Airton José Araújo Libório

Suporte à Evolução Arquitetural de Sistemas Distribuídos Baseados em Componentes de Software

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Programa de Pós–graduação em Informática

Rio de Janeiro
Maio de 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Airton José Araújo Libório

Suporte à Evolução Arquitetural de Sistemas Distribuídos Baseados em Componentes de Software

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. Renato Fontoura de Gusmão Cerqueira

Rio de Janeiro
Maio de 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Airton José Araújo Libório

Suporte à Evolução Arquitetural de Sistemas Distribuídos Baseados em Componentes de Software

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 de La Rocque Rodriguez
Presidente
Departamento de Informática — PUC-Rio

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

Prof. Antônio Tadeu Azevedo Gomes
Coordenação de Sistemas e Redes — LNCC

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

Rio de Janeiro, 16 de Maio de 2013

Airton José Araújo Libório

Graduou-se em Bacharelado em Ciência da Computação pela UFBA. Foi pesquisador do laboratório Tecnologia em Computação Gráfica (TecGraf) da PUC-Rio entre os anos de 2010 e 2011, onde atuou num projeto em parceria com a Petrobras S/A, no desenvolvimento de um sistema de captura e acesso de mídias distribuídas, o CAS.

Ficha Catalográfica

Libório, Airton José Araújo

Suporte à evolução arquitetural de sistemas distribuídos baseados em componentes de software / Airton José Araújo Libório; orientador: Renato Fontoura de Gusmão Cerqueira. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2013.

v., 95 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. Evolução Dinâmica de Software. 3. Sistemas Dinamicamente Adaptáveis. 4. Adaptação de Software. 5. Componentes de Software. 6. Arquitetura de Software. I. Cerqueira, Renato Fontoura de Gusmão. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Em primeiro lugar gostaria de agradecer ao professor Renato Cerqueira pela confiança em mim depositada, pela orientação e segurança que me passou nesses anos de mestrado e pela vital contribuição na construção e desenvolvimento deste trabalho. Agradeço também à PUC-Rio pela infraestrutura e oportunidade de estudo e ao TecGraf pela experiência de trabalhar com pessoas extremamente capacitadas.

Gostaria de agradecer à minha família, em especial aos meus pais Airton e Célia, que desde cedo me proporcionaram a oportunidade, o incentivo e o acesso a um estudo de qualidade, o qual poucos podem ter neste país. Aos avós, tios, primos, irmãos, que contribuíram para a formação do meu caráter e alavancaram o meu potencial fazendo com que eu esteja aqui hoje. Tirando a influência destas pessoas sobre a minha vida, pouco sobra de mim.

Gostaria de agradecer aos meus amigos de Salvador, que por todos estes anos me deram apoio e me proporcionaram os momentos mais ímpares que já vivi. Difícil mesurar o quão difícil é suportar a distância, mas me prendo na alegria de revê-los a cada vez que os visito.

Agradeço também aos amigos que fiz aqui no Rio de Janeiro, muitos os quais, como eu, deixaram os seus estados em busca de estudar e aprender um pouco mais. Agradeço especialmente aos amigos Amadeu, Pablo e Adriano, pela ajuda no desenvolvimento deste trabalho.

Deixo registrada aqui também a minha gratidão por todos que contribuíram, mesmo que indiretamente para este trabalho, seja na formação de idéias ou mesmo por uma palavra ou gesto de incentivo.

Todos vocês foram parte e fizeram uma diferença enorme para mim. Muito obrigado!

Resumo

Libório, Airton José Araújo; Cerqueira, Renato Fontoura de Gusmão. **Suporte à Evolução Arquitetural de Sistemas Distribuídos Baseados em Componentes de Software**. Rio de Janeiro, 2013. 95p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A natureza de certos sistemas de software determina que estes tenham de executar de maneira ininterrupta. Por outro lado, diversos sistemas de software são constantemente sujeitos a mudanças, por questões que incluem, mas não se limitam a, infraestrutura, correções de falhas, adição de funcionalidades e mudanças na lógica de domínio. Evolução dinâmica de software consiste em alterar aplicações durante a sua execução sem interrompê-las, mantendo-as disponíveis mesmo durante a aplicação destas modificações. Sistemas distribuídos baseados em componentes permitem decompor o software em entidades claramente separadas. Nesses casos, a evolução pode ser resumida a remoção, adição e modificação de tais entidades, e se tais atividades podem ser exercidas enquanto a aplicação está em execução, tem-se evolução dinâmica de software. Através disso, neste trabalho foi criada uma abordagem em que é possível se manipular arquiteturas distribuídas desenvolvidas sobre o middleware SCS de maneira a se minimizar a interrupção de partes do sistema enquanto certas adaptações são implantadas. Aplicamos o mecanismo em um sistema distribuído já consolidado, o CAS, que consiste em uma infraestrutura de gravação extensível com suporte a captura e acesso automáticos de mídias distribuídas.

Palavras–chave

Evolução Dinâmica de Software; Sistemas Dinamicamente Adaptáveis; Adaptação de Software; Componentes de Software; Arquitetura de Software;

Abstract

Libório, Airton José Araújo; Cerqueira, Renato Fontoura de Gusmão (Advisor). **Support for Architectural Evolution in Component-based Distributed Systems**. Rio de Janeiro, 2013. 95p. MSc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The nature of some software systems determine that they run without interruption. Furthermore, many software systems are constantly subject to change for reasons that include, but are not limited to, infrastructure changes, bug fixes, addition of functionalities, and changes in the domain logic. Dynamic software evolution consists into changing application during execution without stopping them, keeping them available even when applying these modifications. Component-based distributed systems allows decomposing software into clearly separated entities. In such cases, evolution can be summarized to removal, addition and modification of such entities, and if such activities can be performed while the application is executing, dynamic adaptation is achieved. In this work, we've investigated an approach that aims to allow manipulation of distributed software architectures developed over the SCS middleware, in order to minimize system disruption while certain adaptations are deployed. The mechanism was tested in an already consolidated distributed system, the CAS, which consists of an extensible recording infrastructure that supports automatic capture and access of distributed medias.

Keywords

Dynamic Software Evolution; Dynamically Adaptive Systems; Software Adaptation; Software Components; Software Architecture;

Sumário

Lista de figuras

Vou mostrando como sou E vou sendo como posso, Jogando meu corpo no mundo, Andando por todos os cantos E pela lei natural dos encontros Eu deixo e recebo um tanto E passo aos olhos nus Ou vestidos de lunetas, Passado, presente, Participo sendo o mistério do planeta

Luiz Galvão e Moraes Moreira - Novos Baianos

1
Introdução

Sistemas computacionais que suportam evolução dinâmica tem a capacidade de modificar a sua implementação em tempo de execução, permitindo que seus serviços sejam estendidos, customizados, corrigidos ou melhorados, sem a necessidade de recompilação e/ou reinicialização. Alterar tal comportamento de uma aplicação não é uma tarefa fácil, e abordagens mais tradicionais como desligamentos agendados, uso de redundância, e substituições manuais foram concebidas com o intuito de contornar tal necessidade. Contudo, a natureza de certos sistemas de software determina que estes tenham de executar de maneira ininterrupta após a sua implantação. Usualmente suas execuções são intrisicamente associadas a interesses de natureza crítica (sistemas financeiros, sistemas de suporte a vida, software de segurança e aplicações militares).

Em sistemas de larga escala, antes de aplicar mudanças é essencial entender de maneira precisa o estado corrente de cada parte do sistema afetado. A construção de tais sistemas demanda decomposição em estruturas lógicas bem definidas (componentes) [2]. Componentes de software são "*unidades de composição com interfaces contratualmente especificadas e dependências de contexto explícitas, que podem ser implantados de maneira independente e estão sujeitos a composição por terceiros*" [3]. O uso de sistemas distribuídos baseados em componentes permite decompor o software em entidades claramente separadas, facilitando o desenvolvimento de aplicações complexas pelo desacoplamento de interfaces de implementações e explicitação das partes da arquitetura como um todo. Neste contexto, um sistema distribuído pode ser visto como uma coleção de componentes interagindo em uma rede através de portas de requisição (receptáculos) e provisão (facetas) de serviços. Por isolar interesses diversos, componentes, quando tratados como estruturas de execução, definem uma unidade básica de reconfiguração [4]. Nestes casos, a evolução pode ser resumida a remoção, adição e modificação de tais entidades, e se tais atividades podem ser exercidas enquanto a aplicação está em execução, tem-se evolução dinâmica de software [5].

Um problema importante a ser tratado na engenharia de sistemas baseados em componentes é o uso de notações para descrevê-los de maneira

apropriada. Uma abordagem inicial consiste em utilizar recursos inerentes a linguagens de programação, tais quais classes (representando componentes), interfaces (representando serviços) e associações (interações entre componentes). Contudo, a utilização exclusiva destes mecanismos não é suficiente para se representar algumas construções importantes, como:

- Ligações (do inglês *bindings*) representadas por invocação de métodos, impossibilitando a visualização destes como elementos de design de primeira ordem;
- Descrições hierárquicas a nível de componentes compostos;
- Suporte a descrição de famílias de sistemas, em termos de sintaxe explícita para design de restrições individuais de cada aplicação;
- Propriedades de implantação distribuída;
- Ausência de suporte a explicitação de propriedades não-funcionais.

Linguagens de descrição de arquitetura (*ADL*s, do inglês *Architecture Description Languages*) são notações para se expressar e representar projetos, designs e estilos arquiteturais, descrevendo a estrutura do sistema em alto nível. São usadas para especificar formalmente, desde a concepção até a evolução, os princípios que regem estrutura (componentes e conexões) e comportamento (interações) [6]. Separam ainda as atividades de programação de componentes e configuração de sistemas, tornando-os mais gerenciáveis, modulares e reutilizáveis [7, 2, 8]. *ADL*s bem construídas trazem um melhor entendimento acerca de certas propriedades do sistema em alto nível, como composição, interação, implantação e conformidade de padrões, e podem ser utilizadas para se aliviar estas (e outras) dificuldades [9, 10, 11, 12, 13, 14, 15].

O padrão IEEE 1471-2000 [16], que também tornou-se ISO/IEC 42010:2007, recomenda a provisão de descrições arquiteturais durante todo o ciclo de vida do software, de maneira a mitigar os riscos incorridos na construção, manutenção e evolução desses sistemas. Assim, somente a explicitação da arquitetura inicial da aplicação não é suficiente: deve também acompanhar de forma acurada os artefatos do software, de maneira que mudanças no design sejam imediamente refletidas na implementação, e vice-versa. Uma vez que determinados níveis de qualidade precisam ser atingidos em aspectos como tolerância a falhas, segurança, escalabilidade, desempenho e corretude, é vital que todas as partes do software evoluam em conformidade [17]. De maneira a garantir o cumprimento constante de requisitos de naturezas diversas, são necessários instrumentos que permitam aos desenvolvedores lidar com a evolução de tais sistemas em tempo de execução. Neste sentido, diversas

linguagens de descrição de arquiteturas foram desenvolvidas, porém somente algumas fornecem mecanismos que favoreçam evolução dinâmica [18].

1.1 Objetivos e Contribuições

Este trabalho tem como objetivo o desenvolvimento de uma ferramenta de especificação arquitetural, com mecanismos de descrição para implantação e evolução dinâmica de arquiteturas distribuídas desenvolvidas sobre o modelo de componentes SCS [19]. A ferramenta suporta componentes de software desenvolvidos na linguagem Lua [20], que possui capacidades de extensibilidade, portabilidade, além de reflexão, estando também inserida no contexto de pesquisa do grupo. O trabalho [1] fornece uma abstração para composição de componentes, com mecanismos de introspecção e configuração sobre componentes internos. Tal abordagem é reconhecidamente uma prática favorável à abstração de implementação de estruturas complexas e especificação de sistemas distribuídos baseados em componentes de software. A ferramenta desenvolvida no presente trabalho permite especificar um sistema através de um componente composto, com suas partes, ligações e interações internas. Através do uso deste mecanismo, os componentes só precisam se preocupar em implementar as suas interfaces, deixando a lógica de composição para a descrição arquitetural, respeitando o princípio da sepração de interesses. As principais funcionalidades suportadas são:

Especificação de arquiteturas de sistemas distribuídos : Descrição/instanciação/manutenção de aplicações distribuídas através de reflexão sobre a arquitetura do sistema. Esta funcionalidade pode ser dividida em:

Especificação de estilos arquiteturais Permite definir, de maneira abstrata, os tipos de componentes, os serviços e os relacionamentos que compõem a arquitetura de um sistema distribuído de maneira reutilizável;

Implantação e configuração automáticos Através do estilo arquitetural, é possível definir instâncias de componentes e conexões a serem executados de maneira distribuída. A ferramenta é encarregada de implantar e conectar devidamente as instâncias em um componente composto, assim como verificar o cumprimento de restrições arquiteturais, aliviando a dificuldade incorrida na configuração manual do sistema.

Suporte a evolução dinâmica de software : São fornecidos mecanismos de adaptação dinâmica por meio da configuração da arquitetura do sistema. Estas alterações seguem uma abordagem programática e podem ser aplicadas de maneira remota. Dentre as possíveis operações, temos:

Manipulação de instâncias Permite que componentes e conexões sejam manipulados durante o ciclo de vida do sistema, inclusive em tempo de execução. Componentes podem ser incorporados ou retirados da arquitetura, com verificações de restrições arquiteturais pré-definidas;

Substituição/Alteração de componentes As unidades de execução podem ser completamente substituídas ou parcialmente alteradas. Substituir um componente implica em trocar a implementação dos seus serviços fornecidos, enquanto que alterá-lo incorre em alterar a implementação de um subconjunto dos seus serviços;

Evoluções como elementos de construção de primeira classe A evolução do sistema de software pode ser feita através de elementos que permitem expressar e capturar claramente mudanças arquiteturais durante todo o ciclo de vida do software. A partir do uso da ferramenta para conduzir tais atividades de evolução, histórico da arquitetura do software torna-se rastreável ao longo do ciclo de vida. Além disso, faz com que as mudanças estejam atreladas à arquitetura, criando uma maneira mais gerenciável de lidar com mudanças [21, 18];

A partir do levantamento das tecnologias de evolução dinâmica de arquiteturas distribuídas existentes, as contribuições esperadas deste trabalho são as seguintes:

- De maneira a evitar erros durante as adaptações foi criado um mecanismo de suporte ao controle de estados de componentes. Desta forma, a partir do controle do fluxo de requisições, é possível manter a consistência dos componentes mesmo durante a aplicação de mudanças;
- Desenvolvimento de uma ferramenta sobre o SCS [19] que contemple funcionalidades de descrição, instanciação, reflexão e evolução arquitetural;
- Suporte a evoluções de tipos e estilos arquiteturais, não encontradas na literatura;
- Suporte a adaptações de software arbitrárias, com evolução dinâmica de instâncias distribuídas de componentes em tempo de execução.

Como prova de conceito utilizamos um sistema distribuído já consolidado, o **CAS** (*Capture and Access System*) [22], que consiste em uma infraestrutura de gravação extensível com suporte a captura e acesso automáticos de mídias distribuídas. Este sistema apresenta diversos cenários ideais para a demonstração dos mecanismos desenvolvidos. O CAS já possui uma implementação que utiliza componentes compostos [1], e neste trabalho estendemos a idéia desta implementação para acomodar descrição arquitetural, instanciação e conexão automática de componentes, assim como aplicação de adaptações. No nosso exemplo é demonstrado como a ferramenta pode propiciar a inicialização distribuída automática, alterações na arquitetura e correção de defeitos em tempo de execução em partes isoladas.

1.2 Estrutura do Documento

Este documento está organizado da seguinte maneira. O capítulo 2 apresenta os conceitos básicos inerentes a arquiteturas e evolução dinâmica de software. O capítulo 3 faz um apanhado de trabalhos relacionados a este, com descrição de como outras abordagens implementam cada conceito do capítulo anterior. No capítulo 4 é apresentado o funcionamento da ferramenta **SCS-DynAdapt**, seus componentes internos, interface de programação e mecanismos de especificação e adaptação dos componentes da arquitetura. O capítulo 5 demonstra um exemplo de uso da ferramenta, através da aplicação de adaptações em um sistema distribuído. Por fim, o capítulo 6 apresenta as considerações finais e trabalhos futuros pertinentes à ferramenta.

2
Conceitos Básicos

Antes de descrever as abordagens utilizadas por trabalhos relacionados a este, é interessante fornecer definições acerca dos conceitos básicos pertinentes a esta dissertação. O objetivo deste capítulo é elucidar de forma detalhada a composição de cada um destes conceitos. Na seção 2.1 são apresentados conceitos básicos e definições aceitas de arquiteturas de software. A seção 2.2 descreve elementos comuns à composição das *ADL*s. A seção 2.3 traz à tona evolução dinâmica de software, com classificações encontradas na literatura dos variados tipos de mecanismos, seguida da seção 2.4 que mostra então a aplicação destes mecanismos a arquiteturas de software. A seção 2.5 mostra o funcionamento do middleware SCS e sua extensão de suporte a componentes compostos, o SCS-Composite.

2.1
Arquitetura de Software

Arquitetura de software é uma abordagem que fornece técnicas para descrição estrutural (unidades principais e sua organização) que refletem a base lógica do design de sistemas. É reconhecidamente uma questão central no ciclo de vida do software, dado que a sua ausência traz dificuldades nas atividades de compreensão, criação, customização e evolução do software. É vital na medida em que são documentadas as razões que guiaram cada aspecto de construção (elementos arquiteturais, interações, restrições), ao mesmo tempo trazendo abstrações de alto nível, fazendo que sistemas se tornem mais simples e compreensíveis [6]. Uma definição bem aceita para arquiteturas de software é dada por Perry e Wolf [23]

> Arquitetura de Software = { Elementos, Forma, Lógica }
>
> Ou seja, arquitetura de software é um conjunto de elementos arquiteturais (ou de design) com uma determinada forma.
>
> **Elementos** captam as unidades do sistema, que podem ser de três tipos: de processamento, de dados e de conexão.
> **Forma** capta como os elementos estão organizados, por meio de

> propriedades e relações ponderadas. Isto é, a forma captura como os elementos estão compostos (i.e. configuração), as características das interações, e seus relacionamentos com o ambiente operacional. **Lógica** capta a motivação para a escolha de um estilo arquitetural, a escolha de elementos, e a forma. Isto é, a intenção do designer do sistema, suas hipóteses, escolhas, restrições externas, padrões de design escolhidos, e outras informações não facilmente observáveis a partir da arquitetura (subjacentes).

Segundo Medvidovic e Taylor [24], **Elementos** ajudam a responder perguntas do tipo: "*O que são os elementos de um sistema?*", "*Quais são os propósitos primários e os serviços que estes provêem?*". **Forma** ajuda a responder: "*Como estão os elementos compostos de maneira a se cumprir a tarefa principal do sistema?*", "*Como os elementos estão distribuídos?*". Já a **Lógica**: "*Por que certos elementos em particular são utilizados?*", "*Por que são combinados de determinada maneira?*", "*Por que o sistema está distribuído de determinada forma?*". Uma definição mais recente (e também bem aceita) é dada por Taylor, Medvidovic e Dashofy [24]:

> A arquitetura de um sistema de software é o conjunto das principais decisões de projeto feitas sobre o sistema. Decisões de projeto abrangem cada aspecto do sistema em desenvolvimento, incluindo: estrutura do sistema, comportamento funcional, interação, propriedades não-funcionais e de implementação.

É notável que apesar de existirem diferentes definições na literatura, a maioria cita estrutura e comportamento. *Estrutura* descreve como o sistema está interconetado internamente em unidades denominadas componentes. *Comportamento* é definido como o conjunto de interações para se obter a funcionalidade total do sistema. Esses dois aspectos são especificados formalmente através de uma linguagem de descrição de arquitetura (*ADL*, do inglês *Architecture Description Language*), descrita a seguir.

2.2 *ADL* (*Architecture Description Language*)

Uma *ADL* pode ser descrita como uma linguagem que provê características para modelar a arquitetura conceitual de sistemas de software, dissociada da sua implementação. *ADL*s provêem tanto uma sintaxe concreta quanto um *framework* conceitual. O framework conceitual tipicamente reflete as particularidades do domínio para o qual a *ADL* é destinada e/ou o estilo arquitetural.

Os mecanismos da sintaxe provêem especificação ativa por restringir o espaço de possíveis decisões de design, reduzindo a carga cognitiva imposta sobre o arquiteto de software [25].

Diversas *ADL*s têm sido propostas com o intuito de prover notações formais (ou semi-formais) de modelagem e ferramentas adequadas de análise e desenvolvimento orientado a arquiteturas. O foco aqui é em estruturas de alto nível da aplicação, deixando à parte detalhes de implementação [25]. Ainda não há, contudo, muito consenso acerca do que realmente constitui uma *ADL* e de qual o nível de suporte esta deve prover aos desenvolvedores. Apesar disso, a taxonomia criada em [25] destaca que *ADL*s devem explicitamente prover modelos para *componentes*, *conectores*, *configurações* e *ferramentas* de suporte a desenvolvimento e evolução.

As próximas sub-seções vêm a elucidar cada um desses elementos pertinentes de *ADL*s, para no capítulo 3 ser discutido como são aplicados em abordagens conhecidas.

2.2.1 Componentes

O conceito de componentes constitui a base para a formação de *ADL*s. Uma definição bem aceita é fornecida por Szyperski [3]:

> Componentes de software são unidades de composição com interfaces contratualmente especificadas e dependências de contexto explícitas, que podem ser implantados de maneira independente e estão sujeitos a composição por terceiros

No contexto de arquiteturas de software, são definidos por Medvidovic [24] como "*entidades arquiteturais que encapsulam um subconjunto de funcionalidades do sistema e/ou dados, restringe o acesso a este subconjunto via interfaces bem definidas e tem dependências explicitamente definidas no seu contexto de execução*". Componentes são então unidades de execução utilizadas para se estruturar funcionalidades e são interagíveis somente através de serviços em particular. Estes serviços formam os conjuntos de pontos de interação do componente num sistema, divididos em requeridos (receptáculos) e fornecidos (facetas).

Componentes de software facilitam o desenvolvimento de aplicaçõoes complexas pelo desacoplamento de interfaces de implementações e explicitação da arquitetura como um todo. São tratados como caixas-pretas, personificando os princípios de encapsulamento, abstração e modularidade oriundos de engenharia de software, e podem ser tão simples quanto uma operação ou classe ou

tão complexo quanto um sistema inteiro. A reutilização parcial de fragmentos de design e implementação comuns entre soluções distintas favorece a criaçãao de linhas de produtos de software e facilita a implementação de famílias de produtos.

Por estes motivos, é interessante que a *ADL* faça uso de componentes de software. Na arquitetura, para descrever um componente, comumente são inclusos:

- Variáveis a serem acessadas/modificadas no sistema;
- Portas de requisição e fornecimento de serviços;
- Restrições de execução (ambiente, sistema operacional);
- Contratos a serem respeitados na utilização de serviços.

2.2.2 Conexões

Conexões são construções para modelar interações entre componentes, bem como suas propriedades, semânticas e restrições. Alguns exemplos incluem procedimentos, divulgação de eventos, canais de comunicação, *buffers*, protocolos, *proxies*, etc. Diversas *ADL*s empregam a idéia de conexões como elementos de primeira classe, trazendo maior abstração e podendo encapsular serviços de persistência, invocação, reflexão e transação. Uma conexão usualmente tem associada a si um serviço bem definido e um conjunto de pontos de interação. A depender da implementação, a semântica pode fornecer mecanismos de análise de interação, consistência e aplicação de restrições de comunicação.

2.2.3 Configurações

Configurações arquiteturais (ou topologia) são definidas como grafos em que componentes são vértices e conexões são arestas. É através da configuração que pode-se inferir se conexões têm serviços e protocolos compatíveis com as respectivas portas, se componentes têm suas dependências paramétricas satisfeitas, onde cada componente será implantado e até mesmo como instanciar e evoluir toda a infraestrutura do sistema de software. Algumas abordagens fornecem suporte a composição hierárquica, significando que um determinado sistema pode ser visto como um componente composto recursivamente ou como um único componente que faz parte de uma arquitetura maior. É vital que neste aspecto as *ADL*s forneçam mecanismos que permitam que o sistema seja especificado de maneira heterogênea (independendo de, por exemplo, ambientes de execução, linguagens de programação, sistemas

de componentes), evolutiva (reflexão e manipulação sobre o grafo) e dinâmica (aplicação de modificações em tempo de execução). As configurações podem ser subdividas em duas partes: estilo arquitetural e sistema, descritas a seguir.

2.2.4 Estilos Arquiteturais e Sistemas

São comuns casos em que diferentes níveis de abstração devem ser providos para facilitar o entendimento e a especificação da descrição arquitetural. Isso demanda elementos arquiteturais com granularidades variadas.

Estilos arquiteturais são um conjunto de regras aplicadas sobre os elementos da arquitetura, como os descritos anteriormente, de maneira a se impor padrões determinados ao sistema. São utilizados para representar famílias de descrições arquiteturais de sistemas que têm algo em comum: padrões de configuração, tipos de recursos, restrições, etc. O uso de estilos permite a análise automática de conformidade arquitetural. Permite também a tipagem de componentes, conexões e configurações de maneira a se encapsular o seu uso em unidades reutilizáveis e extensíveis. Isso possibilita que o sistema seja (re)instanciado de maneiras diferentes. A utilização deste conceito aumenta a compreensibilidade das aplicações por expor as propriedades dos componentes através das instâncias. Alguns exemplos incluem: estilos baseados em eventos, cliente-servidor, publicação-assinatura (*publish-subscribe*) e canais de eventos [26, 24].

O conceito de **Sistemas** é por vezes introduzido como um componente composto por outros elementos arquiteturais. Sistemas representam configurações arquiteturais formadas por (instâncias de) conexões e componentes que podem ser construídas de forma hierárquica. Isso permite a composição de sistemas através de outros sub-sistemas. Algumas *ADL*s que utilizam este conceito são *Darwin* [14] e *ACME* [9].

A figura 2.1 demonstra o esquema de um estilo arquitetural cliente-servidor típico com dois sistemas. Os componentes (*A* e *B*) estão ligados pela conexão *CS_CONNECTION*, indicando que um conjunto de clientes pode se conectar a um servidor. Logo abaixo, são demonstradas dois possíveis sistemas, um com um cliente e um servidor (*A1* e *B1*) e outro com dois clientes e um servidor (*A2*, *A3* e *B2*).

A distinção entre estes níveis de abstração traz vantagens importantes no reuso de tipos arquiteturais e manutenção de sistemas. Durante a evolução, estes elementos são vitais para se especificar mudanças na configuração e nos componentes implantados (mais detalhes na seção 2.4).

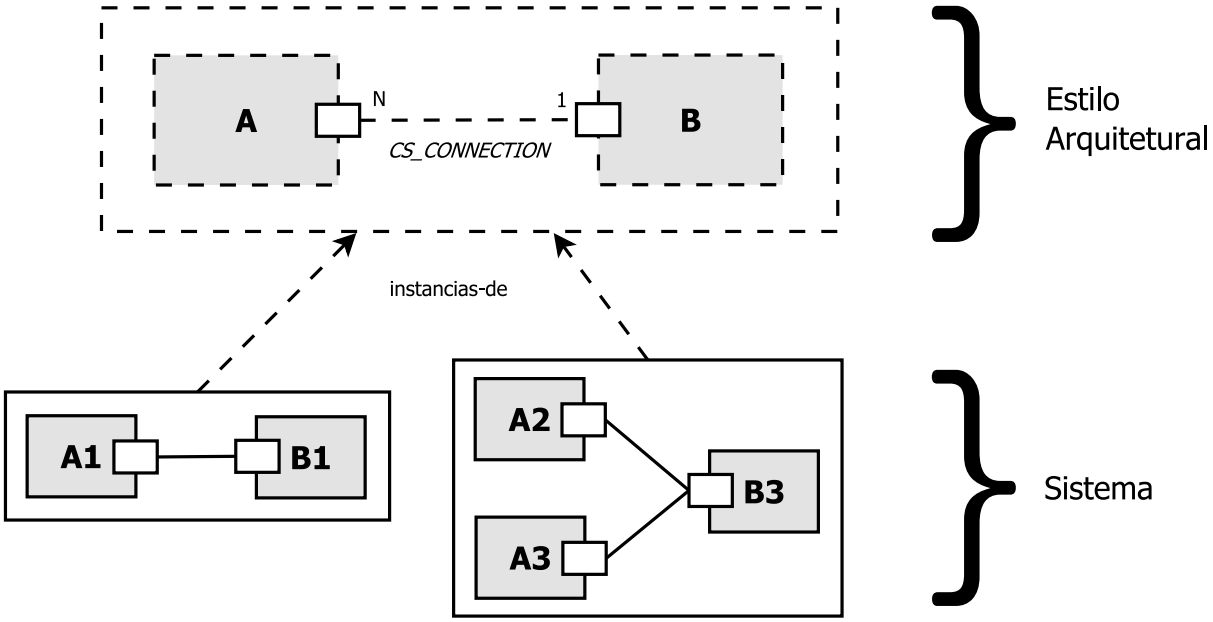


Figura 2.1: Exemplo de estilo arquitetural

2.2.5 Ferramentas

É recomendável que, em paralelo com o desenvolvimento de *ADL*s, sejam utilizados mecanismos que proporcionem racionalização e manipulação arquitetural por (conjunto de) ferramentas. Esses instrumentos auxiliam as atividades de análise, inicialização, controle e evolução de sistemas. Podem também prover assistência a geração de código, múltiplas visões (dependentes da parte interessada) e aplicação e verificação de adaptações.

A figura 2.2 mostra uma captura de tela do sistema Evolve [27] (descrito na seção 3.3) em execução. Esta aplicação gráfica é utilizada para montar sistemas a partir de componentes e conexões. Esta ferramenta também provê suporte ao reuso e extensão de componentes, tornando diversos tipos de evolução possíveis.

2.3 Evolução Dinâmica de Software

Um dos desafios encontrados no ciclo de vida de software é o de que todos os artefatos produzidos estão sujeitos a mudanças, desde requisitos e design até código fonte e executável. Assim, pode-se dizer que no período entre a criação inicial e remoção completa, os sistemas sofrem evolução. O objetivo da evolução de software usualmente é alterar algum artefato de maneira a corrigir, melhorar, estender ou reduzir funcionalidades. Uma definição aceita de evolução de software é fornecida por Chapin et al. [28]:

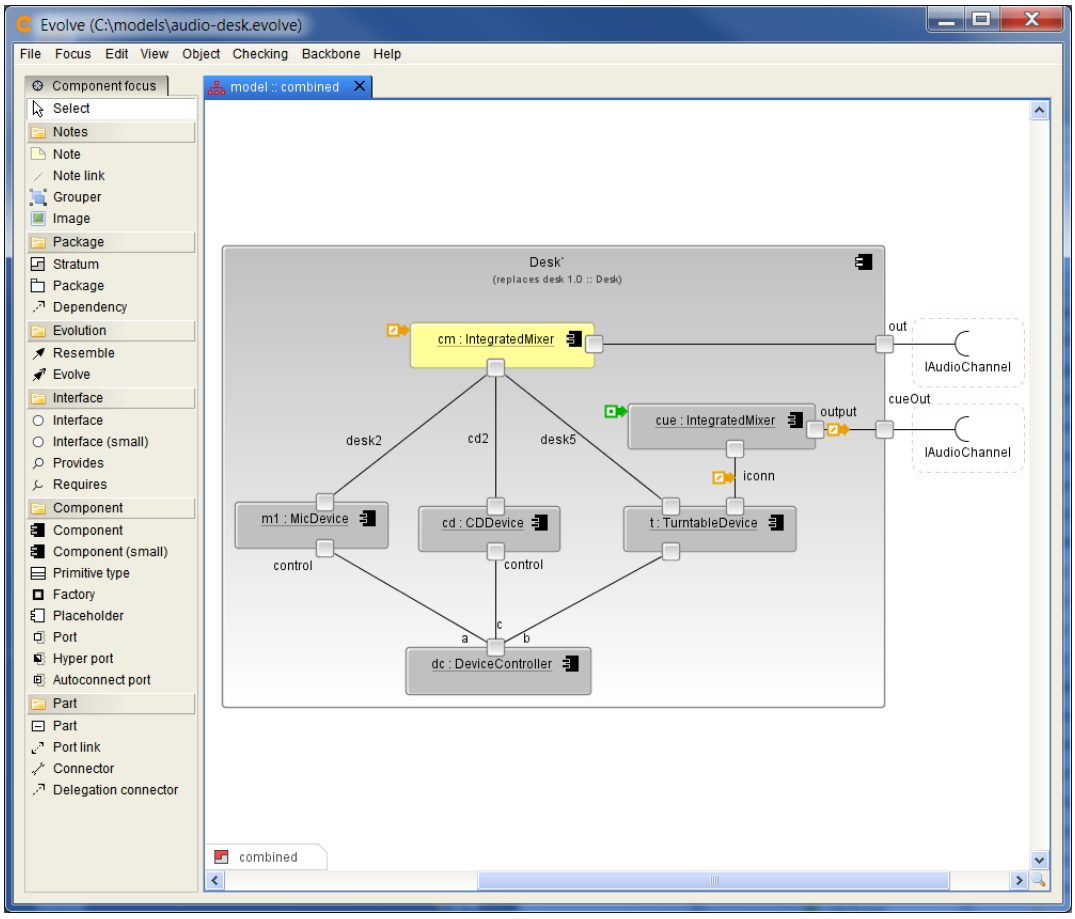


Figura 2.2: Captura de tela do sistema Evolve

> Evolução de software ocorre quando a manutenção feita é do tipo incremental, corretiva ou redutiva, ou altera propriedades sensíveis ao cliente, ou seja, é do tipo adaptativo ou performático

É importante ressaltar que estas atividades ocorrem de modo a satisfazer novos requisitos baseado em experiência com o uso do software e suas interações com o ambiente.

Em alguns casos, as consequências de desligamento de um sistema para aplicação de adaptações não são aceitáveis. Software de suporte a vida, gerenciamento de energia ou de transações bancárias, se desligados, podem acarretar em consequências drásticas. Evolução dinâmica pode então ser vista como "*um processo de mudanças gradual realizado em um sistema de software previamente operacional para corrigir, melhorar, estender ou reduzir parte de sua funcionalidade, que ocorre durante a sua execução, sem influenciar partes não afetadas pela mudança, e que preserva a integridade do sistema*" [6].

A taxonomia criada por Buckley et al. [29] introduz algumas dimensões de evolução, de modo a se responder questões tais quais *quando*, *como*, *onde* e *como* mudanças são aplicadas. Algumas destas definições são vitais ao escopo

deste trabalho e são descritas a seguir.

2.3.1 Tempo

A depender da linguagem de programação ou ambiente de desenvolvimento, pode-se distinguir as fases de ciclo de vida do software tais quais tempo de compilação, tempo de carregamento e tempo de execução. Estas fases são determinantes na criação e definição do mecanismo de adaptação para cada sistema, e três categorias surgem para caracterizar a dimensão de *tempo* de aplicação de mudanças:

Estática ou *offline* A evolução ocorre em tempo de compilação, onde o programador estende ou edita o código fonte da aplicação. Nesses casos, é necessário recompilação e reexecução para que as alterações tenham efeito;

Dinâmica ou em tempo de execução A evolução ocorre de maneira que mudanças são geradas ou aplicadas durante a execução do software. Componentes correntes podem ser trocados/alterados enquanto componentes novos são adicionados à arquitetura, sem a necessidade de se desligar a aplicação por completo. Usualmente, tais sistemas são considerados reflexivos, na medida em que podem analisar e modificar o seu próprio comportamento;

Tempo de carregamento Esta categoria se encontra entre as duas primeiras, com mudanças sendo aplicadas enquanto elementos do software são carregados no sistema executável. Este tipo de evolução é adequada para adaptar componentes compilados estaticamente de maneira a encaixá-los em determinados contextos de implantação. Um exemplo é o *ClassLoader* da linguagem Java, que permite aplicar modificações em classes durante o carregamento.

Essa dimensão influencia diretamente o mecanismo de adaptação, e sistemas que permitem evolução de forma dinâmica devem ter um controle de modo a garantir a integridade da aplicação mesmo durante as mudanças. A aplicação de adaptações incorretas ou de maneira incorreta pode causar mal funcionamento do sistema, comportamento não esperado, introdução de novos defeitos ou quebra de funcionalidades. É importante manter um histórico de mudanças aplicadas através de mecanismos de controle de versionamento, possibilitando a criação de alternativas em face de falhas.

2.3.2 Antecipação de mudanças

Um dos muitos desafios de evolução é que todos os artefatos criados durante o ciclo de vida estão sujeitos a mudanças, desde requisitos e design até arquitetura, código fonte e código executável. O processo de evolução pode ocorrer durante qualquer fase, assim algumas mudanças podem ser identificadas já durante o desenvolvimento, enquanto outras podem não ser previstas até a exposição a usuários finais.

Mudanças antecipadas são captadas em tempo de design e podem ser ativadas quando uma determinada condição é alcançada ou evento ocorre. Nestes casos já há uma especificação anterior indicando o comportamento de cada situação. Um exemplo deste tipo de mudança é a técnica de instanciação tardia do *Darwin* [14], que cria componentes por demanda.

Em contraste, as mudanças não previstas em tempo de desenvolvimento ocorrem após o sistema já estar implantando, devido a mudanças na lógica, infraestrutura, *feedbacks*, mudanças técnicas, etc. Através de um processo de configuração dinâmica *incremental*, adaptações arbitrárias são aplicadas sem a necessidade de reconstrução do sistema inteiro. Essas mudanças incrementais devem ocorrer também sem interromper partes não afetadas [7]. Tipicamente, mudanças antecipadas envolvem muito menos esforço de implementação do que não-antecipadas. O nível corrente de suporte a estas adaptações (também chamadas de arbitrárias) ainda é longe do ideal [8, 29].

2.3.3 Granularidade

Os artefatos envolvidos na evolução definem a *granularidade* das mudanças. As mudanças podem ser aplicadas desde requisitos e arquitetura até código objeto e em execução. Pode-se dividir as granularidades em três categorias:

Baixa Mudanças no nível da arquitetura do sistema, que impactam subsistemas. São realizadas através de reconfiguração, por adicionar/remover componentes e conexões. Um exemplo de reconfiguração é o uso de instanciação tardia do sistema *Darwin* (mais detalhes podem ser encontrados na seção 3.2);

Média Mudanças no nível de definição de tipos das entidades que fazem parte do sistema, como composições de componentes, módulos, classes, assim como todas as suas instâncias. Um exemplo desse tipo de evolução é mecanismo de carregamento de classes (do inglês *ClassLoader*) da

linguagem Java, que permite carregar/descarregar classes de maneira dinâmica, em tempo de execução;

Fina Mudanças que ocorrem a nível de variáveis, métodos e comandos. São transversais às anteriores e implicitamente providas por níveis acima.

Em geral, as abordagens provêem esses mecanismos em um único nível, porém, por serem complementares, quando combinados, provêem um grau de flexibilidade interessante para sistemas de software críticos.

2.3.4 Confiabilidade

Um problema encontrado em evolução dinâmica é a definição de um estado adequado para a aplicação de adaptações. Neste estado (chamado *quiescente*), os componentes devem, garantidamente, não estar efetuando nem respondendo a requisições de outros componentes [30]. Em particular, é interessante evitar impor restrições arquiteturais de qualquer tipo nos componentes, demandar pouco esforço do programador da aplicação e afetar minimamente a aplicação que esteja sendo adaptada.

Para isso, seguimos a abordagem proposta por Krame e Magee [8], que é suficiente para garantir consistência e alcançar estado quiescente em tempo finito. No modelo proposto, um sistema é visto como um grafo dirigido tal que os nós são entidades de processamento que podem iniciar ou servir requisições, e as arestas são as suas conexões. Um componente se encontra em estado quiescente quando as seguintes propriedades são atendidas:

- O componente não está correntemente envolvido em uma transação que iniciou;
- Não pode iniciar uma nova requisição;
- Não pode servir uma transação;
- Nenhuma transação com este componente será iniciada.

Esta abordagem leva o sistema a um estado seguro para ser adaptado, no sentido de que os componentes não contém transações parcialmente completas, e íntegro, já que o estado de cada componente não mudará devido a novas transações.

Outro desafio é a manutenção e transferência de estados do sistema. Os artefatos afetados devem ser alterados com interrupção mínima do sistema e em seguida sofrer uma migração (ou transformação) das estruturas e conteúdo interno em execução [6].

2.4 Evolução Dinâmica em Arquiteturas de Software

Arquiteturas de software podem evoluir através das estruturas descritas na seção 2.2. As adaptações são geralmente expressas como a adição/remoção/manipulação de componentes e conexões durante a execução. Evoluir uma arquitetura pela modificação da sua descrição apresenta desafios como: preservação de propósitos e interesses críticos, análise e verificação da descrição resultante, e uso de adaptações como construções de primeira ordem(*deltas*) [18]. Arquiteturas que mantêm não somente a sua descrição mas também a descrição de evoluções é considerada uma Arquitetura de Software Dinâmica [25]. Representam sistemas que são constituídos não somente por uma estrutura fixa, mas que podem reagir a eventos e requisitos e efetuar reconfiguração de suas entidades. Em *ADL*s, pode-se considerar que existem dois tipos de dinamismos: reconfiguração dinâmica e evolução dinâmica de tipos. No primeiro, o grafo da configuração do sistema é modificado [7], enquanto no segundo são modificados os tipos que compõem as estruturas. Tais comportamentos são complementares, ou seja, uma mudança num tipo de componente pode acarretar na mudança de instâncias dos mesmos, e vice-versa. Ambos são descritas mais detalhadamente a seguir.

2.4.1 Reconfiguração Dinâmica

Reconfiguração dinâmica refere-se às modificações aplicadas em tempo de execução realizadas sobre a estrutura do sistema de software. Este tipo de dinamismo é fundamental para se modelar sistemas que alteram o seu comportamento de maneira arbitrária, tais quais sistemas auto-organizáveis, móveis, sensíveis ao contexto ou tolerantes a falhas. Uma definição bem aceita é apresentada por Magee e Kramer [12]:

> Uma funcionalidade de uma *ADL* que permita a descrição de arquiteturas dinâmicas de software na qual a organização de componentes e conexões podem mudar durante a execução (...) Evolução estrutural inclui mudanças nos *bindings* (conexões) entre componentes e o conjunto de instâncias de componentes

Essas mudanças afetam o conjunto de instâncias arquiteturais de um sistema, mas não os tipos ou especificações que definem o comportamento destas instâncias.

A figura 2.3 mostra como uma reconfiguração pode ocorrer dentro de um sistema (basedo no exemplo da sub-seção 2.2.4). Os elementos em vermelho

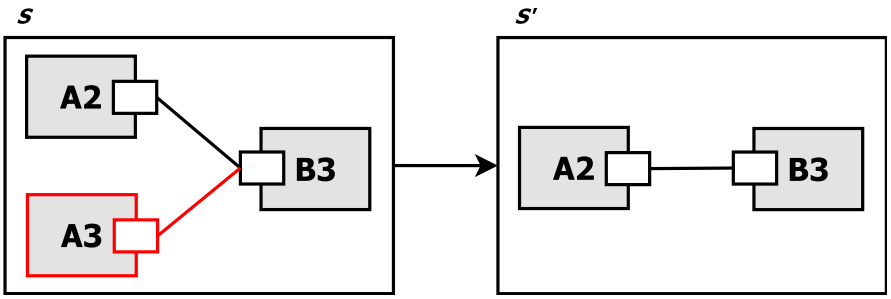


Figura 2.3: Reconfiguração arquitetural

são removidos da arquitetura dinamicamente, mantendo o bom funcionamento e consistência geral da aplicação, que passa do estado *S* para *S'*.

2.4.2 Evolução Dinâmica de Tipos Arquiteturais

De modo a prover suporte a evoluções arbitrárias, é necessário também evoluir os tipos dos elementos arquiteturais e suas instâncias em tempo de execução. Este tipo de dinamismo aumenta o grau de adaptabilidade e favorece a aplicação de evoluções arbitrárias, como a introdução de novos comportamentos ou atualização de implementação de componentes para corrigir um mal funcionamento, criando de fato arquiteturas completamente novas.

Poucos trabalhos suportam este tipo de evolução, e aspectos como substituibilidade e manutenção de restrições ainda se encontram em aberto.

2.5 SCS e SCS-Composite

Neste trabalho, utilizamos um sistema de componentes de software para especificar componentes e dependências, o SCS [19]. O modelo foi criado inspirado no Microsoft COM [31] e OMG CCM [32], com o objetivo de fornecer abstrações mais simples que estes e outros modelos anteriories. Define portas de provimento de serviços (*facetas*) e de requisição (*receptáculos*). Permite que o desenvolvedor interaja com aplicações distribuídas multi-linguagens, especificando, configurando e inspecionando a estrutura da aplicação.

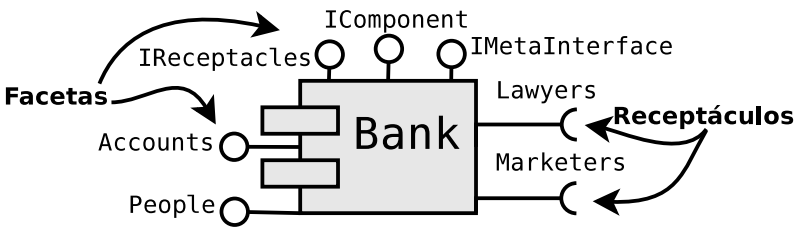


Figura 2.4: Exemplo de um componente SCS

A figura 2.4 ilustra um componente SCS denominado *Bank*. Na figura, os círculos simbolizam as interfaces fornecidas, enquanto os semi-círculos são as interfaces requeridas. O componente *Bank* então fornece um serviço do tipo *People* e outro do tipo *Accounts* e requer um do tipo *Lawyers* e outro do tipo *Marketers*. Os outros três serviços fornecidos são essenciais ao funcionamento da infraestrutura, e criados de maneira automática pelo SCS. O objetivo de cada uma das interfaces é:

Controle Faceta *IComponent*, responsável por identificar, ativar, desativar e prover acesso a outras facetas;

Conexões Faceta *IReceptacles*, gerencia as dependências paramétricas remotas e locais;

Introspecção Faceta *IMetaInterface*, fornece acesso às descrições das dependências e serviços providos pelo componente.

A partir de descrições, o SCS instancia o componente e cria automaticamente todas as suas facetas e receptáculos. Em Santos (2012) [1], estas idéias foram estendidas para prover suporte à composição hierárquica de componentes.

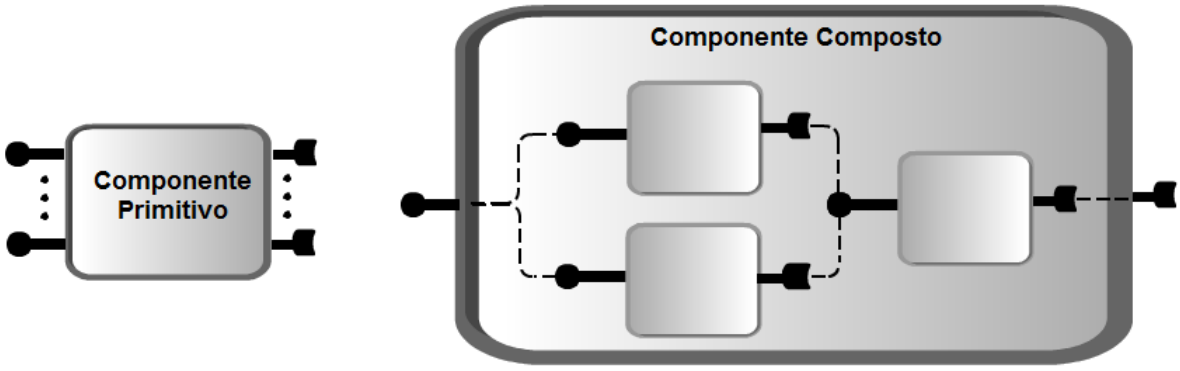


Figura 2.5: Componente primitivo e composto SCS [1]

A figura 2.5 ilustra a diferença entre um componente composto e um primitivo (também chamado de monolítico). O componente monolítico pode ser visto como uma unidade que fornece e requer serviços. Já o composto é o resultado de uma composição de outros componentes, encapsulando estruturas mais complexas que podem, ou não, estarem expostas. Além dos serviços padrões, um componente composto quando criado, também fornece os seguintes serviços:

Navegação Faceta *ISuperComponent*, de acesso aos componentes compostos que encapsulam um determinado componente, para proporcionar navegação hierárquica;

Membrana Faceta *IContentController*, com mecanismos de introspecção de subcomponentes e mapeamento de serviços e dependências.

O uso da composição é essencial na representação/configuração/manipulação do *sistema*. Através desta abstração é possível navegar na hierarquia especificada, inspencionando as instâncias de componentes e conexões descritas na *ADL*. Esta funcionalidade é essencial uma vez que um dos objetivos deste trabalho é justamente modificar essas unidades e relacionamentos, como demonstrado em 2.4.1. O uso de componentes compostos também é essencial na exportação de serviços, fazendo com que famílias de sistemas possam ser conectadas umas às outras.

2.6 Considerações Finais

Neste capítulo foram apresentados os conceitos básicos acerca deste trabalho. A ferramenta **SCS-DynAdapt** desenvolvida utiliza uma descrição arquitetural para criar e evoluir sistemas distribuídos dinamicamente. Desta maneira, é essencial prover a abstração de desenvolvimento orientado a componentes de software. Utilizamos o modelo SCS-Composite para dar suporte a componentes desenvolvidos na linguagem de programação Lua. As unidades são específicadas fornecendo informações de serviços fornecidos e requeridos (dependências), podendo ser reutilizados em diferentes sistemas. Estes serviços são ligados através de conexões expressas na descrição arquitetural, que podem ser desfeitas ou modificadas ao longo do ciclo de vida do software.

A partir dos componentes e conexões, é possível criar configurações de estilos arquiteturais e sistemas. A ferramenta se encarrega de verificar se o sistema obedece às restrições do estilo e se todas as dependências dos componentes estão satisfeitas. Em seguida, é possível implantar automaticamente os componentes em nós distribuídos da infraestrutura.

De maneira a ter controle sobre as requisições dos componentes, a criação de componentes SCS foi modificada para acomodar um serviço de controle de estados, a interface *ILifeCycle*. Com este recurso, é possível aplicar adaptações de maneira segura e sem interromper partes não afetadas pelas mudanças.

Tendo este controle em mãos, a arquitetura especificada pode sofrer evolução dinâmica sobre os seus elementos. Componentes podem ter serviços substituídos em tempo de execução, de maneira a corrigir falhas ou melhorar alguma funcionalidade. Também podem ser trocados completamente, dando espaço a novas implementações. As conexões podem desfeitas ou criadas, dando espaço a reconfiguração arquitetural. Além disso, novos tipos de componentes

podem ser introduzidos, dando suporte a evoluções arquiteturais não previstas em tempo de design.

Todos estes mecanismos estão descritos mais detalhadamente no capítulo 4.

3
Trabalhos Relacionados

Diversas *ADL*s foram propostas de maneira a fornecer descrições estruturais que auxiliem o controle do ciclo de vida do software. Ainda não há, contudo, um consenso na literatura acerca de quais aspectos arquiteturais devem estar presentes na *ADL* ou quais recursos (visuais, semânticos) são mais adequados a determinados problemas. Algumas abordagens focam em domínios particulares, enquanto outras tentam fornecer uma linguagem de modelagem o mais genérica possível [25].

Vale a pena destacar alguns trabalhos que tiveram maior influência neste trabalho:

ACME Provê um formato intermediário de suporte ao mapeamento de especificações arquiteturais de uma *ADL* para outra, possibilitando a integração de ferramentas de suporte. A rigor não é uma *ADL*, porém é muito útil para comparar as linguagens [9];

Wright *ADL* com especificações formais rígidas, utiliza a linguagem *CSP* para descrever padrões de interações. A versão inicial oferecia suporte apenas a arquiteturas estáticas, tendo sido posteriormente estendida para suportar arquiteturas dinâmicas. Componentes e conexões são modelados em alto nível de abstração, de maneira desacoplada da implementação [33];

Aesop Sistema de desenvolvimento de arquiteturas baseadas em estilos arquiteturais de domínios específicos. Possui uma grande variedade de estilos arquiteturais que podem ser obtidos para customizar arquiteturas à maneira do desenvolvedor. Diversas ferramentas podem ser integradas para se obter verificação de consistência de tipos, maior formalidade no protocolo de comunicação e geração automática de código [34].

É importante, para a apresentação da ferramenta desenvolvida neste trabalho, apresentar como funcionam abordagens consolidadas na literatura. Dentre os critérios para seleção de análise dos trabalhos relacionados, a presença de evolução dinâmica com elementos de especificação na própria linguagem arquitetural foi o mais relevante. Neste contexto, poucas *ADL*s

fornecem mecanismos de evolução. As seções a seguir apresentam os trabalhos relacionados.

3.1 CONIC

O *CONIC* [15, 35, 7] foi uma das primeiras *ADL*s a surgirem no meio acadêmico com tratamento de reconfiguração de sistemas distribuídos. O ambiente fornece um conjunto de ferramentas com funcionalidades de compilação, configuração, depuração e execução de programas distribuídos. Nesse trabalho já se separa as atividades de configuração e programação, porém apenas a linguagem *Pascal*(estendida) é suportada.

Componentes são conceituados como módulos de tarefas. Cada tarefa é definida em termos de portas bem tipadas que especificam informações fornecidas e requeridas, através de portas de saída e de entrada. O código 3.1 demonstra a declaração de uma tarefa no *CONIC*:

```
1  task module scale(scalefactor:integer);
2    entryport
3      control: boolean;
4      input: real reply signaltype;
5    exitport
6      output: real reply signaltype;
7    var
8      value: real;
9      active: boolean;
10   begin
11     active := false;
12     loop
13       select
14         receive active from control
15       or
16         when active
17         receive value from input reply signal =>
18           send value/scalefactor to output wait signal;
19       end
20     end
21   end
22 end.
```

Código 3.1: Módulo em *CONIC*

O componente *scale* recebe valores reais na porta de entrada denominada *input* e envia valores para a porta de saída *output* quando a variavel booleana *control* assume valor verdadeiro.

A linguagem de configuração, por sua vez, é especificada através de módulos de agrupamento entre as tarefas, e é ilustrada no exemplo do código 3.2:

```
1  group module monitor(Tfactor,Pfactor:integer);
2    exitport
3      press, temp: real reply signaltype;
```

```
4   entryport
5     control:boolean;
6   use
7     scale; sensor;
8   create
9     temperature: sensor;
10    pressure: sensor;
11    Tscale: scale(Tfactor);
12    Pscale: scale(Pfactor);
13  link
14    temperature.output to Tscale.input;
15    pressure.output to Pscale.input;
16    Tscale.output to temp;
17    Pscale.output to press;
18    control to Tscale.control, Pscale.control;
19  end.
```

Código 3.2: Configuração em *CONIC*

O exemplo demonstra uma configuração entre dois tipos de tarefas (importadas ao escopo com o comando *use*) denominadas *scale* e *sensor*, com instanciação através do comando *create* (criando *temperature*, *pressure*, *Tscale*, *Pscale*). As conexões são feitas através da primitiva *link*, ligando as portas dos componentes através de um mecanismo de troca de mensagem. É interessante notar que a interface de uma configuração é a mesma de um componente, de maneira que configurações podem também ser conectadas, provendo um mecanismo semelhante ao de componentes compostos.

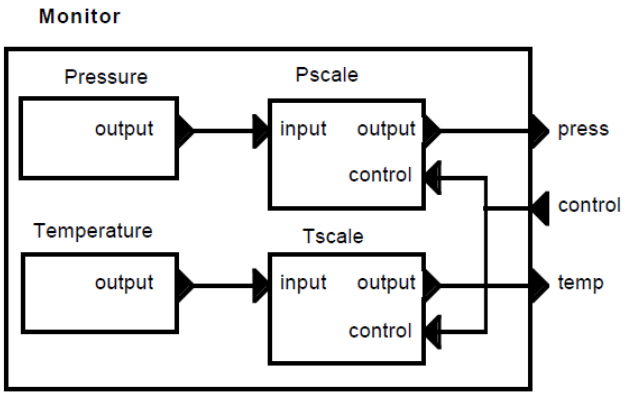


Figura 3.1: Configuração *monitor*

A figura 3.1 ilustra o funcionamento da configuração do agrupamento *monitor*. A configuração é vista de maneira parecida com a visão de um componente composto por subcomponentes com funcionalidades e objetivos específicos. Além dessas funcionalidades, o *CONIC* introduziu um modelo do processo de configuração para evolução de sistemas [7] trazendo à tona um estudo de propriedades de evolução que são importantes, como:

Linguagens de programação : Modularidade, representação de interconexões, primitivas de interconexão, independência contextual;

Especificação de mudanças e configurações : Definição de contextos, unidades de instanciação, modularidade de especificações, linguagens de configuração declarativa;

Sistemas operacionais : Gerenciamenteo de módulos, suporte a comunicação, modificações com restrições de tempo-real, *overhead*;

Validação : Consistência com especificações, novas conexões, novos comportamentos;

Gerenciamento de configurações : *Deployment* de novos componentes e configurações, estratégias de aplicação e alocação de mudanças.

A partir daí é permitido que se especifique mudanças que aplicadas ao documento de especificação da arquitetura resultam em uma nova configuração. As reconfigurações podem ser feitas de duas maneiras: através da edição de um documento de descrição da configuração, que traz a necessidade de recompiação de todo o sistema ou pela especificação de mudanças na configuração da arquitetura, que são aplicadas de maneira dinâmica.

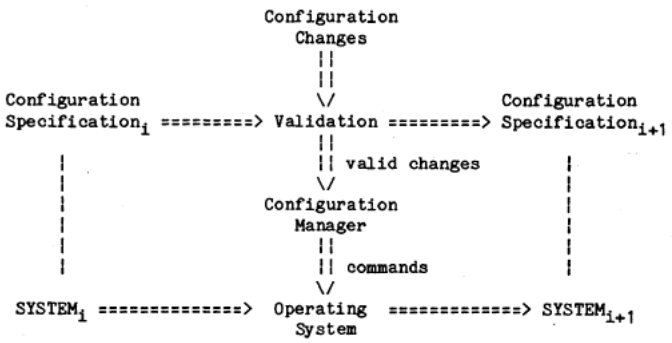


Figura 3.2: Gerente de configuração dinâmica no *CONIC*

A figura 3.2 é uma representação do gerente de configuração dinâmica do *CONIC*. Através de uma especificação numa linguagem de configuração, o gerente valida as mudanças e produz um novo descritor do sistema, em uma forma "compilada". As operações são então executadas via módulos do sistema operacional, que enviam mensagens para as portas de entrada dos componentes indicando as alterações.

```
1 change ward;
2   unlink family k: [I nbed]
3     bed [k] .alarms from nurse.alarms [k];
4     nurse.query[k] from bed [k] .status;
5   delete nurse;
```

```
6     remove nurseunit;
7     use enhancednurseunit;
8     create newnurse:enhancednurseunit;
9     link family k: [1 * nbed]
10      bed [k] .alarms to newnurse.alarms [k],;
11      newnurse.query [k] to bed [k] .status;
12    link newnurse.history to log.history;
13  end.
```

Código 3.3: Especificação de mudanças no *CONIC*

O código 3.3 exemplifica a aplicação de mudanças na configuração de um determinado sistema. Os elementos de construção das mudanças são: *Remove* remove o conhecimento acerca de um tipo de módulo da configuração, *Delete* remove a instância do módulo e *unlink* remove a instância da conexão (portas de saída e entrada). A adaptação é considerada válida somente se todas as ligações aos módulos removidos são desfeitas e todas as instâncias de tipos removidos são também removidas. Apesar de um trabalho relevante, o *CONIC* apresenta certas limitações, como:

- Se utiliza de conjunto fixo de primitivas de comunicação, limitando o alcance das aplicações [7];
- A *ADL* é mapeada diretamente a uma linguagem de programação (Pascal);
- Não provê suporte a diferentes protocolos de comunicação;
- Não existe uma linguagem intermediária para as interfaces (idl), utilizando apenas um mecanismo direto de mensagem;
- Não existe um registro das adaptações ao longo do tempo, nem operações de diferenciação.

3.2 Darwin

Darwin [14] é uma *ADL* declarativa que dispõe de semântica operacional baseada em π-cálculo para criar composições hierárquicas que formam sistemas distribuídos e paralelos. Também utiliza os conceitos básicos de componentes, conexões e configurações, porém somente o componente é expresso de maneira explícita. Separa estruturação de computação e interação, de maneira a prover separação de interesses. Fornece composicionalidade por descrever arquiteturas genéricas que podem ser instanciadas de maneira específica. Componentes abstratos podem ser expressos, de maneira que em tempo de execução um componente arbitrário pode assumir seu lugar, aumentando a flexibilidade de reconfigurações.

O *Darwin* é o sucessor do *CONIC* [15]. De maneira similar, componentes são especificados diretamente em uma linguagem de programação, porém as interfaces de serviço estão na *ADL*.

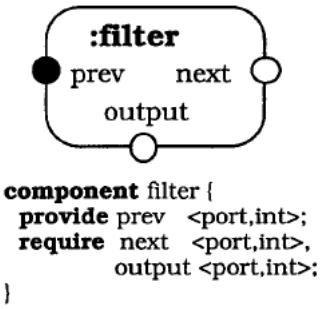


Figura 3.3: Componente filtro em *Darwin*

A figura 3.3 demonstra um componente de filtro juntamente com a sua descrição: os modificadores *provide* e *require* são serviços providos (*prev*) e requeridos (*next* e *output*), respectivamente. A identificação dos serviços é feita a nível de escopo interno, de maneira que não é necessário que a unidade tenha conhecimento específico acerca do ambiente externo (localização de uma certa dependência, por exemplo). Assim, componentes podem ser especificados, implementados e testados de maneira isolada, provendo *independência contextual*. Este conceito facilita o reuso e simplifica a substituição de instâncias durante a evolução.

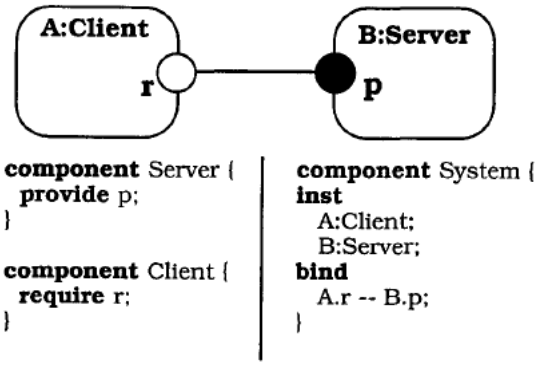
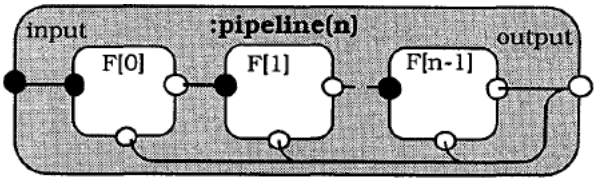


Figura 3.4: Ligação em *Darwin*

As conexões também funcionam através de passagem de mensagens. A figura 3.4 demonstra o funcionamento: é definido um componente composto *System* com uma instância do tipo *Server* e outra do tipo *Client*. O modificador *bind* efetua as ligações entre as instâncias *A* e *B*, referenciando as portas de cada um. Neste momento são feitas verificações de tipo para validar a conexão.

A linguagem pode ser utilizada para expressar estruturas que evoluam dinamicamente com o tempo, e que não têm um conjunto de componentes fixados em tempo de design. O mecanismo é guiado por rígidas bases formais:

instanciação tardia e instanciação dinâmica direta. No primeiro, cada componente é primeiramente declarado e instanciado somente quando algum dos seus serviços é requerido, fazendo com que não seja preciso definir o número de instancias *a priori*. Já no segundo eles são instanciados imediatamente.



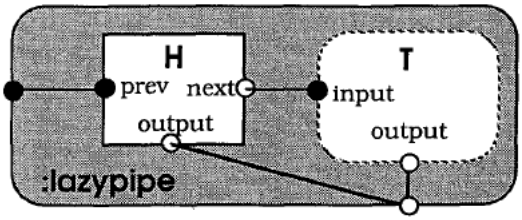

```
component pipeline (int n) {
  provide input;
  require output;

  array F[n]:filter;
  forall k:0..n-1 {
    inst F[k];
    bind F[k].output -- output;
    when k<n-1 bind
      F[k].next -- F[k+1].prev;
  }
  bind
    input -- F[0].prev;
    F[n-1].next -- output;
}
```

Figura 3.5: Instanciação dinâmica direta *Darwin*

A figura 3.5 demonstra o uso de instanciação direta em uma arquitetura de *pipeline* em que não é definido em tempo de design o número de componentes do tipo *Filter*. O número de componentes é definido através de um parâmetro de inicialização, e dentro de uma estrutura de repetição cada comando *inst* instancia um componente e *bind* os conecta.

Já na figura 3.6 o modificador *dyn* faz com que a instância T do tipo *lazypipe* não seja imediatamente instanciado. Quando a instância H tenta acessar o serviço $T.input$, a instanciação de T (assim como de suas ligações) é desencadeada. O princípio de instanciação tardia pode ainda ser combinado com recursão para descrever estruturas, como é o caso. Esta combinação pode ser usada pra descrever uma gama de estruturas distribuídas (árvores de busca, divisão e conquista). Usualmente esta abstração é adequada para situações onde o designer consegue prever como o sistema irá evoluir (não arbitrariamente). Pode-se dizer que *Darwin* provê uma descrição acurada da estrutura potencial do sistema.

Darwin ainda permite a exportação (e importação) de serviços na ADL, de maneira que os subcomponentes de um sistema podem interagir com outras entidades. A instanciação, contudo, ocorre somente para novos componentes, remoções tanto de componentes quanto de conexões não é possível. Não é possível também expressar ligações cíclicas (em anel).




```
component lazypipe {
  provide input;
  require output;
  inst
    H:filter;
    T:dyn lazypipe;
  bind
    input -- H.prev;
    H.next -- T.input;
    H.output -- output;
    T.output -- output;
}
```

Figura 3.6: Instanciação tardia em *Darwin*

3.3 Evolve

O sistema *Evolve* estende as idéias, ferramentas e construções do *Darwin* [14] para permitir mais flexbilidade de evolução arquitetural. Um dos objetivos no desenvolvimento de software orientado a componentes é aumentar o reuso na construção de sistemas. Contudo, usualmente são necessárias mudanças em componentes comuns para serem utilizados compartilhadamente, quebrando este paradigma. Este sistema traz à tona a idéia de semelhança (*resemblance*), que permite que componentes sejam definidos com base em outro já existente. Desta maneira, é possível expressar na *ADL* as transformações que o sistema sofre. O *Evolve* também faz uso extenso de composição hierárquica, *bindings* e integração de sistemas.

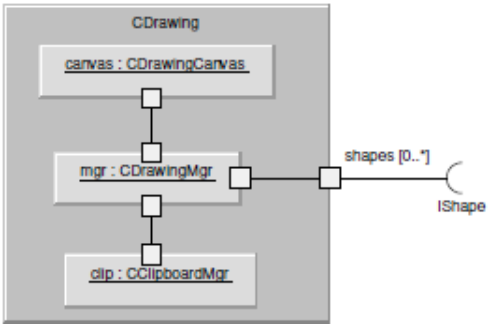


Figura 3.7: Componente *CDrawing*, *framework* de renderização

A figura 3.7 representa um componente com função de *framework* de renderização (*CDrawing*). É formado pelos sub-componentes, *canvas* (tipo *CDrawingCanvas*), *mgr* (tipo *CDrawingMgr*) e *clip* (tipo *CClipboardMgr*).

Uma dependência externa do tipo *IShape* é expressada com aridade múltipla ([0...*]).

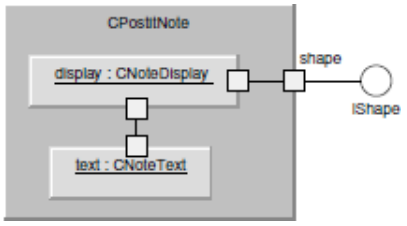


Figura 3.8: Componente *CPostItNote*, renderização de notas (*post-its*)

A figura 3.8 representa um componente renderizador de notas formado pelos sub-componentes *display*, do tipo *CNoteDisplay* e *text*, do tipo *CNoteText*. Este componente exporta um serviço também do tipo *IShape*, que é utilizado pelo *framework* de renderização. Com *Evolve*, alguns requisitos desejáveis à adaptação dos componentes são levantados a partir de cenários de motivação em que vários usuários utilizam componentes passíveis de atualização. Estes requisitos são:

- **Alteração**: Deve ser possível alterar um componente de modo a ser utilizado no novo sistema;
- **Minimização de impacto**: Alterações de reuso não devem impactar em usuários existentes do componente;
- **Ausência de código fonte**: A abordagem deve poder ser utilizada mesmo sem disponibilidade do código fonte;
- **Atualização**: Interessados em reutilizar um componente devem estar sujeitos a eventuais alterações, mesmo em componentes customizados.

A figura 3.9 ilustra a evolução dos componentes *CDrawing* e *CPostitNote*, através de modificações nas suas estruturas internas. Em ambos os casos, *semelhança* pode ser utilizada para especificar essas alterações.

```
1 component CNewDrawing resembles CDrawing {
2   replace-parts:
3     CNullClipboardMgr clip;
4   parts:
5     CZoomMgr z;
6   connectors:
7     zoom joins zoom@z to
8       surface@canvas;
9 }
```

Código 3.4: Criando um novo componente a partir de um componente base

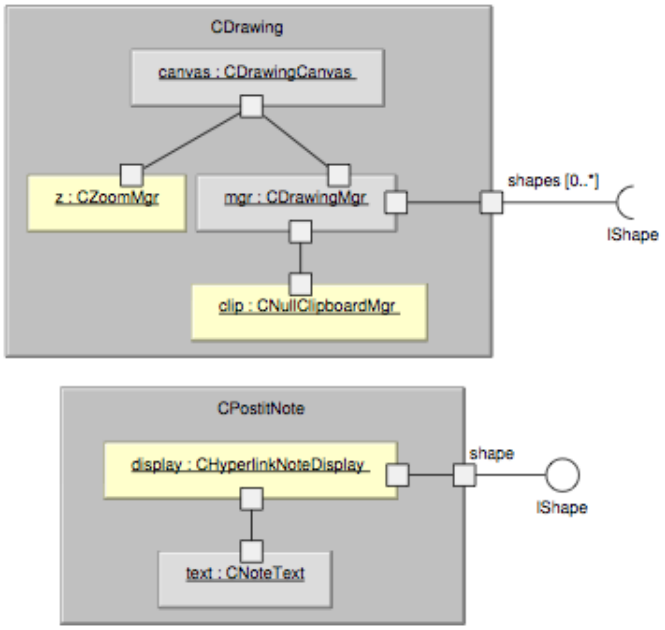


Figura 3.9: O novo componente, *CFastDrawingCanvas*

O código 3.4 ilustra esse mecanismo na *ADL* através da substituição de partes do componente composto. O tipo do componente *clip* é alterado para *CNullClipboardMgr*. Um novo tipo de componente (*CZoomMgr*) é introduzido, e uma instância *z* é adicionada e conectada. Essa abordagem, contudo, cria um novo componente (*CNewDrawing*), sem alterar as instâncias já implantadas. Porém podem ocorrer casos em que é necessário alterar tipos de componentes já existentes.

```
1 redefine-component CDrawing resembles [previous]CDrawing {
2   replace-parts:
3     CNullClipboardMgr clip;
4   parts:
5     CZoomMgr z;
6   connectors:
7     zoom joins zoom@z to
8       surface@canvas;
9 }
```

Código 3.5: Especificação de mudanças no através de semelhança

O código 3.5 demonstra como redefinir um tipo de componente, através do modificador *redefine-component*. Dessa maneira, as alterações são aplicadas a todas as instâncias implantadas.

```
1 redefine-component CDrawing {
2   ports:
3     shapes[0..*] requires IShape;
4   parts:
5     [previous]CDrawing old;
6   connectors:
7     delegator joins shapes to
8       shapes@old;
```

```
9 }
```

Código 3.6: Especificação de mudanças no através de semelhança

Uma outra possibilidade é redefinir o componente sem utilizar operações de diferenciação, como ilustra o código 3.6 [21]. O componente é completamente definido novamente.

Dessa forma, *semelhança* provê um modelo de construção que permite que componentes seja redefinidos em termos de alterações a um componente base, de maneira que o componente original não é afetado. Esta abordagem é semelhante ao conceito de herança, oriunda de orientação a objetos, e favorece o reuso interno (componentes compostos) e externo (sistemas interconectados). *Redefinição* permite que um componente existente seja alterado ou evoluído, e em conjunto com *semelhança* expressa adaptações como alterações na definição anterior. Essas técnicas favorecem a aplicação de mudanças arbitrárias e mantêm um histórico de adaptações, fazendo com que as configurações do sistema sejam rastreáveis.

3.4 Considerações Finais

Neste capítulo foram apresentados os trabalhos relacionados a estes, e como cada um implementa os conceitos apresentados no capítulo 2. Dessa forma, foi apresentado como são especificados componentes, conexões, configurações e sistemas.

O *CONIC* fornece um ambiente de desenvolvimento de componentes baseados numa extensão da linguagem *Pascal*. Neste trabalho a *ADL* encontra-se atrelada de maneira muito próxima à implementação, fazendo com que limite-se a uma única linguagem de programação. Não há também uma linguagem de interface intermediária, fazendo com que toda a comunicação ocorra de maneira direta, a nível de implementação do componente. Já o *Darwin* estende o *CONIC* para provar maior dinamismo arquitetural. Utiliza-se de recursos de instanciação que permitem que o arquiteto projete estruturas inerentemente evolutivas em tempo de design. Apesar disso, as adaptações do *Darwin* são limitadas, visto que remoções não são permitidas (tanto de componentes quanto de conexões) e evoluções arbitrárias não são suportadas. O *Evolve* é o sistema que mais fornece adaptações que favoreçam evolução arbitrária. Os recursos fornecidos favorecem a extensão e alteração de componentes, através de operações de adaptação. Contudo, o *Evolve* ainda não foi testado em um sistema distribuído, nem com evolução dinâmica.

É notável que, apesar de conceitos básicos de arquitetura de software estarem presentes nas *ADL*s mencionadas, estes trabalhos fazem pouca ou nenhuma menção acerca de instalação e implantação distribuída. Ou seja, descrições arquiteturais que dêem suporte a componentes de software com mecanismos de implantação distribuída e evolução dinâmica ainda são pouco exploradas. Neste trabalho, a intenção da ferramenta SCS-DynAdapt é justamente prover suporte a estas funcionalidades.

As idéias do *CONIC* e do *Darwin* de ter controle programático na própria descrição arquitetural foram incorporadas no SCS-DynAdapt. Como a descrição arquitetural é construída sobre uma linguagem intermediária (OMG IDL), é possível utilizar qualquer linguagem com implementação do *middleware* CORBA para fornecer a descrição arquitetural. Dessa maneira, os recursos desta linguagem de programação podem ser utilizados para construir arquiteturas ainda mais elaboradas. O uso de instanciação dinâmica direta de *Darwin* pode ser obtido através de estruturas de repetição e comandos condicionais simples.

Já o *Evolve* influenciou este trabalho na expressão de adaptações inserida na *ADL*. Através deste mecanismo, as alterações que a arquitetura sofre ao longo do tempo ficam claramente documentadas pelo próprio código de descrição. Estas mudanças podem também ser aplicadas em conjunto, de maneira a não deixar o sistema em estado inconsistente em alguns casos. O SCS-DynAdapt ainda guarda estas mudanças, disponibilizando-as ao mantenedor da arquitetura a qualquer momento.

O capítulo 4 a seguir demonstra o funcionamento da ferramenta SCS-DynAdapt, desenvolvida neste trabalho, com mecanismos de especificação e instanciação de sistemas distribuídos baseados em componentes de software SCS.

4
SCS-DynAdapt

A definição e implementação de *ADL*s com capacidades evolutivas apresenta diversos desafios, como demonstrado no capítulo 2. Aspectos como descrição arquitetural, tipagem, rastreabilidade de artefatos, comportamento funcional, tolerância a falhas, independência de linguagem de programação, implantação e alocação de recursos, por exemplo, são chaves na construção de ferramentas flexíveis o suficiente para acomodar os mais variados tipos de construções e evoluções [36].

Este capítulo tem como objetivo apresentar a ferramenta SCS-DynAdapt, principal artefato desenvolvido neste trabalho. A seção 4.1 fornece uma visão geral do funcionamento da ferramenta. A seguir, as seções 4.2, 4.3 e 4.4 mostram como funciona a descrição arquitetural proposta, a implantação remota e a configuração de sistemas distribuídos, respectivamente. A seção 4.5 mostra como foi projetado o mecanismo de ciclo de vida, usado para garantir a consistência na aplicação de adaptações. A seção 4.6 explica como mudanças são inseridas no sistema. A seção 4.7 fornece um exemplo básico de uso da ferramenta, seguida pela seção 4.8 que faz as considerações finais do capítulo.

4.1
Visão Geral

A figura 4.1 fornece uma visão geral acerca do funcionamento da ferramenta desenvolvida. Uma descrição da arquitetura do sistema é dada como entrada para um componente configurador responsável por gerenciar a arquitetura do sistema. Este componente verifica a integridade da descrição (se o estilo arquitetural é compatível com as instâncias ou se alguma dependência não é satisfeita, por exemplo) e efetua a instalação das implementações nos implantadores (*Deployers*), em máquinas físicas distribuídas. Uma vez que todos os componentes estejam implantados corretamente, eles são conectados e inicializados utilizando o SCS. Caso alguma mudança seja requerida, o configurador utiliza os implantadores de maneira a remover/adicionar/atualizar componentes.

Todas estas atividades são realizadas de maneira reativa, através da inte-

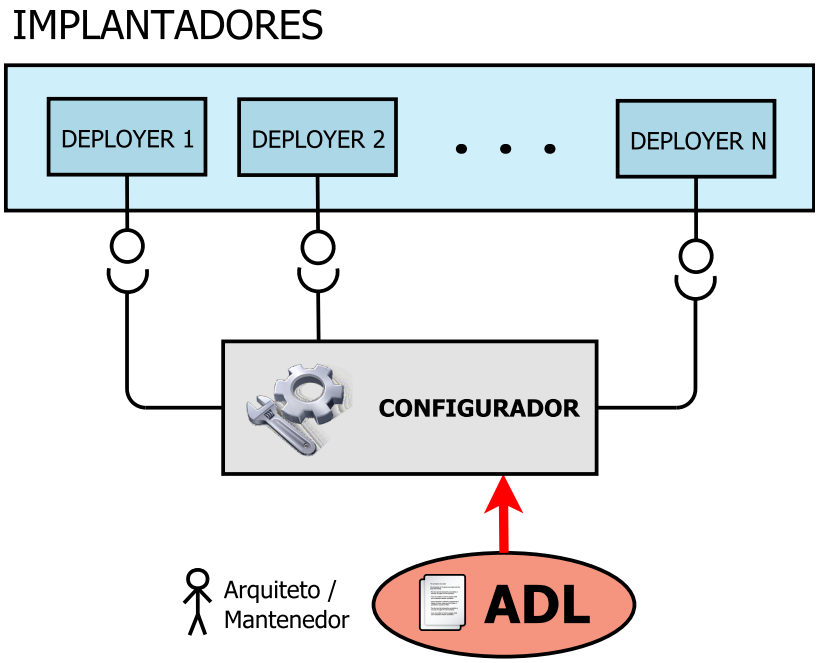


Figura 4.1: Visão Geral do SCS-DynAdapt

ração do arquiteto/mantenedor com o configurador do sistema. É importante ressaltar que existem outras abordagens que atuam de forma *proativa*, como por exemplo sistemas auto-adaptativos, mas este não é o objetivo deste trabalho.

4.2 Descrição Arquitetural

A descrição da arquitetura é implementada através de uma linguagem de definição de interface, mais especificamente OMG IDL [32]. Esta decisão se deu ao fato do sistema de componentes SCS, com um modelo de dados independente de linguagem, utilizar CORBA nos seus serviços. Além disso, com uma linguagem intermediária é possível construir descrições arquiteturais através de qualquer linguagem de programação que consiga se comunicar através de CORBA. Este uso também permite a construção de arquiteturas utilizando recursos típicos de programação, como comandos condicionais e de controle, estruturas de repetição, operações aritméticas, entre outros. O sistema pode ser criado de maneira programática, fazendo do *código* a própria especificação arquitetural. Também não houve a necessidade de construção de *parsers* e compiladores específicos para reconhecer a linguagem do modelo de dados. As sub-seções seguintes demonstram como cada um dos elementos da *ADL* são expressados na IDL.

4.2.1 Estilo arquitetural

Para definir estilos arquiteturais de aplicações distribuídas, primeiramente são criados os tipos que serão associados às abstrações de componentes e relacionamentos. Estes tipos servirão como base para as instâncias definidas posteriormente. O código 4.1 demonstra a estrutura na IDL para definição dos tipos de serviços do sistema.

```
1  // Indicate connection arities of services
2  enum Arity {
3    ONE_TO_ONE, // 1 - 1
4    ONE_TO_MANY, // 1 - n
5    MANY_TO_ONE, // n - 1
6    MANY_TO_MANY, // n - n
7    FREE // No constraints
8  };
9
10 // Service description, without the implementation yet
11 struct Service {
12   string name;
13   string interface_name;
14 };
15 typedef sequence<Service> Services;
```

Código 4.1: Serviços

Um serviço (*Service* no código) é dado através de um nome de identificação e uma interface IDL. A aridade é sempre direcionada do cliente para o servidor, e é definida posteriormente no relacionamento. Também é utilizada na checagem de consistência da arquitetura durante a inicialização do sistema, e podem ser dos tipos:

ONE_TO_ONE 1 cliente conectado a 1 servidor;

ONE_TO_MANY 1 cliente conectado a muitos servidores;

MANY_TO_ONE Muitos clientes conectados a 1 servidor;

MANY_TO_MANY Muitos clientes conectados a muitos servidores;

FREE Sem restrições;

Vale ressaltar aqui que a lógica de controle de requisições encontra-se exclusivamente no código do componente, ou seja, um cliente que esteja conectado a vários servidores tem total controle sobre a escolha de quais dos servidores irá requisitar. Em nenhum momento a ferramenta cria mecanismos para distribuir as requisições e/ou coletar resultados de múltiplas fontes.

Os tipos de dados para descrição de estilos arquiteturais estão definidos no código 4.2:

```
1  // Component template
2  struct ComponentRole {
3    Services provided;
4    Services required;
5    string unique_name;
6  };
7  typedef sequence<ComponentRole> ComponentRoles;
8
9  // Relationship template
10 struct Relationship {
11   ComponentRole client;
12   ComponentRole server;
13   Service service;
14   Arity arity;
15   string unique_name;
16 };
17 typedef sequence<Relationship> Relationships;
18
19 // Architectural style
20 struct Architecture {
21   Relationships relationships;
22   ComponentRoles components;
23 };
```

Código 4.2: Estilo arquitetural

Um componente pode ser definido como uma unidade básica da *ADL* abstrata. A estrutura *ComponentRole* representa um tipo de componente, e é formada por um identificador único, um conjunto de serviços fornecidos (facetas) e outro de requeridos (receptáculos). A necessidade de fornecimento dos serviços requeridos é dado em termo dos relacionamentos criados, ou seja, apesar destes serviços estarem declarados aqui o arquiteto pode optar por não criar ligações para os mesmos.

O relacionamento é dado pela estrutura *Relationship*, com um tipo de componente cliente e um servidor, um nome, um serviço e a aridade do relacionamento. O nome do relacionamento deve ser único no sistema e o serviço, o cliente e o servidor são usados na verificação de tipos de serviço quando as conexões são instanciadas. O cliente deve obrigatoriamente ter o serviço especificado entre os seus receptáculos, e de forma análoga o servidor precisa ter uma faceta declarada nos seus serviços fornecidos. A aridade do relacionamento irá determinar se existe alguma violação de restrição arquitetural.

Por fim o estilo arquitetural (*Architecture*) é determinado pelo conjunto de papéis de componentes e relacionamentos entre eles, de maneira similar a um grafo.

4.2.2 Sistema

Alguns trabalhos incorporam a idéia da definição de sistema como sendo um componente composto de diversos subcomponentes encapsulados em outro, e aqui seguimos esta idéia. Apesar disso o SCS-DynAdapt ainda não suporta composições hierárquicas (componentes encapsulados em componentes de maneira hierárquica).

```
1  // System execution machines config
2  struct Machine {
3    string host; long port;
4    string unique_name;
5  };
6  typedef sequence<Machine> Machines;
7
8  // Component implementation code
9  typedef sequence<octet> Code;
10
11 // Component instance to be deployed
12 struct ComponentInstance {
13   ComponentRole role;
14   string unique_name;
15   Machine machine;
16   Code impl;
17 };
18 typedef sequence<ComponentInstance> ComponentInstances;
19
20 // Connection between two instances
21 struct ConnectionInstance {
22   ComponentInstances clients;
23   ComponentInstances servers;
24   Relationship relationship;
25   string unique_name;
26 };
27 typedef sequence<ConnectionInstance> ConnectionInstances;
28
29 // System, a composite component
30 struct System {
31   ConnectionInstances connections;
32   ComponentInstances components;
33 };
```

Código 4.3: Sistema

De maneira a executar as instâncias de maneira distribuída, é necessário especificar o conjunto de nós de implantação. Esses nós (*Machines*) são definidos por uma identificação de *host*, uma porta de acesso ao ORB remoto e um nome único de identificação. Com essas informações o gerenciador arquitetural pode se conectar ao **Implantador** (*Deployer*) e executar componentes de forma distribuída.

A instância de um componente (*ComponentInstance*) é dada através de um nome único, o papel que ele irá desempenhar na arquitetura (definido no

estilo arquitetural), a máquina onde ele será implantado e o código (*Code*) da sua implementação. Esta estrutura é usada posteriormente para instalar e executar o componente através do implantador.

Uma instância de conexão (*ConnectionInstance*) é composta por componentes físicos da hierarquia. É definida através de um nome, o relacionamento que satisfaz, um conjunto de instâncias de clientes e outro de servidores. O sistema é formado pela configuração de instâncias e conexões resultantes.

Neste trabalho, a ferramenta dá suporte à implantação de componentes de software desenvolvidos na linguagem Lua, que possui funcionalidades de reflexão que permitem que trechos de código sejam carregados dinamicamente, em tempo de execução. A estrutura *Code* é utilizada para transferir códigos Lua através da rede. A partir da utilização de interfaces de controle, a implementação do componente é pode ser carregada para execução nos devidos nós de implantação (através do componente *Implantador*). A função *loadstring* da biblioteca padrão é utilizada para receber um trecho de código em uma *string*, compilá-lo e retornar uma função que executa este trecho. Como cada componente implementa um conjunto de serviços, a estrutura *Code* espera um código Lua que retorne uma tabela com as implementações dos serviços fornecidos pelo componente[1]. Assim, é possível utilizar a função *loadstring* para receber o código do componente e acessar a implementação de cada uma das suas facetas. Como em Lua funções são elementos de primeira ordem, nesta tabela também podem ser incluídas as funções de inicialização (*startup*) e desligamento (*shutdown*) do componente.

```
1  // Architectural Description Language of a system
2  struct ADL {
3    string system_name;
4    Architecture architecture;
5    System system;
6  };
```

Código 4.4: ADL da aplicação

Por fim, a descrição arquitetural é formada por um identificador, um estilo arquitetural e um sistema, como mostra o código 4.4.

[1]Em Lua, tabelas implementam *arrays* associativos, coleções que podem ser indexadas não somente através de números mas também através de outros tipos de dados (como *strings*, por exemplo). Este recurso é utilizado para indexar as implementações de cada faceta.

4.3 Implantador

Na infraestrutura, o **Implantador** (*Deployer*) é um componente responsável pela (re)instalação, remoção, alteração e adição de componentes[2]. É importante notar que este componente não toma nenhuma decisão a partir da IDL, simplesmente recebe comandos do configurador.

```
1 interface IDeployer {
2   void install(in ComponentInstance component_instance) raises (AlreadyRegistered, UnknownService);
3   boolean remove(in string unique_name) raises (NotInstalled);
4   IComponent run(in string unique_name) raises (NotInstalled, NoImplementation);
5   IComponent redeploy(in string unique_name, in Code impl) raises (NotInstalled, RunError);
6 };
```

Código 4.5: Serviço de implantação

O código 4.5 demonstra a IDL do serviço implementada por este componente. O Implantador é responsável por instalar uma determinada instância (*install*), removê-la (*remove*), executá-la (*run*) e atualizá-la (*redeploy*). É importante notar que aqui não existe controle sobre as dependências ou ciclo de vida dos componentes do sistema, este papel é desempenhado pelo Configurador. A instalação envolve adicionar a implementação numa estrutura de dados. A remoção é uma indicação de que o componente será desligado da infraestrutura e que o implantador não mais é responsável por ele (notar que esta funcionalidade é diferente do desligamento).

A execução envolve carregar o código da implementação dos serviços do componente. Como Lua possui capacidades reflexivas, o código recebido é verificado através da função *loadstring*, que compila um trecho de código e o retorna como uma função. Para acessar a tabela com as implementações das facetas, só é preciso executar a função e capturar o valor retornado. Todo este tratamento é feito através de chamadas protegidas, de maneira que o código do componente não afete o funcionamento do implantador. Caso as chaves da tabela não correspondam aos nomes dos serviços fornecidos (facetas), uma exceção é lançada. Os componentes são carregados no mesmo espaço de memória que o implantador, de maneira semelhante a um *container* de componentes. Dessa forma, é possível ter maior controle sobre a interceptação de chamadas feitas ao (ou pelo) componente e aos seus serviços fornecidos e requeridos. Este controle torna possível, por exemplo, desligar as facetas de maneira segura mesmo que o desenvolvedor não implemente o código de desligamento.

[2]O trabalho SCS-DeploySystem desenvolvido em [37] apresenta uma infraestrutura para implantação remota de componentes SCS, porém, infelizmente não possível utilizar tal ferramenta devido a incompatibilidade de versões do SCS

O código recebido na função *redeploy*, que atualiza um dos componentes instalados, deve conter uma tabela com a implementação dos serviços a serem atualizados. Caso a tabela esteja vazia, ou o conteúdo não corresponda a nenhum dos serviços, nada é feito. Isso permite que as partes não afetadas por uma determinada adaptação se mantenham intactas, conceito recorrente em evolução dinâmica de arquiteturas de software. Caso seja interessante, pode-se também substituir um componente completamente e substituí-lo por um outro que implemente os mesmos serviços. Neste caso, o contexto do componente é perdido, obrigando-o a criar um mecanismo de persistência de estado e posterior recuperação, caso seja necessário.

4.4 Configurador

Na sub-seção 2.2.3 foi introduzido o conceito de configuração, e na 2.4.1 o de reconfiguração. O primeiro refere-se à topologia do sistema, enquanto o segundo às mudanças aplicadas nessa topologia. O componente Configurador é responsável por fornecer esses serviços. O código 4.6 explicita a interface IDL implementada por esta unidade:

```
1  interface IArchManager {
2    // Configuration management
3    boolean startSystem(in ADL adl) raises (InvalidADL, StartSystemFailed, UnavailableMachine,
         AlreadyStarted, NoDeployer, InstallError, RunError);
4    boolean addMachine(in Machine machine) raises (MachineAlreadyExists, UnavailableMachine,
         InvalidMachine);
5    boolean shutdownSystem() raises (ShutdownError);
6    boolean shutdownDeployers() raises (UnavailableMachine, NotStarted);
7
8    // Architecture reflection
9    Architecture getArchitectureDescription();
10   System getSystemDescription();
11   IComponent getSystem();
12   Machines getMachines();
13   Adaptation getChanges();
14   string toStringArch();
15
16   // Adapts the system
17   boolean addInstance(in ComponentInstance instance) raises (NonExistentInstance);
18   boolean removeInstance(in string unique_name) raises (NonExistentInstance);
19   boolean addConnection(in ComponentInstance instance) raises (NonExistentInstance);
20   boolean removeConnection(in string unique_name) raises (NonExistentInstance);
21   boolean replaceInstance(in string unique_name, in Code newImpl) raises (NonExistentInstance,
         RunError, NoDeployer);
22   void adapt(in Adaptation adaptation);
23 };
```

Código 4.6: Serviço de configuração

O configurador se encarrega de implantar os componentes nos nós especificados (com ajuda do Implantador), conectá-los e inicializar o sistema

(*startSystem*). Os componentes do sistema são executados uma vez que todas as suas dependências estejam satisfeitas. Neste componente é feito também o controle do ciclo de vida do sistema e dos implantadores. O configurador controla um conjunto de implantadores, que são ligados ou desligados a partir de comandos do mantenedor. Os nós de implantação podem ser adicionados dinamicamente (*addMachine*), e para evitar que o componente *Deployer* tenha de ser executado manualmente, um esquema através do protocolo SSH [38] (*Secure Shell*) foi criado de maneira a facilitar a automatização do processo. Os implantadores ainda podem ser inicializados manualmente, essa foi somente uma alternativa criada.

Uma funcionalidade importante do Configurador é a de analisar e verificar a satisfação de restrições arquiteturais a partir da sua descrição. Caso a especificação não seja válida, por não satisfazer uma dependência ou violar uma aridade, um erro é lançado para o usuário. São também providas funcionalidades de inspeção da descrição da arquitetura (*getArchitectureDescription*) e sistema (*getSystemDescription*). Mesmo após as adaptações, é possível analisar o estado corrente do sistema (*getSystem*) e o conjunto de mudanças aplicadas no sistema (*getChanges*). Através do componente composto, é possível navegar através do conjunto de componentes que compõem o sistema, inspecionando portas e unidades de implementação.

Ainda é possível adaptar a arquitetura tanto através de comandos isolados quanto por mudanças em conjunto (função *adapt*). O usuário pode remover e adicionar elementos através de funções explícitas ou enviá-las em conjunto, para expressar mudanças mais complexas. As funções são responsáveis por adicionar ou remover uma conexão (*addConnection* e *removeConnection*), adicionar ou remover uma instância de componente (*addInstance* e *removeInstance*) ou trocar a implementação de um componente (*replaceInstance*). O funcionamento das estruturas de adaptação e deste mecanismo são explicados na seção 4.6.

4.5 Ciclo de vida

A garantia de *confiabilidade* apresentada na seção 2.3.4 é chave para adaptação dinâmica, pois precisa-se determinar de maneira exata *quando* é seguro aplicar uma reconfiguração [30]. Erros como condição de corrida ou interrupção de requisições em progresso podem ocorrer nesse processo. Isso traz a necessidade de utilizar um estado de *quiescência* dos componentes, uma indicação de que se encontram em condição de serem alterados [10]. Para manter a aplicação de mudanças confiável, foi criado um mecanismo de controle

de requisições CORBA. No nosso contexto, podemos utilizar o *middleware* OiL [39] para interceptar todas as transações dos componentes envolvidos.

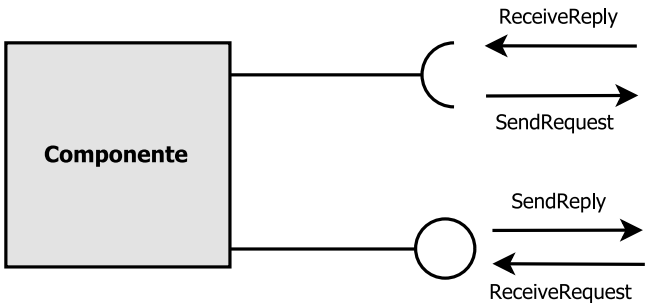


Figura 4.2: Interceptação de chamadas no OiL

A figura 4.2 mostra um componente com uma porta de provisão de serviços e uma de requisição de serviços. O OiL intercepta chamadas através de objetos de interceptação, que são cadastrados em modo cliente (para receptáculos) ou servidor (para facetas). A provisão de serviços é controlada pelos métodos *ReceiveRequest*, executado antes da invocação da requisição, e *SendReply*, executado após a requisição ser completada. Já os serviços requisitados são controlados pelos métodos *SendRequest*, antes da requisição ser despachada, e *ReceiveReply*, após a requisição remota ser respondida. Cada um desses métodos recebe uma tabela Lua com informações acerca da invocação. Esta tabela inclui informações como o objeto remoto sendo invocado (*proxy*), parâmetros e valores de requisição e resposta, a interface e operação associada, entre outras. Três possíveis estados são definidos para os componentes, detalhados a seguir:

RESUMED Estado normal de execução do componente, as chamadas de entrada e saída são repassadas normalmente;

SUSPENDED Componente suspende a sua execução de maneira temporária, enfileirando as chamadas que chegam e saem;

HALTED Nenhuma chamada é processada, o componente simplesmente descarta qualquer requisição, e o requisitante recebe uma exceção.

Desta maneira, pode-se controlar as transações entre os componentes para garantir segurança nas adaptações. A implementação deste controle de estados se deu através da adição de uma faceta aos serviços básicos do SCS. O código 4.7 a seguir demonstra a IDL do controle de estados.

```
1  // Enumeration used to define the possible component states
2  enum State {
3    // Component is running and its services are up
4    RESUMED,
5    //Component is not available to respond requests. Requests are dropped
```

```
6       HALTED,
7       // Component is currently suspended, enqueues calls
8       SUSPENDED
9     };
10
11    // Interface to control component state
12    interface ILifeCycle {
13      // Acquires the current state the component is in
14      State getState();
15
16      //Attempts to change the component state
17      boolean changeState(in State state) raises(CannotChangeState);
18    };
```

Código 4.7: IDL de controle de estados

Os estados são formados em uma enumeração, e a interface de ciclo de vida fornece o acesso ao estado corrente do componente e uma função para alterar o estado. É importante ressaltar como o SCS-DynAdapt implementa a lógica de cada transição de estados:

RESUMED → HALTED Aguarda o término de requisições em processamento e rejeita novas requisições;

RESUMED → SUSPENDED Aguarda o término de requisições em processamento e enfileira novas requisições;

HALTED → SUSPENDED Não há requisições enfileiradas nem em processamento, passa a enfileirar novas requisições;

HALTED → RESUMED Não há requisições enfileiradas nem em processamento, passa a processar novas requisições;

SUSPENDED → HALTED Caso hajam requisições enfileiradas, são todas descartadas e passa a rejeitar novas requisições;

SUSPENDED → RESUMED Caso hajam requisições enfileiradas, estas são todas liberadas para processamento e passa a aceitar novas requisições;

Este conjunto de estados e transições são usados para controlar o mecanismo de adaptação, descrito na seção 4.6.

4.6 Adaptação

As adaptações aplicadas ao sistema através do configurador podem ocorrer através de funções de adição e remoção de instâncias de componentes e conexões. Porém, certas evoluções afetam todo o sistema e não devem ser feitas de maneira pontual, pois podem vir a deixar o sistema em um estado inconsistente. A remoção de relacionamentos, por exemplo, implica na remoção de todas as conexões deste tipo, e se aplicadas de maneira isolada podem deixar um componente com uma dependência não satisfeita. É interessante utilizar uma linguagem que expresse aplicações de diferenciação (*deltas*) em conjunto. No SCS-DynAdapt foram criadas estruturas para este fim. O código 4.8 demonstra a IDL que estrutura as adaptações a nível de estilo arquitetural.

```
1  // Base-level architecture changes
2  struct ArchitectureChange {
3    ComponentRoles newRoles;
4    Relationships removedRelationships;
5    Relationships addedRelationships;
6  };
```

Código 4.8: Adaptações a nível arquitetural

Podem ser introduzidos novos tipos de componentes (*newRoles*) e relacionamentos podem ser removidos (*removedRelationships*) ou adicionados (*addedRelationships*). A remoção de um relacionamento deve implicar na remoção das instâncias de conexão deste tipo, que devem estar especificadas nas adaptações do sistema. Com novos papéis de componentes introduzidos, estes podem ser instanciados de maneira a criar novas arquiteturas.

O código 4.9 demonstra as estruturas de adaptações de sistema, a nível de instâncias distribuídas.

```
1  // Changes to component instances
2  struct ComponentInstanceChange {
3    ComponentInstance previous;
4    Code newImpl;
5  };
6  typedef sequence<ComponentInstanceChange> ComponentInstancesChanges;
7
8  // System changes
9  struct SystemChange {
10   ComponentInstancesChanges componentChanges;
11   ComponentInstances addedComponents;
12   ComponentInstances removedComponents;
13   ConnectionInstances addedConnections;
14   ConnectionInstances removedConnections;
15 };
16
17 // System and architecture adaptation
18 struct Adaptation {
19   ArchitectureChange architectureChange;
```

```
20      SystemChange systemChange;
21  };
```

Código 4.9: Adaptações de sistema

É possível adicionar (*addedComponents*, *addedConnections*) ou remover (*removedComponents*, *removedConnections*) componentes e conexões instanciadas, assim como aplicar alterações a componentes em execução (*componentChanges*). O algoritmo para aplicação de mudanças é descrito a seguir:

1. Verifica-se se as mudanças não violam restrições arquiteturais. Como o objetivo é fornecer suporte a mudanças o mais arbitrárias possível, as próprias restrições podem ser alteradas;

2. Componentes implantados a serem modificados ou reconectados têm seus estados alterados para *SUSPENDED*, assim como os componentes adjacentes a estes;

3. Componentes a serem removidos tem seus estados alterados para *HALTED* e são desconectados e desligados (*shutdown*). Os componentes adjacentes são suspensos até que as modificações estejam completas;

4. Novas instâncias de componentes são instaladas nos implantadores distribuídos;

5. Componentes são alterados;

6. Conexões são feitas e/ou desfeitas;

7. Inicializa novos componentes (*startup*) e altera o estado dos demais para *RESUMED*;

A estrutura *componentChanges* é usada para alterar componentes sem desligá-los. Para este caso, o componente é primeiro suspenso, enfileirando novas requisições. Um código Lua é fornecido, retornando uma tabela com a implementação de cada serviço a ser alterado. Os objetos remotos afetados têm a implementação trocada através do uso do OiL. Como Lua tem capacidades de reflexão, foi desenvolvida durante esta mudança um tratamento de tolerância a falhas, tanto para códigos que não compilam corretamente quanto para os que não podem ser instanciados por não implementar corretamente uma interface. Caso algum desses erros ocorra, a implementação anterior é reimplantada.

Para controlar a entrada e saída de novos componentes, o gerenciador arquitetural utiliza os critérios de controle de estados propostos por Kramer e Magee [8].

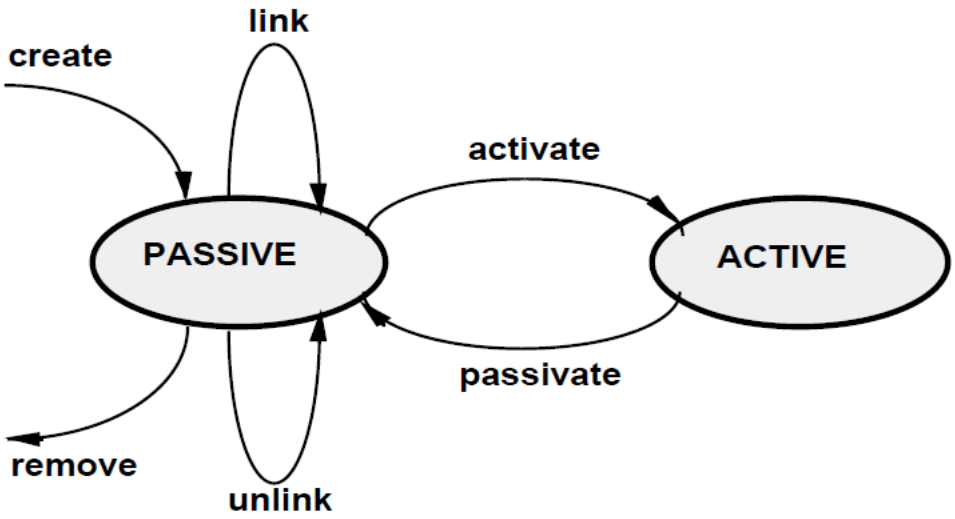


Figura 4.3: Ciclo de vida de componentes

A figura 4.3 (retirada de [8]) demonstra como funciona a transição de estados de componentes do sistema do ponto de vista da configuração. Componentes são criados (*create*) de maneira que inicialmente estão com estado passivo (*Passive*). Após serem devidamente conectados (*link*), passam ao estado ativo (*Active*), onde podem fazer e receber requisições. De maneira análoga, para remover um componente ativo primeiramente este sofre transição para o estado passivo, as ligações são desfeitas (*unlink*) e o componente é desligado da arquitetura (*remove*). A figura 4.4 mostra como essas idéias foram estendidas para atender o modelo adotado no SCS-DynAdapt:

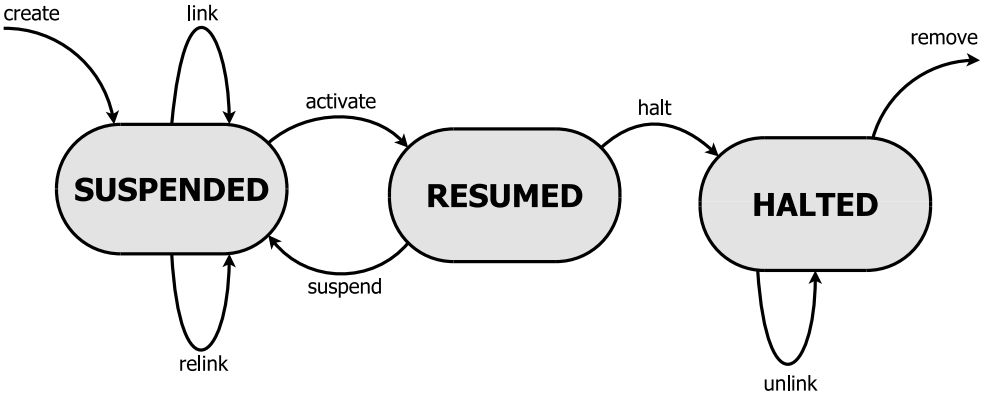


Figura 4.4: Ciclo de vida de componentes no SCS-DynAdapt

O estado *Passive* foi dividido em dois, o suspenso (*SUSPENDED*) e o parado (*HALTED*). O comando *passivate* foi dividido em *suspend* e *halt*, para suspender e parar o componente, respectivamente. Além disso, como componentes podem ser reconectados quando estão suspensos, foi adicionada a funcionalidade de *relink* para este propósito. Componentes parados são eventualmente desconectados e removidos do sistema (*unlink* e *remove*).

4.7 Exemplo

Para ilustar o uso dos mecanismos descritos iremos utilizar inicialmente um exemplo simples. Vamos criar uma estrutura cliente-servidor com um componente provendo um serviço e outros consumindo o serviço fornecido.

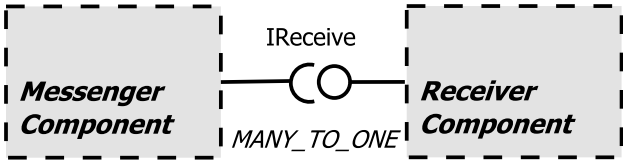


Figura 4.5: Estilo arquitetural *Messenger*, cliente-servidor

A figura 4.5 demonstra o exemplo com um componente do tipo *MessengerComponent* e outro do tipo *ReceiverComponent*. O primeiro requer um serviço do tipo *IReceive* fornecido pelo segundo. O componente *MessengerComponent* envia mensagens advindas da entrada padrão do cliente, enquanto o *ReceiverComponente* recebe tais mensagens e as reproduz na saída padrão do servidor. A aridade do relacionamento é de muitos-para-um, dado que múltiplos clientes podem estar conectados ao servidor. O uso da linha tracejada em volta dos componentes foi proposital, significando que estes são apenas *templates*, não existe ainda uma implementação concreta. O código 4.10 demonstra a interface IDL do serviço *IReceive*.

```
1  interface IReceive {
2    void receiveMessage(in string str);
3  };
```

Código 4.10: Interface *IReceiver*

O código 4.11 demonstra o código de implementação dos componentes. O componente *MessengerComponent* tenta evitar erros ao checar o seu estado antes de enviar mensagens, e fica inativo por um certo período antes de tentar novamente.

```
1  -- Receiver Implementation
2  local Receiver = oo.class{name = "MessageReceiver"}
3  function Receiver:receiveMessage(str)
4    io.write(str)
5  end
6
7  -- Messenger Implementation
8  local Messenger = oo.class{name = "MessageSender"}
9  function Messenger:send()
10   while(true) do
11     local msg = io.read()
12     if (lifeCycle:getState() ~= "SUSPENDED") then
13       receiver:receiveMessage(msg)
14     else
```

```
15          oil.sleep(SLEEP_PERIOD)
16        end
17      end
18    end
```

Código 4.11: Implementação do *MessengerComponent*

4.7.1 Descrição Arquitetural

O código 4.12 demonstra como o exemplo pode ter a arquitetura especificada.

```
1   -- Defining the type of the service
2   ReceiverService = {
3     name = "IReceiver",
4     interface_name = "IDL:messenger/IReceiver:1.0"
5   }
6
7   -- Definig the Receiver role
8   ReceiverComponent = {
9     provided = { ReceiverService },
10    required = {},
11    unique_name = "ReceiverRole"
12  }
13
14  -- Definig the role of a component to use the Receiver service
15  MessengerComponent = {
16    provided = {},
17    required = { ReceiverService },
18    unique_name = "MessengerRole"
19  }
20
21  ReceiverRelationship = {
22    client = MessengerComponent,
23    server = ReceiverComponent,
24    service = ReceiverService,
25    arity = "MANY_TO_ONE",
26    unique_name = "ReceiverRelationship"
27  }
```

Código 4.12: Arquitetura do sistema *Messenger*

O serviço é definido através da tabela Lua *ReceiverService*, juntamente com a interface IDL e uma aridade de muitos-para-um. O papel do componente *ReceiverComponent* é definido com um serviço fornecido e o do componente *MessengerComponent* com um serviço requerido do tipo *ReceiverService*. A seguir, é criado um relacionamento entre esses dois componentes (*ReceiverRelationship*). Agora é preciso definir as instâncias do sistema, em função dos tipos explicitados (código 4.13).

```
1   ReceiverInstance = {
2     role = ReceiverComponent,
3     machine = machines.ubuntu2,
```

```
4       unique_name = "ReceiverInstance",
5       impl = io.open("src/tests/messenger/receiver.lua", "r"):read("*a")
6     }
7
8     MessengerInstance = {
9       role = MessengerComponent,
10      machine = machines.ubuntu2,
11      unique_name = "MessengerInstance",
12      impl = io.open("src/tests/messenger/messenger.lua", "r"):read("*a")
13    }
14
15    ReceiverConnection = {
16      clients = { MessengerInstance },
17      servers = { ReceiverInstance },
18      relationship = ReceiverRelationship,
19      unique_name = "ReceiverConnection"
20    }
```

Código 4.13: Instâncias do sistema *Messenger*

Uma instância do tipo *ReceiverComponent* é declarada na tabela *ReceiverInstance* e uma do tipo *MessengerComponent* é declarada na tabela *MessengerInstance*, com ambas sendo implantadas na máquina física "ubuntu2" (no exemplo esta variável é uma referência para uma tabela omitida). Uma instância do relacionamento *ReceiverRelationship* é declarada pela tabela *ReceiverConnection*. É importante notar que o campo *impl* das instâncias é na verdade o conteúdo do arquivo de implementação de cada componente, em *string*. Desta maneira, é possível estruturar a descrição da arquitetura, como mostra o código 4.14.

```
1   ADL = {
2     -- Name to be placed as the system composite component name
3     system_name = "MessengerArch",
4
5     -- Architectural style
6     architecture = {
7       relationships = { ReceiverRelationship },
8       components = { ReceiverComponent, MessengerComponent }
9     },
10
11    -- System instance
12    system = {
13      connections = { ReceiverConnection },
14      components = { ReceiverInstance, MessengerInstance }
15    }
16  }
```

Código 4.14: Descrição arquitetural do sistema *Messenger*

O passo seguinte envolve utilizar o *Configurador* para implantar a arquitetura e inicializar o sistema. Primeiramente é verificado se todas as restrições arquiteturais se encontram satisfeitas pela instância e se as máquinas de implantação estão todas funcionando corretamente. Em seguida, as imple-

mentações são instaladas nos nós correspondentes e inicializados com estado *SUSPENDED*. O *Configurador* então conecta todas as instâncias e altera os seus estados para *RESUMED*.

A seção a seguir ilustra um cenário de aplicação de adaptações para modificar o sistema e a arquitetura.

4.7.2 Reconfiguração

No cenário de adaptação, iremos introduzir um componente de filtro (*FilterComponent*) que figure entre os dois especificados, de maneira a filtrar algumas mensagens enviadas pelo *MessengerComponent*. A figura 4.6 demonstra a nova configuração da arquitetura com a introdução do novo componente.

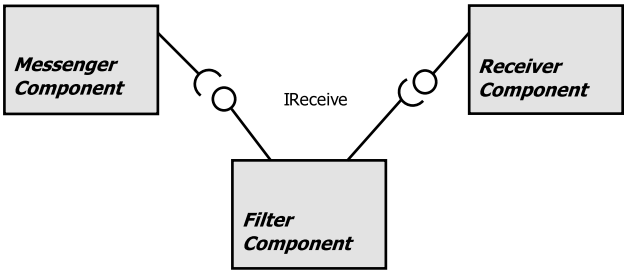


Figura 4.6: Nova arquitetura *Messenger*

O código 4.15 contêm a implementação do componente *FilterComponent*.

```
1  -- Filter implementation
2  local Filter = oo.class{name = "MessageFilter"}
3  function Filter:receiveMessage(str)
4    local orb = oil.orb
5
6    -- Apply filter
7    if (not self:shouldPass(str)) then return end
8
9    -- Simply forwards the call
10   receiver:receiveMessage(str)
11 end
```

Código 4.15: Implementação do componente de filtro

Na descrição, é necessário primeiro alterar o estilo arquitetural para adicionar o componente filtro e os seus relacionamentos. O código 4.16 demonstra como especificar essas mudanças na arquitetura:

```
1  -- Definig the new role, Filter
2  FilterRole = {
3    provided = { ReceiverService },
4    required = { ReceiverService },
5    unique_name = "FilterRole"
6  }
7
8  -- Relationship with ReceiverComponent
```

```
ReceiverFilterRelationship = {
  client = FilterRole,
  server = ReceiverRole,
  service = ReceiverService,
  arity = "ONE_TO_ONE",
  unique_name = "ReceiverFilterRelationship"
}

-- Relationship with MessengerComponent
MessengerFilterRelationship = {
  client = SayRole,
  server = FilterRole,
  service = ReceiverService,
  arity = "MANY_TO_ONE",
  unique_name = "MessengerFilterRelationship"
}
```

Código 4.16: Implementação do *FilterComponent*

O tipo do componente filtro é adicionado, com uma faceta e um receptáculo do mesmo tipo de serviço, *ReceiverService*. Em seguida, as instâncias anteriores precisam de novas conexões e o componente filtro precisa de uma nova instância, como mostra o código 4.17.

```
FilterInstance = {
  role = FilterRole,
  machine = machines.ubuntu2,
  unique_name = "FilterComponent",
  impl = io.open(ARCH_HOME .. "src/tests/helloFilter/filter.lua", "r"):read("*a")
}

ReceiverFilterConnection = {
  clients = { FilterInstance },
  servers = { ReceiverInstance },
  relationship = ReceiverFilterRelationship,
  unique_name = "ReceiverFilterConnection"
}

MessengerFilterConnection = {
  clients = { MessengerInstance },
  servers = { FilterInstance },
  relationship = MessengerFilterRelationship,
  unique_name = "MessengerFilterConnection"
}
```

Código 4.17: Implementação do *FilterComponent*

O relacionamento *ReceiverRelationship* , definido inicialmente, precisa ser removido de maneira a não gerar erros de dependência dos componentes. A *adaptação* é especificada de acordo com o código 4.18:

```
adaptation = {
  architectureChange = {
    newRoles = { FilterRole },
    removed = { ReceiverRelationship },
    added = { ReceiverFilterRelationship, MessengerFilterRelationship },
  },
```

```
7     systemChange = {
8       componentChanges = { },
9       addedComponents = { FilterInstance },
10      removedComponents = { },
11      addedConnections = { ReceiverFilterConnection, MessengerFilterConnection }
12    }
13  }
```

Código 4.18: Implementação do *FilterComponent*

É importante notar que como o relacionamento foi removido, todas as instâncias deste tipo também precisam ser removidas. O relacionamento anterior é removido através da tabela *removed*, enquanto os novos são especificados pela tabela *added*. A nova instância do filtro é fornecida na tabela *addedComponents* e as conexões removidas e adicionadas são dadas nas tabelas *removedConnections* e *addedConnections*, respectivamente. A partir deste momento o configurador toma as seguintes ações:

1. Verifica se as mudanças não quebram regras da arquitetura;
2. Suspende as instâncias dos componentes envolvidos;
3. Instala e inicializa (suspendido) o novo componente filtro (especificado na tabela *addedComponents*);
4. Desfaz a conexão *ReceiverConnection*;
5. Adiciona as conexões *MessengerFilterConnection* e *ReceiverFilterConnection* (especificadas em *addedConnections*);
6. Inicializa o novo componente filtro (*startup*) e altera o estado dos componentes suspensos para *RESUMED*;

Com isso, os componentes voltam a efetuar as requisições normalmente com o sistema em um novo estado arquitetural.

4.8 Considerações Finais

Este capítulo apresentou a ferramenta SCS-DynAdapt, assim como o seu funcionamento, seus principais componentes internos e sua aplicação através de Lua. O autor acredita que as escolhas que permeiam a construção dos elementos de descrição seguem a abordagem usual apresentada na literatura, com a adição de mecanismos que possibilitam implantação remota. É possível definir componentes, conexões, configurações e sistemas, baseados em componentes SCS.

O sistema distribuído é especificado através de uma linguagem intermediária, a OMG IDL. Assim, qualquer linguagem de programação que tenha implementação do *middleware* CORBA pode especificar a arquitetura de maneira remota. Isso também traz a possibilidade de criação de estruturas ainda mais dinâmicas, podendo a descrição arquitetural estar atrelada a ferramentas providas pelas linguagens de programação, como recursão ou estruturas de repetição. A partir da descrição, a ferramenta é capaz de instanciar o sistema distribuído de maneira automática, implantando os componentes nos nós da infraestrutura através dos implantadores.

Componentes SCS são estendidos para acomodarem também a faceta de controle de estados *ILifeCycle* (explicitada no apêndice A.1). Através do auxílio do OiL no controle sobre as requisições do componente, é possível, em conjunto com essa interface, prover segurança na aplicação de adaptações.

As mudanças aplicadas à arquitetura são também especificadas sobre a OMG IDL. É possível adaptar o sistema tanto pelo uso de funções de manipulação (remoção/adição/alteração de componentes e conexões) quanto pela aplicação em conjunto de mudanças mais complexas. A ferramenta é capaz de verificar se estas alterações violam as restrições arquiteturais, e em caso positivo lança erros e não deixa os componentes em estado inconsistente. A aplicação de mudanças é feita com o auxílio da interface *ILifeCycle*, de maneira a interromper minimamente o sistema.

O capítulo 5 a seguir demonstra a utilização da ferramenta na aplicação distribuída CAS [22], mostrando como a ferramenta pode ser aplicada em cenários de inicialização e adaptação do sistema.

5
Exemplo de Uso

Este capítulo apresenta um exemplo de uso sobre um sistema distribuído, o CAS (*Capture and Access System*) [22]. O projeto consiste em uma infraestrutura de gravação distribuída e extensível de *Captura e Acesso*. Tem como objetivo *capturar* experiências sociais (como apresentações, reuniões, aulas, teleconferências, salas de vigilância, etc) ao vivo, extraindo informações contextuais para serem posteriormente *acessadas* na forma de um documento multimídia. É uma iniciativa conduzida pelo TecGraf/PUC-Rio, tendo recebido apoio inicialmente da Petrobras e da *Microsoft Research*. O seu funcionamento está atrelado ao SCS e ao OpenBus [40], um barramento de comunicação para integração de aplicações distribuídas baseadas em componentes de software. Os dispositivos de captura (notebooks, tablets, desktops, smartphones, etc) são agrupados em salas onde são realizadas as atividades.

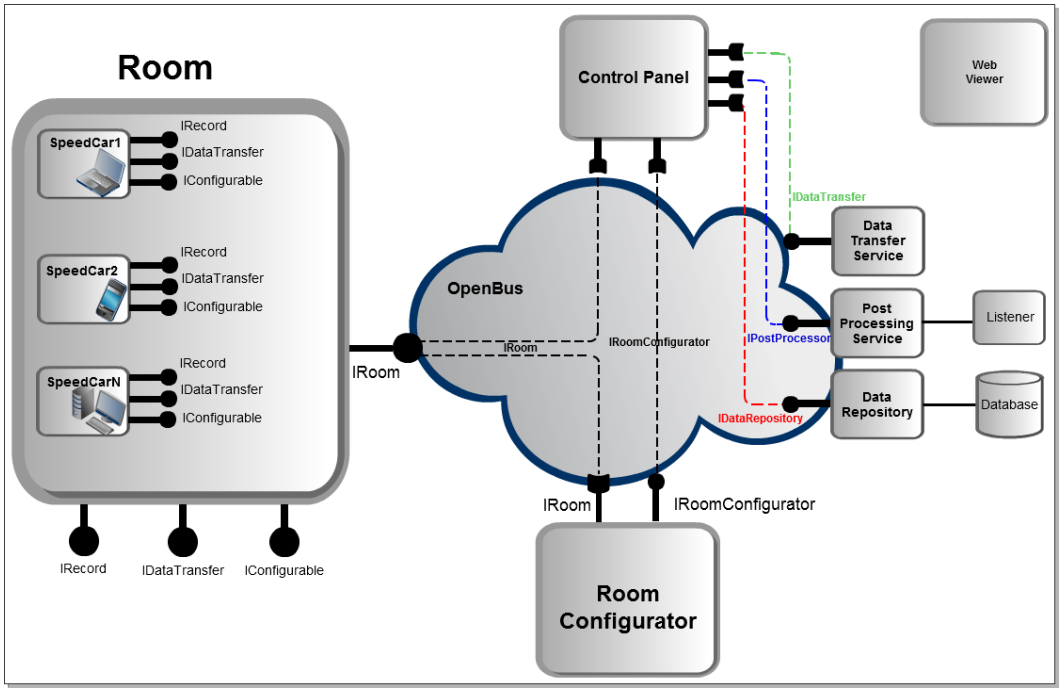


Figura 5.1: Arquitetura do CAS

A figura 5.1 ilustra a arquitetura do sistema, e seus componentes são descritos a seguir:

SpeedCar (SPEcializED CApture driverR) Dispositivos de captura de mídias. Funcionam *drivers* específicos de captura (apresentações, áudio, vídeo, texto) e implementam as seguintes interfaces:

IRecord Interface de gravação genérica. Contém funções de controle da gravação assim como de acesso ao status do dispositivo;

IDataTransfer Interface de transferências de mídias capturadas. As mídias variam de acordo com o tipo de *SpeedCar*, podem ser arquivos de áudio (mp3, ogg), de vídeo (mkv, mpeg), entre outros.;

IConfigurable Interface de configuração (*set*/*get*) de propriedades específicas de tipos de *SpeedCars*. Na captura de vídeo, por exemplo, pode ser importante configurar a qualidade do vídeo, ou resolução, uso de *anti-aliasing*, etc.

Room Representa uma sala de capturas de eventos. Os eventos de ínicio/-término de gravação são recebidos pela sala que repassa aos *SpeedCars*, caso seja apropriado. Desta maneira, também implementa as facetas do *SpeedCar* e se comunica diretamente com estes componentes. Além disso, é responsável por atribuir valor a certas propriedades específicas necessárias para o funcionamento da infraestrutura geral, através da faceta *IConfigurable*;

Post-Processor Componente utilizado para processar as mídias resultantes após as gravações. Arquivos de vídeos podem ter resolução ajustada, os de apresentação podem ter informações de autoria inseridas e os de áudio podem ter tratamentos orientados a ambientes de gravação, por exemplo;

RoomConfigurator Este componente é utilizado com o propósito de configurar as conexões dos componentes durante a inicialização e delisgamento do sistema;

ControlPanel O painel de controle, uma interface gráfica de acesso às salas e controle de gravação geral.

Existe também um módulo de visualização de eventos (*Web Viewer*) e um repositório de dados (*Data Repository*). A seção 5.1 apresenta como foi feita a modelagem do sistema utilizando mecanismos de composição hierárquica, com o SCS-Composite [1]. A seção 5.2 mostra como modelamos o CAS com o SCS-DynAdapt e como este alivia certas dificuldades. A seção 5.3 demonstra cenários de reconfiguração arquitetural, seguida da seção 5.4 que demonstra como corrigir defeitos de implementação através de adaptação dinâmica.

5.1 Modelagem do CAS com SCS-Composite

Inicialmente, o CAS foi desenvolvido utilizando somente componentes monolíticos, sem composições. Contudo a abordagem era conceitualmente próxima da de componentes compostos. Em [1] foi desenvolvida uma modificação na implementação para utilizar o SCS-Composite, removendo esta brecha de expressividade. O componente *Room* na figura 5.1 ilustra esta composição: os *SpeedCars* ficam encapsulados nas salas, escondendo a complexidade da implementação. Dessa maneira, para iniciar/terminar eventos de gravação, só é preciso invocar o serviço *IRecord* oferecido pela sala, e não pelos *drivers* individuais. Ou seja, a lógica de quais *SpeedCars* pertencem a qual sala fica totalmente abstraída no componente *Room*.

Nesta implementação, foi utilizado o conceito de *binding* vertical, aonde o componente *Room* encapsula os *SpeedCars* e expõe as interfaces de maneira coletiva através de um conector.

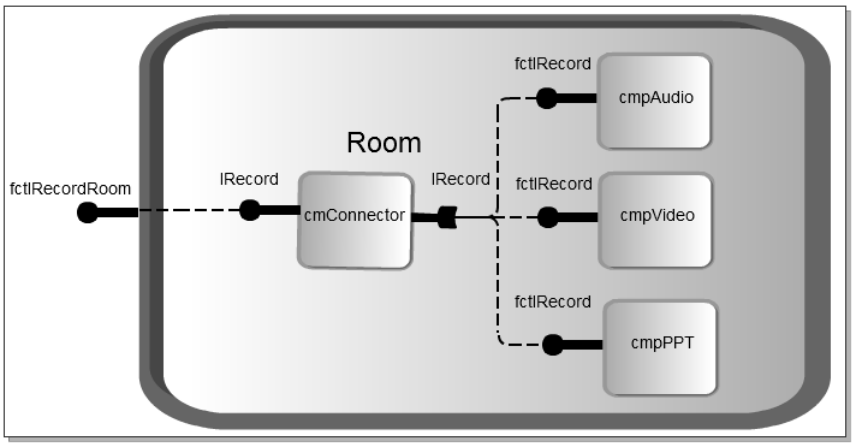


Figura 5.2: Sala utilizando conector

A figura 5.2 (retirada de [1]) ilustra o uso de um conector para acessar facetas internas. Requisições feitas à faceta externalizada *fctIRecordRoom* são repassadas aos *SpeedCars* internos. O controle então passa do componente para o conector.

5.2 Modelagem do CAS com SCS-DynAdapt

Apesar da abordagem anterior favorecer a composição, ainda existem algumas dificuldades na utilização do sistema. Nesta seção, iremos apresentar tais dificuldades e modelar a arquitetura do CAS demonstrando como a ferramenta proposta facilita a inicialização do sistema, corrige possíveis falhas e introduz evoluções de maneira mais natural.

5.2.1 Descrição Arquitetural e Inicialização do Sistema

Considerando os componentes *Room* e *SpeedCar* como um sistema (componente composto), queremos criar adaptações em função desta composição. Para fins de simplificação dos exemplos, apenas estes dois tipos de componentes serão considerados nas adaptações. Na abordagem do SCS-Composite, para inicializar o sistema, o CAS se utiliza do componente configurador *RoomConfigurator*. Os passos de inicialização são:

1. Componente *Room* é executado e cria uma oferta no OpenBus, com um nome como propriedade de identificação;

2. Componente *RoomConfigurator* é executado, e busca a sala com o nome especificado;

3. Componentes *SpeedCar* são executados buscando o configurador, e executam a função *connectComponent* da interface *IRoomConfigurator*, de maneira a serem conectados com o componente *Room* e devidamente configurados com propriedades do sistema;

4. Uma vez que a configuração mínima da sala seja atingida, o componente *Room* passa ao estado pronto (*Ready*), indicando que já se encontra hábil para efetuar gravações.

Como visto, a lógica de conexão envolve que os componentes de captura, ao se incializarem, se conectem ao OpenBus e busquem o configurador da sala à qual devem se conectar. Essa abordagem é falha na medida em que o conhecimento acerca da conexão dos componentes se encontra espalhada pelo *RoomConfigurator* e *SpeedCars*, quando deveria estar em um único lugar, de acordo com o princípio de separação de interesses (do inglês *separation of concerns*).

O uso da descrição arquitetural para fins de configuração alivia este problema, já que a lógica de conexão se encontra toda na descrição arquitetural do sistema. Além disso, como o objetivo do *RoomConfigurator* é exclusivamente conectar e desconectar os *SpeedCars* na inicialização, este componente não é mais necessário na arquitetura.

A figura 5.3 ilustra a composição de componentes proposta. Os relacionamentos possuem aridade *ONE_TO_MANY* pois o componente *Room* pode estar conectado a diversos *SpeedCars* para gravar um evento. Além disso, o componente *Room* também exporta as três interfaces atuando de maneira similar a um *proxy*. Apesar de redirecionar chamadas, a sala também possui uma

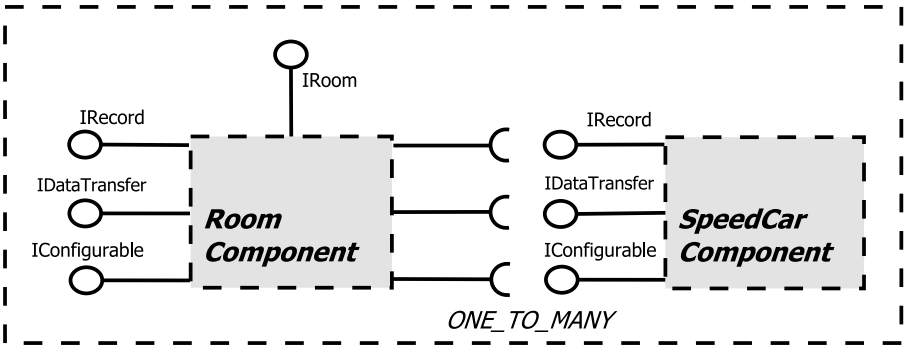


Figura 5.3: Novo estilo arquitetural do CAS

certa inteligência no controle das gravações. Um dos *SpeedCars*, por exemplo, pode estar sendo inicializado ou não estar completamente configurado para iniciar uma gravação. Estas situações são devidamente captadas e tratadas pela sala.

Para modelar o CAS utilizando a abordagem desenvolvida, é necessário primeiramente especificar os serviços que serão utilizados ao longo do ciclo de vida.

```
1  RoomService = {
2    name = "IRoom",              interface_name = "IDL:cas/room/IRoom:1.0"
3  }
4
5  RecordService = {
6    name = "IRecord",             interface_name = "IDL:cas/recorder/IRecord:1.0"
7  }
8
9  ConfigurationService = {
10   name = "IConfigurable",           interface_name = "IDL:cas/configuration/IConfigurable:1.0"
11 }
12
13 DataTransferService = {
14   name = "IDataTransfer",            interface_name = "IDL:cas/transfer/IDataTransfer:1.0"
15 }
```

Código 5.1: Serviços do CAS

O código 5.1 demonstra os serviços fornecidos no CAS pelos componentes *Room* e *SpeedCar*.

```
1  SpeedCarComponent = {
2    provided = { RecordService, ConfigurationService, DataTransferService },
3    required = {},
4    unique_name = "SpeedCarRole"
5  }
6
7  RoomComponent = {
8    provided = { RoomService, RecordService, ConfigurationService, DataTransferService },
9    required = { RecordService, ConfigurationService, DataTransferService },
10   unique_name = "RoomRole"
11 }
```

Código 5.2: Componentes do CAS

O código 5.2 mostra os serviços providos e requeridos de componentes do tipo *SpeedCar* e *Room*. Como indicado na figura 5.3, o componente *Room* implementa todos os serviços do código 5.1, enquanto o *SpeedCar* implementa os de gravação, configuração e transferência de dados.

```
RoomSCRecord = {
  client = RoomComponent,          server = SpeedCarComponent,
  service = RecordService,         arity = ONE_TO_MANY,
  unique_name = "RoomRecord"
}

RoomSCDataTransfer = {
  client = RoomComponent,          server = SpeedCarComponent,
  service = DataTransferService,       arity = ONE_TO_MANY,
  unique_name = "RoomSCDataTransfer"
}

RoomSCConfiguration = {
  client = RoomComponent,          server = SpeedCarComponent,
  service = ConfigurationService,      arity = ONE_TO_MANY,
  unique_name = "RoomSCConfiguration"
}
```

Código 5.3: Relacionamentos do CAS

O código 5.3 apresenta a modelagem dos relacionamentos. É criado um relacionamento para cada ligação entre a sala e o *SpeedCar*, todas com aridade *ONE_TO_MANY*. Aqui também é definido que os *SpeedCars* são considerados provedores do serviço do relacionamento (*server*), enquanto as salas são consumidores (*client*).

```
Room = {
  role = RoomComponent,          machine = machines.CentOS1,
  unique_name = "Presentation␣Room",
  impl = io.open("src/tests/CAS/room.lua", "r"):read("*a")
}

SC1 = {
  role = SpeedCarComponent,          machine = machines.CentOS2,
  unique_name = "SpeedCar1",
  impl = io.open("src/tests/CAS/sc1.lua", "r"):read("*a")
}

SC2 = {
  role = SpeedCarComponent,          machine = machines.Ubuntu1,
  unique_name = "SpeedCar2",
  impl = io.open("src/tests/CAS/sc2.lua", "r"):read("*a")
}

SC3 = {
  role = SpeedCarComponent,          machine = machines.LinuxMint,
  unique_name = "SpeedCar3",
```

```
22    impl = io.open("src/tests/CAS/sc3.lua", "r"):read("*a")
23  }
```

Código 5.4: Instâncias de componentes do CAS

O código 5.4 demonstra um exemplo com uma instância do componente *Room* e três *SpeedCars* (*SC1*, *SC2* e *SC3*). Estas instâncias são descritas de maneira a serem implantadas em diferentes nós (*CentOS1*, *CentOS2*, *Ubuntu1*, *LinuxMint*), cada qual com uma implementação distinta (*impl*). Como as dependências ainda não estão satisfeitas, é preciso criar as instâncias de conexões.

```
1   RoomSCDT = {
2     clients = { Room },
3     servers = { SC1, SC2, SC3 },
4     relationship = RoomSCDataTransfer,          unique_name = "RoomSCDT"
5   }
6
7   RoomSCR = {
8     clients = { Room },
9     servers = { SC1, SC2, SC3 },
10    relationship = RoomSCRecord,          unique_name = "RoomSCR"
11  }
12
13  RoomSCC = {
14    clients = { Room },
15    servers = { SC1, SC2, SC3 },
16    relationship = RoomSCConfiguration,          unique_name = "RoomSCC"
17  }
```

Código 5.5: Instâncias de conexões do CAS

As conexões são criadas no código 5.5, com a criação de ligações entre o componente *Room* e cada um dos *SpeedCars*. Cada uma das instâncias de conexão (*RoomSCDT*, *RoomSCR* e *RoomSCC*) é criada especificando o relacionamento que satisfaz e os consumidores/provedores do serviço (em termos de instâncias de componentes).

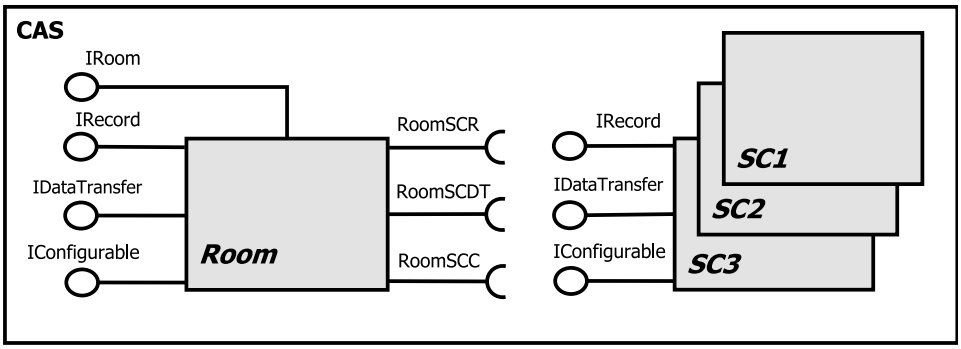


Figura 5.4: Instâncias na nova modelagem

A figura 5.4 demonstra a configuração das instâncias do sistema após a implantação. O componente *Room* possui três receptáculos múltiplos, cada um conectado a uma das facetas de tipo correspondente do *SpeedCar*.

Desta maneira, a inicialização do CAS tem a sua lógica de conexão toda movida dos *SpeedCars* e *RoomConfigurator* para a descrição arquitetural. Uma vez que o sistema seja executado, o componente *Room* já pode fazer requisições de gravação, sem necessidade do *RoomConfigurator*. O desligamento do sistema também pode ser feito através da chamada à função *shutdownSystem* do Configurador arquitetural (apresentado na seção 4.6), que desconecta os *SpeedCars* e o método de finalização dos mesmos.

5.3 Reconfiguração da sala

Como cenários de reconfigurações, temos dois exemplos a serem mostrados. O primeiro caso envolve a adição e remoção de dispositivos (smartphones, tablets) de gravação ao longo do tempo, quando por exemplo, uma pessoa entra ou sai da sala de gravação. Desta maneira, estaríamos adaptando somente as instâncias da arquitetura. O segundo exemplo a ser mostrado utiliza-se de modificações no estilo arquitetural, ou seja, altera-se a base da arquitetura da aplicação. Dar suporte a este tipo de adaptação é importante para prover mudanças arbitrárias, não previstas em tempo de design.

5.3.1 Manipulações de instâncias

Nesta seção exploramos a manipulação das instâncias já implantadas na infraestrutura. Assim, essencialmente, queremos desenvolver casos de adição e remoção de instâncias de componentes e conexões da arquitetura. A adição de um *SpeedCar* de uma sala envolve então:

1. Instalar o novo *SpeedCar* numa máquina de implantação;
2. Resolver as dependências do componente, através da conexão de suas facetas e receptáculos. No caso do *SpeedCar*, existe somente uma conexão a ser feita com o receptáculo da sala;
3. Mudar o estado do componente para *RESUMED* e iniciar a sua execução por invocar o método *IComponent:startup*.

De maneira análoga, a remoção de um *SpeedCar* envolve:

1. Trocar o estado do *SpeedCar* para *HALTED*, indicando que este não se encontra mais disponível para servir requisições;

2. Desfazer todas as conexões do componente. No caso do *SpeedCar* envolve remover somente a conexão com o componente *Room*;

3. Executar o método *IComponent:shutdown* de modo que o *SpeedCar* possa se desligar.

As adaptações citadas podem ser feitas de duas maneiras: a primeira é utilizando o serviço direto do configurador com os métodos de manipulação de instâncias de componentes e conexões do código 4.6 (*addInstance*, *removeInstance*, *addConnection*, *removeConnection*). Uma outra opção é utilizar a estrutura de adaptação (*Adaptation*), presente no mesmo código.

```
1  local archComponent = orb:newproxy("corbaloc:iiop:192.168.1.210:8021/ArchManager", nil, "IDL:scs/
       core/IComponent:1.0")
2  archComponent = orb:narrow(archComponent, "IDL:scs/core/IComponent:1.0")
3
4  local archFacet = archComponent:getFacetByName("ArchManager")
5  archFacet = orb:narrow(archFacet, "IDL:scs/core/arch/IArchManager:1.0")
6
7  local RoomSCR1 = {
8    clients = { Room }, servers = { SC1 },
9    relationship = RoomSCRecord,
10   unique_name = "RoomSCR"
11 }
12 local RoomSCDT1 = { ... }
13 local RoomSCC1 = { ... }
14
15 archFacet:removeConnection(RoomSCR1)
16 archFacet:removeConnection(RoomSCDT1)
17 archFacet:removeConnection(RoomSCC1)
18 archFacet:removeInstance(SC1)
```

Código 5.6: Manipulando instâncias

O código 5.6 demonstra a manipulação de instâncias para remoção do *SpeedCar* SC1 da sala. Primeiro as instâncias de conexão são redefinidas localmente de maneira a se especificar somente a conexão específica a ser removida (somente com SC1). Em seguida, a instância SC1 é removida diretamente. É importante notar que os passos de remoção das conexões não são necessários, uma vez que a remoção de uma instância de componente implica na remoção de todas as suas conexões. Por outro lado, a adição de uma instância de componente implica na necessidade de explicitação das novas conexões, uma vez que a ferramenta não resolve as dependências de maneira automática.

Uma outra opção é especificar mudanças e aplicá-las em conjunto. O autor acredita que esta forma de adaptação é mais interessante, uma vez que ficam documentadas as especificações de mudanças, tornando os estados do sistema mais rastreáveis ao longo do ciclo de vida.

```
1  local adaptation = {
2    architectureChange = { newRoles = { }, removed = { }, added = { }, },
3    systemChange = {
4      componentChanges = { }, addedComponents = { },
5      removedConnections = { }, addedConnections = { },
6      removedComponents = { SC1 }
7    }
8  }
9
10 archFacet:adapt(adaptation)
```

Código 5.7: Manipulando instâncias com linguagem de adaptação

O código 5.7 ilustra a remoção do componente SC1 por meio da linguagem de adaptação. Como o estilo arquitetural não foi alterado, a tabela *architectureChange* só possui conteúdos vazios. Nas mudanças do sistema (*systemChanges*) o componente SC1 é incluído na tabela *removedComponents*, de maneira a ser desconectado e desligado.

5.3.2 Manipulação do estilo arquitetural

Uma dimensão importante em evolução dinâmica de software é a evolução de tipos arquiteturais, descrita na sub-seção 2.4.2. Novos componentes e relacionamentos podem surgir a partir de necessidades identificadas após o sistema entrar em execução, e é importante que sejam levados em conta neste escopo. O exemplo da sub-seção 5.3.1 afeta o conjunto de instâncias arquiteturais do sistema, mas não os tipos ou especificações que definem o comportamento destas instâncias.

Para o nosso exemplo, iremos realocar o controle da faceta *IDataTransfer* do componente *Room* para um novo componente *DataHandler*, de maneira a introduzir novas políticas para a transferência de mídias.

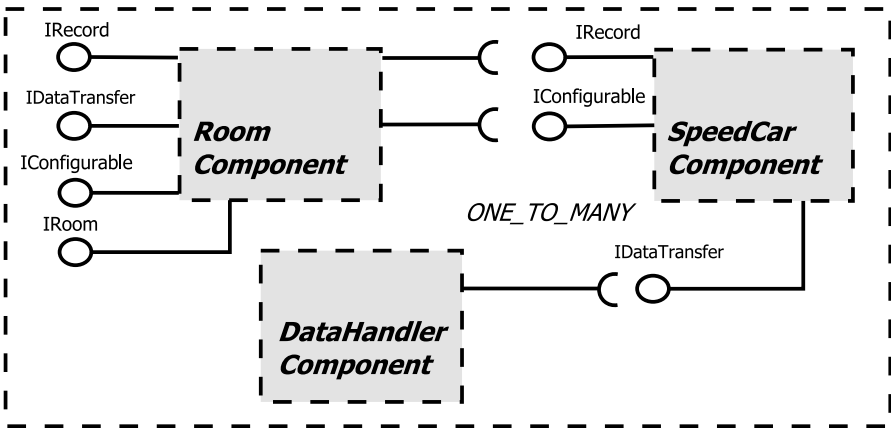


Figura 5.5: Reconfiguração do estilo arquitetural

A figura 5.5 demonstra a estrutura do estilo arquitetural após as modificações. O novo componente precisa ser introduzido na arquitetura, assim como o seu relacionamento com os *SpeedCars*.

```
1  DataHandlerComponent = {
2    provided = { },
3    required = { DataTransferService },
4    unique_name = "DataHandlerRole"
5  }
6
7  DataHandlerSCDataTransfer = {
8    client = DataHandlerComponent,          server = SpeedCarComponent,
9    service = DataTransferService,          arity = ONE_TO_MANY,
10   unique_name = "DataHandlerSCDataTransfer"
11 }
```

Código 5.8: Reconfiguração do estilo arquitetural

O código 5.8 mostra como o novo componente (*DataHandlerComponent*) e a nova relação com os *SpeedCars* (*DataHandlerSCDataTransfer*) são introduzidos. A seguir, é necessário alterar as instâncias dos componentes em execução para acomodar as reconfigurações.

```
1  DataHandler = {
2    role = DataHandlerComponent,          machine = machines.CentOS2,
3    unique_name = "Data␣Handler",
4    impl = io.open("src/tests/CAS/dataTransfer.lua", "r"):read("*a")
5  }
6
7  DataHandlerSCDT = {
8    clients = { DataHandler },
9    servers = { SC1, SC2, SC3 },
10   relationship = DataHandlerSCDataTransfer,          unique_name = "DataHandlerSCDT"
11 }
12
13 adaptation = {
14   architectureChange = {
15     newRoles = { DataHandlerComponent },
16     removed = { RoomSCDataTransfer },
17     added = { DataHandlerSCDataTransfer },
18   },
19   systemChange = {
20     componentChanges = { }, addedComponents = { DataHandler },
21     removedConnections = { RoomSCDT }, addedConnections = { DataHandlerSCDT },
22     removedComponents = { }
23   }
24 }
```

Código 5.9: Reconfiguração das instâncas

No código 5.9 uma nova instância do componente (*DataHandler*) é introduzida, assim como uma conexão (*DataHandlerSCDT*) com os *SpeedCars*. Na especificação da adaptação, o novo componente (*DataHandlerComponent*) e a instância são adicionados, assim como as mudanças nos relacionamentos

e conexões. Assim, o controle a transferência de mídias passa agora a ser da nova instância e não mais da sala.

Uma das limitações da ferramenta é a impossibilidade de expressar remoção de facetas individuais. Esta situação torna-se problemática pela necessidade de redefinição do tipo base do componente. Uma vez que sua faceta seja removida um novo tipo de componente deve ser introduzido, e isso implica em cenários diversos para serem tratados, como por exemplo o que se deve fazer com as outras instâncias deste componente. O autor decidiu por fornecer uma abordagem que favorecesse a simplicidade da implementação da ferramenta, mas esta é uma funcionalidade que pode ser suportada no futuro.

Para este exemplo, é possível ver que o componente *Room*, apesar de não mais ter relação com o serviço *IDataTransfer*, ainda continua a fornecê-lo. Em tempo de execução isto pode vir a ser um problema, uma vez que podem ocorrer erros na chamada do serviço. Uma solução paliativa para estes casos é introduzir um novo tipo para substituir o componente *Room* e trocar todas as suas instâncias na infraestrutura. Um outro problema que surge neste exemplo é que não há uma maneira padrão de remover o código de acesso às dependências. Neste caso, o receptáculo do tipo *IDataTransfer* foi desconectado, porém o componente pode não saber lidar com esta situação, uma vez que quando implantado, assumia-se que esta era uma dependência resolvida. Estes são temas de investigação futura.

5.4 Correção de Defeitos

Durante a fase do ciclo de vida de execução do software, é comum se detectar erros. Em um sistema distribuído, é importante corrigir estes defeitos sem o desligamento da infraestrutura. Para o CAS, vamos considerar a ocorrência de erros na implementação das facetas do *SpeedCar*. Nesta seção vamos demonstrar como substituir implementações de serviços.

O código de implementação do componente deve sempre retornar uma tabela Lua com as facetas que o componente provê. Desta maneira o código a seguir demonstra como é a implementação de um *SpeedCar* Lua:

```
1  local DataTransfer = oo.class{name = "SpeedCarDataTransfer"}
2  -- DataTransfer implementation ...
3
4  local Recorder = oo.class{name = "SpeedCarRecorder"}
5  -- Recorder implementation ...
6
7  local Configurable = oo.class{name = "SpeedCarConfigurable"}
8  -- Configurable implementation ...
9
10 function startup() ... end
11 function shutdown() ... end
```

```
12
13    return {
14      IRecord = Recorder,
15      IConfigurable = Configurable,
16      IDataTransfer = DataTransfer,
17      startup = startup,
18      shutdown = shutdown
19    }
```

Código 5.10: Manipulando instâncias

O código das implementações das facetas está omitido para facilitar o entendimento do exemplo. O código demonstra as classes de cada uma das facetas do componente (*IDataTransfer*, *IRecord*, *IConfigurable*) sendo retornadas. Os campos *startup* e *shutdown* são utilizados para indicar os métodos de inicialização e desligamento do componente, respectivamente. O configurador recebe este código e o repassa para o implantador criar o componente.

Vamos supor que um erro na implementação da faceta *DataTransfer* seja detectado no *SpeedCar* SC2. É interessante então que o *SpeedCar* seja substituído sem necessidade de reinicialização dos outros componentes da sala. Para efetuar a correção, duas opções são apresentadas ao mantenedor da aplicação:

- Substituir o componente por completo;
- Substituir apenas algumas facetas determinadas;

O primeiro pode ser alcançado através da chamada ao método remoto *replaceInstance* do configurador. Utilizar esta função é o mesmo que remover o componente (*removeInstance*), implantar outro em seu lugar (*addInstance*) e refazer suas conexões (*addConnection*). Este método é necessário quando se quer substituir por completo o componente, em função de uma falha em uma faceta afetar outras seções do programa, por exemplo. O código 5.11 demonstra como substituir uma instância de componente por inteiro.

```
1   local archComponent = orb:newproxy("corbaloc:iiop:192.168.1.210:8021/ArchManager", nil, "IDL:scs/
        core/IComponent:1.0")
2   archComponent = orb:narrow(archComponent, "IDL:scs/core/IComponent:1.0")
3
4   local archFacet = archComponent:getFacetByName("ArchManager")
5   archFacet = orb:narrow(archFacet, "IDL:scs/core/arch/IArchManager:1.0")
6
7   NewSC2 = {
8     role = RoomComponent,              machine = machines.Ubuntu1,
9     unique_name = "SpeedCar2",
10    impl = io.open("src/tests/CAS/newsc2.lua", "r"):read("*a")
11  }
12
13  archFacet:replaceInstance(SC2, NewSC2)
```

Código 5.11: Substituindo uma instância de componente

O novo componente é carregado no caminho "*src/tests/CAS/newsc2.lua*" e a instância anterior é substituída. A depender da necessidade do mantenedor e da origem da falha, pode ser inviável substituir uma instância inteira do componente, ou seja, o defeito pode não afetar o componente por inteiro. O código 5.12 demonstra como substituir facetas.

```
1 local DataTransferFix = oo.class{name = "SpeedCarDataTransfer"}
2 -- New DataTransfer implementation ...
3
4 return { IDataTransfer = DataTransferFix }
```

Código 5.12: Substituindo implementação de facetas

A especificação da adaptação é feita com a estrutura *componentChanges*. Neste caso, uma tabela Lua é retornada apenas com a implementação dos serviços a serem atualizados, no caso a faceta *IDataTransfer*. Os passos a seguir descrevem como o mecanismo funciona para este caso:

1. Configurador altera o estado do *SpeedCar* para *SUSPENDED*;
2. Configurador envia esta atualização ao implantador;
3. Implantador desativa a faceta, guardando a implementação anterior;
4. Implantador tenta ativar a faceta com a nova implementação;
5. Caso a ativação não tenha sucesso, o implantador reativa a faceta com a implementação anterior e retorna um erro para o Configurador;
6. Configurador altera o estado do *SpeedCar* para *RESUMED*, que volta a responder requisições.

5.5 Considerações Finais

Este capítulo apresentou um exemplo de uso da ferramenta SCS-DynAdapt sobre o CAS. As funcionalidades demonstradas focaram em descrição, instanciação e adaptação da arquitetura distribuída. Para o exemplo apresentado, foi possível modelar satisfatoriamente os serviços (*IDataTransfer*, *IRoom*, *IRecord* e *IConfigurable*), componentes (*Room* e *SpeedCars*) e conexões do CAS.

A descrição arquitetural foi feita de maneira a expressar o relacionamento entre estes componentes. Para cada relacionamento foi expressa uma aridade

a ser aplicada (um-para-muitos), um serviço CORBA, um nome e os clientes e servidores. A partir destes elementos, foi possível instanciar o sistema sem a necessidade de utilização do barramento de serviços OpenBus [40], facilitando a inicialização. Os componentes são implantados e conectados automaticamente através do *Configurador* e do *Implantador*.

Em seguida são demontrados dois cenários de reconfiguração da sala e um de alteração de componentes em tempo de execução. A primeira reconfiguração envolve adicionar/remover instâncias dos componentes de captura (*SpeedCar*), através de chamadas de funções. Este cenário é relevante, porém não cobre as chamadas mudanças arbitrárias. O segundo exemplo demonstra como manipular o estilo arquitetural, ou a base da arquitetura. Esta manipulação dá espaço à quebra de restrições impostas inicialmente. Um dos relacionamentos é refeito (do serviço *IDataTransfer*) introduzindo um novo tipo de componente na arquitetura. Ainda é possível alterar implementações de facetas em tempo de execução. O exemplo da seção 5.4 mostra como corrigir possíveis defeitos no *SpeedCar* pela troca de uma de suas facetas. Este tipo de adaptação mantém o componente em execução mudando apenas os serviços necessários.

Os recursos implementados mostraram-se satisfatórios para os cenários apresentados, porém ainda existem algumas funcionalidades que devem ser suportadas de maneira a se obter mais flexibilidade nas adaptações. A implementação de evolução de tipos arquiteturais é importante é vital para esta finalidade, de maneira a estender ou alterar os tipos de componentes e conexões existentes. É necessário poder alterar tipos de componentes para, por exemplo, passar a fornecer ou requisitar serviços. Também não foi criada uma maneira padrão dos componentes acessarem os seus receptáculos; isto faz com que seja difícil alterar os códigos de requisição de serviços. Este empecilho dificulta e evolução de interfaces de serviço, não implementada neste trabalho. Uma solução parcial para este problema é introduzir novos serviços, componentes e relacionamentos e reinstanciar componentes afetados, porém o resultado disso é pouco satisfatório.

6
Conclusão

Este trabalho apresentou um mecanismo de descrição de arquiteturas para sistemas distribuídos baseados em componentes de software, com suporte a mecanismos de especificação e evolução dinâmica. Os mecanismos criados foram implementados sobre uma linguagem de definição de interface (OMG IDL), através da tecnologia CORBA [32]. A ferramenta SCS-DynAdapt, desenvolvida neste trabalho, permite a implantação remota de componentes Lua que seguem o modelo SCS [41]. Ainda são fornecidas capacidades de criação, reflexão e manipulação de elementos que compõem a arquitetura.

A linguagem de descrição arquitetural segue uma abordagem programática, permitindo especificar serviços, componentes, conexões e estilos arquiteturais, assim como instâncias de sistemas. O uso de estilo arquiteturais aumenta o reuso dos componentes, permite a análise automática de conformidade arquitetural e facilita a integração entre sistemas [24]. A partir daí, é possível definir instâncias distribuídas de unidades de execução (componentes) e suas conexões, atendendo a determinadas composições.

De maneira a prover suporte a evolução dinâmica, o SCS-DynAdapt também permite que a arquitetura do sistema seja manipulada, com interrupção mínima de partes não afetadas. Assim, componentes implantados podem ser removidos/alterados e novos componentes podem ser adicionados, durante a execução da aplicação. Estas alterações podem ser aplicadas tanto de maneira pontual quanto através de uma linguagem de especificação. A ferramenta guarda as mudanças e as disponibiliza para o mantenedor, tornando mais rastreáveis os estados da arquitetura do sistema durante o seu ciclo de vida.

Para implementar estas funcionalidades, foram desenvolvidos o componente *Implantador* e o *Configurador*. O primeiro é encarregado de (re)instalar, remover, alterar e adicionar instâncias de componentes nas máquinas da infraestrutura. Através de capacidades de reflexão de Lua, as implementações dos serviços fornecidos por estes componentes são criadas de maneira dinâmica, com tratamento de erros. Este componente ainda é responsável por trocar estas implementações. Já o *Configurador* se encarrega de gerenciar o ciclo de vida dos componentes especificados (e dos implantadores), manipular as suas

conexões e garantir que restrições arquiteturais não sejam violadas. Além disso, fornece serviços de reflexão, tornando possível navegar na composição dos componentes e conexões que compõem o sistema. Também guarda o conjunto de mudanças aplicadas no sistema, mantendo um histórico do que ocorreu ao longo do tempo.

A descrição arquitetural foi construída através de uma linguagem intermediária (OMG IDL), podendo ser especificada por qualquer linguagem de programação que tenha implementação do *middleware* de comunicação CORBA. Desta maneira, a criação e documentação da arquitetura são inerentemente programáticas, e os recursos dessas linguagens podem ser utilizados. Assim, por exemplo, o uso de instanciação dinâmica direta de *Darwin* (seção 3.2) pode ser aplicado através de comandos de repetição simples. Além disso, a ferramenta permite a instanciação remota de componentes pela própria descrição da arquitetura, funcionalidade pouco discutida na literatura.

Contudo, este trabalho apresenta algumas limitações. A instalação de dependências estáticas ainda precisa ser feita de maneira manual, e o desenvolvedor precisa ter um conhecimento prévio de como o componente irá acessar suas dependências. Além disso, não há um suporte adequado para expressar essas dependências nem as suas adaptações. A evolução de tipos também é limitada, na medida em que ainda não permite a mudança de interfaces de serviços CORBA em tempo de execução. A única maneira de evoluir um componente desta maneira é removê-lo da arquitetura e inserir um novo com uma nova especificação. A ferramenta também não dá suporte à restauração de estados em componentes completamente substituídos, deixando este gerenciamento exclusivamente a cargo do mantenedor.

Por fim, alguns trabalhos futuros são listados a seguir:

Evolução de tipos É vital fornecer suporte à remoção de tipos de componentes, de maneira que todas as suas instâncias sejam retiradas da arquitetura. Também é preciso prover suporte a evolução de interfaces CORBA em tempo de execução, sem a necessidade de se trocar o componente por inteiro;

Refinamento da suspensão A alteração do componente para o estado suspenso pode ser refinada de maneira a suspender somente as interfaces afetadas e não o componente por inteiro;

Composições hierárquicas O SCS-Composite permite a exportação pelo componente composto de facetas de componentes internos. Esta exposição pode ser aproveitada na descrição arquitetural para prover suporte

a ligações entre sistemas distribuídos, como componentes compostos ligados uns aos outros.

Multi-linguagem O SCS-DynAdapt só fornece no momento suporte ao desenvolvimento de componentes na linguagem Lua. De maneira se ter sistemas ainda mais heterogêneos, é interessante ter implementações para outras linguagens de programação;

Implantação A resolução de dependências da ferramenta é voltada apenas para dependências paramétricas, ou seja, dependências que são expressadas como componentes SCS. A integração com o SCS-DeploySystem [37] permitiria expressar também dependências estáticas, como bibliotecas dinâmicas, executáveis, bancos de dados, etc;

Checkpoints A aplicação de adaptações é capaz de detectar determinados erros, e a ferramenta permite análise do histórico de mudanças aplicadas. Porém, não há um suporte para se desfazer adaptações mal-sucedidas, que introduzam defeitos, ou comportamente inesperado. Assim, é vital fornecer uma mecanismo que permita ao mantenedor regredir a configuração do sistema;

Transferência automática de estados Durante a troca de instâncias de componentes, a transferência de estados do componente removido para o componente inserido deve ser feita de maneira automática, não requerindo ações do mantenedor;

7
Referências Bibliográficas

[1] SANTOS, A. M. dos. **Suporte a Componentes Compostos Para o Middleware SCS**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, 2012. (document), 1.1, 2.5, 2.5, 5, 5.1, 5.1

[2] KRAMER, J.; MAGEE, J. Change management of distributed systems. In: **Proceedings of the 3rd workshop on ACM SIGOPS European workshop: Autonomy or interdependence in distributed systems?** New York, NY, USA: ACM, 1988. (EW 3), p. 1–4. Disponível em: <http://doi.acm.org/10.1145/504092.504113>. 1

[3] SZYPERSKI, C. Component technology: what, where, and how? In: **Proceedings of the 25th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2003. (ICSE '03), p. 684–693. ISBN 0-7695-1877-X. Disponível em: <http://portal.acm.org/citation.cfm?id=776816.776916>. 1, 2.2.1

[4] BRUNETON, E. et al. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 36, p. 1257–1284, September 2006. ISSN 0038-0644. Disponível em: <http://portal.acm.org/citation.cfm?id=1152333.1152345>. 1

[5] EBRAERT, P.; TOURWÉ, T. A reflective approach to dynamic software evolution. In: CAZZOLA, W.; CHIBA, S.; SAAKE, G. (Ed.). **RAM-SE**. [S.l.]: Fakultät für Informatik, Universität Magdeburg, 2004. p. 37–43. 1

[6] SORIA, C. C. **Dynamic Evolution and Reconfiguration of Software Architectures through Aspects**. Tese (Doutorado) — Universitat Politècnica de València. Departamento de Sistemas Informáticos y Computación - Departament de Sistemes Informàtics i Computación, 2011. 1, 2.1, 2.3, 2.3.4

[7] KRAMER, J.; MAGEE, J. Dynamic configuration for distributed systems. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ,

USA, v. 11, p. 424–436, April 1985. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.1985.232231>. 1, 2.3.2, 2.4, 3.1, 3.1, 3.1

[8] KRAMER, J.; MAGEE, J. The evolving philosophers problem: Dynamic change management. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 16, n. 11, p. 1293–1306, nov. 1990. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/32.60317>. 1, 2.3.2, 2.3.4, 4.6, 4.6

[9] GARLAN, D.; MONROE, R. T.; WILE, D. Acme: architectural description of component-based systems. In: LEAVENS, G. T.; SITARAMAN, M. (Ed.). **Foundations of component-based systems**. New York, NY, USA: Cambridge University Press, 2000. cap. Acme: architectural description of component-based systems, p. 47–67. ISBN 0-521-77164-1. Disponível em: <http://portal.acm.org/citation.cfm?id=336431.336437>. 1, 2.2.4, 3

[10] KRAMER, J.; MAGEE, J. Analysing dynamic change in distributed software architectures. **IEE Proceedings - Software**, v. 145, n. 5, p. 146–154, 1998. 1, 4.5

[11] ALLEN, R.; GARLAN, D. Beyond definition/use: Architectural interconnection. In: **Proceedings of the ACM Interface Definition Language Workshop**. [S.l.]: SIGPLAN Notices, 1994. v. 29(8). 1

[12] MAGEE, J.; KRAMER, J. Dynamic structure in software architectures. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 21, p. 3–14, October 1996. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/250707.239104>. 1, 2.4.1

[13] LUCKHAM, D. C. **Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events.** Stanford, CA, USA, 1996. 1

[14] MAGEE, J.; DULAY, N.; KRAMER, J. Structuring parallel and distributed programs. In: **Configurable Distributed Systems, 1992., International Workshop on**. [S.l.: s.n.], 1992. p. 102 –117. 1, 2.2.4, 2.3.2, 3.2, 3.3

[15] KRAMER, J.; MAGEE, J. Conic: An integrated approach to distributed computer control systems. **Proc. Elet. Eng.**, v. 130, n. 1, January 1983. 1, 3.1, 3.2

[16] MAIER, M. W.; EMERY, D.; HILLIARD, R. Ieee std. 1471-200 and systems engineering. **Syst. Eng.**, John Wiley and Sons Ltd., Chichester, UK, v. 7, p. 257–270, September 2004. ISSN 1098-1241. Disponível em: <http://portal.acm.org/citation.cfm?id=1077494.1077499>. 1

[17] TAJALLI, H. et al. Plasma: a plan-based layered architecture for software model-driven adaptation. In: PECHEUR, C.; ANDREWS, J.; NITTO, E. D. (Ed.). **ASE**. [S.l.]: ACM, 2010. p. 467–476. ISBN 978-1-4503-0116-9. 1

[18] MENS, T.; MAGEE, J.; RUMPE, B. Evolving software architecture descriptions of critical systems. **Computer**, IEEE Computer Society, Los Alamitos, CA, USA, v. 43, p. 42–48, 2010. ISSN 0018-9162. 1, 1.1, 2.4

[19] AUGUSTO, C. et al. **SCS: Software Component System**. maio 2006. Disponível em: <http://www.tecgraf.puc-rio.br/scs>. 1.1, 2.5

[20] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; FILHO, W. C. Lua - an extensible extension language. **Software: Practice and Experience**, John Wiley & Sons, Ltd., v. 26, n. 6, p. 635–652, 1996. ISSN 1097-024X. Disponível em: <http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P>. 1.1

[21] MCVEIGH, A.; KRAMER, J.; MAGEE, J. Using resemblance to support component reuse and evolution. In: **Proceedings of the 2006 conference on Specification and verification of component-based systems**. New York, NY, USA: ACM, 2006. (SAVCBS '06), p. 49–56. ISBN 1-59593-586-X. Disponível em: <http://doi.acm.org/10.1145/1181195.1181206>. 1.1, 3.3

[22] PORTELLA, F.; CERQUEIRA, R. **Um serviço de captura e acesso para espaços ativos**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, 2008. 1.1, 4.8, 5, A.2

[23] PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 17, n. 4, p. 40–52, out. 1992. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/141874.141884>. 2.1

[24] MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: foundations, theory, and practice. In: KRAMER, J. et al. (Ed.). **ICSE (2)**. [S.l.]: ACM, 2010. p. 471–472. ISBN 978-1-60558-719-6. 2.1, 2.2.1, 2.2.4, 6

[25] MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 26, n. 1, p. 70–93, jan. 2000. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/32.825767>. 2.2, 2.4, 3

[26] TAMZALIT, D.; MENS, T. Guiding architectural restructuring through architectural styles. In: **Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on**. [S.l.: s.n.], 2010. p. 69–78. 2.2.4

[27] MCVEIGH, A.; KRAMER, J.; MAGEE, J. Evolve: tool support for architecture evolution. In: **Proceeding of the 33rd international conference on Software engineering**. New York, NY, USA: ACM, 2011. (ICSE '11), p. 1040–1042. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985990>. 2.2.5

[28] CHAPIN, N. et al. Types of software evolution and software maintenance. **Journal of Software Maintenance**, John Wiley & Sons, Inc., New York, NY, USA, v. 13, n. 1, p. 3–30, jan. 2001. ISSN 1040-550X. Disponível em: <http://dl.acm.org/citation.cfm?id=371697.371701>. 2.3

[29] BUCKLEY, J. et al. Towards a taxonomy of software change: Research articles. **J. Softw. Maint. Evol.**, John Wiley & Sons, Inc., New York, NY, USA, v. 17, n. 5, p. 309–332, set. 2005. ISSN 1532-060X. Disponível em: <http://dx.doi.org/10.1002/smr.v17:5>. 2.3, 2.3.2

[30] PISSIAS, P.; COULSON, G. Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. **IET Software**, IET, v. 2, n. 4, p. 348–361, 2008. Disponível em: <http://link.aip.org/link/?SEN/2/348/1>. 2.3.4, 4.5

[31] BOX, D. **Essential COM**. Boston, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. Foreword By-Grady Booch and Foreword By-Charlie Kindel. ISBN 0201634465. 2.5

[32] OBJECT MANAGEMENT GROUP. **CORBA Components - Version 3.0**. Needham, USA, jun. 2002. Document: formal/2002-06-65. 2.5, 4.2, 6

[33] ALLEN, R. **A Formal Approach to Software Architecture**. Tese (Doutorado) — Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144. 3

[34] GARLAN, D.; ALLEN, R.; OCKERBLOOM, J. Exploiting style in architectural design environments. In: **Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering**. New York, NY, USA: ACM, 1994. (SIGSOFT '94), p. 175–188. ISBN 0-89791-691-3. Disponível em: <http://doi.acm.org/10.1145/193173.195404>. 3

[35] MAGEE, J.; KRAMER, J.; SLOMAN, M. Constructing distributed systems in conic. **IEEE Trans. Software Eng.**, v. 15, n. 6, p. 663–675, 1989. 3.1

[36] ALLEN, R.; DOUENCE, R.; GARLAN, D. Specifying and analyzing dynamic software architectures. In: **Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)**. Lisbon, Portugal: [s.n.], 1998. 4

[37] JÚNIOR, A. A. B. **Implantação de Componentes de Software Distribuídos Multi-Linguagem e Multi-Plataforma**. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, 2009. 2, 6

[38] YLONEN, T. **SSH - Secure Shell**. IETF, January 2006. RFC 4253. Disponível em: <http://www.ietf.org/rfc/rfc4253.txt>. 4.4

[39] MAIA, R.; CERQUEIRA, R.; CALHEIROS, R. OiL: An object request broker in the Lua language. In: FONSECA, M. S. P. (Ed.). **Proceedings of SBRC'06 - Salão de Ferramentas**. Porto Alegre, Brazil: SBC, 2006. p. 1439–1446. 4.5

[40] TECGRAF. **Openbus - Enterprise Integration Application Middleware**. 2006. Disponível em: <http://www.tecgraf.puc-rio.br/openbus>. 5, 5.5

[41] AUGUSTO, C. E. L. et al. **SCS: Software Component System**. 2009. `http://www.tecgraf.puc-rio.br/~scorrea/scs`. 6

A
Apêndice

A.1
IDLs de acesso ao SCS-DynAdapt

Nesta seção são apresentadas as IDLs de acesso aos serviços da ferramenta SCS-DynAdapt.

A.1.1
IDL de Controle de estados

O código A.1 apresenta a idl CORBA usada no controle de estados de componentes.

```
#ifndef LIFECYCLE
#define LIFECYCLE

module scs {
  module core {
    module lifecycle {
      // Enumeration used to define the possible component states
      enum State {
        // Component is running and its services are up
        RESUMED,
        // Component is not available to respond requests. Requests are dropped
        HALTED,
        // Component is currently suspended, enqueues calls
        SUSPENDED
      };

      //Exception used to indicate that the component cannot change its current execution state. "msg"
          parameter should contain info on why this exception was raised
      exception CannotChangeState { string msg; };


      //Interface to control component state
      interface ILifeCycle {
        //Acquires the current state the component is in
        State getState();

        /**
         * @brief Attempts to change the component state
         * @throws CannotChangeState Thrown if the component cannot change to the new state
         */
        boolean changeState(in State state) raises(CannotChangeState);
      };

```

```
33          //Exception thrown when a non scs/CORBA request is made to a halted component
34          exception HaltedComponent { string msg; };
35        };
36      };
37    };
38
39    #endif
```

Código A.1: Controle de estados em um componente adaptável

A.1.2 IDL de Implantação de Componentes

O código A.2 apresenta a idl CORBA usada nos serviços de implantação remota de componentes.

```
1   #ifndef DEPLOYER
2   #define DEPLOYER
3
4   #include <scs.idl>
5   #include <arch.idl>
6
7   module scs {
8     module core {
9       module arch {
10        module deploy {
11          typedef sequence<octet> OctetSeq;
12
13          exception AlreadyRegistered{ scs::core::arch::ComponentInstance component_instance; };
14          exception UnknownService{ string msg; scs::core::arch::Service service; };
15          exception NotInstalled{ string unique_name; };
16          exception NoImplementation{ string name; string interface_name; };
17
18          interface IDeployer {
19            boolean isAvailable();
20            void install( in scs::core::arch::ComponentInstance component_instance ) raises (
                AlreadyRegistered, UnknownService);
21            //void run (in string unique_name) raises (NotInstalled, NoImplementation);
22            scs::core::IComponent run (in string unique_name) raises (NotInstalled, NoImplementation);
23            scs::core::IComponent replaceInstance(in scs::core::arch::ComponentInstance oldInst, in scs::core
                ::arch::ComponentInstance newInst);
24            void redeploy(in string unique_name, in OctetSeq impl) raises (NotInstalled, scs::core::arch::
                RunError);
25          };
26        };
27      };
28    };
29  };
30
31  #endif
```

Código A.2: Implantador de componentes

A.1.3
IDL de Descrição e Manipulação Arquitetural

O código A.2 apresenta a idl CORBA usada nos serviços de gerenciamento de descrições arquiteturais.

```
#ifndef ARCH
#define ARCH

module scs {
  module core {
    module arch {
      // Indicate connection arities of services
      enum Arity {
        ONE_TO_ONE,
        ONE_TO_MANY,
        MANY_TO_ONE,
        MANY_TO_MANY,
        FREE
      };

      // Service description, without the implementation yet
      struct Service {
        string name;
        string interface_name;
        Arity arity;
      };
      typedef sequence<Service> Services;

      // Component template
      struct ComponentRole {
        Services provided;
        Services required;
        string unique_name;
      };
      typedef sequence<ComponentRole> ComponentRoles;

      // Relationship template
      struct Relationship {
        ComponentRole client;
        ComponentRole server;
        Service service;
        string unique_name;
      };
      typedef sequence<Relationship> Relationships;

      // Architectural style
      struct Architecture {
        Relationships relationships;
        ComponentRoles components;
      };

      // System execution machines config
      struct Machine {
        string host; long port;
        string unique_name;
      };
      typedef sequence<Machine> Machines;

```

```
54        // Component implementation code
55        typedef sequence<octet> Code;
56
57        // Component instance to be deployed
58        struct ComponentInstance {
59          ComponentRole role;
60          string unique_name;
61          Machine machine;
62          Code impl;
63        };
64        typedef sequence<ComponentInstance> ComponentInstances;
65
66        // Connection between two instances
67        struct ConnectionInstance {
68          ComponentInstances clients;
69          ComponentInstances servers;
70          Relationship relationship;
71          string unique_name;
72        };
73        typedef sequence<ConnectionInstance> ConnectionInstances;
74
75        // System, a composite component
76        struct System {
77          ConnectionInstances connections;
78          ComponentInstances components;
79        };
80
81        /**
82         * @brief Architectural Description Language of a system (composite component)
83         */
84        struct ADL {
85          string system_name;
86          Architecture architecture;
87          System system;
88        };
89
90        // Base-level architecture changes
91        struct ArchitectureChange {
92          ComponentRoles newRoles;
93          Relationships removed;
94          Relationships added;
95        };
96
97        // Changes to component instances
98        struct ComponentInstanceChange {
99          ComponentInstance previous;
100         Code newImpl;
101       };
102       typedef sequence<ComponentChange> ComponentChanges;
103
104       // Changes to connection instances
105       struct ConnectionChange {
106         ConnectionInstance oldConnection;
107         ConnectionInstance newConnection;
108       };
109       typedef sequence<ConnectionChange> ConnectionChanges;
110
111       // System changes
112       struct SystemChange {
```

```
113           ComponentInstancesChanges componentChanges;
114           ComponentInstances addedComponents;
115           ComponentInstances removedComponents;
116           ConnectionInstances removedConnections;
117           ConnectionInstances addedConnections;
118         };
119
120         // System and architecture adaptation
121         struct Adaptation {
122           ArchitectureChange architectureChange;
123           SystemChange systemChange;
124         };
125
126         exception InvalidADL{};
127         exception InvalidMachine{};
128         exception StartSystemFailed{};
129         exception AlreadyStarted{ string msg; };
130         exception NotStarted{};
131         exception UnavailableMachine{ string msg; };
132         exception MachineAlreadyExists{};
133         exception NoDeployer{};
134         exception InstallError{ string msg; };
135         exception RunError{ string msg; };
136         exception ShutdownError{ string msg; };
137         exception NonExistentInstance{};
138
139         interface IArchManager {
140           //boolean setSystemName(string system_name);
141           string getSystemName();
142           boolean startSystem(in ADL adl) raises (InvalidADL, StartSystemFailed, UnavailableMachine,
                AlreadyStarted, NoDeployer, InstallError, RunError);
143           boolean addMachine(in Machine machine) raises (MachineAlreadyExists, UnavailableMachine,
                InvalidMachine);
144           boolean shutdownSystem() raises (ShutdownError);
145           Machines getMachines();
146           boolean shutdownDeployers() raises (UnavailableMachine, NotStarted);
147           boolean isStarted();
148           boolean stopSystem();
149
150           // Adapts the system
151           boolean replaceInstance(in string unique_name, in Code newImpl) raises (NonExistentInstance,
                RunError, NoDeployer);
152           //boolean removeInstance(in string unique_name) raises (NonExistentInstance);
153           //boolean addInstance(in ComponentInstance instance) raises (NonExistentInstance);
154           void adapt( in Adaptation adaptation );
155
156           string toStringArch();
157         };
158       };
159     };
160   };
161
162   #endif
```

Código A.3: Gerenciador Arquitetural

A.2 IDLs do CAS

Nesta seção são apresentadas as IDLs dos serviços do sistema CAS [22].

A.2.1 IDLs da sala e configurador

O código A.4 apresenta a IDL CORBA do serviço fornecido pelo componente *Room*.

```
#ifndef ROOM
#define ROOM

#include "monitoring.idl"

//Interface com funcionalidades para as salas de captura da infraestrutura do CAS
module cas {

  //Módulo de interações associadas com uma sala
  module room {

    // Interface com as funcionalidades de uma sala que agregadora de
    // serviços diversos (gravadores de eventos, transferidores de mídias e transcodificadores)
    interface IRoom {
      string getStatus();
      string getName();
      string getCurrentEventProfile();
      void setCurrentEventProfile(in string profile);
      long getCurrentEventID();
      void setCurrentEventID(in long eventID);
      long addFailMonitor(in cas::monitoring::IFailEvent failMonitor);
    };
  };
};

#endif
```

Código A.4: IDL da sala

O código A.5 apresenta a IDL CORBA do serviço fornecido pelo componente *RoomConfigurator*.

```
#ifndef CONFIGURATOR
#define CONFIGURATOR

// Interface com funcionalidades para a configuração de salas de captura da infraestrutura do CAS
module cas {

  //Módulo de interações associadas com uma sala
  module room {

    typedef long ConnectionId;

    //Interface de configuração (reconfiguração, reconexão) dos componentes de uma sala.
    interface IRoomConfigurator {
      ConnectionId connectComponent(in Object comp);
```

```
15          boolean disconnectComponent(in ConnectionId connection_Id);
16        };
17      };
18    };
19
20    #endif
```

Código A.5: IDL do configurador da sala

O código A.6 apresenta a IDL CORBA de gravações de eventos.

```
1   #ifndef RECORDER
2   #define RECORDER
3
4   #include "dataRepository.idl"
5
6   module cas {
7     module recorder {
8       // Componente não se encontra em estado de gravação
9       exception NotRecording { string msg; };
10
11      // Componente já se encontra em estado de gravação
12      exception AlreadyRecording { string msg; };
13
14      // NotConfigured Componente ainda não se encontra devidamente configurado
15      exception NotConfigured { string msg; };
16
17      // Ocorreu um erro no componente que impediu o inicio da gravação
18      exception StartRecordFailure { string reason; };
19
20      // Interface de controle de captura de eventos multimídia
21      interface IRecord {
22
23        // Inicia a gravação de um determinado evento
24        // O componente precisa já estar devidamente configurado
25        void startRecord() raises(NotConfigured, AlreadyRecording, StartRecordFailure);
26
27        // Retorna o estado da entidade de gravação
28        string getStatus();
29
30        // Finaliza a gravação de um evento
31        void stopRecord() raises(NotRecording);
32
33        // Descarta os dados capturados pelo componente
34        void discardRecordedData() raises(NotRecording);
35      };
36    };
37  };
38  #endif
```

Código A.6: IDL de gravação de eventos

O código A.6 apresenta a IDL CORBA de configuração de propriedades.

```
1   #ifndef CONFIGURABLE
2   #define CONFIGURABLE
3
4   module cas {
5     module configuration {
```

```
6
7       // Estrutura de propriedade suportada pelo SpeedCar
8       struct Property {
9         string name;
10        string value;
11      };
12
13      // Propriedades não suportadas por componentes.
14      exception UnsupportedProperty { string msg; };
15
16      // O valor da propriedade não foi bem formado
17      exception MalformedValue { string msg; };
18
19      // Usado quando o componente não pode setar a propriedade do componente
20      exception CannotSetProperty { string msg; };
21
22      // Conjunto de propriedades suportadas pelo componente
23      typedef sequence<Property> supportedProperties;
24
25      // Conjunto de nomes de propriedades suportadas pelo componente
26      typedef sequence<string> supportedPropertiesNames;
27
28      //Interface de configuração de SpeedCars
29      interface IConfigurable {
30        void setProperty(in Property prop) raises (UnsupportedProperty, CannotSetProperty,
              MalformedValue);
31        Property getProperty(in string key) raises (UnsupportedProperty);
32        supportedProperties getProperties();
33        supportedPropertiesNames getSupportedProperties();
34      };
35    };
36  };
37  #endif
```

Código A.7: IDL de configuração

O código A.6 apresenta a IDL CORBA de transferência de mídias resultantes de gravações.

```
1   #ifndef DATA_IDL
2   #define DATA_IDL
3
4   #include "dataRepository.idl"
5
6   module cas {
7     module transfer {
8
9
10      // Dados não-disponíveis para transferência
11      exception DataNotAvailable { string msg; };
12
13      // Erro durante a transferência
14      exception DataTransferError { string msg; };
15
16      // Interface que define a estrutura de listeners de progresso
17      interface IProgressListener {
18        void notifyProgress(in cas::data::Event eventDescriptor, in float progress);
19        void finished(in cas::data::Event eventDescriptor);
20        void error(in cas::data::Event eventDescriptor, in string reason);
```

```
21        };
22
23        // Faceta de transferência de mídias capturadas durante a gravação
24        interface IDataTransfer {
25          // Transfere mídias capturadas para o servidor
26          void transferRecordedData() raises(DataNotAvailable, DataTransferError);
27
28          //Transfere mídias capturadas em um determinado evento para o servidor
29          void transferRecordedDataFromEvent(in cas::data::Event eventDescriptor) raises(DataNotAvailable,
                DataTransferError);
30
31          // Retorna o estado da transferência dos dados
32          string getStatus(in cas::data::Event eventDescriptor);
33
34          // Adiciona uma listener de notificação do progresso da transferência
35          void addProgressListener(in IProgressListener listener);
36
37          // Remove um ouvinte de pós-processamento
38          void removeProgressListener(in IProgressListener listener);
39
40          // Recupera o progresso do processamento
41          float getProgress();
42        };
43      };
44    };
45
46    #endif
```

Código A.8: IDL de transferência de mídias