# 3
# O Modelo de Componentes Proposto

Neste capítulo iremos apresentar o modelo desenvolvido para prover suporte a múltiplas versões. A estrutura deste capítulo é tal que na seção 3.1 apresentamos os pontos fundamentais para prover o suporte a múltiplas versões. Na seção 3.2, nós apresentamos a proposta de modelo com esse suporte, resultado da extensão do modelo tradicional de componentes. Em seguida, na seção 3.3, apresentamos os possíveis cenários de uso do modelo proposto, e na seção 3.4 apresentamos uma implementação do modelo proposto com base no sistema de componentes SCS. Por fim, na seção 3.5, apresentamos as considerações finais do capítulo, cruzando a definição do modelo com os pontos fundamentais levantados, e apresentando as limitações do modelo e da implementação.

## 3.1
## Requisitos do Suporte a Múltiplas Versões

Esse trabalho se originou num estudo sobre o sistema Openbus [30], pois o sistema passou a ter como demanda conciliar múltiplas versões de interfaces de alguns serviços oferecidos. Essa demanda surgiu a partir da atualização da versão 1.4 para a versão 1.5 do Openbus, que são exatamente as versões utilizadas no estudo de caso apresentado no capítulo 4. No apêndice B apresentamos uma breve descrição desse sistema.

Dada essa demanda do Openbus, realizamos um estudo para compreender os tipos de modificações realizadas sobre o sistema, e as estratégias de implementação da compatibilização entre as versões suportadas. Como conclusão desse estudo, realizamos um levantamento dos requisitos necessários para poder realizar um suporte a múltiplas versões de interfaces em componentes de software.

Os requisitos são os pontos que identificamos como fundamentais para prover o suporte a múltiplas versões em um modelo de componentes de software, e existem independentemente das particularidades das decisões de projeto definidas e das estratégias de implementação utilizadas. Os requisitos identificados são listados a seguir:

1. A abstração de componente de software deve explicitar o relacionamento entre as versões das interfaces oferecidas por um componente.

   A abstração utilizada na representação do modelo precisa contextualizar as versões de maneira que mostre que existe uma relação entre as versões e o serviço prestado pela faceta.

2. O modelo deve prever um estado compartilhado entre as versões de um mesmo serviço.

   O modelo precisa permitir, mas não obrigar, que as diferentes versões de um mesmo serviço possam utilizar um mesmo estado compartilhado entre elas.

3. O modelo precisa permitir que um receptáculo aceite conexões de múltiplas versões de um mesmo serviço.

   O modelo precisa permitir que um receptáculo seja capaz de aceitar conexões de um mesmo serviço em diferentes versões. Com isso, também deve ser possível especificar uma coleção de versões com as quais o receptáculo é compatível.

4. Simplificar a realização das conexões.

   Conexões deveriam ser especificadas pelo configurador do sistema de uma forma mais abstrata, sem ter que necessariamente lidar com as versões de uma interface individualmente. Porém, a conexão só é realizada com sucesso se existir alguma interseção entre o conjunto de versões oferecidas pelo serviço e o conjunto de versões compatíveis do receptáculo.

5. Direcionamento das chamadas para as versões desejadas.

   Esse é um requisito de tempo de execução, para a comunicação entre os componentes, onde o modelo tem que permitir que seja identificada para qual versão a mensagem se destina.

6. Diferenciar os nomes das versões de uma interface, de forma a evitar conflitos de nomes na linguagem de programação utilizada.

   Para que seja possível que diferentes versões de uma mesma interface possam coexistir em uma mesma implementação de um componente de software, é necessário realizar alguma diferenciação nos nomes das interfaces para evitar conflitos no nível da linguagem de programação.

## 3.2 O Modelo Estendido

Como foi visto no capítulo 2, modelos de componentes de software tradicionais se baseiam em quatro conceitos principais: componentes, interfaces, facetas e receptáculos. Nesses modelos, tipicamente o termo *serviço* é utilizado tanto para representar a instância do componente quanto para representar a funcionalidade oferecida pelas facetas, não havendo uma distinção clara entre esse dois significados. Neste trabalho, passamos a adotar as definições enunciadas a seguir, fazendo uma analogia com conceitos bem estabelecidos de linguagens orientadas a objetos.

**Definição 1:** Componente representa uma implementação e suas interfaces, da mesma forma que uma classe representa a definição (implementação) de uma entidade em uma linguagem orientada a objetos.

**Definição 2:** Serviço é uma instância de um componente, da mesma forma que um objeto representa uma instância de uma classe.

Com a necessidade de evolução dos sistemas e de seus componentes, passa a existir a possibilidade de versões diferentes de uma mesma interface estarem disponíveis em um sistema. Verificamos então ser necessário incrementar as abstrações dos modelos de componentes de software para comportar o conceito de versões, de forma que venha a facilitar o trabalho do desenvolvedor e do configurador do sistema. Para isso, estendemos o modelo para incluir os conceitos de *sub-serviço*, *faceta*, *interface de acesso* e *tipo abstrato de dados*.

**Definição 3:** *Sub-serviço* representa uma funcionalidade oferecida por um componente, e pode ser descrito ou especificado por diferentes tipos abstratos de dados.

**Definição 4:** *Tipo abstrato de dados* (TAD) especifica uma versão do sub-serviço, e está associado à uma interface de acesso.[1]

**Definição 5:** *Faceta* denota um conjunto de interfaces programáticas semanticamente relacionadas, que representam a evolução do contrato de acesso de um determinado sub-serviço oferecido por componentes de software.

**Definição 6:** *Interface de acesso* é um elemento do conjunto de interfaces da faceta, sendo a interface pela qual se acessa a faceta.

[1]No texto, referenciamos um elemento do conjunto de TAD como uma versão do sub-serviço, ou simplemente versão.

Resumindo, um serviço pode ser composto por uma coleção de sub-serviços, e um sub-serviço pode estar especificado por diferentes TADs. Por analogia, podemos afirmar que um componente é composto por uma coleção de facetas, e uma faceta pode ser acessada (utilizada) por diferentes interfaces de acesso. A figura 3.1 sintetiza essas definições e conceitos apresentados.

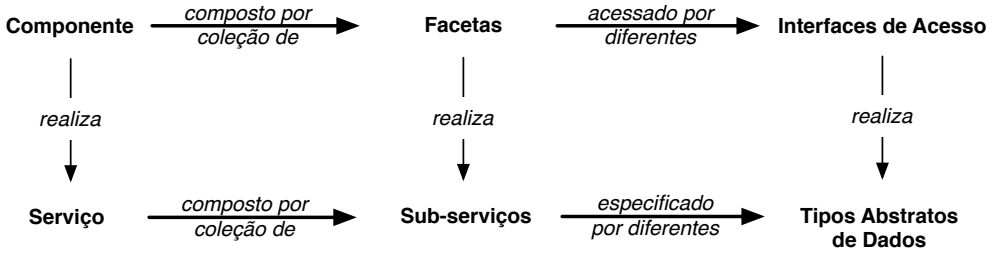


Figura 3.1: Definição dos relacionamentos entre os conceitos do novo modelo SCS.

Para manter a identidade da abstração de componentes e melhor modularizar as informações, entendemos que a faceta precisa ser a representação do sub-serviço e não apenas uma versão específica do sub-serviço. Então, ao invés das facetas estarem amarradas a uma única interface, elas passam a estar atreladas a um nome de sub-serviço, e passam a possuir diferentes interfaces de acesso, para cada versão do sub-serviço que oferecem. A figura 3.2 apresenta essa nova abstração dos componentes.

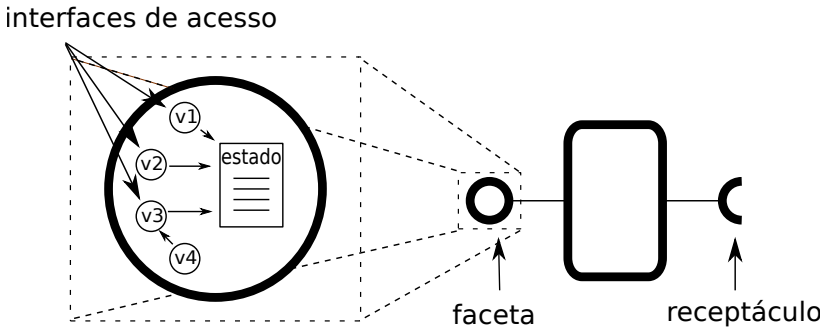


Figura 3.2: Modelo de Componente Estendido

Note que os nomes atribuídos aos sub-serviços ganham um significado muito forte nesse modelo. O nome passa a representar a semântica do sub-serviço, abstraindo um conjunto de interfaces para essa funcionalidade, e, idealmente, esse nome deveria identificar unicamente cada tipo de sub-serviço. No modelo tradicional de componentes, apresentado na seção 2.1, a semântica do sub-serviço e a unicidade de seu nome estavam associadas ao nome da interface do mesmo. Com a possibilidade de existir mais de uma versão

das interfaces, o nome das interfaces pode mudar entre as versões (o que tipicamente ocorre para evitar colisões de nomes no nível da linguagem de programação utilizada na implementação do componente). Então, a amarração da semântica do sub-serviço oferecido pela faceta não pode mais estar atrelado ao nome da interface, e o nome do sub-serviço passa a exercer a função de identificação semântica do mesmo.

Além da faceta poder possuir uma coleção de interfaces de acesso, o modelo precisa prever que as interfaces de acesso podem compartilhar um mesmo estado na mesma faceta, pois o estado da faceta pode ser um fator de influência sobre o comportamento ou a resposta dada por um sub-serviço. Por isso, em alguns casos, para que o comportamento do sub-serviço simule as múltiplas versões suportadas de forma consistente, é necessário que as interfaces de acesso possuam um estado comum compartilhado.

As interfaces de acesso também podem possuir seus próprios estados, que podem ser necessários quando se faz ou não uso do estado compartilhado. Isso é necessário porque pode existir a necessidade de uma interface de acesso possuir informações que são pertinentes apenas para a sua versão, e que é impossível de simular essa versão sem essas informações adicionais. Nesse caso, o estado da interface de acesso deve ser complementar ao estado compartilhado.

Como os componentes são vistos como unidades para a construção de um sistema maior, é possível que os componentes possuam dependências externas, que podem ser resolvidas com uma outra faceta que será conectada ao receptáculo do componente. Acompanhando a evolução das facetas, os receptáculos também deixam de ser amarrados a uma interface específica, e passam a estar atrelados a um sub-serviço. Passa a fazer parte da descrição de um receptáculo a definição de uma lista de versões da interface do sub-serviço associado com as quais é compatível, nomeada de *lista de compatibilidade*.

Com isso, o protocolo de conexão entre facetas e receptáculos deve evoluir também, dada a necessidade de realizar uma comunicação para verificar se a faceta, que está para ser conectada no receptáculo, oferece versões compatíveis com as definidas na lista de compatibilidade do receptáculo.

Um ponto importante que deve ser lembrado é que, para oferecer suporte a múltiplas versões, é necessário manter todas as versões das IDLs das interfaces, para que seja possível gerar *stubs* e *skeletons*, e para fazer o mapeamento dos objetos para as respectivas interfaces. Sendo assim, passa a ser necessário versionar as definições em IDL para se evitar conflitos de nomes. Para contornar esse problema, sugerimos a adição da versão ao módulo onde uma interface IDL é definida. Assim, sempre que houver uma nova versão, será definido um novo escopo de nomes associado ao módulo IDL da nova versão, e

todas as interfaces e tipos correlatos com aquela versão devem estar definidos neste mesmo módulo. O lado negativo dessa abordagem, é que mesmo quando um determinado tipo não é alterado em uma nova versão, ele será redefinido. O Código 3.1 apresenta um exemplo de versionamento de definições em IDL.

Código 3.1: Versões da interface *Hello* com o módulo versionado.

```
1  /* Versão 1.0 */
2  module helloworld {
3    module v1_0 {
4      interface Hello {
5        void sayHello();
6      };
7    };
8  };
9  /* Versão 2.0 */
10 module helloworld {
11   module v2_0 {
12     interface Hello {
13       void sayHello();
14       void sayGoodbye();
15     };
16   };
17 };
18 /* Versão 3.0 */
19 module helloworld {
20   module v3_0 {
21     interface Hello {
22       void say(in string text);
23     };
24   };
25 };
```

## 3.3 Cenários de Uso

Considerando a possibilidade dos sub-serviços possuírem uma ou mais versões, nesta seção iremos apresentar os possíveis cenários de configuração entre componentes cliente e servidor, que podem fazer uso do suporte a múltiplas versões ou não.

Enumerando os possíveis cenários de relacionamento no uso de componentes com suporte a múltiplas versões (MV), podemos ter: (A) cliente e servidor sem MV, (B) cliente sem MV e servidor com MV, (C) cliente e servidor com MV, (D) cliente com MV e servidor sem MV. A figura 3.3 ilustra esses possíveis relacionamentos.

### 3.3.1 Cliente e servidor sem múltiplas versões

O relacionamento A, ilustrado na figura 3.3, é caracterizado pelo fato dos componentes clientes e servidores possuírem apenas uma única versão disponível. Sendo assim, o processo de uso da conexão é o mais direto e simples possível:

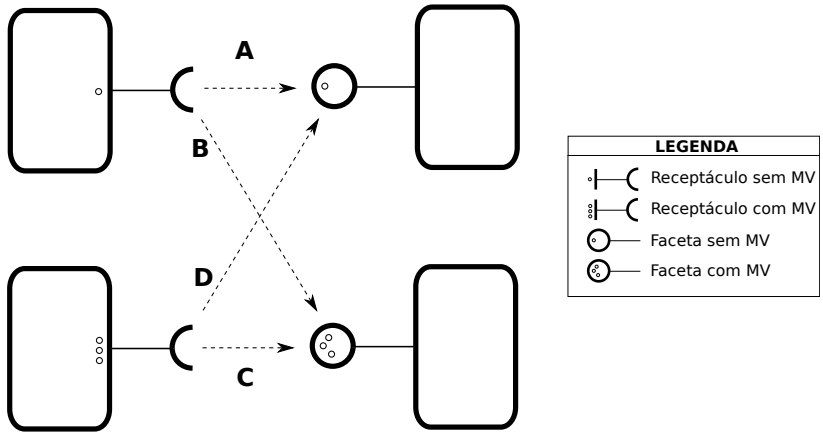


Figura 3.3: Casos possíveis de componentes cliente/servidor com uso ou não de múltiplas versões.

1. O cliente recupera a faceta conectada em seu receptáculo.
2. Como o cliente só conhece uma versão, ele precisa verificar se a faceta do servidor oferece essa mesma versão.
3. Se a versão oferecida é compatível, então recupera a interface de acesso da versão.
4. Faz uso do sub-serviço oferecido pela faceta através da interface de acesso obtida.

### 3.3.2 Cliente sem múltiplas versões e servidor com múltiplas versões

O relacionamento B, ilustrado na figura 3.3, é caracterizado pelo fato dos componentes servidores disponibilizarem múltiplas versões de serviços, e os componentes clientes apenas possuírem conhecimento de uma única versão. Esse é um dos exemplos motivadores do modelo, onde queremos permitir que o servidor evolua sem obrigar a evolução simultânea dos clientes. A idéia de uso da conexão pelos clientes é similar ao relacionamento A:

1. O cliente recupera a faceta conectada em seu receptáculo.
2. Como o cliente só conhece uma versão, ele precisa verificar se a faceta do servidor oferece essa mesma versão, dentre as múltiplas versões oferecidas.
3. Recupera a interface de acesso da versão desejada.
4. Faz uso do sub-serviço oferecido pela faceta através da interface de acesso obtida.

### 3.3.3
### Cliente e servidor com múltiplas versões

O relacionamento C, ilustrado na figura 3.3, é caracterizado pelo fato dos componentes clientes e servidores possuírem múltiplas versões disponíveis. Nesse caso, o servidor disponibiliza para os clientes mais de uma versão do sub-serviço oferecido pela faceta, e os clientes são capazes de se comunicar com os servidores em mais de uma versão.

Para o cliente fazer uso dessas conexões, ele precisa realizar o seguinte procedimento:

1. O cliente recupera a faceta conectada em seu receptáculo.
2. Como o cliente é compatível com mais de uma versão, ele precisa verificar se, dentre as múltiplas versões oferecidas pela faceta do servidor, existem versões disponibilizadas que pertencem à sua lista de compatibilidade.
3. Recupera a interface de acesso de uma das versões compatíveis encontradas na faceta.
4. Faz uso do sub-serviço oferecido pela faceta através da interface de acesso escolhida.

### 3.3.4
### Cliente com múltiplas versões e servidor sem múltiplas versões

Nesse caso D, o servidor só oferece uma única versão, e o cliente é compatível com um conjunto de versões da interface. A idéia de uso da conexão pelos clientes é similar ao relacionamento C:

1. O cliente recupera a faceta conectada em seu receptáculo.
2. Como o cliente é compatível com mais de uma versão, ele precisa verificar se a versão disponibilizada pela faceta do servidor pertence a sua lista de compatibilidade.
3. Recupera a interface de acesso da versão compatível presente na faceta.
4. Faz uso do sub-serviço oferecido pela faceta através da interface de acesso obtida.

## 3.4 SCS-MV

Implementamos neste trabalho uma extensão do sistema de componentes SCS [13] implementado em Lua [20], de forma a representar o modelo de componentes com suporte a múltiplas versões descrito na seção 3.2. Nomeamos essa extensão de SCS-MV.

Para implementar o novo conceito das facetas e o mecanismo de interfaces de acesso da faceta, atendemos todas os requisitos mencionadas na seção 3.1, e adotamos também algumas diretivas de desenvolvimento, que não são essenciais para prover suporte a múltiplas versões, mas auxiliam ou facilitam o desenvolvimento do serviço e de suas versões. Basicamente, as diretivas são decisões de projeto que julgamos serem as mais adequadas, e são inspiradas no Upstart [15, 16]. A seguir, enumeramos as diretivas de desenvolvimento adotadas:

1. Possuir uma única implementação principal para cada serviço oferecido por um componente.

   Cada serviço implementado por um componente pode ser disponibilizado através de várias versões de interface, mas apenas uma delas deve ser considerada a *versão corrente* do serviço. A implementação da versão corrente (implementação principal) deve ignorar a existência de outras versões, com o intuito de simplificar sua implementação. A implementação principal deve ignorar as demais versões suportadas, porque, caso contrário, isso tende a tornar o código mais complexo, ficando mais suscetível a erros e dificultando a manutenção. Dessa maneira, simplificamos a implementação da versão corrente, e permitimos em implantações futuras a remoção das compatibilidades retroativas sem a necessidade de alterar o código da versão corrente. Com isso, diminuímos a chance de introduzir erros na mesma. As implementações das demais versões devem usar a versão corrente da interface como referência, de forma que o estado da implementação principal esteja sempre consistente.

2. Utilizar objetos *simuladores* para oferecer as demais versões.

   As demais versões de uma interface oferecida por um componente devem ser implementadas como simuladores de versão, que não possuem uma implementação completa do serviço, mas recebem as chamadas para a sua versão e as repassam para a implementação principal, realizando as adequações necessárias nos parâmetros e valores de retorno das chamadas.

3. Utilizar um estado compartilhado entre todas as versões de um mesmo serviço.

   O estado pode influenciar o comportamento ou a resposta dada por um serviço. Por isso, para que o comportamento do serviço simule as múltiplas versões suportadas de forma consistente, é interessante que as versões possuam um estado comum compartilhado. Levando em consideração as diretivas 1 e 2, uma implementação natural para atender a esta diretiva é usar a implementação principal como base do estado compartilhado entre todas as versões.

4. Permitir que as versões possuam um estado próprio e complementar ao estado compartilhado.

   Mesmo quando as versões do serviço atuam como simuladores, existem situações onde a versão específica do serviço requer informações que não são comuns às demais versões do serviço. Sendo assim, deve ser possível que as versões possuam um estado próprio e, idealmente, deve ser também complementar ao estado compartilhado.

Adotando essas diretivas, a faceta continua tendo apenas uma implementação em uma versão da interface específica (a versão corrente da faceta), e pode possuir uma coleção de interfaces de acesso para essa implementação. Essa abordagem de ter uma única implementação e várias interfaces de acesso é inspirada no modelo adotado no UPSTART [15, 16], com os objetos de simulação. Dessa maneira, as interfaces de acesso assumem o papel dos objetos de simulação, passando a atuar como *wrappers* [31], realizando a adequação das chamadas de uma versão para outra. Todo o processo de adequação da chamada para compatibilizar com a versão corrente do sub-serviço implementado na faceta deve ser realizada pelas interfaces de acesso. A coleção de interfaces de acesso da faceta inclui a interface de acesso da versão corrente da faceta, sendo que essa interface de acesso não precisa realizar nenhum tipo de adequação nas chamadas recebidas, realizando um repasse direto para a implementação principal.

Com essa extensão do modelo, a descrição da faceta precisou ser modificada na IDL do SCS. A estrutura descritora da faceta (*FacetDescription*), explicitada no Código 3.2, que antes era composta por nome, interface e a referência para o objeto implementador da faceta, passa a ser:

- **Nome:** O nome da faceta.
- **Interface:** A interface da versão corrente da faceta.

– **Versão:** O nome simbólico da versão corrente, que se não foi definido é o mesmo nome da interface da versão corrente.

– **Interfaces de Acesso:** A coleção das descrições das interfaces de acesso da faceta, que são compostas por nome da versão e o nome da interface.

Código 3.2: Estrutura *FacetDescription*.

```
1  module scs {
2    module core {
3      struct FacetDescription {
4        string name;
5        string interface_name;
6        string version;
7        AccessInterfaceDescriptions versions;
8      };
9      typedef sequence<FacetDescription> FacetDescriptions;
10   };
11 };
```

Como a faceta passou a incorporar uma coleção de interfaces de acesso, criamos uma interface que passa a representar a faceta. O nome dessa interface é *IFacet*, e através dela é possível recuperar o nome da faceta, recuperar a referência do componente que possui a faceta, acessar as interfaces de acesso, e obter uma descrição das interfaces de acesso. A definição da interface está explicitada no Código 3.3.

Código 3.3: Interface *IFacet*.

```
1  module scs {
2    module core {
3      interface IFacet {
4        string getName();
5        IFacet getComponent();
6        AccessInterfaceDescriptions getAccessInterfaces();
7        AccessInterface getAccessInterface(in string interface_name)
8          raises (InvalidName);
9        AccessInterface getAccessInterfaceByVersion(in string version)
10         raises (InvalidVersion);
11     };
12   };
13 };
```

Adicionamos as estruturas *AccessInterface* e *AccessInterfaceDescription* para auxiliar na representação, descrição e manuseio das interfaces de acessos. A distinção entre as duas estruturas é que a *AccessInterfaceDescription* apenas contém as informações que descrevem a interface de acesso, que são o nome da interface da versão implementada e o nome da versão, enquanto que a *AccessInterface* também possui a referência para o objeto interface de acesso. O Código 3.4 apresenta a definição das duas estruturas. O campo nome de versão serve para permitir a definição de um rótulo (*tag*) para a versão.

Através da interface *IReceptacles* é possível realizar e desfazer conexões, recuperar as conexões ativas, e obter a lista de descritores das interfaces de

Código 3.4: Definição das estruturas *AccessInterface* e *AccessInterfaceDescription*.

```
module scs {
  module core {
    struct AccessInterface {
      string interface_name;
      string version;
      Object objref;
    };
    typedef sequence<AccessInterface> AccessInterfaces;

    struct AccessInterfaceDescription {
      string interface_name;
      string version;
    };
    typedef sequence<AccessInterfaceDescription> AccessInterfaceDescriptions;
  };
};
```

acesso compatíveis que foram encontradas durante a conexão. O código 3.5 apresenta a definição da interface *IReceptacles*.

Código 3.5: Interface *IReceptacles*.

```
module scs {
  module core {
    interface IReceptacles {
      ConnectionId connect (in string receptacle, in IFacet obj)
        raises (InvalidName, InvalidConnection, AlreadyConnected,
                ExceededConnectionLimit);
      void disconnect (in ConnectionId id)
        raises (InvalidConnection, NoConnection);
      ConnectionDescriptions getConnections (in string receptacle)
        raises (InvalidName);
      AccessInterfaceDescriptions getConnectionsVersions (in string receptacle)
        raises (InvalidName);
    };
  };
};
```

A estrutura descritora de receptáculos (*ReceptacleDescription*) também sofreu algumas modificações com essa extensão do modelo. Ela deixa de ser amarrada a uma interface, e passa a possuir uma lista de versões de interface compatíveis. O Código 3.6 apresenta a definição desta estrutura. Durante a construção do componente, é possível especificar nos descritores para construção de receptáculos as versões com as quais o receptáculo é compatível.

A implementação do protocolo para a realização das conexões sempre faz uma verificação nas versões disponibilizadas pelas facetas com as versões aceitas pelos receptáculos. Caso encontre alguma versão compatível, a conexão é realizada. Caso não encontre nenhuma versão compatível, lança a exceção de *InvalidConnection*. Se nenhuma versão compatível for especificada no receptáculo, o protocolo irá aceitar qualquer versão na validação da conexão. Não especificar a lista de compatibilidade dos receptáculos é o mesmo que

Código 3.6: Definição da estrutura *ReceptacleDescription*.

```
module scs {
  module core {
    struct ReceptacleDescription {
      string name;
      boolean is_multiplex;
      ConnectionDescriptions connections;
      sequence<string> versionList;
    };
    typedef sequence<ReceptacleDescription> ReceptacleDescriptions;
  };
};
```

dizer que o receptáculo não possui restrição de compatibilidade, e que aceita qualquer interface.

### 3.4.1 Suporte a Implementação de Componente

Para construir componentes, é possível fazer uso de um método auxiliar que fabrica componentes, denominado *newComponent*. Este método espera receber uma lista de descritores de facetas, uma lista de descritores de receptáculos, a estrutura identificadora do componente (*ComponentId*), e o ORB que será utilizado para instanciar os objetos servidores, sendo que este último parâmetro é opcional.

Os descritores são responsáveis por conter todas as informações necessárias para a criação do componente. A implementação do novo modelo mantém a característica da implementação original do SCS, onde não é obrigatório fornecer os descritores das facetas básicas do modelo: *IComponent*, *IMetaInterface*, e *IReceptacles*.

Os descritores de facetas foram alterados para incluir o conceito de versão e interfaces de acesso. Os seguintes atributos foram adicionados ao descritor:

- **key:** a chave utilizada para cadastrar o objeto servidor (*servant*) IFacet desta faceta.
- **version:** nome da versão corrente. Se não for definida, utiliza o nome da interface da versão corrente como o nome da versão.
- **versions:** lista de interfaces de acesso da faceta, indexada por nome de versão. Cada entrada da lista é uma tabela com os seguintes atributos:
  - **version:** nome da versão.
  - **interface_name:** nome da interface IDL da interface de acesso.
  - **wrapper_class:** classe que implementa a interface especificada.
  - **wrapper_key:** chave a ser utilizada no cadastramento do *servant* da interface de acesso.

Os descritores de receptáculos também foram alterados para permitir a definição das versões compatíveis com o receptáculo. O seguinte atributo foi adicionado:

– **compatible_versions:** lista de versões com as quais o receptáculo é compatível, podendo não ser definida. Caso a lista não seja definida, o receptáculo irá aceitar a conexão de facetas com qualquer interface, como explicado na seção 3.4.

Todas as implementações de interfaces de acesso possuem uma referência para a implementação principal (implementação da versão corrente) do subserviço da faceta à qual pertencem através do campo *_facet_ref*. A implementação das interfaces de acesso já é feita levando isso em consideração, e no momento da instanciação do componente, que também instancia as facetas e as implementações das interfaces de acesso, o campo *_facet_ref* das interfaces de acesso é atualizado para guardar a referência da implementação da faceta.

Uma divergência que existe no trabalho proposto e o UPSTART está relacionada com o encadeamento das interfaces de acesso. Os *SOs* do UPSTART são encadeados, de tal maneira que durante a atualização da versão corrente do nó só é necessário atualizar a implementação da versão anterior e a corrente. A nossa implementação não amarra o modelo a essa abordagem. É possível implementar as interfaces de acesso sem que elas estejam encadeados, e, nesse caso, todos as interfaces de acesso precisariam ser atualizadas quando a implementação principal de uma faceta fosse atualizada.

O ponto negativo do não encadeamento das interfaces de acesso é a maior quantidade de trabalho necessário para atualizar uma faceta, e manter a compatibilidade com muitas outras versões. O lado positivo, é que essa abordagem não estimula a manutenção de compatibilidade com muitos números de versão, o que induz um processo de evolução contínua de todo o sistema. Ou seja, clientes não poderiam ficar muito tempo congelados numa versão, pois eventualmente o sistema irá deixar de prover aquela versão. Outro aspecto positivo em uma faceta não manter compatibilidade com muitas versões é que esperamos uma tendência de que quanto maior for a distância para a versão corrente da faceta, maior será a complexidade do código de adequação de chamadas entre as versões.

## 3.5 Considerações Finais

Apresentamos neste capítulo os pontos fundamentais para prover suporte a múltiplas versões, definimos um modelo de componente com suporte

a múltiplas versões, e implementamos esse modelo estendendo o SCS. Os requisitos e as diretivas para suporte de múltiplas versões foram os pontos que direcionaram o desenvolvimento do modelo proposto, e procuramos atender a todos eles.

A primeira modificação realizada sobre o modelo tradicional de componentes foi de expressar na abstração do componente a relação de dependência existente entre as versões e o sub-serviço oferecido com a faceta. Para isso definimos que uma faceta pode possuir uma coleção de interfaces de acesso e representa o sub-serviço, e as interfaces de acesso representam as diferentes versões suportadas pela faceta. Com isso, nós atendemos ao requisito 1. Acreditamos que, fazendo uso dessa nova modelagem, resolvemos os problemas de perda da identidade da abstração do componente e da coesão de suas facetas, comentados anteriormente na seção 2.4. Acreditamos que, utilizando-se dessa abordagem de representação das facetas e de suas versões, nós nos aproximamos mais em satisfazer os conceitos de modularidade de continuidade, compreensibilidade e ocultação da informação [14].

Definimos que o modelo pode prover um estado compartilhando entre as versões oferecidas, atendendo ao requisito 2. Com isso o modelo oferece uma abordagem para resolver o problema de sincronia de estado que acontece quando tenta-se fazer uso de múltiplas versões com um modelo tradicional de componentes. Além disso, como as múltiplas versões passam a fazer parte de uma mesma faceta, o desenvolvedor do componente não precisa mais criar várias facetas para prover suporte às múltiplas versões, fugindo da problema de gerenciar os diferentes nomes das facetas para as versões oferecidas.

Os receptáculos passaram a poder possuir uma lista de versões com as quais são compatíveis, para fazer uso do conceito das múltiplas versões e facilitar o trabalho do configurador do sistema, pois este não precisaria realizar conexões individuais para cada versão suportada, atendendo assim aos requisitos 3 e 4.

Como definimos que as versões são representadas por diferentes interfaces de acesso, ao cadastrá-las como objetos servidores, para aceitarem chamadas remotas pelo *middleware* CORBA, definimos uma maneira de direcionar as chamadas nas diferentes versões para seus respectivos objetos. Com isso, atendemos ao requisito 5.

O requisito 6 existe, pois é necessário resolver eventuais conflitos de nomes no nível da linguagem de programação usada na implementação do componente. Então, é necessário definir uma maneira de diferenciar os nomes das diferentes versões de uma interface. Para isso, propomos a inclusão do número da versão no módulo onde a interface IDL é definida.

Ao implementar o SCS-MV, acabamos por nos deparar com decisões de projeto interessantes. As diretivas 1 e 2 sugerem que a faceta possua apenas uma implementação principal e que as interfaces de acesso funcionem como simuladores de versões. Dessa forma, a quantidade de código que o desenvolvedor dos componentes precisa trabalhar e a complexidade para garantir um funcionamento consistente entre as diferentes versões suportadas são menores. Já para prover um funcionamento consistente entre as versões, entendemos que a melhor abordagem é fazer uso de um estado compartilhado entre as mesmas, representado pela implementação principal da faceta, atendendo à diretiva 3.

A diretiva 4 é necessária, pois nem sempre é possível simular uma versão fazendo uso apenas do estado compartilhado. Logo, deve ser possível que as interfaces de acesso possuam um estado próprio, que seja complementar ao estado compartilhado.

O SCS-MV foi desenvolvido pensando nas diretivas de projeto, e existem alguns pontos da implementação que poderiam ser revisados. A definição de que a faceta possui uma única implementação (diretiva 1) é um exemplo disso. A implementação acaba externalizando essa decisão ao definir que no descritor da faceta (*FacetDescription*, definida no Código 3.2) existem os atributos nome da versão e da interface IDL corrente. O SCS-MV não deveria externalizar essas informações pois o desenvolvedor do componente poderia querer adotar outro tipo de estratégia.

Conforme discutido na seção 3.2, os nomes dados para identificar os sub-serviços possuem um significado forte associado a eles: a funcionalidade mais abstrata oferecida pelo sub-serviço, independente de versão. A implementação do SCS-MV não lida bem com essa importância do nome. Não existem mecanismos para tratar esses nomes como únicos no sistema, e com o significado único também, que eventualmente poderiam valer até entre diferentes tecnologias de middleware.

O modelo proposto tornou o uso do SCS mais complexo, pois aumenta o número de indireções. Deveria existir uma maneira de não obrigar essas indireções se a faceta não provesse suporte a múltiplas versões, ou se possuísse uma versão padrão. Por exemplo, o Ice permite isso através da definição da faceta padrão do objeto Ice.