3
Trabalhos Relacionados

Neste capítulo, será feita uma breve descrição de alguns trabalhos relacionados ao tema desta pesquisa, que foram selecionados por apresentarem mecanismos de suporte a contratos de objetos ou componentes distribuídos.

A análise dos trabalhos em questão ajudou a identificar aspectos importantes do uso de contratos neste contexto, servindo de base para o estudo desta dissertação e para a implementação do sistema de contratos realizada.

3.1
Tamago

O Tamago [29] é uma plataforma de suporte ao desenvolvimento de componentes com o uso de contratos. Além do tradicional uso de pré/pós-condições e invariantes, possui características de composição e de transição de estados nos seus contratos. Ele utiliza um contêiner para verificar os contratos durante a execução dos componentes.

Além do monitoramento do contrato em tempo de execução, o Tamago realiza a análise estática para verificar a consistência da sua estrutura e do autômato finito usado para modelar os estados do componente, em busca de estados não atingíveis, transições inválidas, assertivas redundantes, entre outros.

Os contratos do Tamago são apresentados em três níveis de granularidade:

– Contratos de serviço: são os que descrevem as interfaces de um serviço oferecido pelo componente, o que poderia ser comparado a um contrato

de classe. Ele possui pré e pós-condições, invariantes e definição de estados do serviço, além das transições entre eles.

- Contratos de componentes: um componente oferece um ou mais serviços para seus clientes, e ao mesmo tempo utiliza os serviços providos por outros componentes. Este tipo de contrato especifica quais serviços são oferecidos e requeridos por um componente para que este possa funcionar corretamente.
- Contratos de *assemblies* e *composites*: os autores denominam de *assembly* a montagem simbólica dos componentes interconectados de uma aplicação, ou seja, a aplicação deve especificar quais são as conexões entre os componentes utilizados. Um *assembly* por default exporta todos os serviços contidos nos componentes usados, enquanto que o *composite* deixa as operações e atributos protegidos, a não ser que estes sejam explicitamente exportados para uso externo.

Quando ocorre a herança de múltiplos serviços, o Tamago realiza a composição dos autômatos envolvidos, unindo os seus estados e transições e cuidando para que o novo autômato seja consistente com a união dos anteriores, eliminando redundâncias e verificando a sua estrutura. A listagem 3.1 ilustra um exemplo de contrato do Tamago.

Em comparação com os demais sistemas de suporte a contratos descritos neste capítulo, o Tamago apresenta um conjunto de características bastante diversificado, pois além da monitoração de contratos em tempo de execução, possui contratos de composição, mecanismos de verificação estática e máquina de estados. Por este motivo, foi incluído neste capítulo como referência para o trabalho em questão.

3.2 Contract-aware CORBA Container

No artigo de Teiniker e outros co-autores [30], é descrita a implementação de um contêiner CCM (*CORBA Component Model*) com extensão para suporte a contratos. Neste projeto, a especificação dos contratos é feita por meio da linguagem OCL (*Object Constraint Language*), estendendo o metamodelo de componentes do CCM.

A verificação em tempo de execução dos contratos é feita pelo contêiner de forma independente do código do componente. A cada operação chamada,

Listagem 3.1: Exemplo de contrato no Tamago

```
service MessageBufferService {
  property int size;
  property bool isEmpty;
  property Message[] messages;
  invariant (#size ≥ 0) ∧ [#isEmpty ⇔ (#size = 0)];
  invariant #messages.length = #size
  Message match(MessageType type) {
      pre type ≠ null;
      post ∃ m ∈ #messages{ m:getType() = type }
          ⇒ return = m
          ∨ return = null
  }
  bool put(Message m) {
      pre m ≠ null;
      post return ⇔ #size@pre = #size + 1;
  }
  bool contains(Message m) {
      post return ⇒ ∃ msg ∈ #messages {
          m.equals(msg)
      }
  }
  Message take() {
      pre ¬ #isEmpty;
      post return ∧ contains(m)@pre
          ⇒ #size@pre = #size − 1;
  }
  behavior {
      init state empty { allow match, put, contains; }
      state notempty { allow all; }
      transition empty to notempty with put when ¬ #isEmpty
      transition notempty to empty with take when #isEmpty
  }
}

service MessageQueueService
extends MessageBufferService { ... }

component MessageQueue
provides MessageQueueService {
    property const int capacity;
    invariant #capacity ≥ #size;
    bool put(Message m) {
        pre #size < #capacity;
    }
    void flush() {
        post #size ≤ #size@pre;
    }
    behavior {
        state notempty { allow flush; }
    }
}
```

as pré e pós condições são testadas, sendo que as invariantes são verificadas apenas nas operações de criação (construtores) e de alteração de atributos (*set*), visando um ganho de desempenho por não monitorar as invariantes nos métodos que apenas consultam o estado do objeto.

O artigo em questão apresenta um exemplo de componente (listagem 3.2) com interfaces criadas para efetuar testes de desempenho do uso de contratos, variando as condições (pré,pós,invariante) e os tipos de dados (*string*,*long*,*LongList*) sendo avaliados. O contrato do componente está na listagem 3.3.

Listagem 3.2: IDL do componente do exemplo

```
typedef sequence<long> LongList;
interface DbcBenchmarkPre {
  void f_in1(in long l1);
  void f_in2(in string s1);
  void f_in3(in LongList ll1);
};
interface DbcBenchmarkInv1 {
  attribute long longAttr;
  void f0();
};
interface DbcBenchmarkInv2 {
  attribute string stringAttr;
  void f0();
};
interface DbcBenchmarkInv3 {
  attribute LongList seqAttr;
  void f0();
};
interface DbcBenchmarkPost {
  long f_ret1();
  string f_ret2();
  LongList f_ret3();
};
component DbcTest {
  provides DbcBenchmarkPre bmPre;
  provides DbcBenchmarkInv1 bmInv1;
  provides DbcBenchmarkInv2 bmInv2;
  provides DbcBenchmarkInv3 bmInv3;
  provides DbcBenchmarkPost bmPost;
};
home DbcTestHome manages DbcTest {};
```

Os testes de desempenho realizados pelos autores mostraram que, na maioria das execuções, o tempo de verificação de contratos ficava muito abaixo do tempo gasto nas requisições remotas dos objetos CORBA. As chamadas que possuem parâmetros muito grandes como, por exemplo, a seqüência de parâmetros de tipo *LongList* apresentada no exemplo da listagem 3.3, demandam um tempo de execução significativo em relação à verificação das pré e pós condições.

Porém, a verificação da invariante que envolve o teste dos elementos de uma seqüência grande (*DbcBenchmarkInv3*) ultrapassa o custo da chamada remota de CORBA. Neste caso, a chamada da interface não possui parâmetros,

Listagem 3.3: Exemplo de contrato de componente em OCL

```
context DbcBenchmarkInv1
    inv i1: self.longAttr >= 0 and longAttr < 1000000
context DbcBenchmarkInv2
    inv i2: self.stringAttr.size >= 0 and stringAttr.size < 1000000
context DbcBenchmarkInv3
    inv i3: self.seqAttr->forAll(i|i >= 0 and i < 1000000)
context DbcBenchmarkPre::f_in1(l1:Integer)
    pre p1: l1 >= 0 and l1 < 1000000
context DbcBenchmarkPre::f_in2(s1:String)
    pre p2: s1.size() >= 0 and s1.size() < 1000000
context DbcBenchmarkPre::f_in3(ll1:Sequence(Integer))
    pre p3: ll1->forAll(i|i >=0 and i < 1000000)
context DbcBenchmarkPost::f_ret1()
    post p1: result >=0 and result < 1000000
context DbcBenchmarkPost::f_ret2()
    post p2: result.size() >= 0 and result.size() < 1000000
context DbcBenchmarkPost::f_ret3()
    post p3: result->forAll(i|i >=0 and i < 1000000)
```

e a invariante sendo testada no objeto percorre uma seqüência com muitos elementos, fazendo com que o tempo de avaliação do contrato seja significativo quando comparado com o tempo gasto na chamada remota.

3.3 CoCoMod / CoCoGen

O trabalho descrito no artigo [31] possui semelhanças com o anterior, pois também usa uma linguagem baseada na OCL para descrever contratos de componentes para a plataforma CORBA. A linguagem é denominada CDL *Constraint Description Language*.

A linguagem apresentada adiciona o suporte a condições de guarda, usadas para os casos em que ações específicas sejam executadas assim que determinadas assertivas tenham sido satisfeitas.

O trabalho também apresenta características de contratos de composição, pois a CDL possui extensões para a agregação e composição de objetos, além de permitir expressar a sua multiplicidade. Embora este tipo de construção esteja no texto do artigo, o exemplo fornecido não as utiliza.

A implementação da verificação das assertivas de contrato baseia-se na alteração do código dos *skeletons* CORBA gerados pelo compilador IDL. A ferramenta *CoCoMod* é um editor gráfico para a modelagem das restrições de contrato para os componentes, enquanto a ferramenta *CoCoGen* é responsável pela inserção das restrições do contrato no código dos *skeletons*.

Do ponto de vista da portabilidade, esta solução não se mostra muito flexível, pois a edição dos *skeletons* implica um acoplamento muito forte em relação à implementação de CORBA subjacente, uma vez que cada implementação possui particularidades com relação ao código gerado pelo compilador de IDL. Isto impõe a necessidade de adaptadores de código para que a funcionalidade de contratos esteja disponível nas diversas combinações de sistemas de componentes, sistemas operacionais e linguagens de programação.

3.4 CIPV

Este trabalho [32] propõe um modelo de contratos que visa eliminar parte da dificuldade existente na especificação mais formal de restrições, devido à resistência de muitos desenvolvedores a adotar técnicas que exijam um maior rigor matemático, seja por desconhecimento ou por simplificação.

Para atingir uma maior audiência de profissionais com este perfil, a abordagem foi baseada na premissa de que não deveria ser necessário um treinamento em linguagens e métodos de especificação formal para usar o modelo de contratos proposto. Outro requisito do modelo foi permitir a facilidade de adaptação dos contratos às alterações inevitáveis nos componentes desenvolvidos.

O trabalho descreve a implementação de contratos comportamentais através de uma linguagem denominada SPS (*Specification Patterns System*), que permite a especificação de contratos de interação entre os componentes. A linguagem procura "esconder" a sintaxe de uma linguagem declarativa formal através de construções que procuram reproduzir a linguagem natural. Ela possui elementos de contratos de pré e pós-condições, bem como de transições de estados entre chamadas.

Os contratos de interação são representados em dois níveis: *peer level* e *component level*. O primeiro caracteriza as interações entre "vizinhos", ou seja, componentes conectados. O segundo nível não se restringe a componentes conectados, caracterizando as chamadas em âmbito geral. Ou seja, os de tipo *peer* permitem especificar a seqüência relativa entre as chamadas, pois possuem um estado associado. A implementação da ferramenta denominada CIPV (*Component Interaction Property Validator*) é baseada na plataforma CORBA e no uso de *Portable Interceptors* para ter acesso às informações

necessárias à validação em tempo de execução dos contratos especificados em SPS.

A listagem 3.4 contém parte de um contrato do CIPV para um componente de leilão, definido no artigo citado. Ele define as seqüências válidas das chamadas e as condições para as transições de estados do componente.

Listagem 3.4: Exemplo de contrato do CIPV

```
component Auctioneer {
  provides IAuctioneer;
  requires IBidder;
  interaction-contract {
    peer-level {
      wannaBid exists only after register
        until unregister;
      wannaBid precedes youGotIt
        where wannaBid.refNo = youGotIt.refNo
          && wannaBid.result = true;
    ...}
    component-level {
      wannaBid leads to itemSold
        where wannaBid.refNo = itemSold.refNo
          && wannaBid.result = true;
    ...}
  }
}
```


3.5 Considerações Finais

Os trabalhos descritos neste capítulo possuem características distintas e complementares entre si. Dentre elas, destacam-se o uso de contratos específicos para componentes, mecanismos de controle de estados e interação entre componentes.

Os trabalhos apresentados influenciaram o projeto do estudo de caso implementado nesta dissertação, principalmente com relação à definição de estados nos contratos e na criação de contratos de facetas de componentes.

Porém, nos artigos analisados, não há maiores detalhes sobre alguns aspectos relevantes no contexto de contratos em ambientes distribuídos como concorrência, avaliação distribuída de contratos, herança, efeitos colaterais e contratos de sincronização. O trabalho realizado nesta dissertação procurou apresentar algumas contribuições na discussão destes tópicos.