desenvolvimento de novas aplicações exige uma adaptação das linguagens de programação à nova realidade, uma vez que estas precisam de mecanismos que tirem proveito das novas tecnologias. Neste contexto, o presente trabalho propõe mecanismos para tratamento de exceções orientado a contextos, com foco no desenvolvimento de aplicações móveis e ubíquas. O conjunto destes mecanismos foi implementado sobre a plataforma atual de execução do Holoparadigma e validado pela implementação de aplicações móveis e ubíquas.

1. Introdução

A popularização de dispositivos computacionais cada vez menores e com maior poder computacional tem tornado a computação móvel mais comum na vida cotidiana. Neste contexto, inúmeros cenários de computação móvel e ubíqua [19, 16] são usados na literatura para exemplificar possíveis aplicações e evidenciar a sua complexidade [12, 7].

O desenvolvimento dessas aplicações exige uma adaptação das linguagens de programação à nova realidade. Os projetistas devem ter consciência de que a computação móvel e ubíqua constitui um novo paradigma computacional centrado no usuário e suas tarefas, com a computação inserida no ambiente [20]. Precisam, então, desenvolver novos mecanismos que tirem proveito das novas tecnologias.

Neste contexto, a principal contribuição científica deste trabalho é um modelo de mecanismos de tratamento de exceções [10], com foco em aplicações móveis e ubíquas. Este modelo, denominado HoloException, segue os conceitos do Holoparadigma [3] e utiliza um *blackboard* para gerenciamento de tratadores de exceções. Esta característica é importante, porque possibilita a definição de tratadores de uma forma bastante dinâmica (em tempo de execução) e de acordo com os diferentes contextos pelos quais um ente (unidade de modelagem do paradigma) pode mover-se. Assim, mostra-se apropriado para as aplicações móveis e ubíquas, uma vez que estas são caracterizadas pela mobilidade de seus elementos e também pela manipulação de diferentes contextos. Destacam-se ainda como outras contribuições: a identificação de possíveis exceções em aplicações móveis e ubíquas, e a implementação dos mecanismos propostos na atual plataforma de execução do Holoparadigma.

O artigo está organizado da seguinte forma: a Seção 2 aborda de forma resumida o contexto científico e tecnológico no qual foi desenvolvido o trabalho. A Seção 3 apresenta o Holoparadigma e a Hololinguagem. As seções 4 e 5, por sua vez, apresentam o modelo de tratamento de exceções proposto e uma instância de uso do mesmo. A Seção 6 descreve trabalhos relacionados. Por fim, a sétima seção encerra o artigo com considerações finais e perspectivas de trabalhos futuros.

2. Computação Móvel, Computação Ubíqua e Tratamento de Exceções

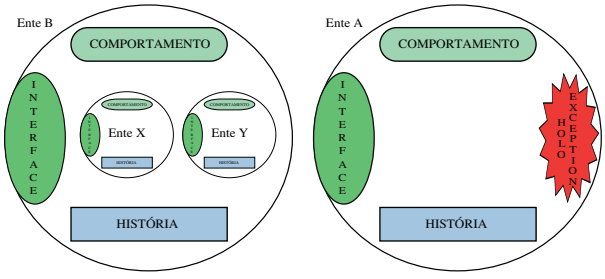
A primeira geração de sistemas de computação ubíqua era voltada para a criação de ambientes integrados [20]. Neste contexto, diversos ambientes para o desenvolvimento de aplicações móveis e ubíquas foram propostos, entre eles Aura [9], One.World [11], Gaia [15] e ubiHolo [2]. Estes ambientes exploram em maior ou menor grau as características da computação móvel e ubíqua, e apresentam idéias inovadoras relacionadas ao usuário, à representação de contexto, a dispositivos heterogêneos e à mobilidade.

No desenvolvimento de aplicações que utilizam esses ambientes, percebe-se que, até o momento, a maior parte das aplicações utiliza apenas o mecanismo de exceções fornecido pelas linguagens de programação subjacentes [8]. Entretanto, as abstrações e mecanismos convencionais não são satisfatórios por vários motivos [5]: (i) a propagação

de exceções deve considerar mudanças contextuais que ocorrem constantemente nas aplicações móveis; (ii) as atividades de recuperação de erros e a estratégia de tratamento de exceções devem ser selecionadas de acordo com informações de contexto, podendo ser necessário ativar diferentes tratadores para uma mesma exceção de acordo com o contexto; e (iii) a própria caracterização de uma exceção pode depender do contexto dos dispositivos, uma vez que um estado do sistema pode ser considerado errôneo em uma dada localização onde o dispositivo se encontra, mas não em outra. Neste contexto, é proposto o HoloException.

3. Holoparadigma e Hololinguagem

O Holoparadigma é um modelo multiparadigma que possui uma semântica simples e distribuída. Através dessa semântica, o modelo estimula a exploração automática da distribuição (distribuição implícita) e estabelece a utilização do **ente** como unidade de modelagem. O Holoparadigma distingue os entes, de acordo com a sua estrutura, em dois tipos: (i) **ente elementar** (Ente X e Ente Y na Figura 1(a)) e (ii) **ente composto** (Ente B na Figura 1(a)). Um ente elementar é organizado em três partes: interface, comportamento e história. A **interface** descreve suas possíveis relações com os demais entes. O **comportamento** contém ações que implementam a funcionalidade de um ente. Holo não estabelece os tipos de ações a serem utilizadas, apenas estabelece que existem dois tipos básicos de comportamento: imperativo e lógico. A **história** é um espaço de armazenamento compartilhado no interior de um ente. Um ente composto possui a mesma organização de um ente elementar; no entanto, suporta a existência de outros entes na sua composição (entes componentes). Cada ente possui uma história. A história fica encapsulada no ente e, no caso dos entes compostos, é compartilhada pelos entes componentes. Os entes componentes participam do desenvolvimento da história compartilhada e sofrem os reflexos das mudanças históricas. Sendo assim, podem existir vários níveis de encapsulamento da história. Os entes acessam somente a história um nível acima.



(a) Ente composto (b) Ente com HoloException

Figura 1. Organização dos entes

O Holoparadigma distingue ainda os entes entre estáticos e dinâmicos. A única distinção entre eles é a função de ambos. Os **estáticos** são utilizados como matrizes estáticas para criação de outros entes. Além disso, estabelecem um estado inicial para execução de programas. Os **dinâmicos**, por sua vez, representam o estado corrente de uma execução. Um programa em execução é composto por entes dinâmicos. Esses entes executam ações e interagem de acordo com seus comportamentos e histórias. A criação de entes nos domínios estático e dinâmico é realizada por meio de clonagem. A **clonagem** é a criação de um ente tendo como matriz outro(s) ente(s).

No Holoparadigma, o modelo de coordenação utilizado é o **repositório** [17]. Este modelo apresenta dois tipos de componentes: uma estrutura de dados central que representa o estado corrente e uma coleção de componentes independentes que operam no armazenamento central. Nesse caso, os componentes são entes e o repositório é a história. Como o estado atual do repositório é a principal fonte de controle dos componentes, este repositório é um *blackboard* [18].

A Hololinguagem permite a especificação de aplicações através dos conceitos do Holoparadigma. Os aspectos gerais desta linguagem de programação foram apresentados à comunidade acadêmica em [1], e a exploração de paralelismo e distribuição foi analisada em [4]. Um programa em Holo é formado por descrições de entes (entes estáticos). Um programa sempre possui um ente estático denominado `holo`. O ciclo existencial deste ente determina a dinâmica dos programas. Sendo assim, um programa em Holo é executado em três fases: (i) fase de criação – um programa sempre inicia com uma clonagem de transição automática pertencente ao ente estático `holo`. Este ente deve estar descrito no domínio estático. Cria-se assim um ente dinâmico que recebe o nome `d_holo` (*dynamic* holo); (ii) fase da existência – o ente `d_holo` executa sua ação-guia. A execução desta ação estabelece a dinâmica de execução do programa; e (iii) fase de extinção – um programa termina quando o ente `d_holo` é extinto.

A Hololinguagem possui ações imperativas pré-definidas que realizam atividades básicas. Dentre as ações imperativas pré-definidas na Hololinguagem, destacam-se `move` (implementa a mobilidade lógica de entes) e `clone` (permite a clonagem de entes).

4. Modelo HoloException

No modelo proposto, cada exceção é representada por um ente e os tratadores de exceção são definidos por tuplas de um *blackboard*, utilizando assim o modelo de coordenação do paradigma. O Holoparadigma suporta a solução de problemas que necessitem de composição dinâmica de contextos (conhecida também como “suporte

a grupos"). Cada ente representa um contexto e, assim, os entes compostos representam contextos compostos. A associação de tratadores e o modelo de propagação são definidos de forma que os mecanismos de tratamento de exceções propostos utilizem a composição de entes e mantenham a característica de sensibilidade ao contexto.

As próximas subseções descrevem as decisões de projeto envolvidas na definição dos mecanismos de tratamento de exceções propostos. Estas decisões foram tomadas a partir da análise de uma taxonomia proposta por [8] que identifica questões comuns a projetos de mecanismos e classifica diferentes soluções de projeto durante a construção de mecanismos de tratamento de exceções.

4.1. Representação da Exceção

HoloException representa exceções como entes (objetos de dados [8]), que constituem uma abstração apropriada para a Hololinguagem. No contexto de orientação a objetos, a representação equivalente seria uma classe. Esta representação é mais adequada para este modelo por preservar a modularidade do programa e manter informações de contexto. A possibilidade de manipular informações de contexto agrega valor, uma vez que o programador pode fazer uso das mesmas durante o tratamento da exceção.

Quando uma exceção é levantada no HoloException, ocorre uma operação de clonagem. Neste caso, o ente matriz é o ente que define a exceção. Vale notar que as exceções definidas pelo usuário são entes estáticos, que diferem dos demais pela palavra "*exception*" que aparece em sua descrição. Se no comportamento pertencente ao ente clonado existir uma ação-guia definida, a mesma é executada automaticamente. A exceção tem sua execução controlada por esta ação e nela o programador poderá manipular as informações de contexto da exceção. Terminada a execução do comportamento, HoloException buscará pelo tratador da exceção levantada. O ente que representa a exceção será extinguido quando a execução do tratador terminar.

Na Figura 2, é definida a exceção que tem por objetivo indicar que um determinado número não é um número natural. Esta exceção poderia ser levantada, por exemplo, em um teste de valores numéricos. Nas linhas 2 a 4, é descrita a ação-guia que faz a manipulação de uma informação de contexto. Neste caso, escreve o número passado por parâmetro na história de seu pai (o ente que levantou a exceção). A informação é passada por parâmetro no momento da clonagem, que ocorre quando a exceção é levantada.

As exceções são levantadas de forma explícita com o comando `raise`. Este comando recebe como parâmetro o nome da exceção e os argumentos que serão usados na clonagem da exceção quando a mesma for levantada. O nome da exceção pode ser pré-definido ou pode ser

```
1 naoEhNatural exception () {
2   naoEhNatural (Numero) {
3     out(history)!numero(Numero);
4   }
5 }
```

Figura 2. Exemplo de definição de exceção

```
1 holoexception {
2   <nome_excecao>(<ativ_trata>, <acoes_associadas>),
3   <nome_excecao>(<ativ_trata>, *)
4 }
```

Figura 3. Definição estática de um tratador

um nome de uma exceção declarada pelo programador (exceção de usuário). A sintaxe desse comando é dada por: `raise <nome_excecao>(<arg1>, ..., <argN>);`.

4.2. Associação e Definição de Tratadores

A associação de tratadores pode ser feita em nível de (i) ente ou (ii) ação. A possibilidade de associar tratadores a múltiplos níveis permite a definição de contextos diferentes para a manipulação de exceção em uma única aplicação. Esta característica é especialmente útil para os sistemas que têm regiões críticas que requerem um nível mais elevado de tolerância a falhas [6].

O gerenciamento dos tratadores de exceção é feito com o *blackboard* holoexception. Em função desta forma de gerenciar tratadores, foi necessária a modificação da estrutura organizacional de um ente: agregou-se um novo *blackboard*, denominado holoexception (Figura 1(b)). O *blackboard* separa a atividade normal da atividade anormal, uma vez que cada tupla de holoexception representa um tratador de exceção. O nome da tupla é igual ao nome da exceção que o tratador definido pela tupla irá tratar. O primeiro elemento representa a atividade de tratamento da exceção. Este elemento pode indicar uma ação que será executada pelo tratador. A ação pode ser pré-definida, definida no comportamento do ente ou pode ser acessada através da interface de outro ente. De forma alternativa, este elemento pode indicar um ente. Neste caso, quando o tratador é acionado, o ente é clonado e a sua ação-guia é executada. Os demais elementos da tupla definem as ações associadas ao tratador. Quando o segundo elemento for igual a um asterisco (*), o tratador definido pela tupla está associado ao ente.

A Figura 3 mostra a sintaxe de definição de tratadores. A linha 1 indica o início do bloco de inicialização do *blackboard* holoexception. Na linha 2, tem-se a primeira tupla (ou primeiro tratador) de holoexception. A tupla é formada pelo nome da exceção (<*nome_excecao*>) e seus elementos, que são a atividade de tratamento (<*ativ_trata*>) e o nome das ações associadas ao tratador (<*acoes_associadas*>). A atividade de tratamento pode indicar o nome de um ente ou o

nome de uma ação. Na linha 3, tem-se a segunda tupla de holoexception. Esta difere da primeira por definir um tratador associado ao ente (*). A linha 4 indica o fechamento do bloco de inicialização do *blackboard*.

A definição da atividade de tratamento da exceção por meio de uma ação requer algumas considerações. O comportamento de um ente é dinâmico (pode ser alterado em tempo de execução). Logo, uma ação pode ser alterada ou removida durante a execução. Assim, o tratador definido no *blackboard* pode perder a semântica original. Em função disso, o ciclo de vida do tratador é unificado o com o ciclo de vida da ação. Neste caso, se uma ação for removida, o tratador também será removido. Portanto, alterar uma ação significa perder os tratadores de suas exceções. Caso os tratadores continuem sendo necessários após a alteração da ação, o programador precisa reinserir os mesmos. É importante destacar que a alteração de uma ação é feita em dois passos: primeiro, a ação é removida do comportamento e, em seguida, ela é inserida novamente com código-fonte alterado.

Ainda no contexto de uma ação como atividade de tratamento, tem-se outra questão importante: o que acontece quando a ação pertence à interface de outro ente e este é extinto durante a execução da ação? Quando esta situação ocorrer, será sinalizada a exceção pré-definida *BeingExtinguished*, e na seqüência será realizado o tratamento desta nova exceção. Nota-se assim uma característica que merece consideração: durante a própria atividade de tratamento podem acontecer novas exceções que também serão tratadas.

No *blackboard* holoexception de um ente, afirmações e perguntas apresentam a seguinte semântica: **afirmação** – insere um tratador. Será permitida apenas a inserção de uma tupla associada a cada ação para uma exceção. A tupla que define um tratador pré-existente só será substituída pelo novo tratador se todos os elementos da tupla forem iguais; **pergunta bloqueante não-destrutiva** – retorna sua resposta. Se o tratador requisitado não existir, então o fluxo de execução aguarda pela inserção do mesmo; **pergunta bloqueante destrutiva** – retorna sua resposta e remove o tratador em questão. Se o tratador requisitado não existir, então o fluxo de execução aguarda pela inserção do mesmo; **pergunta não-bloqueante não-destrutiva** – retorna sua resposta. Se o tratador requisitado não existir, é retornada uma falha; e **pergunta não-bloqueante destrutiva** – retorna sua resposta e remove o tratador em questão. Se o tratador requisitado não existir, é retornada uma falha.

A semântica de operação do *blackboard* destaca uma característica interessante do modelo: tratadores de exceção podem ser definidos em tempo de execução. Na descrição estática de um ente, o *blackboard* holoexception pode ser inicializado (conforme visto na Figura 3), mas isso não é obrigatório. Em função disso, tem-se tratadores essencialmente dinâmicos e que não podem ser verificados em tempo de compilação. Na Figura 4, tem-se a sintaxe de manipulação do *blackboard* em tempo de execução.

```
1 holoexception!<nome_excecao>(<ativ_trata>, <acoes_associadas>);
2 holoexception!<nome_excecao>(<ativ_trata>, *);
```

Figura 4. Definição dinâmica de um tratador

4.3. Ligação de Tratadores

HoloException utiliza ligação dinâmica de tratadores, uma vez que os tratadores são definidos por tuplas de um *blackboard* que têm semântica de operação bastante dinâmica. Vale ressaltar que as operações realizadas sobre o *blackboard* holoexception podem inclusive substituir tratadores definidos na inicialização feita na descrição estática de um ente.

Em função da ligação dinâmica, HoloException define em tempo de execução o tratador que será executado. Nota-se então que o tratador não pode ser determinado em tempo de compilação. Assim, torna-se desnecessária a verificação de existência de todos os tratadores na compilação. Dado o gerencimento dos tratadores por meio de um *blackboard*, não é possível definir estaticamente a ligação de um tratador com as ocorrências de exceção, nem mesmo limitadas ao sinalizador. Esta ligação depende do fluxo de execução e necessita da busca de tratadores em tempo de execução. A desvantagem da ligação dinâmica está na perda de legibilidade.

4.4. Propagação de Exceção

HoloException adota o modelo de propagação híbrido. Esta solução também é adotada por uma série de linguagens de programação, entre as quais Java e C++. Este modelo combina a propagação explícita e a propagação automática. O modelo tenta encontrar um tratador para a exceção no *blackboard* do nível (contexto) em que a exceção foi levantada. Em um contexto, os tratadores são procurados de acordo com o nome da exceção levantada. Se o tratador for encontrado, a atividade de tratamento pode fazer a propagação explícita da exceção (usando o comando `raise`). Se o tratador específico não for encontrado, a exceção pré-definida "*UncaughtException*" é levantada e propagada automaticamente.

A propagação explícita segue a hierarquia dos entes. Em outras palavras, quando a exceção é propagada em um tratador associado a uma ação, ela será propagada para o ente. Quando a exceção é propagada em um tratador associado ao ente, ela será propagada para seu pai. Desta forma, os mecanismos de tratamento de exceções utilizam a composição de entes e mantém a sensibilidade ao contexto. A propagação

automática, por sua vez, é usada para propagar a exceção que indica o não tratamento da exceção original (*UncaughtException*). Esta propagação utiliza a pilha de ativação, pois todo o fluxo de execução pode ser afetado e precisa ser avisado da exceção.

O modelo não quebra a modularidade do programa, uma vez que não é repassada uma exceção para um nível em que ela pode ser invisível. De outra forma, pode-se dizer que as exceções não são propagadas automaticamente para um nível em que talvez não sejam tratadas. A abordagem de HoloException apenas repassa uma exceção genérica, que mantém como parâmetro a exceção levantada originalmente. Conhecer a exceção original pode ser importante, inclusive para exibir ao usuário em uma última instância, por exemplo, quando o programa é encerrado.

HoloException não utiliza a abordagem convencional de busca de tratadores apropriados na propagação explícita. Em vez de percorrer a pilha de ativação inversamente, o modelo propõe a busca em relação a composição de entes. A seqüência, neste caso, é dada por: quando uma exceção é levantada em uma ação, a primeira busca é por tratadores associados àquela ação. Se o tratador for encontrado e propagar a exceção, a busca continuará no ente cujo comportamento contenha a ação que levantou a exceção. Se houver um tratador associado ao ente e se este propagar a exceção, a busca continuará no *blackboard* holoexception de seu pai. O limite é o ente `d_holo` (nível zero), pai dos entes do primeiro nível de composição. Nota-se que a composição dos entes em tempo de execução pode ser representada em uma estrutura denominada HoloTree. Esta é uma estrutura hierárquica (árvore) usada para organizar os entes em tempo de execução. A árvore implementa o encapsulamento dos entes em níveis de composição conforme proposto pelo Holoparadigma. Considerando a HoloTree, a propagação explícita percorre seus nodos das folhas para a raiz. No caso da propagação automática, HoloException utiliza a abordagem convencional de busca de tratadores apropriados. Logo, percorre a pilha de ativação inversamente até encontrar um tratador apropriado para a exceção *UncaughtException*.

4.5. Continuação do Fluxo de Execução

HoloException adota o modelo de parada. Desta forma, a atividade do nível em que a exceção foi levantada não pode ser retomada. Conceitualmente, isto significa que a atividade é finalizada no sinalizador. Quando for levantada uma exceção durante a execução de uma ação tem-se a seguinte seqüência: (i) a execução normal é interrompida; (ii) ocorre a clonagem da exceção levantada. Caso o ente que defina a exceção tenha ação-guia, esta será executada; (iii) realiza-se a busca do tratador; (iv) caso o tratador seja encontrado, a atividade de tratamento é executada. Caso contrário, acontece propagação automática da exceção *UncaughtException*. Em função disso, volta-se à primeira etapa desta seqüência. É importante salientar que a busca por um tratador para esta exceção recomeça no invocador da ação que sinalizou a exceção original; (v) a execução retorna ao invocador da ação que sinalizou a exceção. Nota-se que a continuação do fluxo de execução segue segundo a pilha de ativação. Se a exceção for sinalizada em uma aplicação com apenas um ente de uma única ação, esta será encerrada e o ente `d_holo` extinto. Neste caso, a aplicação será encerrada.

4.6. Exceções Pré-definidas

HoloException conta com um conjunto de exceções pré-definidas. Conforme a classificação de [10], estas exceções são declaradas de forma implícita e associadas com condições errôneas detectadas pela máquina virtual, *middleware* ou *hardware* em tempo de execução.

No caso do HoloException, há uma hierarquia de exceções pré-definidas obtida por clonagem estática (semelhante à herança na orientação a objetos). Este conjunto de exceções engloba (i) características particulares do Holoparadigma e (ii) problemas relacionados com o ambiente de execução. As seguintes exceções são do primeiro grupo: ***UncaughtException*** – indica, conforme anteriormente mencionado, que não foi encontrado um tratador apropriado para a exceção levantada associado à ação ou ao ente que sinalizou a exceção; ***BeingNotFound*** – indica que um ente não foi encontrado. Ela pode ser levantada, por exemplo, (i) quando se tenta mover um ente (ação `move`) para o contexto de um ente que não existe ou (ii) quando se tenta executar uma ação da interface de um ente que não existe mais; ***ActionNotFound*** – indica que uma ação não foi encontrada. Esta é sinalizada, por exemplo, quando uma ação que foi removida do comportamento de um ente é invocada; ***BeingExtinguished*** – indica que um ente foi extinto. Esta exceção é levantada quando uma ação invoca uma ação de outro ente que é extinto enquanto a segunda ação ainda está em execução. Vale notar que quando um ente é extinto, seu contexto também deixa de existir. Assim, a primeira invocação também perde seu significado; ***InvalidPermission*** – refere-se a questões relacionadas com permissão de acesso. Uma possível especialização desta seria *ActionInvocationInvalidPermission*. Neste caso, a exceção é levantada quando um ente tenta executar uma ação de outro ente que não está relacionada na interface do ente destino.

O segundo grupo, por sua vez, tem a exceção *EnvironmentException*. Esta pode ser diretamente relacionada com uma série de exceções da máquina virtual. Além disso, pode ser especializada em *UnavailableService* e *NetworkException*. A utilização efetiva deste modelo certamente ajudará na identificação de novas possíveis exceções. Desta forma,

vale salientar que as exceções pré-definidas não foram esgotadas neste trabalho e que futuramente este conjunto poderá ser ampliado.

5. Aplicação

O modelo apresentado foi implementado com a inclusão dos mecanismos de tratamento de exceções na plataforma atual de execução do Holo. O compilador e o montador foram alterados para traduzir as novas construções referentes aos mecanismos escritos na Hololinguagem para *bytecodes* Holo. O desenvolvimento destas duas etapas foi facilitado pelo uso das ferramentas Flex e Bison.

```
1  holo() {
2    holo(Quantas) {
3      holoexception!mineiroNaoQuerTrabalhar(trata, *);
4      for Cont := 1 to Quantas do clone(mina(Cont), Minas[Cont]);
5      clone(mineiro(Minas, Cont), o_mineiro);
6      for Cont := 1 to Quantas do history#dado(Cont,#Result[Cont]);
7    }
8    trata() { writeln('Mineiro precisa de férias!'); }
9  }

11 mineiro() {
12   mineiro(Minas, Nminas) {
13     time(Inicio);
14     for Cont := 1 to Nminas do {
15       time(Agora);
16       move(self, Minas[Cont]));
17       out(holoexception)!mineiroNaoQuerTrabalhar
18                         (raise mineiroNaoQuerTrabalhar, *);
19       minera(Agora - Inicio, Cont, Result);
20       move(self, out);
21       out(history)!dado(Cont,Result);
22     }
23   }
24   minera(Difenca, Ident, Result) {
25     if (Diferenca > 5 )
26       raise mineiroNaoQuerTrabalhar;
27     out(history)#dado(Ident, #Result);
28   }
29   trata() { raise mineiroNaoQuerTrabalhar; }
30   holoexception{
31     mineiroNaoQuerTrabalhar(trata, minera).
32     mineiroNaoQuerTrabalhar(trata, *).
33   }
34 }

36 mina() {
37   mina(Ident) {
38     for Cont := 1 to 4 do
39       if ((Ident + Cont) > 6)
40         raise dadosDemais(Ident);
41       else history!dado(Ident, Cont);
42   }
43 }

45 mineiroNaoQuerTrabalhar() exception { }

47 dadosDemais() exception {
48   dadosDemais(Ident) { writeln('Dados demais na mina', Ident); }
49 }
```

Figura 5. Simulação de mineração de dados

É importante salientar que várias verificações não foram adicionadas ao compilador em função da linguagem suportar mobilidade de código. Em outras palavras, a descrição estática de um ente pode levantar uma exceção que não está definida na máquina virtual em que o mesmo irá executar inicialmente. No entanto, este ente pode mover-se para uma máquina virtual que defina a exceção e que torne a sinalização da mesma possível.

A máquina virtual ubiVM (implementada em *ANSI C++*) foi alterada para suportar a mudança na estrutura organizacional de um ente proposta no modelo, bem como

```
1) passa como argumento as minas a criar (no caso, três);
2) inicia com a clonagem de transição automática de holo;
3) inicia a execução da ação-guia pertecente ao holo;
4) realiza uma afirmação no blackboard holoexception, que
   define um tratador associado ao ente holo para a exceção
   mineiroNaoQuerTrabalhar;
5) ocorre uma iteração, são criadas (clone, linha 5) as
   minas que serão exploradas pelo mineiro. Estas minas
   iniciam novos fluxos de execução, e cada uma delas irá
   executar a sua ação-guia;
5.1) a ação-guia da mina realiza uma iteração, na qual há
   um teste. Se o teste for verdadeiro (o que acontece na
   criação da 3a mina), é levantada a exceção dadosDemais.
   É clonado o ente e executado seu comportamento (impres-
   são de mensagem com a informação de contexto manipulada).
   A seqüência é a execução do tratamento. Como não foi
   definido nenhum tratador para esta exceção associado à
   ação-guia ou ao ente, é propagada UncaughtException pa-
   ra a ação invocadora (ação-guia pertencente ao ente ho-
   lo). Como a ação-guia pertencente ao ente holo também
   não define um tratador para esta exceção, a aplicação
   encerra. Se o teste for falso, é realizada uma afirmação
   na história na mina. Através deste processo, são incluí-
   das tuplas que serão mineiradas pelo mineiro;
6) é criado um mineiro (linha 6). O mineiro criado inicia
   um novo fluxo de execução, e executa a sua ação-guia;
6.1) a ação-guia do mineiro armazena a hora (ação time);
6.2) inicia uma iteração;
6.3) armazena a hora novamente;
6.4) o mineiro (self) é movido para o contexto de um ente
   mina (ação move);
6.5) realiza uma afirmação no blackboard holoexception de
   um ente mina usando a ação pré-definida out sensível ao
   contexto (linhas 17 e 18). Esta manipulação insere uma
   tupla no blackboard, que define um tratador associado ao
   ente tipo mina para a exceção mineiroNaoQuerTrabalhar;
6.6) invoca a ação minera, que faz um teste. Se o teste
   for verdadeiro, é levantada a exceção mineiroNaoQuerTra-
   balhar. O ente é clonado. Como este não define comporta-
   mento, segue a busca de tratadores. Há dois tratadores
   definidos para esta exceção no blackboard holoexception.
   Primeiro é executado o tratador associado à ação sinali-
   zadora (definido na linha 31). A atividade de tratamento
   é definida pela ação trata e faz a propagação explícita
   da exceção para o ente. Realiza nova busca de tratador.
   O tratador é encontrado (definido na linha 32). A ativi-
   dade de tratamento é definida pela ação trata e faz a
   propagação explícita da exceção para o ente-pai. Reali-
   za-se nova busca de tratador no contexto de um ente do
   tipo mina. O tratador é encontrado (antes de mover-se
   para uma mina, o mineiro inclui um tratador associado
   ao ente mina para esta exceção). Este tratador faz a
   propagação explícita da exceção. A busca por um trata-
   dor segue então no contexto do ente holo. O tratador é
   encontrado mais uma vez, porque no início da execução a
   manipulação dinâmica com o blackboard inclui um trata-
   dor associado ao ente holo para a exceção mineiroNao-
   QuerTrabalhar. A atividade de tratamento é definida pe-
   la ação trata pertencente ao ente holo. Esta ação im-
   prime uma mensagem (linha 8). Terminada a execução da
   ação, o fluxo de execução volta para a ação invocadora
   da ação que sinalizou a exceção (ação-guia do mineiro);
   Se o teste for falso, o mineiro realiza uma pergunta à
   história na mina. Através deste processo, as tuplas da
   mina são mineiradas pelo mineiro;
6.7) o mineiro (self) volta para o contexto relativo ao
   ente d_holo (out) (ação move da linha 20);
7) o mineiro afirma o resultado da mineração na história
   pertencente ao ente d_holo;
8) o ente d_holo realiza perguntas a sua história, aguar-
   dando o resultado das minerações.
```

Figura 6. Seqüência de execução

para permitir a execução das novas instruções definidas para a Hololinguagem. Para tanto, a classe *Being*, que representa um ente, teve um atributo adicionado: *_holoexception*, o *blackboard* agregado pelo modelo. A classe *HoloProcessor* foi modificada de forma a suportar os novos *bytecodes*. Entre essas modificações, destaca-se a referente ao *bytecode* RAISE, que levanta uma exceção. Este evento realiza a clonagem de um ente que representa a exceção. Além disso, cria uma nova ação que será responsável pela busca do tratador para a exceção sinalizada. Por fim, a classe *BlackBoard* foi alterada para suportar um novo tipo de *blackboard* (holoexception). Este tipo está encapsulado em *BlackBoardType*. Além disso, vale notar que os métodos desta classe foram alterados para que a semântica de operação do novo *blackboard* ficasse de acordo com o proposto no modelo.

A fim de validar a implementação, foi criada uma aplicação que segue os conceitos do Holoparadigma e utiliza os mecanismos de tratamento de exceções propostos. A Figura 5 apresenta o código-fonte de uma aplicação que simula uma mineração de dados (*data mining*). A aplicação cria um determinado número de minas (ente *mina* – linha 36) e um mineiro (ente *mineiro* – linha 11). O número de minas é um parâmetro passado na linha de comando (parâmetro *Quantas*). O mineiro entra nas minas (comando `move`), realiza uma ação de mineração (*minera*, linha 24) e armazena o resultado na história de `d_holo`. Os dados manipulados são elementos de tuplas e o processamento realizado pelo mineiro consiste em perguntas à história da mina.

O mineiro utiliza a ação `move` para entrar nas minas. A visão de contexto do mineiro muda após cada execução de move. O acesso à história depende da localização do mineiro. Na aplicação, o acesso ao exterior de um ente é realizado com a ação pré-definida `out`. O ente mineiro utiliza este recurso uma vez para o deslocamento (ação `move`) e duas vezes para acesso a história do composto no qual está localizado. Este tipo de acesso é sensível ao contexto, ou seja, depende da movimentação de um ente. No exemplo, o primeiro acesso externo (`out` na ação *minera*, linha 27) é direcionado para a história das minas. O segundo acesso externo (`out` no final do `for`, linha 21) é direcionado para a história de `d_holo`.

Além disso, a aplicação tem dois entes que representam exceções definidas pelo programador. São eles: (i) *mineiroNaoQuerTrabalhar* (linha 45) e (ii) *dadosDemais* (linha 47). A primeira exceção é levantada quando o tempo de processamento do mineiro ultrapassa cinco segundos, conforme definido no teste condicional na linha 25. Esta exceção não possui comportamento definido. A segunda, por sua vez, é levantada quando o número de tuplas que se tenta inserir na história da mina mais seu identificador é maior do que o número seis (o que pode ser verificado no teste da linha 39). Esta exceção possui ação-guia, e manipula uma informação de contexto (referente ao identificador da mina). Uma instância de execução da aplicação é dada pela seqüência mostrada na Figura 6.

6. Trabalhos Relacionados

Existem poucos trabalhos na literatura que consideram questões relacionadas ao tratamento de exceções em aplicações móveis. Normalmente, estas aplicações utilizam mecanismos fornecidos pelas linguagens de programação utilizadas em *middlewares* de desenvolvimento. No entanto, raramente estes mecanismos preocupam-se com características próprias da computação móvel e ubíqua, como cenários dinâmicos.

[14] apresentam como principal problema relacionado com tratamento de exceções distribuídas: cada processo afetado deve invocar o tratador de exceção correto, para não levantar necessariamente a mesma exceção em todos os processos afetados. Para resolver esta questão, os autores propõem o Modelo Guardião (*Guardian Model*) para tratamento de exceções em sistemas distribuídos. O modelo é baseado na noção de tratamento de exceção global. A principal desvantagem deste modelo é a impossibilidade de existirem tratadores no contexto individual dos processos. Outro problema é a centralização do tratamento de exceções em uma entidade especializada. Além disso, a proposta não considera as necessidades específicas de aplicações móveis e ubíquas, como a utilização de informação de contexto no tratamento de exceções.

[13] propõem um novo modelo de tratamento de exceções que permite a recuperação de erros específicos da aplicação em sistemas de agentes móveis baseados em coordenação. O mecanismo proposto é assíncrono e preserva a comunicação anônima dos agentes. Além disso, pode ser facilmente incorporado em *middleware* de agentes móveis baseados em coordenação. Este tipo de tratamento de exceções ajusta-se bem às principais características de sistemas ubíquos, pois é aberto e dinâmico por natureza não impondo restrições à assincronia e à dinamicidade dos agentes. A principal característica do modelo constitui também sua principal limitação: a abordagem é uma solução específica para aplicações que baseiam suas interações em espaços de tuplas. Além disso, este modelo também não considera as informações de contexto no tratamento de exceções.

[5] propõe um modelo de tratamento de exceções sensível ao contexto construído sobre o sistema MoCA (*Mobile Collaboration Architecture*). O modelo definido apresenta suporte explícito para especificação de "contextos excepcionais". Além disso, permite buscas sensíveis ao contexto por tratadores de exceção. Existe escopo de tratamento multi-nível que fornece novas abstrações, bem como abstrações relacionadas ao *middleware* sensível ao contexto subjacente, como dispositivos, regiões, e servido-

res. A propagação de erros é sensível ao contexto. O modelo permite, ainda, tratamento de exceções pró-ativo. A limitação da solução constitui-se do fato dela ser específica para aplicações que utilizam o paradigma *publish-subscribe* e o *middleware* MoCA.

7. Considerações Finais

Diante das novas possibilidades percebidas com a popularização da computação móvel e ubíqua, é necessária a criação de novos mecanismos relacionados à área. A importância deste trabalho é dada por esta necessidade, considerando que sua principal contribuição é, justamente, a proposta de um modelo novo de tratamento de exceções, orientado a contextos e focado em aplicações desenvolvidas segundo o novo paradigma de computação, que se mostra cada dia mais presente no nosso dia-a-dia. Este modelo adequa-se perfeitamente a este paradigma por ter sido proposto considerando as suas necessidades, isto é, por apresentar características diretamente associadas a mobilidade e a sensibilidade ao contexto.

Propõe-se como trabalhos futuros: (i) desenvolver novas aplicações Holo que explorem os mecanismos de tratamento de exceções propostos, de forma a verificar a necessidade de extensão das exceções pré-definidas; (ii) analisar a possibilidade de propor um método de propagação de exceções que possa intercalar a propagação em nível de hierarquia de entes (contextos) e em nível de fluxo de execução; (iii) analisar a viabilidade de propor uma alternativa que permita a utilização do modelo de continuação ao que se refere a continuação do fluxo de execução após o tratamento das exceções; e (iv) definir a semântica formal dos mecanismos apresentados.

Referências

[1] J. Barbosa and C. Geyer. Uma Linguagem Multiparadigma Orientada ao Desenvolvimento de Software Distribuído. *V Simpósio Brasileiro de Linguagens de Programação (SBLP), Curitiba*, 6, 2001.

[2] J. Barbosa, R. Hahn, D. Bonatto, F. Cecin, and C. Geyer. Evaluation of a large-scale ubiquitous system model through peer-to-peer protocol simulation. In *DS-RT '07: Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 175–181, Washington, DC, USA, 2007. IEEE Computer Society.

[3] J. Barbosa, A. Yamin, P. Vargas, I. Augustin, and C. Geyer. Holoparadigm: a multipradigm model oriented to development of distributed systems. In *Ninth International Conference on Parallel and Distributed Systems (ICPADS)*, pages 165–170, 2002.

[4] J. Barbosa, A. Yamin, P. Vargas, D. Ferrari, and A. Schaeffer. Using Mobility and Blackboards to Support a Multiparadigm Model Oriented to Distributed Processing. *Symposium on Computer Architecture and High Performance Computing, Pirenópolis*, 13:187–194, 2001.

[5] K. Damasceno, N. Cacho, A. Garcia, A. Romanovsky, and C. Lucena. Context-aware exception handling in mobile agent systems: the moca case. In *SELMAS '06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pages 37–44, New York, NY, USA, 2006. ACM Press.

[6] F. J. C. de Lima Filho. *Tratamento de Exceções no Desenvolvimento de Sistemas Tolerantes a Falhas Baseados em Componentes*. PhD thesis, Universidade Estadual de Campinas, Campinas, SP, Brasil, 2006.

[7] K. Dey, D. Salber, and G. D. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human Computer Interaction*, 16(2-4):97–166, 2001.

[8] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *The Journal of Systems and Software*, 59(2):197–222, 2001.

[9] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002.

[10] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[11] R. Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[12] H.-G. Hegering, A. Küpper, C. Linnhoff-Popien, and H. Reiser. Management Challenges of Context-Aware Services in Ubiquitous Environments. In *DSOM*, pages 246–259, 2003.

[13] A. Iliasov and A. Romanovsky. Exception handling in coordination-based mobile environments. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 341–350, Washington, DC, USA, 2005. IEEE Computer Society.

[14] R. Miller and A. Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Transactions on Software Engineering*, 30(12):1008–1022, 2004.

[15] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[16] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, 1996.

[17] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

[18] S. Vranes and M. Stanojevic. Integrating multiple paradigms within the blackboard framework. *Software Engineering, IEEE Transactions on*, 21(3):244–262, 1995.

[19] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pages 94–10, sep 1991.

[20] A. C. Yamin and I. Augustin. Computação Pervasiva: Como programar aplicações, 2006. Tutorial no X Simpósio Brasileiro de Linguagens de Programação.