4
Estudo de Caso: Contratos em Lua

Este capítulo descreve o sistema de suporte a contratos criado como estudo de caso para permitir a avaliação dos principais aspectos do uso de contratos em componentes de software, conforme a proposta desta dissertação. O sistema em questão foi denominado *LUC*.

O principal objetivo do desenvolvimento do sistema detalhado ao longo deste capítulo foi obter uma forma de experimentar o uso de contratos com objetos distribuídos baseados na plataforma OiL/CORBA e componentes SCS, sendo ambos projetos do grupo de pesquisa em Sistemas Distribuídos da PUC-Rio.

O fato de tanto o OiL como o SCS terem sido desenvolvidos em Lua determinou que a linguagem fosse também escolhida para a implementação deste projeto. A flexibilidade e extensibilidade de Lua permitiram o desenvolvimento da solução de forma rápida quando comparada a outras linguagens, uma vez que o sistema apresenta características de difícil implementação em linguagens com poucos recursos dinâmicos e de introspecção.

As seções a seguir apresentam as principais características do sistema, sua arquitetura, modo de funcionamento, alguns detalhes de implementação, exemplos, além de limitações e dificuldades encontradas.

4.1
Características

Nesta seção, serão apresentadas as principais características da ferramenta, para dar uma visão geral do sistema e permitir uma boa compreensão dos exemplos e experimentos contidos nas seções subseqüentes.

4.1.1
Suporte a objetos Lua/OiL e Componentes SCS

Embora a linguagem Lua não possua internamente os conceitos de classe e interface, é possível aplicar o paradigma de programação orientada a objetos por meio dos seus recursos [4], ou através do uso de bibliotecas de extensão como a LOOP [33], que serve de base para o OiL e o SCS. Assim, foi implementado no LUC o suporte a contratos para objetos Lua convencionais e para objetos OiL/CORBA.

Para objetos Lua, como o conceito de interface ou classe de um objeto não é padronizado, a associação de um contrato a uma interface deve ser feita explicitamente pelo programador. Neste caso, a interceptação das chamadas ao objeto é baseada no uso de um *proxy* dinâmico, que repassa as chamadas para o objeto original e avalia as condições do contrato. Uma alternativa para o uso de objetos Lua seria o suporte específico a classes da biblioteca LOOP, o que pode fazer parte de uma extensão futura deste trabalho.

Para objetos criados no OiL, o LUC intercepta as chamadas ao objeto e, através da identificação da sua interface CORBA, é capaz de decidir qual contrato se aplica a este objeto, uma vez que a interface tenha sido previamente associada a algum contrato. Desta forma, o programador não precisa especificar o contrato para cada objeto criado, apenas para as interfaces.

Quando há herança de interface no OiL, o sistema também é capaz de identificar que o objeto faz parte de uma hierarquia e automaticamente recupera os contratos das interfaces superiores (se existentes) e os inclui na verificação sendo realizada.

O suporte a contratos de componentes SCS foi implementado sobre o tratamento de chamadas no OiL, uma vez que os componentes SCS são formados por objetos CORBA. A diferença reside no fato de que o contrato de um componente envolve um nível mais alto de abstração, pois especifica condições sobre as suas facetas e receptáculos. Facetas são os serviços oferecidos por um componente, na forma de objetos que têm uma interface definida e um nome associado. Um componente pode ter também um ou mais receptáculos, que são os pontos de conexão de objetos que representam suas dependências externas. Assim como as facetas, os receptáculos possuem um nome e interface associados, com a diferença que a interface se refere ao objeto externo que será conectado. Um receptáculo pode ser simples ou múltiplo, o que indica se permite a conexão de um ou vários objetos. A figura 4.1 ilustra um componente

SCS, com suas facetas, receptáculos e interfaces padrão do modelo.

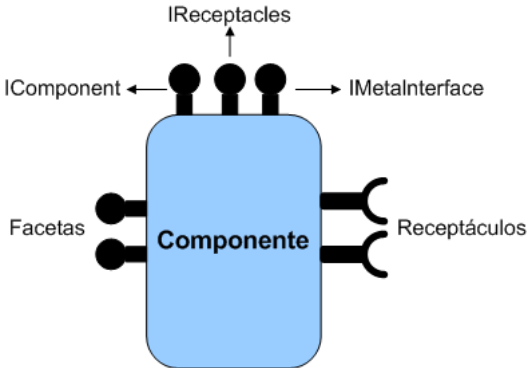

Figura 4.1: Componente SCS

As interfaces *IComponent*, *IReceptacles* e *IMetaInterface* compreendem o modelo básico de componentes do SCS. A interface *IComponent* é a única obrigatória e serve de ponto de partida para a obtenção das facetas e receptáculos de um componente. A interface *IReceptacles* oferece os mecanismo de conexão e desconexão de objetos aos receptáculos, além de métodos para a obtenção das conexões existentes. Já a interface *IMetaInterface* implementa o mecanismo de reflexão de um componente, pois fornece informações sobre as facetas e receptáculos disponíveis. A definição das interfaces do modelo do SCS encontra-se listada no apêndice A.

O contrato de um componente SCS especifica os nomes e interfaces das facetas que devem constar na sua implementação. Como cada faceta está associada a uma interface CORBA, para cada uma deve ser definido o contrato da interface correspondente. Caso o contrato da faceta não seja definido dentro do contrato do componente, se houver um contrato genérico para a interface em questão ele passa a ser utilizado nas chamadas ao objeto da faceta. Isto permite uma configuração mais flexível para o sistema através da especialização do contrato de uma interface para um determinado componente.

Para cada receptáculo, o contrato de componente define o seu nome e as interfaces dos objetos a serem conectados. Como a chamada para a conexão do receptáculo recebe um tipo CORBA *Object* genérico, com o contrato é possível aumentar o nível de detalhe da especificação, além de permitir a verificação em tempo de execução da compatibilidade da interface do objeto conectado com a interface esperada.

4.1.2 Linguagem de Contratos

Além do seu uso na implementação do LUC, a linguagem Lua também foi escolhida para especificar os contratos. Na verdade, a linguagem de especificação de contratos é um subconjunto da linguagem Lua, uma vez que as assertivas devem ser expressões booleanas e não podem conter estruturas de controle como *while*, *for* ou *if*.

Como visto no capítulo 2, algumas soluções de contratos adotam linguagens de natureza declarativa baseadas na OCL ou similar. Porém, o uso de Lua para especificar os contratos mostrou-se adequado numa análise preliminar, o que simplificou tanto a criação da ferramenta como o seu uso. A opção por Lua foi definida em função da sua flexibilidade e simplicidade, dispensando a necessidade de se criar ou aprender uma outra linguagem específica para a elaboração dos contratos.

Algumas funcionalidades foram adicionadas à linguagem de contratos para permitir operações comuns a sistemas desta natureza. A tabela 4.1 lista as funcionalidades básicas incluídas na linguagem, que serão descritas ao longo deste capítulo e nos experimentos do capítulo 5.

Extensão	Descrição
PRE	Usada na pós-condição, retorna o valor prévio de uma expressão (anterior à execução da operação)
RESULT	Resultado da operação realizada, referenciado na pós-condição
FORALL	Quantificador lógico "para todos"
EXISTS	Quantificador lógico "existe"
IMPLIES	Implicação lógica
EXCEPTION	Indica se ocorreu uma determinada exceção durante a execução operação (usada na pós-condição)
PCALL	Executa uma operação em modo "protegido" de Lua, para tratamento de erros
EVAL	Permite a avaliação de código Lua em assertivas

Tabela 4.1: Extensões da linguagem de contratos

Além das extensões básicas, foram incluídas outras funcionalidades voltadas para a avaliação de contratos no SCS e no OiL. Elas tratam, principalmente, de questões relacionadas ao uso de facetas, receptáculos e interfaces de objetos CORBA. A tabela 4.2 contém a descrição destas extensões específicas para objetos OiL e componentes SCS.

A tabela *FACETS* é indexada pelo nome da faceta, e cada elemento corresponde a uma outra tabela com os campos listados na tabela 4.3.

Extensão	Descrição
FACETS	Tabela de facetas de um componente SCS
RECEPTACLES	Tabela de receptáculos de um componente SCS
IS_A	Função que verifica se um objeto implementa uma determinada interface
INTERFACEID	Retorna o ID CORBA da interface de um objeto

Tabela 4.2: Extensões da linguagem de contratos para componentes SCS

Item	Descrição
name	Nome da faceta
interface_name	Nome da interface da faceta
facet_ref	Referência para o objeto que implementa a faceta

Tabela 4.3: Campos de descrição das facetas na tabela *FACETS*

De forma análoga, a tabela *RECEPTACLES* é indexada pelo nome do receptáculo associado, e cada entrada possui uma tabela com os campos descritos abaixo.

Item	Descrição
name	Nome do receptáculo
interface_name	Nome da interface do objeto a ser conectado
interface_id	Id CORBA da interface do objeto a ser conectado
numConnections	Número de objetos conectados no receptáculo

Tabela 4.4: Campos de descrição dos receptáculos da tabela *RECEPTACLES*

4.1.3 Especificação de Contratos

A especificação de contratos em alguns sistemas é feita diretamente no código fonte da implementação, na forma de comentários ou usando alguma outra sintaxe estendida. No caso do LUC, uma possibilidade seria especificar as assertivas dos contratos em comentários dos arquivos IDL, que especificam as interfaces CORBA. Porém, era necessário obter uma forma independente de CORBA para especificar os contratos, já que deveria ser possível associá-los a objetos Lua convencionais, componentes SCS ou mesmo a outros tipos de objeto que não tenham suas interfaces especificadas em arquivos IDL.

Assim, a forma escolhida para a especificação do contrato de uma interface foi a criação de um arquivo à parte com conteúdo em Lua, que deve ser incluído e interpretado pelo sistema de contratos. Este formato é independente do mecanismo de chamada, ou seja, pode ser usado para aplicar contratos a objetos simples, objetos OiL/CORBA, componentes SCS ou mesmo a outras tecnologias de chamada remota de objetos que venham a ser suportadas.

Uma vantagem de permitir a especificação do contrato em comentários no próprio arquivo IDL seria facilitar a sua criação e manutenção, devido à proximidade das informações relacionadas. Este benefício poderia ser obtido por meio de uma ferramenta que extraísse as assertivas de comentários do arquivo IDL e fizesse a conversão para o formato adotado na solução. Outra opção seria criar uma ferramenta que lesse o arquivo IDL e o contrato e criasse uma interface única que apresentasse as informações de forma integrada.

No formato escolhido, a definição de um contrato ocorre por meio de uma chamada à função *interface*, localizada no módulo *luc.contract*. A listagem 4.1 ilustra parte da definição de um contrato para uma interface denominada *Stack*.

Listagem 4.1: Exemplo de contrato no LUC

```
luc.contract.interface
{
    __interface = "::stack::Stack",

    __inv = {
        inv = {
            non_negative_size = [[ self:size() >= 0 ]]
        },
    },
    top = {
        pre = {
            not_empty = [[ self:size() > 0 ]]
        },
        post = {
            result_at_top = "RESULT == self:itemAt(self:size())"
        }
    },
...
}
```

Embora o código do exemplo tenha a aparência de um arquivo de configuração simples, ele é na verdade um *script* Lua em que cada chamada à função *luc.contract.interface* estabelece uma nova definição de contrato para a interface especificada pela chave *__interface*. A função recebe uma tabela como único parâmetro, sendo que Lua neste caso dispensa o uso de parênteses na chamada. A tabela é composta por chaves e valores que, em alguns casos, são outras tabelas aninhadas, sempre delimitadas por { e }. Vale lembrar também que, em Lua, as *strings* podem ser delimitadas por aspas simples, duplas ou pelos marcadores [[e]].

Além da chave com o nome da interface, existem outras reservadas para a definição do contrato, listadas na tabela 4.5. O uso de cada item da tabela será visto ao longo deste capítulo.

As demais chaves do primeiro nível da tabela (e.g. *top*) representam nomes de operações da interface. A cada chave deste tipo está associada uma

Chave	Valor
__interface	Nome da interface associada ao contrato
__inv	tabela que define a invariante da interface
__state	tabela que define os estados e transições da interface
__sync	tabela com as informações de sincronização de operações
__functions	tabela com as funções criadas pelo usuário que podem ser usadas nas assertivas

Tabela 4.5: Chaves reservadas na definição de um contrato de interface

tabela com as chaves *pre* e *post*, que definem as pré e pós-condições da operação em questão, respectivamente. Associadas a elas, por sua vez, encontram-se tabelas nas quais cada entrada corresponde a um par do tipo (rótulo,assertiva). A assertiva é uma *string* contendo uma expressão booleana a ser avaliada. O rótulo identifica a semântica da assertiva, assim como na linguagem Eiffel, e é usado na notificação da violação para facilitar a identificação da sua causa.

Além do uso de *strings* nas assertivas, é possível também especificar uma função booleana em Lua para avaliar uma condição mais complexa, o que confere maior extensibilidade ao mecanismo de contratos, porém diminui a sua robustez em relação a erros ou efeitos colaterais, já que uma função pode conter código complexo também sujeito a defeitos, como visto no capítulo 2.

O resultado da avaliação das invariantes ou pré e pós-condições é a conjunção das assertivas (ou funções) especificadas nas tabelas correspondentes à operação sendo chamada. Caso uma das avaliações falhe, o mecanismo de notificação será acionado para informar a violação do contrato.

Na definição do contrato, algumas verificações estruturais e sintáticas são realizadas, alertando o usuário em caso de erros. Porém, nenhuma inferência lógica ou verificação estática é realizada sobre o conteúdo das assertivas.

4.1.4 Quantificadores Lógicos

Como visto na subseção 2.3.1, segundo Meyer, a inclusão de quantificadores universais e existenciais na linguagem de especificação de contratos não é imprescindível, pois eles não são suficientemente expressivos em alguns casos, podendo ser substituídos por funções. Porém, em geral eles oferecem uma funcionalidade bastante útil, pois eliminam a necessidade de se construir uma função sempre que for preciso verificar uma assertiva sobre coleções.

Na implementação do LUC, foram criadas as funções *FORALL* e *EXISTS*

para avaliar uma assertiva sobre uma coleção de elementos, que em Lua é representada por uma tabela. No caso de um método de objeto CORBA, estas funções se aplicam a seqüencias de tipos básicos ou objetos, que no mapeamento para Lua são representadas por tabelas de índices inteiros (*arrays*). As funções iteram na tabela em questão, avaliando a assertiva dada e interrompendo o processo na primeira falha, no caso do *FORALL*, ou no primeiro sucesso, no caso de *EXISTS*.

A sintaxe para o uso dos quantificadores lógicos é:

```
{FORALL | EXISTS} (var, tab, expr)
```

Onde:

- *var*: *string* contendo o nome da variável usada na expressão *expr* que representa o elemento corrente da tabela *tab*
- *tab*: tabela contendo os elementos a serem verificados na expressão *expr*
- *expr*: assertiva a ser verificada para cada elemento da tabela *tab*

Para ilustrar a sintaxe dos quantificadores, seja $f(v)$ uma função que recebe como parâmetro um vetor de inteiros v cujos elementos devem ter valores entre 1 e 10. Para expressar esta condição numa assertiva, é possível escrever:

```
FORALL( 'x', v, 'x >= 1 and x <= 10' )
```

Algumas variações destes quantificadores foram criadas para explorar o paralelismo em requisições de objetos remotos, para tentar aumentar o desempenho nestes casos. Maiores detalhes serão apresentados na seção 4.3.

Além destes quantificadores, foi criada também uma função *IMPLIES(a,b)*, para representar a expressão lógica de implicação $a \Rightarrow b$. Na verdade, ela é apenas uma forma mais inteligível de escrever a expressão equivalente (*not a or b*). Por um lado, ela facilita o trabalho de quem escreve o contrato, pois em geral oferece uma melhor expressividade. Porém, por se tratar de uma função, ela não permite tirar proveito da avaliação em curto circuito da expressão booleana equivalente, pois as expressões têm que ser avaliadas antes da chamada ser efetuada.

4.1.5 Transição de Estados

Para auxiliar na modelagem do comportamento de uma interface ou componente, foi introduzido o suporte a transições de estados, que permite especificar as

seqüências válidas de operações. Esta é uma característica também presente em alguns dos trabalhos citados no capítulo 3.

O funcionamento da máquina de estados é baseado na especificação de uma tabela contendo os estados possíveis, o estado inicial, e uma outra tabela com as regras para as transições de estado. O exemplo da listagem 4.2 ilustra o mecanismo de transição de estados do LUC.

Listagem 4.2: Exemplo de transições de estado em contrato

```
__state = {
  states = {
    "empty",
    "not_empty"
  },
  initial_state = "empty",
  transitions = {
    empty = {
      push = { condition="true", state="not_empty" },
      pop = {},
      top = {},
      itemAt = {},
    },
    not_empty = {
      pop = { condition="self:size() == 0", state="empty" },
    }
  }
}
```

O exemplo da listagem 4.2 faz parte do contrato da listagem 4.1, sendo *__state* a chave reservada para a especificação das transições de estado da interface. A tabela 4.6 descreve as chaves reservadas para a descrição dos estados e transições associadas.

Chave	Valor
states	tabela com os possíveis estados para a interface
initial_state	*string* contendo o estado inicial. Se omitido, vale o primeiro da tabela *states*
transitions	tabela contendo uma chave por estado, sendo os valores tabelas descrevendo as transições possíveis

Tabela 4.6: Conteúdo da tabela *__state*

A tabela *transitions* contém uma chave para cada estado, associada a uma outra tabela que descreve as transições possíveis a partir daquele estado. Nesta outra, as chaves são os nomes de operações da interface, associadas a valores de tabela contendo duas informações: *condition*, uma expressão booleana, e *state*, o nome do novo estado. Juntas, essas informações são usadas da seguinte maneira: se no estado atual, a operação foi realizada com sucesso e a condição *condition* é verdadeira, então ocorre uma transição para o estado *state*.

No exemplo dado, se no estado *empty* a operação *push* for chamada com sucesso, sempre ocorre a transição para o estado *not_empty*, pois *condition='true'*. Já as operações *pop*, *top* e *itemAt* não podem ser chamadas neste estado, pois não há elementos na pilha. Por isso, estão associadas a tabelas vazias, o que é interpretado

pelo LUC como uma indicação de que esta chamada não pode ocorrer no estado associado. Neste caso, a chamada seria tratada como uma violação do contrato. Por outro lado, caso uma operação não seja especificada na tabela do estado, ela é sempre permitida e não ocasiona nenhuma transição.

4.1.6 Sincronização de Operações

Para efetuar experimentos com contratos de nível 3, foi criado um mecanismo básico de contratos de sincronização que permite especificar relações de exclusão mútua entre as operações de uma interface, ou seja, para cada objeto, as operações especificadas como mutuamente exclusivas no contrato da interface só podem ser executadas caso nenhuma outra esteja em andamento. Caso isto ocorra, a *thread* que tentou executar a operação é suspensa e fica aguardando o término de uma chamada para que possa novamente tentar obter a permissão de execução.

Em [20], Bertrand Meyer argumenta que quando há concorrência envolvida, as pré-condições de um contrato assumem uma semântica de sincronização, ao invés de apenas estabelecer os critérios de correção do software. Portanto, no contexto de componentes de software distribuídos, a sincronização possui um papel importante para a especificação dos contratos. Assim, além da exclusão mútua entre operações, o contrato no LUC também permite especificar pré-condições de sincronização para uma interface.

As condições de sincronização são definidas através da chave *__sync* na tabela do contrato, sendo associada a uma tabela contendo a chave *__mutex* e outras chaves com o nome das operações. À chave *__mutex* associa-se uma tabela contendo os nomes das operações mutuamente exclusivas. As chaves de operações devem ser associadas a tabelas com a chave *pre*, que por sua vez é associada a uma *string* contendo a expressão booleana que deve ser satisfeita para que a operação possa ser executada. A listagem 4.3 ilustra um exemplo de contrato de sincronização.

As funcionalidades de sincronização estão disponíveis apenas para objetos OiL/CORBA, pois utilizam o suporte a *threads* disponível no OiL, que é baseado no mecanismo de co-rotinas de Lua. Assim, o suporte a sincronização do LUC utiliza as primitivas do escalonador do OiL para suspender ou restaurar uma *thread*, conforme o caso. A alternância entre as *threads* ocorre, em geral, quando estas invocam alguma operação bloqueante, que envolvam requisições remotas, suspensão, ou espera por algum resultado de operação.

É importante ressaltar que o mecanismo de controle de sincronização é bastante simplificado, pois existem diversas implicações relacionadas à concorrência que ainda

Listagem 4.3: Exemplo de contrato de sincronização

```
luc.contract.interface
{
    __interface = "::demo::Buffer",
    __sync =
    {
        __mutex = {
            {'put','get'}
        },
        put = {
            pre='not self:full()',
        },
        get = {
            pre='not self:empty()',
        }
    },
}
```

não são tratados pelo sistema, e que serão descritos na seção 4.6. Ainda assim, existem diversos casos em que este tipo de mecanismo, ainda que com limitações, permite expressar no contrato as condições de sincronização necessárias para que o software funcione da forma esperada, como será visto em alguns experimentos no capítulo 5.

Os contratos de sincronização possuem uma característica peculiar em relação aos contratos comportamentais, pois deixam de ser "opcionais" para serem uma peça fundamental para o correto funcionamento do objeto ou componente envolvido. Isto se deve ao fato de que, mais do que verificar se o software funciona de acordo com a sua especificação, os contratos de sincronização *definem* e *implementam* as condições para o funcionamento do objeto ou componente em ambiente de concorrência. Portanto, se por um lado os contratos comportamentais podem ser "desligados" ao se constatar um nível razoável de correção do software em questão, o mesmo não se aplica a contratos de sincronização.

4.1.7 Avaliação de Contratos no Cliente

A chamada de operações de componentes distribuídos pode envolver a serialização (*marshalling*) de argumentos de tamanho significativo como, por exemplo, vetores com muitos elementos, conforme ilustrado na subseção 3.2. Além disso, a avaliação da pré-condição sobre parâmetros com essas características pode envolver um processamento intenso no servidor, ainda mais se este tiver de lidar com muitas requisições simultâneas.

Para melhor escalabilidade da avaliação de contratos nessas condições, foi implementado um mecanismo de avaliação de contratos no cliente, que pode ser aplicado no caso de chamadas remotas a objetos do OiL. Desta forma, o uso de interceptadores no cliente permite que o contrato seja tratado antes mesmo que a

chamada deixe o computador de origem. Caso alguma violação seja detectada ainda no cliente, ela é notificada o mais rapidamente possível e a chamada deixa de ser feita, sendo que o servidor nem recebe a requisição. Com isso, reduz-se o processamento no servidor e o consumo de banda de rede.

Para que a avaliação de um contrato ocorra ainda no cliente, o ideal é que este não contenha chamadas ao próprio objeto sendo invocado, pois neste caso não há como evitar a chamada remota para que a assertiva seja verificada, o que pode impôr um custo ainda maior para a avaliação.

Para habilitar a verificação de contratos no cliente de um objeto remoto, é necessário definir o valor de uma variável que indica o tipo de interceptador OiL que deve ser instalado pelo LUC. Assim, é necessário incluir a atribuição `inteceptor_type='client'` na linha de comando do interpretador Lua ou no programa sendo carregado, antes da inclusão do módulo *interceptor.lua*. Mais detalhes sobre o mecanismo de carga dos módulos do LUC encontram-se na seção 4.3.

Este recurso, se usado em combinação com os quantificadores lógicos que avaliam as assertivas em paralelo pode representar uma redução de tempo na verificação do contrato, pois não sobrecarrega o servidor e pode reduzir as chamadas remotas, principalmente quando os objetos passados como parâmetro estão mais próximos do cliente.

Cabe lembrar que, ao avaliar o contrato no cliente, o mecanismo de sincronização de chamadas não deve ser utilizado, pois a sua verificação de forma distribuída não foi implementada e, portanto, não irá funcionar corretamente.

4.1.8 Notificação de Violações

A notificação de violações é o mecanismo que permite ao usuário do sistema tomar conhecimento de que alguma condição definida no contrato não foi satisfeita, para que uma medida preventiva ou corretiva possa ser tomada.

Conforme visto na subseção 2.3.3, existem algumas variações para o comportamento de um sistema de contratos quando ocorre uma violação. No caso do LUC, as seguintes ações podem ocorrer:

- Exceção: o sistema gera uma exceção no cliente ou servidor, dependendo do tipo de avaliação sendo feita;
- Log: uma entrada é adicionada ao arquivo de log do sistema;

– Delegação: uma chamada externa é feita a um objeto que se registra no LUC com essa finalidade, como um observador, e que irá tomar as ações necessárias para a notificação da violação.
– Bloqueio: ocorre em contratos de sincronização quando a condição em questão não é satisfeita.

Com relação ao mecanismo de delegação, o objeto que será responsável por processar a notificação deve implementar a interface *Observer* apresentada na listagem 4.4 e registrar-se junto ao objeto do sistema que implementa a interface *Observable*, responsável pelo envio de notificações. Esse registro é feito através da chamada ao método *attach*.

A cada violação, é feita uma chamada ao método *update* de todos os observadores registrados, e é passada uma estrutura *Notification* contendo as informações correspondentes.

Listagem 4.4: Interface de notificações do LUC

```
module luc {

   struct Notification {
      string ifname;
      string operation;
      string checkType;
      string label;
   };

   interface Observer {
      void update(in Notification notif);
   };

   interface Observable {
      long attach(in Observer obs);
      void detach(in long id);
   };

};
```


4.1.9 Herança de Interfaces

A linguagem IDL permite a herança na especificação de interfaces. Portanto, foi necessário implementar na solução o suporte à hierarquia de interfaces na avaliação de contratos. Durante a verificação de uma operação, o sistema consulta o repositório de interfaces do OiL para determinar se existem interfaces superiores na hierarquia que possuam contratos associados. Se houver, estes contratos são levados em conta no processo de avaliação da operação.

Assim como em Eiffel, o suporte a herança de interfaces é baseado no conceito de "correção fraca" de uma classe, conforme visto na subseção 2.3.5. Durante a

avaliação das assertivas, é feita a disjunção das pré-condições da interface em questão com as das interfaces superiores, enquanto que para as pós-condições e invariantes é feita a conjunção.

Quando o contrato da operação não é definido na interface sendo avaliada e é definida na interface base, esta última é usada para verificar o contrato. O oposto também é verdade, ou seja, caso só haja assertivas na interface derivada, a assertiva considerada para a base é "true". Quando ambas possuem assertivas, vale a regra da disjunção e conjunção do parágrafo anterior.

O ideal para a verificação das pré-condições seria a abordagem da "correção forte", em que as pré-condições da interface base devem implicar logicamente as da interface derivada, o que tornaria a avaliação de contratos mais robusta no sentido de garantir a correção da especificação sob a ótica da teoria de *behavioral subtyping*. Para obter esta funcionalidade, seria necessário implementar verificadores formais para provar as implicações lógicas envolvidas.

Além disso, na implementação atual, a herança do contrato de interfaces não contempla a fusão das máquinas de estado que modelam o seu comportamento, o que limita o uso nestes casos. Para contornar este problema, é necessário reescrever a descrição dos estados de acordo com o subtipo criado. O mesmo ocorre com o mecanismo de sincronização de operações, pois a hierarquia ainda não é levada em conta neste caso. A seção 4.6 aborda estas e outras limitações com maiores detalhes.

4.1.10 Hierarquia de Contratos

Há dois tipos de contratos que podem ser definidos no LUC: de interface e de componente. Para contratos de componentes, a definição da interface de uma faceta permite especificar um contrato diferente daquele que foi associado à interface fora do contexto do componente. Caso não seja definido este contrato específico, o contrato genérico da interface se aplica também às operações da faceta.

Uma interface que recebe tratamento especial é a *IComponent*. Ela possui um contrato bem definido, porém algumas operações como *startup* e *shutdown* possuem, normalmente, requisitos específicos de cada componente sendo criado. Portanto, o contrato de um componente pode especializar as assertivas para operações da interface *IComponent*, o que permite adequar o contrato genérico às particularidades de cada componente. Para isso, a chave *IComponent* deve ser criada na tabela do contrato do componente e deve ser associada a uma tabela que redefine as condições necessárias. O capítulo 5 possui exemplos de contratos de componentes que usam este recurso para redefinir a operação *IComponent::startup*.

4.1.11
Gerenciamento remoto de contratos

Uma outra facilidade implementada é a possibilidade de se registrar um contrato remotamente, quando este se aplica a um objeto do OiL. Esta característica foi necessária para permitir o registro de contratos para componentes SCS, uma vez que estes são carregados e instanciados remotamente no contêiner através da faceta *ComponentLoader*.

O mecanismo de gerenciamento remoto pode ser utilizado através da interface *ContractManager*, implementada no módulo *interceptor* do LUC. Ele provê a interface contida na listagem 4.5.

Listagem 4.5: Interface de gerenciamento de contratos

```
module luc {
    interface ContractManager
    {
        boolean addContract(in string contract);
    };
};
```

O parâmetro passado para a chamada a *addContract* é o mesmo conteúdo dos arquivos Lua usados para definição do contrato. A seção 4.4 contém exemplos deste tipo de arquivo.

4.1.12
Configuração

A configuração do LUC permite selecionar quais os tipos de verificações de contratos que serão realizadas, tanto no nível de cada interface como de forma global. Assim, é possível desligar, por exemplo, apenas a verificação das pós-condições para uma determinada interface. Esse mecanismo permite o controle das verificações de pré e pós-condições, invariantes, sincronização e transição de estados.

A configuração pode ser alterada através da chamada à função `luc.contract.configuration.checks()`, que recebe dois parâmetros: uma tabela com as alterações a serem feitas, e uma *string* opcional, contendo o nome da interface à qual a alteração se aplica. O conteúdo da tabela de alterações é o nome da verificação a ser ligada ou desligada, associada a um valor booleano. Para especificar o tipo de verificação, as opções são: *pre*, *post*, *inv*, *state* e *sync*. Por exemplo, a chamada abaixo desliga a verificação de transição de estados para todas as interfaces presentes no sistema.

```
luc.contract.configuration.checks{ state=false }
```

Este tipo de mecanismo é útil quando temos em um mesmo sistema componentes com diferentes graus de confiabilidade, o que permite um controle mais rigoroso sobre componentes adicionados ao sistema e que ainda não foram testados por um longo período. Ao mesmo tempo, para componentes já considerados mais confiáveis, é possível relaxar a verificação de correção da implementação.

Além das verificações, é possível controlar os tipos de notificação habilitadas, através da chamada a `luc.contract.configuration.notifications()`. O parâmetro da chamada é uma tabela semelhante à anterior, porém com as seguintes opções de chaves: *exception*, *log* e *remote*, correspondendo aos três tipos de notificação já mencionados neste capítulo.

Como visto na subseção 4.1.6, os contratos de sincronização não são "opcionais" como os contratos comportamentais. Assim, o desligamento da sua verificação pode comprometer o funcionamento correto de um componente. Porém, optou-se por permitir a desativação da verificação deste tipo de contrato pois ela pode ser desejável durante o processo de desenvolvimento de um componente, para tentar isolar um determinado problema não relacionado ao mecanismo de concorrência.

4.2 Arquitetura

Nesta seção, é apresentada uma visão geral da arquitetura da solução e da sua interação com o OiL e o SCS. A partir dela, é possível identificar os elementos que compõem o sistema criado e entender de forma mais clara o seu funcionamento, servindo de base para a seção 4.3, que descreve a implementação realizada em maior detalhe.

4.2.1 Interceptação de Chamadas

A subseção 2.3.7 apresentou diversas formas de implementação de suporte a contratos nas linguagens de programação. No caso do LUC, duas delas foram utilizadas: *proxies* e interceptadores OiL/CORBA.

O uso de *proxies* aplica-se aos casos em que o contrato está associado a um objeto Lua, sendo utilizado de forma independente da plataforma OiL/CORBA. É gerado um *proxy* dinâmico para o objeto em questão, e este é associado a um contrato de interface. A cada chamada do objeto original, é feita a verificação da pré-condição do contrato, se existir. Após a chamada do objeto original, a pós-condição

é verificada antes do retorno pelo *proxy*. As invariantes também são verificadas nos dois momentos. A figura 4.2 ilustra este mecanismo em funcionamento. O uso de *proxies* dinâmicos é detalhado na subseção 4.3.1.

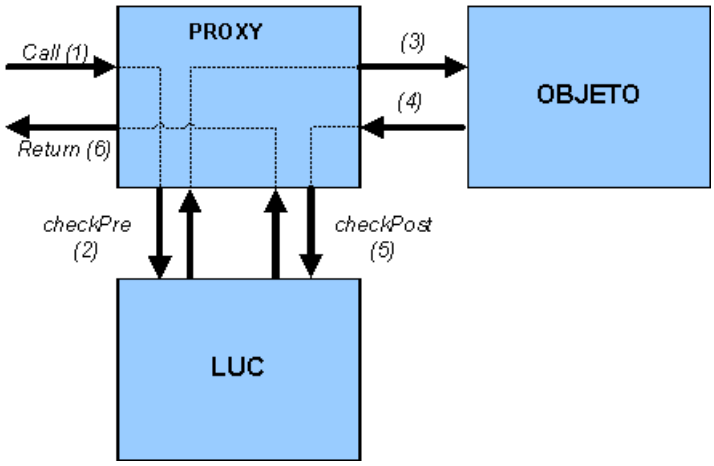

Figura 4.2: Interceptação de chamadas via *proxy*

Para objetos do OiL, a captura das chamadas é realizada através de interceptadores CORBA, módulos especializados que se registram no ORB (*Object Request Broker*) para receber notificações antes e depois de cada chamada. Isto permite que um mecanismo análogo ao usado no caso dos *proxies* seja utilizado para verificar as condições do contrato. Na verdade, a infra-estrutura de verificação de assertivas é o mesmo em ambos os casos, o que muda é apenas a forma de interceptar a chamada que está sendo feita. A tabela 4.7 descreve as chamadas feitas pelo OiL ao módulo interceptador do LUC, e o seu significado.

Função	Interceptador	Descrição
receiverequest	servidor	chamada antes do objeto receber a requisição
sendreply	servidor	chamada logo após o retorno da operação do objeto
sendrequest	cliente	chamada antes de a requisição ser enviada
receivereply	cliente	chamada antes de o objeto cliente receber a resposta da requisição

Tabela 4.7: Funções de um interceptador no OiL

A figura 4.3 ilustra a dinâmica de funcionamento do LUC com os interceptadores, mostrando a seqüência de chamadas que ocorrem durante a chamada de uma operação de um objeto OiL, no lado do servidor. Para o lado do cliente, o mecanismo é análogo, mudando apenas as chamadas do interceptador. Este mesmo processo se aplica às chamadas de uma faceta de componente SCS, uma vez que também se trata de um objeto CORBA.

O uso de interceptadores torna o uso de contratos no ambiente OiL bastante flexível e transparente para o servidor do objeto, uma vez que a verificação dos contratos pode ser habilitada ou desabilitada sem que nenhuma alteração de código do servidor seja necessária.

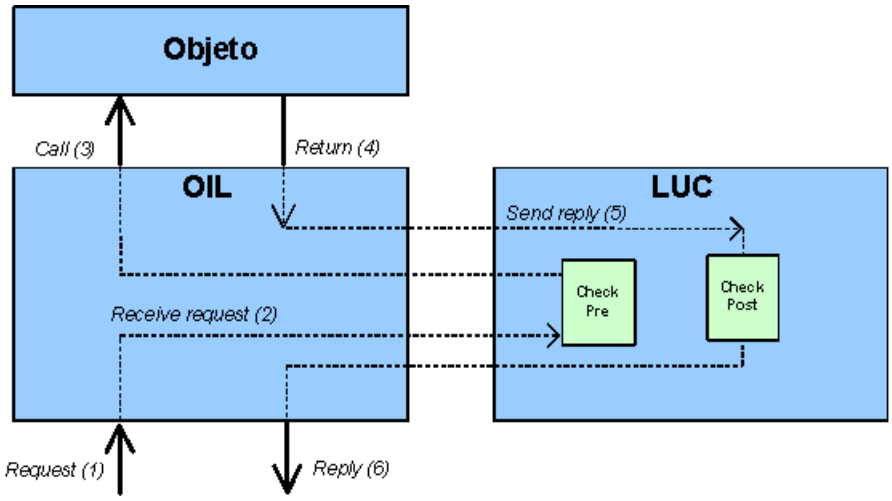

Figura 4.3: Interceptação de chamadas de servidor no OiL

4.2.2 Estrutura de Módulos

A arquitetura da solução implementada é baseada em módulos, procurando atribuir a cada um deles as funcionalidades classificadas de acordo com o seu objetivo. A tabela 4.8 relaciona os módulos criados e as principais responsabilidades de cada um.

Módulo	Descrição
luc.contract	módulo principal, servindo de "fachada"para a carga e configuração do sistema pelo cliente
luc.config	configuração das verificações e notificações
luc.base	módulo básico responsável pelas verificações de assertivas e transição de estados
luc.debug	módulo auxiliar de mensagens para depuração
luc.interceptor	interceptador OiL que serve de ponto de entrada para aplicações nesse ambiente
luc.log	geração das notificações em arquivos de log
luc.notification	notificações remotas
luc.parse	concentra as funções relacionadas à interpretação e manipulação de código Lua
luc.quantifiers	implementação dos quantificadores lógicos e suas variações
luc.scs	tratamento específico de interfaces de componentes SCS
luc.sync	controle de sincronização das chamadas
luc.util	funções de uso genérico utilizadas pelos demais módulos

Tabela 4.8: Módulos do sistema

Apesar de haver diversos módulos, em geral o usuário do sistema irá referenciar apenas dois deles: *luc.contract* e *luc.interceptor*. O primeiro é o que oferece as chamadas necessárias para o registro de um contrato e para a configuração do sistema, e o segundo efetua a interceptação do ORB em aplicações OiL. Os demais módulos são utilizados indiretamente pelos dois primeiros, e a princípio não precisam ser referenciados diretamente pelo usuário do subsistema de contratos.

4.2.3
Funcionamento

A interação do subsistema de contratos com o OiL ocorre através do registro do interceptador de chamadas, como já foi visto. Durante a carga da aplicação, o módulo *luc.interceptor* é responsável por este registro e pela inicialização relacionada ao uso do Oil. Em seguida, o módulo *luc.contract* deve ser carregado para permitir que as chamadas de definição de contratos sejam efetuadas. Neste momento, é possível carregar o contrato das interfaces a serem monitoradas, e iniciar a aplicação.

Para realizar este processo de forma transparente para a aplicação, deve ser criado um módulo que encapsula esta seqüência de operações, a ser carregado imediatamente antes do início da aplicação. O diagrama UML da figura 4.4 fornece uma visão geral do relacionamento entre a aplicação cliente (*app*), os módulos do LUC e o Oil.

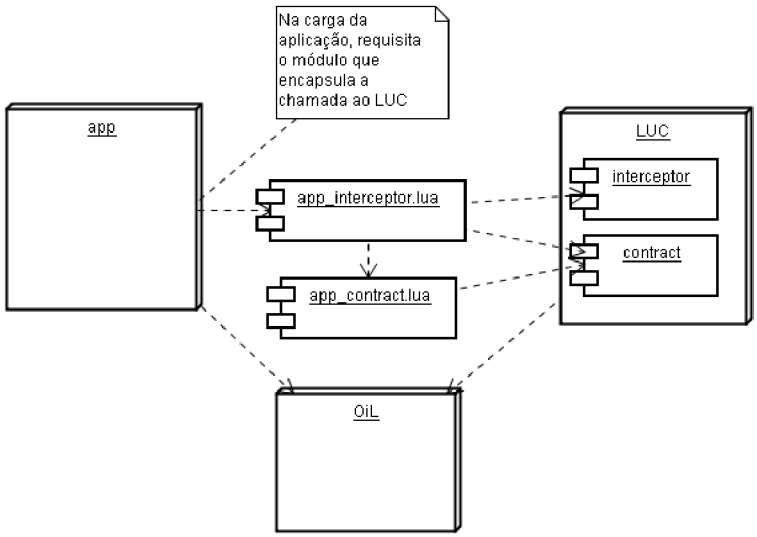

Figura 4.4: Relacionamento entre a aplicação, LUC e o OiL

Embora não seja a única forma de carregar o LUC ao iniciar uma aplicação, o mecanismo ilustrado na figura 4.4 estabelece um padrão para inserção de contratos numa aplicação baseada no OiL. O usuário deve criar um arquivo Lua (chamado de *app_interceptor.lua* no exemplo) que será responsável por carregar os módulos *interceptor* e *contract* do LUC, além do contrato da aplicação (*app_contract.lua*), nesta ordem. Como Lua permite que a carga (*require*) de um módulo seja especificada na linha de comando, é possível que o processo todo ocorra antes de a aplicação ser iniciada, tornando-o transparente, ou seja, não é necessário alterar o código fonte da aplicação para habilitar o contrato.

No caso de uma aplicação Lua convencional, um mecanismo semelhante de ativação do LUC pode ser usado, com a diferença de que não é necessário carregar o

interceptador de chamadas do OiL. Por outro lado, é necessário criar explicitamente os *proxies* para os objetos que devem ser monitorados por contratos, através da chamada à função `luc.contract.createProxy()`. Ela recebe como parâmetros o objeto a ser monitorado e o nome da interface cujo contrato foi registrado no sistema. A partir daí, as chamadas devem ser feitas sobre o *proxy* retornado, para que a verificação do contrato seja efetuada com sucesso.

O suporte a componentes SCS funciona de forma semelhante ao suporte a objetos OiL, com a principal diferença de que é necessário identificar quando um objeto é uma faceta de um componente para que, nas chamadas posteriores, o contexto do componente ao qual o objeto pertence seja recuperado e o contrato correto seja aplicado. Esta identificação pode ser feita através do monitoramento das chamadas `IComponent::getFacet()` e `IComponent::getFacetByName()`. Estes métodos são o ponto de partida para o uso de um componente, e portanto o seu retorno é verificado para que os objetos sejam identificados no sistema como facetas do componente em questão.

No caso de receptáculos, as conexões são verificadas durante a chamada ao método `connect()` da interface *IReceptacles*. A declaração deste método prevê a exceção *InvalidConnection* para os casos em que o objeto passado não é compatível com o receptáculo especificado. Neste caso, o contrato da interface *IReceptacles* pode reforçar essa verificação, validando a implementação do SCS para garantir que o objeto passado está de acordo com o contrato especificado para o componente. É possível também alterar o contrato da interface *IReceptacles* para que este interprete a conexão de um objeto incompatível como sendo uma violação de contrato. Porém, neste caso haveria uma sobreposição de função com a própria interface definida para o SCS, o que não é recomendável. Este contrato será visto com mais detalhes no capítulo 5, e as principais interfaces do SCS encontram-se listadas no apêndice A.

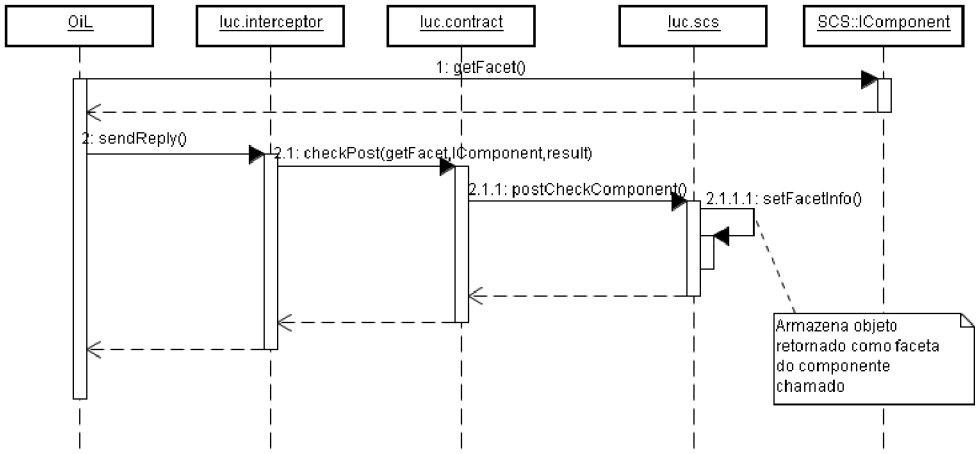

Figura 4.5: Seqüência de operações para detecção de faceta do SCS

4.3 Implementação

Esta seção apresenta alguns aspectos relevantes da implementação da solução, porém sem entrar num nível de detalhe que fuja ao escopo deste documento. O objetivo é ilustrar algumas das alternativas e dificuldades relacionadas ao projeto, para dar uma visão mais precisa das suas principais características.

4.3.1 Avaliação de assertivas

A avaliação de assertivas dos contratos é implementada no LUC através da geração dinâmica de funções que retornam o resultado da expressão booleana correspondente. O suporte à interpretação dinâmica de código em Lua por meio da função *loadstring* facilita este tipo de tarefa.

O conceito de *environment* da linguagem permite definir as variáveis e funções que devem ser visíveis no escopo de execução de uma determinada função. Este recurso é utilizado no LUC com duas finalidades. A primeira é permitir que a função de avaliação da assertiva referencie corretamente os parâmetros da operação sendo avaliada, bem como o próprio objeto chamado (*self*) e as funções adicionadas à linguagem de contratos do LUC. A segunda finalidade é limitar o escopo de atuação da assertiva, criando uma espécie de *sandbox* para a sua execução que restringe o uso de funções das bibliotecas da linguagem, por exemplo.

Durante a avaliação de uma assertiva, é criado um *proxy* dinâmico para o objeto sendo chamado, o que permite a avaliação recursiva de contratos, ou seja, caso a assertiva possua chamadas a outras operações do próprio objeto, as condições do contrato associadas a estas operações também são verificadas. Além disso, caso alguma das operações envolvidas retorne um objeto, é criado um *proxy* para ele, visando a avaliação do seu contrato, se existir. Para decidir sobre a criação ou não de um *proxy* nesta situação, o LUC consulta o repositório de interfaces do OiL para obter o tipo do retorno do método chamado.

Os *proxies* criados no sistema são "dinâmicos" porque os métodos que encapsulam a chamada ao objeto original são gerados sob demanda, em tempo de execução. Os mecanismos de *metatables* e *closures* de Lua facilitam este tipo de implementação, e oferecem uma grande flexibilidade para o programador.

Cada vez que um método do *proxy* é chamado pela primeira vez, é criada uma *closure* que efetua as chamadas para a verificação das pré e pós-condições da operação, além de efetuar a chamada ao objeto original. Esta função é armazenada

numa tabela para que possa ser reutilizado nas chamadas subseqüentes, o que contribui para um melhor desempenho do sistema.

O suporte a avaliações recursivas permite a detecção de inconsistências nas chamadas sendo efetuadas durante a avaliação das assertivas. Por outro lado, introduzem o risco de dependência mútua entre assertivas de duas operações. Por exemplo, se a pré-condição do método `f()` de uma interface se baseia na chamada do método `g()`, e vice-versa, haveria um *loop* infinito na avaliação. Para contornar este problema, a implementação efetua o controle sobre quais métodos estão sendo verificados no contrato para cada objeto. Assim, caso seja detectada uma avaliação reentrante do contrato, a verificação das assertivas deixa de ser feita.

4.3.2
Tratamento de PRE

Uma característica comum a praticamente todos os sistemas de suporte a contratos é o salvamento do estado referente a alguma variável ou expressão no momento anterior à chamada da operação sendo verificada, para uso na pós-condição. Na linguagem Eiffel, seria o equivalente ao uso do operador *old* sobre uma determinada expressão.

Na implementação realizada, esta funcionalidade está disponível para o usuário através da função *PRE*, conforme visto na seção 4.1. Ela recebe como parâmetro uma *string* contendo uma expressão Lua a ser avaliada no momento da pré-condição, para ser usada posteriormente na pós-condição.

Para detectar referências a *PRE* nas pós-condições dos contratos, foi utilizada a biblioteca LuaFish [34], que permite a construção e manipulação da árvore de sintaxe abstrata de Lua (*abstract syntax tree*, ou *AST*). Ela é baseada na biblioteca LPEG [35, 36], voltada para a geração de interpretadores de gramáticas em Lua, sendo de uso mais genérico.

Com a montagem da AST correspondente ao código das assertivas das pós-condições, é possível percorrê-la em busca de chamadas da função *PRE* e obter as expressões que devem ser avaliadas antes da chamada da operação.

A montagem da AST e a busca das expressões do *PRE* é feita uma só vez, no momento em que o contrato é adicionado ao sistema. A partir daí, a cada chamada a expressão é avaliada durante o tratamento da pré-condição com os valores correntes (parâmetros, atributos, objeto chamado), para que possa ser referenciada mais tarde durante o tratamento da pós-condição.

Para valores simples, este procedimento de salvar o estado do *PRE* funciona

satisfatoriamente. Porém, dependendo da expressão sendo referenciada, o mecanismo pode não retornar o valor pretendido. Isto ocorre quando, por exemplo, o resultado da expressão usada no *PRE* é um objeto ou uma estrutura de dados mais complexa, como uma tabela. A princípio, a referência para um objeto ou tabela são os mesmos tanto antes como depois da chamada efetuada. Porém, o conteúdo deles pode ter sofrido alterações durante a chamada, sendo que isto não é refletido pelo resultado da avaliação do *PRE*, que retorna apenas o primeiro nível do objeto, ou seja, sua referência.

Uma possível alternativa a este problema seria realizar uma "cópia profunda" (*deep copy*) do objeto sendo retornado pela chamada a *PRE*. Ainda assim, haveria problemas relacionados à associação com outros objetos, inclusive remotos, que nem sempre poderiam ser facilmente copiados. Além disso, é necessário evitar ciclos nas estruturas do objeto. Um outro problema é o desempenho, pois o salvamento do estado de um objeto complexo pode ser uma operação muito custosa para ser realizada repetidamente.

Portanto, o ideal é, sempre que possível, evitar esta situação na especificação dos contratos, tentando sempre chegar ao nível de valores básicos nas expressões das chamadas a *PRE*. Por exemplo, seja uma assertiva de pós-condição que verifica se o valor retornado pelo método *count* foi incrementado. Uma possibilidade seria escrever

```
PRE("obj"):count() + 1 == obj:count()
```

Porém, embora válida, a assertiva não obtém o resultado esperado, pois a referência para o objeto antes e depois da chamada é a mesma, e seu estado interno não seria copiado. A assertiva adequada neste caso seria

```
PRE("obj:count()") + 1 == obj:count()
```

Como desta vez a expressão avaliada antes da chamada seria um valor numérico, a assertiva obtém o resultado pretendido para a verificação da semântica da chamada.

4.3.3 Quantificadores lógicos

A avaliação de assertivas que envolvem um grande número de elementos pode ter um custo alto em termos de desempenho na verificação de um contrato. Um caso freqüente deste tipo de avaliação é o uso dos quantificadores lógicos *FORALL* e *EXISTS*, que operam sobre uma seqüência de elementos.

Em algumas situações, o uso dos quantificadores está associado a chamadas

de métodos de objetos remotos, o que aumenta ainda mais este custo, pois além do tempo de processamento existe o tempo de comunicação envolvido. Por outro lado, o uso de chamadas remotas favorece o paralelismo na avaliação, uma vez que, possivelmente, as chamadas serão tratadas por processos em computadores distintos ou multiprocessados. Portanto, é interessante poder enviar as requisições para esses objetos remotos em paralelo e tirar proveito da independência existente entre elas.

Além da vantagem imediata de se efetuar as requisições em paralelo, é possível ainda obter o benefício da avaliação em "curto-circuito" dos quantificadores, uma vez que ambos podem ser interrompidos antecipadamente dependendo dos resultados obtidos. Por exemplo, para o *FORALL*, basta que um elemento do conjunto não satisfaça a assertiva proposta para que o resultado da função seja falso. Para o *EXISTS*, a situação se inverte: basta que um elemento satisfaça a assertiva para que o resultado seja verdadeiro. No pior caso, para ambos os quantificadores, é necessário avaliar a assertiva para todos os elementos do conjunto.

Para explorar as variações sobre este tema, foram implementados três mecanismos de avaliação das assertivas dos quantificadores: seqüencial, *thread pool* e chamadas *deferred*.

A primeira, e mais imediata, é a avaliação seqüencial, ou seja, a assertiva é verificada para um elemento do conjunto de cada vez, interrompendo o processo antes da hora se algum resultado assim permitir.

A segunda técnica utiliza um *thread pool*, em que cada *thread* é alocada a um elemento do conjunto de cada vez para avaliar a assertiva em paralelo, se possível. Se o número de elementos for maior que o de *threads*, o que pode ocorrer com freqüência, as *threads* vão sendo alocadas a outros elementos do conjunto à medida que as avaliações vão terminando. Assim como no caso seqüencial, o processo também pode parar antes de chegar ao final do conjunto. A figura 4.6 ilustra o mecanismo de avaliação baseado em *threads*.

Comparada com as anteriores, a terceira técnica é uma forma intermediária de obter um certo grau de paralelismo: a cada avaliação da assertiva, o sistema tenta "adiantar" as chamadas remotas, efetuando-as de forma assíncrona, utilizando o recurso de chamada *deferred* do OiL. Assim, ao avaliar a expressão para o próximo elemento do conjunto, o sistema verifica se o resultado da chamada remota já se encontra disponível. Caso não esteja, o sistema espera que a operação se complete. O número de chamadas simultâneas neste esquema é configurável, e permite que se estabeleça um controle sobre o nível de "antecipação" das chamadas. Diferentemente do mecanismo anterior, este utiliza apenas uma *thread* no OiL, que dispara as chamadas assincronamente e verifica os resultados sob demanda. A implementação das chamadas *deferred* é feita através do uso de *proxies*, semelhantes aos que são usados para os contratos. Quando uma chamada ao proxy é feita, ele verifica se o

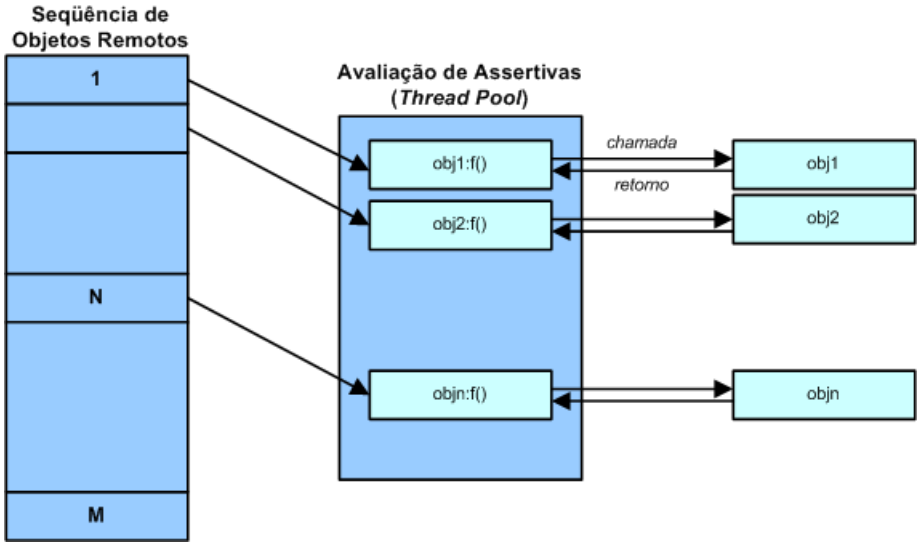

Figura 4.6: Implementação de quantificador com *thread pool*

resultado já está disponível. Caso não esteja, efetua a chamada para o próximo objeto de forma assíncrona, e em seguida realiza a sua chamada sincronamente. A figura 4.7 ilustra este mecanismo.

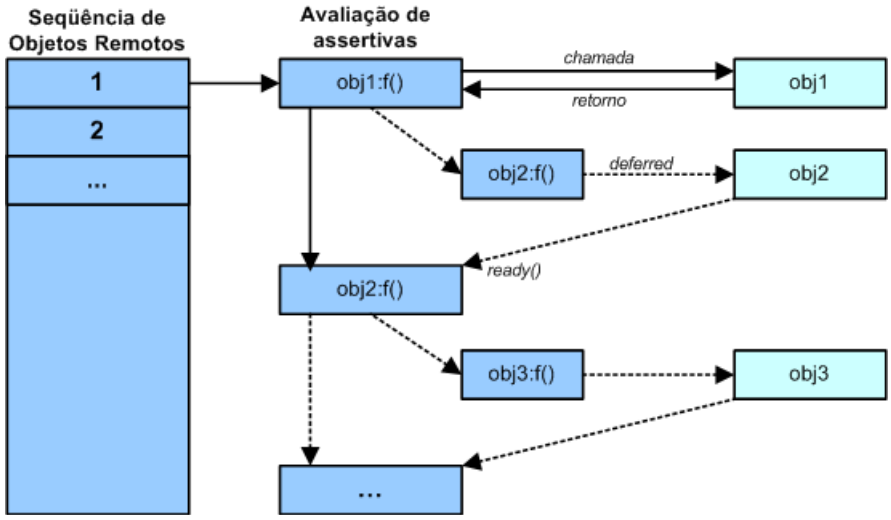

Figura 4.7: Implementação de quantificador com chamada *deferred*

É importante lembrar que, por suas características, as duas técnicas que exploram algum grau de paralelismo só estão disponíveis quando se trata de um contrato de objeto OiL, pois os recursos utilizados dependem da sua estrutura. Além disso, elas fazem sentido apenas no contexto de quantificadores lógicos aplicados a objetos remotos.

Outra observação importante a ser feita é que a avaliação das chamadas em paralelo abre a possibilidade de efeitos colaterais sobre os resultados das assertivas. Isto pode ocorrer quando o tipo de avaliação e os objetos envolvidos pressupõem a avaliação seqüencial, como no caso em que o resultado de uma é usado na expressão

seguinte. Neste caso, é critério do programador, ao especificar o contrato, selecionar o mecanismo a ser utilizado no quantificador lógico. A sintaxe do contrato permite especificar tanto um mecanismo *default* para os dois tipos de quantificador, bem como a utilização direta de um tipo determinado na expressão.

A utilização direta dos quantificadores nas assertivas pode ser especificada por meio de um sufixo que identifica o método desejado. Para compor o nome, basta concatenar o nome do quantificador em minúsculo com um ponto e o sufixo escolhido. As opções são: *sequential*, *threads* e *deferred*. Alguns exemplos de combinação seriam: *forall.sequential*, *exists.threads*, *forall.deferred*, e assim por diante.

A chamada para o quantificador *default* da configuração escolhida se faz através da chamada a *FORALL* ou *EXISTS*. Caso não seja especificada, a versão *default* dos quantificadores é a seqüencial. Para alterá-la, basta especificar na tabela do contrato as chaves *__forall* ou *__exists*, associando-as a uma *string* que contém um dos sufixos citados no parágrafo anterior.

4.3.4 Sincronização de chamadas

O mecanismo de sincronização de chamadas implementado é baseado no suporte a *threads* do OiL, caracterizado pelo uso de co-rotinas da linguagem Lua.

A cada avaliação de pré-condição, caso o suporte a sincronização esteja habilitado na configuração e a interface sendo chamada possua cláusulas de sincronização no seu contrato, é feita a verificação das chamadas em andamento para o objeto em questão. Caso alguma delas entre em conflito com a chamada corrente, a *thread* é suspensa, aguardando a sinalização do fim das demais operações. As *threads* com requisições suspensas são colocadas numa fila associada ao objeto chamado, para que possam ser notificadas sempre que uma chamada deste objeto seja finalizada.

Ao final de cada operação, após a verificação da pós-condição é sinalizado o seu fim para as possíveis *threads* aguardando por este evento. Isto é feito através da chamada *oil.tasks:resume()* do escalonador do OiL, o que permite que as *threads* suspensas verifiquem novamente a condição de execução da operação que aguarda a sua liberação.

Além das verificações de exclusão mútua, como visto na subseção 4.1.6, as pré-condições de sincronização do contrato são avaliadas antes de a operação ser invocada. Caso a assertiva seja verificada com sucesso, a *thread* corrente é autorizada a executar a operação.

O mecanismo de multitarefa cooperativa de Lua facilita a implementação deste

tipo de controle, uma vez que os pontos em que ocorre a preempção da execução do código são previsíveis. Com isso, são bem menores as possibilidades de haver condições de corrida nos testes de sincronização implementados. Porém, ainda assim existem limitações da solução em relação a este tópico, que serão abordados mais detalhadamente na seção 4.6.

4.4 Exemplos

Nesta seção, alguns exemplos simples de uso do sistema de contratos implementado são apresentados, com o intuito de consolidar a sintaxe de especificação dos contratos e a forma de utilizá-los. Contratos mais elaborados serão objeto do capítulo 5, sendo que os exemplos a seguir possuem um caráter mais didático sobre o uso da ferramenta.

4.4.1 *Stack*

Um dos exemplos clássicos para a especificação de contratos é a estrutura de dados de pilha, representada neste exemplo através da interface denominada *Stack*. Foram criados dois tipos de teste: um com o uso de contratos em objetos Lua simples e outro efetuando chamadas a um objeto remoto do OiL. Neste último, foram criados dois programas para implementar o lado cliente e servidor do teste.

Por ser o primeiro exemplo completo, a maioria dos arquivos utilizados no teste com o OiL serão apresentados, excluindo-se apenas os arquivos para o funcionamento via o mecanismo de *proxy*. A tabela 4.9 lista os arquivos criados e a finalidade de cada um.

Arquivo	Descrição
stack.idl	especificação da interface na linguagem IDL
stack.lua	implementação da interface *Stack*
stack_contract.lua	arquivo contendo o contrato
stack_interceptor.lua	arquivo que carrega os módulos do LUC no teste com o OiL
stack_server.lua	servidor do objeto para teste no OiL
stack_client.lua	cliente para teste no OiL
stack_proxy.lua	programa que executa o teste usando *proxy* para objeto local
stack_tests.lua	parte comum aos testes com o OiL e *proxy*, contendo as chamadas do objeto

Tabela 4.9: Arquivos do exemplo *Stack*

Os arquivos seguem o padrão descrito na seção 4.2 sobre a divisão de papéis e nomes para os testes com o OiL. Alguns arquivos da tabela serão detalhados nesta seção para uma melhor compreensão do exemplo baseado no OiL.

A listagem 4.6 apresenta o conteúdo do arquivo *stack.idl*, que especifica a interface CORBA do exemplo.

Listagem 4.6: Interface *Stack*

```
module stack {
    interface Stack {
        void push(in long x);
        long pop();
        long top();
        long size();
        long itemAt( in long pos );
    };
};
```

O arquivo *stack_interceptor.lua*, que deve ser carregado no início da aplicação *stack_server.lua*, é responsável pela inicialização do LUC e pela carga do contrato. Seu conteúdo encontra-se na listagem 4.7.

Listagem 4.7: Arquivo *stack_interceptor.lua*

```
require "luc.interceptor"
require "luc.contract"

-- Definicao do contrato
require "stack_contract"
```

O servidor para o teste no OiL está na listagem 4.8, seguido do cliente na listagem 4.9. A especificação do contrato encontra-se no arquivo *stack_contract.lua*, apresentado na listagem 4.10.

Listagem 4.8: Arquivo *stack_server.lua*

```
require "stack"
require "oil"
orb = orb or oil.init()

oil.main(function()
  print "starting stack server ..."
  orb:loadidlfile("stack.idl")

  local servant = orb:newservant(Stack{}, nil, "stack::Stack")

  oil.writeto("stack.ior", orb:tostring(servant))

  orb:run()
end)
```

As figuras 4.8 e 4.9 ilustram a execução do servidor e do cliente, respectivamente, estando o cliente configurado para provocar uma violação do contrato. Neste caso, a chamada sendo feita para a violação é a operação *top* com a pilha vazia, depois de terem sido inseridos e removidos alguns elementos.

Listagem 4.9: Arquivo *stack_ client.lua*

```
require "oil"
require "stack_tests"

local orb = orb or oil.init{}

function catchException(_, e)
  print( "-------------------------------------------------------\n\n" )
  print( "Ocorreu uma excecao: " .. e[1] )

  if e.traceback then
      print( e.traceback )
  end

  os.exit(1)
end

oil.main(function()
  orb:setexcatch(catchException)

  orb:loadidlfile("stack.idl")

  local stack = orb:newproxy( assert(oil.readfrom("stack.ior")) )

  test.a(stack)

  print("stack_client finished.")
end)
```

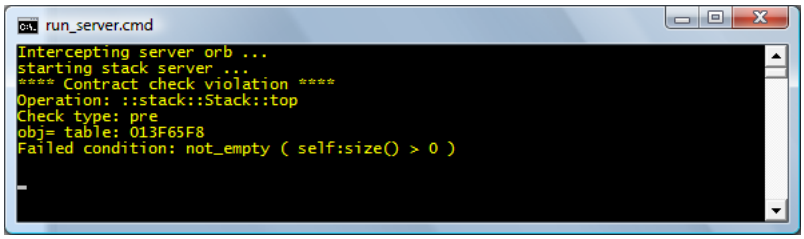

Figura 4.8: Execução do servidor *stack*

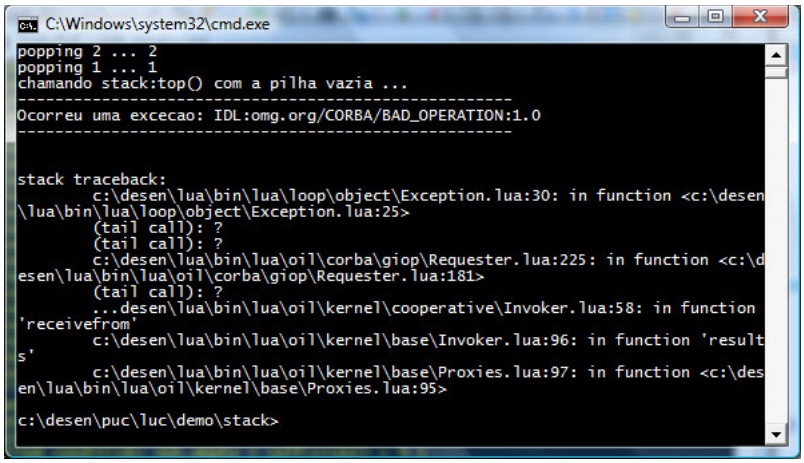

Figura 4.9: Execução do cliente *stack*

Listagem 4.10: Contrato para a interface *Stack*

```
luc.contract.interface
{
  __interface = "::stack::Stack",

  __inv = {
    inv = {
      non_negative_size = [[ self:size() >= 0 ]]
      },
  },

  top = {
    pre= {
      not_empty = [[ self:size() > 0 ]]
    },
    post = {
      return_is_at_top = "RESULT == self:itemAt(self:size())"
    }
  },

  push = {
    param_names={"x"},
    pre = {
      element_not_nil = [[ x ~= nil ]]
    },
    post = {
        element_inserted_at_top = [[ self:top() == x ]],
        size_increased = [[ self:size() == PRE("self:size()")+1 ]]
    }
  },

  pop = {
    pre = {
      not_empty = [[ self:size() > 0 ]]
    },
    post = {
      result_is_previous_top = [[ RESULT == PRE"self:top()" ]],
      size_decreased = [[ self:size() == PRE("self:size()")-1 ]]
    }
  },

  itemAt = {
    param_names = {"pos"},
    pre = {
      pos_inside_bounds = [[ pos > 0 and pos <= self:size() ]]
    }
  },

  __state = {
    states = {
      "empty",
      "not_empty"
    },
    initial_state = "empty",
    transitions = {
      empty = {
        push = { condition="true", state="not_empty" },
        -- operacoes nao permitidas com a pilha vazia
        pop = {},
        top = {},
        itemAt = {},
      },
      not_empty = {
        pop = { condition="self:size() == 0", state="empty" },
      }
    }
  }
}
```

4.4.2 *List*

O segundo exemplo é baseado na implementação de uma lista "imutável", ou seja, uma lista que não pode ser alterada, apenas serve de base para a criação de novas listas a partir da adição de um elemento ou da geração de uma sub-lista. Este exemplo é baseado no seu equivalente em Eiffel extraído de [16]. A listagem 4.11 apresenta o conteúdo do arquivo *list.idl*, que descreve a interface da lista.

Listagem 4.11: Interface *List*

```
interface List {
    string head();       //retorna o primeiro elemento da lista
    List tail();         //retorna a lista formada a partir do segundo elemento
    boolean is_empty();
    long count();
    List preceded_by(in string x);  //retorna uma copia precedida por 'x'
    boolean is_equal( in List other );  //compara o conteudo de duas listas
    string item( in long i );       //retorna o i-esimo elemento
    List sublist(in long from, in long to); //sublista [from,to]
};
```

O arquivo *list_interceptor.lua* é similar ao seu correspondente no exemplo *Stack*. O servidor OiL do objeto lista também é análogo ao do exemplo anterior e por isso também pode ser omitido. A listagem 4.12 contém a implementação do exemplo para a interface List.

O contrato para a interface *List* é um exemplo interessante pois utiliza definições recursivas para as assertivas que especificam as operações. Para facilitar a descrição do contrato, ele será apresentado de forma segmentada nas listagens a seguir.

A operação *count* (listagem 4.13) é uma consulta básica da interface, e é definida em termos da consulta *is_empty* e de uma chamada recursiva sobre o resultado da operação *tail*. No primeiro caso, foi utilizada a implicação lógica para expressar que o resultado deve ser zero sempre que *is_empty* retorna *true*, o que é intuitivo. No segundo caso, a implicação lógica do resultado de *is_empty* ser *false* é que o resultado de *count* é a contagem de elementos da lista *tail* somada de um. A segunda assertiva não utiliza a função *IMPLIES* porque, como visto, ela não implementa o curto-circuito de operações booleanas, e portanto a segunda parte da expressão seria avaliada mesmo que a lista fosse vazia, o que causa um *loop* com a avaliação recursiva da operação *count*.

Observa-se que, em casos como o método *count*, a especificação da assertiva do contrato é mais complexa que a implementação em si, o que às vezes é usado como argumento contra o uso de contratos. Em determinadas situações, é possível que o contrato seja mais complicado de entender que a implementação da funcionalidade, cabendo ao analista decidir se deve aplicar as assertivas ou não. Porém, mesmo nestes

Listagem 4.12: Implementação da interface *List*

```
local oo = require "loop.simple"
List = oo.class{ l = {} }

function List:__init( t )
    return oo.rawnew(self, { l = t or {} } )
end

function List:head()
    return self.l[1]
end

function List:tail()
    local t = {}
    for i = 2, #self.l do
        table.insert( t, self.l[i] )
    end
    return List(t)
end

function List:is_empty()
    local result = (#self.l == 0)
    return result
end

function List:count()
    return #self.l
end

function List:preceded_by( x )
    local t = { x }
    for k,v in ipairs(self.l) do
        table.insert( t, v )
    end
    return List(t)
end

function List:is_equal( other )
    if self:count() ~= other:count() then
        return false
    end
    for i, v in ipairs(self.l) do
        if v ~= other:item(i) then
            return false
        end
    end
    return true
end

function List:item( i )
    return self.l[i]
end

function List:sublist( from, to )
    local t = {}
    for i = from, to do
        table.insert( t, self.l[i] )
    end
    return List(t)
end
```

Listagem 4.13: Operação *List::count()*

```
count = {
    post = {
        empty_implies_count_is_zero =
            [[ IMPLIES( self:is_empty(), RESULT == 0) ]],
        not_empty_implies_tail_count_plus_1 =
            [[ self:is_empty() or (RESULT == self:tail():count() + 1) ]]
    },
},
```

casos as assertivas provêem uma especificação precisa da semântica da interface de modo a aumentar a sua compreensão e permitir a sua verificação automática.

A listagem 4.14 ilustra o contrato da operação *preceded_by*. Ela define três assertivas na sua pós-condição, que indicam o comportamento deste método em função das consultas básicas *is_empty*, *head* e *tail*.

Listagem 4.14: Operação *List::preceded_by()*

```
preceded_by = {
    param_names = {"x"},
    post = {
        not_empty = [[ not RESULT:is_empty() ]],
        head_is_x = [[ RESULT:head() == x ]],
        tail_is_original_list = [[ RESULT:tail():is_equal(self) ]]
    }
},
```

O contrato da operação *item(i)* (listagem 4.15) apresenta pré-condições que garantem que o índice passado encontra-se dentro dos limites. Esta é uma verificação típica da teoria de *Design by Contract*, que prega que o cliente é responsável pela correção dos argumentos passados, livrando o provedor da operação da responsabilidade de verificá-los. Cabe à infra-estrutura de contratos monitorar esta condição, apontando como culpado em caso de violação o lado cliente da operação. Se, por uma lado o código do servidor fica mais "limpo" de verificações de consistência, é necessário que todos os clientes verifiquem os parâmetros passados antes de efetuar a operação.

Listagem 4.15: Operação *List::item(i)*

```
item = {
    param_names = {"i"},
    pre = {
        i_large_enough = [[ i >= 1 ]],
        i_small_enough = [[ i <= self:count() ]]
    },
    post = {
        base_and_recursive_cases = [[
            (i == 1 and RESULT == self:head())
            or (RESULT == self:tail():item(i-1))
        ]],
    },
},
```

A operação *is_equal* (listagem 4.16) expressa a definição da igualdade entre duas listas. No caso de as listas não serem vazias, compara os elementos do início da lista e compara as listas resultantes recursivamente.

Listagem 4.16: Operação *List::is_equal()*

```
is_equal = {
    param_names = {'other'},
    pre = {
        other_exists = [[ other ~= nil ]]
    },
    post = {
        same_contents = [[
            RESULT == self:is_empty() and other:is_empty()
            or ( not self:is_empty()
                and not other:is_empty()
                and self:head() == other:head()
                and self:tail():is_equal(other:tail()) )
        ]]
    }
},
```

A última parte do contrato é a especificação da operação *sublist*, na listagem 4.17. Como os próprios rótulos das assertivas ajudam a descrever, a pré-condição valida a consistência dos índices usados na operação.

Seguindo os princípios apresentados na subseção 2.3.9, a sua pós-condição é definida a partir das consultas básicas *is_empty*, *head* e *tail*. A primeira assertiva garante a base da recursão utilizada na última assertiva, ou seja, caso o parâmetro *from* seja maior que *to*, então o resultado deve ser a lista vazia. Na segunda assertiva, a consistência do primeiro elemento da lista resultado é verificada com relação ao elemento da posição *from* especificada. Por fim, na terceira, a "cauda" da lista resultante é comparada recursivamente com a sublista gerada a partir da posição *from* + 1.

Listagem 4.17: Operação *List::sublist()*

```
sublist = {
    param_names = {"from","to"},
    pre = {
        from_position_large_enough = [[ from >= 1 ]],
        from_position_small_enough = [[ from <= to+1 ]],
        to_position_small_enough = [[ to <= self:count() ]],
    },
    post = {
        is_empty_consistent_with_from_and_to_position =
            [[ RESULT:is_empty() == (from > to) ]],
        result_head_is_at_from_position_in_self =
            [[ IMPLIES( from <= to, RESULT:head() == self:item(from) ) ]],
        result_tail_is_correct_sublist_within_self = [[
            IMPLIES( from <= to ,
                RESULT:tail():is_equal(self:sublist(from+1,to)) )
        ]]
    }
},
```

4.5 Desempenho

Como visto na subseção 2.3.2, um dos principais motivos de de crítica do uso de contratos é o impacto que eles podem ter sobre o desempenho do sistema. Este impacto varia conforme o tipo das verificações sendo realizadas nas assertivas.

A partir dos exemplos apresentados, e dos experimentos a serem descritos no capítulo 5, é possível observar que as características dos contratos variam bastante de uma interface para outra, o que também ocorre com o o custo extra das verificações das assertivas. É possível, inclusive, que o processamento da verificação do contrato ultrapasse a própria execução da operação a ser monitorada.

Portanto, definir de forma genérica o custo extra de processamento que uma ferramenta de suporte a contratos impõe não é uma tarefa simples, pois este custo varia em função do contrato sendo aplicado e da complexidade das verificações quando comparadas com a implementação em questão.

Porém, para ilustrar como este impacto pode se manifestar, efetuamos alguns testes com o exemplo *List*, apresentado na subseção 4.4.2. Ele ilustra um contrato que utiliza definições recursivas nas assertivas, e com isso executa uma série de chamadas extras para a sua verificação, principalmente nas pós-condições.

A tabela 4.10 ilustra os tempos médios para a execução de um programa de testes que executa diversas operações da interface *List*. Ele cria uma lista de 25 elementos por meio de chamadas repetidas a `preceded_by()`, e depois faz uma iteração na lista chamando a operação `item()` para cada elemento.

Descrição do teste	Tempo médio (s)	Tempo relativo
Sem contratos	0,23	1,00
Com contratos	1,51	6,64
Apenas pré-condições	0,24	1,06

Tabela 4.10: Testes de desempenho com a interface *List*

A segunda coluna da tabela 4.10 apresenta os tempos médios das execuções de cada teste, e na terceira coluna estão as razões do tempo médio obtido na segunda coluna sobre o tempo da execução sem contratos.

Foram feitas 5 execuções de cada teste, usando um computador portátil HP com processador AMD Turion 64x2, 2GB de RAM e sistema operacional *Windows Vista Business*, com Lua 5.1.3 e OiL 0.4. Os testes foram todos feitos localmente para evitar a influência do uso da rede sobre o desempenho total do programa.

Vale lembrar que o código do subsistema de contratos ainda não sofreu uma revisão mais detalhada com o objetivo de melhorar o seu desempenho. Portanto, é possível que os tempos apresentados possam ser reduzidos no futuro.

A partir dos resultados obtidos, verifica-se que o custo dos contratos pode ser bastante significativo quando utilizados em sistemas críticos do ponto de vista de desempenho.

Porém, é comum a prática de deixar apenas as pré-condições habilitadas para verificar as chamadas provenientes dos clientes. No exemplo anterior, o tempo obtido ao testar apenas as pré-condições é bastante próximo do tempo original, o que permite usar esta configuração mesmo em ambiente de produção onde o desempenho é um fator quase sempre crítico.

Portanto, como abordado na subseção 2.3.2, é possível "relaxar"a verificação do contrato quando este se aplica a um objeto ou componente que possa ser considerado estável, habilitando apenas as pré-condições para protegê-lo dos erros de aplicações clientes.

4.6 Limitações

Nesta seção, são discutidas as principais limitações identificadas na implementação atual do suporte a contratos, sendo que algumas delas já foram citadas previamente no texto. Soluções para parte destes problemas foram incluídas entre os possíveis trabalhos futuros citados no capítulo 6.

Efeitos colaterais nas assertivas

Como visto no capítulo 2, as assertivas que compõem um contrato devem ser isentas de efeitos colaterais, não devendo chamar métodos ou funções que alterem o estado observável do objeto sendo avaliado. Em geral, quando esta verificação não é um recurso disponível no compilador ou interpretador, torna-se bastante complexa de ser implementada em linguagens imperativas [37]. Neste caso, cabe ao analista que especifica o contrato determinar quais métodos podem ser usados nas assertivas por não causar efeitos colaterais. Como a especificação do contrato deve ocorrer juntamente com a da interface, não é muito difícil manter a coerência em relação à utilização de métodos sem efeito colateral nas assertivas.

O problema maior ocorre no momento da implementação, pois embora um método possa ter sido especificado sem efeitos colaterais, o programador pode não respeitar ou não atentar para esta condição, inserindo efeitos colaterais no código de operações usadas em assertivas, o que poderá não ser identificado pelas ferramentas

utilizadas, embora em algum momento esta situação deverá provocar uma violação do contrato devido aos efeitos colaterais.

A linguagem de especificação JML (*Java Modeling Language*) [17] permite especificar métodos "puros", isto é, isentos de efeitos colaterais, através do modificador *pure*. Apenas métodos deste tipo podem ser usados nas assertivas de contratos. Embora prevista na linguagem, a definição de métodos puros serve apenas de indicação para o compilador JML de quais métodos são isentos de efeitos colaterais. Porém, a verificação do código para determinar se esta condição é respeitada depende do uso de ferramentas mais específicas [38].

A implementação atual do LUC não impõe restrições sobre a chamadas de métodos Lua/OiL, cabendo ao usuário evitar chamadas com efeitos colaterais. A linguagem IDL de CORBA também não prevê a definição de operações sem efeito colateral. O *middleware* ICE [39], inspirado em CORBA, prevê a declaração de operações com o modificador *idempotent*, servindo de "dica" para o ambiente de execução tentar ser mais agressivo na recuperação de operações com esta característica, uma vez que podem ser chamadas novamente sem que haja efeitos colaterais.

Concorrência na avaliação

A multitarefa cooperativa presente na linguagem Lua e no OiL facilita o tratamento da concorrência na avaliação dos contratos, pois permite um maior controle sobre os pontos de preempção do código. Porém, podem ocorrer condições de corrida durante a avaliação das pré e pós-condições, caso seja efetuada alguma operação remota que provoque a suspensão da tarefa corrente.

O uso da função *PRE* na pós-condição também pode ser influenciado pela concorrência ao objeto chamado, pois o valor salvo para uma expressão pode não corresponder ao valor no momento exato da chamada sendo avaliada, já que pode haver uma interrupção entre o salvamento do estado e a chamada da operação. Além disso, caso a expressão salva dependa de objetos remotos, a concorrência no acesso a esses objetos também pode causar mudanças no valor da expressão entre o seu salvamento e o uso na pós-condição.

Uma possível melhoria seria implementar a exclusão mútua no acesso ao objeto durante a avaliação do contrato, para evitar que a rotina avalie a assertiva parcialmente e seja interrompida, o que poderia dar chance a uma outra chamada ser aceita e ocasionar a alteração do objeto. Porém, isto poderia causar a ocorrência de *deadlocks*, pois uma chamada remota durante a avaliação poderia efetuar uma outra chamada ao objeto original, como uma "callback", por exemplo.

Além da exclusão mútua nas operações do objeto, seria necessário em alguns

casos o bloqueio (*lock*) dos objetos remotos envolvidos nas assertivas, para garantir que o seu estado não mude durante a avaliação da pré-condição e a execução do método. Porém, o uso de um mecanismo de sincronização distribuído pode tornar o custo para o uso de assertivas muito alto, além de aumentar a possibilidade de *deadlocks*.

O mesmo problema pode ocorrer na avaliação da pós-condição, porém com menor incidência, pois é comum haver mais dependências de objetos externos na pré-condição do que na pós. Em geral, as assertivas da pós-condição estão mais ligadas ao próprio objeto, que também pode sofrer mudança de estado devido a chamadas concorrentes. Neste caso, seria necessário bloquear o objeto entre a execução da chamada e a avaliação da pós-condição, para evitar falsos positivos e falsos negativos na violação do contrato.

Para evitar o bloqueio mútuo na avaliação, seria necessário implementar um mecanismo de transação distribuída que atribuísse um identificador às operações que fazem parte de uma mesma transação. Este assunto será abordado novamente no tópico sobre limitações relacionadas aos contratos de sincronização.

Vale lembrar que estes problemas relacionados à concorrência são inerentes ao mecanismo de avaliação de contratos e, portanto, não são exclusivos da implementação realizada para este estudo.

Salvamento de estado

Uma limitação já apresentada neste capítulo é a ausência de um mecanismo de *deep copy* para o salvamento do estado de um objeto ao ser referenciado pela função *PRE* na pós-condição. Embora a princípio seja uma característica relevante, a sua ausência não prejudicou os experimentos realizados com o sistema de contratos.

A cópia "profunda" de um objeto pode ter semânticas diferentes a respeito da relação com objetos subjacentes. Por exemplo, quando um objeto se relaciona com outro por meio de agregação, a necessidade de criar uma cópia do objeto que é parte do original se torna mais evidente. Porém, no caso de uma relação de delegação, a criação da cópia do objeto externo pode fazer ou não sentido, dependendo do contexto. No caso de componentes, esta situação se complica ainda mais com relação às dependências externas (receptáculos), isto é, deve-se determinar até onde vai a cópia do estado de um objeto, e fazer isso de modo genérico nem sempre produz o resultado mais adequado.

Uma possibilidade de melhorar a questão do salvamento do estado de um objeto seria incluir na interface um método que implementasse a sua "clonagem", semelhante ao que ocorre na hierarquia de objetos em Java com o método *clone*. Porém, ao

deixar esta funcionalidade para a implementação da interface não há como o sistema de contratos garantir a sua confiabilidade, apenas referenciá-la, além de criar uma dependência do contrato em relação à implementação, que não é desejável.

Uma outra desvantagem do uso do *deep copy* é o custo associado à operação, pois quando o objeto possui estruturas complexas no seu estado o salvamento pode ter um grande impacto no desempenho. Há ainda as questões relativas à concorrência, citadas no tópico anterior desta seção. Ainda assim, seria desejável ter na ferramenta de contratos a possibilidade de implementar a cópia de um objeto, o que será avaliado em trabalhos futuros.

Exceções de violação

O tratamento de exceções CORBA não permite que uma chamada ao objeto sendo avaliado pelo contrato lance uma exceção diferente daquelas especificadas na interface, ou das exceções genéricas previstas no padrão de objetos. Portanto, a tentativa de criar uma exceção específica para a violação de um contrato não foi bem sucedida, uma vez que esta não é reconhecida ao ser lançada durante a execução da operação.

Para contornar o problema, a exceção *BAD_OPERATION* de CORBA foi escolhida para a notificação. Porém, por não haver campos suficientes na exceção, o cliente deixa de receber todos os detalhes em relação à operação que causou a violação, como o rótulo associado à assertiva que não foi satisfeita.

Para o cliente obter a informação sobre a violação do contrato, a alternativa é o uso da exceção em conjunto com outros mecanismos de notificação apresentados na seção 4.1. O uso do *stack trace* de Lua também ajuda a identificar a chamada no cliente que causou a violação.

Caso o interceptador do lado cliente esteja habilitado, é possível mapear a exceção CORBA enviada pelo servidor num erro de Lua que inclua as informações detalhadas da violação ocorrida.

Máquina de estados

O suporte ao uso de máquina de estados para a modelagem do comportamento de uma interface não contempla a possibilidade de compôr duas ou mais especificações de contratos desta natureza quando ocorre a herança de interfaces. O Tamago (ver 3.1), por exemplo, implementa um algoritmo que unifica estas máquinas de estado para gerar uma outra equivalente. A falta desta característica dificulta

a modelagem de estados quando ocorre a herança de interface, pois a máquina de estados deve ser copiada integralmente, o que aumenta a chance de inconsistências decorrentes de alterações do contrato.

Uma outra limitação é que apenas contratos de interface podem especificar estados e transições, pois ainda não há contratos de transição de estados para componentes. A implementação desta funcionalidade aumentaria ainda mais a necessidade de fundir os autômatos das facetas que integram o componente, para que o seu comportamento pudesse ser representado num autômato único. Porém, na implementação atual, os autômatos das facetas são independentes.

Sincronização

O suporte a contratos de sincronização no LUC, embora simplificado, é útil para experimentos com cenários em que a concorrência nas chamadas aos objetos representa um papel importante para o funcionamento correto do sistema. Porém, existem diversas implicações que derivam do uso de condições de sincronização em contratos de componentes.

Assim como no caso da avaliação de assertivas, a definição da exclusão mútua entre operações no contrato de sincronização pode causar um *deadlock* no sistema, pois caso ocorra uma *callback*, a nova chamada poderá ficar bloqueada indefinidamente. Para resolver este problema, seria necessário implementar um mecanismo de transação distribuída, em que cada operação aninhada fosse associada a algum rótulo que a identificasse como parte do contexto da chamada original. Isto permitiria evitar que uma *callback* fosse bloqueada pelo mecanismo de exclusão mútua, mesmo sendo parte do processamento da chamada que detém o controle da região crítica. Este tipo de bloqueio seria perpétuo (*deadlock*), pois nem a *callback* ou a chamada original poderiam ser completadas. Um exemplo de mecanismo de transação com estas características é o Jironde [40], do *framework* de componentes Fractal.

Um outro aspecto dos contratos de sincronização que pode representar uma limitação é a dificuldade de o objeto interagir com a infra-estrutura que implementa a sincronização. Por exemplo, caso haja uma chamada em exclusão mútua que invoque uma operação bloqueante, pode ser necessário liberar algum recurso ou seção crítica durante o período de suspensão. Sendo o controle de sincronização efetuado pelo subsistema de contratos e transparente para a implementação, nem sempre é possível para o objeto realizar esta liberação. Por outro lado, o mecanismo de sincronização é desacoplado da implementação, e por isso não não tem como realizar esta tarefa. Para tentar resolver essa questão, seria possível implementar uma interface no mecanismo de sincronização que pudesse ser chamada pelo objeto, o que tornaria o mecanismo

menos transparente.

A delegação do controle de concorrência para o contrato pode representar também um problema em relação à granularidade da sincronização efetuada, pois a exclusão passa a ser feita no nível de chamadas de métodos. Portanto, além do problema relatado no parágrafo anterior, o nível em que as operações de sincronização são feitas pode resultar no uso menos eficiente dos recursos compartilhados, bem como aumentar o tempo médio de bloqueio na espera por determinadas condições. Esse problema pode ter um impacto maior em objetos distribuídos, pois possuem uma "menor granularidade" para o bloqueio de sincronização e, geralmente, estão submetidos a uma maior concorrência.

Apesar disso, a distinção entre a implementação das funcionalidades de um objeto e a especificação das suas condições de sincronização promove o isolamento do código responsável pelo tratamento da concorrência, podendo reduzir a complexidade do código funcional e a incidência de erros. Os requisitos de sincronização passam a ser especificados no contrato e são garantidos pelo ambiente de execução. Esta separação está de acordo com os argumentos da programação orientada a aspectos, ou *aspect-oriented programming*, no sentido de promover uma maior modularidade das funcionalidades.

Outro benefício dos contratos de sincronização é a redução do conflito existente entre herança de implementação e sincronização. Ele ocorre ao implementar uma classe que deriva de uma base com sincronização explícita, pois ao implementar a classe derivada é necessário compatibilizar os requisitos de sincronização da nova classe com os métodos herdados, o que nem sempre é simples de ser feito. Este problema é conhecido nas linguagens orientadas a objeto concorrentes como "anomalia da herança"[41].