# 5
# Experimentos

Este capítulo apresenta alguns testes realizados para validar a solução implementada e experimentar as características de contratos de objetos e componentes distribuídos, tanto no nível comportamental como no de sincronização.

Os experimentos selecionados obedeceram ao critério de tentar explorar diferentes aspectos relacionados ao uso de contratos em objetos e componentes, de acordo com o que foi discutido ao longo do texto. Quando possível, foram utilizadas aplicações já existentes, mesmo que bem simples ou de demonstração, para evitar a criação de exemplos específicos para os testes em questão.

Cada seção a seguir descreve um experimento, incluindo considerações sobre os pontos mais relevantes de cada um no contexto desta dissertação. Entre as características abordadas, estão o uso de contratos em componentes, tratamento de erros e exceções, contratos de sincronização, uso de quantificadores lógicos com objetos remotos, especificação de transição de estados, entre outros.

Nos exemplos a seguir, alguns aspectos discutidos durante o texto não fizeram parte das características dos contratos apresentados. Entre os tópicos que não foram abordados nos experimentos encontram-se a avaliação de contratos no cliente, a influência da herança de objetos na verificação da correção de software e o uso de contratos nos receptáculos de componentes SCS.

## 5.1
## SCS - Componente Básico

Um dos experimentos selecionados para a elaboração de contratos foi o próprio sistema de componentes usado como base para a dissertação, o SCS. Como ainda é um sistema em evolução, a especificação de contratos para as suas interfaces ajudou a consolidar a implementação existente e colaborou até mesmo para a correção de defeitos e para a melhoria das interfaces existentes.

O SCS é dividido em duas partes principais: o modelo de componentes e a infra-estrutura de execução. O modelo de componentes é formado pelas interfaces básicas: *IComponent*, *IReceptacles* e *IMetaInterface*, já citadas no capítulo 4. A infra-estrutura de execução é formada pelos componentes *ExecutionNode*, *Container* e *ComponentRepository*. O *ExecutionNode* implementa o ponto de acesso ao SCS num computador da rede, e é responsável pela criação e manipulação de *Containers*. Estes, por sua vez, são os hospedeiros de componentes e oferecem o ambiente de execução com espaço de endereçamento reservado, promovendo maior robustez e organização lógica para a arquitetura dos sistemas. Já o *ComponentRepository* implementa um repositório de componentes, disponibilizando-os para carga e execução no contêiner.

O SCS foi um candidato natural para realizar experimentos com a elaboração de contratos, por se tratar de um projeto em uso em algumas aplicações e com diversas interfaces e componentes. Mesmo o SCS já estando em funcionamento, seus contratos foram elaborados posteriormente à sua implementação com o objetivo de avaliar os potenciais benefícios do uso de contratos em interfaces com suas características e as dificuldades envolvidas no processo.

Esta seção descreve as interfaces básicas de um componente SCS, sendo que outros componentes e interfaces do modelo serão tratados nas seções seguintes. Por uma questão de organização do documento, a listagem dos arquivos IDL do SCS encontra-se no apêndice A.



### 5.1.1 *IComponent*

A principal interface do modelo do SCS é a *IComponent*, cujo contrato genérico encontra-se na listagem 5.1. As operações *startup* e *shutdown* não foram incluídas porque suas condições dependem das particularidades de cada componente e, portanto, devem ser definidas no contrato do componente em questão.

A primeira operação é a `getFacet()`, que retorna a referência do objeto de uma faceta a partir do nome da interface. A pré-condição define que o parâmetro passado não deve ser vazio, que não é um nome de interface válido. Por se tratar de uma linguagem de tipos dinâmicos, poderia haver outras restrições de tipo, como verificar se o parâmetro é realmente uma *string*. Porém, como a chamada é feita através do *middleware* CORBA, o próprio ORB garante esta condição, pois a tentativa de passar um parâmetro de tipo inválido provoca uma exceção que indica erro no *marshalling*.

Seria possível incluir na pré-condição uma chamada a `string.match()` especificando um padrão (*pattern*) Lua para validar o formato do parâmetro com o nome da interface, de acordo com o padrão CORBA. Este seria um refinamento do sistema

Listagem 5.1: Contrato da interface *IComponent*

```
luc.contract.interface
{
    __interface = "::scs::core::IComponent",

    getFacet = {
        param_names = { "facet_interface" },
        pre = {
            param_not_empty = [[ string.len(facet_interface) > 0 ]],
        },
        post = {
            result_interface_matches_param = [[ RESULT == nil
                or IS_A(RESULT,facet_interface) ]],
            result_is_valid_facet = [[ RESULT == nil
                or EXISTS('f', FACETS, 'f.facet_ref == RESULT')
            ]]
        },
    },

    getFacetByName = {
        param_names = { "facet" },
        pre = {
            param_not_empty = [[ string.len(facet) > 0 ]],
        },
        post = {
            result_interface_matches = [[ RESULT == nil
                or RESULT == FACETS[facet] ]],
        },
    },

    __state = {
        states = { "initial", "started", "finished", },
        initial_state = "initial",

        transitions = {
            initial    = {
                startup = { condition="true", state="started" },
                shutdown = {},
            },
            started = {
                startup = {},
                shutdown = { condition="true", state="finished" },
            },
            finished = {
                startup = {},
                shutdown = {},
                getFacet = {},
                getFacetByName = {},
                getComponentId = {},
            },
        },
    },
}
```

de tipos para limitar as *strings* que podem ser passadas no parâmetro. De qualquer modo, caso seja passado um nome inválido o resultado será nulo.

A pós-condição especifica dois casos possíveis para o resultado da operação. No primeiro, o resultado é nulo, se a faceta não existir. No segundo, o objeto da faceta é retornado e a sua interface deve ser compatível com a especificada no parâmetro da chamada. A função `IS_A` verifica se o objeto retornado implementa a interface definida no arquivo IDL, percorrendo a hierarquia de interfaces.

A operação `getFacetByName()` é bastante semelhante à anterior, com a diferença de que o parâmetro passado é o nome simbólico da faceta. Portanto, é possível expressar a sua pós-condição em função do método `getFacet()`, verificando a consistência entre a implementação das operações, além de aumentar a precisão da semântica da especificação.

A modelagem de estados da interface *IComponent* especifica as operações que podem ser invocadas em cada estado do objeto, para garantir que a seqüência de operações realizadas sobre o componente seja válida. Por exemplo, a operação `startup()` só deve ser chamada uma vez, o que é garantido pela transição de estado especificada. Durante os testes realizados na pesquisa, esta verificação detectou seqüências incorretas nas chamadas de métodos, o que é comum de ocorrer, motivada por simples distração ou má interpretação da especificação.

Uma possível alteração futura para este contrato seria levar a definição das transições de estados do nível da interface da faceta para o nível do componente. Com isso, seria possível garantir, por exemplo, que nenhuma outra faceta fosse utilizada depois que a operação `shutdown()` do componente tivesse sido invocada.



### 5.1.2 *IReceptacles*

A interface *IReceptacles* ilustra aspectos interessantes do ponto de vista da elaboração de contratos, pois apresenta diversas exceções e dependências entre as operações que a compõem. Por se tratar de um contrato um pouco mais extenso, ele será listado por partes, uma para cada operação.

#### *IReceptacles::connect*

A primeira operação da interface *IReceptacles* é a `connect()`, e seu contrato encontra-se na listagem 5.2.

Listagem 5.2: Contrato da operação *IReceptacles::connect()*

```
connect = {
    param_names = { "receptacle", "obj" },
    pre = {
        receptacle_not_empty = "string.len(receptacle) > 0",
    },
    post = {
        check_valid_receptacle = [[
            RECEPTACLES[receptacle]
            or EXCEPTION['IDL:scs/core/InvalidName:1.0']
        ]],
        check_valid_obj = [[
            (obj ~= nil)
            or EXCEPTION[ 'IDL:scs/core/InvalidConnection:1.0' ]
        ]],
        check_compatible_interface = [[
            IS_A(obj,RECEPTACLES[receptacle].interface_id)
            or EXCEPTION[ 'IDL:scs/core/InvalidConnection:1.0' ]
        ]],
        check_conn_count = [[
            IMPLIES( PRE('#self:getConnections(receptacle)') > 0,
                RECEPTACLES[receptacle].is_multiplex )
            or EXCEPTION['IDL:scs/core/ExceededConnectionLimit:1.0']
        ]],
        check_duplicate_conn = [[
            IMPLIES( not EXCEPTION['IDL:scs/core/ExceededConnectionLimit:1.0']
                     and EXISTS('c',
                        PRE'self:getConnections(receptacle)',
                        'c.objref:_is_equivalent(obj)'),
                    EXCEPTION['IDL:scs/core/AlreadyConnected:1.0'] )
        ]],
    },
},
```

As assertivas para a operação `connect()` ilustram boa parte dos recursos disponíveis na linguagem de contratos do LUC como, por exemplo, *PRE*, *RECEPTACLES*, *EXISTS* e *EXCEPTION*.

A pré-condição especificada é bem simples, e apenas define que o nome do receptáculo passado como parâmetro não deve ser vazio, pois seria considerado inválido. Novamente, este tipo de pré-condição é um refinamento do tipo usado no parâmetro.

A pós-condição faz algumas referências à tabela *RECEPTACLES* para ter acesso às propriedades do receptáculo da instância do componente ao qual pertence. Como visto no capítulo 4, ela possui uma entrada para cada receptáculo especificado na interface do componente e descreve as suas propriedades, sendo definida em tempo de execução de acordo com o objeto sendo avaliado no contrato.

As assertivas apresentadas definem a semântica de cada exceção contida na interface do método, o que é útil tanto para o desenvolvedor que implementa a interface como para os usuários do modelo de componentes. Cada assertiva segue um padrão em que a condição de erro verificada é avaliada numa disjunção com a exceção que a sinaliza. Desta forma, verifica-se a consistência da implementação em relação à detecção do erro e ao lançamento da exceção correspondente.

O contrato da operação é avaliado através da conjunção das assertivas. Com isso, pode ser que nem todas sejam efetivamente avaliadas em função do "curto-circuito" da expressão booleana, pois a falha de uma das assertivas é suficiente para que as demais sejam dispensadas. Cria-se, assim, uma relação de dependência entre a ordem das assertivas e a ordem de verificação das condições de erro na implementação da interface, uma vez que os possíveis erros da chamada são confrontados com o resultado da operação. Em outras palavras, caso haja mais de um erro na chamada efetuada pelo cliente, apenas um deles será sinalizado por uma exceção e, se a ordem da verificação na implementação for diferente da especificada no contrato, será notificada uma violação, pois a exceção esperada será diferente da efetivamente lançada.

Neste exemplo, a situação do parágrafo anterior ocorre nas assertivas associadas aos rótulos `check_conn_count` e `check_duplicate_conn`. Na primeira, é feito o teste da seguinte condição: o fato de a quantidade de conexões previamente existentes ser maior que zero implica a multiplicidade do receptáculo. Caso contrário, a exceção que indica que o limite de conexões foi excedido deve ser lançada, pois a conexão em andamento não pode ser efetuada. A segunda assertiva verifica se o mesmo objeto foi conectado duas vezes ao receptáculo.

Eventualmente, é possível que as duas condições mencionadas se verifiquem simultaneamente, mas apenas uma delas será sinalizada por um lançamento de exceção. Para isso, basta que um cliente faça a chamada da operação `connect()` a um receptáculo simples (não múltiplo) duas vezes seguidas passando o mesmo objeto. As duas condições seriam violadas ao mesmo tempo e ambas as exceções poderiam ser lançadas.

Para contornar o problema neste caso específico, foi introduzido um teste na assertiva `check_duplicate_conn` sobre a ocorrência da exceção *ExceededConnectionLimit*, para detectar se já houve o lançamento desta exceção. Isto evita que ocorra uma violação de contrato nesta assertiva, pois houve o erro de conexão duplicada mas a exceção correspondente não foi lançada. A mesma situação pode ocorrer com outras assertivas. Por exemplo, caso ocorra uma conexão que exceda o limite e use um objeto incompatível com a interface esperada, seriam violadas as condições associadas a `check_compatible_interface` e `check_conn_count`.

A partir desta análise, é possível concluir que a implementação da interface deve seguir o padrão de verificação definido no contrato, caso contrário ela poderá causar uma violação mesmo que, a rigor, esteja correta. Este pode ser considerado um "efeito colateral" deste tipo de contrato.

Há ainda um outro fator que pode influenciar na incidência deste problema, pois a ordem em que as assertivas são avaliadas depende da implementação do algoritmo que percorre uma tabela em Lua. Ou seja, a ordem em que são definidas

no contrato não garante que a função `pairs()` de Lua irá retorná-las na mesma ordem. Isto pode ser minimizado caso a tabela das pós-condições seja transformada em *array*, bastando para isso omitir os rótulos em cada assertiva ou transformá-los em índices inteiros. Para resolver este problema específico, a implementação pode ser melhorada através da criação de um mecanismo que permita definir a ordem da avaliação das assertivas.

É importante lembrar que, mesmo sem mudar a especificação da interface, o contrato poderia adotar diferentes abordagens em relação à verificação da semântica das operações. No exemplo acima, o contrato tem seu foco mais voltado para a verificação da consistência da implementação em relação ao tratamento de erros. Uma abordagem alternativa seria verificar os parâmetros passados pelo cliente na pré-condição e notificar uma violação nas mesmas condições que geram as exceções apresentadas no contrato. Porém, haveria uma sobreposição de responsabilidades entre o contrato e a implementação da interface, o que não é desejável.

O experimento com as interfaces do SCS foi realizado posteriormente à sua implementação, com o sistema já funcional. Portanto, as interfaces já haviam sido definidas há algum tempo e encontravam-se em uso por aplicações. Não fosse este o caso, seria possível adotar uma estratégia mais alinhada com as recomendações da teoria de contratos, na qual a verificação dos erros do cliente da chamada seria feita nas pré-condições. Caso estas não fossem satisfeitas, seria notificada uma violação. Assim, o código da implementação da operação poderia dispensar os testes associados às exceções apresentadas, e ficaria voltado apenas para a funcionalidade principal. Porém, existe outra versão do SCS em Java que compartilha as interfaces com a versão Lua, e portanto as interfaces devem ser preservadas para que a compatibilidade seja mantida.

O contrato em questão, embora tenha contribuído para a verificação da implementação que já estava disponível, pouco faz para protegê-la de erros provenientes do cliente. Uma possibilidade seria adotar a estratégia do parágrafo anterior, eliminando o código da implementação que efetua as verificações de erro e deixá-las para o contrato. Esta seria uma abordagem mais interessante do ponto de vista de um sistema de componentes, uma vez que este encontre-se estável, pois o risco de defeitos volta-se mais para o lado do cliente. Por outro lado, o uso da infra-estrutura de contratos deixa de ser opcional para tornar-se parte integrante do sistema.

Um outro ponto que deve ser destacado em relação ao uso de contratos para esta operação é que ele permitiu a detecção de defeitos ainda existentes na implementação, pois a sua elaboração permitiu a avaliação sistemática das possíveis condições da chamada. Embora o SCS já estivesse em uso há algum tempo, havia erros em relação à semântica do tratamento das exceções previstas na interface que ainda não eram conhecidos.

### *IReceptacles::disconnect*

O contrato da operação `disconnect()` encontra-se na listagem 5.3. Ele é bem mais simples que o anterior, e adota uma abordagem semelhante em relação à verificação do tratamento de erros.

Listagem 5.3: Contrato da operação *IReceptacles::disconnect()*

```
disconnect = {
    param_names = { "id" },
    post = {
        valid_id_or_exception = [[
            PRE("PCALL(self.getConnection,self,id)[1]")
            or EXCEPTION['IDL:scs/core/InvalidConnection:1.0']
        ]],
        not_connected_anymore = [[
            PCALL(self.getConnection, self, id)[2][1]
                == 'IDL:scs/core/InvalidConnection:1.0'
        ]],
    },
},
```

Nesta operação, surge um novo recurso não usado na anterior, que é a chamada à função `PCALL`. Como o método referenciado na assertiva pode gerar uma exceção, a chamada deve ser protegida para não causar uma violação no contrato. Baseada na função `pcall` de Lua, ela recebe como parâmetro e função a ser chamada, seguida dos parâmetros. Seu retorno é uma tabela em que a primeira posição indica o status da operação, e os demais elementos são os retornos da função chamada. Por exemplo, a assertiva da pós-condição `PRE("PCALL(self.getConnection,self,id)[1]")` verifica se o identificador passado como parâmetro era válido antes da desconexão, pois efetua a chamada protegida do método `getConnection` e testa o status da operação. Se o identificador não era válido no momento da chamada, significa que a exceção `InvalidConnection` deve ter sido lançada..

A outra assertiva verifica se a conexão associada ao identificador passado deixou de existir depois da execução da operação, para garantir que a implementação cumpriu com a sua parte no contrato. Novamente, a chamada protegida é feita, porém desta vez ela testa a ocorrência de uma exceção, pois o identificador deve ter deixado de ser válido. Como o erro é esperado, a chamada a `PCALL` retorna uma tabela com a exceção lançada na segunda posição.

### *IReceptacles::getConnections*

A operação recebe o nome de um receptáculo e retorna um *array* com descritores das conexões existentes. A listagem 5.4 contém o seu contrato.

A pré-condição está comentada, pois a ela verifica se o nome de receptáculo passado como parâmetro é válido, o que é redundante em relação à exceção prevista

Listagem 5.4: Contrato da operação *IReceptacles::getConnections()*

```
getConnections = {
    param_names = { "receptacle" },
    pre = {
        -- valid_receptacle = "RECEPTACLES[receptacle]"
    },
    post = {
        result_valid_or_exception = [[
            RESULT ~= nil or EXCEPTION['IDL:scs/core/InvalidName:1.0']
        ]],
        check_result = [[
            #RESULT == RECEPTACLES[receptacle].numConnections
        ]],
        check_connections = [[
            FORALL('c', RESULT, 'RECEPTACLE[receptacle].connections(c.id) == c
                ' )
        ]],
    },
},
```

na interface. Portanto, ela serve apenas de exemplo de uma potencial pré-condição caso a verificação fosse transferida para o contrato, como discutido no exemplo da operação anterior.

A primeira assertiva da pós-condição verifica se houve lançamento da exceção *InvalidName*, e neste caso retorno deve ser nulo. Na segunda, é verificada a consistência do resultado com relação ao conteúdo da tabela RECEPTACLES. Na última, é feito o teste de consistência de cada descritor de conexão retornado com relação à operação `getConnection`, ou seja, a busca de cada um pelo seu ID deve retornar o próprio.

### *IReceptacles::getConnection*

Originalmente, o método `getConnection` não existia na interface do SCS. Ele foi introduzido durante a análise para a elaboração do contrato, pois detectamos que não era possível obter as informações de uma conexão a partir do seu ID. Este foi outro tipo de benefício obtido durante os experimentos: a verificação da consistência e completude das interfaces. O contrato para a operação está na listagem 5.5.

Listagem 5.5: Contrato da operação *IReceptacles::getConnection()*

```
getConnection = {
    param_names = {"id"},
    post = {
        valid_id_or_exception = [[
            EXCEPTION['IDL:scs/core/InvalidConnection:1.0']
            or (RESULT and
                EXISTS('c',
                    RECEPTACLES[RESULT.receptacle_name].connections,
                    "c.id == RESULT.id"))
        ]]
    },
},
```

A operação é semelhante à anterior, com a diferença de que retorna a descrição de apenas uma conexão, já que o relacionamento entre identificador e conexão é de um para um. Mais uma vez, a pós-condição verifica a consistência da implementação em relação ao possível erro no parâmetro passado pelo cliente. Ela usa o quantificador *EXISTS* sobre o retorno da operação `getConnections()` para garantir que o descritor de conexão retornado é válido, verificando a consistência entre as operações.

### Sincronização

A parte referente à sincronização do contrato da interface *IReceptacles* está na listagem 5.6.

Listagem 5.6: Contrato de sincronização da interface *IReceptacles*

```
__sync = {
    __mutex = {
        {'connect','disconnect'}
    },
}
```

A sincronização estabelece a exclusão mútua entre as operações `connect()` e `disconnect()`, já que elas manipulam as estruturas internas do objeto e alteram o seu estado.

A análise do código da implementação existente mostrou que esta condição do contrato é realmente necessária, pois pode haver a preempção do código do método `connect()`, uma vez que nele há uma chamada ao método `is_a()` do objeto passado para conexão. Ela faz parte do padrão CORBA e verifica se um objeto implementa uma determinada interface. Se, durante a chamada à `is_a()`, chegar uma nova requisição para a operação `connect()` ou `disconnect()`, pode haver uma condição de corrida na manipulação do estado interno do objeto, que poderá se tornar inconsistente. Assim, o uso do contrato de sincronização tornou a implementação existente mais robusta em relação à concorrência.

## 5.1.3 *IMetaInterface*

A interface *IMetaInterface* provê o mecanismo de reflexão de um componente SCS, e permite que um cliente obtenha as informações sobre as suas facetas e receptáculos.

Listagem 5.7: Contrato da interface *IMetaInterface*

```
luc.contract.interface
{
    __interface = "::scs::core::IMetaInterface",

    getFacets = {
        post = {
            consistent_with_FACETS = [[
                FORALL('f', RESULT, 'FACETS[f.name]')
            ]]
        },
    },
    getFacetsByName = {
        param_names = { "names" },
        pre = { param_not_nil = "names ~= nil", },
        post = {
            valid_results = [[
                EXCEPTION['IDL:scs/core/InvalidName:1.0']
                or FORALL('fn', names, "FACETS[fn] and EXISTS('f', RESULT,'fn
                    == f.name')")
            ]],
        },
    },
    getReceptacles = {
        post = {
            consistent_with_RECEPTACLES = [[
                FORALL('r', RESULT, 'RECEPTACLES[r.name]')
            ]]
        },
    },
    getReceptaclesByName = {
        param_names = {'names'},
        pre = { param_not_nil = "names ~= nil", },
        post = {
            valid_results = [[
                EXCEPTION['IDL:scs/core/InvalidName:1.0']
                or FORALL('rn', names,
                    "RECEPTACLES[rn] and EXISTS('r',RESULT,'rn == r.name')")
            ]],
        },
    },
}
```

Seu contrato é bem simples, e apenas valida os resultados das operações com as primitivas básicas de tratamento de componentes da linguagem de contratos. O contrato da interface encontra-se na listagem 5.7.

## 5.2 *SCS - Container*

Esta seção trata do *Container*, responsável pelo ambiente de execução dos componentes do SCS. Suas facetas serão descritas nas subseções a seguir.

### 5.2.1

### *ComponentLoader*

A interface *ComponentLoader* é uma das facetas do componente *Container*. Ela fornece os serviços para carregar e descarregar um componente, e seu contrato encontra-se na listagem 5.8.

Listagem 5.8: Contrato da interface *ComponentLoader*

```
luc.contract.component
{
    id = { name = "ComponentContainer", version = 1, },

    facets = {

        ComponentLoader = {
            __interface = "IDL:scs/repository/ComponentLoader:1.0",
            load = {
                param_names={"id", "args"},
                post = {
                  component_loaded_or_exceptions = [[
                    EXISTS( 'c',
                      FACETS.ComponentCollection.facet_ref:getComponent(id),
                      'c.instance_id==RESULT.instance_id')
                    or EXCEPTION['IDL:scs/container/ComponentAlreadyLoaded
                        :1.0']
                    or EXCEPTION['IDL:scs/container/ComponentNotFound:1.0']
                    or EXCEPTION['IDL:scs/container/LoadFailure:1.0']
                  ]] ,
                },
            },
            unload = {
                param_names={"handle"},
                post = {
                  check_if_was_loaded = [[
                    PRE("EXISTS('c',
                      FACETS.ComponentCollection.facet_ref:getComponents(),
                      'handle.instance_id==c.instance_id')")
                    or EXCEPTION['IDL:scs/container/ComponentNotFound:1.0']
                  ]],
                  not_loaded_anymore = [[
                    not EXISTS('h',
                        FACETS.ComponentCollection.facet_ref:getComponent(
                            handle.id),
                        'h.instance_id == handle.instance_id' )
                  ]],
              },
            },
        },

.....

}
```

O contrato desta interface utiliza o recurso da tabela *FACETS* para especificar o efeito da operação `load()` em função da faceta *ComponentCollection*, também pertencente ao *Container*. Assim como a *RECEPTACLES*, a tabela *FACETS* é montada dinamicamente e contém as facetas do componente ao qual o objeto corrente pertence. Estes recursos permitem especificar condições do contrato no nível de componente, deixando de atuar apenas de forma isolada nas interfaces.

Na operação `load()`, a pós-condição verifica se a instância do componente recém-carregada aparece na faceta *ComponentCollection*, como previsto na especificação do *Container*. Caso isto não aconteça, então alguma das exceções listadas deve ter sido lançada, sendo que o contrato não pode determinar a exceção exata porque isto depende dos detalhes internos da implementação.

O método `unload()` tem o efeito contrário do anterior, e libera uma instância de componente. A primeira assertiva da sua pós-condição determina se a exceção *ComponentNotFound* deveria ter ocorrido, em função da pré-existência do componente na lista da faceta *ComponentCollection*. A segunda assertiva serve para garantir que a instância passada como parâmetro para o método não deve mais existir após o término da operação de descarregamento.

### 5.2.2 *ComponentCollection*

A interface *ComponentCollection* é a faceta do *Container* que permite obter as informações sobre as instâncias de componentes carregadas, como já visto no contrato da interface *ComponentLoader*. O seu contrato está na listagem 5.9.

Neste contrato, surge a definição de uma função auxiliar denominada `compareIDs()`, cujo objetivo é facilitar a comparação das estruturas que identificam um componente, que é feita mais de uma vez no contrato. Ela ilustra a possibilidade de criar funções que permitam realizar operações repetitivas ou que possam facilitar a compreensão do contrato. Outra função adicionada é `belongs()`, para verificar se um elemento pertence a uma tabela.

A pós-condição das operações `getComponent()` e `getComponents()` referenciam-se mutuamente, formando um ciclo na avaliação. Assim, a especificação dada reforça a consistência entre as operações mas não tem como verificar de forma absoluta o seu resultado, pois apenas a primeira iteração de chamadas do ciclo é avaliada.

No contrato da operação `getComponents()`, a completude do resultado não pode ser testada pois não há como verificar se todos os componentes carregados foram realmente retornados pelo método.

Seria interessante efetuar a exclusão mútua entre estas operações e as operações da faceta *ComponentLoader*, pois elas manipulam o estado do contêiner para armazenar ou liberar os componentes carregados. Porém, a especificação de contratos de sincronização entre facetas ainda não se encontra disponível na versão corrente do sistema.

Listagem 5.9: Contrato da interface *ComponentCollection*

```
ComponentCollection = {
    __interface = "IDL:scs/repository/ComponentCollection:1.0",
    __functions = {
        compareIDs = function(id1,id2) return id1.name==id2.name and id1.
            version==id2.version end,
        belongs =
            function(x, collection)
                for _,v in ipairs(collection) do
                    if v == x then return true end
                end
                return false
            end
    },
    getComponent = {
        param_names = { "id" },
        pre = {
            valid_id = "string.len(id.name) > 0 and id.version "
        },
        post = {
            results_match_id = [[
                FORALL('c', RESULT, 'compareIDs(c.id,id)')
            ]],
            no_invalid_result = [[
                not EXISTS('c', RESULT, 'not compareIDs(c.id,id)')
            ]],
            all_components_returned = [[
                not EXISTS('c', self:getComponents(), 'compareIDs(c.id,id) and
                     not belongs(c,RESULT)' )
            ]]
        },
    },
    getComponents = {
        post = {
            all_components_valid = [[
                FORALL('c', RESULT, 'EXISTS("c2", self:getComponent(c.id), "
                    compareIDs(c.id,c2.id)")' )
            ]],
        }
    }
},
```

## 5.3
## *SCS - ExecutionNode*

Para completar os principais contratos do SCS, a listagem 5.10 apresenta a definição do contrato da interface *ExecutionNode*, responsável pela criação e manipulação dos contêineres de componentes. Ela é a principal faceta do componente de mesmo nome.

O método `startContainer()` cria um novo contêiner associado ao nome passado como parâmetro. Ele pode lançar a exceção *ContainerAlreadyExists* caso já exista outro contêiner com o mesmo nome, e a primeira pós-condição especifica este comportamento. Porém, pode haver outros erros na carga de um contêiner, e neste caso o retorno da chamada é nulo, e não há exceção prevista para eles. A segunda pós-condição especifica o efeito positivo do método: caso o retorno seja não nulo, deve ser consistente com o resultado da chamada a `getContainer()` para o mesmo nome passado.

O método `killContainer()` deve lançar a exceção *InvalidContainer* caso o contêiner especificado não exista. Caso contrário, seu efeito é inverso ao do método anterior, pois o contêiner especificado não deve mais aparecer na lista do método `getContainer()`.

A exemplo de outros contratos anteriores, as consultas `getContainer()` e `getContainers()` são definidas mutuamente para reforçar a consistência entre seus resultados. Assim como ocorre na `ComponentCollection::getComponents()`, não é possível garantir a completude da operação `getContainers()` pois esta informação pertence à implementação subjacente.



## 5.4
## *SCS - EventService*

O serviço de eventos já existia na versão Java do SCS, e foi implementado também para a versão Lua. Ele é baseado no uso de canais para a distribuição de eventos num mecanismo de *push*, em que os objetos se registram para receber passivamente a notificação de eventos. As interfaces do serviço encontram-se no apêndice A. A figura 5.1 ilustra o mecanismo de funcionamento do canal de eventos.

O componente central deste serviço é o *EventManager*, composto pelas facetas *ChannelFactory* e *ChannelCollection*. A primeira permite a criação e destruição de canais, enquanto a segunda oferece informações sobre os canais existentes.

As interfaces das facetas deste exemplo encontram-se no apêndice A. O

Listagem 5.10: Contrato da interface *ExecutionNode*

```
luc.contract.interface
{
    __interface = "::scs::execution_node::ExecutionNode",

    startContainer = {
        param_names={"container_name", "props"},

        pre = {
            container_name_not_empty = [[ string.len(container_name) > 0 ]] ,
        },

        post = {
            container_already_exists = [[
                IMPLIES( PRE"self:getContainer(container_name)",
                  EXCEPTION['IDL:scs/execution_node/ContainerAlreadyExists
                      :1.0'])
            ]],
            container_created = [[
                IMPLIES( RESULT, self:getContainer(container_name):
                    _is_equivalent(RESULT) )
            ]],
        }
    },

    killContainer = {
        param_names = {"container_name"},
        post = {
            valid_container_name = [[
                PRE'self:getContainer(container_name)'
                or EXCEPTION['IDL:scs/execution_node/InvalidContainer:1.0']
            ]],
            container_killed = [[
                not self:getContainer(container_name)
            ]],
        },
    },

    getContainer = {
        param_names = {"container_name"},
        pre = {
            container_name_not_empty = [[ string.len(container_name) > 0 ]] ,
        },
        post = {
            returns_valid_container = [[
                not RESULT
                or EXISTS("c", self:getContainers(), "c.container_name ==
                    container_name and RESULT:_is_equivalent(c.container)")
            ]],
        },
    },

    getContainers = {
        post = {
            all_valid_containers = [[
                FORALL("c", RESULT, "self:getContainer(c.container_name)")
            ]],
        },
    },
}
```

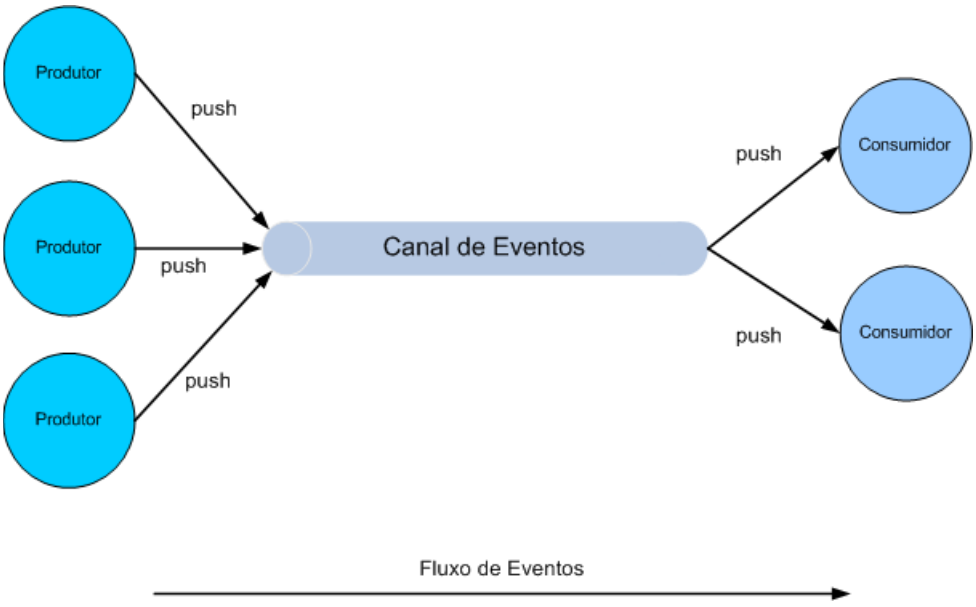


Figura 5.1: Funcionamento do canal de eventos

contrato para o componente *EventManager* está na listagem 5.11.

Assim como outros exemplos do SCS, o contrato está mais voltado para a verificação da consistência da implementação, já que os possíveis erros das chamadas do cliente são sinalizadas por exceções definidas no arquivo IDL.

O outro componente do serviço é o canal de eventos (*EventChannel*), que possui uma faceta de interface *EventSink* usada para a operação `push()` de um evento. Ele também possui um receptáculo múltiplo de nome *EventSource*, no qual os objetos interessados em receber os eventos do canal podem se registrar. Os objetos devem implementar a mesma interface *EventSink* para receber os eventos enviados pelo canal. O contrato do *EventChannel* está na listagem 5.12.

O contrato do componente *EventChannel* é um exemplo que ilustra a dificuldade que pode ocorrer ao descrever o comportamento de uma interface com poucos elementos públicos. A interface *EventSink* possui apenas duas operações, `push()` e `disconnect()`. Para descrever o seu comportamento, o conteúdo da interface não é suficiente, pois faltam elementos (atributos e métodos) que permitam expressar os efeitos das operações sobre o canal de eventos. Assim, apenas a definição das facetas e receptáculos foram incluídas no contrato do componente, além da exclusão mútua entre as operações da faceta *EventSink*.

Como este é um contrato de componente, ele deve conter a definição do seu ID, das facetas e dos receptáculos, através das chaves *id*, *facets* e *receptacles*, respectivamente.

Listagem 5.11: Contrato do componente *EventManager*

```
luc.contract.component
{
  id = { name = "EventManager", version = 1, },

  facets = {
    ChannelFactory = {
      __interface = "::scs::event_service::ChannelFactory",

      create = {
        param_names = {'name'},
        pre = {
          name_not_empty = [[ string.len(name) > 0 ]],
        },
        post = {
          check_exception = [[
            IMPLIES(
              PRE('FACETS.ChannelCollection.facet_ref:getChannel(name)'),
              EXCEPTION['IDL:scs/event_service/NameAlreadyInUse:1.0'] )
          ]],
          consistent_with_collection = [[
            not RESULT
            or RESULT:_is_equivalent( FACETS.ChannelCollection.facet_ref:
                getChannel(name) )
          ]],
        },
      },

      destroy = {
        param_names = {'name'},
        pre = {
          name_not_empty = [[ string.len(name) > 0 ]],
        },
        post = {
          check_exception = [[
            IMPLIES(
              not PRE('FACETS.ChannelCollection.facet_ref:getChannel(name)'),
              EXCEPTION['IDL:scs/event_service/InvalidName:1.0']
          ]],
          destroyed = [[
            not FACETS.ChannelCollection.facet_ref:getChannel(name)
          ]]
        },
      },

      __sync = {
        __mutex = {'create','destroy'},
      },
    },

    ChannelCollection = {
      __interface = "::scs::event_service::ChannelCollection",

      getChannel = {
        param_names = {'name'},
        pre = {
          name_not_empty = [[ string.len(name) > 0 ]]
        },
        post = {
          consistent_with_getAll = [[
            not RESULT or EXISTS( 'c', self:getAll(), 'c.name==name and c==
                RESULT' )
          ]],
        },
      },
      getAll = {
        post = {
          consistent_with_getChannel = [[
            FORALL('c', RESULT, 'self:getChannel(c.name) == c.channel')
          ]],
        },
      },
    }
  },
}
```

Listagem 5.12: Contrato do componente *EventChannel*

```
luc.contract.component
{
    id = { name = "EventChannel", version = 1, },

    facets = {
        EventSink = {
            __interface = "::scs::event_service::EventSink",
            __sync = { __mutex = {'push','disconnect'}, },
        },
    },

    receptacles = {
        EventSource = "IDL:scs/event_service/EventSink:1.0",
    },
}
```

## 5.5 *PingPong*

O *PingPong* é um componente de demonstração do SCS que, apesar de muito simples, serve para ilustrar algumas funcionalidades do sistema de contratos quando aplicados a componentes SCS. Ele é composto de uma faceta também denominada *PingPong*, cuja interface está na listagem 5.13. O componente também possui uma faceta *IReceptacles* cujo único receptáculo chama-se *PingPongRec* e aceita conexões de objetos desta mesma interface. A figura 5.2 ilustra a conexão entre os componentes e as chamadas de suas operações.

Listagem 5.13: Interface da faceta *PingPong*

```
interface PingPong {
    void setId(in long identifier);
    long getId();
    void ping();
    void pong();
};
```

O funcionamento deste exemplo consiste em conectar dois componentes *PingPong* através dos seus receptáculos para que possam trocar mensagens entre si através de chamadas alternadas aos métodos `ping()` e `pong()`. Quando os componentes são iniciados, o primeiro chama o método `ping()` do objeto *PingPong* conectado ao seu receptáculo que, por sua vez, responde com a chamada a `pong()`. Depois disso, eles invertem o papel, e o segundo chama a operação `ping()` do primeiro, e o ciclo se repete indefinidamente. O contrato do componente está na listagem 5.14.

O contrato redefine a operação `IComponent::startup()` com o intuito de verificar se a conexão do receptáculo já foi feita antes de o componente ser iniciado. Isto garante a semântica da operação do componente na especificação, pois se o cliente esquecer de conectar os receptáculos a aplicação simplesmente não faz nada. Neste

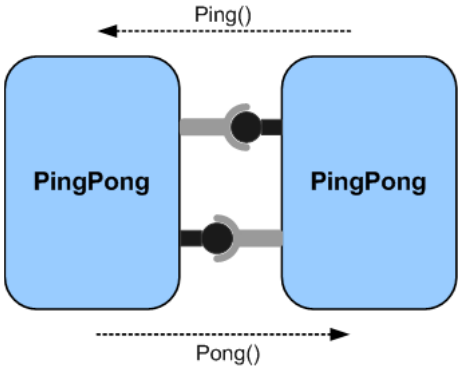


Figura 5.2: Componentes *PingPong*

caso, não seria possível criar uma exceção específica para sinalizar o problema para o cliente porque o método é genérico, uma vez que pertence à interface *IComponent*.

Neste exemplo específico, a falta da notificação sobre o problema da dependência externa não resolvida no `startup()` não é algo complicado de detectar e corrigir, devido à simplicidade da aplicação. Porém, num sistema mais complexo, o uso de contratos exerce um papel importante para identificar inconsistências desta natureza.

A definição do contrato da faceta não inclui pré ou pós-condições das operações, apenas a exclusão mútua entre elas e a modelagem dos seus estados. Neste caso, os estados alternam entre as operações esperadas, ou seja, depois de um `ping` deve ocorrer um `pong`, e vice-versa. O mecanismo de sincronização garante que apenas uma delas ocorre de cada vez, para que a chamada alternada dos métodos obedeça à seqüência prevista.

## 5.6 *Produtores e Consumidores*

O experimento dos produtores e consumidores é baseado no conhecido problema de concorrência em que um *buffer* de capacidade limitada é compartilhado entre dois tipos de tarefas: inserção de elementos pelos produtores e remoção de elementos pelos consumidores, realizadas de forma assíncrona e concorrente. Para garantir o bom funcionamento do sistema, é necessário vincular a inserção de elementos à condição de o *buffer* não estar cheio e, por outro lado, a remoção de elementos só deve ser feita caso o *buffer* não esteja vazio.

Este exemplo, apesar de simples, ilustra bem o uso das pré-condições de sincronização. Elas controlam a concorrência no acesso ao *buffer*, bloqueando as chamadas de acordo com a necessidade. A listagem 5.15 contém a definição da interface do *buffer* compartilhado.

Listagem 5.14: Contrato do componente *PingPong*

```
luc.contract.component
{
    id = { name = "PingPong", version = 1 },

    IComponent = {
        startup = {
            pre = {
                connected = "RECEPTACLES.PingPongRec.numConnections == 1"
            },
        },
    },
    facets = {
        PingPong = {
            __interface = "IDL:scs/demos/pingpong/PingPong:1.0",
            __state = {
                states = { "initial", "ping", "pong" },
                __initial_state = "initial",
                transitions = {
                    initial = {
                        ping = {condition='true', state='pong'},
                        pong = {condition='true', state='ping'},
                    },
                    ping = {
                        ping = {condition='true', state='pong'},
                        pong = {},
                    },
                    pong = {
                        ping = {},
                        pong = {condition='true', state='ping'},
                    }
                }
            },
            __sync = {
                __mutex = {'ping','pong'},
            },
        }
    },
    receptacles = {
        PingPongRec = "IDL:scs/demos/pingpong/PingPong:1.0",
    },
}
```

Listagem 5.15: Interface *Buffer*

```
module demo {
    interface Buffer {
        void put(in string s);
        string get();
        long maxSize();
        long size();
        boolean empty();
        boolean full();
    };
};
```

Listagem 5.16: Contrato da interface *Buffer*

```
luc.contract.interface
{
    __interface = "::demo::Buffer",
    __inv = {
        inv = {
            size_limits = [[
                self:size() >= 0 and self:size() <= self:maxSize()
            ]],
            empty_implies_size_zero = [[
                IMPLIES( self:empty(),
                    self:size()==0 and not self:full() )
            ]],
            full_implies_maxSize = [[
                IMPLIES( self:full(),
                    self:size()==self:maxSize() and not self:empty() )
            ]],
        },
    },
    __sync =
    {
        __mutex = { 'put','get' },
        put = { pre='not self:full()' },
        get = { pre='not self:empty()' },
    },
}
```

As operações `get()` e `put()` são bloqueantes, ou seja, a *thread* da requisição é suspensa caso a operação não possa ser efetuada imediatamente. Estas condições são especificadas no contrato da interface, contido na listagem 5.16.

As cláusulas do contrato de sincronização definem a exclusão mútua entre as operações `put()` e `get()`, além das suas pré-condições, que funcionam como condições de guarda para permitir o início da execução das operações.

As assertivas da invariante da interface estabelecem a relação entre as consultas `full()`, `empty()`, `maxSize()` e `size()`, garantindo a consistência do objeto a respeito da semântica destas operações.

Como visto no capítulo 4, a combinação entre a exclusão mútua e a pré-condição de sincronização determinam que, ao tentar executar uma operação, a tarefa que trata a requisição corrente no servidor só inicia a execução se conseguir entrar na região crítica e se a avaliação da condição for verdadeira. Caso contrário, ela deixa a região crítica e entra numa fila de *threads* suspensas.

Quando uma rotina da interface termina, ela permite que as *threads* que estão aguardando na fila testem novamente a pré-condição para tentar iniciar a operação bloqueada.

Este experimento serviu para testar uma situação em que o mecanismo de sincronização está contido no contrato, exercendo um papel tanto de especificação como de implementação, uma vez que o controle da concorrência é exercido pelo mecanismo de suporte a contratos. Neste caso, o contrato faz parte da aplicação e não é opcional como no caso dos contratos comportamentais.

Com este tipo de contrato, o código que implementa o objeto *Buffer* é isento de verificações das condições `empty` e `full` nos métodos `put` e `get`. Além disso, ele também dispensa o código de exclusão mútua entre as operações, sendo voltado apenas para a implementação das suas funcionalidades básicas. Uma vantagem imediata desta separação é o fato de que o código da implementação do *Buffer* é o mesmo tanto para execução concorrente como para execução seqüencial.

A listagem 5.17 apresenta parte do código da implementação da interface com as operações `put()` e `get()`. Ele corresponde a uma implementação de *buffer* circular sobre um vetor, com uma posição vazia para permitir a diferenciação entre cheio e vazio.

Listagem 5.17: Implementação das operações `Buffer::get()` e `Buffer::put()`

```
function Buffer:put( s )
    self.buf[self.next] = s
    self.count = self.count+1
    self.next = (self.next % self.max) + 1
    print("buffer count: " .. self.count)
    if self.observer then self.observer:update(self) end
end

function Buffer:get()
    local result = self.buf[self.oldest]
    self.oldest = (self.oldest % self.max) + 1
    self.count = self.count-1
    print("buffer count: " .. self.count)
    if self.observer then self.observer:update(self) end
    return result
end
```

As chamadas `self.observer:update(self)` notificam um observador remoto a cada inserção ou remoção no *buffer*, e servem para testar o funcionamento do mecanismo de exclusão mútua das operações no servidor, uma vez que elas permitem a troca de contexto para outras *threads* dentro do modelo de multitarefa cooperativa do OiL. Durante esta troca de contexto, nenhuma outra *thread* deve ser capaz de inserir ou remover elementos, já que existe uma outra na região crítica.

## 5.7 *Echo*

Este experimento foi realizado com o objetivo de medir o impacto dos mecanismos de avaliação de assertivas com quantificadores lógicos sobre o desempenho do sistema. Foi criada uma interface denominada *Echo* que, como o nome indica, apenas ecoa as *strings* passadas como parâmetro. As interfaces criadas estão na listagem 5.18.

Listagem 5.18: Interfaces do exemplo *Echo*

```
module echo {
    interface MyString {
        void add( in string s );
        string get();
        MyString sub(in long from, in long to);
        long count();
    };

    interface StringHome {
        MyString create( in string s );
        void destroy(in MyString s);
    };

    typedef sequence<MyString> StringSeq;

    interface Echo {
        attribute string last;
        string echo( in string what );
        string echoAll( in StringSeq seq );
    };
};
```

A interface e o programa usados no teste são intencionalmente simples, e servem apenas para medir o custo da avaliação de uma assertiva sobre uma seqüência de objetos distribuídos, comparando as abordagens de implementação dos quantificadores lógicos.

O teste realizado utilizou três computadores. No primeiro, foram executados o cliente (*echo_client.lua*) e o servidor principal (*echo_server.lua*). Ele possui um processador AMD Athlon 64 X2, com 2 GB de memória RAM e sistema operacional Windows 2003 Server. Os outros dois computadores utilizados executaram servidores de objetos tipo *MyString*, e ambos são equipados com processador AMD Athlon 2000+, contêm 1 GB de memória RAM e sistema operacional Windows 2000 Server. Em todos os computadores, o ambiente de teste era composto pela versão 5.1.3 da linguagem Lua e a versão 0.4 do OiL.

O primeiro passo do teste realizado é a criação dos diversos objetos de tipo *MyString* nos dois servidores remotos. Esta tarefa é realizada pelo módulo *echo_client.lua*, que cria metade do número de elementos em cada servidor e preenche o *array* com os objetos criados, alternadamente. Em seguida, é feita uma chamada à operação `Echo::echoAll()`, passando o *array* criado como parâmetro. O método chamado concatena todas as *strings* passadas e retorna como resultado da chamada.

As variáveis usadas para diferenciar os testes são o tamanho da seqüência de objetos criada e a técnica de avaliação do quantificador a ser utilizada (seqüencial, *threads* ou *deferred*). O contrato especificado para o exemplo (listagem 5.19) utiliza o *FORALL* nas assertivas das pré-condições, que expressam um teste simples para verificar o tamanho das *strings* passadas. Para a sua avaliação ser bem sucedida é preciso percorrer toda a seqüência. Como os parâmetros utilizados satisfazem a condição especificada, não ocorre nenhuma violação de contrato nos testes.

Listagem 5.19: Contrato da interface *Echo*

```
luc.contract.interface
{
    __interface = "::echo::Echo",

    __forall = 'sequential',

    echo = {
        param_names={ "what" },

        pre= {
            not_empty = [[ string.len(what) > 0 ]],
        },

        post = {
            param_in_result = [[ RESULT == what ]],
        },
    },

    echoAll = {
        param_names={ "seq" },

        pre= {
            seq_not_empty = [[ #seq > 0 ]],
            forall_test = [[ FORALL("x", seq, "x:count() > 1") ]],
        }
    },
}
```

Os resultados de desempenho obtidos nas execuções realizadas estão na figura 5.3. O gráfico mostra o tempo em segundos da execução da chamada a `Echo::echoAll()` para três tamanhos de seqüência de objetos (100, 200 e 500). Os tempos apresentados são a média de cinco execuções sucessivas para cada configuração do teste, variando o tipo de quantificador usado na avaliação. Foi incluído também o teste sem contratos, em que o interceptador do OiL não foi habilitado, para comparar o impacto do uso do contrato sobre o desempenho.

Os tempos médios de execução de cada teste (em segundos) estão na tabela 5.1. O desvio padrão dos tempos medidos estão entre parênteses na tabela. O valor do desvio padrão confirma a pequena dispersão dos tempos medidos, o que indica um comportamento homogêneo dos testes realizados.

| Tipo de Teste | Seq. 100 | Seq. 200 | Seq. 500 |
|---|---|---|---|
| FORALL Seqüencial | 49,37 (0,78) | 98,06 (1,25) | 244,62 (1,92) |
| FORALL Threads (5) | 29,81 (0,41) | 59,03 (0,34) | 147,88 (2,01) |
| FORALL Threads (50) | 25,25 (0,17) | 51,14 (0,78) | 126 (1,49) |
| FORALL Threads (100) | 24,97 (0,09) | 49,99 (0,67) | 124,36 (1,69) |
| FORALL Deferred | 37,35 (0,35) | 73,95 (0,69) | 185,08 (1,24) |

Tabela 5.1: Tempo médio dos testes e seu desvio padrão

A partir do gráfico, é possível perceber que o pior tempo foi obtido ao se utilizar o contrato com a avaliação seqüencial das chamadas. Isto está de acordo com o esperado, pois neste caso não há nenhum paralelismo na verificação das assertivas, e o grande número de chamadas remotas adicionais e em seqüência aumenta bastante o tempo total de execução do programa. O resultado mostra também que o tempo

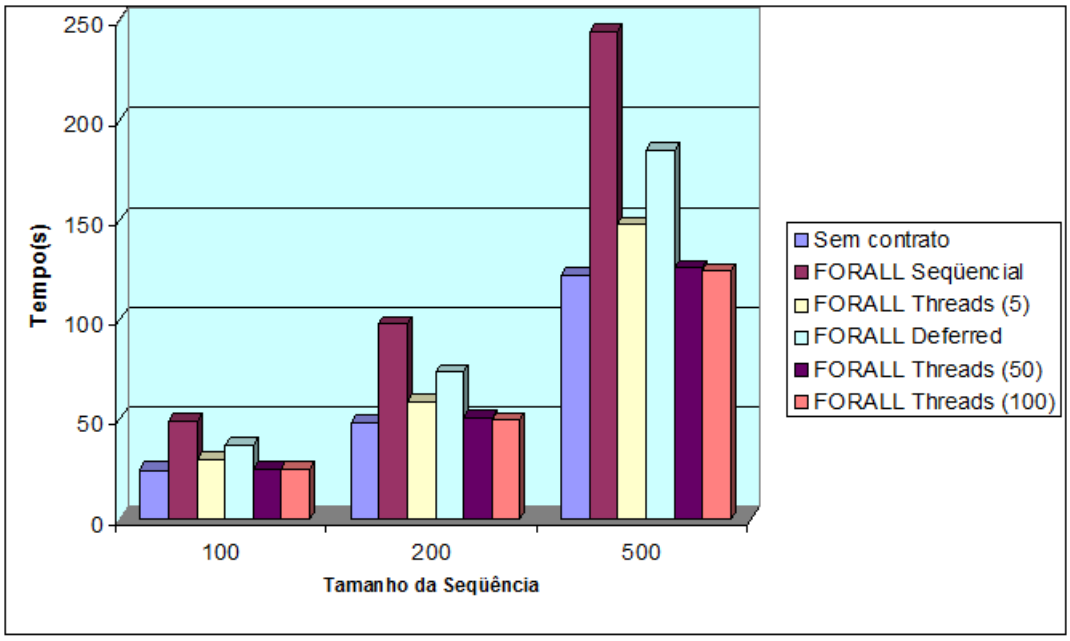


Figura 5.3: Comparativo de desempenho de contrato com FORALL

de execução em cada teste varia linearmente conforme o aumento de tamanho da seqüência.

A tabela 5.2 apresenta o desempenho das variantes dos testes em comparação com a execução sem contratos. Cada valor representa a razão entre o tempo do teste em questão e o teste sem contratos.

| Tipo de Teste | Seq. 100 | Seq. 200 | Seq. 500 |
|---|---|---|---|
| FORALL Seqüencial | 1,98 | 2,01 | 1,99 |
| FORALL Threads (5) | 1,20 | 1,21 | 1,21 |
| FORALL Threads (50) | 1,01 | 1,05 | 1,03 |
| FORALL Threads (100) | 1,00 | 1,03 | 1,01 |
| FORALL Deferred | 1,50 | 1,52 | 1,51 |

Tabela 5.2: Tempos relativos à execução sem contratos

A análise dos resultados mostra que o impacto no desempenho pode ser significativo quando a assertiva sendo avaliada envolve a verificação de objetos remotos de forma seqüencial. O paralelismo nas verificações possibilitou uma grande redução do tempo total de execução.

Deve-se observar também que o fato de apenas dois processos servirem os objetos *MyString* usados no testes reduz a possibilidade de paralelismo entre as chamadas, pois o processamento das requisições é atômico e não dá chance de outras requisições serem tratadas simultaneamente. Isto significa que, caso fossem utilizados mais processos servidores, o resultado no teste que utiliza *threads* poderia ser ainda mais favorável, embora o resultado obtido seja bastante satisfatório.

É importante ressaltar que nem sempre as avaliações de assertivas dos con-

tratos devem ser feitas em paralelo, pois elas são mais favoráveis apenas quando as expressões envolvem chamadas a objetos remotos. No caso de avaliações com parâmetros de tipos simples, como *long* ou *boolean*, o uso de *threads* não ajuda a melhorar o desempenho porque não há espera pelo resultado de chamadas remotas, e portanto o paralelismo não contribui para a redução do tempo de execução.

Outra observação a partir dos resultados é que o aumento do número de *threads* só ajuda a melhorar o desempenho até um determinado limite, a partir do qual não há mais ganhos. Para os testes realizados, não houve praticamente diferença entre os que utilizaram 50 e 100 *threads*. Assim, a tendência é que, para seqüências muito grandes, o aumento do número de *threads* na mesma proporção possa piorar o desempenho, devido ao custo adicional provocado pela troca excessiva de contexto e pelo aumento de consumo de recursos do sistema.

## 5.8 Conclusões

Durante a realização dos experimentos em questão, foram observados diversos aspectos positivos do uso de contratos nas interfaces específicas de cada caso. Algumas incompletudes e incorreções foram identificadas e corrigidas, principalmente devido à análise mais detalhada de cada operação das interfaces e seus possíveis efeitos. Isto ocorreu, por exemplo, na elaboração do contrato da interface *IReceptacles*, descrita na subseção 5.1.2. Por outro lado, houve casos em que os contratos foram de difícil elaboração, ou de pouca utilidade, como no exemplo do *EventChannel*, na seção 5.4.

Apesar de o conjunto de exemplos apresentados ser reduzido, foi possível observar alguns benefícios do uso de contratos no contexto de componentes, tais como o auxílio na correção de defeitos da implementação corrente do SCS, bem como a compreensão mais ampla da semântica dos seus componentes. Além disso, ao expressar as características das facetas de um componente e a relação entre elas, reduz-se a sua ambigüidade, o que favorece a utilização mais correta do componente por parte dos seus clientes.

Mesmo com as limitações apresentadas no capítulo 4, o uso de contratos mostrou-se uma ferramenta útil nos diferentes cenários que surgiram durante a elaboração dos componentes e interfaces descritos.