Um Mecanismo de Composição de Eventos para Resolução de Exceções Sensíveis ao Contexto

Frederico Lopes[1], Nélio Cacho[2], Thais Batista[1]

[1]Departamento de Informática e Matemática Aplicada – Universidade Federal do Rio Grande do Norte (UFRN)

[2] Computing Department, Lancaster University, United Kingdom

fred.lopes@gmail.com, n.cacho@lancaster.ac.uk, thais@dimap.ufrn.br

Resumo. *Esse artigo apresenta um mecanismo de composição de eventos (CES) que trabalha em cooperação com mais de uma infra-estrutura publish-subscribe simultaneamente e oferece uma linguagem com um conjunto de operadores para definição de eventos compostos permitindo o compartilhamento de informações contextuais. Além disso, o artigo apresenta o uso do CES para permitir a resolução de exceções concorrentes em aplicações sensíveis ao contexto.*

Abstract. *This paper presents a composite event mechanism (CES) that works in conjunction with more that one publish-subscribe system simultaneously and offers a language with a set of operators to the definition of composite events. Moreover, the paper presents the use of CES to resolve concurrent exceptions in context-aware systems.*

1. Introdução

A comunicação baseada em eventos caracteriza-se pela sua natureza assíncrona, por eliminar o acoplamento direto entre componentes e permitir a disseminação seletiva de informação baseada nos interesses dos elementos comunicantes. Esse tipo de comunicação tem sido amplamente utilizada [Sacramento 2004] em aplicações sensíveis ao contexto [Dey and Abowd 2000] nas quais mudanças no ambiente geram eventos contextuais que devem ser tratados pela aplicação.

Aplicações sensíveis ao contexto podem capturar, representar e processar tipos de informações contextuais (localização, velocidade, capacidade de processamento, temperatura, estação ao ano, etc) e adaptar sua funcionalidade de acordo com as informações de contexto atual [Dey and Abowd 2000]. Tipicamente, aplicações sensíveis ao contexto executam em ambientes móveis. Nesse cenário, o contexto pode mudar freqüentemente e, como conseqüência, eventos contextuais emergem concorrentemente, de forma dinâmica e de diferentes locais. Essas peculiaridades demandam, do mecanismo de comunicação baseada em eventos, suporte para composição de eventos concorrentes. Tal composição permite transformar um conjunto de eventos de menor significado prático em um evento que pode ser utilizado para adaptar eficientemente o sistema. Adicionalmente, mudanças nas informações de contexto (Ex.: alta temperatura) também podem degradar ou interromper os serviços fornecidos pelos dispositivos e atuadores que implementam a aplicação sensível ao contexto. Dessa forma, exceções concorrentes podem ser levantadas para indicar uma

falha causada por uma dessas mudanças. Nesse cenário, exceções concorrentes também devem ser resolvidas para capturar a semântica das exceções levantadas e invocar o tratador adequado.

Na literatura existem vários sistemas de composição de eventos [Moreto 2001, Li 2005, Pietzuch 2003], os quais todos estes usam uma única infra-estrutura *publish-subscribe* subjacente para capturar a semântica dos eventos contextuais concorrentes. Entretanto, em aplicações sensíveis ao contexto, esta abordagem representa uma grande limitação na representação das mudanças nas informações contextuais. Isto ocorre porque aplicações sensíveis ao contexto são normalmente desenvolvidas através da utilização de várias infra-estruturas *publish-subscribe* que capturam diferentes tipos de informações contextuais, como localização, visão, movimento, áudio, etc. A restrição por apenas uma infra-estrutura implica em dizer que certas informações contextuais não serão consideradas pelo sistema de composição de eventos. Na prática, isto termina por limitar a expressividade do mecanismo e restringir a adaptação das aplicações para certos tipos de informações contextuais.

Esse artigo tem como objetivo apresentar um mecanismo para composição de eventos – *Composite Event System* (CES) – que permite o uso de várias infra-estruturas *publish-subscribe* simultaneamente e o compartilhamento de informações de contexto entre eventos que formam a composição de uma subscrição. Os operadores de composição são definidos com base nos requisitos de composição necessários para tratar informações contextuais. Além disso, esse trabalho apresenta como o CES é utilizado no contexto de tratamento de exceções concorrentes.

Esse artigo está estruturado da seguinte forma. A seção 2 apresenta os conceitos básicos relacionados com esse trabalho – composição de eventos (seção 2.1) e sensibilidade ao contexto (seção 2.2). A seção 3 descreve um mecanismo de tratamento de exceções para aplicações sensíveis ao contexto e as limitações de tal mecanismo no contexto de tratamento de exceções concorrentes e sensíveis ao contexto. A seção 4 apresenta a arquitetura do CES, sua estratégia para resolução de eventos compostos utilizando diversas infra-estruturas *publish-subscribe* e o compartilhamento de informações sensíveis ao contexto. A seção 5 descreve como as funcionalidades do CES foram utilizadas para estender o mecanismo de tratamento de exceções. A seção 6 apresenta os trabalhos relacionados e análise comparativa do CES com outros sistemas. A seção 7 apresenta as conclusões.

2. Conceitos

Esta seção apresenta a contextualização dos temas discutidos nesse trabalho. A seção 2.1 contém os conceitos relacionados com composição de eventos e características dos mecanismos de composição. A seção 2.2 conceitua sensibilidade ao contexto.

2.1 Composição de eventos

Comunicação *publish-subscribe* é um paradigma para construção de sistemas distribuídos que tem como vantagem baixo acoplamento entre os componentes, alta escalabilidade e um modelo simples para desenvolvimento de aplicações [Pietzuch 2003] as quais componentes podem registrar interesse em determinados eventos (*subscribers*) e/ou notificar acontecimento de eventos (*publishers*). Evento é o acontecimento de um interesse [Mansouri-Samani 1997] anteriormente registrado por

algum componente. Eventos isolados muitas vezes não são interessantes para determinado cliente por não representar uma informação realmente relevante em uma tomada de decisão. Em muitos casos, o próprio cliente teria que reunir informações contextuais dos eventos notificados para tentar chegar a alguma conclusão importante. Para exemplificar, suponha que para um componente tomar uma decisão ele necessite receber a notificação de que dois eventos (*a* e *b*) aconteceram. Sendo assim, esse componente subscreve-se para receber a notificação do evento *a* e do evento *b* separadamente. Quando ele receber a notificação de que o evento *a* aconteceu, ele tem que guardar essa informação e esperar por uma possível notificação do evento *b* para poder tomar sua decisão. Desse modo, o evento *a*, por si só, não traz uma informação contextual relevante, o mesmo valendo para *b*. Isso é, um evento primitivo pode não possuir um significado que resulte em uma mudança de estado ou na execução de uma tarefa. Porém, combinações destes eventos podem formar padrões de eventos mais especializados para determinada aplicação.

Para resolver esse problema, pode-se trabalhar com a noção de *hierarquias de eventos* ou *eventos compostos* (CE) [Pietzuch 2003]. Um evento composto é formado por combinações de eventos simples e, até mesmo, de outros eventos compostos. Sendo assim, os eventos *a* e *b* do exemplo anterior formariam um evento composto e o *Handler* somente informaria ao *subscriber* quando os dois eventos acontecessem. O uso de CEs é vantajoso por permitir subscrições a eventos que sejam completas e que signifiquem algo importante para o *subscriber*. Isso evita a inundação de notificações e diminui a complexidade dos *subscribers*, pois usando apenas eventos primitivos eles próprios teriam que juntar as notificações recebidas para identificar alguma informação relevante.

Uma subscrição composta é formada por um conjunto de publicações de eventos independentes e que podem ocorrer em tempos e lugares diferentes. Elas oferecem uma visão de alto nível aos *subscribers* pelo enriquecimento de expressividade de uma *linguagem de subscrição* [Li 2005]. Uma subscrição composta consiste em um conjunto de subscrições atômicas interligadas por operadores de composição de eventos.

De um modo geral, mecanismos de composição de eventos não implementam toda a infra-estrutura *publish-subscribe*. Em vez disso, aproveitam mecanismos existentes e implementam somente um mecanismo de detecção de eventos compostos desacoplados dos mecanismos *publish-subscribe* tradicionais. Essa característica além de permitir o foco somente na composição de eventos, ainda permite que o mecanismo de composição possa ser utilizado em diferentes infra-estruturas *publish-subscribe* com algumas modificações para adequar às especificidades de cada infra-estrutura.

Existem várias implementações de algoritmos para detecção de eventos compostos. [Gehani 1992] utiliza um autômato finito enquanto [Gatziu 1994] usa Redes de Petri. Porém, autômatos finitos são essencialmente seqüenciais e não são adequados para manipulação de eventos concorrentes e assíncronos. Redes de Petri são muito complexas e comparativamente ineficientes para reconhecer eventos em tempo de execução [Pietzuch 2003]. Por outro lado, algoritmos de árvores são os mais utilizados na literatura [Pietzuch 2003, Mansouri-Samani 1997, Li 2005] por serem simples e permitirem uma representação intuitiva da estrutura de uma expressão de um evento composto.

Para combinar os eventos é necessária a especificação de uma *linguagem de composição de eventos*. Essas linguagens devem ser intuitivas para que os usuários que não são programadores possam, facilmente, especificar seus interesses. Elas devem permitir a construção de expressões de eventos compostos contendo eventos simples e conectores que os relacionem. Cada conector representa um tipo específico de composição. Na literatura existem inúmeros conectores, os mais gerais podem ser encontrados em [Moreto 2001, Chakravarthy 1993, Gatziu 1994]. A Figura 1 ilustra alguns desses conectores com uma possível notação para cada um deles:

<table>
<tr><td>AND
Conector: ‘&’
Ex.: (Sístole>16 & Diástole>10)</td><td>OR
Conector: ‘|’
Ex.: (Sístole>16 | Diástole>10)</td><td>SEQUENTIAL
Conector: ‘;’
Ex.: (Sístole>16 ; Diástole>10)</td></tr>
<tr><td>INTERACTOR
Conector: ‘[’<repetitions>‘]’
Ex.: (Sístole>16) [20]</td><td>INTERVAL
Conector: ‘\’<time>‘\’
Ex.: (Sístole>16 \5\ Diástole>10)</td><td>PARALLEL
Conector: ‘/’
Ex.: (Sístole>16 / Diástole>10)</td></tr>
</table>

Figura 1: Conectores tradicionalmente utilizados em composição de eventos.

O conector *AND* define dois eventos que devem acontecer independentemente de tempo ou ordem. O conector *OR* indica que um dos eventos ligados pelo conector deve acontecer. O conector *SEQUENTIAL* exige que dois eventos devem acontecer obrigatoriamente na ordem pré-estabelecida, da esquerda para a direita. O conector *INTERACTOR* define a quantidade de vezes que o evento deve acontecer. O conector *INTERVAL* define que dois eventos têm que acontecer dentro do tempo pré-estabelecido, sem importar a ordem. Esse tempo é disparado a partir do momento em que um dos dois eventos envolvidos sejam notificados. Por fim, o conector *PARALLEL* exige que dois eventos aconteçam ao mesmo tempo.

Para exemplificar o uso de conectores, a Figura 2 apresenta dois exemplos de subscrição para monitoramento das funções cardíacas de um paciente. Na figura 2*A* o evento composto subscrito é dividido em forma de árvore onde os nós raiz e intermediários representam eventos compostos e cada nó-folha representa um evento simples que será subscrito a uma infra-estrutura *publish-subscribe* e os nós intermediários e raiz representam composições de evento. A figura 2*B* apresenta uma outra visão mostrando que o evento composto *(Sístole > 16 & Diástole > 10)* tem que ser publicado por cinco vezes para que o evento composto subscrito *(Sístole > 16 & Diástole > 10)[5]* seja notificado.

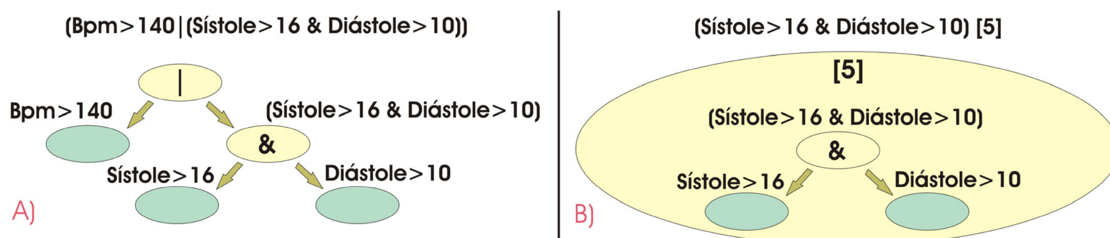

Figura 2: Representação dos nós em exemplos de expressões de composição de eventos.

2.2 Sensibilidade ao Contexto

A diminuição do custo das tecnologias de sensoriamento e a crescente utilização da computação móvel viabilizaram a implementação de aplicações sensíveis ao contexto. Tais aplicações são caracterizadas pela forte interação entre a aplicação e o ambiente

que a envolve. Para permitir tal interação, estas aplicações usam mecanismos de descoberta de serviços para obter os sensores e atuadores que estão disponíveis em um determinado local. A partir dos sensores, tais aplicações obtêm as informações contextuais que serão utilizadas para guiar o processo de adaptação. Com base nestas informações, a aplicação modifica o seu comportamento e configura o ambiente de acordo com as necessidades do usuário [Dey and Abowd 2000].

Sensibilidade ao contexto é uma propriedade que permite que uma aplicação possa reagir de acordo com as informações de contexto recebidas. Aplicações sensíveis ao contexto são geralmente implementadas por *middlewares* ou *frameworks*. MoCA[Sacramento 2004] é um exemplo de *middleware* que permite a coleta de informações contextuais, a disseminação de tais informações e a notificação dos clientes interessados nas modificações de tais informações. Por outro lado, *Context ToolKit* [Dey and Abowd 2000] é um exemplo de *framework* que foi desenvolvido para suportar a implementação de aplicações sensíveis ao contexto. Tais *frameworks* são estruturados de modo a facilitar o desenvolvimento de tais aplicações por meio de abstrações que evitam o acoplamento entre as aplicações e os detalhes de coleta de dados e adaptação.

3. Tratamento de Exceções Sensível ao Contexto

As técnicas de tratamento de exceção [Goodenough 1975, Parnas and Würges 1976] provêem as abstrações e os mecanismos essenciais para a construção de sistemas robustos. Tais técnicas garantem a modularidade do sistema na presença de falhas através do fornecimento de abstrações para (i) representar os estados errôneos de um sistema por meio da utilização de *exceções*, (ii) encapsular o tratamento de tais exceções dentro do código dos *tratadores,* (iii) introduzir *regiões protegidas* ou *escopos* para limitar o espaço de atuação de cada tratador e (iv) explicitamente definir as *interfaces excepcionais* de cada um dos módulos.

Entretanto, a evolução histórica dos mecanismos de tratamento de exceções tem mostrado que as abstrações fornecidas por tais mecanismos são fortemente relacionadas ao sistema de módulos utilizado [Miller and Tripathi 1997] e também as características das aplicações alvo. Por exemplo, os mecanismos de tratamento de exceção utilizados em sistemas concorrentes e distribuídos [Xu et al 1995] requerem a utilização de conceitos relativos a colaboração (*ações atômicas*) para que as exceções levantadas em uma *thread* sejam propagadas e tratadas de forma síncrona por todas as *threads* envolvidas na colaboração. Algo semelhante ocorre para os mecanismos utilizados no tratamento de exceções em sistemas *multi-agentes* [Souchon 2004], onde é necessário estender o mecanismo de definição de escopo para permitir a associação de tratadores com serviços, agentes e *papéis*.

Considerando tal evolução, uma série de estudos tem mostrado que aplicações móveis [Mostinckx 2006] e sensíveis ao contexto [Tripathi et al 2005, Cacho 2006, Damasceno 2006] requerem novas abstrações e mecanismos para tratar exceções em ambientes com instabilidade de conexão, presença de comunicação assíncrona e a sensibilidade ao contexto. Um conjunto de novas abstrações para tratar tais condições é definido por [Cacho 2006, Damasceno 2006] e resumidos na próxima subseção.

3.1 Abstrações para o Tratamento de Exceções Sensível ao Contexto

Para modelar os estados errôneos, o mecanismo descrito por [Cacho 2006, Damasceno, 2006] permite definir exceções que podem ser levantadas automaticamente através da utilização de um "*contexto excepcional*" ou explicitamente através do método *signal* (Figura 3, linhas 1-2). Um contexto excepcional corresponde a um conjunto indesejável de condições contextuais que, quando violadas, representam uma situação excepcional. A Figura 3 (linha 3) ilustra como relacionar a ocorrência de um contexto excepcional com uma exceção. O encapsulamento do código de tratamento é feito por meio da definição de um tratador sensível ao contexto. Tal tratador deve estender a classe *Handler* (Linha 4-7) e implementar os métodos *verifyContextCondition* e *execute*. O primeiro método define as condições contextuais na qual o tratador será invocado enquanto que o segundo define o código de tratamento da exceção.

A fim de suportar uma abordagem sensível ao contexto para o tratamento adequado de erros, exceções podem ser capturadas através de quatro tipos diferentes de escopos: um dispositivo, um grupo de dispositivos, um servidor e uma região. Nos três primeiros tipos de escopo o sistema deve indicar explicitamente através do método *addDeviceScope* quando o dispositivo entrar no escopo e usar o método *removeDeviceScope* para informar que o dispositivo deixou o escopo. Diferentemente, um escopo de região tem um comportamento mais dinâmico, uma vez que a entrada ou saída de um dispositivo no escopo depende da posição espacial deste dispositivo. Ou seja, o escopo de região define uma região (Sala, Departamento, etc) que limita os dispositivos que fazem parte ou não deste tipo de escopo. As linhas 9 e 10 ilustram a associação de um tratador com o escopo de região de um dispositivo móvel.

```
1  HeartAttack hattack = new HeartAttack();
2  hattack.signal();
                 <ou>
3  HeartAttack hattack = new HeartAttack("(BPM<40 and SystolicBloodPressure > 180)");

4  public class FirstHelp extends Handler {
5     public boolean verifyContextCondition(Exception ex, Context context){ ... }
6     public boolean execute(){ ... }
7  }
8  FirstHelp fhelp = new FirstHelp(hattack);
9  RegionScope regionScope =  RegionScope.getInstance();
10 regionScope.attachHandler(fhelp);
```

Figura 3: Abstrações utilizadas para tratar exceções em aplicações sensíveis ao contexto.

3.2 Limitações do mecanismo atual

As abstrações definidas na seção anterior foram utilizadas na implementação de dois estudos de caso [Cacho 2006, Damasceno 2006] que tratam exceções em aplicações móveis e sensíveis ao contexto. Tais estudos demonstraram a necessidade de se estender o atual mecanismo para suportar a resolução de exceções concorrentes. Assim como eventos que requerem um mecanismo de composição para capturar a semântica da ocorrência de vários eventos simultâneos, exceções também requerem o mesmo mecanismo. Entretanto, neste caso, eles são chamados de *mecanismos de resolução de exceções* [Campbell and Randell 1986]. Em geral, estes mecanismos disponibilizam uma *função de resolução* que determina a estratégia utilizada para resolver as exceções. A Figura 4 ilustra um cenário em que vários dispositivos levantam exceções concorrentes e tais exceções são resolvidas em uma *exceção universal* através da utilização de duas estratégias [Campbell and Randell 1986]: *hierarquia de exceções* e *árvore de exceções.* A primeira estratégia associa a cada exceção uma prioridade e, por

conseguinte considera a exceção universal como sendo aquela que possuir a maior prioridade. A simplicidade da primeira estratégia não é capaz de expressar a complexa semântica da resolução de exceções. Por este motivo, *árvore de exceções* foram introduzidas para representar na raiz da árvore a exceção universal e nas folhas as exceções inicialmente levantadas pelas aplicações.

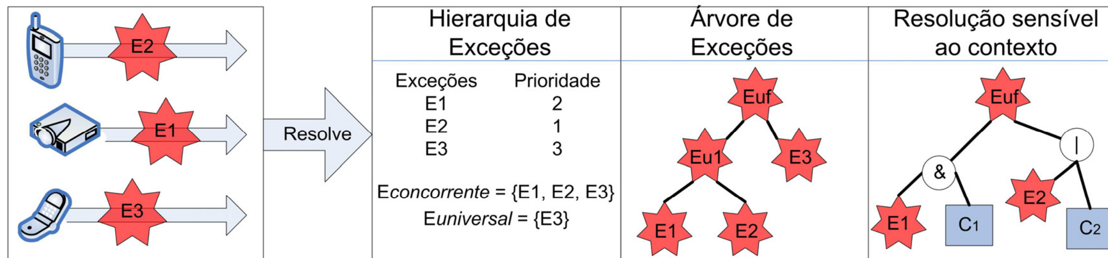

Figura 4: Abordagens para resolução de exceções concorrentes.

Apesar da semântica das árvores de exceções ser capaz de determinar uma ordem parcial para o levantamento das exceções [Xu et al 2000], essa abordagem não é capaz de resolver adequadamente exceções em ambientes móveis e sensíveis ao contexto. A figura 4 mostra que a resolução de exceções em tais ambientes requer o tratamento das informações contextuais como parte da árvore de resolução. Estas informações são utilizadas para permitir que o contexto da exceção (E_1) seja comparado com o tipo de contexto (localização, temperatura, velocidade, etc.) do dispositivo móvel (C_1).

A utilização de informações contextuais em uma árvore de resolução de exceções pode ser implementada através da utilização de um mecanismo de composição de eventos contextuais. Neste caso, o mecanismo de composição de eventos deve suportar a separação entre os nós que representam exceção e os nós que descrevem condições contextuais. Esta separação é importante para manter a principal motivação de se utilizar um mecanismo de tratamento de exceções que é exatamente separar no fluxo excepcional as abstrações responsáveis por modelar as situações de erro [Lee and Anderson 1990].

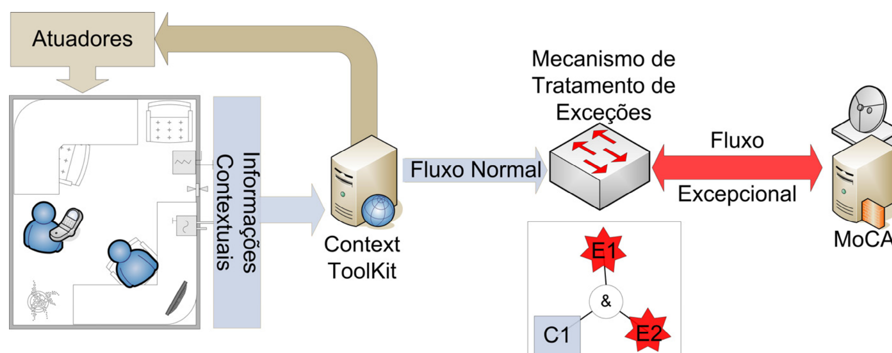

Figura 5: Cenário de utilização do mecanismo de tratamento de exceções

A separação entre fluxo normal e excepcional pode ser vista na Figura 5. Esta figura descreve um cenário onde o mecanismo de tratamento de exceções é introduzido em uma aplicação que já utiliza o *Context Toolkit* para tratar as informações contextuais e a adaptação da aplicação. Para cenários como este, o mecanismo de resolução de exceções deve ser capaz de prover duas funcionalidades: (i) utilizar duas ou mais infra-estruturas *publish-subscribe* para resolver em uma mesma árvore informações

contextuais e excepcionais, (ii) suportar uma linguagem de definição de exceção universal que abstraia a separação entre os estes dois tipos de informações.

No sentido de suportar tal mecanismo de resolução de exceções, a próxima seção descreve a arquitetura e os detalhes de implementação de um mecanismo de composição de eventos que foi desenvolvido para prover as duas características descritas acima. A seção 5 descreve a arquitetura estendida do mecanismo de tratamento de exceções e um conjunto de novas abstrações introduzidas por meio da utilização do mecanismo de composição de eventos.

4. Serviço de Composição de Eventos

O CES (*Composite Event System*) propõe duas novas características para sistemas de composição de eventos: (i) disponibiliza um mecanismo que coopera com várias infra-estruturas *publish-subscribe,* simultaneamente, que não possuem tratamento para CE ou que possuem tratamento apenas para alguns tipos de composição mais simples; (ii) compartilha informações obtidas em um evento de uma subscrição composta para determinar, dinamicamente, condições de outros eventos na mesma subscrição composta.

4.1 Arquitetura

A Figura 6 ilustra a arquitetura do CES. O mecanismo dispõe de três interfaces. Duas delas oferecidas aos clientes que desejem fazer subscrições (*Admin* e *LookupOut*) e uma utilizada para que o mecanismo possa subscrever-se às infra-estruturas *publish-subscribe* (*LookupIn*).

A finalidade da interface *Admin* é permitir que o cliente possa inserir informações de configuração no mecanismo. Como o CES pode subscrever-se para eventos simples em diferentes infra-estruturas *publish-subscribe*, o cliente precisa informar, antes de se subscrever, quais *publish-subscribe* o CES irá utilizar, passando a referência para cada uma das infra-estruturas e podendo ainda escolher uma delas como *default*. Além disso, o cliente precisa especificar, para cada evento simples, qual infra-estrutura *publish-subscribe* o CES irá subscrever-se. Caso o cliente não especifique essas informações sobre determinado evento, o CES usará a infra-estrutura *default* para subscrevê-lo. Essa facilidade permite que uma mesma subscrição composta receba informações de contexto de diferentes infra-estruturas *publish-subscribe*. O módulo *Configurator* é responsável por controlar a inserção das configurações e as armazenar no repositório *Configuration*. A configuração é dinâmica e pode ser ajustada em tempo de execução. A existência do *Configurator* representa a principal diferença entre o CES e outros mecanismos de composição de eventos.

A interface *LookUpOut* é responsável por receber as subscrições do cliente e notificá-lo quando o evento composto for detectado. A interface *LookupIn* é responsável por permitir que o CES faça subscrições para eventos simples às infra-estruturas *publish-subscribe*. Ao receber uma subscrição, o módulo *SubscriptionsManager,* que é o responsável por controlar as subscrições do cliente, requisita que o módulo *Parser* decomponha a subscrição composta e invoque o módulo *TreeFactory* para que este gere a árvore relativa a subscrição. Para cada evento simples identificado, o *TreeFactory* consulta no repositório *Configuration* para qual *publish-subscribe* ele deve subscrever-

se para o evento simples envolvido. Após isso, ele requisita ao *ListenerFactory* a criação de um *listener* específico para o mecanismo de eventos, de acordo com a configuração. Essa especificidade é fortemente necessária, pois a interface dos mecanismos de eventos são diferentes uma das outras. Após gerada toda a árvore, o *TreeFactory* a entrega ao *SubscriptionsManager*.

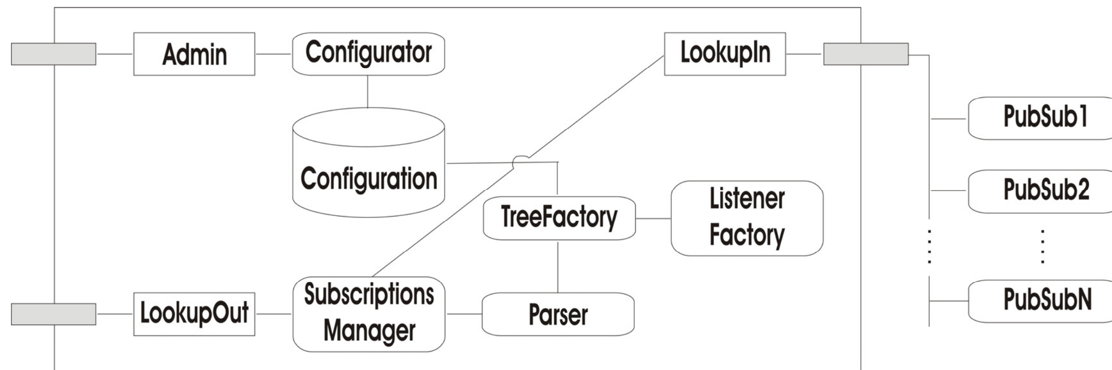

Figura 6: Arquitetura do CES

De posse da árvore relativa a subscrição, o módulo *SubscriptionsManager*, através da interface *LookupIn* faz todas as subscrições, cada uma a infra-estrutura *publish-subscribe* especificado na configuração. Nesse momento o CES já está monitorando os eventos simples notificados. Quando identifica que as condições de um evento composto 'X' foram satisfeitas, o *SubscriptionsManager* utiliza a interface *LookupOut* para notificar ao cliente que 'X' aconteceu. Detalhes sobre a árvore e sobre o controle da detecção de eventos compostos serão vistos na próxima seção.

4.2 Resolução de Eventos Compostos

A linguagem de subscrição do CES é formada pelos seis conectores exemplificados na seção 2.1 e permite uma fácil especificação de infinitos eventos compostos. Exemplos de subscrição utilizando esta linguagem podem ser vistas ainda na Figura 1. Existe ainda um conector especial (*and*) usado para que uma única subscrição a uma infra-estrutura publish-subscribe possa conter múltiplas condições. Isso é necessário para que o CES não diminua o poder de expressividade dos eventos suportados pelas infra-estruturas *publish-subscribe*. Um exemplo utilizando esse conector será ilustrado na seção 4.3.

O algoritmo de árvore é usado pelo mecanismo para detecção de eventos. Essa escolha, além dos benefícios citados anteriormente, permite que o mecanismo interaja mais facilmente com mais de uma infra-estrutura *publish-subscribe* já que cada nó folha (apresentado abaixo como *LeafNode*) é responsável por fazer uma subscrição relativa a um evento simples e essas subscrições são totalmente independentes uma das outras.

Existem dois tipos de nós em uma árvore que representa uma subscrição: *LeafNode* e *DecisionNode*. Cada *LeafNode* representa um evento subscrito a um *publish-subscribe* e é através deste nó que o CES recebe a notificação deste evento. Quando uma notificação acontece, o *LeafNode* avisa ao nó que está no nível imediatamente a cima do seu de que o evento chegou. Esse nó obrigatoriamente é um *DecisionNode*. A Figura 7 identifica os tipos de nós em uma árvore para a subscrição *(ev1 ; ev2)[3]*. Cada *DecisionNode* possui um autômato que vai mudando de estado de

acordo com os avisos de notificação recebidos através dos *LeafNodes*. Quando um *DecisionNode* chega a seu estado final, de modo semelhante aos *LeafNodes*, avisa ao nó que está no nível imediatamente a cima do seu. Caso o *DecisionNode* que se encontra no maior nível da árvore chegue ao seu estado final, ele avisa ao *SubscriptionManager* que, através do *listener* passado pelo cliente no momento da subscrição, notifique o evento composto subscrito pelo cliente. O autômato de cada tipo de *DecisionNode* possui estados e transições diferentes. Pela forma como os autômatos são definidos, novos autômatos podem ser facilmente inseridos sem que para isso seja necessário um grande conhecimento da estrutura do CES.

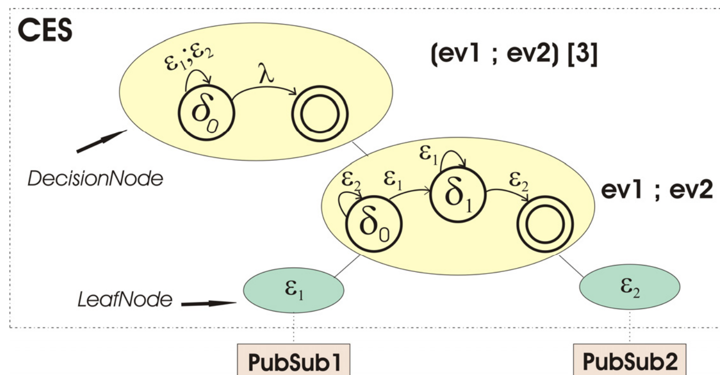

Figura 7: *DecisionNode* e *LeafNodes* em uma subscrição composta

O cliente do CES pode, dinamicamente, configurar o ambiente de composição de eventos através da interface *Admin*. Para adicionar uma nova infra-estrutura *publish-subscribe* ele usa a operação *setPS* passando como parâmetro o nome do repositório, um objeto representando a conexão com o qual os *LeafNodes* subscrevem-se às infra-estruturas *publish-subscribe* e um *boolean* para informar se esse é o *publish-subscribe default* do ambiente. É importante ressaltar que, como cada infra-estrutura possui sua própria interface e representação específica de informações de contexto, o módulo *ListenerFactory* tem que conhecer previamente essas peculiaridades, ainda em tempo de compilação. Em tempo de execução, no momento da configuração, *setPS* apenas recebe a referência (ou conexão) para as infra-estruturas *publish-subscribe* que o cliente deseja utilizar. Desse modo, para inserir um *publish-subscribe* na lista dos que o CES suporta, basta implementar o *listener* da referida infra-estrutura. Para cada atributo que deseja envolver nas suas subscrições, o cliente deve especificar, através da mesma interface, utilizando o método *setAttributes,* passando como parâmetros uma lista de atributos e o nome da infra-estrutura *publish-subscribe* que receberá subscrições para estes atributos.

4.3 Compartilhamento de Informações de Contexto

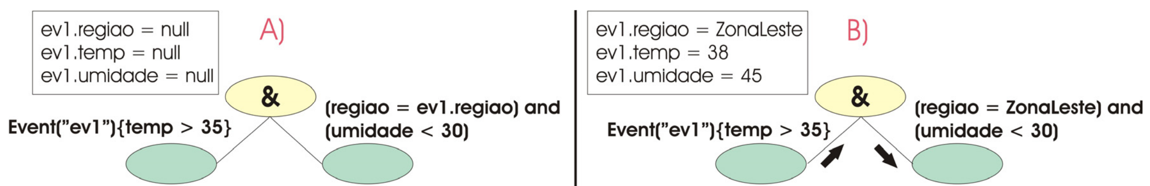

Figura 8: Exemplo de compartilhamento de informações de contexto

Informações contextuais recebidas em uma notificação de uma infra-estrutura *publish-subscribe* podem ser compartilhadas para serem utilizadas em outras subscrições simples envolvidas na mesma subscrição composta. Esse recurso é interessante, pois

permite que informações contextuais recebidas em uma notificação possam ser utilizadas para determinar a própria expressão de composição de eventos. Para dar suporte a esse recurso é necessário que se possa nomear os eventos os quais suas informações contextuais serão utilizadas em compartilhamento no intuito de permitir que uma outra parte da expressão de subscrição possa utilizar essas informações. Para isso ser possível, foi necessário a criação da palavra chave *Event* na linguagem. Essa palavra chave é utilizada em uma expressão no padrão *Event("<nomedoevento>") {<condiçõescontextuais>}*.

A Figura 8 ilustra esse compartilhamento. A parte *A* mostra a subscrição *Event("ev1"){temp>35} & ((região = ev1.regiao) and (umidade < 30))*. O *LeafNode* a esquerda se subscreve a uma infra-estrutura *publish-subscribe* para receber notificação do evento *ev1* e a subscrição do *LeafNode* mais a direita depende da informação de contexto do outro *LeafNode*. Por sua vez, a parte *B* ilustra a notificação do evento no *LeafNode* a esquerda. Nesse momento, como ilustra o quadro com o valor das informações de contexto, são populadas no *DecisionNode* essas informações e repassadas para o *LeafNode* a esquerda, usando assim a informação em sua subscrição a infra-estrutura *publish-subscribe*. Esse exemplo mostra ainda o uso do conector especial *and* no *Leafnode* da esquerda. O uso desse conector não representa uma composição de eventos mas apenas o uso de várias condições em uma única subscrição a um *publish-subscribe*.

5. Mecanismo de Resolução de Exceções Concorrentes

A seção anterior descreveu a arquitetura e os detalhes de implementação do CES. Esta seção descreve como o CES foi estendido para suportar os requisitos de um mecanismo de resolução de exceções. A extensão do mecanismo de composição de evento é, entretanto, apenas uma parte do mecanismo de resolução de exceções. Apesar de possuírem similaridades, tais mecanismos são diferentes por natureza, uma vez que eventos são normalmente utilizados para modelar situações normais do sistema enquanto que exceções modelam condições de erro. A existência de uma condição de erro pode indicar que parte do sistema esteja inoperante. Neste caso, o mecanismo de tratamento de exceções deve ser capaz de garantir que mesmo em tais condições, o tratamento de exceções deve ser executado para garantir o retorno do sistema ao seu estado normal. Dessa forma, as próximas seções descrevem como o mecanismo de composição de eventos foi estendido (Seção 5.2) e como esta extensão é integrada à arquitetura do mecanismo de tratamento de exceções (Seção 5.1).

5.1. Arquitetura Estendida

A introdução do mecanismo de resolução de exceções requereu a extensão da arquitetura descrita em [Damasceno 2006] para a forma como esta ilustrada na Figura 9. No geral, a extensão consistiu na introdução do componente *ContextualExceptionResolution* e na extensão dos componentes *ContextualException* e *HandlingScope*. O componente *ContextualExceptionResolution* foi introduzido entre o middleware *publish-subscribe* e o componente *ContextualException* para gerenciar a utilização simultânea de vários middlewares e para tratar a resolução de exceções concorrentes. Este componente encapsula uma versão do CES que foi estendida para suportar os requisitos peculiares do tratamento de exceções. Por sua vez, o componente

ContextualException é responsável pela especificação de exceções, bem como o gerenciamento das informações relacionadas a ela, tais como nome, descrição, contexto excepcional, e assim por diante. Adicionalmente, este componente interage com o *ContextualExceptionResolution* para levantar as exceções universais, tão logo estas sejam resolvidas. As exceções levantadas devem ser propagadas para seus respectivos escopos.

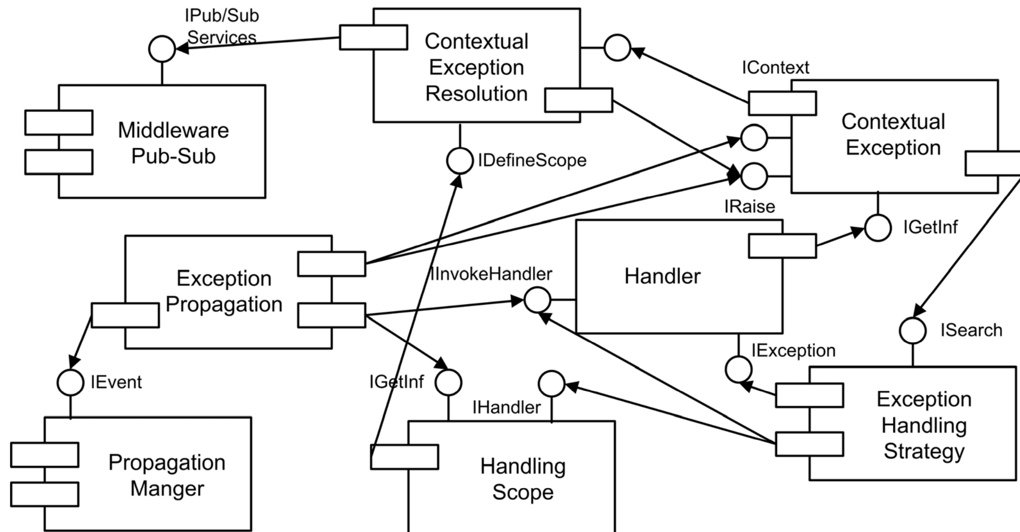

Figura 9: Arquitetura estendida do mecanismo de tratamento de exceções

A propagação de exceções é definida pelos componentes *PropagationManager* e *ExceptionPropagation*. O primeiro componente gerencia o mecanismo de troca de mensagens para garantir que uma determinada exceção propagada seja tratada. Ou seja, este componente atua como elemento que levanta a exceção. Por outro lado, *ExceptionPropagation* atua como o elemento que recebe a exceção. Ele é responsável por receber exceções propagadas por outros dispositivos, recuperar os tratadores locais associados e garantir que depois da execução do tratador tal exceção possa ser novamente propagada. Por sua vez, o componente *Handler* é responsável por especificar os tratadores e realizar as verificações de condições de contexto, necessárias para realizar um tratamento sensível ao contexto.

O componente *ExceptionHandlingStrategy* é responsável por realizar o gerenciamento das exceções e seus respectivos tratadores, mantendo um controle geral desta informação, o qual pode ser acessado por todos os outros componentes. Finalmente, o componente *HandlingScope* permite especificar os escopos de tratamento do mecanismo, gerenciar os tratadores associados aos escopos e recuperar dinamicamente quais os dispositivos que estão associados a um escopo. No sentido de suportar a resolução de exceções, este componente foi estendido para permitir a associação de árvores de resolução de exceções sensíveis ao contexto com cada um dos escopos definidos pela aplicação.

5.2. Abstrações para o Suporte a Resolução de Exceções

A utilização do CES para suportar a resolução de exceções concorrentes permitiu a introdução de novas abstrações (Figura 10) que definem exceções universais e o escopo de resolução destas exceções. Exceções universais são definidas por meio de extensões

da classe *UniversalException* (Linhas 1 - 4). Como descrito na seção 3.2, exceções universais são unicamente levantadas pelo mecanismo de tratamento de exceções quando da resolução da árvore de exceções. Neste caso, a árvore de resolução pode ser definida por meio da utilização de uma expressão de resolução. A linha 5 mostra que a expressão de resolução deve ser passada como parâmetro para o construtor da exceção, neste caso *GeneralException*.

A expressão de resolução pode utilizar todos os operadores definidos pelo CES e ainda a palavra chave *Exception.* Esta palavra chave estende *Event* para permitir a comparação do contexto da exceção levantada como o contexto do dispositivo móvel. As informações de contexto de uma exceção podem ser definidas por meio da sintaxe *Exception("<nomedaexcecao>"){<condições>}*, onde *nomedaexcecao* indica o nome da exceção levantada. O atributo *condições* é opcional e pode ser utilizado para definir outras condições para a resolução desta exceção. A Figura 10 ilustra três exemplos de expressões de resolução que podem ser utilizadas para definir a exceção universal *GeneralException*.

A primeira expressão define que a exceção universal *GeneralException* só será levantada se duas outras exceções *FireAlarm* e *SmokeAlarm* também forem levantadas. A comparação entre o contexto da exceção e o contexto do dispositivo é exemplificada pela expressão *B*. Neste caso, a exceção universal só será levantada se ocorrer o levantamento da exceção *FailToStartHeater* e se a temperatura do local no qual a exceção foi levantada for menor que 15 graus. Finalmente, a última expressão ilustra que o contexto da exceção pode ser utilizado para definir a própria expressão de resolução. Neste caso, a expressão define que a primeira exceção *FailToStart* irá definir quanto tempo uma outra exceção deve ser levantada para que isso seja caracterizado como uma exceção universal.

```
1   public class GeneralException extends UniversalException{
2        public String regiao;
3        . . .
4    }
5  GeneralException uniExc = new GeneralException(ExpressaoResolucao);
6  RegionScope regionScope =  RegionScope.getInstance();
7  regionScope.attachUniversalException(uniExc);

//Possíveis Expressões de resolução
A ="(Exception('FireAlarm')) & (Exception('SmokeAlarm'))"
B ="(Exception('FailToStartHeater')) & ((FailToStartHeater.regiao =regiao)and(temp<15))"
C ="(Exception('FailToStart')) |FailToStart.timeRetry| (Exception('FailToStart'))"
```

Figura 10: Novas abstrações para suportar resolução de exceções.

6. Trabalhos Relacionados

Essa seção apresenta duas comparações importantes. A primeira é a comparação do CES com outros mecanismos que tratam composição de eventos. A segunda é uma comparação entre a extensão proposta nesse trabalho, do mecanismo de resolução de exceções concorrentes, com outros mecanismos de mesma natureza.

6.1 Comparação entre o CES e Outros Mecanismos de Eventos Compostos

Os critérios de comparação desta seção são: expressividade da linguagem de composição de eventos com relação aos conectores suportados, possibilidade de uma

subscrição composta envolver mais de uma infra-estrutura *publish-subscribe*, e suporte ao compartilhamento de informações de contexto em uma subscrição. A Tabela 1 mostra as diferenças relacionadas a esses critérios entre o CES, EPS [Moreto 2001] e PADRES [Li 2005].

Com relação a expressividade da linguagem de composição, algumas vezes os mecanismos utilizam nomes diferentes para a mesma funcionalidade. Todos implementam o *And* e o *OR*. O *Iteractor* do CES é bastante semelhante ao *Repetition* do PADRES. Já o *TimesEQ* e *TimesDF* tem uma semântica um pouco diferente. O *TimeEq* considera a repetição de um evento com o mesmo valor para a informação contextual deste evento e o *TimesDf* considera a mesma repetição só que o valor da informação contextual em cada repetição deve ser diferente. O *Sequence* do PADRES define o tempo em que dois eventos, em seqüência da esquerda para a direita, devem acontecer, desse modo implementando características do *Interval* e do *Sequential* do CES. Adicionalmente o CES ainda possui o *Parallel* que indica que dois eventos devem ocorrer paralelamente.

Tabela 1. Comparação entre mecanismos de composição de eventos.

Mecanismo	Expressividade da linguagem	Suporte a várias infras-estrutura publish-subscribe simultâneo	Compartilhamento de informações de contexto
CES	And, Or, Sequential, Interval, Iteractor and Parallel	Sim	Sim
EPS	And, Or, TimesEq and TimesDf	Não	Não
PADRES	And, Or, Sequence and Repetition	Não	Sim

O suporte a mais de uma infra-estrutura *publish-subscribe* simultânea é uma característica inerente e exclusiva do CES, o qual dá suporte à configuração para se definir em qual mecanismo cada evento simples será subscrito. O EPS, diferentemente dos outros mecanismos analisados, implementa a detecção dos eventos simples e compostos, consequentemente não interagindo com algum *publish-subscribe*.

O compartilhamento de informações contextuais existe no CES e no PADRES. No CES uma subscrição composta usa informações contextuais recebidas em um nó folha da árvore pode ser utilizado na subscrição realizada por outro nó folha desta mesma árvore. O PADRES armazena informações de contexto recebidas anteriormente e o *subscriber* pode utilizar essas informações em suas subscrições através da definição de variáveis.

6.2 Comparação entre Outras Abordagens de Resolução de Exceções Concorrentes

Como descrito na seção 3.2, as árvores de resolução de exceções foram um dos primeiros mecanismos utilizados para resolver exceções concorrentes em sistemas distribuídos. Entretanto, novas abordagens foram introduzidas para estender tal mecanismo. Arche [Issarny 2001] define uma estratégia que permite associar uma função de resolução a um procedimento executado paralelamente (*multiprocedures*). Neste caso, a função de resolução é invocada sempre que todas as invocações paralelas terminam sua exceção. Isto permite que a função de resolução receba como parâmetro

um vetor com todas as exceções levantadas por tal *multiprocedure*. A partir de tal vetor, a aplicação define a estratégia que será utilizada para decidir qual exceção universal será levantada. SaGE [Souchon 2004] estende o mecanismo anterior ao suportar a associação de funções de resolução à serviços e papéis (*roles*) em um sistema *multi-agentes*. Funções de resolução são invocadas no SaGE sempre que uma nova exceção é levantada. AmbientTalk [Mostinckx 2006] provê a construção *group-resolve* que se assemelha a desenvolvida por Arche. Neste caso, a construção *group* é utilizada para agrupar várias invocações assíncronas, enquanto que *resolve* define uma função de resolução que será invocada para tratar um vetor de exceções geradas a partir das exceções levantadas no bloco de execução *group*.

Na comparação com a abordagem descrita neste artigo, as soluções descritas anteriormente são limitadas à utilização de uma função de resolução. O problema de tal função é que esta só é invocada quando exceções são levantadas. Isto implica em dizer que o desenvolvedor terá que gerenciar a subscrição das condições contextuais ao mesmo tempo em que tenta resolver as exceções levantadas. Como tais eventos são geralmente assíncronos, o desenvolvedor terá que implementar uma infra-estrutura extra para tentar definir uma semântica de resolução.

7. Conclusões

Esse artigo apresentou um mecanismo para composição de eventos (CES) bem como o uso desse mecanismo no suporte a resolução de exceções concorrentes em aplicações sensíveis ao contexto. As principais características inovadoras do CES são a possibilidade de usar vários mecanismos *publish/subscribe* como infra-estrutura subjacente e a provisão de uma linguagem de definição de exceções universais com compartilhamento de informações de contexto. Em relação a outros mecanismos para composição de eventos, o diferencial do CES está tanto na expressividade da sua linguagem de composição quanto no suporte a mais de uma infra-estrutura *publish-subscribe* simultânea. Em relação aos outros mecanismos para tratamento de exceções, a expressividade fornecida pelo CES permitiu introduzir novas abstrações que suportam a resolução de exceções concorrentes. Portanto, as contribuições desse trabalho ultrapassam a composição de eventos e incluem também suporte a tratamento de exceções em aplicações sensíveis ao contexto que, tipicamente, caracteriza-se pela ocorrência de vários eventos simultaneamente.

O CES está sendo integrado ao *Context Toolkit* e ao MOCA. Como trabalho futuro pretendemos avaliá-lo em um estudo de caso real e contrastar seu desempenho em cada uma dessas duas infra-estruturas *publish-subscribe*.

Referências

Cacho, N. et al. (2006) “Handling exceptional conditions in mobile collaborative applications: An exploratory case study”, In: *15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE '06,* pages 137-142, England.

Campbell, R. and Randell, B. 1986. “Error recovery in asynchronous systems”, IEEE Trans. Softw. Eng. 12, 8 (Aug. 1986), pages 811-826.

Chakravarthy, S., Mishra, D. (1993) "Snoop: An Expressive Event Specification Language For Active Databases", In: *Technical Report UF-CIS-TR-93-007*, pages 1-26, Dept. of Computer and Information Sciences, Univ. of Florida.

Damasceno, K. et al. (2006) "Tratamento de Exceções Sensível ao Contexto". Anais do 20° Simpósio Brasileiro de Eng. de Software (SBES'2006), Florianópolis, SC, Outubro 2006.

Dey, A. and Abowd, G. (2000) "Towards a Better Understanding of Context and Context-Awareness", In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, p.304-307, September 27-29, 1999, Karlsruhe, Germany.

Gatziu, S. and Dittrich, K. (1994) "Detecting Composite Events in Active Database Systems Using Petri Nets", In *Proc. of the 4th RIDE-AIDS*. (1994)

Gehani, N. et al. (1992) "Event Specication in an Active Object-Oriented Database", In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

Goodenough, J. B. (1975) "Exception handling: issues and a proposed notation", In *Commun. ACM 18, 12 (Dec. 1975)*, p. 683-696.

Issarny, V. (2001) "Concurrent exception handling", In *Advances in Exception Handling Techniques*, A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, Eds. Springer-Verlag New York, New York, NY, 111-127.

Lee, P., Anderson, T. (1990) "Fault Tolerance: Principles & Practice", In *Springer*, 2nd ed, Wien, Austria.

Li, G and Jacobsen, H. (2005) "Composite Subscriptions in Content-based Publish/Subscribe Systems", In *International Middleware Conference*. Grenoble, France.

Mansouri-Samani, M. and Sloman, M. (1997) "GEM: A Generalised Event Monitoring Language for Distributed Systems", In *IEE/IOP/BCS Distributed Systems Engineering Journal 4*, pages 96-108.

Miller, R.; Tripathi, A. (1997) "Issues with Exception Handling in Object-Oriented Systems", In *Proc. ECOOP'97 - Object-Oriented Programming*, p. 85-103, LNCS-1241, Finland.

Moreto, D. and Endler, M. (2001) "Evaluating Composite Events using Shared Trees", In *IEE Proceedings – Software*, Vol. 148, pages 1-10.

Mostinckx, S. et al. (2006) "Ambient-Oriented Exception Handling, In "Advanced Topics in Exception Handling Techniques", C. Dony, J. L. Knudsen, A. Romanovsky, A. Tripathi (eds.), Lect. Not. Comp. Sc. 4119, pp. 141-160, Springer Verlag, 2006

Parnas, D. and Würges, H. (1976) "Response to Undesired Events in Software Systems", In *Proc. 2nd Intl. Conference on Software Engineering*. p. 437-446, California, USA.

Pietzuch P. et al. (2003) "A Framework for Event Composition in Distributed Systems", In *International Middleware Conference*, pages 72-82. Rio de Janeiro, Brazil. June 2003.

Sacramento, V. et al. (2004) "MoCA: A Middleware for Developing Collaborative Applications for Mobile Users", In *IEEE Distributed Systems Online*, v.5, n. 10, October 2004.

Souchon, F. et al. (2004) "Improving exception handling in multi-agent systems", In *Software engineering for multi-agent systems II, Research issues and practical applications*, C. Lucena, A. Garcia, A. Romanovsky, J. Castro and P. Alencar Editors, Springer-Verlag, LNCS 2940.

Tripathi, A., Kulkarni, D. and Ahmed, T. (2005) "Exception Handling Issues in Context Aware Collaboration Systems for Pervasive Computing", In *Proc. ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems*. TR 05-050. LIRMM. Montpellier-II University. 2005. July. France.

Xu, J., Romanovsky, A. and Randell, B. (2000). "Concurrent Exception Handling and Resolution in Distributed Object Systems", In *IEEE Trans. Parallel Distrib. Syst. 11*, 10 (Oct. 2000), pages 1019-1032.

Xu, J. et al. (1995) "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery", In *Proc. 25th Int. Symp. on Fault-Tolerant Computing,* pages 499-508, Pasadena.