PUC-Rio - Certificação Digital Nº 1121796/CA

Mauricio Arieira Rosas

Estudo da aplicação de componentes hierárquicos em um Sistema de Captura e Acesso

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
Agosto de 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Mauricio Arieira Rosas

Estudo da aplicação de componentes hierárquicos em um Sistema de Captura e Acesso

PUC-Rio - Certificação Digital Nº 1121796/CA

Dissertação de Mestrado

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Prof. Noemi Rodriguez

Rio de Janeiro
Agosto de 2014

Pontifícia Universidade Católica do Rio de Janeiro

Mauricio Arieira Rosas

Estudo da aplicação de componentes hierárquicos em um Sistema de Captura e Acesso

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Noemi Rodriguez
Orientador
Departamento de Informática — PUC–Rio

Prof. Alexandre Sztajnberg
Departamento de Informática e Ciências da Computação — UERJ

Prof. Markus Endler
Departamento de Informática — PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 12 de Agosto de 2014

PUC-Rio - Certificação Digital Nº 1121796/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Mauricio Arieira Rosas

Graduou-se em Bacharelado em Engenharia da Computação pela PUC-Rio. Foi pesquisador do laboratório Tecnologia em Computação Gráfica (Tecgraf) da PUC-Rio de Janeiro 2009 a Outubro de 2011, onde trabalhava no desenvolvimento do *middleware* Openbus e SCS que apoia diversos projetos em parceria com a Petrobras S/A.

PUC-Rio - Certificação Digital Nº 1121796/CA

Ficha Catalográfica

Rosas, Mauricio Arieira

Estudo da aplicação de componentes hierárquicos em um sistema de captura e acesso / Mauricio Arieira Rosas; orientador: Noemi Rodriguez. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2014.

v., 66 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1; Informática – Tese. 2; Modelos de Componentes de Software; Programação Orientada a Componentes; Componentes Compostos; Middleware; Arquiteturas de software. I. Rodriguez, Noemi. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

PUC-Rio - Certificação Digital Nº 1121796/CA

Aos meus familiares e amigos.

Resumo

Rosas, Mauricio Arieira; Rodriguez, Noemi. **Estudo da aplicação de componentes hierárquicos em um Sistema de Captura e Acesso**. Rio de Janeiro, 2014. 66p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O objetivo deste trabalho é avaliar um sistema de componentes de software que oferece em seu modelo uma abstração de componentes compostos. O sistema de componentes estudado é o SCS, que define um conjunto de regras de aninhamento, encapsulamento e compartilhamento para reger o comportamento de seus componentes compostos. O foco deste estudo está na avaliação da eficácia dessas regras de composição para auxiliar o desenvolvedor de aplicações. Para realizar esta avaliação, adaptamos o sistema de Captura e Acesso CAS, construído com o middleware SCS, para utilizar componentes compostos, e criamos um cenário para analisar o modelo e a implementação do middleware.

PUC-Rio - Certificação Digital Nº 1121796/CA

Palavras–chave

Modelos de Componentes de Software; Programação Orientada a Componentes; Componentes Compostos; Middleware; Arquiteturas de software.

Abstract

Rosas, Mauricio Arieira; Rodriguez, Noemi (Advisor). **A Study of hierarchical component in a Capture and Access System**. Rio de Janeiro, 2014. 66p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The aim of this work is to evaluate a software component system that provides in its model an abstraction of composite components. We chose for this study SCS as the components system, which defines a set of rules for nesting, encapsulation and sharing to manage the behavior of its composite components. The main focus of this study is to evaluate the effectiveness of these composition rules to assist application developers. To conduct this evaluation, we adapted the Capture and Access system CAS, developed via the middleware SCS in order to employ the composite components, and created a scenario to analyze the model and implementation of the middleware.

PUC-Rio - Certificação Digital Nº 1121796/CA

Keywords

Software Component Models; Component-oriented programming; Composite Component; Middleware; Software Architecture.

Sumário

1 Introdução **10**
1.1 Estrutura do Documento 12

2 Tecnologias básicas **13**
2.1 SCS 13
2.2 SCS Composite 14
2.3 CAS 15
2.4 Infraestrutura do CAS 17

3 SCS Composite Revisado **18**
3.1 Regras 18
3.1.1 Componente 18
3.1.2 Mecanismo de *Binding* Horizontal 19
3.1.3 Componente composto 20
3.1.4 Hierarquia de Componentes 20
3.1.5 Compartilhamento de subcomponentes 21
3.1.6 Externalização de Facetas e Receptáculos 22
3.1.7 Conexão 23
3.2 Estudo preliminar 24
3.3 Implementação para o trabalho 25
3.3.1 Implementação das regras 27
3.3.2 Proxy no Receptáculo 33
3.3.3 Proxy nas Facetas 35
3.3.4 Ferramentas de inspeção de componentes 36

4 Análise **38**
4.1 Implementação do CAS com SCS-Composite 38
4.2 Cenários 42
4.3 Análise do Cenário 44
4.3.1 Bind de Faceta 46
4.3.2 Bind de Receptáculo 47
4.3.3 Compartilhamento de Componentes 49
4.4 Reconfiguração dos componentes 50

5 Conclusão **51**

6 Referências Bibliográficas **53**

A Interface IDL do SCS-Composite **56**

B Exemplo de uso do SCS-Composite **63**

C Exemplo de testes de Prolog **65**

PUC-Rio - Certificação Digital Nº 1121796/CA

Lista de figuras

1.1 Representação de um componente composto. 10

2.1 Um Componente SCS com as três facetas básicas. 13
2.2 Exemplo de *binding vertical* de facetas de dois conectores. 15

3.1 Exemplo de compartilhamento do subcomponente SCB1 em dois componentes compostos. 21
3.2 Exemplo de indireção de um receptáculo. 23
3.3 (a) Medeiros; (b) um componente que representa todas as conexões; (c) Um componente para cada conexão. 34
3.4 Diagrama de sequência que descreve a criação do Proxy na operação *connect()* da Faceta *IReceptacles*. 35
3.5 Cenário criado para verificar a necessidade da regra 10. 35
3.6 Diagrama de sequência que descreve a criação do Proxy na operação *bindFacet()* da Faceta *IContentController*. 36

4.1 Representação gráfica do cenário descrito. 43
4.2 Representação dos componentes do cenário descrito. 44
4.3 Representação do cenário utilizando a ferramenta de inspeção de componentes desenvolvida nesse trabalho. 45
4.4 Subconjunto do cenário descrito na seção 4.2. 47
4.5 Subconjunto do cenário descrito na seção 4.2. 48
4.6 (a) Opção com compartilhamento de componentes (b) Opções com conectores e *bind* de receptáculo. 49

2.2 Exemplo de *binding vertical* de facetas de dois conectores (reprodução da imagem do Capítulo 2). 63

PUC-Rio - Certificação Digital Nº 1121796/CA

PUC-Rio - Certificação Digital Nº 1121796/CA

"What we have done for ourselves alone, dies with us; what we have done for others and the world, remains and is immortal."

Albert Pike

1
Introdução

A área de Componentes de Software é bastante estudada desde a década de 60. Desde então, foram feitos muitos estudos nesta área, tanto sobre a fase de implantação dos componentes (Flissi et al., 2008), (Flissi and Merle, 2006) e (Hall et al., 1997), quanto nas áreas de adaptação dinâmica (Taylor et al., 2009), (Oreizy et al., 1999) e (Cerqueira, 2000) e de componentes compostos (Lau et al., 2007), (Elizondo et al.) e (Lau et al., 2006). Crnkovic et al. (2010) classificou diferentes modelos de componentes, destacando suas principais similaridades e diferenças, e concluiu que embora não exista uma definição padrão bem aceita, a maioria dos modelos possuem características bem estabelecidas em diversos *frameworks*.

O conceito de componente composto permite que um componente encapsule outros componentes e seja responsável por disponibilizar interfaces de configuração e introspecção, sem comprometer as funcionalidades do componente primitivo - a criação de facetas e receptáculos. A utilização de componentes compostos permite abstrair estruturas complexas, onde existem vários componentes conectados, além de promover o reuso dessas composições (Lau et al., 2007). Na figura 1.1, ilustramos um componente composto com três subcomponentes conectados.

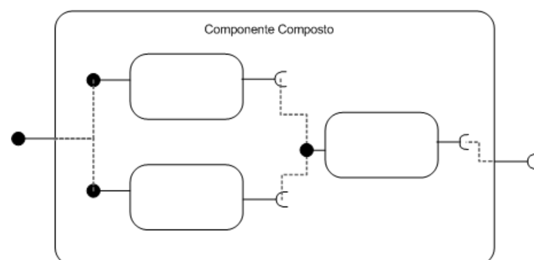

Figura 1.1: Representação de um componente composto.

O conceito de componente compostos é implementado por diversos modelos, como OpenCOM (Coulson et al., 2008), Fractal (Bruneton et al., 2006), Koala (van Ommering et al., 2000) e SaveCCM (Hansson et al., 2004). Porém, cada um possui suas características específicas no que diz respeito ao mapeamento de serviços dos subcomponentes nos componentes compostos;

PUC-Rio - Certificação Digital Nº 1121796/CA

a necessidade, construção e uso de conectores; e ao compartilhamento de componentes.

Em nosso trabalho iremos analisar um modelo de componentes compostos que foi criado com base em um estudo de diversos modelos amplamente conhecidos, com destaque para OpenCOM e Fractal. O estudo desenvolvido por Medeiros (2012) produziu uma especificação formal e uma implementação inicial desta especificação.

Em nossa análise iremos revisitar o conjunto de regras descritas na especificação formal, testando-as e validando-as, a fim de identificar regras muito restritivas e que possam comprometer a usabilidade do *middleware* que implementa o modelo. Nesses casos, nós iremos destacar as limitações, refiná-las ou descrever uma forma de contornar a restrição encontrada. Para realizar esta análise iremos criar o suporte a componentes compostos no *middleware* SCS (*Sistema de Componente de Software*) (Augusto et al., 2009), uma infraestrutura leve que disponibiliza o conceito de componentes de software distribuídos para a arquitetura CORBA (OMG, 2006), e que permite a reconfiguração e adaptação do sistema em tempo de execução. O SCS já foi desenvolvido em diversas linguagens de programação (Lua, C, Java e C++) e é suportado em diversas plataformas. Também iremos aplicar os conceitos de componentes compostos em um sistema de Captura e Acesso.

A área de Captura e Acesso, também conhecida pela sigla C&A, se encaixa na área de Computação Ubíqua e tem como propósito capturar eventos que estão acontecendo em um espaço específico, processá-los a fim de gerar um documento que possua todas as informações consideradas relevantes, e disponibiliza-las para o usuário final.

Já existem alguns sistemas de C&A, principalmente para ambientes educacionais (Abowd et al., 1998) e profissionais (Lamming and Flynn, 1994). Neste trabalho vamos utilizar o Sistema de Captura e Acesso CAS (*Capture & Access Service*) (Portella, 2008). O CAS foi desenvolvido primeiramente para dar suporte a salas de aula e palestras. Porém, o CAS pode ser facilmente estendido para outros ambientes por seguir um paradigma orientado a componentes, que permite com que cada fase do processo e seus tratadores sejam facilmente criados, atualizados ou removidos. O CAS utiliza o *middleware* SCS, que habilita o sistema a interagir com um grande número de dispositivos que variam de servidores a sensores, com naturalidade.

O CAS trabalha com o conceito de espaço, que representa o local onde o evento será capturado. O espaço é representado por um componente que tem a responsabilidade de configurar e controlar todos os dispositivos de gravação. Como o SCS atual não oferece suporte ao conceito de componentes compostos,

PUC-Rio - Certificação Digital Nº 1121796/CA

a implementação do CAS foi obrigada a simular tal funcionalidade.

1.1 Estrutura do Documento

Este trabalho foi organizado em cinco capítulos. O Capítulo 2 descreve as tecnologias básicas utilizadas no trabalho: o SCS, modelo de componente de software onde faremos nossa implementação; e o CAS, sistema de Captura e Acesso que utilizaremos para avaliar o SCS.

No Capítulo 3 apresentaremos o nosso estudo no modelo SCS. Inicialmente destacaremos os predicados e regras do modelo escolhido. Depois faremos um estudo preliminar das regras para, em seguida, apresentar a implementação proposta.

O Capítulo 4 é dedicado à análise da implementação do SCS-Composite. Neste capítulo veremos a adaptação do CAS para a utilização do SCS-Composite e apresentaremos um cenário de teste complexo que exercite todas as funcionalidades do *middleware*.

Por fim, o Capítulo 5 apresentará as conclusões do projeto, as considerações finais e sugerirá alguns trabalhos futuros.

PUC-Rio - Certificação Digital Nº 1121796/CA

2
Tecnologias básicas

2.1
SCS

O SCS (*Sistema de Componente de Software*) Augusto (2008) é uma infraestrutura leve que disponibiliza o conceito de componentes de software distribuídos para a arquitetura CORBA. O SCS já possui implementações em diversas linguagens de programação e oferece suporte para diversas plataformas. O SCS permite que componentes provejam interfaces de acesso, representadas por facetas, e explicitem dependências, representadas por receptáculos. A versão atual do SCS define a criação de três facetas básicas, que são responsáveis pelo gerenciamento do componente como um todo. A Figura 2.1 representa um componente SCS com suas três facetas básicas, de acordo com a notação de UML.

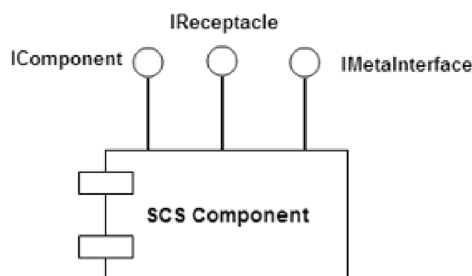

Figura 2.1: Um Componente SCS com as três facetas básicas.

A faceta básica **IComponent** provê acesso a outras facetas utilizando o nome da faceta ou a interface da faceta como informação para a pesquisa. A mesma faceta também provê operações para controlar o ciclo de vida do componente – atualmente acessados pelas operações *startup* e *shutdown*. Tais funcionalidades definem a faceta *IComponent* como a faceta principal do componente.

A faceta básica **IReceptacles** gerencia todas as conexões dos receptáculos do componente. Cada receptáculo é representado por um identificador, uma interface e uma variável binária que define se o receptáculo suporta

PUC-Rio - Certificação Digital Nº 1121796/CA

múltiplas facetas conectadas simultaneamente. A faceta oferece uma interface simples com uma operação para conectar uma faceta a um receptáculo e outra operação para desconectar uma faceta. É de responsabilidade da faceta *IReceptacles* verificar se a nova conexão é compatível com a interface esperada pelo receptáculo do componente.

A última faceta básica, chamada de **IMetaInterface**, é responsável pela introspecção do componente. A interface fornece operações responsáveis por obter descrições de todas as facetas, *FacetDescription*, e receptáculos, *ReceptacleDescription*.

2.2 SCS Composite

O SCS Composite foi implementado no modelo SCS conforme sua especificação formal (Medeiros, 2012), que foi baseada nos modelos Open COM Box (1998), Fractal (Bruneton et al., 2006) e CORBA Component Model - CCM OMG (2006). Para tal foi necessário adicionar ao SCS o conceito de *binding* vertical (ou conexão), a faceta *ISuperComponent*, obrigatória para todos os componentes, e a faceta *IContentController*, obrigatória para todos os componentes compostos.

É classificado como *binding* qualquer ligação entre dois componentes. No SCS, os *bindings* entre facetas e receptáculos são feitos diretamente, sem a necessidade de uma entidade para representá-los. Por outro lado, a conexão entre um componente composto e um ou mais subcomponentes, também chamado de *binding* vertical, necessitam que seja criada uma entidade *conector*, que será representada no modelo como um componente. O conector permite que o controle da aplicação seja desacoplado do componente composto. Desta forma é possível criar um conector específico para cada *binding* e manter a implementação do componente composto sem alterações. O componente *connector* permite criar uma lógica específica para agregar um conjunto de componentes que serão exportados como facetas e receptáculos. O conector também permite explorar diversas formas de abstração das facetas dos subcomponentes, isto é, nós podemos criar uma interface mais alto nível que esconda a complexidade dos subcomponentes.

A faceta obrigatória **ISuperComponent** permite o acesso a todos os componentes compostos que encapsulam o componente em questão. É utilizada em testes de validação de conexões realizados na etapa de *binding* e para restringir o acesso às conexões dos receptáculos externos ao componente pai, garantindo que só haja interação entre componentes que pertençam ao mesmo componente composto – requisito da especificação formal que veremos no

PUC-Rio - Certificação Digital Nº 1121796/CA

capítulo 3.

A faceta **IContentController** fornece mecanismos para configuração do componente composto, permitindo a adição, remoção e introspecção de sub-componentes, como também fornece operações para adicionar e remover conexões (*bindings* verticais). A oferta da faceta *IContentController* caracteriza o componente como um componente composto.

A abstração de componente composto introduzida no modelo do SCS altera também o comportamento da faceta básica **IReceptaIces**, com o intuito de adicionar algumas regras de verificação de conexões definidas como requisitos obrigatórios.

Como exemplo de uso do SCS-Composite, descrevemos um cenário de vigilância de uma casa (representado por um componente composto) que possui cinco câmeras (representadas como cinco componentes), duas câmeras instaladas na parte interna da casa e outras três na parte externa. Este sistema de vigilância proverá interfaces de controle para os dois ambientes separadamente (representadas como facetas do componente composto). Dois conectores serão adicionados para agrupar as câmeras, onde o primeiro agrupará apenas as câmeras externas da casa, enquanto o segundo agrupará as câmeras internas. A figura 2.2 contém uma representação deste cenário e o Apêndice B mostra como seria codificado esse exemplo.

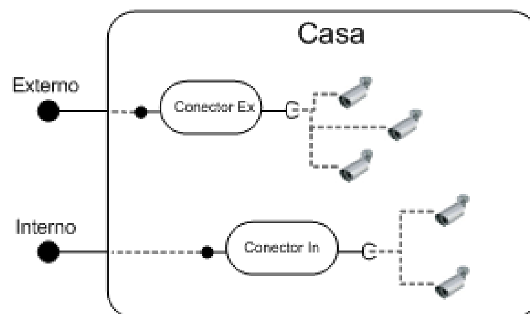

Figura 2.2: Exemplo de *binding vertical* de facetas de dois conectores.

2.3 CAS

O Projeto CAS (*Capture & Access Service*) provê apoio a aplicações de captura e acesso em um ambiente instrumentados. A infraestrutura auxilia diversas atividades como: captura de diferentes mídias através de dispositivos heterogêneos; transferência, armazenamento e pós-processamento de mídias capturadas; geração de documentos multimídia para acesso à informação de forma estruturada. O objetivo do projeto é prover uma infraestrutura que possibilite criar diversas formas de captura em um ambiente colaborativo,

PUC-Rio - Certificação Digital Nº 1121796/CA

onde são utilizados câmeras pré-configuradas e dispositivos móveis de captura de posse dos próprios participantes. Os dispositivos podem ser agrupados logicamente de forma a refletir sua distribuição espacial ou de forma a abstrair tal distribuição, como no caso de videoconferências (Brandão et al., 2013).

O objetivo inicial do projeto CAS (Portella, 2008) foi prover um Sistema de Captura e Acesso para aulas, palestras e reuniões, ou seja, um evento contendo uma apresentação de slides, áudio e vídeo, mas permite ser facilmente estendido para outros ambientes por seguir um paradigma orientado a componentes. O projeto segue a mesma estrutura sugerida por Abowd Abowd et al. (1998) com as fases de pré-produção, captura, pós-produção e acesso. Cada fase possui um ou mais componentes responsáveis.

A fase pré-produção, também conhecida como fase de configuração, é responsável pela preparação da sessão de captura. O administrador do sistema normalmente é o responsável por essa fase, que tem o propósito de verificar se os dispositivos de gravação estão devidamente configurados e prontos para iniciar a gravação. O CAS provê uma interface web, chamada CASWeb, que permite ao administrador agendar ou iniciar a gravação de um evento.

O CAS possui componentes especializados de captura dos dispositivos de gravação, também chamado de SpeedCar (*Specialized Capture Driver*). Cada dispositivo possui seu próprio SpeedCar, logo um computador que esteja com a apresentação de slide e um gravador de voz terá duas instâncias de SpeedCar diferentes, uma para cada mídia. A fase de captura é iniciada quando os SpeedCars iniciam a gravação e é finalizada quando os mesmos enviam os dados capturados para o serviço de armazenamento de dados.

Na fase de pós-produção, as mídias são primeiramente sincronizadas e convertidas para um formato que permita maior compactação e compatibilidade com os diferentes players. Em seguida a etapa de transformação é iniciada com o objetivo de adicionar ou remover informações às mídias. As transformações permitem a criação de diferentes formas de navegação no documento. Como exemplo, vale destacar os transformadores de detecção de contexto nos slides da apresentação, que têm a capacidade de identificar mudanças de assuntos e marca-las na linha do tempo (*timeline*) do documento multimídia (Portella, 2008). O transformador de remoção de transições curtas também é muito importante para remover transições de slides que ocorrem quando o apresentador está navegando entre os slides.

Como resultado da fase de pós-produção é gerado um documento multimídia, no formato NCL (Lab/PUC-Rio), com diferentes formas de navegação. Todo o conteúdo é então compactado e armazenado em um repositório de dados. O CAS também se encarrega de publicar o documento em uma página

PUC-Rio - Certificação Digital Nº 1121796/CA

web automaticamente, com suas meta-informações: título, sinopse, notas de slide, nome do evento, nome do apresentador, data e hora de captura, idioma, tipo de evento e mídias capturadas.

2.4 Infraestrutura do CAS

A infraestrutura básica do projeto CAS é composta pelos componentes *Space*, *Space Configurator* e *Control Painel*, os serviços de pós-processamento e repositório de dados. Também fazem parte da infraestrutura do CAS os componentes *SpeedCars* (*Specialized Capture Driver*) que são responsáveis por capturar diferentes tipos de mídia. Cada *SpeedCar* possui três facetas obrigatórias: *IRecord*, responsável pela captura de mídias; *IConfigurable*, responsável pela configuração dos dispositivos; *IDataTransfer*, responsável por enviar as informações gravadas ao repositório de dados central.

O componente *Space Configurator* tem como objetivo configurar e conectar os *SpeedCars* no componente *Space*. Também é de responsabilidade do *Space Configurator* verificar se todas as configurações pré-definidas estão sendo atendidas antes que a captura seja iniciada.

O componente *Space* agrega todos os *SpeedCars* que farão parte do ambiente de captura. Também é seu papel coordenar o início e suspensão de uma gravação, bem como a transmissão das mídias gravadas para o repositório de dados. Como a versão atual do CAS (versão 2.0 criada em 12/06/2012) utiliza o SCS 1.2.1, que possui o conceito de componentes compostos, o componente *Space* simula um componente composto, criando uma faceta para cada faceta obrigatória do *SpeedCar*. Estas facetas apenas repassam as chamadas para os *SpeedCars* conectados. O componente *Space* também possui a faceta *ISpace* que permite obter e definir algumas características específicas do espaço.

O mecanismo de componente composto no projeto CAS irá remover a necessidade de simulação de componente composto do componente *Space*, que não permite uma navegação padrão entre seus subcomponentes (*SpeedCars*), como também não permite hierarquização de diversas salas, e.g. uma sala está contida em um espaço, por sua vez contido em um espaço maior, e assim por diante. Tal limitação dificulta a utilização do sistema em eventos com múltiplos espaços físicos, como uma teleconferência ou uma palestra que será assistida em múltiplos ambientes.

PUC-Rio - Certificação Digital Nº 1121796/CA

3
SCS Composite Revisado

Este capítulo apresenta a revisão do *middleware* SCS-Composite, que utiliza como base a especificação formal de Medeiros (2012). A seção 3.1 detalha as regras e predicados da especificação formal do SCS-Composite, que são descritas na linguagem declarativa Prolog. O trabalho de Medeiros também desenvolveu uma implementação e uma interface IDL do modelo. Em nosso estudo verificamos a necessidade de algumas evoluções na interface (seção 3.2) e realizamos uma nova implementação do modelo (seção 3.3.1). Nós também analisamos e revisamos as regras propostas no trabalho de Medeiros, identificamos alguns pontos restritivos no modelo e verificamos se os predicados e regras são suficientes ou se são muito restritivos – seção 3.2. Por fim, a seção 3.3, apresenta como nós implementamos todas as regras descritas na API do SCS-Composite.

3.1
Regras

O modelo SCS-Composite utilizou um conjunto de predicados e regras na linguagem Prolog para especificar a semântica de componentes compostos no SCS. Para que fosse possível testar as regras, o modelo também especificou as regras do SCS original. As seções a seguir irão descrever todos os predicados e regras do SCS (seções 3.1.1 e 3.1.2) e do SCS-Composite (seções 3.1.3 a 3.1.7).

3.1.1
Componente

Um componente primitivo é representado no SCS por um conjunto de facetas **F** e receptáculos **R**.

$$C = \langle F, R \rangle \, , com \; F = \{f_1, f_2, \ldots, f_n\} \; e \; R = \{r_1, r_2, \ldots \, r_k\}$$

Os predicados a seguir representam facetas e receptáculos:

Predicado 1 *facet(identifier, type, component)*

Predicado 2 *receptacle(identifier, type, cardinality, component)*

PUC-Rio - Certificação Digital Nº 1121796/CA

Onde **identifier** representa um identificador único; **type** representa uma interface; **component** o componente que possui a faceta ou o receptáculo; e **cardinality** um valor que descreve se o receptáculo aceita uma ou mais conexões simultâneas.

Predicado 3 *component(c, f, r)*

O predicado 3 representa um componente **c** válido, com facetas **f** e receptáculos **r**.

3.1.2 Mecanismo de Binding Horizontal

O *binding* horizontal representa a conexão entre a faceta de um componente com o receptáculo de outros componentes. As regras e predicados a seguir representam o *binding* e as restrições necessárias para validá-los.

Predicado 4 *connection(c1, f, c2, r)*

Predicado 5 *compatible(x, y)*

Regra 1 *isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1), receptacle(R, T2, multiple, C2), compatible(T1, T2)*

Regra 2 *isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1), receptacle(R, T2, simple, C2), compatible(T1, T2), connection(C3, Fq, C2, R)* $\rightarrow (C3 = C1), (Fq = F)$

O primeiro predicado representa uma conexão, informando que o componente **c1** possui uma faceta **f** que se conecta no componente **c2** através do receptáculo **r**. O segundo predicado representa que o tipo **x** é compatível com o tipo **y**. As regras 1 e 2 expressam que uma conexão da faceta **F** do componente **C1**, que implementa a interface **T1**, com o receptáculo **R** do componente **C2** que aceita conexões de uma interface **T2**, só seja válida se **T1** e **T2** forem compatíveis entre si. O predicado de compatibilidade de interfaces é implementado pela tecnologia utilizada na implementação do *middleware*.

Os receptáculos possuem duas diferentes configurações de cardinalidade, simples e múltipla. A regra 1 garante que os receptáculos classificados como múltiplo aceitem mais de uma conexão, desde que as interfaces sejam compatíveis entre si. Por sua vez, a regra 2 garante que receptáculos classificados como simples aceitem apenas uma conexão e as interfaces sejam igualmente compatíveis entre si.

PUC-Rio - Certificação Digital Nº 1121796/CA

3.1.3
Componente composto

Um componente composto **CC** é representado por uma composição **A**, facetas **Fa** e um conjunto de receptáculos **Ra**. Onde uma composição é definida como um conjunto de componentes e *binds* criados entre eles.

$$CC = \langle A, F_A, R_A \rangle$$

O componente composto disponibiliza duas visões de componente. A primeira visão é de um componente primitivo com as três facetas básicas (*IComponent, IReceptacles e IMetaInterface*), facetas providas pelo próprio componente composto e facetas e receptáculos externalizados de seus subcomponentes. A segunda visão representa a visão de componente composto, que permite configurar e interagir com os subcomponentes utilizando as facetas *IContentController* e *ISuperComponent* – adicionadas no SCS-Composite.

3.1.4
Hierarquia de Componentes

Para navegação entre componentes e subcomponentes foram criados os conceitos "**y** pertence ao componente composto **x**" e "**x** possui o subcomponente **y**", capturados pelos predicados e regra a seguir:

Predicado 6 *subcomponent(x, y)*

Regra 3 *composite(X, Y) :- subcomponent(Y, X)*

O predicado 6 representa que **x** é subcomponente de **y**. Já a regra 3 expressa a simetria entre a relação "**x** é subcomponente de **y**" e "**x** tem **y** como um de seus componentes".

Vale destacar que é permitido que um componente composto encapsule outro componente composto. Não foi necessário adicionar regras, já que a regra 3 não restringe o aninhamento recursivo dos componentes.

PUC-Rio - Certificação Digital Nº 1121796/CA

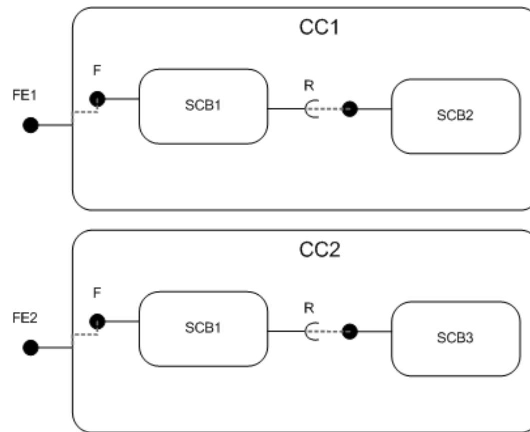

Figura 3.1: Exemplo de compartilhamento do subcomponente SCB1 em dois componentes compostos.

3.1.5 Compartilhamento de subcomponentes

É possível utilizar uma mesma instância de um componente em diversas estruturas, o que permite a economia de recursos, sem perder a característica de preservação do encapsulamento do componente. Porém, quando utilizado em uma infraestrutura de componentes compostos, o compartilhamento de componentes cria um problema de ambiguidade. O cenário descrito pela figura 3.1 evidencia o problema com mais clareza.

Neste cenário temos dois componentes compostos **CC1** e **CC2**, um componente compartilhado **SCB1** e dois componentes **SCB2** e **SCB3**, subcomponentes de **CC1** e **CC2**, respectivamente. Note que o componente **SCB1** possui duas conexões, um *binding* com **SCB2** e outro *binding* com **SCB3**. A ambiguidade ocorre porque **SCB1** não conseguiria identificar de onde foi executada uma chamada – via **FE1** ou **FE2**. Tal ambiguidade corromperia o encapsulamento de componentes, visto que uma chamada a **FE1** geraria uma chamada no subcomponente do componente **CC2**, que pode não ser o efeito desejado em algumas aplicações.

O SCS-Composite permite o compartilhamento da instância de um mesmo componente em mais de um componente composto, desde que o subcomponente não possua receptáculos. Esta restrição remove do modelo a ambiguidade entre dependências e preserva o encapsulamento dos subcomponentes. Por este motivo foi criada a regra 4 que expressa que um subcomponente **Sub** pode ser compartilhado entre os componentes **CC1** e **CC2**, desde que ele não possua receptáculos.

Regra 4 *isAValidComponentSharing(Sub, CC1, CC2) :- subcomponent(Sub, CC1), subcomponent(Sub,CC2), not(receptacle(, , , Sub)), not(CC1 = CC2)*

PUC-Rio - Certificação Digital Nº 1121796/CA

3.1.6
Externalização de Facetas e Receptáculos

O componente composto deve permitir externalização, ou *binding* vertical, de facetas e receptáculos dos subcomponentes. O *binding* vertical torna acessíveis os recursos (facetas) e as dependências (receptáculos) dos subcomponentes através do componente composto. O predicado 7 representa um subcomponente **sub** do componente composto **cc** que possui uma faceta **fsub** externalizada através da faceta **fcc** do componente composto **cc**.

Predicado 7 *exposedFacet(sub, fsub, cc, fcc)*

Regra 5 *isAValidExposedFacet(Sub, Fsub, CC, Fcc) :- exposedFacet(Sub, Fsub, CC, Fcc), facet(Fsub, T1, Sub), facet(Fcc, T2, CC), subcomponent(Sub, CC), compatible(T1, T2)*

A regra 5 expressa que o *binding* vertical só é valido se **Sub** for um subcomponente de **CC**, **Fsub** e **Fcc** forem facetas de **Sub** e **CC**, respectivamente, e **T1** e **T2** forem interfaces compatíveis. De forma análoga, a externalização do receptáculo obedece o predicado e a regra:

Predicado 8 *exposedReceptacle(sub, rsub, cc, rcc)*

Regra 6 *isAValidExposedReceptacle (Sub, Rsub, CC, Rcc) :- receptacle (Rsub, T1, Cardinality, Sub), receptacle(Rcc, T2, Cardinality, CC), exposedReceptacle(Sub, RSub, CC, Rcc), subcomponent(Sub, CC),compatible(T2, T1)*

Onde **Sub** representa o subcomponente do componente composto **CC**, **Rsub** representa o receptáculo que será externalizado no componente composto representado por **Rcc** e, **T1** e **T2** representam as interfaces aceitas por **Rsub** e **Rcc**, respectivamente.

Tanto na regra 5 quanto na regra 6, **CC** é um componente composto, e como tal, deve obedecer todos as regras de um componente primitivo, com destaque para a regra 3, que verifica se suas conexões (*bindings* horizontais) são válidas.

PUC-Rio - Certificação Digital Nº 1121796/CA

3.1.7 Conexão

Para capturar as questões do conceito de componentes compostos, adicionamos algumas regras suplementares ao conjunto de regras do SCS. No SCS-Composite, dois componentes só podem ser conectados se pertencerem ao mesmo componente composto ou se ambos não pertencerem a nenhum componente composto. Para representar essa restrição, foram criadas as regras:

Regra 7 *isAValidConnection(C1, F, C2, R) :- not(subcomponent(C1,)), not(subcomponent(C2,)), facet(F, T1, C1), receptacle(R, T2, , C2), compatible(T1, T2)*

Regra 8 *isAValidConnection(C1,F,C2,R) :- subcomponent(C1, U), subcomponent(C2, U), facet(F, T1, C1), receptacle(R, T2, , C2), compatible(T1, T2).*

Medeiros entendeu que era importante garantir que a externalização de um receptáculo do subcomponente no componente composto não fosse feita simplesmente expondo diretamente a referência do receptáculo do subcomponente. Tal abordagem iria ferir as regras 7 e 8. Por isso foi criada uma terceira regra que obriga a criação de uma indireção para todo *binding* vertical de um componente composto - descrito pela regra 9.

Regra 9 *connectedByComposite(Fext, Cext, Rsub, Sub, Rcc, CC):- connection(Cext, Fext, CC, Rcc), exposedReceptacle(Sub, Rsub, CC, Rcc)*

Onde **Fext** representa a faceta do componente **Cext** que será conectado ao receptáculo **Rsub** do subcomponente **Sub**, através de uma indireção representada pelo receptáculo **Rcc** situado no componente **CC**. O cenário da figura 3.2 descreve um exemplo visual que demonstra a necessidade da indireção.

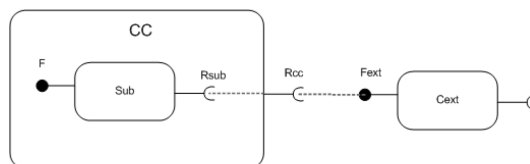

Figura 3.2: Exemplo de indireção de um receptáculo.

Sem tal regra não seria possível a conexão entre facetas **Fext** e receptáculos **Rcc**, já que o componente **Sub** é um subcomponente de um componente composto e **Cext** não pertence ao mesmo subcomponente que **Sub** – regras 7 e 8.

PUC-Rio - Certificação Digital Nº 1121796/CA

3.2 Estudo preliminar

O modelo foi definido pelos predicados e regras descritos na seção 3.1. A maioria dos predicados e regras são bem claros e foram criados para explicitar como o suporte a componentes compostos deve ser introduzido em um sistema de componentes de software. No entanto, algumas regras – 4, 7, 8 e 9 – podem restringir a construção de alguns projetos.

Juntamente com as regras 7 e 8, as regras 4 e 9 são regras restritivas e foram criadas propositalmente para garantir o melhor encapsulamento do componente composto. Elas permitem manter a visão do componente composto como um componente primitivo, o que permite que o usuário o enxergue como uma caixa preta, sem a necessidade de adicionar complexidade ao resto do código e ao próprio controle do componente composto. Sem a restrição da regra 9, por exemplo, o usuário conseguiria ter acesso aos recursos internos do componente composto ao conseguir acessar o receptáculo diretamente. Da mesma forma, sem a regra 4, seria muito complicado para o usuário implementar, por exemplo, a operação *shutdown()* da faceta *IComponent*, visto que o componente precisaria identificar quais receptáculos ele deveria desconectar antes de se desligar por completo.

Além dessas duas regras nós sentimos a necessidade de criar uma nova regra, análoga à regra 9, para o *binding* da faceta. Assim como o modelo exige a criação de uma indireção representada por receptáculos de subcomponentes para que estes sejam conectados a facetas de componentes externos, é necessário exigir indireções representadas por facetas externalizadas para que essas possam ser conectadas a receptáculos de componentes externos. A indireção possibilita que a faceta externalizada do componente composto faça parte do próprio componente em questão e não do subcomponente - vide regra 10.

Regra 10 *connectedByComposite(Rext, Cext, Fsub, Sub, Fcc, CC):- connection(CC, Fcc, Cext, Rext), exposedFacet(Sub, Fsub, CC, Fcc)*

Onde **Rext** representa o receptáculo do componente **Cext** que será conectado à faceta **Fsub** do subcomponente **Sub**, através de uma indireção representada pela faceta **Fcc** situado no componente **CC**. Esta e outras regras da especificação formal foram testadas em Prolog (SWI Prolog[1]) para garantir que nenhum cenário criado por Medeiros deixasse de funcionar corretamente.

Nosso estudo também analisou a interface proposta pelo SCS-Composite, que introduziu as novas funcionalidades no SCS. Em nossa análise observamos

[1]http://www.swi-prolog.org/

PUC-Rio - Certificação Digital Nº 1121796/CA

que a proposta não direcionou esforços na criação de exceções para tratamento de falhas. Também observamos a necessidade de adicionar duas operações, *unbind()* e *retrieveBinding()* à interface *IContentController*. As interfaces com as modificações sugeridas se encontram no Apêndice A. Mais detalhes serão destacados na seção 3.3 que trata da implementação do SCS-Composite.

3.3 Implementação para o trabalho

Este trabalho implementou uma nova versão da especificação formal do SCS-Composite, com base na SCS versão 1.2.3. O SCS já foi implementado em diversas linguagens como C++, Lua, Java e C#. Porém nós optamos por implementar o suporte a componente compostos apenas na versão Lua, por ser uma linguagem de fácil prototipação.

Nossa implementação do SCS-Composite utilizou como base a implementação inicial de Medeiros, que era baseada na versão 1.2.1 do SCS. Entretanto houveram várias mudanças na implementação do SCS entre as versões 1.2.1 e 1.2.3. Na versão 1.2.1 o SCS possuía um módulo principal chamado *scs.core.base* que continha toda a implementação do *ComponentContext* e das três facetas básicas. Nesta versão as estruturas de dados são organizadas de uma forma que tornava difícil para o usuário do *middleware* estender as facetas básicas e a identificar a responsabilidade de cada entidade do modelo de componentes.

Como a versão atual do SCS traz a facilidade de extensão e organização do código, nós optamos por implementar o SCS-Composite praticamente do zero. Para a construção do mesmo, nós criamos um módulo a parte, permanecendo com o módulo *scs.core* inalterado e criando o módulo *scs.composite*. Nós também implementamos as modificações sugeridas neste trabalho, que foram descritas na seção 3.2 e no Apêndice A, com destaque para o tratamento de exceções e a implementação da nova regra 10. Por conta das regras 7 e 8 nós tivemos que implementar uma extensão ao código da faceta *IReceptacles*, o que torna o código do SCS-Composite um pouco mais intrusivo. Por outro lado, o SCS já está preparado para aceitar novas implementações de facetas básicas (função *updateFacet()* do *ComponentContext*), como descrito no manual do SCS Lua[2] seção 3.3.

A mudança de versão do SCS também nos obrigou a atualizar a versão do ORB OiL Tecgraf/PUC-Rio (a) para a versão 0.5 e da biblioteca auxiliar Loop Tecgraf/PUC-Rio (b) versão 3.0 beta. Podemos destacar no OiL a nova forma

[2]https://jira.tecgraf.puc-rio.br/confluence/download/attachments/18612229/scsLuaTutorial-1.2.2_0.pdf

PUC-Rio - Certificação Digital Nº 1121796/CA

de lançamento de exceções CORBA e a maior facilidade de identificar erros de programação no ORB. No Loop, podemos destacar as novas diretrizes para a criação de classes – vide manual no website da biblioteca[3]. Duas mudanças positivas, mas implicaram em alterar inteiramente o código do SCS-Composite.

No OiL, uma tabela que seja exportada como objeto CORBA (*org.omg.CORBA.Object*) oferece duas visões. A primeira visão da própria tabela com suas funções e parâmetros, e a segunda visão representa o objeto que possui apenas as funções da interface CORBA. Como otimização, a nova versão do OiL fornece ao código local da aplicação a visão local da tabela, ao invés de fornecer a visão da interface CORBA - chamada pelo OiL de *proxy*. O problema dessa abordagem é que o OiL retorna visões diferentes do mesmo objeto dependendo se a tabela se encontra na mesma ou em outra instância do OiL. Para que o OiL volte a enviar a visão *proxy* do objeto, é necessário iniciar o ORB como descrito no código 3.1, caso contrário o SCS-Composite não funcionará corretamente.

```
1 local oil = require "oil"
2
3 -- OiL configuration
4 local orb = oil.init({localrefs = "proxy"})
```

Código 3.1: Forma correta de iniciar o OiL para o uso do SCS-Composite.

Também fizemos pequenas refatorações na estrutura do SCS 1.2.3 para conseguirmos reutilizar algumas funcionalidades do atual código nas novas facetas. Em particular, refatoramos a função *putFacet()* da entidade *ComponentContext*. A função é responsável por adicionar uma faceta à estrutura de dados, verificar se a faceta pode ser adicionada e registrar a faceta no ORB *newservant()*. Na nossa versão, a verificação e o registro da faceta são feitos pela nova função *registerFacet()*, o que permite que outras entidades, como a faceta *IContentController* possam utilizar esta função na implementação do *bindFacet()*. Desta forma, conseguimos reutilizar o código do SCS que já valida um conjunto de regras do modelo.

As seções a seguir irão descrever a implementação, na linguagem Lua, de cada um dos predicados e regras descritos neste capítulo e destacar os principais desafios encontrados na implementação do SCS-Composite. Em seguida será apresentado uma ferramenta que criamos para visualização da estrutura do componente (facetas, receptáculos e subcomponentes) em tempo de execução.

[3]http://loop.luaforge.net/

PUC-Rio - Certificação Digital Nº 1121796/CA

3.3.1 Implementação das regras

O SCS Lua 1.2.3 é implementado pelas três facetas básicas e a entidade *ComponentContext* que representa o componente localmente, guarda os metadados do componente e fornece operações para registrar, remover e consultar facetas e receptáculos. Os fatos relativos aos predicados 1 e 2 são adicionados pela entidade *ComponentContext* na fase de criação do componente, com as operações representadas no Código 3.2:

```
1 function addFacet(self, name, interface, implementation, key) --
    Fato 1
2 function addReceptacle(self, name, interface, multiplex) -- Fato 2
```

Código 3.2: Operações de adicionar facetas e receptáculos no componente.

O predicado 3 é igualmente refletido nas funções **addFacet()** e **addReceptacle()** do *ComponentContext*.

As regras do mecanismo de *binding* horizontal também são implementadas no SCS. A operação **is_a()** verifica se o uma interface é compatível com um outra interface - predicado 5. O **connect()** do SCS representa o predicado 4 e implementa as regras 1 e 2 - representado no código 3.3:

```
1 -- connection(C1, F, C2, R)
2 function connect(self, receptacle, object)
3   ....
4   -- isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1),
       receptacle(R, T2, _, C2), compatible(T1, T2)
5   -- compatible(x, y)
6   status, err = pcall(object._is_a, object, desc.interface_name)
7   if not (status and err) then
8     error{ _repid = "IDL:scs/core/InvalidConnection:1.0" }
9   end
10  ....
11
12  -- isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1),
       receptacle(R, T2, simple, C2),
13  --     compatible(T1, T2), not(connection(C3, _,C2,R)), not(C1 =
      C3).
14  if not desc.is_multiplex and self._numConnections > 0 then
15    error{ _repid = "IDL:scs/core/AlreadyConnected:1.0" }
16  end
17
18  -- isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1),
       receptacle(R, T2, multiple, C2),
19  -- compatible(T1, T2).
20  ....
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
21 end
```

Código 3.3: A nova implementação do connect() da faceta *IReceptacles*.

Os predicados e regras anteriores foram introduzidos pela versão original do SCS, as regras a partir da 2 foram adicionadas pelo modelo de componentes compostos, que foi implementado no SCS-Composite. Apesar de Medeiros ter criado uma implementação em seu trabalho, em nosso trabalho nós fizemos uma reengenharia da implementação das regras do SCS-Composite, sem alterar suas características principais – como as duas facetas adicionais. A faceta *ISuperComponent* foi introduzida para que seja possível verificar se um componente **y** pertence ao componente **x** – regra 3. O código 3.4 descreve a interface CORBA da faceta *ISuperComponent*.

```
1 interface ISuperComponent {
2   void addSuperComponent(in scs::core::IComponent icomponent)
        raises(InvalidComponent);
3   boolean removeSuperComponent(in scs::core::IComponent iComponent
        );
4   IComponentSeq getSuperComponents();
5 };
```

Código 3.4: A interface da faceta *ISuperComponent*.

Uma das responsabilidades da faceta *IContentController* é adicionar, remover e recuperar os subcomponentes do componente composto. O código 3.5 descreve as operações de configuração dos subcomponentes no componente composto – predicado 6.

```
1 interface IContentController {
2   ...
3   MembershipId addSubComponent(in scs::core::IComponent icomponent
        ) raises(InvalidComponent, ComponentFailure,
        UnshareableComponent);
4   boolean removeSubComponent(in MembershipId id);
5   MembershipDescriptionSeq getSubComponents();
6   ...
7 };
```

Código 3.5: Operações de configuração da faceta *IContentController*.

A operação **addSubComponent()** implementa a regra 4 que expressa que um subcomponente com receptáculos não pode ser compartilhado entre mais de um componente composto. O código 3.6 verifica se o componente possui receptáculos (operação **getReceptacles()**) e se o componente já faz parte de um componente composto (operação **getSuperComponents()**). A

PUC-Rio - Certificação Digital Nº 1121796/CA

operação lança a exceção **UnshareableComponent**, caso a regra não seja cumprida.

```
1 function ContentController:addSubComponent(subComponent)
2   ...
3
4   ok, metaInterfaceFacet = pcall(subIComponent.getFacetByName,
      subIComponent, utils.IMETAINTERFACE_NAME)
5   if not ok or not metaInterfaceFacet then
6     error( orb:newexcept{ _repid = compositeIdl.throw.
        InvalidComponent })
7   end
8   metaInterfaceFacet = orb:narrow(metaInterfaceFacet, utils.
      IMETAINTERFACE_INTERFACE)
9
10  -- isAValidComponentSharing(Sub, CC1, CC2) :- subcomponent(Sub,
      CC1),
11  --     subcomponent(Sub,CC2), not(receptacle( , , , Sub)), not(
      CC1 = CC2)
12  if #metaInterfaceFacet:getReceptacles() > 0 and #superCompFacet:
      getSuperComponents() > 0 then
13    error( orb:newexcept{ _repid = compositeIdl.throw.
        UnshareableComponent })
14  end
15 end
```

Código 3.6: O código da função *addSubComponent()* da faceta *IContentController*.

O SCS-Composite reflete o predicado 7 através da função **bindFacet()** da faceta *IContentController*. Nós implementamos a regra 5 na própria operação **bindFacet()**. Como esta operação reusa operações do *ComponentContext*, que testam praticamente todo corpo da regra em questão, nós só precisamos testar o predicado **subcomponent(Sub, CC)** na operação **bindFacet()**, descrito no código 3.7.

```
1 function ContentController:bindFacet(connectorID,
    internalFacetName, externalFacetName)
2   ...
3   -- connectedByComposite(Rext, Cext, Fsub, Sub, Fcc, CC ):-
4   --     connection(CC, Fcc, Cext, Rext), exposedFacet(Sub, Fsub,
      CC, Fcc)
5   facet = orb:narrow(facet, interfaceName)
6   local proxyFacetRef = Proxy(facet)
7   ...
8   context:registerFacet(externalFacetName, interfaceName,
      proxyFacetRef, nil)
9   context:setFacetAsBind(externalFacetName)
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
10
11   return context:getFacetByName(externalFacetName).bindingId
12 end
```

Código 3.7: O código da função *bindFacet()* da faceta *IContentController*.

De forma análoga, o SCS-Composite reflete o predicado 8 através da função **bindReceptacle()** da faceta *IContentController*, que também implementa a regra 6. Como visto no código 3.8, a operação **bindReceptacle()** obtêm **T1**, **T2** e **Cardinality** do próprio receptáculo **Rsub**. Como utilizamos operações do *ComponentContext*, também não precisamos validar *receptacle (Rsub, T1, Cardinality, Sub), receptacle(FC, T2, Cardinality, CC)*, restando apenas o trecho *exposedReceptacle(Sub, RSub, CC, Rcc)*.

```
1 function ContentController:bindReceptacle(connectorID,
     internalReceptacleName, externalReceptacleName,
     componentPermission)
2   ...
3   --isAValidExposedReceptacle (Sub, CC, Rsub, FC) :- receptacle (
       Rsub, T1, Cardinality, Sub), receptacle(FC, T2, Cardinality,
       CC),
4   --  exposedReceptacle(Sub, RSub, CC, Rcc), subcomponent(Sub, CC)
       ,compatible(T2, T1)
5   local ok, subcomponent = pcall(self.findComponent, self,
       connectorID)
6    if not ok or not subcomponent then
7     error( orb:newexcept{ _repid = compositeIdl.throw.
         ComponentNotFound, id = connectorID })
8   end
9   ...
10  -- Obtem descricao do receptaculo do subcomponente.
11  local recptacleDescription = descriptions[1]
12  local isMultiplex = recptacleDescription.isMultiplex
13  local interfaceName = recptacleDescription.interface_name
14
15  context:addReceptacle(externalReceptacleName, interfaceName,
       isMultiplex)
16  context:setReceptacleAsBind(externalReceptacleName,
       iSubReceptacle, internalReceptacleName, componentPermission)
17  ...
18 end
```

Código 3.8: O código da função *bindReceptacle()* da faceta *IContentController*.

As regras 7 e 8 foram criadas para garantir o melhor encapsulamento do componente composto e são validadas na fase de conexão de uma faceta em um receptáculo. Essa validação deve existir independente da conexão ter sido

PUC-Rio - Certificação Digital Nº 1121796/CA

feita entre componentes primitivos ou entre *binds* de componentes compostos. Para garantir este comportamento, estendemos a faceta básica *IReceptacles*. Caso as regras não sejam cumpridas, a operação lança uma exceção CORBA **InvalidComponent**, como descrito no código 3.9:

```
1 function newConnect(self, name, connection)
2   ...
3   -- Verifica compatibilidade entre a faceta e o receptaculo
4   if not verifyCompatibility(self, name, iCompConnection) then
5     error( orb:newexcept{ _repid = compositeIdl.throw.
          InvalidComponent })
6   end
7
8   -- facet(F, T1, C1), receptacle(R, T2, , C2), compatible(T1, T2)
       .
9   return SuperIReceptacles.connect(self, name, connection)
10 end
11
12
13 function verifyCompatibility(self, name, iComponent)
14   ...
15   -- not(subcomponent(C1, )), not(subcomponent(C2, ))
16   if #superComponentList == 0 and #connSuperComponentList == 0
       then
17     return true
18   end
19
20   if ((#superComponentList > 0 and #connSuperComponentList == 0)
21       or (#superComponentList == 0 and #connSuperComponentList >
           0)) then
22     ...
23     return false
24   end
25
26   -- subcomponent(C1, U), subcomponent(C2, U)
27   for _,superComponent in ipairs(superComponentList) do
28     local superComponentFacet = orb:narrow(superComponent, utils.
          ICOMPONENT_INTERFACE)
29     local superContentFacet = superComponentFacet:getFacetByName(
          utils.ICONTENTCONTROLLER_NAME)
30     superContentFacet = orb:narrow(superContentFacet, utils.
          ICONTENTCONTROLLER)
31
32     for _,connSuperComponent in ipairs(connSuperComponentList) do
33       local connSuperComponentFacet = orb:narrow(
           connSuperComponent, utils.ICOMPONENT_INTERFACE)
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
34        local connSuperContentFacet = connSuperComponentFacet:
              getFacetByName(utils.ICONTENTCONTROLLER_NAME)
35        connSuperContentFacet = orb:narrow(connSuperContentFacet,
              utils.ICONTENTCONTROLLER)
36
37        if superContentFacet:getId() == connSuperContentFacet:getId
              () then
38          return true
39        end
40      end
41    end
42
43    ...
44    return false
45 end
```

Código 3.9: A nova implementação da função *connect()* da faceta *IReceptacles*.

A regra 9 e a regra 10, criadas neste trabalho, foram implementadas nas facetas *IReceptacles* e *IContentController*, respectivamente. As duas regras adicionam a necessidade de lidar com *proxies* de facetas (código 3.10) e receptáculos (código 3.11). Nós estudamos algumas opções de implementação destas duas regras, que serão descritas nas próximas seções.

```
 1 function newConnect(self, name, connection)
 2   ...
 3   -- connectedByComposite(Fext, Cext, Rsub, Sub, Rcc, CC ):-
 4   --     connection(Cext, Fext, CC, Rcc), exposedReceptacle( Sub,
         Rsub, CC, Rcc)
 5   local proxy = Proxy(orb)
 6   local connIComponentFacet = orb:narrow(connection:_component(),
         utils.ICOMPONENT_INTERFACE)
 7   proxy:setProxyComponent(connIComponentFacet, permission)
 8   ...
 9   proxySuperComponent:addSuperComponent(iComponent)
10   ...
11   local proxyFacet = proxy:getFacet(connectionInterface).facet_ref
12   bindIReceptacle:connect(receptacleName, proxyFacet)
13   ...
14   return SuperIReceptacles.connect(self, name, connection)
15 end
```

Código 3.10: A implementação da regra nova 10 que cria o Proxy da Faceta na função *bindFacet()* da faceta *IContentController*.

```
 1 function ContentController:bindFacet(connectorID,
       internalFacetName, externalFacetName)
 2   ...
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
3   -- connectedByComposite(Rext, Cext, Fsub, Sub, Fcc, CC ):-
4   --     connection(CC, Fcc, Cext, Rext), exposedFacet(Sub, Fsub,
        CC, Fcc)
5   facet = orb:narrow(facet, interfaceName)
6   local proxyFacetRef = Proxy(facet)
7   ...
8   context:registerFacet(externalFacetName, interfaceName,
        proxyFacetRef, nil)
9   context:setFacetAsBind(externalFacetName)
10
11  return context:getFacetByName(externalFacetName).bindingId
12 end
```

Código 3.11: A implementação da regra 9 que cria o Proxy do receptáculo na função *connect()* da faceta *IReceptacles*.

3.3.2
Proxy no Receptáculo

PUC-Rio - Certificação Digital Nº 1121796/CA

Ao implementar a regra 9, Medeiros optou por criar um *proxy* bem simples que apenas sobrescrevia a função CORBA **get_component()**, chamada no OiL de **_component()**, que é chamado na operação **connect()** do componente composto. Essa abordagem obrigou a implementação a tratar a conexão entre o *proxy* e o receptáculo do subcomponente como algo especial, porque o *proxy* não possui as facetas (*IComponent* e *ISuperComponent*) necessárias para validar as regras 7 e 8 –implementadas na operação **connect()**. Medeiros então, adicionou a função **sentByComposite()** que verifica se o objeto a ser conectado é um *proxy* criado pelo componente composto, em caso positivo as regras 7 e 8 não são testadas.

Como tentativa de melhor robustez da implementação, nós criamos mais duas possíveis implementações para a regra 9. As duas implementações criavam um *proxy* de um componente ao invés de um *proxy* apenas da faceta. Tanto a implementação de Medeiros, quanto as duas implementações propostas estão representadas na figura 3.3.

A primeira ideia que tivemos (representada na figura 3.3 (b)) foi criar um componente especial, interno ao componente composto, que fosse instanciado na operação **bindReceptacle()**, contendo um receptáculo com todas as conexões do receptáculo do componente composto. Como o *proxy* seria implementado como um componente, nós não teríamos o problema de ignorar (*bypass*) as validações do **connect()**. Por outro lado nós criaríamos alguns problemas com essa abordagem, como: (i) se criarmos o receptáculo no **bindReceptacle()** e o conectarmos no subcomponente responsável pelo *bind*, nós

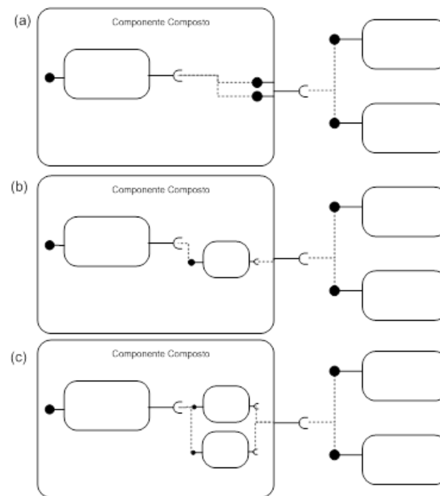

Figura 3.3: (a) Medeiros; (b) um componente que representa todas as conexões; (c) Um componente para cada conexão.

poderíamos gerar um mau funcionamento em componentes que possuírem uma lógica de executar uma série de chamadas quando uma nova conexão acontecer; (ii) com apenas um componente criado no **bindReceptacle()** poderíamos gerar um mau funcionamento em componentes que necessitarem de mais de uma conexão em seus receptáculos; (iii) o *proxy* teria que lidar com formas de agrupar os resultados, para o caso de facetas com operações que possuam retorno.

Os problemas encontrados nos fizeram chegar na segunda implementação (representada na figura 3.3 (c)). A ideia foi criar um *proxy* de um componente, interno ao componente composto, para cada conexão bem sucedida. Como na primeira implementação (figura 3.3 (a)), o *proxy* é criado na operação **connect()** do componente composto – diagrama de sequência expresso na figura 3.4. Essa abordagem nos permite tratar a conexão entre o proxy e o receptáculo do subcomponente da mesma forma que tratamos qualquer outro tipo de conexão, removendo a necessidade de criar funções auxiliares que ignoram validações descritas na regra do SCS-Composite – como na implementação de Medeiros – e permitir que o subcomponente trate uma conexão feita pelo *bind* como outra qualquer. Por esses motivos, nós optamos por essa implementação de *proxy*.

Assim como no SCS, o *proxy* que criamos permite que o subcomponente tenha acesso às outras facetas do componente conectado, via a faceta *IComponent* do *proxy*. Para que o *proxy* tivesse este comportamento foi necessário sobrescrever as operações **getFacetByName()** e **getFacet()**. Para que o *proxy* do componente funcione como um subcomponente, e respeite as regras 7 e 8, nós implementamos a faceta do *ISuperComponent* como uma faceta padrão.

PUC-Rio - Certificação Digital Nº 1121796/CA

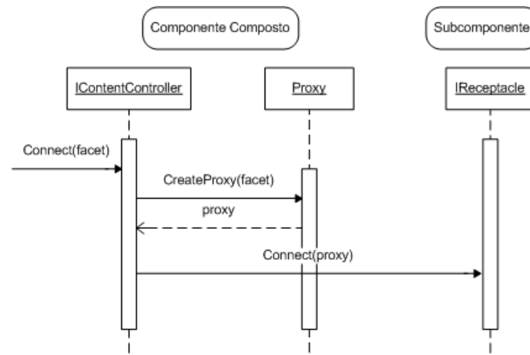

Figura 3.4: Diagrama de sequência que descreve a criação do Proxy na operação *connect()* da Faceta *IReceptacles*.

O acesso ao componente via *proxy* pode ser encarado como uma quebra das regras de encapsulamento do componente composto, por isso preferimos adicionar o parâmetro *permission* na operação **bindReceptacles()**. Esta variável permite definir se o receptáculo do subcomponente terá acesso às demais facetas da conexão. Assim deixamos a cargo do desenvolvedor a possibilidade de permitir este acesso.

PUC-Rio - Certificação Digital Nº 1121796/CA

3.3.3 Proxy nas Facetas

Na seção 3.2 nós descrevemos a necessidade de existir uma faceta (objeto CORBA) no componente composto que represente a faceta do *binding* – criada na operação **bindFacet()** da faceta *IContentController*. A falta desta regra (regra 10) foi identificada quando tentamos criar um teste com componentes aninhados. O teste, descrito na figura 3.5, consiste em criar um componente composto **CC1** que exporta a faceta do componente **CC2**. O Componente **CC2** possui dois subcomponentes que implementam a mesma faceta e um conector que foi criado para agrupar as facetas de **Sub1** e **Sub2** e exporta-las como uma única faceta **F**.

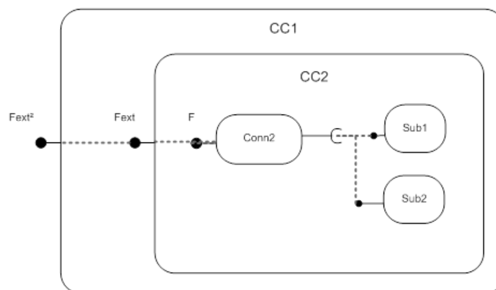

Figura 3.5: Cenário criado para verificar a necessidade da regra 10.

Sem a regra 10, quando tentávamos identificar o componente responsável pela faceta **Fext2**, nós recebíamos a referência para o componente **Conn2**

como retorno da operação **get_component()** do objeto CORBA, quando deveríamos receber como retorno o componente **CC1**. Esse comportamento acontecia porque os *binds* das facetas apenas exportavam as facetas dos subcomponentes. O mesmo erro seria identificado caso a faceta **Fext2** tentasse se conectar com um componente com o mesmo grau de aninhamento, visto que na realidade a tentativa de conexão seria feita entre a faceta do componente **Conn2** e um receptáculo que não pertenceria ao mesmo subcomponente – quebra da regra 7.

Diferente do proxy nos receptáculos, nós desenvolvemos o *proxy* das facetas de forma simples, criando um objeto *proxy* para cada *bind* bem sucedido. O *proxy* direciona todos as chamadas diretamente para a faceta, com exceção da operação **_component()**, que define a faceta como parte do componente composto em questão. Desta forma o *proxy* funciona como uma faceta como outra qualquer, que se registra no *ComponentContext* e é fornecida nas operações das facetas básicas. Adicionalmente, a operação **bindFacet()** cria um identificador que representa o *bind* no componente, que nós optamos por armazenar em um mapa dentro do *ComponentContext*. A figura 3.6 descreve o diagrama de sequência responsável pela criação do *bind* da faceta.

PUC-Rio - Certificação Digital Nº 1121796/CA

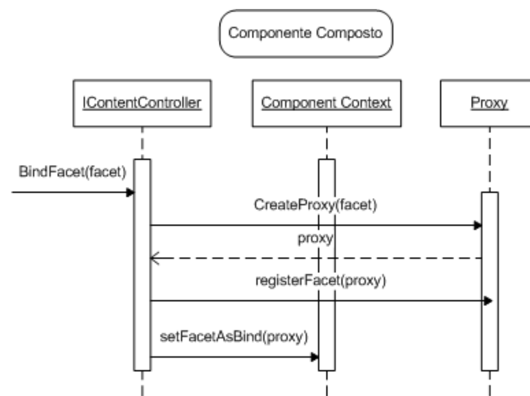

Figura 3.6: Diagrama de sequência que descreve a criação do Proxy na operação *bindFacet()* da Faceta *IContentController*.

3.3.4
Ferramentas de inspeção de componentes

Enquanto criávamos diferentes cenários e testes para o SCS-Composite, nós vimos a necessidade de uma ferramenta que permitisse inspecionar a organização dos componentes em tempo de execução. A ferramenta que criamos, apelidada de *componentInspector*, recebe um componente composto como parâmetro e gera uma saída com as entidades e os *binds* em formato

de uma árvore de relacionamentos. A ferramenta permite configurar quais entidades serão mostradas na estrutura como: componentes e subcomponentes; facetas; receptáculos; conexões. No configurador também é permitido criar padrões para identificar entidades específicas, como conectores e *proxies*.

Inicialmente nós geramos a saída da ferramenta no próprio console, mas depois que começamos a testar o *middleware* em modelos muito grandes, nós optamos por criar também uma saída em HTML e CSS que permite visualizar a estrutura com mais clareza, disponibilizando um conceito de árvore, com objetos que permitem expandir e contrair os nós da árvore (componentes e componentes compostos) e que possuem imagens para representar as entidades do sistema.

A ferramenta atualmente é um módulo Lua que espera uma referência para o componente raiz, uma entidade de visualização (atualmente IO ou HTML), e a tabela de configuração onde o usuário define as entidades que serão mostrada na visualização. O *componentInspector* foi criado utilizando o padrão *Decorator* para gerar a saída da ferramenta, sendo mais fácil a criação de outros visualizadores.

O *componentInspector* foi criado para a fase de debug, mas pode ser muito interessante para administradores de sistemas grandes, com muitos componentes e conexões, que precisam visualizar o estado do projeto de forma visual e em tempo real.

PUC-Rio - Certificação Digital Nº 1121796/CA

4
Análise

Este capítulo se dedica a analisar as mudanças nas regras e implementação do SCS-Composite em um ambiente de captura e acesso. Para esta análise nós escolhemos o Sistema de Captura e Acesso CAS que utiliza o SCS como modelo de componente se software. A versão atual do CAS possui um componente *Space* que agrega um conjunto de *SpeedCars* responsáveis pela captura do evento – maiores detalhes na seção 3.3. Este componente simula um componente composto através de uma implementação específica, que exporta todas as facetas e receptáculos dos *Speedcars*.

Tal abordagem introduz problemas, tanto na forma de acessar e controlar os componentes internos, quanto limitar a usabilidade do componente. A impossibilidade de aninhar espaços para criar um evento com múltiplos espaços é uma limitação identificada pelos desenvolvedores do projeto. Ao introduzirmos o conceito de componente composto no sistema, nós conseguirmos adicionar o conceito de espaços aninhados, remover a necessidade de simulação do componente composto e organizar o código do componente *Space* de forma mais simples, ao tirar validações e códigos já implementados e testados pelo SCS-Composite.

A seção 4.1 irá mostrar como foi feita a adaptação do projeto CAS para utilizar componentes compostos e a seguir irá descrever um cenário onde poderemos exercitar e validar o uso do SCS-Composite no CAS (seções 4.2 e 4.3). Por fim, na seção 4.4 nós iremos descrever algumas dificuldades que encontramos ao utilizar as interfaces e a API do SCS-Composite.

4.1
Implementação do CAS com SCS-Composite

O projeto CAS é implementado seguindo um paradigma orientado a componentes ao utilizar o *middleware* SCS. Como visto na seção 2.4 o CAS possui diversos componentes principais, onde cada um possui suas próprias responsabilidades em diferentes fases do processo. O componente *Space* representa o espaço de captura, onde estarão localizados os dispositivos de gravação representados no sistema como componentes *SpeedCar*. Antes de nossa imple-

PUC-Rio - Certificação Digital Nº 1121796/CA

mentação, o *Space* simulava um componente composto.

A adaptação do componente *Space* para um componente composto foi bem fácil de ser concluída, visto que o componente já simulava alguns conceitos de componentes compostos. Removemos as implementações das facetas *IRecord*, *IConfigurable* e *IDataTransfer*, que repassam as chamadas para as respectivas facetas do *SpeedCar*. Em seu lugar criamos conectores com facetas que repassam as informações e receptáculos que explicitam as dependências. Foi necessário também remover as operações de adição e remoção de *SpeedCars* que existiam na faceta *ISpace* do componente *Space*.

O componente *SpeedCar* depende do componente *Logger*, que representa o serviço de log distribuído do projeto. Com ele conseguimos identificar com mais facilidade erros que envolvam mais de um componente distribuído. Como o *Space* funcionava apenas como uma simulação de um componente composto, os projetistas do CAS optaram por deixar o componente *Logger* como uma referência interna do componente, ao invés de exportar a dependência como um receptáculo. Como o uso de receptáculos é uma boa prática, nós mudamos o *SpeedCar* para que a dependência do *Logger* fosse um receptáculo do componente. Desta forma nós criamos também um conector responsável por conectar o *Logger* em todos os componentes *SpeedCar*.

Também foi necessário alterar o componente *Space Configurator* que conecta e configura os *SpeedCars* no componente Space. O código 4.1 mostra esquematicamente como era a configuração antes da modificação, ao utilizar a faceta *ISpace* para adicionar os *SpeedCars* no espaço. Após nossas alterações, o componente passou a buscar a faceta *IContentController*, adicionar o componente como um subcomponente do *Space* e conectar as facetas e receptáculos do *SpeedCar* nos conectores do *Space* - código 4.2.

```
1 -- Faceta ISpace | Componente Space Configurator
2 function connectSpeedCar(self,speedcarComponent)
3   ....
4   local spaceFacet = context.spaceFacet
5   local connectionID = spaceFacet:addSpeedCar(speedcarComponent)
6
7   for _,listener in pairs(context.listeners) do
8     oil.newthread(listener.connected,listener,speedcarComponent)
9   end
10
11  return connectionID
12 end
13
14 -- Faceta ISpace | Componente Space
15 function addSpeedCar(self,speedcar)
16   ....
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
17
18     local index = context.connectionID
19     local component = orb:narrow(speedcar,"IDL:scs/core/IComponent
           :1.0")
20     local config = orb:narrow(component:getFacetByName("
           IConfigurable"),"IDL:cas/configuration/IConfigurable:1.0")
21     local recorder = orb:narrow(component:getFacetByName("IRecord"),
           "IDL:cas/recorder/IRecord:1.0")
22
23     --- Armazena as facetas do Speedcar em uma estrutura interna
24     context.speedcars[index] = {}
25     context.speedcars[index] = component
26     context.speedcars[index]["IRecord"] = recorder
27     context.speedcars[index]["IConfigurable"] = config
28
29     context.connectionID  = context.connectionID+1
30
31     return index
32 end
```

Código 4.1: Implementação do *ConnectSpeedCar* utilizado no *Space Configurator* sem utilizar o SCS-Composite.

```
 1 --- Faceta ISpace | Componente Space Configurator
 2 function Configurator:connectSpeedCar(speedcarComponent)
 3     ...
 4     local spaceContentFacet = spaceFacet:getFacetByName(scsUtils.
           ICONTENTCONTROLLER_NAME)
 5     connectionID = addAsSpeedCar(orb, spaceContentFacet,
           speedcarComponent)
 6
 7     for _,listener in pairs(self.listeners) do
 8       oil.newthread(listener.connected, listener, speedcarComponent)
 9     end
10
11     return connectionID
12 end
13
14 function addAsSpeedCar(orb, iContentFacet, icomponent)
15     local membershipId = iContentFacet:addSubComponent(icomponent)
16     local bindingDescrptions = iContentFacet:getSubComponents()
17
18     -- Conectar o ConfigurableConnector e o RecordConnector
19     for _, bindDesc in ipairs(bindingDescrptions) do
20       ...
21       local iReceptacle = iComponent:getFacetByName(scsUtils.
             IRECEPTACLES_NAME)
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
22      iReceptacle = orb:narrow(iReceptacle, scsUtils.
          IRECEPTACLES_INTERFACE)
23
24      if componentID.name == "RecordConnector" then
25        local recordFacet = icomponent:getFacetByName(Utils.
            IRECORD_NAME)
26        iReceptacle:connect(Utils.IRECORD_NAME, recordFacet)
27      elseif componentID.name == "ConfigurableConnector" then
28        local confFacet = icomponent:getFacetByName(Utils.
            ICONFIGURABLE_NAME)
29        iReceptacle:connect(Utils.ICONFIGURABLE_NAME, confFacet)
30      elseif componentID.name == "DataTransferConnector" then
31        local confFacet = icomponent:getFacetByName(Utils.
            IDATATRANSFER_NAME)
32        iReceptacle:connect(Utils.IDATATRANSFER_NAME, confFacet)
33      end
34    end
35
36    return membershipId
37  end
```

Código 4.2: Implementação do *ConnectSpeedCar* utilizado no *Space Configurator* utilizando o SCS-Composite.

PUC-Rio - Certificação Digital Nº 1121796/CA

Como não existia um conjunto de testes padrão para o CAS, adicionamos alguns testes para facilitar a validação de nossas alterações no sistema. Antes de criá-los, foi necessário implementar uma versão do componente *SpeedCar* para testes, porque cada *SpeedCar* possui um dispositivo de gravação que precisa estar ligado e conectado ao sistema (como câmeras, microfones, apresentação de slides e etc.), o que dificultaria a execução dos testes. A implementação foi criada em Lua e possui as mesmas funcionalidades do componente, com mensagens de log que permitem a compreensão das trocas de mensagens ocorridas entre os componentes do sistema.

Para os testes criamos também um simulador de eventos, que permite a criação da infraestrutura básica do CAS e a criação de um evento com espaços aninhados de forma rápida e prática. O simulador orienta o usuário a iniciar os diversos componentes do CAS na ordem correta, permitindo ao usuário testar diferentes configurações de ambiente, antes de criar um teste completo, além de facilitar testes pontuais durante a fase de desenvolvimento. A mesma ideia do simulador é utilizada nos três testes que criamos.

O teste mais simples tem o intuito de verificar se a infraestrutura foi configurada corretamente e testar o funcionamento da versão de teste do componente *SpeedCar*. Criamos também um teste com espaços aninhados, onde temos dois espaços virtuais, Brasil e Rio de Janeiro e dentro desses espaços

aninhados temos dois outros espaços reais, o primeiro chamado de PUC e o segundo chamado de UERJ. Estes dois últimos, por sua vez, possuem três *SpeedCars* cada um. Para verificar se o teste conseguiu criar corretamente o aninhamento de espaço, utilizamos a ferramenta de inspeção de componentes apresentada na seção 3.3.4.

Enquanto o teste anterior se preocupava com a correta organização dos componentes, o terceiro teste verifica se as conexões entre os diferentes espaços, conectores e dispositivos foram criadas de forma correta. Para este teste nós criamos uma infraestrutura com três espaços aninhados e verificamos se a operação **start()** da faceta *IRecord* do espaço mais externo repassa a chamada até o *SpeedCar* localizado no espaço mais específico. Também verificamos se a operação **stop()** da faceta *IRecord* executada no espaço mais específico altera apenas o estado de seus subcomponentes, deixando os espaços mais externos a ele no estado anterior – estado *recording*.

4.2 Cenários

Para analisar as alterações feitas no sistema, nós optamos por simular um evento com múltiplos ambientes e dispositivos de gravação. O cenário descreve uma palestra que está ocorrendo no auditório do prédio RDC na PUC-Rio. Consideramos que devido ao grande número de participantes foi necessário adicionar uma sala anexa ao auditório. Também foi criada uma sala no auditório da UERJ - local geograficamente distante. As duas salas possuem apenas um microfone e uma câmera que serão utilizados para perguntas e respostas. Já o auditório possui quatro dispositivos de gravação: duas câmeras; um microfone; e um computador gravando a apresentação de slides.

A versão do CAS com o SCS-Composite permite que criemos um espaço virtual para agrupar os três ambientes e controlá-los com maior facilidade, de forma que o administrador opere os três ambientes como um ambiente apenas. Esta característica só é possível com o conceito de aninhamento de ambientes adicionado neste trabalho. A figura 4.1 representa o cenário, onde observamos um ambiente virtual, chamado *Evento*, o ambiente principal onde está ocorrendo a palestra (Auditório PUC-Rio) e as duas salas auxiliares, uma na PUC-Rio e outra na UERJ. Ainda na figura 4.1 temos uma entidade que representa o painel de controle do administrador.

Este cenário é implementado na arquitetura do CAS como um conjunto de componentes. Os principais tipos de componentes presentes são:

- SpeedCar: representa um driver para dispositivos de captura de mídias. Possui a faceta *IRecord* que disponibiliza operações para iniciar, pausar,

PUC-Rio - Certificação Digital Nº 1121796/CA

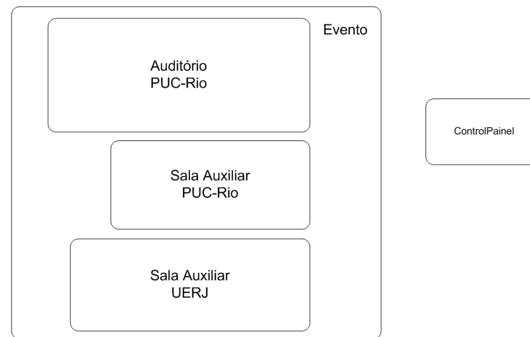

Figura 4.1: Representação gráfica do cenário descrito.

parar e verificar o estado do dispositivo, a faceta *IDataTransfer* que permite transferir as mídias geradas, e a faceta *IConfigurable* que possui operações para definir e recuperar propriedades específicas do componente. O componente também possui o receptáculo *Logger* que explicita a necessidade de conexão com um componente.

– Space: representa o ambiente de captura. Este componente agrega um conjunto de subcomponentes que podem ser SpeedCars ou outros ambientes de captura. Possui as mesmas facetas e receptáculos do SpeedCar e adicionalmente a faceta *ISpace* que fornece operações para recuperar e definir algumas características do espaço.

– Control Panel: disponibiliza ao administrador do sistema maneiras de controlar a infraestrutura como um todo – configuração, monitoramento, registro de eventos e etc..

– SpaceConfigurator: é responsável por configurar o ambiente da sala.

A figura 4.2 descreve como os diferentes componentes se comunicam na infraestrutura do CAS. Podemos notar que a organização dos componentes se assemelha muito com a organização descrita na figura 4.1. Para facilitar o entendimento, nós retiramos os seguintes componentes que não eram relevantes para a avaliação do cenário: os componentes de pós-produção, o repositório de dados, o *SpaceConfigurator*, o barramento de dados Openbus (Tecgraf/PUC-Rio, c) e o *Integration Manager*.

A figura 4.2 descreve os três ambientes reais e um ambiente virtual que agrupa todos os três componentes. Note que todos os quatro ambientes possuem ao menos os três conectores de facetas e um conector de receptáculo. Os conectores foram criados para agrupar os *SpeedCars* (ou os *Spaces*) ao mesmo tempo que desacopla o controle de retorno das operações do componente composto.

A figura 4.1 mostra uma entidade que representa o painel de controle da aplicação, que permite ao administrador gerenciar a infraestrutura do

PUC-Rio - Certificação Digital Nº 1121796/CA

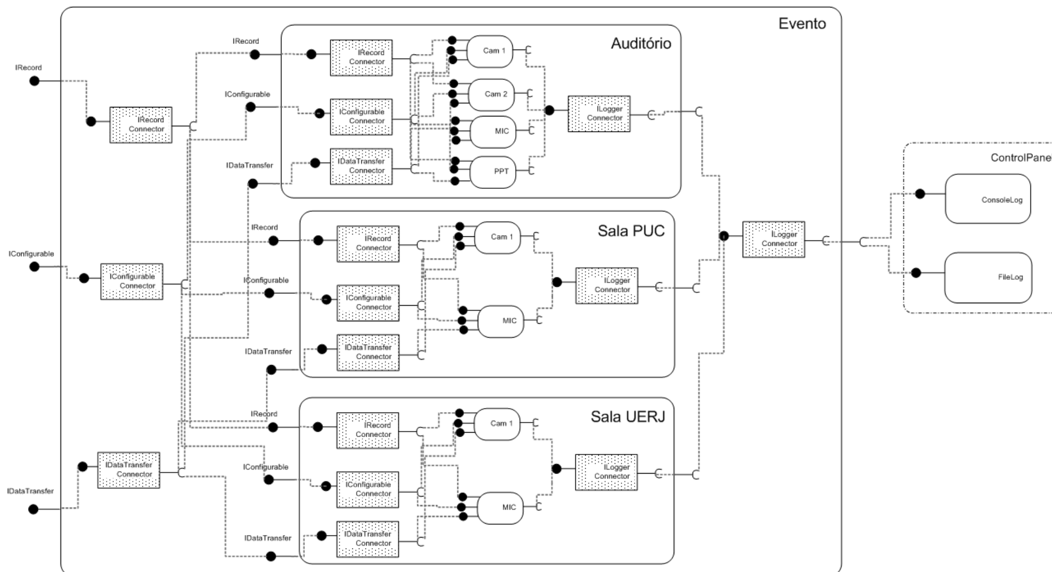

Figura 4.2: Representação dos componentes do cenário descrito.

CAS. Na figura 4.2 nós resumimos o painel de controles a dois componentes Logger que são conectados no receptáculo do espaço virtual. O componente *ConsoleLog* permite ao administrador receber todos os logs importantes (níveis *warning* e *error*) diretamente no console do painel de controles, permitindo ao administrador encontrar soluções que resolvam o problema naquele momento. Já o componente *FileLog* registra todos os logs em um arquivo que pode ser usado pelo administrador, a posteriori, para identificar pontos de falha dos componentes e da comunicação entre os componentes do sistema.

Para inicializar o cenário, utilizamos o simulador de eventos que criamos neste trabalho (descrito na seção 4.1). Após iniciar a infraestrutura básica do CAS, nós devemos criar o espaço virtual para depois criar os outros três espaços. O próprio simulador é responsável por inicializar os conectores de cada espaço. Após a inicialização dos espaços, podemos criar os SpeedCars e conectá-los em cada espaço. Por fim, iniciamos os dois componentes Logger. A figura 4.3 representa o cenário organizado em uma árvore de relacionamentos.

4.3 Análise do Cenário

O cenário (seção 4.2) nos permitiu analisar se as regras e a API do SCS-Composite trazem benefícios para a infraestrutura do CAS. A primeira análise nos traz um sentimento que as mudanças de implementação não foram tão grandes como o imaginado. No *Space*, as implementações das facetas que repassavam as chamadas para os *SpeedCars* foram migradas para a lógica dos conectores. Nós removemos as operações **addSpeedCar()** e **removeSpeed-**

PUC-Rio - Certificação Digital Nº 1121796/CA

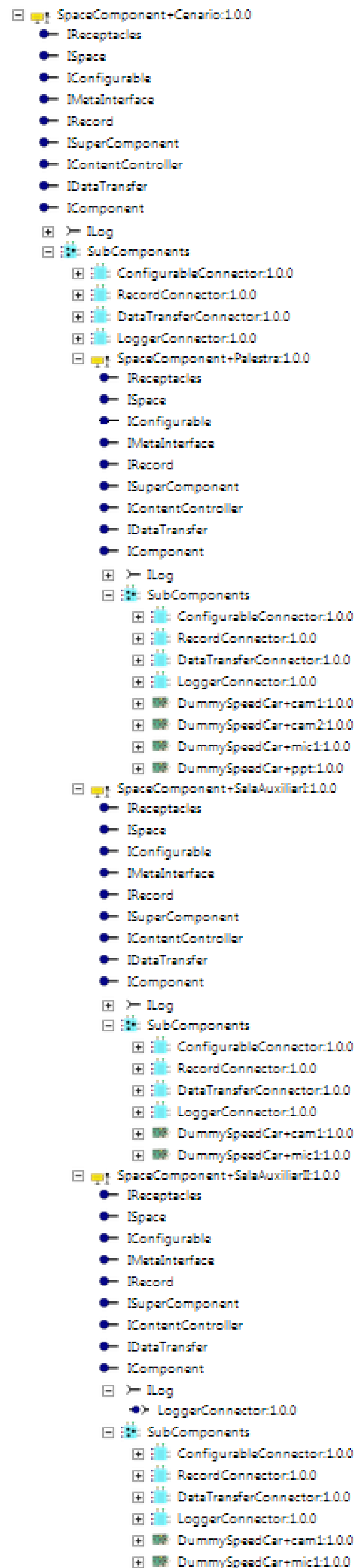

Figura 4.3: Representação do cenário utilizando a ferramenta de inspeção de componentes desenvolvida nesse trabalho.

PUC-Rio - Certificação Digital Nº 1121796/CA

Car(), como também removemos a estrutura de dados responsável por armazenar os *Speedcars* no espaço.

Por outro lado, se olharmos com mais detalhes, veremos que a adição do SCS-Composite nos permitiu (i) adicionar com facilidade uma dependência do componente *Logger* como um receptáculo, tanto do *SpeedCar* quanto do *Space*. Sem o componente composto seria necessário replicar vários mecanismos que já existem no SCS-Composite; (ii) separar a lógica de repassar as chamadas para os *SpeedCars* e agregar o retorno das operações em um conector que também permite explicitar as dependências via receptáculo. Na versão do CAS sem suporte a componente compostos, o componente simulava um *bind* de faceta 1-n; (iii) utilizar a ferramenta de inspeção da infraestrutura do espaço – descrita na seção 3.3.4; (iv) a facilidade de navegar e controlar o componente composto com uma API padrão, amplamente estudada e testada; (v) não precisar reimplementar um controle de componentes compostos; (vi) repassar a responsabilidade de controle dos subcomponentes para o *middleware*, que possui um conjunto de regras que foram criadas para minimizar a inconsistência e aumentar a robustez do sistema.

Após essa pequena análise superficial da infraestrutura do CAS, as próximas seções irão descrever algumas característica e desafios de regras da especificação formal apontadas no capítulo 3 como restritivas e que poderiam limitar o uso do SCS-Composite.

4.3.1 Bind de Faceta

O SCS-Composite implementou a regra 10 de modo que a API crie uma faceta *proxy* no componente composto a cada chamada **bindFacet()**. O *proxy* é criado internamente pela própria implementação do SCS-Composite e adicionado como uma faceta real do componente composto. Com isso, a faceta é retornada nas operações de introspecção da faceta *IMetaInterface* e nas operações da faceta *IComponent*.

A figura 4.4 descreve um subconjunto do cenário – figura 4.1. Nessa figura observamos que existem três *proxies* de facetas do componente composto *Auditório* que foram criados ao exportar a faceta dos conectores – linhas 5, 10 e 15 do código 4.3.

```
1 -- Configurable Connector
2 ...
3 local configurableIComponent = orb:narrow(configurableConnector.
    IComponent, scsUtils.ICOMPONENT_INTERFACE)
4 local configurableConnID = spaceIContent:addSubComponent(
    configurableIComponent)
```

PUC-Rio - Certificação Digital Nº 1121796/CA

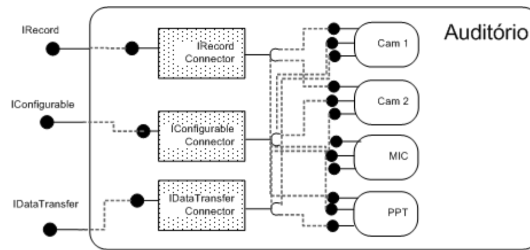

Figura 4.4: Subconjunto do cenário descrito na seção 4.2.

```
5 local configurableBindID = spaceIContent:bindFacet(
      configurableConnID, casUtils.ICONFIGURABLE_NAME, casUtils.
      ICONFIGURABLE_NAME)
6
7 -- Criando o Record Connector
8 ...
9 local recordConnID = spaceIContent:addSubComponent(recordConnector
      .IComponent)
10 local recordBindID = spaceIContent:bindFacet(recordConnID,
      casUtils.IRECORD_NAME, casUtils.IRECORD_NAME)
11
12 -- Cirando o DataTransfer Connector
13 ...
14 local dataTranfConnID = spaceIContent:addSubComponent(
      dataTranfConnector.IComponent)
15 local dataTranfBindID = spaceIContent:bindFacet(dataTranfConnID,
      casUtils.IDATATRANSFER_NAME, casUtils.IDATATRANSFER_NAME)
```

Código 4.3: O código descreve como ocorre o *bind* das facetas no cenário proposto.

A falta da regra 10, e consequentemente do *proxy*, faria com que as chamadas fossem feitas diretamente nos conectores internos do *Auditório*. Sem os *proxies*, os conectores do componente *Evento* não conseguiriam se conectar com as facetas do componente *Auditório*, visto que na realidade a conexão seria feita na faceta dos conectores do *Auditório* que não estão no mesmo subcomponente que os receptáculos do *Evento* – regra 7 e 8. A conexão de um subcomponente com um componente externo não permitiria a visualização do componente composto como um componente primitivo, um conceito importante prescrito no modelo - seção 3.2.

4.3.2 Bind de Receptáculo

O SCS-Composite implementou a regra 9 de modo que toda a conexão feita em receptáculos exportados, através da chamada **bindReceptacle()**, crie

PUC-Rio - Certificação Digital Nº 1121796/CA

um componente *proxy* no componente composto. A própria implementação do SCS-Composite cria o componente *proxy* para cada conexão bem sucedida – função **connect()** da faceta *IReceptacle*. Por ser um componente específico da implementação, a API não os retorna na função **getSubComponents()** da faceta *IContentController*.

A figura 4.5 descreve como os componentes são realmente representados. Como o receptáculo do *Cenário* possui duas conexões, a implementação criou dois componentes *Proxy*, representados na figura como *Proxy I* e *Proxy II*. A linhas 6 do código 4.4 descreve como criamos o *bind* do receptáculo. A API só criará os *proxies* quando a função **connect()**, da faceta *IReceptacles*, for executada.

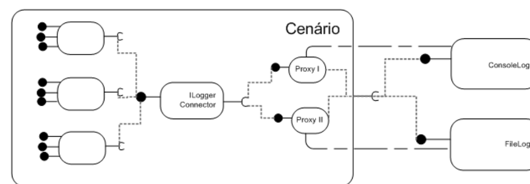

Figura 4.5: Subconjunto do cenário descrito na seção 4.2.

```
1 -- Criando o Logger Connector
2 local loggerCompId = { name = "LoggerConnector", major_version =
     1, minor_version = 0, patch_version = 0, platform_spec = "" }
3 local loggerConnector = ComponentContext(orb, loggerCompId)
4 ...
5 local loggerConnID = spaceIContent:addSubComponent(loggerConnector
     .IComponent)
6 local loggerBindID = spaceIContent:bindReceptacle(loggerConnID,
     casUtils.ILOG_NAME, casUtils.ILOG_NAME, "ALL")
```

Código 4.4: O código descreve como ocorre o *bind* das facetas no cenário proposto.

A falta da regra 9, e consequentemente do *proxy* do receptáculo, não permitiria que as facetas dos componentes de Log fossem conectadas no subcomponente *ILogger Connector*, visto que a faceta e o receptáculo não pertencem ao mesmo componente – regras 7 e 8. Ao criar um *proxy* para cada conexão bem sucedida, o SCS-Composite permite que o subcomponente tenha acesso às demais facetas do componente conectado, via faceta *IComponent* do próprio *proxy* – mais informações na seção 3.3.2.

PUC-Rio - Certificação Digital Nº 1121796/CA

4.3.3 Compartilhamento de Componentes

O SCS-Composite implementou a regra 4 de modo a proibir o compartilhamento de componentes que possuam receptáculos. Na API o teste é feito na função **addSubComponent()** da faceta *IContentController*, que verifica se o componente em questão já faz parte de algum componente composto, por meio da função **getSuperComponents()** da faceta *ISuperComponent*.

Antes do CAS com suporte a componentes compostos, o sistema não exportava a dependência da faceta *Logger* como um receptáculo dos componentes *SpeedCar* e *Space*. Quando atualizamos a versão do CAS, nos deparamos com essa possível melhoria no sistema e optamos por implementá-la. Vislumbramos duas possíveis soluções para adicionar o receptáculo *Logger* nos componentes. A primeira opção seria adicionar a referência do *Logger* diretamente no receptáculo do componente, descrito na figura 4.6 (a). Essa abordagem nos obrigaria a adicionar o *Logger* em cada componente composto que por ventura os utilize. A segunda opção, descrita na figura 4.6 (b), seria exportar a dependência do Logger para o componente composto e conectar o *Logger* apenas uma vez no componente mais externo.

PUC-Rio - Certificação Digital Nº 1121796/CA

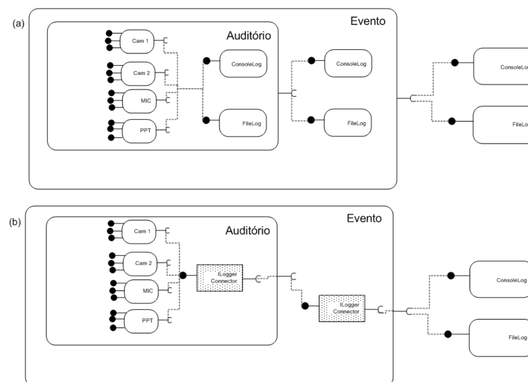

Figura 4.6: (a) Opção com compartilhamento de componentes (b) Opções com conectores e *bind* de receptáculo.

A primeira opção, figura 4.6 (a), não possui o conector *ILogger Connector*, mas em compensação inclui dois componentes *Logger* como subcomponente de cada espaço. Assim como os *SpeedCars*, os espaços também dependem do componente *Logger* para cadastrar seus Logs, por isso o componente *Space* manteve o receptáculo.

A diferença é que na segunda opção o receptáculo do Space é um *bind* vertical (expondo a faceta do conector *ILogger Connector*) e na primeira opção temos um receptáculo do próprio *Space*. Como o compartilhamento de componentes causaria uma complexidade adicional desnecessária, optamos pela segunda opção por ser uma implementação mais clara e fácil de manter.

Mesmo se optássemos pela primeira opção, este cenário não possuiria componentes compartilhados com receptáculos, e desta forma, nós continuaríamos não testando as limitações da regra 4. Acreditamos que essa seja a regra que possa gerar mais restrições de projeto. Porém, ainda não esbarramos em um exemplo com cenários reais onde a regra 4 gere problemas no desenvolvimento do projeto.

4.4
Reconfiguração dos componentes

A criação e organização de todos os componentes na fase de configuração do cenário funcionou de forma satisfatória. Porém, quando pensamos em organizar o ambiente de forma incremental, adicionando espaços a medida que fosse necessário, nos deparamos com um problema de reconfiguração. O cenário descrito neste capítulo possui três ambientes, onde seus subcomponentes foram criados, adicionados e conectados ainda na fase de configuração.

De forma alternativa, podemos pensar na construção do cenário de forma incremental, onde só exista o espaço principal (*Auditório PUC-RIO*) e o CAS adicione novos espaços à medida que seja necessário. O problema dessa abordagem é que a API do SCS-Composite não possui uma forma de criar um componente composto que contenha um conjunto de subcomponentes, i.e. o *middleware* não permite que o usuário crie um espaço externo ao cenário já construído, sendo possível apenas criar um componente que representa o espaço e adicionar os subcomponentes (espaços e conectores) um por um. A abordagem de criar os espaços de forma incremental deixaria a infraestrutura inconsistente durante a fase de reconfiguração, já que em algum momento o cenário teria subcomponentes conectados a componentes externos.

Uma solução neste caso seria colocar todos os componentes envolvidos na reconfiguração em um estado quiescente, onde o sistema permitiria uma inconsistência temporária controlada do ambiente. Libório (2013) e Câmara (2014) estudaram várias técnicas de reconfiguração e atualização dinâmica, e as testaram no SCS. Assim como o conceito de componentes compostos, estes trabalhos poderiam trazer funcionalidades interessantes para o CAS.

A abordagem de criar um estado quiescente levanta algumas dúvidas a respeito de falhas na reconfiguração, por exemplo: Se existir uma inconsistência ao final da reconfiguração, o que o sistema faria? Como tratar o erro? Qual entidade iria identificar o erro? Como informar a todos da inconsistência? Para solucionar tais problemas, Leger (2009) sugeriu uma abordagem transacional para lidar com recuperação de erros e para administrar concorrência na fase de reconfiguração de sistemas baseados em componentes.

PUC-Rio - Certificação Digital Nº 1121796/CA

5
Conclusão

Apresentamos neste trabalho uma análise de um modelo de componentes compostos implementado no SCS. Avaliamos este modelo em um ambiente de Captura e Acesso, adaptando o projeto CAS. A infraestrutura do CAS é inteiramente modelada seguindo o paradigma de componente de software. Consideramos que o CAS foi uma boa escolha para a avaliação do modelo e da implementação desenvolvida neste trabalho, visto que se trata de um sistema em utilização e com necessidades reais de composição.

Escolhemos utilizar o modelo de componentes de software proposto por Medeiros. Ele estudou principalmente os modelos Fractal e OpenCOM e propôs um modelo pouco restritivo, mas que mantivesse os conceitos comuns entre diversos modelos (Crnkovic et al., 2010). Medeiros especificou o modelo em Prolog e fez um estudo experimental em uma versão do SCS.

Na primeira fase da análise, avaliamos os predicados e regras do modelo escolhido e identificamos alguns pontos de melhoria do modelo. Na segunda fase da análise, avaliamos os contratos da API do SCS-Composite - versão experimental desenvolvida no trabalho de Medeiros. Fizemos algumas alterações nos contratos para facilitar a introspecção e tratamento de falhas e analisamos a implementação proposta por Medeiros.

Em nossa adaptação e reimplementação do SCS-Composite, utilizamos o SCS na linguagem Lua. Mesmo com os contratos IDL quase inalterados, a implementação em Lua sofreu grandes mudanças entre a versão utilizada por Medeiros e a versão atual. Nós optamos por implementar o modelo do zero na nova versão, por entender que esta versão é muito melhor estruturada e possui uma API muito mais simples e robusta.

Na terceira fase da análise, avaliamos se a implementação proposta respeitava todos os conceitos impostos pelo modelo. Nesta fase nós descrevemos onde cada regra está sendo verificada no código e discutimos algumas soluções que foram levantadas no desenvolvendo da API. Durante a implementação, nós revimos alguns pontos levantados por Medeiros em seu trabalho, como exemplo, destacamos a regra referente aos *bindings*. Medeiros adicionou a validação da regra dentro dos mecanismos básicos do SCS, mais especificamente

PUC-Rio - Certificação Digital Nº 1121796/CA

na faceta IReceptacles. Essa abordagem era mais intrusiva do que ele gostaria. Para não seguirmos o mesmo caminho, optamos por deixar a implementação do suporte a componente compostos em um módulo separado.

Para analisar a API criada, escolhemos o sistema de Captura e Acesso CAS, implementado utilizando o SCS e que já possuía a necessidade de uso do conceito de componentes compostos. Como o SCS não possuía tais recursos, o grupo de desenvolvimento do CAS foi obrigado a implementar mecanismos que simulassem um componente composto. Para que o CAS utilizasse o SCS-Composite, reimplementamos alguns componentes do próprio modelo CAS, atualizamos o sistema para utilizar a versão mais nova do SCS (já que a última versão mais estável do CAS ainda utilizava uma versão antiga do SCS) e organizamos parte do código para facilitar o desenvolvimento dos testes e do cenário apresentado no trabalho.

O SCS-Composite adicionou novas funcionalidades interessantes no CAS, com destaque para a possiblidade de aninhar espaços de captura e a possibilidade de colocar o componente *Logger* como receptáculos dos componentes *Speedcar* e *Space* - a simulação de componentes compostos do CAS tornava inviável a externalização de receptáculos dos subcomponentes nos componentes compostos. O uso do SCS-Composite também permitiu a remoção das operações e estruturas de dados que simulavam o componente composto, permitindo que os desenvolvedores do CAS se preocupem apenas com a lógica da aplicação.

Após a implementação e os testes, tanto do SCS quanto do CAS, criamos um cenário de Captura e Acesso que simulasse um evento real de uma palestra acontecendo em um auditório e duas salas auxiliares que reproduziam a palestra ao vivo. Este cenário utiliza todos os mecanismos do SCS-Composite e as novas funcionalidades do CAS (dependência do componente *Log* e aninhamento de espaços). A priori, tínhamos o sentimento que as regras do modelo, ou da API, poderiam limitar alguma funcionalidade do cenário. Porém, nossa análise mostrou que a API atendeu de maneira satisfatória durante todo o desenvolvimento.

Por outro lado, quando começamos a pensar na reconfiguração de espaços e componentes, vimos que a API não está preparada para receber atualizações dinâmicas e reconfigurações. Como trabalhos futuros podemos destacar a implementação e teste dos trabalhos de Libório (2013) e Câmara (2014) no CAS com suporte a componentes compostos e a implementação do SCS-Composite em todas as linguagens que o SCS tem suporte – C#, Java e C++.

PUC-Rio - Certificação Digital Nº 1121796/CA

6
Referências Bibliográficas

A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the grid with DeployWare," in *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08)*. IEEE Computer Society, 2008, pp. 177–184. 1

A. Flissi and P. Merle, "A generic deployment framework for grid computing and distributed applications," in *Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06)*, ser. Lecture Notes in Computer Science, vol. 4279. Springer-Verlag, Nov. 2006, pp. 1402–1411. 1

R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf, "An architecture for post-development configuration management in a wide-area network," in *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 269. 1

R. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation," in *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. IEEE Computer Society, 2009, pp. 171–180. 1

P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An architecture-based approach to self-adaptive software," in *Intelligent Systems and their Applications, IEEE (Volume:14 , Issue: 3)*. IEEE Computer Society, May 1999, pp. 54–62. 1

R. F. d. G. Cerqueira, "Um modelo de composição dinâmica entre sistemas de componentes de software," Master's thesis, PUC-Rio, Brasil, 2000. 1

K.-K. Lau, L. Ling, and P. V. Elizondo, "Towards composing software components in both design and deployment phases," in *Proceedings of the 10th International Symposium in Component-Based Software Engineering*, ser.

PUC-Rio - Certificação Digital Nº 1121796/CA

Lecture Notes in Computer Science, H. W. Schmidt, I. Crnkovic, G. T. Heineman, and J. A. Stafford, Eds., vol. 4608. Springer, 2007, pp. 274–282. 1

P. V. Elizondo, M. K., and C. Ndjatchi, "Functional specification of composite components." 1

K.-K. Lau, M. Ornaghi, and Z. Wang, "A software component model and its preliminary formalisation," 2006. 1

I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2010. 1, 5

G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transactions on Computer Systems*, vol. 26, no. 1, pp. 1–42, 2008, http://doi.acm.org/10.1145/1328671.1328672. 1

E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006, http://dx.doi.org/10.1002/spe.v36:11/12. 1, 2.2

R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000. [Online]. Available: http://dx.doi.org/http://dx.doi.org/10.1109/2.825699 1

H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren, "Saveccm - a component model for safety-critical real-time systems," in *EUROMICRO*, 2004, pp. 627–635. 1

A. Medeiros, "Suporte a componentes compostos para o middleware scs," Master's thesis, PUC-Rio, Brasil, 2012. 1, 2.2, 3

C. E. L. Augusto, R. G. Cerqueira, S. Correa, E. Fonseca, L. Marques, and H. Hoenick, "SCS: Software Component System," 2009, http://www.tecgraf.puc-rio.br/ scorrea/scs. 1

OMG, "CORBA Component Model specification," Object Management Group, Tech. Rep., 2006,

PUC-Rio - Certificação Digital Nº 1121796/CA

http://www.omg.org/technology/documents/formal/components.htm. 1, 2.2

D. G. Abowd, G. C. Atkeson, B. Jason, T. Enqvist, P. Gully, and J. LeMon, "Investigating the capture, integration and access problem of ubiquitous computing in an educational setting," 1998. 1, 2.3

M. Lamming and M. Flynn, ""forget-me-not"intimate computing in support of human memory," 1994. 1

F. A. Portella, "Um serviço de captura e acesso para espaços ativos," Master's thesis, PUC-Rio, Brasil, 2008. 1, 2.3

C. E. L. Augusto, "Uma infra-estrutura para a execução distribuída de componentes de software," Master's thesis, PUC-Rio, Brasil, 2008. 2.1

D. Box, *Essential COM (DevelopMentor Series)*. Addison-Wesley Professional, Jan. 1998. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201634465 2.2

R. Brandão, R. G. Cerqueira, F. A. Portella, and A. M. Santos, "The cas project: A general infrastructure for pervasive capture and access systems," 2013. 2.3

T. Lab/PUC-Rio, "Nested context language," http://www.ncl.org.br/. 2.3

Tecgraf/PUC-Rio, "Oil: Object request broker in lua," http://www.tecgraf.puc-rio.br/~maia/oil. 3.3

——, "Loop: Lua object-oriented programming," http://www.tecgraf.puc-rio.br/~maia/oil. 3.3

——, "Openbus: Um middleware para integração de aplicações baseadas em componentes," http://www.tecgraf.puc-rio.br/openbus. 4.2

A. J. A. Libório, "Suporte à evolução arquitetural de sistemas distribuídos baseados em componentes de software," Master's thesis, PUC-Rio, Brasil, 2013. 4.4, 5

E. C. M. Câmara, "Um estudo sobre atualização dinâmica de componentes de software," Master's thesis, PUC-Rio, Brasil, 2014. 4.4, 5

M. Leger, "Fiabilité des reconfiguration dynamiques dans les architecture a composants," Master's thesis, PUC-Rio, Brasil, 2009. 4.4

PUC-Rio - Certificação Digital Nº 1121796/CA

A
Interface IDL do SCS-Composite

PUC-Rio - Certificação Digital Nº 1121796/CA

```
 1 #ifndef SCS_MEBRANE
 2 #define SCS_MEMBRANE
 3
 4 #include "scs.idl"
 5
 6 module scs {
 7   module composite{
 8
 9     /** \brief O identificador que representa o componente no
         componente composto. */
10     typedef unsigned long MembershipId;
11     /** \brief Uma lista de MembershipIds. */
12     typedef sequence<MembershipId> MembershipIdSeq;
13     /** \brief O identificador que representa o binding do
         componente no componente composto. */
14     typedef unsigned long BindingId;
15     /** \brief Uma lista de IComponent. */
16     typedef sequence<scs::core::IComponent> IComponentSeq;
17
18     /** \brief O componente nao foi construido corretamente. */
19     exception InvalidComponent{};
20     /** \brief O componente possui receptaculo e ja faz parte de
         um componente composto. */
21     exception UnshareableComponent{};
22     /** \brief O Componente nao foi encontrado. */
23     exception ComponentNotFound{
24       MembershipId id; /**< \brief O Id que representa o
           componente. */
25     };
26     /** \brief A faceta nao foi encontrada. */
27     exception FacetNotFound{};
28     /** \brief O componente ja possui a faceta definida. */
29     exception FacetAlreadyExists{};
30     /** \brief A interface nao e incompativel com as interface de
         entrada.*/
31     exception IncompatibleInterfaces{};
32     /** \brief O receptaculo nao foi encontrado.*/
33     exception ReceptacleNotFound{};
```

```
34    /** \brief O componente ja possui o receptaculo definido.*/
35    exception ReceptacleAlreadyExists{};
36    /** \brief Falha interna do componente.*/
37    exception ComponentFailure{
38      string msg; /**< \brief Mensagem com o detalhamento do erro
          .*/
39    };
40
41    /** \brief Estrutura de dados que representa um binding.*/
42    struct BindingInformation {
43      MembershipId id; /**< \brief Identificador do subcomponente
          .*/
44      string name; /**< \brief Nome da entidade do subcomponente
          que sera conectada.*/
45    };
46    typedef sequence<BindingInformation> BindingInformationSeq;
        /**< \brief Conjunto de BindingInformation */
47
48    /** \brief Descrição de conexoes.*/
49    struct BindingDescription{
50      BindingId id; /**< \brief Identificador da conexao entre o
          subcomponente e o componente composto. */
51      string name; /**< \brief Nome da faceta ou receptáculo */
52      boolean isFacet; /**< \brief True caso a conexao seja uma
          faceta, Falso caso seja um receptaculo */
53    };
54    typedef sequence<BindingDescription> BindingDescriptionSeq;
        /**< \brief Conjunto de BindingDescription */
55
56    /** \brief Descrição de subcomponentes.*/
57    struct MembershipDescription{
58      scs::core::IComponent icomponent; /**< \brief Um
          subcomponente do componente composto. */
59      MembershipId id; /**< \brief Identificador do subcomponente
          .*/
60    };
61    typedef sequence<MembershipDescription>
        MembershipDescriptionSeq;
62
63    enum Permission {
64      CURRENT,
65      ALL
66    };
67
68    /**
69    * \brief Interface que gerencia o componente composto,
        resposavel por: adicionar e remover subcomponentes;
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
         realizar e desfazer os bindings; inspecionar subcomponentes
         .
70     */
71     interface IContentController {
72
73       /**
74        * \brief Fornece o identificador do componente composto.
75        *
76        * \return O identificador do componente composto.
77        */
78       string getId();
79
80       /**
81        * \brief Adiciona um subcomponente no componente composto.
82        *
83        * \param [in] icomponent O componente que sera adicionado no
              componente composto
84        *
85        * \return O identificador que representa o subcomponente no
             componente composto
86        *
87        * \exception InvalidComponent Caso o componente nao possua a
              faceta ISuperComponent necessaria.
88        *
89        * \exception ComponentFailure Falha na operacao
90        *
91        * \exception UnshareableComponent Caso o componente possua
             receptaculo e ja seja um subcomponente de um componente
             composto.
92        */
93       MembershipId addSubComponent(in scs::core::IComponent
             icomponent) raises(InvalidComponent, ComponentFailure,
             UnshareableComponent);
94
95       /**
96        * \brief Remove um subcomponente do componente composto.
97        *
98        * \param [in] id O identificador que representa o
             subcomponente
99        *
100       * \return True caso o componente seja removido com sucesso,
             False caso contrario
101       *
102       * \exception
103       */
104      boolean removeSubComponent(in MembershipId id);
105
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
106         /**
107         * \brief Fornece todos os subcomponentes adicionados no
                componente composto.
108         *
109         * \return Uma lista contendo todos os Ids e referencias para
                os subcomponentes
110         */
111         MembershipDescriptionSeq getSubComponents();
112
113         /**
114         * \brief Fornece todas as conexoes entre os subcomponentes e
                o componente composto
115         *
116         * \return Uma lista contendo todos os Ids e referencias
117         */
118         BindingDescriptionSeq retrieveBindings();
119
120         /**
121         * \brief Busca subcomponentes utilizando o identificador do
                mesmo no componente composto.
122         *
123         * \param [in] id O identificador que representa o
                subcomponente
124         *
125         * \return A referencia para o subcomponente
126         *
127         * \exception ComponentNotFound Caso id nao seja valido.
128         *
129         * \exception ComponentFailure Falha na operacao
130         */
131         scs::core::IComponent findComponent(in MembershipId id)
                raises(ComponentNotFound, ComponentFailure);
132
133         /**
134         * \brief Cria uma conexao de uma faceta do conector
135         *
136         * \param [in] connector O identficador do conector no
                componente composto.
137         *
138         * \param [in] internalFacetName O nome da faceta do conector
                que sera exportada.
139         *
140         * \param [in] externalFacetName O nome da nova faceta do
                componente composto.
141         *
142         * \return O identificador da conexao.
143         *
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
144        * \exception ComponentNotFound Caso id nao seja valido.
145        *
146        * \exception InvalidComponent Caso o componente nao
               implemente uma operacao corretamente
147        *
148        * \exception FacetNotFound Caso a faceta do conector nao
               seja encontrada
149        *
150        * \exception FacetAlreadyExists Caso ja exista uma faceta
               com o mesmo nome no subcomponente
151        *
152        * \exception ComponentFailure Falha na operacao
153        */
154        BindingId bindFacet(in MembershipId connector, in string
               internalFacetName, in string externalFacetName)
155            raises(ComponentNotFound, InvalidComponent,
                   FacetNotFound, FacetAlreadyExists, ComponentFailure);
156
157        /**
158        * \brief Cria uma conexao de um receptaculo do conector
159        *
160        * \param [in] connector O identficador do conector no
               componente composto.
161        *
162        * \param [in] internalReceptacleName O nome do receptaculo
               do conector que sera exportado.
163        *
164        * \param [in] externalReceptacleName O nome do novo
               receptaculo do componente composto.
165        *
166        * \param [in] componentPermission A permissão de
               visibilidade do subcompente para com os componentes
               conectados ao bind.
167        *
168        * \return O identificador da conexao.
169        *
170        * \exception ComponentNotFound Caso id nao seja valido.
171        *
172        * \exception InvalidComponent Caso o componente nao possua
               uma operacao corretamente
173        *
174        * \exception ReceptacleAlreadyExists Caso ja exista um
               receptaculo com o mesmo nome no subcomponente
175        *
176        * \exception ComponentFailure Falha na operacao
177        */
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
178        BindingId bindReceptacle(in MembershipId connector, in
               string internalReceptacleName, in string
               externalReceptacleName, in Permission componentPermission
               )
179             raises(ComponentNotFound, InvalidComponent,
                    ReceptacleNotFound, ReceptacleAlreadyExists,
                    ComponentFailure);
180
181        /**
182        * \brief Desfaz uma conexao criada
183        *
184        * \param [in] id Identificado da conexao
185        *
186        * \return True caso a conexao tenha sido desfeita com
               sucesso, False caso contrario.
187        */
188        void unbind(in BindingId id);
189
190     }; interface ISuperComponent {
191
192        /**
193        * \brief Adiciona a referencia do componente composto pai ao
                subcomponente. Essa operacao deve ser chamada de forma
               automatica pela operacao addSubCompoent da faceta
               IContentController.
194        *
195        * \param [in] icomponent O componente que sera adicionado no
                componente composto
196        *
197        * \exception InvalidComponent Caso o componente nao
               implemente uma operacao corretamente
198        */
199        void addSuperComponent(in scs::core::IComponent icomponent)
               raises(InvalidComponent);
200
201        /**
202        * \brief Remove a referencia do componente composto.
203        *
204        * \param [in] icomponent O componente que sera adicionado no
                componente composto
205        *
206        * \return True caso o componente composto tenha sido
               removido, False cason contrario.
207        *
208        */
209        boolean removeSuperComponent(in scs::core::IComponent
               iComponent);
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
210
211        /**
212        * \brief Fornece os componentes compostos no qual o
              subcomponente em questao faz parte.
213        *
214        * \return Uma lista com todas as referencias para os
              componentes compostos.
215        */
216        IComponentSeq getSuperComponents();
217      };
218    };
219 };
220
221 #endif
```

Código A.1: Interface IDL do SCS-Composite.

PUC-Rio - Certificação Digital Nº 1121796/CA

B
Exemplo de uso do SCS-Composite

O código B.1 representa um exemplo de uso do SCS-Composite em um cenário de vigilância de uma casa - representada pela figura 2.2. As linhas 8 a 15 descrevem como que os conectores são instanciados e adicionados no componente composto (linhas 20 e 21). As linhas 24 e 25 descrevem como que uma das câmeras é adicionada ao conector e como as facetas dos conectores são exportadas para o componente composto (linhas 29 e 30). Por fim, as linhas 33 a 40 mostram como que a faceta pode ser utilizada como uma faceta do próprio componente composto.

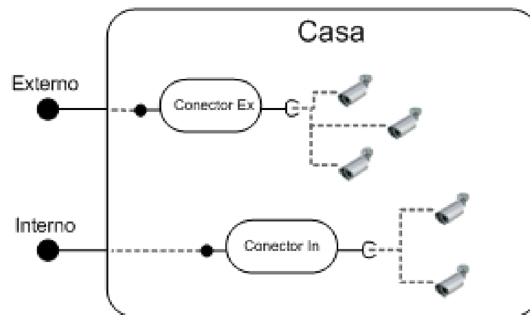

Figura 2.2: Exemplo de *binding vertical* de facetas de dois conectores (reprodução da imagem do Capítulo 2).

```
 1 ...
 2 local orb = oil.init({localrefs = "proxy"})
 3
 4 oil.main(function()
 5   ...
 6
 7 --- 4.3 Criar dois conectores
 8   local connectorComptId = { name = "RecordConnector",
        major_version = 1, minor_version = 0, patch_version = 0,
        platform_spec = "" }
 9   local conectorExterno = ComponentContext(orb, connectorComptId)
10   conectorExterno:addFacet("IRecord", "IDL:IRecord:1.0",
        IRecordConnector())
11   conectorExterno:addReceptacle("IRecord", "IDL:IRecord:1.0", true
        )
12
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
13   local conectorInterno = ComponentContext(orb, connectorComptId)
14   conectorInterno:addFacet("IRecord", "IDL:IRecord:1.0",
         IRecordConnector())
15   conectorInterno:addReceptacle("IRecord", "IDL:IRecord:1.0", true
         )
16
17 ——— 4.4 Adicionar o componentes na casa
18   houseContentController:addSubComponent(cameraEx1)
19   ...
20   local connExId = houseContentController:addSubComponent(
         conectorExterno.IComponent)
21   local connInId = houseContentController:addSubComponent(
         conectorInterno.IComponent)
22
23 ——— 4.5 Conecta as cameras nos conecteores
24   local recordFacet = orb:narrow(cameraEx1:getFacetByName("IRecord
         "), "IDL:IRecord:1.0")
25   receptacle:connect("IRecord", recordFacet)
26   ...
27
28 ——— 4.6 Faz o bind das Facetas dos conectores
29   houseContentController:bindFacet(connExId, "IRecord", "IRecordEx
         ")
30   houseContentController:bindFacet(connInId, "IRecord", "IRecordIn
         ")
31
32 ——— 4.7 Inicia a gravação das cameras externas, para a gravação
       das cameras externas e inicia a gravação das cameras internas
33   local externalCameras = houseIComponent:getFacetByName("
         IRecordEx")
34   externalCameras = orb:narrow(externalCameras, "IDL:IRecord:1.0")
35   local internalCameras = houseIComponent:getFacetByName("
         IRecordIn")
36   internalCameras = orb:narrow(internalCameras, "IDL:IRecord:1.0")
37
38   externalCameras:record()
39   externalCameras:stop()
40   internalCameras:record()
41 end)
```

Código B.1: Fragmento do código Lua do cenário da figura 2.2.

PUC-Rio - Certificação Digital Nº 1121796/CA

C
Exemplo de testes de Prolog

O código C.1 representa um exemplo dos testes feitos em Prolog. Neste caso estávamos interessados em testar as regras 1 e 2, que descrevem se um *binding* horizontal é válido, isto é, se é possível conectar uma faceta a um receptáculo. Para isso foi necessário primeiramente definir diferentes predicados, são eles: interfaces; compatibilidades entre interfaces; identificadores; componentes; facetas; receptáculos; conexões. Criamos quatro conexões, sendo duas em um receptáculo múltiplo e outras duas em um receptáculo simples. O teste verifica que um receptáculo simples é inválido quando existe mais de uma faceta conectado a ele, por outro lado um receptáculo múltiplo aceita múltiplas conexões simultaneamente.

```
1 interface(interfaceA).
2 interface(interfaceB).
3 interface(interfaceC).
4 compatible(interfaceA, interfaceB).
5 compatible(interfaceA, interfaceA).
6 compatible(interfaceB, interfaceB).
7 compatible(interfaceC, interfaceC).
8
9 identifier(fA).
10 identifier(r1).
11 identifier(r2).
12 identifier(r3).
13 identifier(fC).
14 identifier(rC).
15
16 component(compA).
17 component(compB).
18 component(compC).
19
20 facet(fC, interfaceB, compC).
21 facet(fA, interfaceB, compA).
22 receptacle(r1, interfaceB, multiple, compB).
23 receptacle(r2, interfaceB, simple, compB).
24 receptacle(r3, interfaceB, simple, compB).
25
26 connection(compA, fA, compB, r1).
```

PUC-Rio - Certificação Digital Nº 1121796/CA

```
27 connection(compC, fC, compB, r1).
28
29 connection(compC, fC, compB, r3).
30
31 connection(compC, fC, compB, r2).
32 connection(compA, fA, compB, r2).
33
34
35 isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1), receptacle(R
      , T2, multiple, C2), compatible(T1, T2).
36 isAValidConnection(C1, F, C2, R) :- facet(F, T1, C1), receptacle(R
      , T2, simple, C2), compatible(T1, T2), connection(C3, Fq, C2,
      R) -> (C3=C1), (Fq = F).
```

Código C.1: Teste em Prolog.

```
 1 1 ?- isAValidConnection(compA, fA, compB, r1).
 2 true.
 3
 4 2 ?- isAValidConnection(compC, fC, compB, r1).
 5 true .
 6
 7 3 ?- isAValidConnection(compC, fC, compB, r2).
 8 true.
 9
10 4 ?- isAValidConnection(compA, fA, compB, r2).
11 false.
```

Código C.2: Consultas feita em Prolog.

Nós carregamos o código C.1 no interpretador SWI-Prolog e fizemos as consultas descritas no código C.2. As consultas (1) e (2) testaram a regra 1, mostrando que o receptáculo composto **r1** do componente **compB** aceita mais de uma conexão simultaneamente.

Em seguida, as consultas (3) e (4) testaram a regra 2, mostrando que o receptáculo simples **r2** do componente **compB** não aceita mais de uma conexão simultaneamente. Note que a consulta (3) retornou *true*, visto que é a primeira conexão do receptáculo **r2**. A consulta (4) retornou *false*, visto que o receptáculo **r2** já possui uma conexão.

PUC-Rio - Certificação Digital Nº 1121796/CA