GERAÇÃO DE PLANOS DE RECONFIGURAÇÃO EM TEMPO DE EXECUÇÃO

Marco Eugênio Madeira Di Beneditto

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientador: Cláudia Mria Lima Werner

Rio de Janeiro
Julho de 2014

GERAÇÃO DE PLANOS DE RECONFIGURAÇÃO EM TEMPO DE EXECUÇÃO

Marco Eugênio Madeira Di Beneditto

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof.ª Cláudia Maria Lima Werner, D.Sc.

Prof.ª Cecília Mary Fischer Rubira, Ph.D.

Prof. Renato Fontoura de Gusmão Cerqueira, D.Sc.

Prof. Geraldo Bonorino Xexéo, D.Sc.

Prof. Toacy Cavalcante de Oliveira, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2014

Di Beneditto, Marco Eugênio Madeira

Geração de Planos de Reconfiguração em Tempo de Execução/Marco Eugênio Madeira Di Beneditto. – Rio de Janeiro: UFRJ/COPPE, 2014.

XII, 154 p.: il.; 29, 7cm.

Orientador: Cláudia Mria Lima Werner

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 91 – 99.

1. de Componentes. 2. Reconfiguração Dinâmica. I. Werner, Cláudia Mria Lima. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais, por seu apoio em todos os momentos da minha vida.

Agradecimentos

À Professora Cláudia Werner, pelo apoio e paciência, sem os quais seria impossível a realização deste trabalho.

Ao professor Renato Cerqueira, pelos minutos valiosos de conversas sobre o tema da minha pesquisa.

A todo o pessoal do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ pela ajuda e, em especial, aos colegas da Equipe de Reutilização de Software, que sempre mantiveram um ambiente descontraído e de camaradagem.

Ao Contra-Almirante Cid Augusto Claro Junior, pela compreensão da minha ausência do CASNAV em alguns momentos ao longo do último ano, para poder cumprir as demandas do doutorado.

Este trabalho também contou com a valorosa ajuda do Henrique, durante o uso do Robô da Lego, e do Fábio, para aplicar esta pesquisa ao modelo de componentes iPOJO/OSGi.

Aos professores Cecília Rubira, Geraldo Xexéo e Toacy de Oliveira, por terem aceitado participar da banca e pelos valorosos comentários que, certamente, elevarão o nível do trabalho.

Por fim, agradeço à minha família, minha esposa Elizabeth e minha filha Ana Beatriz. Por mais que nos empenhemos, não há como evitar a ausência decorrente da participação em um curso de doutorado. A compreensão de vocês foi fundamental neste período atribulado.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

GERAÇÃO DE PLANOS DE RECONFIGURAÇÃO EM TEMPO DE EXECUÇÃO

Marco Eugênio Madeira Di Beneditto

Julho/2014

Orientador: Cláudia Mria Lima Werner

Programa: Engenharia de Sistemas e Computação

Considerando-se um sistema de software baseado em componentes e que necessita ser reconfigurado dinamicamente, em algumas situações a utilização de um conjunto pré-definido de regras de reconfiguração pode ser incapaz de lidar com a tarefa de reconfiguração. Se a geração da sequência de ações de reconfiguração, denominado de *plano de reconfiguração*, é feita em tempo de execução, ao invés de num instante prévio e de forma fixa, novas soluções podem ser geradas para tratar situações inesperadas, aproveitando-se novas oportunidades. No entanto, a geração deste plano implica a solução de novas questões como ser consistente em relação aos componentes, composições e conexões providos pelo modelo de componentes subjacente.

Nesta tese, apresentamos um procedimento de reconfiguração que gera um plano de reconfiguração em tempo de execução. Dadas duas configurações de software, a atual e a objetivo, um procedimento gera, se existir, um plano de reconfiguração que irá conduzir a evolução arquitetural entre essas configurações. O procedimento também pode gerar uma sequência de iniciação se a configuração atual estiver vazia. O procedimento baseia-se no conceito de Planejamento Automatizado, uma área da inteligência artificial. Como estudo de caso, o procedimento foi utilizado para o modelo de componentes FRACTAL e aplicado em dois diferentes sistemas de software baseados em componentes.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

GENERATION OF COMPOSITIONAL RECONFIGURATION PLANS AT RUNTIME

Marco Eugênio Madeira Di Beneditto

July/2014

Advisor: Cláudia Mria Lima Werner

Department: Systems Engineering and Computer Science

Considering a component-based software system that needs to be dynamically reconfigured, in some situations the use of a predefined set of reconfiguration rules may be unable to handle the reconfiguration task. If the generation of the sequence of reconfiguration actions, called *reconfiguration plan*, is made at runtime, instead of in a previous and fixed way, new solutions may be generated treating unexpected situations and taking advantage of new opportunities. However, the generation of this plan implies the solution of new issues like being architecturally-consistent in respect to the components, compositions and connections provided by the underlying component model.

In this paper, we present a reconfiguration procedure that generates a reconfiguration plan at runtime. Given two software configurations, current and the goal one, the procedure generates, if any, a reconfiguration plan that will drive the architectural evolution among these configurations. The procedure can also generate an initialization sequence if the current configuration is empty. The proposed procedure is based on the concept of Automated Planning, an Artificial Intelligence area. As a case study, the procedure was employed for the FRACTAL component model and applied it to two different software applications developed in a component based fashion.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A alteração das preferências e novos requisitos do usuário, as flutuações na conectividade da rede e o surgimento de novos dispositivos e serviços podem levar a necessidade de se alterar a configuração de um sistema de software em tempo de execução.

Segundo KRAMER e MAGEE (1990), reconfiguração dinâmica é a capacidade de um sistema de software ser alterado em tempo de execução sem interromper o processamento das partes do sistema que não são diretamente afetadas pela mudança. Esta capacidade é um atributo desejável em sistemas de software (OREIZY *et al.*, 1998) e, dado que uma reconfiguração é necessária, uma questão a ser resolvida é quando este ajuste será especificado. Se ele for definido preliminarmente, em tempo de desenvolvimento, o que geralmente é feito por meio de um conjunto de regras como em GARLAN *et al.* (2004), GEORGAS e TAYLOR (2008), CHENG e GARLAN (2012) e HUBER *et al.* (2014), novas opções de configuração podem ser deixadas de lado, pois estas opções são decorrentes de outros recursos que se tornaram disponíveis após a aplicação entrar em produção e, por isso, não puderam ser descritas previamente, ou mesmo tornar inviável algum destes ajustes preliminares pois um determinado recurso previsto e necessário deixou de existir.

Assim, tratar de como um software deve ser reconfigurado dinamicamente para se atingir uma nova configuração no momento em que for necessário, e não preliminarmente, trará mais oportunidades, pois permitirá considerar novas possibilidades para a reconfiguração necessária.

Este ajuste deve estar de acordo com a capacidade que o software possui de ser alterado que, segundo SALEHIE e TAHVILDARI (2009), é o QUE e ONDE o software pode ser ajustado. Um dos meios para se realizar essas adaptações seria manipular o comportamento da aplicação no nível da linguagem, estendendo o comportamento de uma classe existente, ou mesmo tentando introduzir uma nova versão de uma classe. Esta abordagem resulta em um mecanismo de reconfiguração que é específico para um domínio de aplicação em particular ou linguagem de

implementação.

Assim como já foi observado anteriormente por OREIZY *et al.* (1998), OREIZY *et al.* (1999) e KRAMER e MAGEE (2007), a arquitetura de software responde à exigência de um nível maior de abstração para descrever e efetivar reconfigurações que não são específicas de um determinado domínio. Estes benefícios são alcançados tratando-se o software como uma composição de blocos de software, ou componentes, independentemente de quem os criou.

A abordagem em nível arquitetural vem ao encontro de uma das formas de adaptação de um software. Segundo MCKINLEY *et al.* (2004), há duas formas de se ajustar um software: 1) *Adaptação por parâmetro* - onde são modificadas variáveis do programa que determinam um comportamento, a fim de ajustar o programa ao contexto; e 2) *Adaptação composicional* - que adiciona algoritmos e componentes estruturais ao sistema ou efetua a sua troca por outros mais adequados ao contexto atual.

Para SZYPERSKI (2002), um componente é uma unidade de composição com interfaces especificadas contratualmente e com dependências apenas de contexto. Ele pode ser implantado independentemente e está sujeito a composição por terceiros. Um ponto chave em qualquer metodologia de desenvolvimento baseado em componentes (DBC) é o seu modelo de componentes subjacente, que irá definir como são os componentes e como podem ser construídos, compostos, conectados e implantados. Segundo COUNCILL e HEINEMAN (2001), um componente de software é um elemento que está de acordo com um modelo de componentes e pode ser independentemente implantado e composto sem modificação, de acordo com um padrão de composição.

Dentre os modelos de componentes disponíveis, CRNKOVIC *et al.* (2011) os classifica em dois grupos: a) especializados; e b) de propósito geral. Os requisitos do domínio de aplicação, como sistemas embarcados, permeiam os modelos de componentes especializados e, consequentemente, deixam estes modelos pouco úteis em outros domínios. Já os de propósito geral fornecem mecanismos básicos para a especificação e composição dos componentes, sem assumir qualquer arquitetura específica.

Definindo-se o nível de abstração no qual as reconfigurações serão descritas, cabe agora prover o mecanismo que irá efetivá-las em tempo de execução. Em SALEHIE e TAHVILDARI (2009), os autores listam uma série de técnicas empregadas e que permitem que os artefatos de um sistema de software sejam alterados durante a execução, tais como: padrões de projeto, estilos arquiteturais, *middleware* e programação orientada a aspectos. Algumas destas técnicas e outras foram incorporadas em alguns modelos de componentes para prover a capacidade de reconfiguração dinâmica nestes modelos, como em: FRACTAL (BRUNETON *et al.* (2006)), iPOJO

(ESCOFFIER *et al.* (2007)) e OpenCom (COULSON *et al.* (2008)). Cada um destes modelos possui uma capacidade de ser reconfigurado, que envolve seus elementos estruturais, o estado destes elementos e as relações entre eles.

1.1 Motivação

Este trabalho é relevante para uma ampla gama de áreas de aplicação, em especial, aquelas que exigem alta disponibilidade em face de ambientes altamente dinâmicos ou com os requisitos do usuário em rápida evolução. Neste contexto, se incluem os sistemas de apoio ao comando e controle de um operação militar que, segundo ADAMS *et al.* (2008), são sistemas que proveem a consciência situacional para a decisão do comando, incluindo-se aqui os sistemas de resposta à crises, como desastres naturais e calamidades.

Numa operação em curso apoiada por sistemas computacionais deste tipo, o usuário pode dar diferentes prioridades aos objetivos deste sistema, privilegiando um subconjunto dos objetivos em relação a outros. Ele também pode identificar a necessidade de integração com outros sistemas computacionais, que proverão informações de maneira automatizada. Estas novas necessidades refletem a dinâmica do contexto e podem ter como consequência uma configuração de software diferente daquela em execução, devendo o sistema ser reconfigurado dinamicamente para continuar a apoiar a operação.

Atualmente, há pouco apoio para a geração de uma reconfiguração dinâmica em tempo de execução, pois a maioria dos mecanismos propostos emprega uma abordagem onde a decisão é estaticamente descrita e aplicada em tempo de execução, como as citadas anteriormente. As poucas abordagens que se propõem a gerar a reconfiguração no momento em que ela é necessária, empregam modelos de componentes especializados e cuja capacidade de reconfiguração é controlada pelo modelo de componentes, como em SYKES *et al.* (2008) e TAJALLI *et al.* (2010).

Além disso, nestes trabalhos os domínios de aplicação são mais restritos, como a robótica, e a simplicidade dos exemplos dificulta ou mesmo, inviabiliza, o seu uso para modelos de componentes gerais citados anteriormente.

1.2 Objetivos

Esta pesquisa trata da geração da reconfiguração dinâmica de um sistema de software baseado em componentes, considerando que esta reconfiguração é fundamentada na capacidade reflexiva provida pelo modelo de componentes empregado no desenvolvimento. Como visto anteriormente, a geração prévia da reconfiguração impõe uma

série de limitações, tanto nas configurações possíveis de serem obtidas quanto nas possibilidades de reconfiguração para se obter a nova configuração.

Assim, esta tese responderá à seguinte questão de pesquisa:

- **QP** : Considerando um sistema de software baseado em componentes, como gerar uma reconfiguração dinâmica a partir da nova configuração a ser assumida pelo sistema ?

1.3 Enfoque da Solução

Neste trabalho é apresentada uma proposta que, dado uma nova configuração de software, irá gerar a sequência de ações para obter esta nova configuração, a partir da atual. Esta geração se baseia numa representação (modelo) da capacidade de reconfiguração do modelo de componentes empregado e é realizada no momento em que se faz necessário ajustar o software a um novo contexto. Para que a reconfiguração produzida pelo modelo seja executável sobre o modelo de componentes, evitando falhas geradas pela sua execução, o modelo deve ser consistente com a realidade que ele representa permitindo gerar reconfigurações consistentes com a capacidade reflexiva do modelo de componentes

A principal contribuição deste trabalho é fornecer uma técnica declarativa, onde um modelo indica as tarefas que precisam ser feitas e um procedimento decide como empregar estas tarefas para reconfigurar sistemas baseados em componentes. A abordagem proposta para solucionar o problema é desenvolvida a partir de outros trabalhos mas, ao invés de empregar um modelo de componentes simplificado ou uma especificação teórica sem uma implementação executável, empregou-se um modelo de componentes já existente, de uso geral e maduro e capaz de desenvolver aplicações complexas.

O processo de geração de uma reconfiguração se baseia na ideia de um agente inteligente que, de posse de um modelo de causa e efeito, ao receber um objetivo emprega este modelo para obter a sequência de ações que o levará até este objetivo RUSSELL e NORVIG (2010). Este processo de busca por uma reconfiguração não está previamente descrito e será acionado sempre que uma nova configuração de software tiver que ser assumida pelo sistema reconfigurável.

A reconfiguração representa o ajuste de um software que irá levá-lo de uma configuração inicial para uma nova configuração, ou configuração objetivo. O modelo de causa e efeito representa a capacidade de reconfiguração de um modelo de componentes, onde o conjunto de ações de reconfiguração, suas precondições, efeitos e restrições são representados para serem empregados na busca por uma reconfiguração. Este modelo deve ser validado perante a capacidade de reconfiguração do

modelo de componentes representado, para que as reconfiguração geradas possam ser corretamente executadas sobre a aplicação.

1.4 Organização

O restante desta tese está organizada da seguinte forma. No Capítulo 2, é descrita a fundamentação teórica deste trabalho e no Capítulo 3, são tratados os trabalhos relacionados e respectivas limitações. No Capítulo 4, a abordagem proposta será apresentada e no Capítulo 5 ela será empregada numa aplicação. Finalmente, no Capítulo 6, são discutidos os resultados obtidos e os trabalhos futuros.

Capítulo 2

Reconfiguração Dinâmica de Sistemas Baseados em Componentes

A reconfiguração dinâmica está relacionada a alterações numa aplicação ocorridas em tempo de execução. O problema de geração da reconfiguração pode ser visto como uma decisão a ser tomada por um mecanismo responsável por alterar o sistema de software. A decisão deste mecanismo é determinar a sequência de ações que irá alterar uma configuração para se obter uma outra configuração.

Este capítulo trata dos fundamentos teóricos desta tese que apoiam a geração, em tempo de execução, da reconfiguração dinâmica de um sistema baseado em componentes. Na Seção 2.1 trata dos conceitos básicos sobre a decisão de uma reconfiguração. A Seção 2.2 trata da inserção de uma abordagem baseada em modelos num mecanismo de gerenciamento da reconfiguração. Na Seção 2.3, é vista a teoria de planejamento automatizado, uma técnica da abordagem baseada em modelo, e que irá gerar as ações que comporão uma reconfiguração. Por fim, a Seção 2.4 trata do uso de um modelo de componentes que permite este tipo de reconfiguração e de aspectos de reconfiguração dinâmica.

2.1 Conceitos Básicos de Tomada de Decisões

Uma alteração entre duas configurações pode ser representada por meio de um *Diagrama de Transição de Estados*, em que:

- um estado representa uma determinada configuração do software.
- uma transição representa uma reconfiguração entre dois estados quaisquer.

Na Figura 2.1, está ilustrado o diagrama de transição de estados que representa a reconfiguração decorrente da ação `conectar(`$comp_C$`,` $comp_S$`)` sobre o estado E_1. Esta ação conecta o componente $comp_C$ ao componente $comp_S$ e a sua execução,

sobre o estado E_1, onde $comp_C$ = servidorA e $comp_S$ = servidorB, irá gerar a configuração do estado E_2.

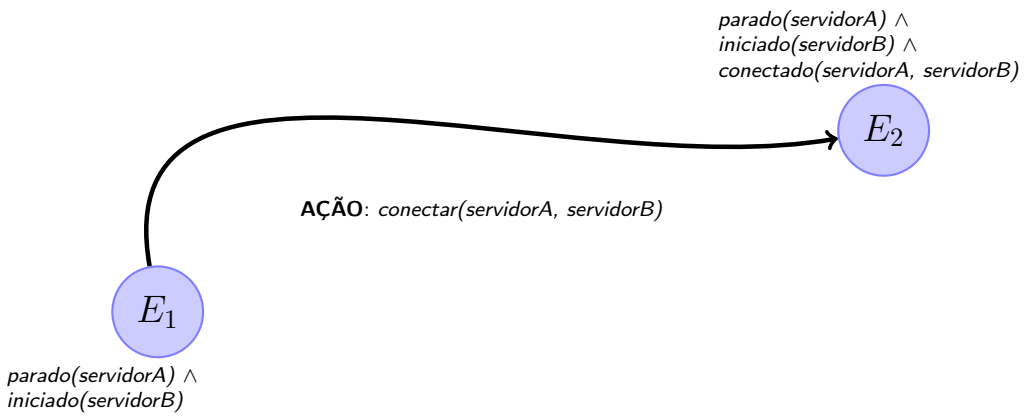


Figura 2.1: Transição entre configurações.

A decisão do mecanismo responsável por alterar o sistema de software é determinar a sequência de ações que irá alterar uma configuração para se obter uma outra configuração. Segundo GEFFNER (2010), este problema de tomada de decisão pode ser resolvido por três diferentes abordagens:

1. Abordagem baseada em programação: esta é a abordagem onde a decisão é previamente descrita, pois o problema é resolvido pelo programador, codificado e expresso por um programa de alto nível que é executado quando for necessário.

2. Abordagem baseada em aprendizado: a decisão não é tomada pelo programador e sim pela experiência do agente, por meio de um aprendizado por reforço ou pela experiência de um professor, num esquema supervisionado.

3. Abordagem baseada em modelo: a decisão não é obtida pelo aprendizado mas sim automaticamente derivada de um modelo de ações, sensores e objetivos.

Na literatura, encontram-se diferentes soluções para o mecanismo que irá gerar a reconfiguração e que se baseiam em uma destas três diferentes abordagens para a tomada de decisão. A mais utilizada é a abordagem programática e nela as transições entre diferentes configurações de software são descritas pelo projetista, geralmente por meio de um conjunto de regras. Porém, esta abordagem é limitada pois, como mencionado no Capítulo 1, pode ser que não haja regras aplicáveis a um estado E_1 ou que as regras não permitam gerar o estado desejado E_2.

Uma outra dificuldade em se empregar regras surge no momento da sua elaboração, quando o desenvolvedor deve descrever o que deve ser feito numa determinada

condição ou evento. Cada regra irá descrever uma transição entre as diferentes configurações de um software e, observando-se o diagrama de estados da Figura 2.1, quanto maior for o número de estados, maior o número de possíveis regras a serem descritas e o levantamento de todas as regras pode se tornar intratável. Além disso, quando ocorrem conflitos entre as regras, isto é, quando mais de uma regra pode ser aplicada numa determinada condição, o mecanismo que escolhe a regra deve decidir, por meio de algum critério, qual delas disparar.

Na abordagem baseada em aprendizado, o mecanismo tem que "aprender" a como empregar as ações de reconfiguração do software. Em geral, este aprendizado se dá por meio de uma recompensa - representada por uma variável quantitativa - que o mecanismo recebe ao efetuar uma ação, ou seja, o mecanismo de ajuste irá descobrir que reconfigurações são mais adequadas num determinado contexto. Por exemplo, em DOWLING e CAHILL (2004), um componente emprega um aprendizado por reforço colaborativo para decidir o melhor balanceamento de carga a ser feito. A maior flexibilidade desta abordagem vem acompanhada do maior desafio de implementar este tipo de solução, pois agora há uma etapa de aprendizado para depois empregá-lo na reconfiguração de um software. Nesta etapa de aprendizado, o mecanismo pode aprender algumas reconfigurações, mas ao longo do tempo, um software pode ter que assumir uma configuração que ainda não foi aprendida.

Sobre a abordagem baseada em modelo, ela transfere a complexidade para o mecanismo de decisão. Nela, dado que uma nova configuração deva ser obtida e a partir de um modelo de causa e efeito, a decisão do que deve ser feito é uma solução deste modelo. Esta decisão permite, reconfigurar o software, realizando a transição entre a configuração atual e a nova configuração, que é o objetivo pretendido.

Nesta pesquisa, a solução para se gerar uma reconfiguração dinâmica em tempo de execução será uma abordagem baseada em modelo, pois ela permite gerar uma reconfiguração sempre que necessária. Para ROTHENBERG (1989), um modelo representa a realidade para um dado propósito, sendo uma abstração da realidade, pois ele não pode representar todos os seus aspectos. No caso desta pesquisa, o modelo a ser empregado irá representar a capacidade de reconfiguração dinâmica de um modelo de componentes e o seu propósito é permitir obter a sequência de ações que irá alterar uma dada configuração, obtendo-se uma outra configuração que é a desejada.

Além das características do modelo de componentes, uma aplicação a ser reconfigurada deve ser colocada num estado que permita a ela retornar ao seu modo normal de funcionamento após ser reconfigurada, como já observado anteriormente em KRAMER e MAGEE (1990). Um modelo de componentes pode prover ações explícitas que alteram o estado de estruturas da aplicação, com o propósito de colocá-la no estado adequado para ser reconfigurada e depois retornar ao seu modo

normal. Neste caso, estas ações também podem estar representadas no modelo que será utilizado para se gerar a reconfiguração,

Por fim, um outro aspecto que deve ser tratado numa abordagem baseada num modelo é a sua corretude perante a realidade que ele representa. A solução do modelo, que permitirá reconfigurar um software, deve ser exequível e capaz de alterá-lo para se obter a configuração desejada. A reconfiguração não deve causar uma falha durante a sua execução por violar as restrições do modelo de componentes.

Este capítulo trata dos fundamentos teóricos desta tese que apoiam a geração, em tempo de execução, da reconfiguração dinâmica de um sistema baseado em componentes. A Seção 2.2 trata da inserção de uma abordagem baseada em modelos num mecanismo de gerenciamento da reconfiguração. Na Seção 2.3, é vista a teoria de planejamento automatizado, uma técnica da abordagem baseada em modelo, e que irá gerar as ações que comporão uma reconfiguração. Por fim, a Seção 2.4 trata do uso de um modelo de componentes que permite este tipo de reconfiguração e de aspectos de reconfiguração dinâmica.

2.2 Gerenciamento da Reconfiguração

Como destacado em MÜLLER *et al.* (2008), o mecanismo responsável por ajustar um software deve ser explicitado em termos de projeto ou ser claramente observável na implementação. Desta forma, o mecanismo passa a ser externo ao software que será reconfigurado por ele, como a abordagem externa citada em SALEHIE e TAHVILDARI (2009) e ilustrada na Figura 2.2.

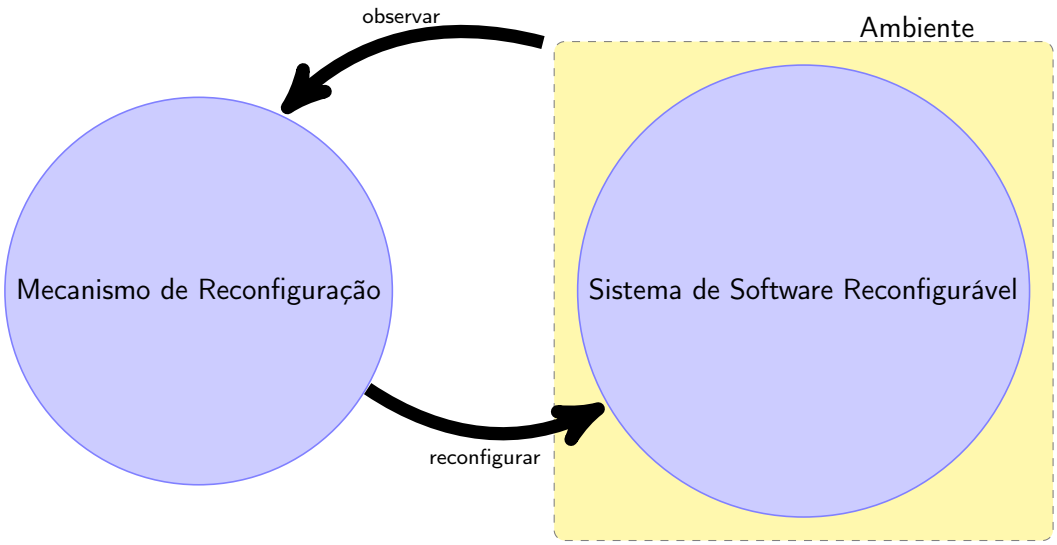


Figura 2.2: Mecanismo de reconfiguração externo ao sistema reconfigurável.

Nesta abordagem, o mecanismo de reconfiguração deve observar o ambiente, decidir uma reconfiguração e reconfigurar o software. Este mecanismo é responsável

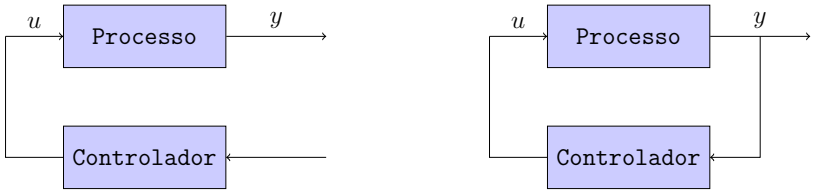
por uma série de decisões. A partir das observações provenientes do ambiente, que inclui o próprio software reconfigurável, o mecanismo irá decidir, dentre outras coisas, quando uma alteração se faz necessária. A partir disso, também será decidido o que deve ser alterado e como esta alteração será efetuada por meio de ações de reconfiguração apropriadas. Este mecanismo tem sido tratado usando-se a teoria de engenharia de controle, como citado em DOBSON *et al.* (2006) e CHENG *et al.* (2009) e também pode ser visto por meio do conceito de agente inteligente, como o ciclo de decisão **MAPE-K** descrito em KEPHART e CHESS (2003).

Para a estrutura do Software Reconfigurável, esta pesquisa trata de sistemas baseados em componentes cujo modelo provê a capacidade de ser reconfigurado.

2.2.1 Engenharia de Controle

Segundo DEAN e WELLMAN (1991), intuitivamente um *processo* é uma série de mudanças no estado do mundo e *controlar* um *processo* consiste em executar certas mudanças no estado do mundo, a fim de influenciar o *processo*.

Já na Figura 2.3a, é ilustrado um **Sistema de Controle a Malha Aberta**, onde o **Controlador** não recebe informações do **Processo**, ou seja, um controlador malha aberta não observa a saída do processo controlado. Um exemplo é um sinal de trânsito que abre e fecha o semáforo de acordo com o tempo e as alterações do sinal não são observadas pelo controlador.



(a) Malha Aberta (b) Malha Fechada

Figura 2.3: Malhas de Controle.

O termo realimentação, retroação ou *feedback* (em inglês) se refere a situação na qual dois ou mais sistemas estão conectados, tal que cada sistema influencia o outro e, desta forma, têm a dinâmica fortemente acoplada (ASTROM e MURRAY (2008)).

Segundo OGATA (2010), um **Sistema de Controle a Malha Fechada** ou **Sistema de Controle com Retroação**, ou ainda **Realimentação**, como o ilustrado na Figura 2.3b, é um sistema com um **Controlador**, que mede o valor da *variável controlada y* de um processo a ser controlado e aplica o valor conveniente na *variável manipulada u*, de modo a afetar o valor da variável controlada. A *variável controlada y* é a grandeza ou a condição medida e controlada e a *variável manipulada u* é

a grandeza ou a condição variada pelo controlador. A variável controlada pode ser medida continuamente ao longo do tempo ou em instantes de tempo determinados e, neste último caso, a frequência em que é efetuada esta medida denomina-se de *frequência de amostragem.*

Fundamentalmente, a teoria de controle deve escolher as ações ao longo do tempo para influenciar um processo, baseado em algum modelo deste processo, como as funções de transferência. Um exemplo de aplicação deste paradigma é descrito em DIAO *et al.* (2005). Neste trabalho, um controlador determina automaticamente o número máximo de usuário (*MaxUsers*) a partir das chamadas remotas de procedimentos no servidor (*RIS*), e a função de transferência utilizada descreve como a entrada *MaxUsers* é transformada na saída *RIS*.

Entretanto, os objetivos podem mudar ou mesmo o mecanismo de controle pode ter que escolher dentre duas diferentes maneiras de influenciar o processo. Estes novos cenários podem ser tratados de uma maneira mais ampla com a visão de agentes inteligentes.

2.2.2 Agentes Inteligentes

Na área de Inteligência Artificial, um *agente inteligente* percebe o seu *ambiente* por meio de sensores e age sobre este *ambiente* por meio de atuadores RUSSELL e NORVIG (2010). Este processo se repete inúmeras vezes ao longo do tempo, onde o agente emprega um mecanismo de tomada de decisão que, dentre outras coisas, determina que ações deve executar a partir dos dados dos sensores. Ou seja, o agente desempenha o papel do mecanismo de reconfiguração, decidindo e executando as reconfigurações no software.

Uma das questões centrais para um comportamento inteligente é a tomada de decisão, ou seja, selecionar qual a próxima ação a ser tomada para se atingir o objetivo. Por exemplo, considerando um agente responsável por controlar um software, na Tabela 2.1 estão ilustradas as possíveis **Percepções**, **Ações**, **Objetivos** do agente e o **Ambiente** no qual ele irá operar. Este agente poderia ter como objetivo adaptar o software para atender a novos requisitos, por meio da reconfiguração dos componentes que podem ser utilizados na configuração de um software.

O comportamento do agente depende das suas percepções e é a partir delas que o agente decide e executa um conjunto de ações. Matematicamente falando, o comportamento do agente é descrito por uma função que mapeia qualquer sequência de percepções para uma ação. Uma maneira simples de efetuar este mapeamento é por meio de uma tabela em que cada sequência percebida pelo agente é mapeada para um conjunto de ações, ou seja, a tabela especifica qual a ação que o agente deve tomar em resposta ao que ele percebe. Neste caso, a construção da tabela pode ser

Percepções	a) Configuração de software atual b) Regras de composição dos componentes c) Estilos arquiteturais d) Componentes disponíveis
Ações	a) Instanciar ou Destruir um componente b) Conectar ou Desconectar dois componentes c) Iniciar ou Parar um componente
Objetivos	a) Melhorar o desempenho b) Atender a requisitos
Ambiente	a) Software Reconfigurável b) Número de requisições ao longo do tempo

Tabela 2.1: Descrição de um agente por meio das Percepções, Ações, Objetivos e Ambiente.

uma tarefa intratável, devido ao grande número de possibilidades, e o agente não tem capacidade de se adaptar a mudanças pois, caso uma dada percepção não esteja mapeada, o agente não irá desempenhar qualquer ação.

Segundo RUSSELL e NORVIG (2010), os modos como este *mapeamento* pode ser feito foram classificados nos seguintes tipos de agentes:

1. Agente com reflexo simples - este tipo de agente decide a ação a ser executada baseando-se na percepção atual, ignorando o histórico. O mapeamento entre a percepção e a ação é feito por meio de um conjunto de regras de condição-ação, também denominadas de regras de produção. Este tipo de agente é de fácil implementação e tem sucesso quando o ambiente é totalmente observável e se a decisão correta puder ser tomada apenas com base na situação corrente. As desvantagens são o tamanho do conjunto de regras, que pode ser bastante grande e a possibilidade de laços *loops* infinitos quando o ambiente é parcialmente observável.

2. Agente com reflexo baseado em modelo - neste tipo, o estado atual do agente é obtido com base na sequência de percepções e ações. O agente deve possuir dois tipos de conhecimento: a) a informação de como o mundo evolui, independente do agente, que é denominada de *modelo do mundo*; e b) a informação de como as ações do agente afetam o mundo. Determinado o estado em que ele se encontra, o agente escolhe a ação da mesma forma que o agente com reflexo simples.

3. Agente baseado em objetivos - o conhecimento sobre o estado atual do ambiente pode não ser o suficiente para decidir o que fazer. Um agente baseado em objetivos possui a informação sobre o objetivo, ou seja, a descrição das situações desejáveis e que, em conjunto com a informação de como as ações do agente afetam o mundo, lhe permite escolher dentre diferentes ações possíveis aquela que atingirá o objetivo ou uma sequência delas. Neste agente, a tomada

de decisão sobre que ação executar compreende considerações no futuro como, "o que irá ocorrer se eu (agente) fizer isto ou aquilo" e "o que me (agente) deixará feliz".

4. Agente baseado em utilidade - o agente baseado em objetivos sabe discernir estados objetivos de estados não-objetivos. Entretanto não sabe diferenciar, dentre os estados, qual deles é o melhor ou o preferido. Dizer que um estado é preferido em relação ao outro significa dizer que um estado é mais *útil* para o agente que o outro. A *utilidade* é uma função que mapeia um conjunto de estados em um número real expressando o grau de preferência associado e um agente que emprega esta abordagem escolhe a ação que maximiza a utilidade esperada dos resultados da ação. Esta característica é bastante importante quando há objetivos conflitantes e surge a necessidade de compará-los ou quando há diferentes maneiras de se chegar a um objetivo e há a necessidade de escolher uma delas.

5. Agente que aprende - são agentes que melhoram o seu desempenho por meio da experiência. Este agente possui um componente responsável por realizar melhorias no seu desempenho. Este componente utiliza uma realimentação proveniente da crítica do que ele está fazendo, a fim de modificar a realização das percepções e a escolha das ações.

Uma diferença fundamental no processo de tomada de decisão entre os agentes baseados em reflexo e os baseados em objetivo e utilidade é que, nos agentes baseados em reflexo, o objetivo não está explícito, pois o agente não se *preocupa* com o estado que ele precisa alcançar, ele apenas executa ações descritas nas regras e espera chegar até este estado.

Na computação autonômica, esta visão de agente inteligente é materializada por meio do ciclo de decisão **MAPE-K**, proposto por KEPHART e CHESS (2003) e ilustrado na Figura 2.4. O Gerente Autonômico cumpre a função do mecanismo de reconfiguração, ajustando o software quando for necessário.

Esta pesquisa emprega o agente baseado em objetivos para gerar as reconfigurações necessárias. O **objetivo** a ser alcançado é uma nova configuração arquitetural e a **decisão** é como obter esta nova configuração, ou seja, a sequência de ações que precisam ser executadas para se atingir o objetivo. Outras decisões que também são tomadas, como o instante de tempo em que uma alteração se faz necessária, ou mesmo, qual a nova configuração a ser alcançada, não são objetos desta pesquisa.

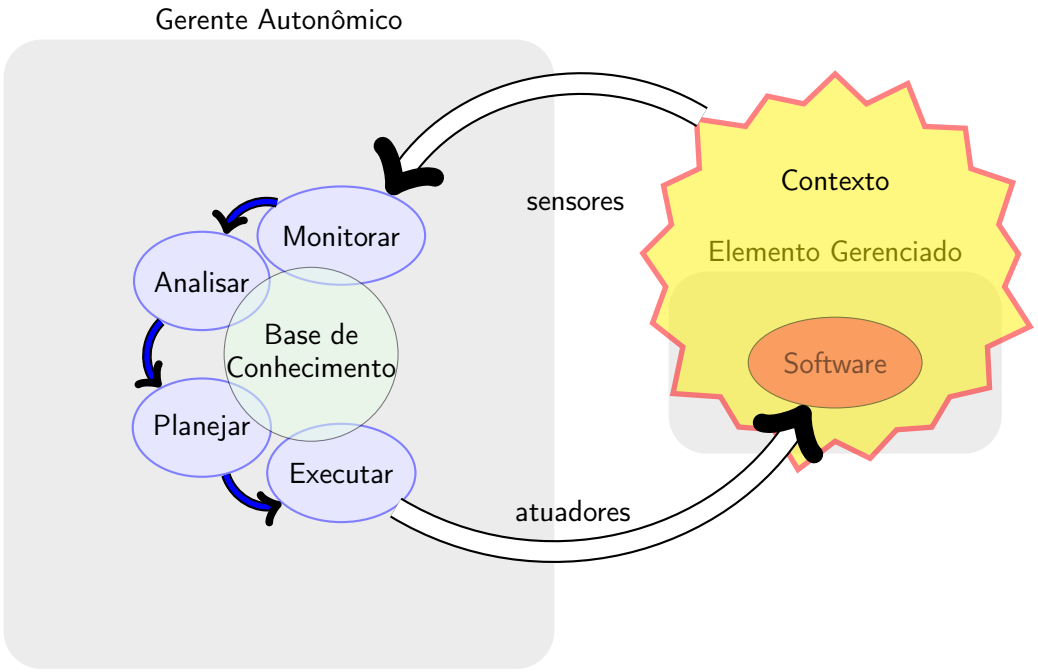


Figura 2.4: Atividades do ciclo de decisão **MAPE-K** (adaptado de KEPHART e CHESS (2003)).

2.3 Planejamento Automatizado

O *planejamento automatizado* é uma abordagem que se baseia num modelo e também é uma implementação do agente baseado em objetivos, descrito na Seção 2.2.2. Em NAU *et al.* (2004), os autores definem *Planejamento* como o processo de decidir uma escolha de ações a fim de atingir algum objetivo, a partir de um estado inicial.

Numa formulação *clássica* e simples, em WELD (1999), o problema de planejamento é definido a partir de três entradas:

1. uma descrição do estado inicial do mundo em alguma linguagem formal;
2. uma descrição da meta do agente, i.e., o objetivo desejado, em alguma linguagem formal; e
3. uma descrição das ações possíveis que podem ser realizadas, novamente em alguma linguagem formal.

Cada ação possui precondições e efeitos. As precondições são o que deve ser válido num estado a fim de que a ação possa ser aplicável neste estado. Os efeitos são as mudanças produzidas no estado após a aplicação da ação. Um *planejador* é um programa que realiza a busca por uma sequência de ações, denominada de *plano*, que quando executadas num mundo que satisfaça o estado inicial atingirão o estado objetivo.

Usualmente, o conjunto de ações disponíveis é um modelo de causa e efeito, e representa o *domínio*, sendo aplicável a um grande número de problemas de um certo tipo. Por meio deste modelo, pode-se prever o que ocorrerá se for tomada uma determinada ação, sem precisar fazê-la e depois observar o estado obtido.

2.3.1 Abordagem clássica

O Planejamento pode ser descrito por meio de um modelo conceitual que descreve seus principais elementos. Em geral, este modelo é um sistema de transição de estados, ou seja, é uma tupla $\mathbf{\Sigma} = (S, A, E, \gamma)$, onde:

- $S = \{s_1, s_2, ...\}$ é um conjunto finito ou enumerável de estados.
- $A = \{a_1, a_2, ...\}$ é um conjunto finito ou enumerável de ações.
- $E = \{e_1, e_2, ...\}$ é um conjunto finito ou enumerável de eventos.
- $\gamma : S \times A \times E \rightarrow 2^S$ é uma função de transição de estados.

Com o intuito de simplificar este modelo, a abordagem clássica de planejamento supõe oito premissas restritivas:

1. **$\mathbf{\Sigma}$ finito** - O número de estados é finito.
2. **$\mathbf{\Sigma}$ é totalmente observável** - Tem-se um completo conhecimento sobre o estado de Σ, isto é, não há dúvida sobre qual o estado em que Σ está.
3. **$\mathbf{\Sigma}$ é determinístico** - Para todo estado s, e para toda ação a, se uma ação é aplicável a um estado, sua aplicação conduzirá a apenas um único estado.
4. **$\mathbf{\Sigma}$ é estático** - o sistema Σ não possui uma dinâmica interna; ele permanece no mesmo estado até que uma ação seja realizada, ou seja, não há eventos e o conjunto $E = \varnothing$.
5. **Metas restritas** - o planejador trata apenas de metas restritas que são especificadas como um estado meta explícito s_g ou um conjunto de estados meta S_g. Metas estendidas, tais como estados a serem evitados ou mesmo visitados, não são tratadas.
6. **Planos sequenciais** - a solução do problema, i.e., *o plano*, é uma sequência finita de ações linearmente ordenada.
7. **Templo implícito** - ações e eventos não possuem duração, eles são transições de estado instantâneas.

8. **Planejamento offline** - o planejador não se preocupa com qualquer mudança que possa ocorrer enquanto ele está planejando, ou seja, ele planeja para o estado inicial dado e os estados meta, sem considerar qualquer dinâmica.

Assim, na abordagem clássica, o domínio de Planejamento é uma tupla $\mathbf{\Sigma} = (S, A, \gamma)$, onde:

- $S = \{s_1, s_2, ...\}$ é um conjunto finito ou enumerável de estados.
- $A = \{a_1, a_2, ...\}$ é um conjunto finito ou enumerável de ações.
- $\gamma : S \times A \to S$ é uma função de transição de estados.

Com o intuito de exemplificar o emprego do planejamento automatizado na tarefa de reconfigurar um software, será descrito o esquema de representação clássica. Este esquema emprega uma notação derivada da lógica de primeira ordem. Os estados são representados como conjuntos de átomos lógicos que são verdadeiros ou falsos segundo alguma interpretação. As ações são representadas por *operadores de planejamento* que alteram os valores verdade destes átomos.

Estados

Considerando uma linguagem de primeira ordem $\mathcal{L}$, na qual há um número finito de *símbolos predicativos* e *símbolos de constantes*, e não há funções, um estado s é um conjunto de átomos de $\mathcal{L}$. Como não há funções em $\mathcal{L}$, o conjunto S de todos os estados possíveis é finito.

Por exemplo, uma formulação para um problema de reconfiguração de um software, no qual há dois componentes de software (c_1, c_2) e três interfaces (i_1, i_2, i_3), o conjunto de constantes é $\{c_1, c_2, i_1, i_2, i_3\}$. Neste problema, os símbolos predicativos são:

- $ativo(x)$: significa "x está ativo";
- $conectado(x, y, i)$: significa "x está conectado a y na interface i";
- $requer(x, y)$: significa "x requer a interface i";
- $prove(y, i)$: significa "y provê a interface i";

Operadores e Ações

A função de transição de estados γ é definida genericamente por um conjunto de operadores de planejamento que são instanciados em ações. Um *operador de planejamento* é uma tripla $o = (nome(o), precond(o), efeitos(o))$, cujos elementos são:

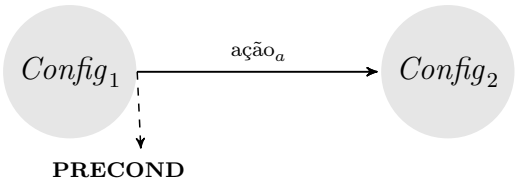


Figura 2.5: Uma ação de reconfiguração.

- $nome(o)$, é o nome do operador, na forma $n(x_1, ..., x_n)$, onde n é o símbolo denominado de *símbolo operador*, $x_1, ..., x_n$ são os símbolos de variáveis que aparecem em qualquer lugar em o, e n é único, ou seja, não há dois operadores com o mesmo símbolo operador.

- $precond(o)$ e $efeitos(o)$, são a *precondição* e os *efeitos* de o, respectivamente. Ambos são formados por conjuntos de átomos.

A partir da definição de operador e considerando o domínio de reconfiguração de um software, pode-se ter os seguintes operadores descritos na Tabela 2.2, onde x, y e i são variáveis, o símbolo $\wedge$ significa a conjunção lógica "e" e o símbolo $\neg$ significa a negação.

NOME	**DESCRIÇÃO**	**PRECOND**	**EFEITOS**
$ativar(x)$	ativa o componente x	$\neg ativo(x)$	$ativo(x)$
$desativar(x)$	desativa o componente x	$ativo(x)$	$\neg ativo(x)$
$conectar(x, y, i)$	conecta o componente x a y por meio da interface i	$ativo(x) \wedge$ $ativo(y) \wedge$ $prove(y, i) \wedge$ $requer(x, i) \wedge$ $\neg conectado(x, z, i)$	$conectado(x, y, i)$
$desconectar(x, y, i)$	desconecta o componente x de y na interface i	$conectado(x, y, i)$	$\neg conectado(x, y, i)$

Tabela 2.2: Operadores de planejamento para um domínio de reconfiguração

Na Tabela 2.2, o operador $conectar(x, y, i)$ tem como precondição: o componente que requer a interface - representado pela variável x - estar ativo, não estar conectado a um outro componente por meio da interface requerida i e esta interface ser provida por um outro componente - representado pela variável y. Já o componente que provê a interface ele também deve estar ativo.

Se o operador for instanciado por uma ação como $conectar(c_1, c_2, i_3)$, a sua execução efetuará a transição para um outro estado onde o predicado $conectado(c_1, c_2, i_3)$ é verdadeiro.

2.3.2 Geração do plano

A partir dos conceitos anteriores, o problema de Planejamento se reduz ao seguinte enunciado:

> Dado um domínio $\mathbf{\Sigma}$, um estado inicial s_0 e um subconjunto de estados meta S_g, ache uma sequência de ações $\langle a_1, a_2, \ldots, a_k \rangle$ correspondendo a uma sequência de transição de estados $\langle s_0, s_1, \ldots, s_k \rangle$ tal que $s_1 \in \gamma(s_0, a_1)$, $s_2 \in \gamma(s_1, a_2)$, $\ldots$, $s_k \in \gamma(s_{k-1}, a_k)$ e $s_k \in S_g$.

A expressão $s_1 \in \gamma(s_0, a_1)$ significa que há uma transição a partir do estado s_0 chegando no estado s_1 por meio da ação a_1. Dado um problema de planejamento clássico, um *planejador* consiste num procedimento que efetua uma busca tentando encontrar um caminho entre o estado inicial s_0 até um estado meta $g \in S_g$. Se este caminho for encontrado, o procedimento retorna a sequência de ações que define este caminho e que representa o plano; caso contrário, o planejador devolve falha.

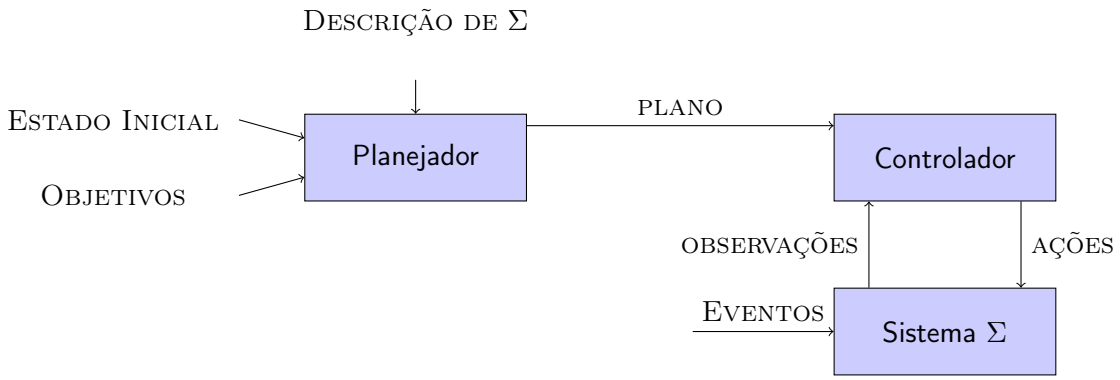


Figura 2.6: Um modelo conceitual para o planejamento - adaptado de NAU *et al.* (2004)

Um modelo conceitual para o planejamento pode ser visto na Figura 2.6. Um planejamento *offline* é obtido pela seguinte interação entre os três componentes:

- Um PLANEJADOR recebe como entrada a descrição do sistema Σ, o estado inicial, o OBJETIVO e sintetiza um plano para o CONTROLADOR.
- Um CONTROLADOR recebe como entrada o plano, observa o estado atual do sistema e fornece como saída a ação aplicável aquele estado e de acordo com o plano.
- Um sistema Σ evolui como especificado pela sua função de transição de estados γ, e de acordo com os EVENTOS e AÇÕES que ele recebe.

Neste modelo de planejamento, o controlador trabalha *online* com o sistema Σ, diferentemente do planejador, que não está diretamente conectado a Σ, pois o planejador se baseia num modelo do sistema, juntamente com um estado inicial para o problema de planejamento e o objetivo desejado. O planejador não está "preocupado" com o estado atual do sistema no momento em que o planejamento ocorre, mas com os estados nos quais o sistema poderá estar quando o plano está em execução.

Para situações mais reais, este modelo pode ser melhorado, intercalando planejamento, atuação e incluindo supervisão, revisão do plano e mecanismos de replanejamento, sendo necessária uma realimentação entre o planejador e o controlador, que pode ser feita por uma saída do CONTROLADOR para realimentar o PLANEJADOR. Esta realimentação retorna ao planejador o estado da execução do plano, permitindo o planejamento dinâmico ou o replanejamento.

Na prática, algumas das premissas anteriores para o modelo restrito da abordagem clássica são inadequadas para alguns domínios reais, como a suposição de determinismo. Pode-se citar alguns motivos para que o domínio se comporte de maneira não-determinística: a primeira é a possibilidade de ocorrência de *eventos* que correspondem a dinâmica interna do sistema. Eventos causam transições de estado, assim como as ações, mas não são controlados pelo executor do plano (o controlador na Figura 2.6). Por exemplo, a falha repentina de um componente de software. A segunda é que uma ação pode ter um efeito incerto, como um serviço Web que deve ser chamado mas que pode estar indisponível no momento.

Geralmente, um algoritmo de planejamento tem que considerar um grande número de possíveis ações e isto torna o espaço de busca muito grande, com uma complexidade computacional exponencial. Para tratar desta alta complexidade, ao invés de considerar qualquer ação possível, alguns planejadores empregam uma descrição de domínio específica para o Domínio de Planejamento, como a Rede de Tarefas Hierárquica (*Hierarchical Task Network* - HTN).

2.3.3 Planejamento Hierárquico

A descrição de domínio numa Rede de Tarefas Hierárquica se aproveita da natureza hierárquica inerente a muitos problemas práticos. Nesta tese, este tipo de planejador foi escolhido pois é um planejador independente de domínio, possui uma semântica bastante rica para a descrição do domínio, apresenta bom desempenho em diferentes domínios e possui implementações abertas e algoritmos corretos, o que é uma característica importante para a corretude dos planos gerados. Ele é amplamente utilizado em diversas áreas como: planejamento para montagem dos foguetes Ariane-V da Agência Espacial Européia (ESA) ARENTOFT *et al.* (1992),

web semântica SIRIN *et al.* (2004) e, em jogos de computador, como citado em KELLY *et al.* (2007).

Seu sucesso deve-se parcialmente aos métodos HTN, que fornecem um meio conveniente de representar "receitas" para resolver partes de um problema, correspondentes ao modo como humanos especialistas de domínio raciocinam ao planejar.

O domínio de planejamento é um conjunto de *métodos*, onde um método é uma prescrição de como solucionar um pequeno problema por meio da decomposição em subtarefas. Um método representa um meio de se decompor uma tarefa, expandindo-a num conjunto de subtarefas (tarefas menores). Uma tarefa pode ser composta ou primitiva. Uma tarefa composta será tratada por um método de decomposição e uma tarefa primitiva representa o operador de planejamento.

Uma versão simples de um método possui três partes:

- A tarefa para a qual o método se aplica;
- A precondição que deve ser verdadeira num estado para se aplicar uma determinada decomposição daquele método; e
- As subtarefas que precisam sem realizadas para se realizar a decomposição daquele método.

Um método pode ter diferentes decomposições e o planejador escolhe aquela que é aplicável naquele estado, verificando se as precondições são verdadeiras para a decomposição. A fim de ilustrar um método, na Figura 2.7 foi listado o método `destroySingle`, descrito na notação de um planejador HTN.

O método `destroySingle` resolve a remoção de um componente simples (não composto) de uma configuração. Este método possui duas possíveis decomposições. A primeira, identificada por `_0destroySingle`, é aplicável a um estado no qual um componente `?comp` está presente na configuração inicial $Config_{ini}$, mas não está presente na configuração objetivo $Config_{goal}$. Este estado é descrito pela expressão lógica: `(component ?comp single)` $\wedge$ `(active ?comp)` $\wedge$ `(not (to-active ?comp))`. Assim, o componente identificado pela variável `?comp` deve ser removido. Na segunda forma de decomposição em tarefas, identificada por `_9destroySingle`, a precondição significa que não há mais componentes simples a serem removidos.

A decomposição `_0destroySingle` contém cinco subtarefas:

Tarefa 1 : `(removeBindingTo ?comp)` - linha 12.

Tarefa 2 : `(removeBindingFrom ?comp)` - linha 13.

Tarefa 3 : `(removeSingleFromAllParent ?comp)` - linha 14.

Tarefa 4 : `(!removeComponent ?comp)` - linha 15.

Tarefa 5 : `(destroySingle)` - linha 16.

```
(:method (destroySingle)

  _0destroySingle
  ; PRECONDITIONS single component to be removed
  (
    (component ?comp single)
    (active ?comp)
    (not (to-active ?comp))
  )
  ; TASK DECOMPOSITION
  (
    (removeBindingTo ?comp)
    (removeBindingFrom ?comp)
    (removeSingleFromAllParent ?comp)
    (!removeComponent ?comp)
    (destroySingle)
  )

  _9destroySingle
  ; PRECONDITIONS no more components to be removed
  (
    (not (
      (component ?comp single)
      (active ?comp)
      (not (to-active ?comp))
    ))
  )
  ; TASK DECOMPOSITION
  ()
); END (:method (destroySingle)
```

Figura 2.7: A descrição textual do método `destroySingle`.

Se uma determinada decomposição for aplicável, o algoritmo irá seguir a ordem das tarefas contidas nesta decomposição. A tarefa **4** é um exemplo de tarefa primitiva, que representa o operador `(!removeComponent ?comp)` - o ponto de exclamação (**!**) indica um operador na notação empregada. Por fim, a tarefa **5** irá causar uma nova decomposição desta tarefa por meio do mesmo método, como numa recursão. Se não houver mais componentes a serem removidos, a decomposição `_0destroySingle` deixa de ser aplicável e o planejador irá testar se alguma outra é. Neste caso, a decomposição `_9destroySingle` passa a ser verdadeira, e o planejador a escolhe para cumprir o método `destroySingle`. Nesta decomposição, nada precisa ser feito (`TASK DECOMPOSITION` está vazio) e o método termina.

2.3.4 Corretude do Domínio

Um planejador que empregue um algoritmo correto irá gerar planos corretos em relação ao domínio de planejamento. Cabe agora saber se o domínio de planejamento, que no caso desta pesquisa representa a capacidade de reconfiguração do modelo de componentes, está correto em relação a realidade que ele representa. Em outras palavras, o domínio de planejamento deve ser consistente com o modelo de componentes para que um plano gerado não cause uma falha ao ser executado.

Na literatura, encontram-se duas maneiras de se verificar um domínio de planejamento: empregando métodos formais ou usando testes tradicionais BENSALEM *et al.* (2014). A verificação de modelos, em teoria, pode demonstrar a ausência de erros - se nenhum erro é encontrado então não há erros - em contrapartida, testes podem apenas mostrar a presença de erros - se nenhum erro é encontrado, não é garantido que não há erros.

A técnica de verificação de modelos irá olhar para todos os estados alcançáveis no domínio de planejamento e, consequentemente, é computacionalmente mais "cara" que apenas o teste. Como o número de estados é, em geral, exponencial em relação ao tamanho do domínio, apenas domínios moderados podem ser tratados pelas técnicas de verificação de modelos para uma verificação exaustiva. Um outro aspecto é o formalismo empregado para descrever um domínio de planejamento, tal como o empregado no planejador HTN utilizado nesta pesquisa. A tradução entre a especificação nesta linguagem e a empregada num verificador de modelos ou é impossível ou pode gerar formar artificiais e resumidas. Por fim, o emprego desta técnica envolve descrever novamente o modelo, mas agora na linguagem do verificador, e isto levaria a uma outra questão que é se este modelo do verificador é correto perante a realidade que ele representa.

Diante disso, esta pesquisa adotará o seguinte processo para validar o domínio de planejamento:

a. verificar se os operadores - precondições e efeitos - estão de acordo com a capacidade de reconfiguração do modelo de componentes utilizado; e

b. verificar se o plano gerado obedece a relação de ordem existente entre as ações de reconfiguração do modelo de componentes adotado.

c. testar os métodos individualmente com casos de testes positivos.

A primeira atividade irá verificar se um operador de planejamento não será instanciado numa ação de reconfiguração inaplicável, e se os efeitos previstos no operador geram transições de estado - reconfiguração - corretas. A segunda atividade irá mostrar se o plano como um todo é executável. Esta relação de ordem entre

ações ficará mais clara ao ser descrito o domínio para o modelo de componentes utilizado. A terceira atividade visa verificar os métodos com casos de testes positivos, que devem resultar num plano de reconfiguração para a configuração objetivo pretendida.

2.4 Modelos de Componentes Reflexivos

Como já mencionado na Seção 1, se o nível de abstração considerado para decidir sobre uma reconfiguração fosse de mais baixo nível, como o algorítmico ou a linguagem de programação, o mecanismo controlador seria específico a um domínio de aplicação particular ou a uma linguagem de programação, além de ter que lidar com uma série de aspectos específicos desta linguagem, o que poderia comprometer a escalabilidade da solução e o seu emprego em outras soluções adaptáveis. Assim, os benefícios em se considerar a adaptação no nível arquitetural são:

- a possibilidade de ser empregada em vários domínios de aplicação devido a sua generalidade;
- estar num nível de abstração mais elevado que o nível algorítmico;
- ter o potencial de escalar para grandes sistemas; e
- possibilidade de integração com mecanismos formais de descrição arquitetural já existentes.

De acordo com o padrão ANSI/IEEE 1471-2000, a arquitetura de software define os elementos chave do sistema, os relacionamentos entre estes elementos e o ambiente, e os princípios que governam seu projeto e evolução, descrevendo as decisões de projeto feitas sobre um sistema.

Segundo TAYLOR *et al.* (2009), a relação entre arquitetura e implementação é um problema de *mapeamento*, onde os conceitos definidos no nível arquitetural devem ser diretamente conectados aos artefatos no nível de implementação, sendo que esta correspondência não é necessariamente de um para um. Os autores citam que um modo importante de reduzir a *distância* entre conceitos arquiteturais e as tecnologias de implementação de sistemas é empregar um *arcabouço de implementação de arquitetura*. Para os autores, a definição deste arcabouço é:

> Uma peça de software que age como uma ponte entre um dado estilo arquitetural e um conjunto de tecnologias de implementação. Este arcabouço provê elementos chave do estilo arquitetural na forma de código, de maneira a ajudar os desenvolvedores a implementar sistemas que estejam em conformidade com as prescrições e restrições do estilo.

Um *arcabouço de implementação de arquitetura* é uma tecnologia que apoia os desenvolvedores na tarefa de implementar em conformidade com um estilo arquitetural, e provê aos desenvolvedores serviços de implementação que não estão disponíveis nativamente na linguagem de programação ou no sistema operacional subjacente. Segundo os autores, o conceito de *middleware* e *modelo de componentes* são exemplos de arcabouços.

A proposta de componentes de software foi criada para suprir a necessidade de desenvolver sistemas de forma mais rápida e com menor custo por McIlroy em MCILROY (1968). O Desenvolvimento Baseado em Componentes (DBC) é uma alternativa para apoiar a adaptação composicional, como citado em MCKINLEY *et al.* (2004), onde o desenvolvedor pode adicionar, remover ou reconfigurar os componentes em tempo de execução. Segundo LAU e WANG (2007), não há uma terminologia universalmente aceita para o DBC, bem como não há um critério padrão para o que constitui um componente. Uma definição amplamente aceita para *componente* é citada em SZYPERSKI (2002), na qual:

> Um *componente* de software é uma unidade de composição com interfaces especificadas contratualmente e apenas dependências de contexto. Um *componente* de software pode ser implantado independentemente e está sujeito a composição por terceiros.

Um ponto chave em qualquer metodologia de DBC é o seu modelo de componentes subjacente, que irá definir como são os componentes e como podem ser construídos, compostos e implantados. Em WEINREICH e SAMETINGER (2001), os autores definem um modelo de componentes como um conjunto de padrões para a implementação, interoperabilidade, personalização, composição, evolução e implantação de componentes. Este modelo também especifica padrões para a implementação do modelo de componentes associado, ou seja, o conjunto dedicado de software executável necessário para apoiar a execução dos componentes que estão em conformidade com o modelo. Desta forma, considerando o modelo de componentes, em COUNCILL e HEINEMAN (2001), os autores definem que:

> Um componente de software é um elemento que está de acordo com um modelo de componentes e pode ser independentemente implantado e composto sem modificação, de acordo com um padrão de composição.

A capacidade de uma aplicação baseada em componentes ser reconfigurada dinamicamente pode ser obtida pela própria aplicação desenvolvida sobre ele, ou pelo próprio modelo de componentes empregado. No primeiro caso, caberá ao desenvolvedor considerar na arquitetura da aplicação as funcionalidades que permitirão

apoiar esta capacidade e, neste caso, a solução tende a ser específica desta aplicação em particular.

No segundo caso, onde a solução de reconfiguração pertence ao modelo de componentes, ela poderá ser empregada em qualquer aplicação desenvolvida sobre ele, tornando-a mais geral. O modelo de componentes deve ser capaz de fornecer os meios para que estes ajustes sejam feitos em tempo de execução, levando ao emprego de mecanismos que forneçam a capacidade de *reflexão computacional*, que é um comportamento introspectivo que permite ao sistema manipular e raciocinar sobre si mesmo. Do ponto de vista do agente inteligente que desempenha a tarefa de reconfigurar um software, a *reflexão computacional* permite que ele obtenha as informações necessárias sobre o software reconfigurável, contribuindo para as *percepções*, e provê os meios para que o agente execute as ações de reconfiguração decididas, materializando os *atuadores*.

2.4.1 O modelo de componentes FRACTAL

A escolha do modelo de componentes a ser utilizado nesta pesquisa recaiu sobre o FRACTAL BRUNETON *et al.* (2006). O FRACTAL é um modelo aberto e de emprego geral, cujo propósito é implementar, implantar e gerenciar sistemas complexos. Ele provê sua capacidade reflexiva por meio de um conjunto explícito e bem definido de interfaces, e que também pode ser ampliado pelo desenvolvedor para se adequar às suas necessidades. As suas principais características são:

- Componentes compostos (*composite*) - componentes que contém subcomponentes para permitir uma visão da aplicação em diferentes níveis de abstração (hierarquia de componentes).
- Componentes compartilhados (*shared*) - que modela recursos compartilhados enquanto mantém o encapsulamento do componente (componente contido em mais de um supercomponente).
- Capacidade de introspecção - a fim de monitorar e controlar a execução de um sistema.
- Capacidade de reconfiguração - para implantar e dinamicamente configurar um sistema.

A especificação do FRACTAL é implementada por diferentes tecnologias e, na linguagem Java, ela é apoiada pelo *framework* JULIA, que incorpora o uso de *mixin* de classes para permitir a definição e combinação de componentes de controle arbitrários. Isto fornece ao desenvolvedor meios eficazes e extensíveis para lidar com diferentes aspectos transversais nos controladores.

Além destas características, a sua maturidade e emprego em diferentes projetos de software anteriores o qualificam para o desenvolvimento das aplicações de interesse a esta pesquisa, como as citadas na Seção 1.1.

Um componente é uma entidade encapsulada, possui uma identidade distinta e possui uma ou mais interfaces. Uma interface é um ponto de acesso ao componente que implementa o tipo de uma interface, ou seja, as operações que este ponto de acesso possui. Uma interface pode ser de dois tipos diferentes: a) "servidor", que corresponde a pontos de acesso que aceitam a chamada de operações de entrada; e b) "cliente", que corresponde a pontos de acesso que efetuam a saída de chamadas de operações. Ele é composto de uma membrana que, por meio de um conjunto de interfaces controladoras, permite a introspecção e reconfiguração de suas características e conteúdo, que consiste de um conjunto finito de outros componentes, ou subcomponentes.

A membrana pode ter interfaces externas e internas. Interfaces externas são acessíveis por fora do componente, enquanto as internas são acessíveis apenas pelos subcomponentes. Uma interface externa de um subcomponente é exportada como uma interface externa do seu supercomponente por meio de um *interceptador*, que pode interceptar a chamada de operações de entrada e saída e acrescentar comportamento ao tratar destas chamadas.

A comunicação entre componentes é possível apenas quando há um enlace entre suas interfaces. Há dois tipos de enlaces: o enlace simples (ou primitivo) e o enlace composto (ou coleção). Um enlace simples é feito entre uma interface cliente e uma interface servidor, no mesmo espaço de endereços (que pode ser visto como um componente). Um enlace significa que chamadas de operação emitidas pela interface cliente devem ser aceitas pela interface servidor. Um enlace composto é um conjunto de enlaces primitivos. Uma interface cliente pode ser mandatória, quando ela deve possuir um enlace quando o componente estiver executando, ou ser opcional, quando o componente pode estar em execução sem haver um enlace nesta interface cliente. O componente estar executando, ou não, está relacionado ao seu ciclo de vida e ele está em execução no estado *iniciado* e não está em execução no estado *parado*.

A capacidade de reconfiguração de um componente no FRACTAL é provida pelas interfaces controladoras da membrana. A especificação do FRACTAL provê alguns exemplos de formas de controladores, que podem ser combinados e ampliados para gerar diferentes características reflexivas. Os controladores previstos na especificação são:

- *Attribute controller* - um atributo é uma propriedade de um componente. Esta interface permite obter e alterar uma propriedade.
- *Binding controller* - permite efetuar enlaces entre interfaces cliente e servidor

e desfazê-los por meio de enlaces primitivos.

- *Content controller* - permite listar, adicionar e remover sub componentes do conteúdo de um componente.

- *Life-cycle controller* - provê a capacidade de controlar o ciclo de vida de um componente, em apoio a reconfiguração dinâmica.

Configuração e Reconfiguração no FRACTAL

Como pode-se ver, um modelo de componentes como o FRACTAL possui uma série de conceitos que, segundo LÉGER *et al.* (2010), acarretam num conjunto de restrições que são classificadas em dois tipos:

- *restrições de integridade* - que representam as restrições em relação a combinações de elementos arquiteturais e seus estados; e

- *restrições das transições* - que significam as *pré* e *poscondições* das ações de reconfiguração.

Estas restrições também são separadas por níveis de abstração:

1. Nível de modelo, que corresponde ao nível do modelo de componentes.

2. Nível de perfil, que corresponde a restrições genéricas e comuns a um conjunto de aplicações que compartilham um determinado estilo arquitetural.

3. Nível de aplicação, que adiciona restrições específicas para uma dada configuração da aplicação.

Para se considerar uma aplicação consistente com o modelo de componentes subjacente, ela deve estar de acordo com as restrições de integridade no nível de modelo. Por exemplo, um componente só pode ser subcomponente de um componente composto ou um enlace só pode ocorrer entre interfaces contidas no mesmo espaço de endereços.

Já numa reconfiguração dinâmica, as ações que compõem a reconfiguração devem respeitar as restrições de transição do mesmo modelo. Para ilustrar um exemplo deste tipo de restrição, seja $A = (E, R)$ um grafo que representa uma dada configuração de um sistema, onde os vértices E representam o conjunto de elementos arquiteturais, que inclui os componentes e as interfaces, e as arestas R representam as relações entre elementos arquiteturais. Uma reconfiguração é a transformação entre configurações $A \xrightarrow{op} A'$, onde uma configuração $A = (E, R)$ é transformada numa outra configuração $A' = (E', R')$ por intermédio de uma operação *op*.

Para ilustrar, considere que a operação *op* seja a inserção de um componente num outro. Esta operação está ilustrada na Tabela 2.3, com o nome de *add*(*parent, child*), onde o componente *child* é inserido no componente composto *parent*, tornando *child* um sub-componente de *parent*.

add(*Component parent, Component child*)
precondições $parent, child \in E$ $\exists iface \in E, hasInterface(parent, iface) \wedge contentCtrlItf(iface)$ $\neg(parent = child)$ $\neg hasChild^{trans}(child, parent)$
poscondições $E' = E$ $R' = R \cup \{hasChild(parent, child)\}$

Tabela 2.3: A operação de adição do componente *child* no componente *parent* para o FRACTAL, descrita em LÉGER (2009).

Respeitadas as precondições, após a execução da operação *add(parent, child)*, o conjunto E' permanece igual a E, pois não há alteração de elementos. Já o conjunto R' é resultante da união entre R e a nova relação de composição, representada pelo predicado $hasChild(parent, child)$. Ainda na Tabela 2.3, o predicado $hasInterface(c, i)$ representa a relação entre um componente e uma interface que ele expõe. O predicado $contentCtrlItf(i)$ significa que a interface i é a interface de controle de conteúdo, que gerencia a adição e remoção de componentes. $hasChild^{trans}(p, c)$ é um predicado que representa a relação de composição entre os componentes p e c e ela é transitiva, por isso o $trans$. Esta precondição significa que um componente não pode conter ele mesmo ao longo de toda a hierarquia de composição.

Em LÉGER (2009), todas as restrições de integridade e transição foram codificadas por meio de expressões lógicas na linguagem do verificador de modelos Alloy[1]. Assim, uma dada configuração do sistema de software ou reconfiguração sua pode ser validada perante este modelo em Alloy. O trabalho não trata da geração da reconfiguração e, num processo onde a reconfiguração é gerada em tempo de execução, a garantia deve ser feita na própria geração. Isso evita adotar uma abordagem de tentativas de reconfiguração que podem não ser executáveis perante o modelo e, assim, desperdiçar recursos neste processo.

2.4.2 Aspectos de Reconfiguração Dinâmica

Quando uma aplicação é reconfigurada, é importante que os dados de execução não sejam perdidos e que, após isso, a aplicação seja deixada num estado consistente.

[1] http://alloy.mit.edu/alloy/

Segundo KRAMER e MAGEE (1990), a aplicação deve ser colocada num estado que permita a ela retornar ao seu modo normal de funcionamento após ser reconfigurada. Assim, além das ações que alteram a estrutura da aplicação, como a ação de conectar dois componentes vista anteriormente, há as ações que alteram o ciclo de vida ou o estado de estruturas da aplicação, com o propósito de colocá-la no estado que permita a ela ser reconfigurada. Em KRAMER e MAGEE (1990), os autores propõem que as ações e os respectivos estados de um componente sejam os ilustrados na Figura 2.8.

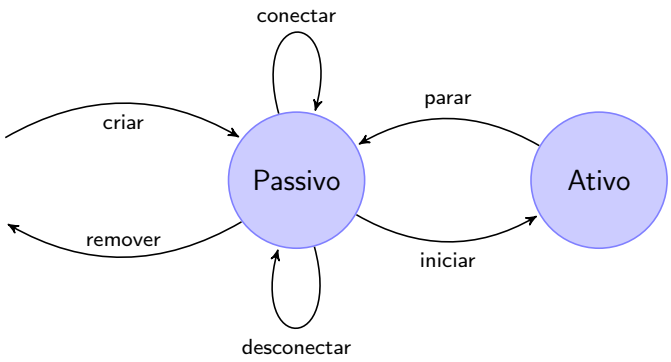


Figura 2.8: As transições entre os estados do ciclo de vida de um componente, segundo KRAMER e MAGEE (1990).

Um componente pode estar num dos dois possíveis estados - *Ativo* ou *Passivo* - e as transições são promovidas pelas ações que podem ser executadas sobre um componente num determinado estado. A partir da possíveis transições entre os estados, os autores descrevem um protocolo de reconfiguração que estabelece uma relação de ordem parcial entre as ações de reconfiguração. O protocolo não considera que haja composição entre componentes e ele possui as seguintes ações que obedecem a ordem parcial abaixo:

1. colocar no estado passivo os componentes necessários.
2. desconectar.
3. remover os componentes.
4. criar os componentes (um componente pode ser criado a qualquer momento antes de conectar).
5. conectar os componentes.
6. colocar no estado ativo os componentes no estado passivo.

Este protocolo foi empregado na reconfiguração de sistemas baseados em componentes e distribuídos. Todavia, com a evolução natural dos modelos de componentes,

em especial daqueles com capacidade reflexiva, este protocolo deixa de ser aplicável em alguns casos. Por exemplo, no modelo de componentes iPOJO/OSGi se um componente que provê uma determinada interface for removido, e esta interface pertencer a um enlace, o novo componente que irá prover esta mesma interface deverá estar criado antes da remoção do primeiro componente, pois o framework é responsável por efetuar os enlaces entre interfaces requeridas e providas. Outro aspecto é sobre a semântica de alguns modelos de componentes que permitem a composição de componentes, ou seja, modelos hierárquicos como o FRACTAL.

Para KRAMER e MAGEE (1990), um componente no estado passivo deve aceitar e servir a transações, porém:

1. não está engajado numa transação que ele tenha iniciado; e
2. não iniciará novas transações.

A partir da definição de estado passivo, os mesmos autores definem uma propriedade denominada *quiescente*. Um componente no estado *quiescente* também está no estado passivo - proposições 1 e 2 abaixo - e:

1. não está engajado numa transação que ele tenha iniciado;
2. não iniciará novas transações;
3. não está engajado em servir a uma transação; e
4. nenhuma transação está ou será iniciada por outros componentes que requerem algum serviço deste componente.

É no estado *quiescente* que um componente pode ser removido de uma configuração de software. Neste estado, o componente não possui resultados parciais de transações e os dados da aplicação que porventura ele possua não serão alterados. A obtenção de um estado quiescente depende do estado passivo do componente e dos componentes que possuem conexões com ele pois, conforme a definição do estado passivo, o componente aceita e serve a transações providas por ele. Então, é preciso parar os componentes que podem enviar requisições e, desse modo, evitar o serviço de uma transação.

Capítulo 3

Trabalhos Relacionados

Neste capítulo serão revisadas e analisadas as abordagens encontradas na literatura que, dada uma nova configuração do software, geram um plano de reconfiguração em tempo de execução.

Resumidamente, as abordagens utilizam um algoritmo para buscar por um plano de reconfiguração que leve o software à configuração objetivo. Em cada uma das abordagens, essa busca se dá sobre um modelo de reconfiguração próprio, bem como a especificação da configuração objetivo guarda as suas particularidades. Cada abordagem será analisada de acordo com as seguintes dimensões:

1. Definição do modelo de componentes.
2. Definição do modelo de reconfiguração e validação deste modelo.
3. Definição do problema de reconfiguração.
4. Geração da reconfiguração.

Resumidamente, as abordagens utilizam um algoritmo para buscar por um plano de reconfiguração que leve o software à configuração objetivo. Em cada uma das abordagens, essa busca se dá sobre um modelo de reconfiguração próprio, bem como a especificação da configuração objetivo guarda as suas particularidades. Cada abordagem será analisada de acordo com as características do modelo de componentes empregado, da representação da capacidade de reconfiguração dinâmica e da reconfiguração a ser gerada por meio deste modelo.

Ao final deste capítulo, na Seção 3.5 é feita uma análise comparativa das abordagens encontradas com o intuito de destacar como outros trabalhos trataram de problemas similares à esta pesquisa.

3.1 Implantação e Reconfiguração Ótima para Sistemas Distribuídos

A primeira proposta identificada na literatura que emprega a abordagem baseada em modelos está descrita em ARSHAD *et al.* (2003) e ARSHAD *et al.* (2007). Nelas, os autores propõem uma abordagem para implantar (*deploy*) e reconfigurar um sistema distribuído por meio de um plano ótimo, onde o melhor plano é aquele que tem o menor tempo de execução. A abordagem não utiliza um modelo de componentes real e não trata da execução do plano sobre o sistema de software, apenas da geração do plano.

Definição do modelo de reconfiguração

O domínio de planejamento é descrito na linguagem *Planning Domain Definition Language* (PDDL) 2.1 (FOX e LONG (2003)) que possui como uma das suas características a introdução do tempo, de maneira que os planos descrevem um comportamento relativo a uma linha de tempo. A busca por um plano ótimo reside naquele cuja execução se dará no menor tempo, de acordo com as funções utilidades descritas no domínio, que descrevem o tempo consumido em cada operação. Os operadores de planejamento alteram os estados de componentes, conectores e máquinas. Um conector ou componente podem assumir os seguintes estados: INATIVO, ATIVO, CONECTADO ou DESLIGADO (*killed*). Já a máquina pode estar: EM BAIXA (*down*), EM CIMA (*up*) ou DESLIGADA (*killed*).

Uma configuração é descrita por meio de predicados que podem ser vistos como uma ADL simplificada. Esta ADL é composta por *componente*, *conector* e *máquina*. Um *componente* contém a lógica do sistema e ele é uma entidade que pode ser gerenciada. Num dado instante de tempo, apenas uma instância de um componente pode existir numa *máquina* e um componente possui apenas o seu nome e não expõem ou requer qualquer interface. Para comunicarem entre si, os componentes precisam estar ligados a um *conector*. Um *conector* provê o enlace de comunicação entre componentes e entre conectores. O conceito de *máquina* representa o local onde componentes e conectores são implantados e uma *máquina* possui restrições que controlam o número de componentes e conectores que podem ser designados para esta máquina.

O modelo de componentes empregado não possui uma implementação e, por isso, não há necessidade de validar este modelo perante a realidade.

Definição do problema de reconfiguração

Nesta abordagem, um elemento de software denominado de *Planit* planeja novas configurações de software em duas ocasiões: a) para a implantação inicial dos com-

ponentes e conectores nas respectivas máquinas; e b) quando ocorre algum problema no sistema e o mesmo deve ser dinamicamente reconfigurado.

Uma determinada configuração arquitetural pode ser especificada de duas maneiras: a) uma configuração implícita; ou b) uma configuração explícita. A configuração implícita especifica predicados sobre o sistema que devem ser verdadeiros na configuração objetivo, sem fornecer uma configuração completa. Deste modo, é possível ter uma especificação parcial da informação, por exemplo, pode ser dito que um componente $Comp_A$ deve estar conectado sem especificar com o que este componente estará conectado. A configuração explícita descreve, na totalidade, os artefatos e suas configurações, empregando predicados para descrevê-la.

No caso da implantação, o estado inicial descreve a lista de artefatos e os respectivos fatos relacionados ao fator tempo, como tempo gasto para iniciar, tempo gasto para conectar e tempo gasto para uma máquina estar em cima (*up*). O estado objetivo neste caso é a operação normal do sistema, com todas as máquinas funcionando e todos os componentes e conectores designados para as respectivas máquinas.

No caso de ocorrer algum problema numa configuração já implantada, um componente denominado de Gerente de Problema é o responsável por recuperar o estado atual, isto é, os componentes, conectores e máquinas que não falharam e determinar o novo estado objetivo, formando um novo problema a ser entregue ao planejador.

Geração da reconfiguração

A geração do plano de implantação e reconfiguração é feita por meio de um planejador que emprega técnicas de planejamento em grafos e faz uso de heurísticas independentes de domínio. Os planos gerados consideram *funções de utilidade* como forma de ordenar os planos e obter um plano ótimo que será escolhido para execução.

3.2 Adaptação e Montagem Arquitetural Autônoma

Em SYKES *et al.* (2007), SYKES *et al.* (2008) e SYKES (2010), Sykes utiliza um modelo de referência em três camadas, ilustrado na Figura 3.1, como um arcabouço para o desenvolvimento de sistemas autônomos. em especial no domínio da robótica. Neste arcabouço, a camada intermediária - cuja pesquisa e desenvolvimento são descritos em SYKES (2010) - é responsável por gerar uma arquitetura de software que irá atender aos objetivos de alto nível determinados na *Camada de gerenciamento de objetivos*.

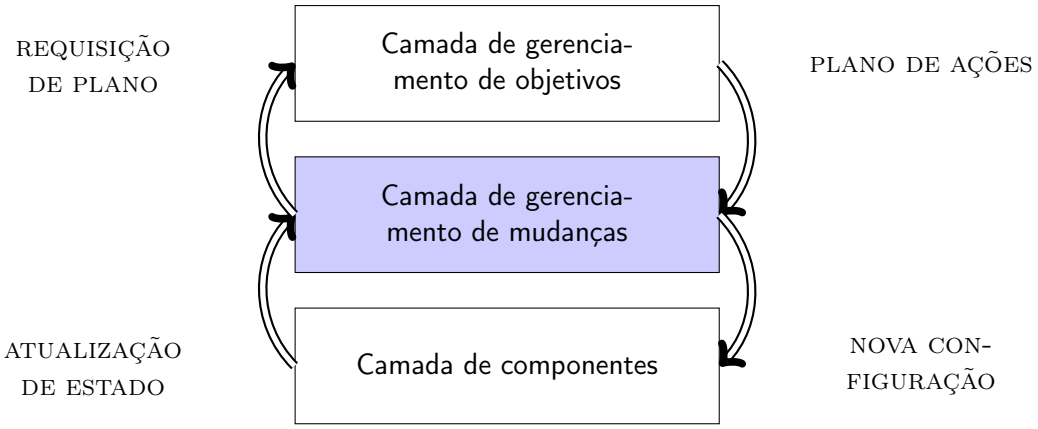


Figura 3.1: Modelo de referência em três camadas

O modelo de componentes

O plano de reconfiguração contém a lista de componentes que devem ser instanciados pelo modelo de componentes empregado, denominado *Backbone*. Neste modelo, um componente é implementado como um objeto em Java, seguindo a linguagem de prescrição do *Backbone*. Este linguagem pode ser vista como uma ADL utilizada para descrever a aplicação a ser instanciada pelo *Backbone*.

Este modelo de componentes foi ampliado para permitir que a instanciação de uma aplicação fosse feita de forma dinâmica. Não há o conceito de componente composto e os meta-objetos de um componente são: interfaces requeridas e providas, e anotações que descrevem características não-funcionais.

Definição do modelo de reconfiguração

Cada componente de software é descrito pelo seu nome e conjunto de *portas*. Cada *porta* pode prover ou requerer uma interface. Estas interfaces indicam a funcionalidade que um componente fornece e resolvem as dependências entre os componentes. Os componentes podem ter anotações que descrevem atributos não-funcionais que possuem um valor associado, formando um par, e a natureza destas anotações significa que componentes funcionalmente equivalentes podem ser ordenados. Por exemplo, se o usuário fornecer uma função utilidade para cada um dos atributos, as configurações candidatas podem ser ordenadas pela utilidade agregada de cada uma.

Definição do problema de reconfiguração

A *Camada de gerenciamento de objetivos* entrega um plano - contendo as ações que devem ser executadas para se atingir um objetivo - para a *Camada de gerenciamento de mudanças*. As ações contidas neste plano representam as funcionalidades que o software a ser (re)configurado deve possuir para atender aos requisitos da aplicação. Deste modo, o plano recebido da *Camada de Gerenciamento de Objetivos*

constitui o estado objetivo a ser obtido pela *Camada de gerenciamento de mudanças*. O processo de montagem de uma configuração de software - ilustrado na Figura 3.2 - começa a partir destas funcionalidades.

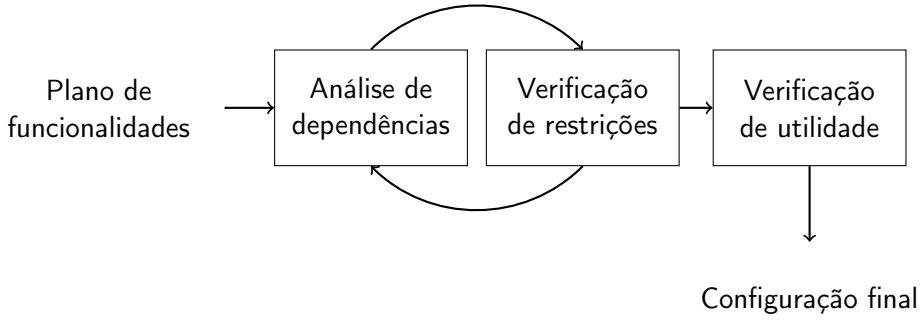


Figura 3.2: Processo de geração agregado proposto em SYKES (2010).

Com o *plano de funcionalidades*, a atividade de Análise de Dependências irá obter as implementações de componentes que provêem estas funcionalidades por meio do seguinte mapeamento:

$$Implementa : Funcionalidades \rightarrow P(Interfaces)$$

Este mapeamento indicará as interfaces necessárias que permitirão obter os respectivos componentes, determinando qual o componente de software que pode ser utilizado para satisfazer um requisito funcional.

Geração da reconfiguração

A partir do estado objetivo são identificados os componentes e respectivas dependências entre os mesmos, descritas por meio de interfaces requeridas. Estas dependências são satisfeitas por um algoritmo que realiza uma busca em profundidade. Sanadas as dependências entre os componentes, as configurações candidatas precisam sofrer uma verificação sobre as restrições de projeto, para determinar se uma configuração é válida perante estas restrições.

A verificação de restrições é feita por um programa na linguagem Prolog, descrito por um desenvolvedor, que verificará as restrições que ele deseja. Se uma determinada configuração não é válida perante uma restrição, o mecanismo tenta alterar a configuração para torná-la válida antes de descartá-la e verificar a próxima candidata. Depois disso, para cada configuração gerada, é calculado o seu valor utilidade, por meio das propriedades dos componentes e respectivas funções utilidade, e a configuração com a maior utilidade é a escolhida. Caso haja mais de uma configuração com a mesma utilidade, a que possuir o menor número de componentes será a escolhida.

O plano de reconfiguração contém a lista de componentes que devem ser instanciados pelo modelo de componentes empregado denominado Backbone. Neste modelo, um componente é implementado como um objeto em Java, seguindo a linguagem de prescrição do Backbone. Este linguagem pode ser vista como uma ADL utilizada para descrever a aplicação a ser instanciada pelo Backbone.

3.3 PLASMA: uma arquitetura em camadas baseada em planejamento para a adaptação de software

Em TAJALLI *et al.* (2010), Tajalli et al. utilizam uma arquitetura em três camadas, para a adaptação arquitetural de um software que controla um robô. Destas camadas, a de interesse para esta pesquisa é a camada de Planejamento, ilustrada na Figura 3.3, que contém: a) um *Planejador de Aplicação*, que é responsável por decidir a configuração arquitetural objetivo; e b) um *Planejador de Adaptação*, que gera o plano de reconfiguração da aplicação para se atingir a configuração arquitetural objetivo decidida pelo *Planejador de Aplicação*.

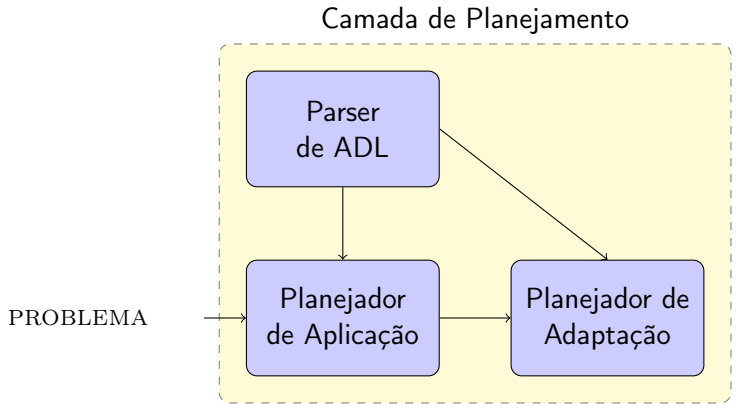


Figura 3.3: Camada de Planejamento - adaptado de TAJALLI *et al.* (2010)

No *Planejador da Aplicação*, o desenvolvedor fornece um *Problema* descrito na linguagem NPDDL, descrita em BERTOLI *et al.* (2003). O problema contém o estado inicial e o estado objetivo, o qual está descrito por meio das variáveis de estado e representa o que o robô deve executar, como carregar algum objeto, navegar ou seguir um outro robô. A saída desta atividade é um **plano de ações** que, se executadas pelo robô, atingem o estado objetivo. Em seguida, o mesmo *Planejador da Aplicação* determina os componentes de software necessários e respectiva topologia, a partir das ações contidas no **plano de ações** e as dependências entre estes componentes, fazendo o casamento das interfaces requeridas e providas. O casamento

entre ações do robô e os componentes de software ocorre por meio das ações contidas no **plano de ações**, pois as ações que o robô deverá executar são as interfaces dos componentes de software a serem invocadas e que foram obtidas da descrição numa ADL específica, cujo nome é SADLE, e que será descrita mais adiante. Estes dois *Planejadores* possuem domínios diferentes e esta pesquisa está interessada no *Planejador de Adaptação*, que é o responsável por gerar o plano de reconfiguração arquitetural.

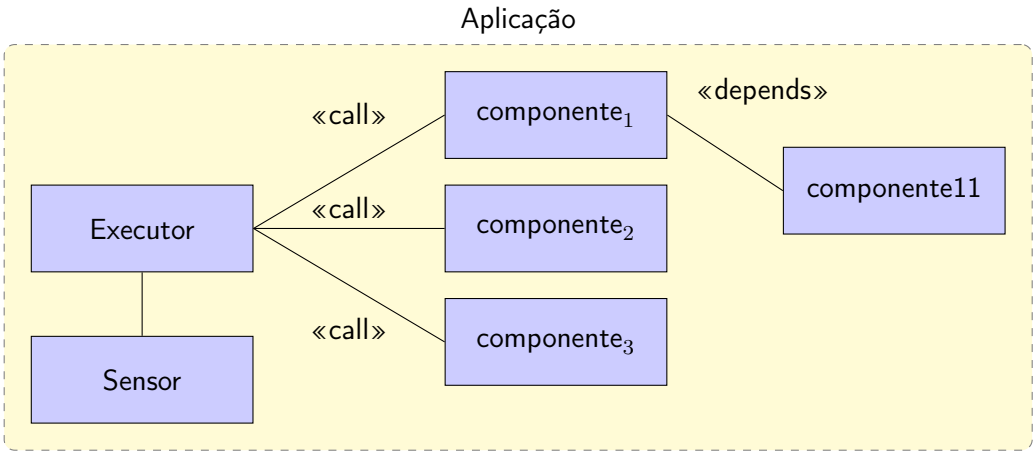


Figura 3.4: Arquitetura da aplicação a ser executada no robô - adaptado de TAJALLI *et al.* (2010)

O estilo arquitetural empregado pelo software a ser executado no robô, ilustrado na Figura 3.4, é o "call-return", onde um componente, denominado de *Executor*, é o responsável por chamar as interfaces dos componentes descritas no plano de ação, funcionando como um orquestrador. O momento em que estas chamadas ocorrem depende da ocorrência de eventos, que são obtidos pelo *Sensor*, e irão determinar em que estado o robô se encontra. Após determinar o estado, a ação referente a este estado e que está descrita no plano será executada.

O modelo de componentes

O modelo de componentes empregado nesta abordagem denomina-se Prism-MW e os seus constituintes são: Arquitetura, Componente e Conector. Uma *Arquitetura* serve como um contêiner de execução para uma configuração de software ou topologia. Os componentes implementam serviços e os conectores as interações entre estes serviços por meio da troca de eventos entre as portas. O Prism-MW é um middleware com capacidade de acessar e alterar a arquitetura. Do artigo entende-se que as alterações permitidas se referem apenas a configuração arquitetural, ou seja, os enlaces podem ser alterados. Não há o conceito de componente composto e os meta-objetos de um componente derivados da ADL utilizada que são: atributos, interfaces requeridas e providas, e pré e pós-condições destas mesmas interfaces,

são empregados para a geração do plano de reconfiguração e não são alterados pelo mecanismo de reconfiguração.

A artigo não descreve o gerenciamento do ciclo de vida do software, nem de seus componentes, mas a partir do exemplo citado, conclui-se que o sistema todo é parado, reconfigurado e depois reiniciado.

Definição do modelo de reconfiguração

Para gerar a configuração objetivo, o *Planejador de Aplicação* obtém o domínio a partir da descrição arquitetural da aplicação por meio de uma ADL, denominada de SADLE. A SADLE possui três partes principais: *componentes*, *conectores* e *topologia*. A *Topologia* é uma configuração arquitetural, que define as instâncias de componentes e conectores e respectiva interconexão. Um *Componente* define variáveis de estado, interfaces providas e requeridas, e cada interface especifica pré e pós-condições de sua invocação por meio de expressões em lógica de primeira ordem, envolvendo as variáveis de estado do *Componente*. Obtida a configuração objetivo, ela é entregue como um problema para o *Planejador de Adaptação*, cujo domínio foi fornecido pelo arquiteto do sistema. As ações neste domínio se referem ao ciclo de vida dos componentes de software: instanciar, encerrar (*kill*), adicionar, remover, conectar e desconectar, e a configuração de software gerada obedece ao estilo arquitetural descrito anteriormente.

Definição do problema de reconfiguração

O estado objetivo é entregue ao *Planejador de Adaptação* como um conjunto de componentes de software, contendo os respectivos nomes dos componentes e um plano de reconfiguração é gerado, obedecendo o estilo arquitetural anteriormente descrito.

Geração da reconfiguração

Um planejador da área de IA, que emprega a técnica de Planejamento Baseado em Verificação de Modelos (*Plan Based on Model Checking*) é empregado tanto para o Planejamento da Aplicação, quanto para o Planejamento da Adaptação. O tipo de plano que este planejador fornece é uma política, ou seja, um série de pares **(estado, ação)**. Este plano será o "programa" a ser executado pelo componente de software *Executor*, pois executando-o o robô irá atingir o estado objetivo.

Após a geração da configuração inicial de software, um replanejamento pode ocorrer em duas situações. A primeira é decorrente da mudança do modelo descrito em SADLE, que pode ocorrer numa evolução dos requisitos do sistema. Esta mudança gera um novo domínio que dispara o replanejamento no *Planejador de Aplicação* e um novo problema de adaptação é gerado e entregue ao *Planejador*

de Adaptação. Neste caso, o estado inicial é a configuração atual e, consequentemente, um novo plano é criado para reconfigurar a atual arquitetura. A segunda causa de um replanejamento é o erro de algum componente da aplicação, que irá comprometer a execução do plano pelo *Executor*. Neste caso, o erro de um componente representa a sua remoção do domínio da aplicação, o que irá disparar o replanejamento no *Planejador de Aplicação*, desconsiderando o componente que falhou. Caso não haja um plano de ações que atinja o objetivo, a abordagem retorna uma falha ao arquiteto do sistema.

3.4 Um protocolo de reconfiguração para sistemas baseados em componentes

Em BOYER *et al.* (2013), os autores propõem um protocolo de reconfiguração para um sistema baseado em componentes, mas cujo modelo de componentes é apenas uma especificação e não possui uma implementação.

Definição do modelo de reconfiguração

Os invariantes arquiteturais do modelo de componentes estabelecem uma relação entre o ciclo de vida do componente e a semântica das interfaces requeridas por ele, que podem ser interfaces mandatórias ou opcionais. O trabalho considera que o modelo de componentes possui os seguintes invariantes arquiteturais:

I_1- Todos os componentes no estado *started* possuem enlaces em todas as suas interfaces mandatórias.

I_2- Todos os componentes no estado *started* somente possuem enlaces com componentes no estado *started*.

I_3- Não há enlaces entre componentes falhos ou destruídos.

I_4- Não há dependência circular entre interfaces mandatórias de componentes.

Já a gramática de reconfiguração especifica em que estado do ciclo de vida de um componente uma operação de reconfiguração pode ser executada. Nesta gramática, ilustrada na Figura 3.5, o ciclo de vida de um componente possui três estados: STOPPED, STARTED e FAILED.

Quando um componente é construído ele está no estado STOPPED, significando que ele ainda não está funcional. Neste mesmo estado, um componente pode fazer e desfazer os enlaces entre quaisquer interfaces (mandatórias e opcionais). Já no estado STARTED, apenas os enlaces opcionais podem ser feitos

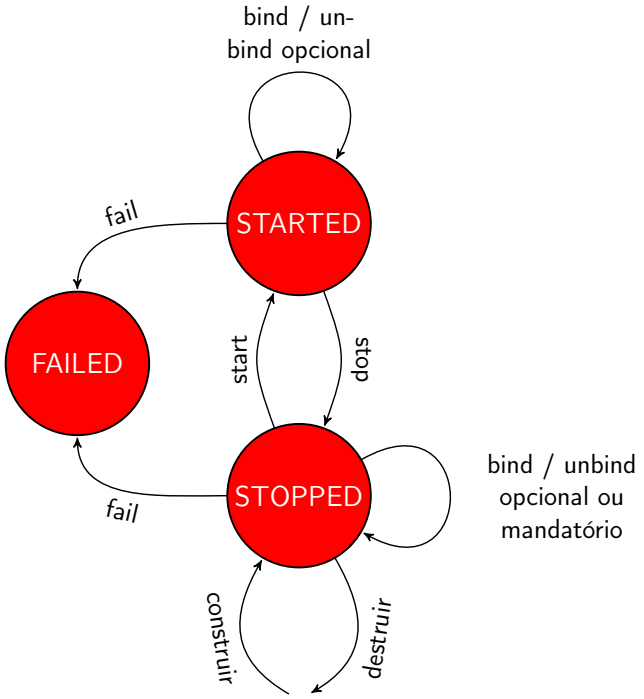


Figura 3.5: Um grafo ilustrando a gramática de reconfiguração.

ou desfeitos. Quando uma operação de reconfiguração falha, o protocolo de reconfiguração suspenderá a reconfiguração em andamento e iniciará o processo de recuperação, marcando o componente que apresentou a falha no estado FAILED.

Definição do problema de reconfiguração

Dada uma configuração objetivo e a configuração atual, um protocolo irá determinar o que precisa ser removido e o que precisa ser criado. O protocolo foi formalizado no assistente de provas Coq[1], que é um provador de teorema interativo.

Geração da reconfiguração

O trabalho considera que os componentes envolvidos numa reconfiguração já foram colocados no estado quiescente, no nível dos componentes, como proposto em KRAMER e MAGEE (1990), mas não descreve como a lista de componentes que devem estar neste estado foi obtida. A parada de componentes descritas a seguir, são decorrentes do protocolo proposto. O protocolo de reconfiguração contém duas grandes fases, a primeira denominada de fase para baixo, e a segunda denominada de fase para cima. O protocolo deve respeitar a *gramática de reconfiguração* e o conjunto de *invariantes arquiteturais* descritos anteriormente. A fase para baixo é composta de três procedimentos:

[1]http://coq.inria.fr/

1. o primeiro determina um conjunto de metas que representam o que deve ser removido por meio de uma diferença entre a arquitetura inicial e a objetivo;

2. o segundo emprega um conjunto de regras de propagação para inserir as ações referentes ao controle do ciclo de vida que tornam aplicáveis as metas estabelecidas no primeiro passo, onde estas metas representam a remoção de enlaces ou a destruição de componentes; e

3. o terceiro efetua a ordenação das ações obtidas no segundo passo, de maneira a obedecer a *gramática de reconfiguração*.

No segundo procedimento, as metas são transformadas em ações de reconfiguração por intermédio de regras de propagação. Estas regras servem para preservar os invariantes arquiteturais anteriormente descritos. Um exemplo de regra usada na abordagem é aquela que traduz uma meta que desfaz um enlace numa outra meta seguida de uma ação. Por exemplo, se há a meta *unwired(w)*, onde w é um enlace entre componentes e o enlace w é de uma interface obrigatória, então o componente que requer a interface referente a este enlace deve ser parado antes do enlace ser removido.

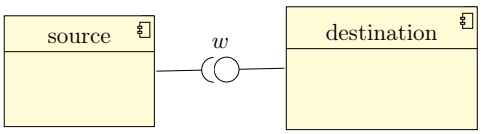


Figura 3.6: Um enlace entre componentes em BOYER *et al.* (2013)

A descrição original desta regra está na tabela abaixo, onde *w.src* significa origem do enlace w - o componente *source* na Figura 3.6.

Regra que preserva o invariante I_1
unwired(w) $\Rightarrow$ *if mandatory(w) stopped(w.src) unwire(w)*

Esta regra força o invariante I_1 e significa que a meta *unwired(w)* pode ser substituída da seguinte forma:

- se o enlace w for obrigatório, acrescentar a meta *stopped(w.src)*, significando que o componente que requer a interface deverá ser parado; e

- acrescentar a ação *unwire(w)*, significando a remoção do enlace w.

Para se preservar o invariante I_2, uma outra regra é aplicada nas metas *stopped(c)*. A descrição desta segunda regra está na tabela abaixo, onde *w.src* significa a origem do enlace w - o componente *source* na Figura 3.6 - e *w.dst* significa o destino do enlace w - o componente *destination* na Figura 3.6.

Regra que preserva o invariante I_2
stopped(c) $\Rightarrow \forall w \mid w.dst = c$ *if mandatory(w) then stopped(w.src) else unwired(w) stop(c)*

Esta segunda regra irá parar os componentes que dependem do componente que deve ser parado e ela efetua a seguinte substituição:

- se o enlace w for obrigatório, acrescentar a meta *stopped(w.src)*, significando que o componente que requer a interface deverá ser parado; e
- acrescentar as ações *unwire(w)* e *stop(c)*.

Definido o conjunto de ações de reconfiguração, o protocolo irá ordená-las obedecendo a ordem ilustrada na Figura 3.7, a fim de atender à gramática de reconfiguração ilustrada na Figura 3.5.

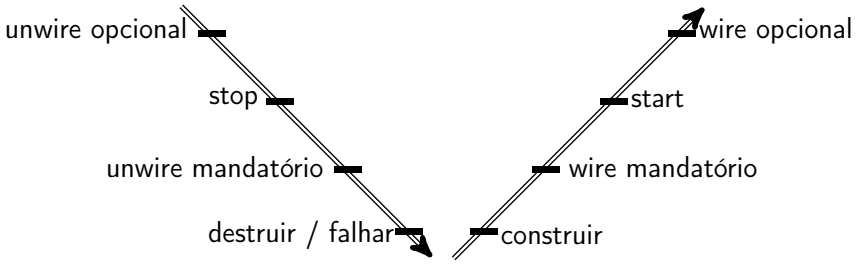


Figura 3.7: A ordem entre as ações de reconfiguração nas fases "para baixo" e "para cima".

3.5 Considerações sobre as abordagens revistas

O primeiro trabalho, analisado na Seção 3.1, tem como problema implantar uma nova configuração de software no menor tempo. Apesar de não tratar da execução do plano implantação gerado, ele mostra a viabilidade de se empregar uma abordagem baseada num modelo para se gerar as reconfigurações quando se fizerem necessárias.

O segundo e o terceiro trabalhos, analisados nas Seções 3.2 e 3.3 respectivamente, empregam um modelo de componentes real e tratam da execução do plano gerado sobre o software, reconfigurando-o efetivamente. Entretanto, os modelos de componentes empregados em cada um dos trabalhos simplificam a execução da reconfiguração, fazendo com que o plano de reconfiguração seja mais próximo de uma lista de componentes que deverá ser instanciada pelo modelo de componentes, sem considerar explicitamente os enlaces entre os mesmos. Esta simplificação se reflete no modelo de reconfiguração utilizado para se gerar o plano e, consequentemente, reduz a complexidade da sua geração. Como mencionado no segundo trabalho, a

geração do plano é feita por uma busca em profundidade. Ambos os trabalhos não consideram a corretude do domínio e apenas no segundo trabalho o plano gerado sofre algum tipo de validação perante o modelo de componentes.

O trabalho mais próximo desta pesquisa é o analisado na Seção 3.4. Ele emprega um procedimento correto para a geração do plano e considera a capacidade reflexiva de um modelo de componentes teórico. A corretude do plano é em relação ao modelo codificado no provador de teoremas empregado entretanto, a inexistência de uma implementação real do modelo de componentes utilizado, não permite avaliar se o modelo de reconfiguração é correto perante a realidade que ele deveria representar. Um outro ponto é que questões ligadas ao estado da aplicação para permitir uma reconfiguração também não são tratadas pela abordagem, pois ele considera que este estado já foi obtido antes de se gerar a reconfiguração. Quanto a iniciação da aplicação não há uma ordem a ser respeitada.

Os invariantes I_1 e I_2 podem acarretar na parada de um grande número de componentes. Por meio do grafo na Figura 3.8, onde cada vértice representa um componente e cada aresta um enlace entre componentes, se o enlace w_d tiver que ser removido, os enlaces w_b, w_c, w_e e w_f forem obrigatórios e o enlace w_a for opcional, então os componentes C_2, C_3, C_4, C_6 e C_7 deverão ser parados.

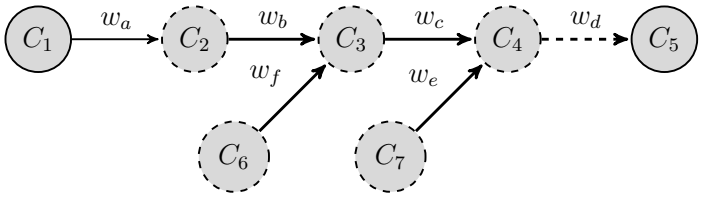


Figura 3.8: Um grafo ilustrando componentes e respectivos enlaces.

Esta parada *em cascata* pode comprometer uma reconfiguração dinâmica a ponto de ser melhor parar toda a aplicação, efetuar as operações sem considerar o ciclo de vida, e depois reiniciá-la. Além disso, esse invariante pode não ser necessário em algumas implementações de modelos de componentes como o FRACTAL, pois um componente pode estar iniciado - estado *started* - e ainda sim ter um enlace obrigatório a um componente parado - estado *stopped*. Neste caso, para se remover o componente C_5, apenas C_4 precisaria estar parado.

3.6 Resumo das abordagens

A partir das dimensões analisadas, os trabalhos estão resumidos na Tabela 3.1 por meio das características abaixo enumeradas. O item **a** informa se o modelo de componentes utilizado é de propósito geral ou especializado. A importância dos itens **b** e **c** refletem a aplicabilidade de uma solução num problema real, ou seja, num

sistema baseado em componentes e com capacidade reflexiva. Quanto ao item **d**, ele irá garantir que os planos gerados são corretos perante o modelo de componentes empregado, aumentando a confiabilidade de uma reconfiguração.

a. Modelo de componentes.

b. Modelo de Reconfiguração - Representação da capacidade reflexiva do modelo de componentes.

c. Modelo de Reconfiguração - Representação de aspectos de reconfiguração dinâmica.

d. Verificação do modelo de reconfiguração.

Nas abordagens que empregam modelos de componentes com uma implementação (2 e 3), estes modelos são especializados e extremamente simples, pois a capacidade de reconfiguração se limita a instanciar uma nova aplicação e não a executar ações que transformam uma configuração numa outra. Nenhuma das abordagens trata de aspectos de reconfiguração dinâmica, onde as partes da aplicação afetadas pela mudança precisam ser paradas e depois reiniciadas. Por fim, a verificação do modelo de reconfiguração perante a realidade não é feita em nenhuma abordagem. A corretude do modelo apenas é considerada na abordagem (4), mas falta a ela implementar o respectivo modelo de componentes, pois ele é apenas uma especificação.

Abordagem	**a**	**b**	**c**	**d**
(1) Arshad et al.	Geral - especificação	Sim	Não	Não
(2) Sykes et al.	Especializado - implementação	Sim (simplificado)	Não	Não
(3) Tajalli et al.	Especializado - implementação	Sim (simplificado)	Não	Não
(4) Boyer et al.	Geral - especificação	Sim	Não	Sim (é o próprio modelo)

Tabela 3.1: Resumo dos trabalhos revisados.

Capítulo 4

Abordagem Proposta para a Geração de Planos Dinâmicos

A solução para a geração de planos de reconfiguração em tempo de execução é baseada na visão de agente inteligente, da área de Inteligência Artificial, e a respectiva implementação como um agente baseado em objetivos. Esta implementação foi materializada pela técnica de planejamento automatizado empregando-se um planejador de redes hierárquicas de tarefas, o JSHOP2[1]. A solução pode ser desmembrada nas seguintes etapas abaixo ilustradas.

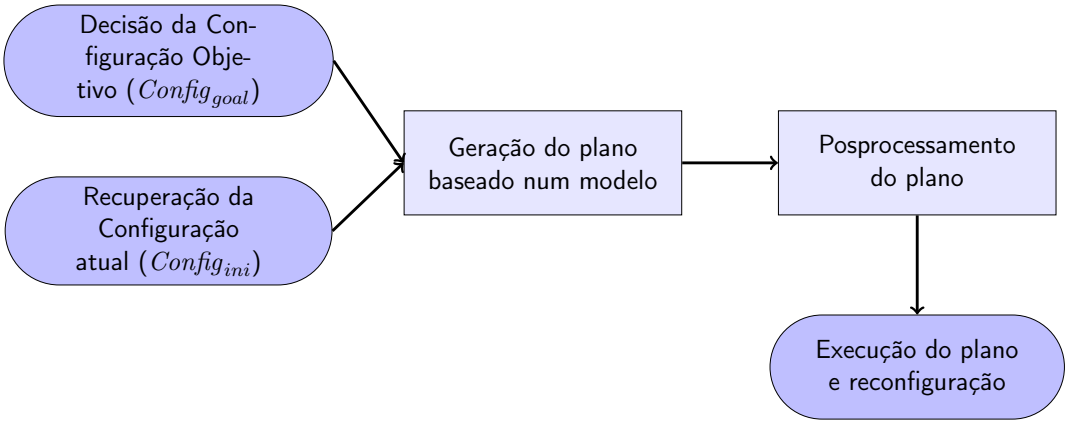


Figura 4.1: Sequência de atividades na geração de um plano de reconfiguração.

A decisão sobre qual será a configuração objetivo não é tratada nesta pesquisa e ela pode ser feita de diferentes maneiras. Por exemplo, manualmente, quando um desenvolvedor decide qual a nova configuração, ou de maneira automática, como em TAJALLI *et al.* (2010), permitindo integrar a solução desta pesquisa com outras abordagens que tratem deste problema. Neste caso, o ponto a ser observado nesta integração é quanto ao modo de se especificar esta configuração, que deve obedecer aos predicados que compõem a linguagem que descreve um problema de

[1]http://sourceforge.net/projects/shop/files/JSHOP2/

planejamento.

A recuperação da configuração inicial, que irá formar o estado inicial, pode ser feita automaticamente por meio da capacidade de introspecção provida pelo modelo de componentes. Após a geração do plano, pode ser necessário efetuar algum posprocessamento para permitir a sua execução sobre a aplicação a ser reconfigurada. Este posprocessamento irá remover ações do plano que não devem ser traduzidas. Por fim, as ações de reconfiguração são traduzidas para a API de reflexão do FRACTAL.

Este capítulo trata das duas atividades identificadas por retângulos na Figura 4.1. A Seção 4.1 explica como o domínio de planejamento foi desenvolvido para o modelo de componentes FRACTAL, destacando-se as restrições do modelo e a sua tradução na linguagem do domínio de planejamento. A Seção 4.2 trata da validação deste domínio, a fim de garantir a corretude dos planos gerados perante o modelo de componentes e a sua execução correta. Por fim, a Seção 4.3 trata da limpeza do plano gerado e da sua tradução para a linguagem FSCRIPT.

4.1 O Domínio de Planejamento para o FRACTAL

Para a descrição do domínio de planejamento, serão utilizados os conceitos estruturais do modelo de componentes FRACTAL, as ações de reconfiguração que alteram estes conceitos e as características de JULIA, que é a implementação do FRACTAL na linguagem Java.

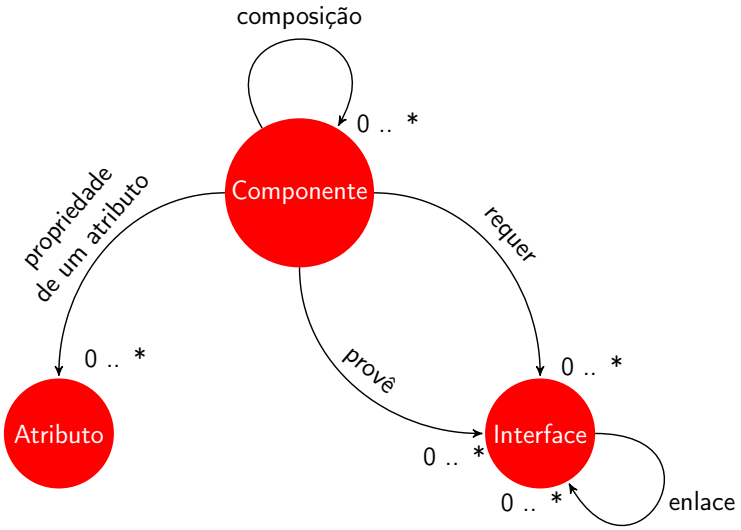


Figura 4.2: Elementos arquiteturais do modelo FRACTAL.

Os conceitos estruturais estão ilustrados na Figura 4.2 por meio de um grafo, onde os vértices são os elementos arquiteturais definidos pelo modelo de componentes e as arestas os relacionamentos entre estes elementos. Os elementos arquiteturais são:

`Componente`, `Atributo` e `Interface` e as relações são: `composição`, `propriedade` de um atributo, `requer` uma interface, `provê` uma interface e `enlace` entre interfaces.

A relação de composição entre componentes define um supercomponente e um subcomponente. Um supercomponente deve ser um componente composto que pode conter outros componentes, sejam eles componentes compostos ou simples. Um subcomponente está contido e um ou mais de um supercomponente. No caso do componente estar contido em mais de um supercomponente, ele é denominado de componente compartilhado. A restrição da relação de composição é que um componente não pode estar contido nele mesmo, seja diretamente, por uma autorelação de composição, ou transitivamente na hierarquia de composição.

Um componente pode requerer e prover interfaces e possuir atributos. Quanto a cardinalidade destas relações, um componente pode prover nenhuma ou várias interfaces, bem como requerer nenhuma ou várias interfaces. Uma interface requerida pode ser opcional ou mandatória e, neste último caso, deve haver um enlace nesta interface para poder se iniciar o componente.

Um atributo é uma propriedade que pode ser alterada e, em geral, são tipos primitivos usados para representar o estado de um componente. Um componente pode ter nenhum ou vários atributos.

As relações `propriedade` de um atributo, `requer` uma interface e `provê` uma interface são imutáveis, ou seja, declarado que um componente requer e provê interfaces e possui atributos, isto não poderá ser modificado, bem como os tipos dos atributos e interfaces.

Um enlace entre componentes é feito entre uma interface requerida por um componente e uma interface provida por um outro componente. Há interfaces requeridas que são coleções e, neste caso, elas podem ter enlaces com mais de uma interface providas por outros componentes.

Na Figura 4.3, há um diagrama de componentes na notação UML que ilustra os enlaces entre interfaces que podem ocorrer no FRACTAL. Neste diagrama, as interfaces requeridas pertencem ao componente composto `SocketServer`, ou seja, os enlaces são feitos a partir deste componente. Os enlaces identificados por $bind_1$ e $bind_2$ são de interfaces externas, enquanto $bind_3$ é de uma interface interna sua, que só pode existir se o componente for composto.

O enlace identificado por $bind_1$ é um *enlace normal* entre interfaces de dois componentes contidos no mesmo supercomponente `CommSystem`, ou seja, os componentes são irmãos. Os enlaces $bind_2$ e $bind_3$ são enlaces entre uma interface interna e um subcomponente. O enlace $bind_2$ representa um *enlace de importação* da interface requerida pelo `SocketServer`. Ela se dá por meio da interface interna provida pelo componente pai a qual está associada a uma interface externa requerida por ele. O enlace $bind_3$ representa um *enlace de exportação* onde a interface interna $iFaced_b$

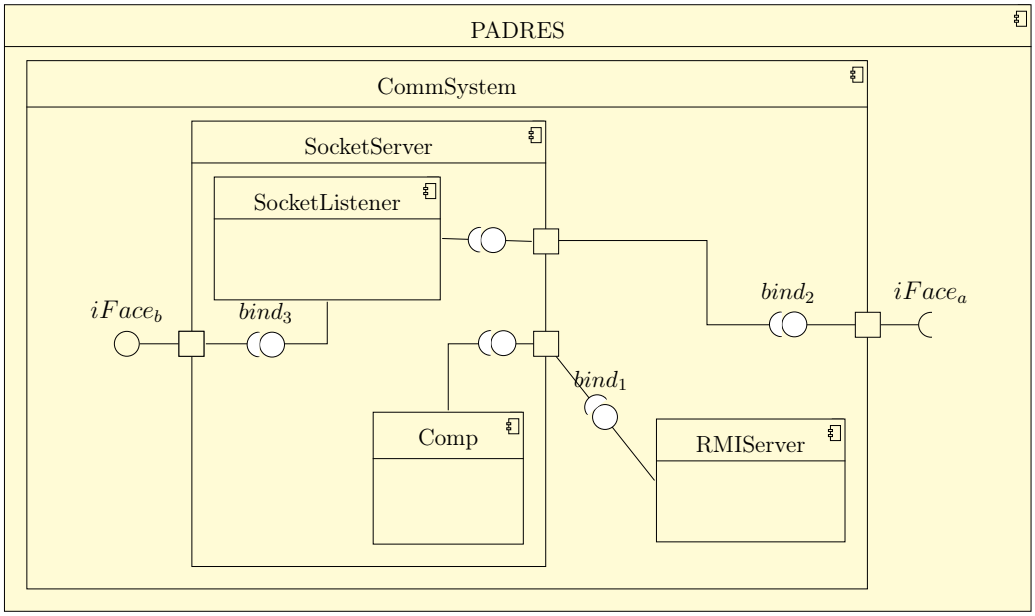


Figura 4.3: Tipos de enlaces de interfaces requeridas pelo componente `SocketServer`.

requerida por `SocketServer` é associada à interface externa de um subcomponente. Esta interface requerida terá um enlace com uma interface provida por um subcomponente, neste caso o subcomponente `SocketListener`. No FRACTAL, as interfaces internas são criadas automaticamente pela implementação, bem como a conexão entre uma interface externa e a respectiva interna num componente composto.

Os três tipos de enlaces vistos anteriormente são denominados de *enlaces locais*, que são enlaces entre irmãos, enlaces de exportação e enlaces de importação. No FRACTAL, um enlace não pode atravessar o supercomponente. Assim não é possível haver um enlace direto entre o componente `RMIServer` e os subcomponentes de `SocketServer`.

Além dos elementos arquiteturais e suas relações, um `Componente` possui um ciclo de vida, ilustrado na Figura 4.4. Os vértices desta figura representam os estados do ciclo de vida e as arestas descrevem as ações de reconfiguração do FRACTAL que podem ocorrer em cada estado. O controle do ciclo de vida irá atender a duas necessidades: a primeira é decorrente do modelo de componentes e a segunda do estado seguro para se efetuar uma reconfiguração. Quanto ao modelo de componentes, na Figura 4.4, observa-se que a ação para remover um enlace (unbindFc) só pode ser efetuada quando o componente que requer a interface no enlace está parado. O mesmo acontece ao se remover um subcomponente (removeFcSub) e, neste caso, o supercomponente também deverá estar parado. Isto irá compor as precondições dos operadores no domínio de planejamento.

Quanto ao ciclo de vida, ele está relacionado às atividades que um componente pode executar. Estas atividades estão descritas abaixo e ilustradas na Figura 4.5:

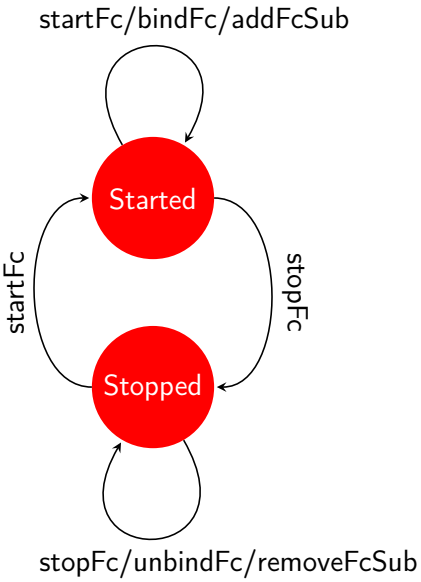


Figura 4.4: Ciclo de vida de um componente FRACTAL.

(a) um componente aceita a requisição ① numa interface provida e responde em ② a esta requisição sem empregar outros componentes externos.

(b) um componente aceita a requisição ① numa interface provida e responde em ④ a esta requisição, depois de efetuar uma outra requisição ② por meio de uma interface requerida e receber a respectiva resposta em ③.

(c) o componente inicia a requisição ① e recebe a resposta em ②.

(d) o componente executa uma atividade sem necessitar enviar alguma requisição externa.

Estas atividades permitem compreender como obter o estado seguro na implementação JULIA do modelo de componentes FRACTAL. Na implementação JULIA, a atividade dos componentes corresponde ao fluxo de execução (*thread*) do Java e a comunicação entre componentes são invocações síncronas de métodos Java. Nesta pesquisa, considera-se que os componentes são locais, ou seja, são executados numa mesma máquina virtual.

Na definição de KRAMER e MAGEE (1990), para se colocar a aplicação num estado que permita a ela ser reconfigurada, e depois retornar ao seu modo normal de funcionamento, um componente deve assumir um estado denominado de *quiescente*. Ele é obtido indiretamente, alterando-se o ciclo de vida do componente que possui dois estados apenas: *ativo* e *passivo*. No estado *ativo* o componente está funcionando normalmente. No estado *passivo* ele não deve aceitar novas requisições nem iniciar requisições externas. Porém, com o intuito de evitar um impasse (*deadlock*)

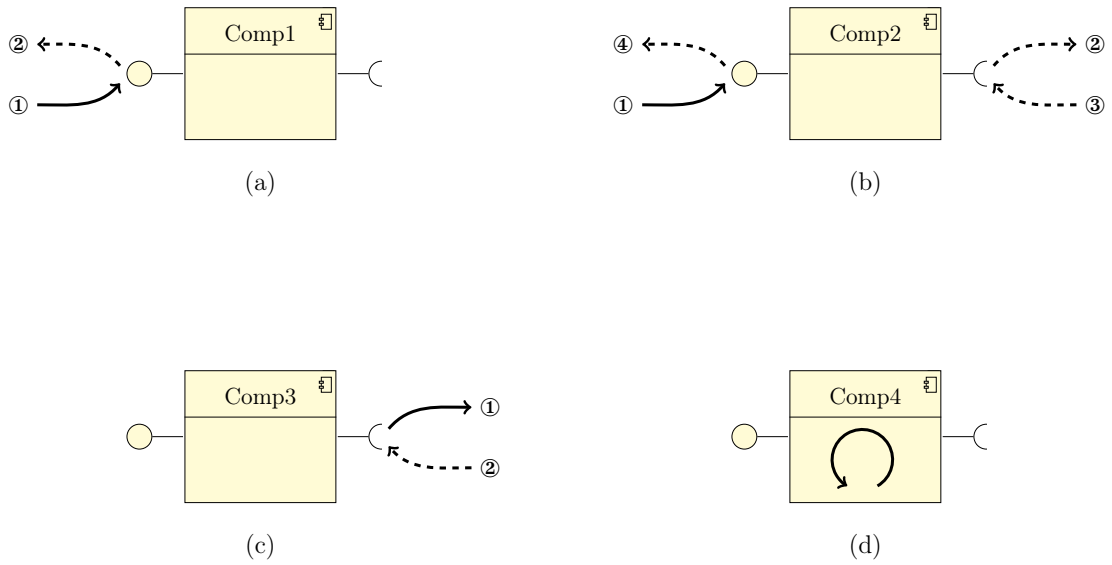


Figura 4.5: Tipos de atividades que podem ocorrer num componente.

e permitir que os componentes que dependem dele também possam atingir este estado, o componente no estado *passivo* deve ser capaz de completar as execuções em andamento a partir das suas interfaces providas. Assim, um componente *passivo* deixa de executar as atividades (c) e (d) da Figura 4.5, mas continua a executar (a) e (b). Deste modo, um componente é *quiescente* quando ele e os componentes que dependem dele estão no estado *passivo*.

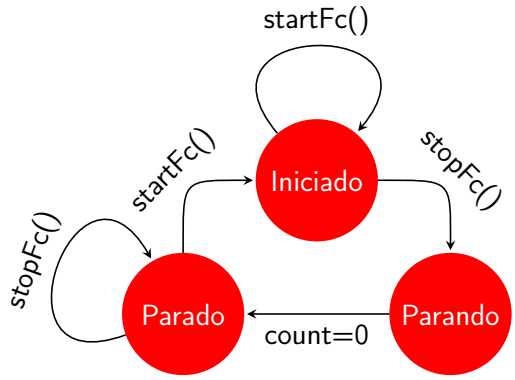


Figura 4.6: Ciclo de vida de um componente em JULIA.

Em JULIA, protocolo de alteração do estado de um componente pode ser descrito por meio do diagrama de transição de estados ilustrado na Figura 4.6. O estado *Iniciado* é o estado normal de funcionamento onde qualquer umas das quatro atividades ilustradas na Figura 4.5 pode ocorrer. O estado *Parando* (Stopping) é obtido após a chamada do método stopFc() do controlador do ciclo de vida do componente. Neste estado intermediário, o componente já não aceita mais requisições nas suas interfaces providas, mas continua a execução das requisições em andamento, iniciadas antes da chamada a stopFc() e que ainda não terminaram.

Para saber que requisições estão em andamento, o protocolo emprega o método de contagem de *threads*. Neste método, a cada requisição feita a uma interface provida um contador é incrementado e, no retorno desta requisição, o contador é decrementado. Quando as requisições em andamento terminarem, levando o contador ao valor 0, então o componente passará ao estado *Parado* (Stopped). Este protocolo resolve as atividades (a) e (b) ilustradas na Figura 4.5 e, para que as atividades (c) e (d) também sejam paradas quando o componente atingir o estado *Parado*, o programador deverá cuidar para que este comportamento ocorra. Desta forma, em JULIA um componente é *quiescente* quando ele está no estado *Parado*.

O estado parado de um componente é precondição para se executar duas ações de reconfiguração no FRACTAL:

1. remoção de enlaces entre interfaces; e

2. remoção de um subcomponente de seu supercomponente.

Algumas situações exigem que haja uma ordem entre os componentes que devem ser parados numa reconfiguração, para evitar que um componente nunca atinja o estado necessário e torne a reconfiguração inexequível. Por exemplo, na Figura 4.7, há quatro componentes conectados e uma requisição de `Comp1` para `Comp2` (① → ②) gera o fluxo (① → ② → ③ → ④). Em JULIA, esse fluxo corresponde a uma mesma *thread*, iniciada em `Comp1`, e a cada requisição de uma interface, o contador de *threads* de cada componente é incrementado.

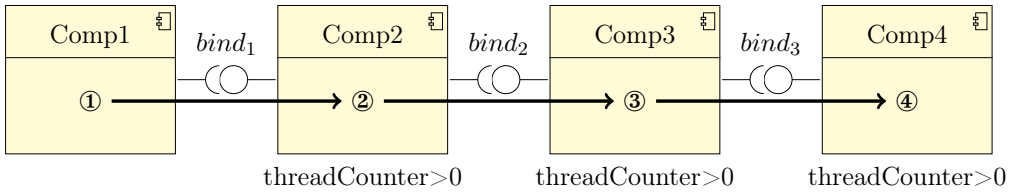


Figura 4.7: O fluxo de execução num componente.

Se o componente `Comp3` tiver que ser removido, os enlaces $\texttt{bind}_2$ e $\texttt{bind}_3$ também terão que ser removidos. Para a remoção do enlace $\texttt{bind}_2$ o componente `Comp2` deve estar parado e, para se remover o enlace $\texttt{bind}_3$, o componente `Comp3` deve estar parado. Se `Comp3` for parado antes de `Comp2`, uma nova requisição de `Comp1` que gere o mesmo fluxo anterior (① → ② → ③ → ④) será interceptada por `Comp3` e ficará ativa em `Comp2`, incrementando o seu contador de *threads*. Ao se ordenar parar `Comp2` para a remoção do enlace $\texttt{bind}_2$, ele entrará no estado *parando* e aguardará até que o contador atinja o valor nulo. Porém, isto não ocorre pois a requisição iniciada em ① nunca termina e `Comp2` não atinge o estado *parado*, evitando que a precondição para se remover o enlace $\texttt{bind}_2$ seja verdadeira.

Assim, para se remover um componente, primeiro são removidos os enlaces que dependem dele, feitos com as interfaces providas pelo componente, para depois remover os enlaces requeridos pelo componente.

Entretanto, no caso de dependência circular, ou mútua, apenas a relação de dependência não permite distinguir que componente deve ser parado primeiro para se remover um enlace. Sejam `Comp1` e `Comp2` dois componentes conectados como na Figura 4.8.

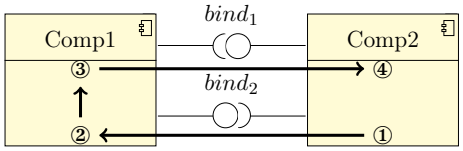


Figura 4.8: Dependência circular.

Uma requisição no enlace $\texttt{bind}_2$, iniciada por `Comp2` (① → ②), incrementa o contador em `Comp1`. Esta mesma requisição continua e gera a requisição no enlace $\texttt{bind}_1$ (③ → ④), incrementando o contador em `Comp2`.

Se `Comp2` for parado antes da requisição no enlace $\texttt{bind}_1$ (③ → ④) ocorrer, ele passará ao estado *parado* pois o seu contador de *thread* está nulo neste momento, evitando que a *thread* iniciada em ① termine. Por causa da *thread* iniciada em ①, `Comp1` está com o seu contador maior que zero. Assim, se for requisitado a `Comp1` parar, ele entrará no estado *parando* e ficará aguardando a requisição recebida no enlace $\texttt{bind}_2$ (① → ②) terminar, o que não ocorre pois o trecho (③ → ④) da requisição não termina nunca. Deste modo, a precondição para se remover os enlaces $\texttt{bind}_1$ e $\texttt{bind}_2$ não ocorre num tempo finito.

No caso ilustrado, para se poder remover os enlaces $\texttt{bind}_1$ e $\texttt{bind}_2$, `Comp1` deveria ser parado antes de `Comp2`, mas não há como saber desta ordem de parada entre os componentes apenas pela relação de dependência.

Após os devidos componentes serem parados e a aplicação reconfigurada, ela deve retornar ao seu estado funcional, colocando-se os componentes no estado *iniciado*. Para se colocar os componentes neste estado, a precondição é que as interfaces obrigatórias estejam conectadas. Além disso, deve-se estabelecer uma ordem entre os componentes que atendam a esta precondição, com o intuito de evitar um travamento que leve a aplicação a nunca retornar ao seu estado funcional.

Na Figura 4.9, estão ilustradas as situações para serem consideradas na iniciação dos componentes:

1. O componente `Comp1` não depende de outros componentes.

2. O componente `Comp2` possui uma dependência identificada pelo enlace $\texttt{bind}_2$ mas não efetua uma chamada na sua iniciação.

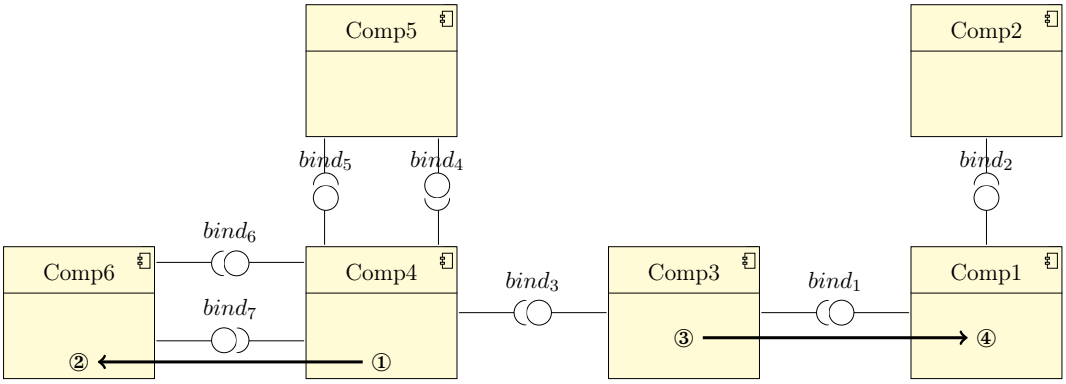


Figura 4.9: Ordem de iniciação de componentes.

3. O componente `Comp3` possui uma dependência identificada pelo enlace $\texttt{bind}_1$. Na iniciação, `Comp3` efetua uma chamada a `Comp1`, ilustrada pela seta (③ → ④).

4. Os componentes `Comp4` e `Comp6` possuem uma dependência circular entre si, identificada pelos enlaces $\texttt{bind}_6$ e $\texttt{bind}_7$, e `Comp4` efetua uma chamada a `Comp6` na iniciação, ilustrada pela seta (① → ②).

5. Os componentes `Comp4` e `Comp5` possuem uma dependência circular entre si, identificada pelos enlaces $\texttt{bind}_4$ e $\texttt{bind}_5$.

Os componentes na situação **1** devem ser os primeiros a serem iniciados pois assim, um componente que dependa deles, como na situação **2**, já terá as suas dependências iniciadas ao ser colocado neste estado. Os componentes na situação **3** devem ser iniciados após as suas dependências terem sido iniciadas, pois eles precisam efetuar uma chamada na iniciação, como a ilustrada por (③ → ④).

Para os componentes que possuem dependência circular, apenas os casos **4** e **5** são possíveis. O caso onde ambos os componentes se chamam na iniciação não é uma configuração admissível, pois cada um ficaria esperando o outro alterar o seu estado, e permaneceriam parados indefinidamente. No caso **4**, `Comp6` deve estar iniciado para que `Comp4` possa ser iniciado. Por fim, no caso **5**, apesar da dependência de `Comp5` não estar iniciada, ele pode ser iniciado pois não efetua uma chamada na iniciação.

Assim, uma ordem de iniciação entre as diferentes dependências dos componentes é:

1. Iniciar componentes que não possuem dependência.
2. Iniciar componentes que possuem todas as suas dependências já iniciadas.
3. Iniciar componentes que não possuem todas as suas dependências já iniciadas, mas não efetuam uma chamada na iniciação.

Na ordem acima e usando-se o diagrama da Figura 4.9, a sequência com que os componentes seriam iniciados é: `Comp1`, `Comp2` ou `Comp3`, `Comp5` ou `Comp6`, e `Comp4`.

4.1.1 Os predicados utilizados para descrever um estado

O interesse desta pesquisa reside na alteração da arquitetura da aplicação, ou seja, dos componentes, suas composições e respectivas interconexões por meio das interfaces.

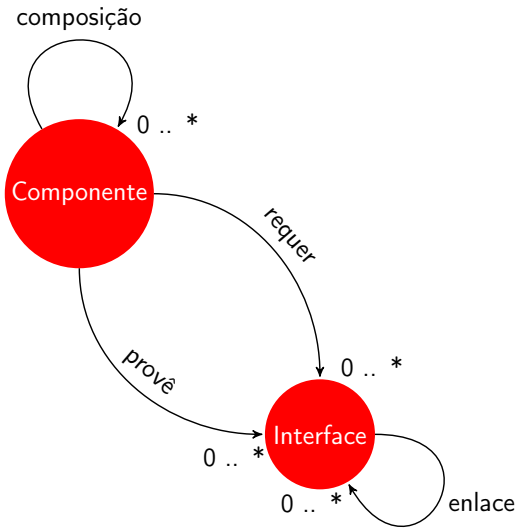


Figura 4.10: Elementos arquiteturais do FRACTAL possíveis de serem alterados .

Assim, considerando a Figura 4.10, as possibilidades de reconfiguração recaem sobre

- o componente;
- a relação de composição entre componentes; e
- a relação de enlace entre interfaces.

Apesar do FRACTAL permitir que um atributo tenha o seu valor alterado em tempo de execução, este tipo de alteração não será considerado neste trabalho, pois esta alteração não modifica a arquitetura da aplicação.

Os conceitos estruturais e as relações entre eles serão traduzidos num conjunto de símbolos proposicionais para serem utilizados na descrição dos estados, como mencionado na Seção 2.3. Estes símbolos estão na listagem abaixo, onde as variáveis são precedidas pelo sinal de interrogação (`?`) e os predicados g_1 a g_3 são utilizadas apenas no estado objetivo.

p_1 - `(component ?nome ?tipoComp)`: O componente denominado `?nome` é do tipo `?tipoComp`, onde o tipo pode ser `single`, no caso de componentes simples e `composite`, no caso de componentes compostos.

p_2 - `(active ?nome)`: O componente denominado `?nome` está criado na configuração inicial.

p_3 - `(contain ?superComp ?subComp)`: O componente denominado `?superComp` contém o componente denominado `?subComp` na configuração inicial. Isso só pode ocorrer se `?superComp` for um componente composto.

p_4 - `(interface ?ifaceName ?ifaceType)`: Uma interface cujo nome é `?ifaceName` e o tipo é `?ifaceType`.

p_5 - `(reference ?compReq ?ifaceName ?cardinality ?contingency)`: Uma interface requerida pelo componente `?compReq`. O nome desta interface é `?ifaceName`, sua cardinalidade é `?cardinality` e sua contingência é `?contingency`. A cardinalidade pode assumir os valores `single` ou `collection` e ela pode ser mandatória, e ter a contingência `mandatory`, ou ser opcional e ter a contingência `optional`.

p_6 - `(internalreference ?comp ?ifaceName)`: Uma interface interna requerida pelo componente composto `?comp`. O nome desta interface é `?ifaceName`. Uma interface interna requerida possui um enlace com uma interface externa provida pelo próprio componente composto.

p_7 - `(service ?comp ?ifaceName)`: Uma interface externa provida pelo componente `?comp`. O nome desta interface é `?ifaceName`.

p_8 - `(internalservice ?comp iface1)`: Uma interface interna provida pelo componente composto `?comp`. O nome desta interface é `?ifaceName`.

p_9 - `(bind ?compReq ?ifaceReq ?compProvided ?ifaceProv)`: Um enlace entre a interface requerida `?ifaceReq`, do componente `?compReq`, e a interface provida `?ifaceProv`, do componente `?compProvided`. Este enlace existe na configuração inicial.

p_{10} - `(started ?comp)`: O componente denominado `?comp` está no iniciado, isto é, um estado onde ele pode enviar e receber requisições.

p_{11} - `(activated ?comp)`: O componente denominado `?comp` está instanciado num estado diferente do inicial.

p_{12} - `(binded ?compReq ?ifaceReq ?compProvided ?ifaceProv)`: Um enlace entre a interface requerida `?ifaceReq`, do componente `?compReq`, e a interface provida `?ifaceProv`, do componente `?compProvided`. Este enlace existe num estado diferente do inicial.

p_{13} - `(contained ?compSuper ?compSub)`: O componente denominado `?compSuper` contém o componente denominado `?compSub`. Esta composição existe num estado diferente do inicial.

p_{14} - `(callatstart ?comp ?ifaceName)`: O componente `?comp` efetua uma chamada na interface requerida `?ifaceName` durante a sua iniciação (alteração do ciclo de vida de *parado* para *iniciado*).

g_1 - `(to-active ?comp)`: O componente denominado `?comp` pertence à configuração objetivo.

g_2 - `(to-contain ?compSuper ?compSub)`: Na configuração objetivo o componente denominado `?compSuper` é um supercomponente do componente `?compSub`.

g_3 - `(to-bind ?compReq ?ifaceReq ?compProvided ?ifaceProv)`: Na configuração objetivo há um enlace entre a interface requerida `?ifaceReq` do componente `?compReq` e a interface `?ifaceProv` provida pelo componente `?compProvided`.

A representação explícita das interfaces internas - p_6 e p_8 - visa facilitar a tradução do plano gerado para ser executado pela API de reflexão. Por exemplo, se for usada a linguagem FPath, nela uma interface interna de um componente composto é obtida pela expressão `internal-interface`.

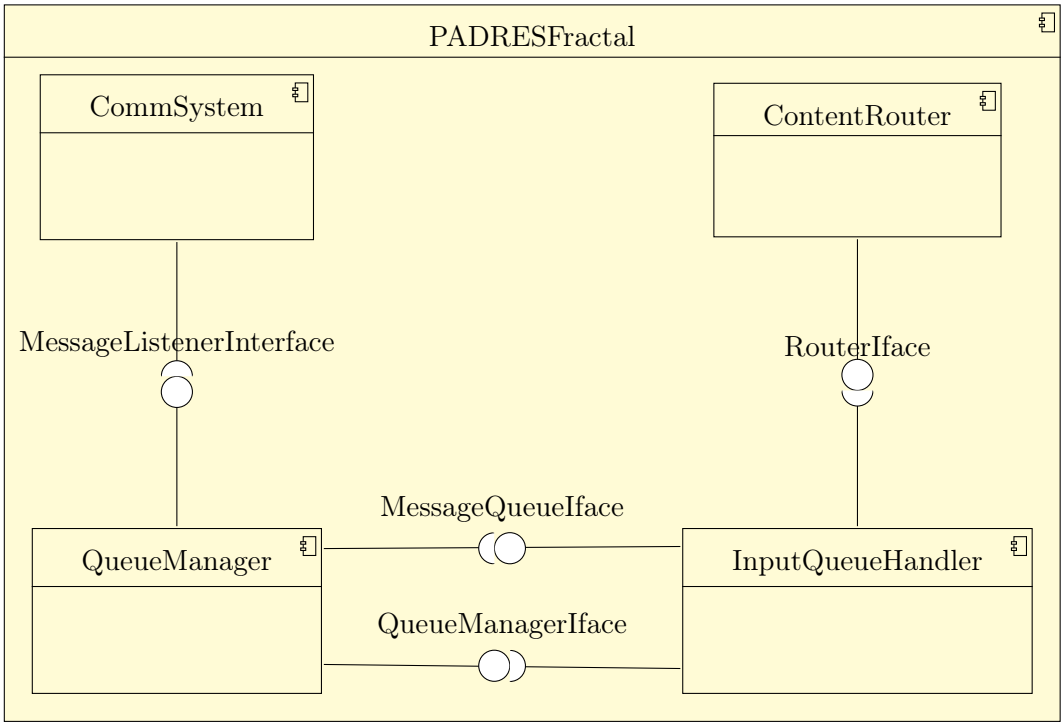


Figura 4.11: Diagrama de componentes de uma configuração.

Considerando-se o diagrama de componentes ilustrado na Figura 4.11, a descrição de uma configuração inicial, por meio dos predicados anteriormente descritos, está na listagem a seguir. Por motivo de espaço, os componentes compostos `CommSystem` e `ContentRouter` não tiveram o seu conteúdo descrito.

```
(component padres composite)
(component commsystem composite)
(component contentrouter composite)
(component inputqueuehandler single)
(component queuemanager single)
(interface ifaceRouter RouterIface)
(interface ifaceMsgQueue MessageQueueIface)
(interface ifaceQueue QueueManagerIface)
(interface ifaceMsgList MessageListenerIface)
(reference inputqueuehandler ifaceRouter single mandatory)
(reference inputqueuehandler ifaceQueue single mandatory)
(reference commsystem ifaceMsgList single mandatory)
(reference queuemanager ifaceMsgQueue single mandatory)
(internalservice commsystem ifaceMsgList)
(internalreference contentrouter ifaceRouter)
(service contentrouter ifaceRouter)
(service inputqueuehandler ifaceMsgQueue)
(service queuemanager ifaceMsgList)
(service queuemanager ifaceQueue)
(active padres)
(active commsystem)
(active queuemanager)
(active inputqueuehandler)
(active contentrouter)
(contain padres commsystem)
(contain padres queuemanager)
(contain padres inputqueuehandler)
(contain padres contentrouter)
(bind commsystem ifaceMsgList queuemanager ifaceMsgList)
(bind queuemanager ifaceMsgQueue inputqueuehandler ifaceMsgQueue)
(bind inputqueuehandler ifaceQueue queuemanager ifaceQueue)
(bind inputqueuehandler ifaceRouter contentrouter ifaceRouter)
(started padres)
(started commsystem)
(started queuemanager)
(started inputqueuehandler)
(started contentrouter)
```

Cada elemento, ou relação, possível de ser reconfigurado pode estar numa das seguintes situações:

1. O elemento existe na configuração inicial, mas não existe na configuração objetivo.

2. O elemento existe na configuração inicial e na configuração objetivo.

3. O elemento não existe na configuração inicial, mas existe na configuração objetivo.

No processo de busca por um plano, o planejador deve ter condições de identificar cada uma destas três situações acima num determinado estado. Os predicados p_{11}, p_{12} e p_{13} servem para diferenciar os estados em que representam as situações 2 e 3.

O predicado p_{11} indica um componente instanciado, pois ele pertence a configuração objetivo. Se este mesmo componente já existia na configuração inicial, então o predicado p_2 também é verdadeiro neste mesmo estado. Isto também vale para as relações de enlace e composição, que envolvem os predicados p_9 - p_{12} e p_3 - p_{13} respectivamente.

Para ilustrar, na Figura 4.12, s_i, s_{i+1} e s_{i+2} são estados e cada um representa uma configuração do software. As transições entre os estados ocorrem pela execução das ações de reconfiguração ilustradas em cada transição.

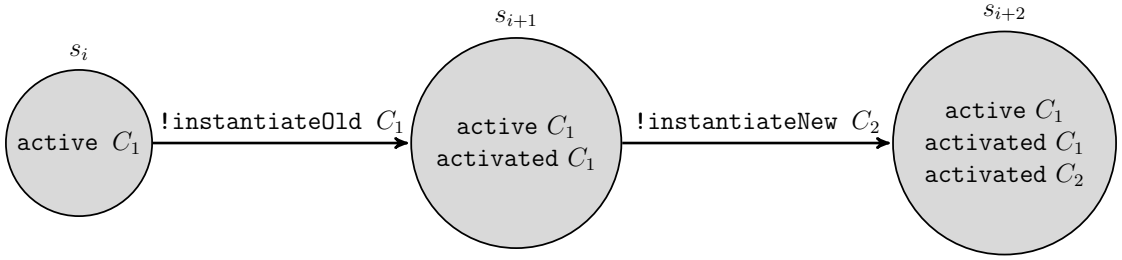


Figura 4.12: Um diagrama de transição de estados.

No estado s_i, o componente C_1 está ativo, indicando que este componente está instanciado na configuração inicial s_0. Considerando que $C_1 \in s_g$, então ele deverá ser instanciado, o que é feito pela ação `!instantiateOld` C_1, que irá efetuar uma transição do estado s_i para o estado s_{i+1} por meio da adição do predicado do tipo p_{11}. Um outro componente C_2, não pertencente a configuração inicial mas $C_2 \in S_g$, também deverá ser instanciado, o que é feito pela ação `!instantiateNew` C_2, que irá efetuar uma transição do estado s_{i+1} para o estado s_{i+2} acarretando na adição de um outro predicado do tipo p_{11}.

Um operador de planejamento com sufixo `Old` serve para tratar elementos arquiteturais e relações que existem em s_0 e S_g, como o caso de C_1. Já um operador com sufixo `New` trata daqueles que existem apenas em S_g, como o caso de C_2. As ações com sufixo `Old` alteram um estado usado pelo planejador porém, em relação a arquitetura da aplicação, elas não efetuam uma reconfiguração de fato. Assim, essas

ações do plano não serão traduzidas em ações de reconfiguração do modelo de componentes e serão removidas do plano obtido pelo planejador no posprocessamento.

Os predicados precedidos de `to-` (g_1, g_2 e g_3) representam o estado objetivo. Estes predicados descrevem quais os componentes que estarão instanciados, quais as relações de composição e os enlaces que devem existir neste estado.

Não foi modelado a relação entre tipos de uma interface. O tipo de uma interface não admite subtipo ou supertipo e um enlace só pode ser feito entre interfaces do mesmo tipo.

4.1.2 Os operadores de planejamento

A capacidade do FRACTAL de ser reconfigurado dinamicamente é provida por um conjunto de classes de controle (*controllers*). Há também uma *Domain Specific Language* (DSL) denominada FPATH/FSCRIPT (DAVID *et al.*, 2009) que facilita o uso desta API. A FPATH permite fazer uma introspecção na aplicação, navegando pela sua arquitetura, e a FSCRIPT permite realizar as operações de reconfiguração. As classes da API e respectivas ações de reconfiguração em FSCRIPT utilizadas estão descritas a seguir.

Factory - Cria componentes de um tipo.

Os principais métodos da API em Java são:

- `public Component newFcInstance()`.

Em FSCRIPT, a criação de componentes é feita pela ação:

- `new(definition)`.

Content Controller - Adiciona e remove componentes, permitindo formar a composição entre componentes.

Os principais métodos da API em Java são:

- `public void addFcSubComponent (Component subComponent)`.
- `public void removeFcSubComponent (Component subComponent)`.

Em FSCRIPT, a adição e remoção de componentes são realizadas pelas seguintes ações:

- `add(parent, child)`.
- `remove(parent, child)`.

Binding Controller - Realiza e desfaz o enlace entre as interfaces requeridas e providas dos componentes, sejam elas externas ou internas.

Os principais métodos da API em Java são:

- `public void bindFc (String clientItfName, Object serverItf)`.
- `public void unbindFc (String clientItfName)`.

Em FSCRIPT, o controle de enlaces é feito pelas seguintes ações:

- `bind (clientItfName, serverItf)`.
- `unbind (clientItfName)`.

Lifecycle Controller - Gerencia o ciclo de vida do componente.

Os principais métodos da API em Java são:

- `public void startFc()`.
- `public void stopFc()`.

Em FSCRIPT, o controle do ciclo de vida do componente é feito pelas seguintes ações:

- `start()`.
- `stop()`.

Uma reconfiguração deve respeitar o modelo de componente subjacente e, caso isso não seja feito, corre-se o risco da aplicação falhar durante a sua atualização. Assim, para se alterar um sistema baseado em componentes com segurança, em LÉGER (2009), o autor propõe o conceito de restrições de integridade para garantir a confiabilidade desta reconfiguração, ou seja, que uma reconfiguração não viole as restrições do modelo de componentes causando uma parada na aplicação.

Este conceito de restrições de integridade, descrito na Seção 2.4, está dividido em dois aspectos: o aspecto estático e o dinâmico. No estático, verifica-se a corretude da *configuração* de uma determinada aplicação perante o modelo de componentes empregado. Quanto a parte dinâmica, verifica-se a corretude da *reconfiguração* em relação as características reflexivas do modelo de componentes.

Para que um plano gerado respeite estas restrições, o domínio de planejamento deve considerá-los, a fim de gerar planos corretos perante o modelo. Assim, os operadores de planejamento que serão instanciados em ações devem considerar estas restrições para poderem ser instanciados. Considerando a semântica de planejamento, estas restrições serão traduzidas em precondições e poscondições, ou seja,

para que uma operação seja aplicável num determinado estado - que representa uma configuração do software - a precondição tem que valer neste estado. Após a aplicação do operador instanciado, obtém-se um outro estado, por meio da aplicação dos efeitos da operação.

Uma alteração num enlace, ou a remoção de um subcomponente, enquanto os componentes estão executando pode causar a perda de mensagens, colocar uma aplicação num estado inconsistente ou mesmo fazê-la falhar. Assim, o ciclo de vida de um componente pode ser controlado apoiando a reconfiguração.

Após listar as proposições que são utilizadas para descrever um estado, serão descritos os operadores de planejamento. Um operador de planejamento será instanciado numa ação caso as precondições sejam verdadeiras neste estado e o operador seja aplicável. A execução de uma ação, irá alterar uma configuração de software e esta alteração é decorrente dos efeitos desta ação. Estes efeitos são descritos por meio da remoção e adição de proposições ao estado em que a ação foi aplicada.

A seguir, os operadores serão agrupados e descritos num nível de abstração mais elevado que a linguagem dos operadores. A descrição completa dos operadores de planejamento está contida no Anexo A.

Instanciação e destruição de componentes - um componente que estará presente na configuração objetivo S_g deve ser instanciado. A instanciação de um componente é feita por meio dos operadores `(!instantiateNew ?comp)` e `(!instantiateOld ?comp)`. O operador `(!instantiateNew ?comp)` coloca um componente que não pertencia a configuração inicial s_0 em condições de ser empregado, representando a instanciação do componente. O operador `(!instantiateOld ?comp)` é usado quando o componente `?comp` pertence a S_g e já existia na configuração inicial. Quanto à remoção, no modelo de componentes FRACTAL não há uma operação de reconfiguração que remova um componente explicitamente. Na implementação JULIA, se a instância não é mais referenciada em nenhuma parte do sistema ela se torna um componente isolado. Assim, ele é removido da memória por meio do mecanismo coletor de lixo (*garbage collector*) do Java. Entretanto, no domínio de planejamento foi utilizada esta ação para efetuar a remoção do componente da aplicação e evitar que uma ação se torne aplicável indevidamente. A precondição da operação de remoção é o componente não estar contido em nenhum outro e não possuir qualquer tipo de enlace numa interface.

Inserção e remoção de componentes na hierarquia - o modelo de componentes FRACTAL permite a composição entre componentes. Um componente composto não possui uma implementação, ele apenas encapsula os subcomponentes. Ele pode conter outros componentes, sejam eles compostos ou simples, mas um componente não pode ser filho dele mesmo, seja diretamente ou na hierarquia de composição. Este último caso ilustra a transitividade da relação de composição no

FRACTAL. Por exemplo, seja C o conjunto de componentes de uma aplicação e $\{c_1, c_2, c_3\} \subseteq C$. Seja $\mathcal{R}$ uma relação de composição entre componentes tal que $\mathcal{R} \subseteq C \times C$ e $(c_p, c_c) \in \mathcal{R}$ significa que c_p é o supercomponente de c_c. Então, se c_1 é um subcomponente de c_2 e c_2 é um subcomponente de c_3 tem-se que $(c_1, c_2) \in \mathcal{R}$ e $(c_2, c_3) \in \mathcal{R}$. Como $\mathcal{R}$ é transitiva, então $(c_1, c_3) \in \mathcal{R}$ e então $(c_3, c_1) \notin \mathcal{R}$.

Em ambos os operadores - `!addChildNew` e `!addChildOld` - a expressão (`child` ?`compFather` ?`compChild`) significa o axioma `child` que retorna verdadeiro quando ?`compChild` é um subcomponente (filho) de ?`compFather` na configuração objetivo. Desta forma este axioma verifica a relação de composição considerando a sua transitividade. A expressão lógica deste axioma, que faz parte do domínio, está na Listagem 4.13. Neste axioma, cada linha é uma precondição lógica e um axioma é verdadeiro se alguma precondição é verdadeira.

```
(:- (child ?compFather ?compChild)
        (contained ?compFather ?compChild)
        ((contained ?compFather ?compAnyChild) (child ?compAnyChild ?compChild))
)
```

Figura 4.13: Axioma `child`.

Para que um componente seja retirado de um supercomponente, ele não deve possuir enlaces locais (ver Seção 4.1) e o supercomponente deve estar no estado parado.

Enlaces entre interfaces e remoção de enlaces - na conexão entre duas interfaces de componentes, uma interface requerida é conectada a uma interface provida e as interfaces possuem o mesmo tipo. Se o enlace ainda não existe na configuração inicial, ele será efetuado por meio da operação (`!bindNew`). Se ele já existe em s_0, então a operação (`!bindOld`) é empregada.

Quanto a interfaces internas, que ocorrem em componentes compostos, a operação de reconfiguração para o FRACTAL é a mesma para uma interface externa. Entretanto, para se referenciar este tipo de interface, a API é específica, bem como na expressão em FPATH. Para tornar mais fácil a tradução do plano gerado pelo planejador, foram criadas duas operações para efetuar o enlace de uma interface interna e uma operação para desfazê-lo.

A desconexão de um enlace só pode ser feita quando o componente da interface requerida encontra-se no estado parado do seu ciclo de vida. Quanto a interfaces internas, novamente foi criada uma operação para a desconexão deste tipo de interface.

Modificação do ciclo de vida do componente - o controle do ciclo de vida do componente permite obter as precondições de operações de reconfiguração, bem como atingir o estado para se efetuar uma reconfiguração e depois recolocar a apli-

cação em funcionamento.

Para componentes compostos, a semântica esperada é que os subcomponentes sempre acompanhem o estado do supercomponente. Com isto, a membrana do supercomponente é responsável por parar e iniciar todos os seus subcomponentes. Entretanto, como explicado anteriormente e ilustrado nas Figuras 4.7 e 4.9, faz-se necessário obedecer a uma ordem de iniciação e parada, para se reduzir o risco de que a aplicação nunca atinja o estado necessário, e comprometa o seu funcionamento. Assim, ou a membrana possui o conhecimento necessário para saber a ordem de parada e iniciação de todos os seu subcomponentes, ou isto é considerado pelo planejador ao gerar o plano de reconfiguração. A solução adotada para o controle do ciclo de vida foi considerar a ordem necessária na geração do plano, que será discutida na próxima seção.

4.1.3 As relações de ordem numa reconfiguração

Num modelo de componentes hierárquico, há novos conceitos e restrições que levam a modificação da relação de ordem parcial entre as ações de reconfiguração. Assim, para o modelo de componentes FRACTAL, as seguintes restrições devem ser respeitadas por um plano de reconfiguração e, consequentemente, pelo procedimento de geração do plano:

- Na remoção de componentes, primeiro são removidos os enlaces nas interfaces providas pelos componentes para depois serem removidos os enlaces das interfaces requeridas. Isso está de acordo com a explicação dada na Seção 4.1 sobre a ordem de parada de componentes para a remoção dos enlaces.

- Um componente, após ser criado, deve ser inserido no supercomponente. Por isso, primeiro são criados e movimentados os componentes compostos para depois serem criados e movimentados os componentes simples. O supercomponente mais alto da aplicação não pode ser alterado entre duas configurações, garantindo que sempre haja um supercomponente disponível.

- Como no FRACTAL os enlaces devem ser locais (ver Seção 4.1), eles só podem ser efetuados após o componente estar posicionado na hierarquia.

- Se uma interface está conectada na configuração inicial e deve ser conectada a uma outra interface na configuração objetivo, primeiro o enlace precisa ser removido para depois ser efetuado o novo enlace.

- A remoção de um componente composto implica na remoção de seus subcomponentes, por isso, antes da remoção de um componente composto já foram

feitas as devidas remoções da aplicação de componentes simples e o reposicionamento na hierarquia (ver os dois primeiros itens).

- Um componente só pode ser iniciado se todas as suas interfaces requeridas possuírem enlaces. Além disso, como explicado na Seção 4.1, primeiro iniciam-se os componentes que não possuem dependências, depois os componentes que possuem todas as suas dependências já iniciadas e por fim os componentes que não possuem todas as suas dependências já iniciadas e não efetuam uma chamada na iniciação.

Dada as restrições anteriores, uma relação de ordem identificada para atender às restrições de integridade do FRACTAL é:

1. Remover todos os componentes simples *comp* quando:

- $comp \in Config_{ini} \quad \wedge \quad comp \notin Config_{goal}$

1.1 Para cada *comp*, remover todos os enlaces a interfaces providas por *comp*.

1.2 Para cada *comp*, remover todos os enlaces a interfaces requeridas por *comp*.

1.3 Para cada *comp*, remover *comp* de todos os supercomponentes nos quais está contido.

2. Reconfigurar a hierarquia de componentes quando:

- $(comp \in Config_{ini} \quad \wedge \quad comp \in Config_{goal}) \quad \vee \quad (comp \notin Config_{ini} \quad \wedge \quad comp \in Config_{goal})$

2.1 Criar e inserir os componentes compostos nos supercomponentes criados.

2.2 Remover os componentes compostos dos supercomponentes.

2.3 Criar e inserir os componentes simples nos supercomponentes criados.

2.4 Remover os componentes simples dos supercomponentes.

3. Remover os enlaces *bind* entre componentes quando:

- $bind \in Config_{ini} \quad \wedge \quad bind \notin Config_{goal}$

4. Efetuar os enlaces *bind* entre componentes quando:

- $bind \in Config_{goal}$

5. Remover os componentes compostos *composite* quando:

- $composite \in Config_{ini} \quad \wedge \quad composite \notin Config_{goal}$

5.1 Para cada *composite*, remover todos os enlaces a interfaces providas por *composite*.

5.2 Para cada *composite*, remover todos os enlaces a interfaces requeridas por *composite*.

5.3 Para cada *composite*, remover *composite* de todos os supercomponentes nos quais está contido.

6. Iniciar os componentes *comp* quando:

- $comp \in Config_{goal} \quad \wedge \quad \neg started(comp)$

6.1 Iniciar os componentes que não possuem interfaces requeridas.

6.2 Iniciar os componentes que possuem suas interfaces requeridas com enlaces efetuados e já iniciados.

6.3 Iniciar os componentes que possuem suas interfaces requeridas com enlaces efetuados e sem dependência na iniciação.

Esta ordem será refletida no conjunto de métodos que compõem o domínio de planejamento e será discutido na seção a seguir.

4.1.4 Os métodos do domínio de planejamento

O domínio de planejamento desenvolvido nesta tese, para o FRACTAL, está descrito no Apêndice B. Aqui será explicada a estrutura do domínio e o procedimento do planejador na busca por planos, o que irá permitir mostrar que a ordem anteriormente identificada para o FRACTAL é respeitada.

Num planejador hierárquico, o domínio é um conjunto de *métodos*, onde um método corresponde a um meio de decompor alguma tarefa, expandindo-a num conjunto de subtarefas (tarefas menores). Uma *tarefa* pode ser composta ou primitiva. Uma *tarefa composta* pode ser expandida por um outro método de decomposição de tarefas enquanto uma *tarefa primitiva* é representada por um operador de planejamento. Um método pode conter diferentes maneiras de decompor uma tarefa e cada decomposição contida num método, possui suas precondições que indicam os estados onde ela é aplicável.

O principal método do domínio está ilustrado na Figura 4.14. Este é o método indicado no problema de planejamento. Este método possui apenas uma decomposição, que está identificada por `_0configure`. As tarefas que decompõem um método possuem uma ordem total e esta ordem é de cima para baixo, ou seja, a subtarefa

```
(:method (configure)
    _0configure
    ; PRECONDITIONS
    (
      (component ?comp ?var1)
    )
    ; SUBTASKS
    (
      (destroySingle)
      (reconfigureHierarchy)
      (removeBind)
      (makeBind)
      (destroyComposite)
      (startComponents)
    )
); END
```

Figura 4.14: Método `configure`.

`(destroySingle)` é executada antes de `(reconfigureHierarchy)`. A ordem da decomposição `_0configure` segue a relação descrita anteriormente, na Seção 4.1.3.

Um método pode conter várias formas de decomposição em tarefas, como o método `destroySingle` ilustrado na Figura 4.15, que contém duas decomposições: `_0destroySingle` e `_9destroySingle`. A precondição para se empregar a decomposição `_0destroySingle` é haver um componente simples *comp* tal que $comp \in s_0 \quad \wedge \quad comp \notin s_g$. Para a decomposição `_9destroySingle` a precondição é verdadeira quando não há um componente simples a ser removido, ou seja, todos os componentes simples a serem removidos já o foram e, nesta decomposição, não há subtarefas pois nada precisa ser feito.

Num método, o planejador busca por uma decomposição aplicável no estado atual de cima para baixo. Para ilustrar, ainda na Figura 4.15, a aplicabilidade da decomposição `_0destroySingle` é verificada antes de `_9destroySingle`. Enquanto houver componentes simples a serem removidos a decomposição `_0destroySingle` é utilizada. Deste modo, a subtarefa `destroySingle`, chamada recursivamente na decomposição `_0destroySingle`, irá remover todos os componentes simples.

Na decomposição das tarefas o planejador monta a rede hierárquica de tarefas. Um exemplo desta rede para o domínio empregado está ilustrada na Figura 4.16, que foi formada pelos métodos: `configure`, `destroySingle`, `removeBindingTo` e `removeBindingComp`.

Os retângulos de borda arredondada indicam as tarefas compostas, os de borda chanfrada indicam as precondições para uma decomposição em tarefas e os retângulos indicam um operador, cujo nome é precedido do sinal de exclamação **!**, e os de borda chanfrada indicam as precondições para uma decomposição em tarefas. A ordem de execução das subtarefas, que na listagem do método `configure` (Figura 4.14) era de cima para baixo, na rede ilustrada é da esquerda para a direita, ou seja, `destroySingle` é decomposta antes de `reconfigureHierarchy` e esta ordem

```
(:method (destroySingle)

  _0destroySingle
  ; PRECONDITIONS single component not present in goal configuration
  (
    (component ?comp single)
    (active ?comp)
    (not (to-active ?comp))
  )
  ; SUBTASKS
  (
    (removeBindingTo ?comp)
    (removeBindingFrom ?comp)
    (removeComponentFromAllParent ?comp)
    (!removeComponent ?comp)
    (destroySingle)
  )

  _9destroySingle
  ; PRECONDITIONS no more single components to be removed
  (
    (not (
      (component ?comp single)
      (active ?comp)
      (not (to-active ?comp))
    ))
  )
  ; SUBTASKS
  ()
); END
```

Figura 4.15: Método `destroySingle`.

está indicada pela direção do vetor pontilhado e rotulado de "ordem de decomposição". Um plano é formado apenas pelos operadores - as folhas da rede - e eles são executados da esquerda para a direita, como os operadores `!stop ?compReference` e `!unbind ?compReference ?ifaceRef ?comp ?ifaceServer` que serão instanciados em ações e irão compor um plano. Na Figura 4.16, o caminho indicado pelas arestas destacadas - setas em linha dupla - mostra um possível caminho até um operador de planejamento (tarefa primitiva) que será instanciado numa ação.

4.2 Validação do domínio

O planejador JSHOP2 é a versão na linguagem Java do planejador hierárquico SHOP2 e, em NAU *et al.* (2001), os autores mencionam que o planejador SHOP2 é correto e completo. Porém, para garantir que os planos sejam corretos, deve-se mostrar que o domínio é correto. Em outras palavras, se o domínio de planejamento é correto em relação à realidade que ele representa.

Como descrito na Seção 2.3.4, neste trabalho, para se garantir que um domínio de planejamento gera planos corretos, serão executadas as seguintes atividades:

a. Verificar os operadores.

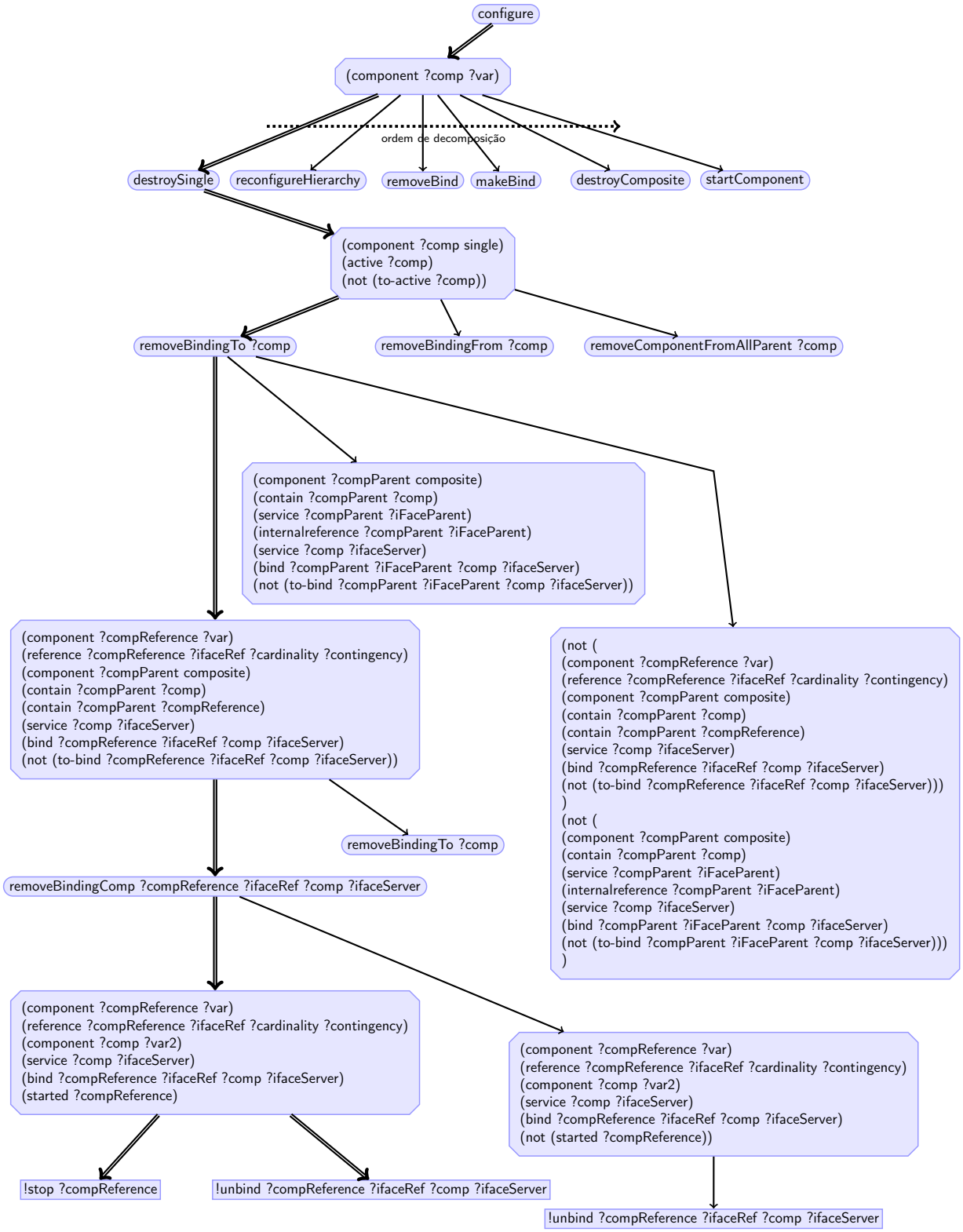


Figura 4.16: Uma representação gráfica da rede de tarefas dos métodos: `configure`, `destroySingle`, `removeBindingTo` e `removeBindingComp`.

b. Verificar que a relação de ordem é obedecida.

c. Aplicar testes positivos aos métodos.

O item **A** pode ser feito comparando-se os operadores com as ações de reconfiguração provenientes do FRACTAL. Como ilustrado na Figura 4.16, um operador é obtido após aplicar a decomposição descrita nos métodos e isto significa que as precondições de cada uma das decomposições, até se chegar ao operador, também devem ser verdadeiras no estado em que o operador é aplicável. Assim, para se mostrar o item **A**, todos os possíveis caminhos para cada operador serão verificados.

Já o item **B** deve levar em conta a ordem de decomposição estabelecida pelos métodos, que irá prover uma ordem entre os operadores. Por exemplo, observando-se a Figura 4.14, vê-se que um componente simples é removido antes de serem criados novos componentes ou que enlaces são removidos antes de serem efetuados novos enlaces.

4.2.1 Validação dos operadores

A rede de tarefas montada para o domínio usado possui 99 diferentes caminhos entre o método `configure` e um operador de planejamento. Os caminhos foram agrupados por operador, formando o conjunto das possíveis precondições que podem ocorrer para cada operador. Primeiramente serão vistos os operadores de instanciação e destruição de componentes. Em seguida, os operadores de composição de componentes, que incluem a inserção e remoção de um componente num supercomponente. Depois, os operadores que efetuam e desfazem os enlaces entre as interfaces dos componentes. Por fim, os operadores de controle do ciclo de vida dos componentes.

Instanciação e destruição de componentes

Os operadores de instanciação são `!instantiateOld` e `!instantiateNew`. Para o operador `!instantiateNew` um dos caminhos nunca ocorre, pois se trata do caminho percorrido para se iniciar o componente mais elevado da hierarquia. Como explicado na Seção 4.1.3, para o componente mais elevado somente o operador `!instantiateOld` é aplicável e é obtido pela decomposição `_0activeOld`. O caminho que nunca ocorre é formado pelo encadeamento da decomposição `_0reconfigureHierarchyHighestComposite` do método `reconfigureHierarchy`, com a decomposição `_1activeNew` do método `activate`. Neste caminho, a precondição contém os predicados `(active ?comp)` e `(not (active ?comp))` e não é possível uma substituição para `?comp` que seja verdadeira.

Assim, os operadores são obtidos na seguinte quantidade de caminhos:

- `!instantiateOld` - 3 caminhos.
- `!instantiateNew` - 3 caminhos.

Uma outra observação é que, no domínio de planejamento considera-se que um componente só pode ser criado se os seus supercomponentes já estiverem criados.

Isto é obtido pela precondição ilustrada na listagem 4.17 e que faz parte dos caminhos aplicáveis a instanciação dos componentes que não estão posicionados no topo da hierarquia de composição. Nela, o planejador verifica se todos os componentes que deverão conter o componente a ser criado - predicado `(to-contain ?compFather ?comp)` - já estão criados - predicado `(activated ?compFather)`.

```
(forall
    (?compFather)
    (to-contain ?compFather ?comp)
    (activated ?compFather)
)
```

Figura 4.17: Precondição para se criar um componente.

Como há sempre um componente para o qual esta precondição é aplicável, pois o componente mais elevado da hierarquia foi criado antes desta condição ser verificada, os subcomponentes diretos já podem ser instanciados. Assim a criação da hierarquia de composição é formada de cima para baixo, similar a uma busca em largura. Esta precondição foi introduzida pois um componente é criado e inserido no seu supercomponente, evitando que após a sua criação ele venha a ser removido pelo mecanismo coletor de lixo.

Como já mencionado, no FRACTAL não há uma ação de remoção de um componente de uma aplicação e, um componente que não esteja contido em outro, é removido da aplicação pelo mecanismo coletor de lixo. Entretanto, no domínio este operador foi criado e utilizado para permitir alterar os estados necessários na busca por um plano. Estes dois caminhos ocorrem em dois métodos: `destroySingle` que remove os componentes simples e `destroyComposite` que remove os componentes compostos.

- `!removeComponent` - 2 caminhos.

Composição de componentes

A adição de componentes na hierarquia ocorre ao ser decomposto o método `reconfigureHierarchy`. Dentro dele, os componentes são adicionados nos respectivos supercomponentes antes de serem removidos evitando que um componente fique sem um supercomponente ao longo de uma sequência de reconfiguração. Para os operadores de composição, o número de caminhos possíveis para cada um deles está descrito abaixo.

- `!addChildOld` - 2 caminhos.

- `!addChildNew` - 2 caminhos.

A remoção de um subcomponente irá ocorrer nas tarefas que decompõem os métodos:

- `destroySingle` - retira da aplicação os componentes simples *comp* tais que: $comp \in Config_{ini} \wedge comp \notin Config_{goal}$.

- `reconfigureHierarchy` - reposiciona os componentes *comp* compostos e simples, nesta ordem, tais que: $comp \in Config_{ini} \wedge comp \in Config_{goal}$; e

- `destroyComposite` - retira da aplicação os componentes compostos *comp* tais que: $comp \in Config_{ini} \wedge comp \notin Config_{goal}$.

O método cuja decomposição contém o operador `!removeChild` é o método `(removeCompFromFather ?compFather ?comp)`, que trata da remoção de um componente `?comp` do supercomponente `?compFather`. A remoção ocorre em:

- `!removeChild` - 16 caminhos.

Enlace entre interfaces

Para os operadores que efetuam e desfazem os enlaces, o número de caminhos possíveis para cada um deles está descrito abaixo. Relembra-se que foram criados métodos específicos para representar enlaces que envolvem uma interface interna, a fim de facilitar a tradução do plano para a API do FRACTAL. Um enlace entre interfaces é efetuado pelo método `makeBind`, que efetua os enlaces *bind* tais que: $bind \in Config_{goal}$.

- `!bindOld` - 1 caminho.

- `!bindNew` - 1 caminho.

- `!bindInternalOld` - 2 caminhos.

- `!bindInternalNew` - 2 caminhos.

O operadores que removem os enlaces, abaixo descritos, ocorrem em:

- `!unbind` - 10 caminhos.

- `!unbindInternal` - 16 caminhos.

A remoção de enlaces é feita nos seguintes métodos:

- `destroySingle` e `destroyComposite` - retiram da aplicação os componentes *comp* tais que: $comp \in Config_{ini} \wedge comp \notin Config_{goal}$. Para um componente ser removido da aplicação, ele deve primeiro ser removido de seu supercomponente e por isso não deve possuir enlaces locais.

- `reconfigureHierarchy` - este método reposiciona os componentes na hierarquia de composição. Um componente que pertence a $Config_{ini}$, permanece em $Config_{goal}$ mas num outro supercomponente, precisa ser removido. Para ser removido de um supercomponente, ele não deve possuir enlaces com irmãos e com o supercomponente.

- `removeBind` - remove os enlaces *bind* tais que: $bind \notin Config_{goal}$.

Na ordem da decomposição em tarefas do método `configure`, ilustrada na Listagem 4.14, o método `destroyComposite` vem após o método `removeBind`. Por isso, os enlaces dos componentes compostos a serem removidos serão desfeitos por `removeBind` não existindo enlaces no instante do método `destroyComposite`.

Ciclo de vida do componente

Há dois operadores que controlam o ciclo de vida: `!stop` e `!start`. A chamada ao operador `!stop` é para se obter o estado parado de um componente, que é a precondição dos operadores `!removeComponent`, `!removeChild`, `!unbind` e `!unbindInternal`. Ao longo da decomposição descrita nos métodos `destroySingle`, `reconfigureHierarchy`, `removeBind` e `destroyComposite`, a chamada ocorre em:

- `!stop` - 33 caminhos.

Os componentes parados devem ser iniciados. Isto ocorre no método `startComponents`. O operador `!start` ocorre:

- `!start` - 6 caminhos.

4.2.2 Ordem dos operadores

A ordem entre os operadores será garantida pelo modo como o planejador executa as subtarefas numa decomposição existente num método. Já foi dito que, num método, o planejador busca por uma decomposição aplicável na sequência com que as diferentes decomposições são colocadas, isto é, de cima para baixo. Identificada uma decomposição aplicável, as subtarefas desta decomposição também são resolvidas de cima para baixo. Em outras palavras, numa decomposição a primeira subtarefa é resolvida antes de se resolver a seguinte.

Devido ao número de métodos e respectivas decomposições, ficaria impraticável ilustrá-las por completo num grafo. Com o intuito de ilustrar como a decomposição dos métodos reflete a ordem descrita na Seção 4.1.3, foi desenhada na Figura 4.18 alguns métodos e respectivas decomposições, formando uma árvore. A raiz desta árvore é o método principal `configure`, chamado para resolver o problema de planejamento.

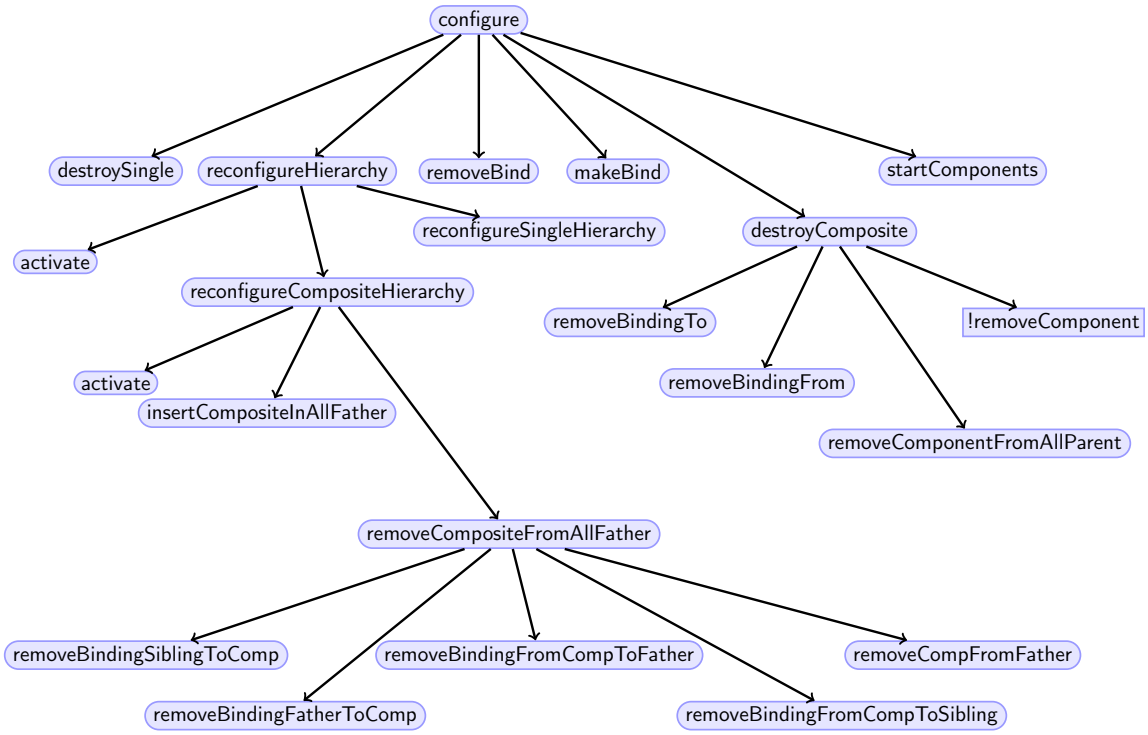


Figura 4.18: Um grafo ilustrando a ordem entre alguns métodos de decomposição pertencentes ao domínio de planejamento.

Primeiramente, todos os componentes simples que devem ser removidos são tratados pelo método `destroySingle`. Em seguida, a hierarquia é reconfigurada. Este reconfiguração significa a instanciação de componentes e a respectiva inserção na hierarquia. O método `reconfigureHierarchy` primeiramente identifica o componente mais alto na hierarquia, que é único, e aplica o método `active`. Definido o componente mais alto, a hierarquia é montada de cima para baixo, primeiro nos componentes compostos, que é tratado pelo método `reconfigureCompositeHierarchy`, e depois nos componentes simples, que é tratado pelo método `reconfigureSingleHierarchy`. Assim, um componente após ser criado tem sempre o seu supercomponente já criado e pode ser inserido na sua posição.

Componentes que permanecem na aplicação, existentes na configuração inicial e na objetivo, podem ser reposicionados na hierarquia. Este reposicionamento é tratado pelos métodos `insertCompositeInAllFather` e `removeCompositeFromAllFather`. Deste modo, primeiro um componente é inserido no novo supercomponente e depois ele é removido do supercomponente anterior. Como um componente pode estar em mais de um supercomponente, devido ao conceito de componente compartilhado, deve-se percorrer todos supercomponentes de um componente a ser movimentado.

Resolvida a hierarquia, os enlaces são removidos pelo método `removeBind` e em

seguida são efetuados pelo método `makeBind`. Agora que toda a hierarquia está montada e os enlaces feitos, os componentes compostos serão removidos. Por fim, os componentes parados são iniciados.

4.2.3 O conjunto de testes

Devido ao número de métodos envolvidos, foram escolhidos os seguintes métodos para serem testados positivamente, onde um problema de planejamento com um plano existente era resolvido pelo planejador que deveria gerar o plano esperado. Para se testar os métodos abaixo, a decomposição do método principal era modificada, deixando apenas a subtarefa que, decomposta, levaria ao método desejado, ou seja, o método `configure` ficava com apenas uma subtarefa na sua decomposição.

- **(removeBindingTo ?comp)**: este método remove todos os enlaces para o componente `?comp`. Os enlaces podem ser provenientes de um componente irmão ou de um supercomponente. No teste, o componente possui dois enlaces de irmãos e dois enlaces de supercomponente. A colocação de dois enlaces em cada tipo visa observar a chamada recursiva do método `(removeBindingTo ?comp)` efetuada na decomposição.

- **(removeBindingFrom ?comp)**: este método remove todos os enlaces do componente `?comp`. Os enlaces podem ser para um componente irmão ou para um supercomponente. No teste, o componente possui dois enlaces para irmãos e dois enlaces para o supercomponente. Como no primeiro teste, a colocação de dois enlaces em cada tipo visa observar a chamada recursiva do método `(removeBindingFrom ?comp)` efetuada na decomposição.

- **(removeComponentFromAllParent ?comp)**: este método remove o componente `?comp` de todos os supercomponentes nos quais está contido. No teste, o componente está contido em dois supercomponentes para observar a chamada recursiva do método `(removeComponentFromAllParent ?comp)` efetuada na decomposição.

- **reconfigureCompositeHierarchy**: este método reconfigura a hierarquia de componentes compostos. Ele é chamado após o componente mais alto da hierarquia ter sido instanciado. No teste, um componente composto deveria ser inserido em dois supercomponentes e removido de outros dois supercomponentes.

- **reconfigureSingleHierarchy**: este método reconfigura a hierarquia de componentes compostos. Ele é chamado após o método `reconfigureCompositeHierarchy`, garantindo que o supercomponente

já estará instanciado e posicionado na hierarquia. No teste, um componente simples deveria ser inserido em dois supercomponentes e removido de outros dois supercomponentes.

- **(makeBindFrom ?comp)**: este método efetua os enlaces a partir de componente `?comp`, da interfaces requeridas por ele. O enlace pode ser de três tipos: enlace para um irmão, enlace para um supercomponente ou enlace para um subcomponente. Foram feitos testes para um componente simples, que cobrem os dois primeiros tipos, e para um componente composto, que cobre os rês tipos. Novamente, para cada tipo foram colocados 2 enlaces a serem verificados.

- **(startComponent ?comp)**: este método efetua a iniciação dos componentes parados. Há 6 possibilidades a serem testadas, onde 3 se aplicam a componentes simples e 3 para componentes compostos. As situações já foram descritas na Seção 4.1.3: primeiro são iniciados os componentes simples e compostos sem dependências, depois aqueles cujas dependências já foram iniciadas e, por fim, os componentes simples e compostos cujas dependências não foram iniciadas, mas eles não efetuam uma chamada durante a iniciação. Cabe lembrar, que para os componentes compostos, ele só é iniciado se os seus subcomponentes já o estiverem.

4.3 Posprocessamento do plano de reconfiguração

A atividade de posprocessamento do plano gerado removerá as ações de planejamento que não devem ser traduzidas para ações de reconfiguração do FRACTAL. Estas ações são:

- as ações com sufixo `Old`.

- as ações `(!removeComponent ?comp)`.

Após a remoção destas ações, o plano será traduzido para a linguagem FPATH / FSCRIPT. Esta tradução é feita de modo manual mas com uma relação de um para um para facilitar, num trabalho futuro, a tradução automatizada. Sobre esta tradução, cabe relembrar que, no caso do enlace de interfaces internas, foram utilizadas duas operações de planejamento específicas: a `(!bindInternal)` e a `(!unbindInternal)`. Isto visa facilitar a identificação das interfaces que devem ser obtidas por uma expressão em FPATH, que é iniciada por `internal-interface`, e que serão colocadas como parâmetro das expressões em FSCRIPT. A tradução destas operações para FSCRIPT será `bind` e `unbind` respectivamente.

4.4 Considerações Finais

Neste capítulo, foi descrita a abordagem proposta para responder as duas perguntas de pesquisa. Para a primeira pergunta, foi empregada a técnica de planejamento automatizado e a Seção 4.1 trata de como esta técnica foi empregada. Seu principal elemento, o domínio de planejamento, foi descrito para o modelo de componentes FRACTAL e considerando os aspectos de reconfiguração dinâmica. Para a segunda pergunta, o processo de validação do domínio foi descrito na Seção 4.2.

Com o intuito de aplicar esta abordagem em exemplos reais de reconfiguração dinâmica e ilustrar como empregar a abordagem, no Capítulo seguinte serão descritas as aplicações e os problemas de planejamento identificados.

Capítulo 5

Estudos de Caso

A abordagem proposta no Capítulo 4 foi empregada na reconfiguração de duas aplicações distintas. A primeira é um servidor HTTP bastante simplificado e frequentemente utilizado em artigos para validar alguma abordagem que use o modelo de componentes FRACTAL. Este exemplo foi utilizado no início da pesquisa como base para iniciar a confecção do domínio de planejamento e entender como diferentes problemas de planejamento afetam a busca por um plano.

Num momento posterior, iniciou-se o uso de uma segunda aplicação - um *broker Publish/Subscribe* - que não havia sido desenvolvida baseada em componentes, mas é voltada para a área de aplicações de interesse desta pesquisa. Ela foi refatorada para ser baseada em componentes e com intuito de ser empregada num cenário de integração de aplicações. No uso desta segunda aplicação, o domínio de planejamento foi refinado, decorrente de novas situações encontradas como a ordem de parada e iniciação de componentes, o comportamento do ciclo de vida de um componente composto e a possibilidade de um componente ser compartilhado.

5.1 O Servidor HTTP

A aplicação baseada em componentes **Comanche** é um servidor HTTP mínimo, com 11 componentes e descrito como exemplo no sítio do FRACTAL[1].

O seu diagrama de componentes está ilustrado na Figura 5.1 e, nele, os enlaces entre interfaces não estão nomeados. O componente `RequestReceiver` recebe as requisições e as escalona, por meio do `Scheduler`, para ser processada pelo `Backend`. O `Backend` é um componente composto que processa as requisições. Dentro deste componente, o `RequestAnalyzer` irá ler a requisição para obter a URL requisitada. O `RequestDispatcher` irá encaminhar cada requisição para o seu respectivo componente. A requisição de um arquivo é tratada pelo `FileReqHandler`, que tentará

[1] http://fractal.ow2.org/tutorials/comanche.html

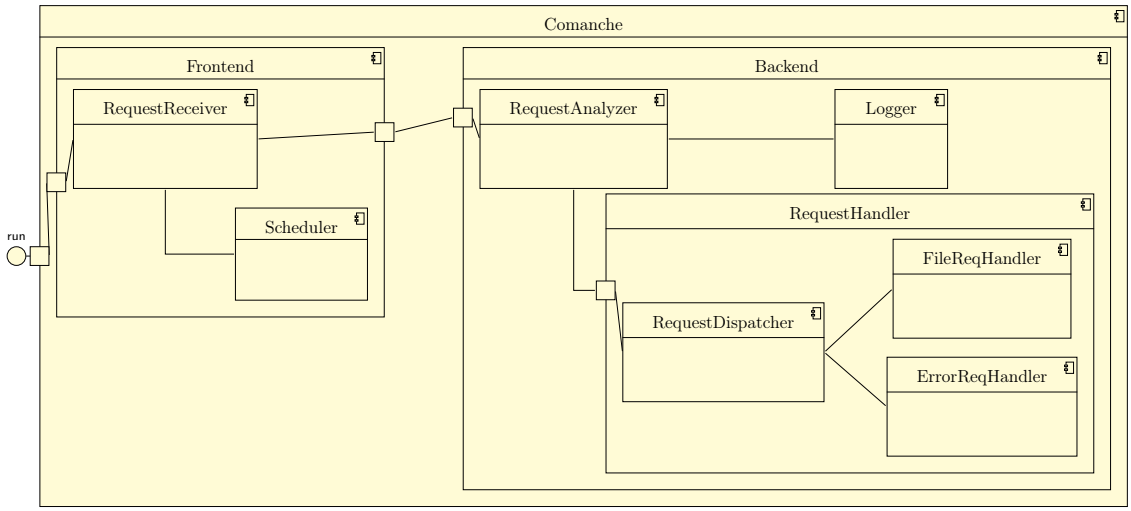


Figura 5.1: Diagrama de Componentes do Servidor Comanche.

obter e enviar ao cliente o arquivo cujo nome corresponde a URL requisitada, se ele existir. O `ErrorReqHandler` irá enviar ao cliente uma mensagem de erro.

5.1.1 Cenário de Uso

Numa configuração inicial, o componente `Scheduler` efetua um escalonamento sequencial, onde as requisições são tratadas uma de cada vez. A partir disso, na configuração objetivo o componente que trata destas requisições deverá fazê-lo em paralelo, gerando uma *thread* para cada requisição. Nela, o componente `schedulerSeq` efetua o escalonamento sequencial e o novo componente `schedulerMT`, que irá substituí-lo, efetua o escalonamento *multithread*.

A reconfiguração é na verdade uma troca de componentes e na Tabela 5.1 está descrito o plano gerado que representa esta troca de componentes. Este plano é obtido após o posprocessamento e as ações de planejamento foram traduzidas para a linguagem FPath e FScript e colocadas na terceira coluna.

Nr	Ações	FPath/FScript
1	(!stop requestReceiver)	stop($frontend/child::requestReceiver)
2	(!stop schedulerSeq)	stop($frontend/child::schedulerSeq)
3	(!unbind requestReceiver schedulerSeq s)	unbind($frontend/child::requestReceiver/client::s, $frontend/child::schedulerSeq/service::Scheduler)
4	(!stop frontend)	stop($frontend)
5	(!removechild frontend schedulerSeq)	remove($frontend, $schedulerSeq)
6	(!instantiateNew schedulerMT)	schMT = new('file:///src/resources/schedulerMT')
7	(!addchildnew frontend schedulerMT)	add($frontend, $schMT/child::schedulerMT)
8	(!bindnew requestReceiver schedulerMT s)	bind($frontend/child::requestReceiver/client::s, $frontend/child::schedulerMT/server::Scheduler)
9	(!start schedulerMT)	start($schMT/child::schedulerMT)
10	(!start requestReceiver)	start($frontend/child::requestReceiver)
11	(!start frontend)	start($frontend)

Tabela 5.1: Plano de reconfiguração da troca do escalonador.

Na coluna **FPath/FScript**, os nomes precedidos de $, como $frontend, repre-

sentam variáveis na linguagem FPath. No caso do $frontend, esta variável aponta para a instância do componente `frontend`, assim como $schedulerSeq representa o componente `schedulerSeq` e $schMT representa o componente `schedulerMT`

5.1.2 Impacto do Problema

Foram avaliadas duas situações na busca por um plano:

1. qual o impacto de uma aplicação maior; e
2. qual o impacto de uma reconfiguração maior.

A primeira situação foi avaliada empregando-se a mesma reconfiguração anterior. Entretanto, a troca do componente `schedulerSeq` pelo `schedulerMT`, foi repetida em dois diferentes cenários. No primeiro, a aplicação foi reduzida de tamanho e o `Comanche` foi composto por apenas o componente `Frontend` e seus 2 subcomponentes, totalizando 4 componentes na aplicação. No segundo, a mesma troca foi feita com a aplicação completa, ilustrada na Figura 5.1, contendo os 11 componentes.

Nr	Reconfiguração	Quantidade de trocas de variáveis
1	Troca (*swap*) de um componente - Comanche com 4 componentes	153
2	Troca (*swap*) de um componente - Comanche com 11 componentes	329

Tabela 5.2: Avaliação da troca de um componente em aplicações de diferentes tamanhos.

Com uma aplicação maior, há um maior número de predicados para representá-la. Isto acarretará num maior número de possibilidades para as variáveis que precisam ser unificadas nas decomposições em tarefas dos métodos. Em ambas as execuções o plano possui o mesmo tamanho, contendo 11 ações, conforme a Tabela 5.1.

A segunda situação a ser avaliada foi uma terceira reconfiguração, que envolve a troca de 3 componentes e a inclusão de um novo componente. Neste caso, os componentes `Scheduler`, `Logger` and `RequestAnalyzer` são trocados por outros e um novo *handler* é adicionado e conectado ao `RequestDispatcher`.

Nr	Reconfiguração	Quantidade de trocas de variáveis
2	Troca (*swap*) de um componente - Comanche com 11 componentes	329
3	Troca (*swap*) de três componentes e inserção de um novo - Comanche com 11 componentes	620

Tabela 5.3: Avaliação do tamanho da reconfiguração em aplicações com o mesmo tamanho.

Um maior número de alterações presentes numa reconfiguração aumenta o número de trocas de variáveis devido ao maior número de ações a serem executadas.

Comparando-se o número de ações, a execução 2 possui 11 ações (tamanho do plano) enquanto na execução 3 o plano contém 36 ações.

Estas avaliações são bem simples e mostram um comportamento esperado devido a relação direta entre os parâmetros avaliados.

5.2 O *broker* de *Publish/Subscribe*

A segunda aplicação utilizada para efetuar as reconfigurações foi o *broker* **PADRES**[2], que efetua um roteamento baseado em conteúdo no paradigma de comunicação Publicação/Assinatura (*Publish/Subscribe*) ou, simplesmente, Pub/Sub. Há inúmeras aplicações de um *broker* deste tipo como disseminação seletiva de informação, serviços baseados em localização e gerenciamento de redes de computadores e, esse tipo de aplicação foi escolhido por possuir características que são importantes para as classes de aplicação de interesse desta pesquisa.

No paradigma de comunicação Pub/Sub, os produtores de informação enviam dados como publicações para o *broker* e os consumidores indicam o seu interesse por meio de assinaturas EUGSTER *et al.* (2003). Uma assinatura possui um conjunto de notificações, que é um conjunto de publicações potenciais, ou seja, publicações que correspondem à assinatura. Ao receber uma publicação, o *broker* determina o subconjunto de assinantes correspondentes e os notifica. Os sistemas de Pub/Sub que efetuam roteamento baseado em conteúdo introduzem um esquema de assinatura baseado no conteúdo dos eventos que são publicados. Os eventos não são previamente classificados de acordo com um esquema, como um tópico, mas de acordo com as suas propriedades.

O PADRES é um *broker* de Pub/Sub distribuído e com roteamento baseado em conteúdo. Ele pode formar um conjunto de *brokers* conectados por uma rede ponto-a-ponto FIDLER *et al.* (2005). A linguagem de assinatura no PADRES é baseada no predicado: **[***atributo, operador, valor***]**, e uma assinatura é uma conjunção de predicados. Cada mensagem possui uma tupla mandatória descrevendo a classe da mensagem, que provê uma seletividade garantida similar ao tópico, empregado em sistemas Pub/Sub baseados em tópicos.

Além destas características funcionais, ele apresenta a possibilidade de se montar uma rede de *broker*, permitindo que um assinante de um *broker* receba uma mensagem publicada num outro *broker*. Há também um componente responsável por manter esta rede atualizada, por meio de mensagens de *heartbeat* enviadas e recebidas pelos *broker* conectados.

O *broker* PADRES foi desenvolvido em Java e seu código fonte está disponível em `http://www.msrg.utoronto.ca/projects/padres/downloads/download_`

[2]http://www.msrg.utoronto.ca/projects/padres/

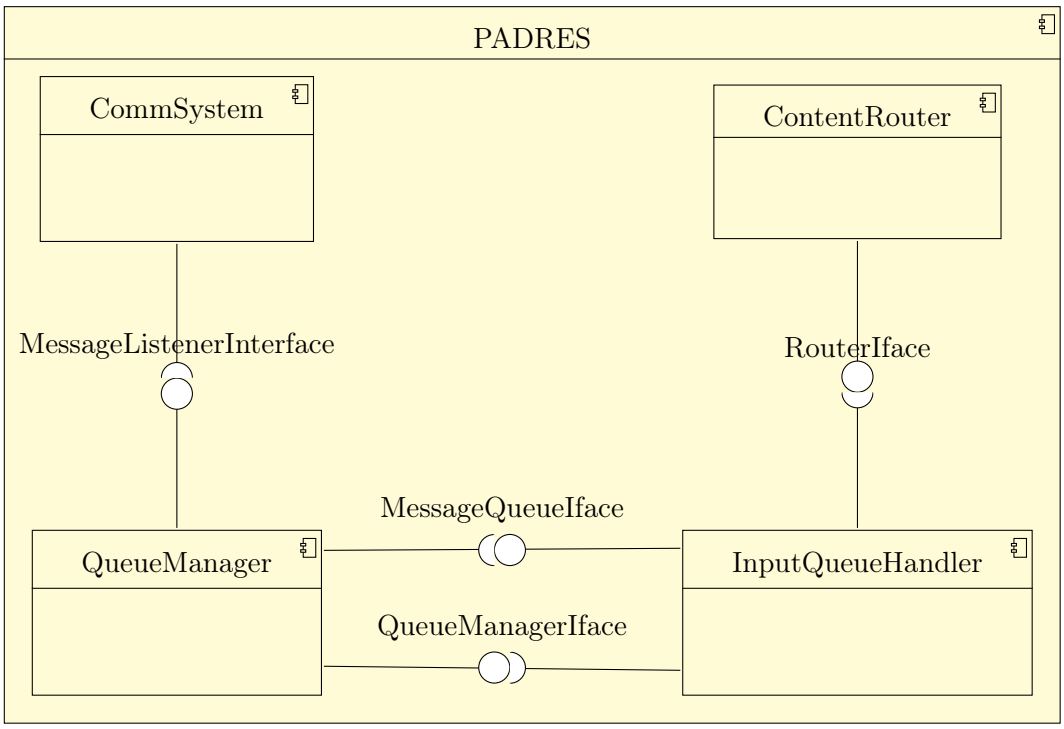


Figura 5.2: Diagrama de componentes do *broker* PADRES.

`binary.php`. Nesta pesquisa, ele foi refatorado e implementado como uma aplicação baseada em componentes, empregando-se o FRACTAL como modelo de componentes. Foram identificados 16 componentes de software e, na Figura 5.2 está ilustrado um diagrama do PADRES que mostra os principais componentes da sua arquitetura.

A geração dos componentes foi feita utilizando-se de um arcabouço de anotações, denominado FRACLET (ROUVOY e MERLE, 2009), que permite anotar o código na linguagem de programação Java, com elementos do FRACTAL e, empregando-se uma abordagem generativa, este código anotado é completado de acordo com este modelo de componentes. Cada componente foi colocado no formato de arquivo empacotado **JAR** (*Java ARchive*) e respectivas classes. No caso de um componente composto, apenas a sua descrição na ADL do FRACTAL é incluída no arquivo JAR, pois há um arquivo para cada um dos subcomponentes.

Numa descrição em alto nível da aplicação, o componente composto `CommSystem`, ilustrado na Figura 5.3, é responsável pela comunicação com os clientes do *broker*. As mensagens recebidas por ele são enviadas ao `QueueManager` que irá destiná-las ao local interno correto. Isto ocorre pois há mensagens que trafegam apenas entre *brokers*, destinadas a manter a rede de *broker* e por isso não fazem parte do roteamento padrão entre assinantes e publicadores. As mensagens dos clientes são enviadas ao `InputQueueHandler` que as repassa ao componente composto `ContentRouter`.

O `ContentRouter` é responsável por fazer o emparelhamento entre publicações e assinaturas, identificando os destinatários da mensagem. A sua composição in-

terna está ilustrada na Figura 5.4. A principal característica do PADRES é o uso do algoritmo de Rete (FORGY, 1990) para executar o emparelhamento baseado no conteúdo, e determinar quais os assinantes que devem receber uma publicação. Este emparelhamento é feito pelo componente `ReteMatcher` que está contido em `ContentRouter`.

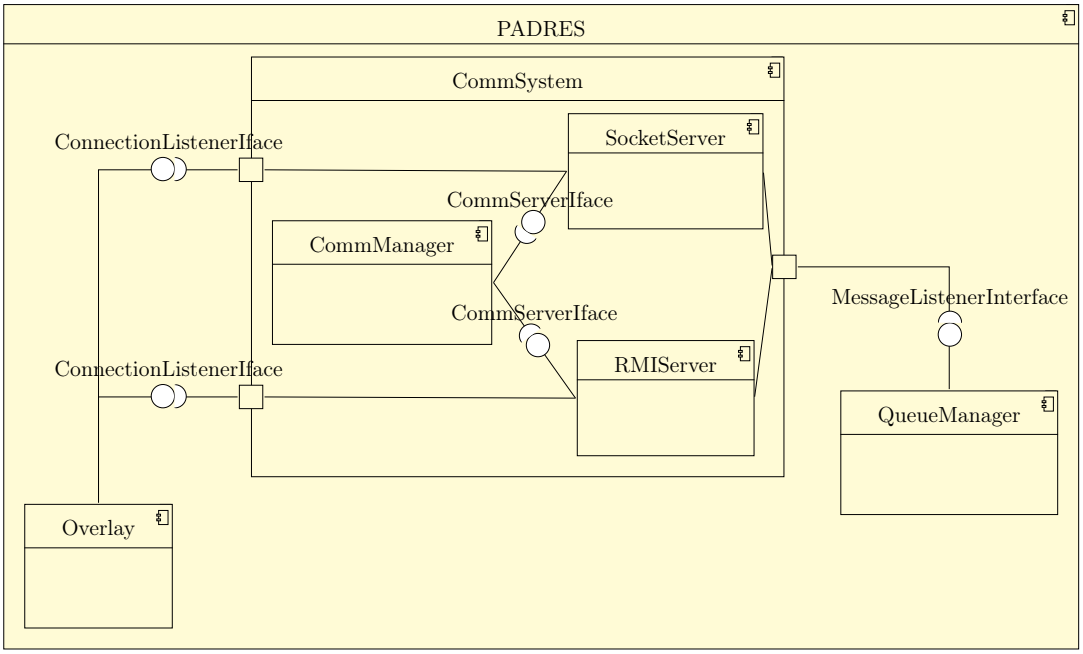


Figura 5.3: Diagrama do Componente Composto `CommSystem`.

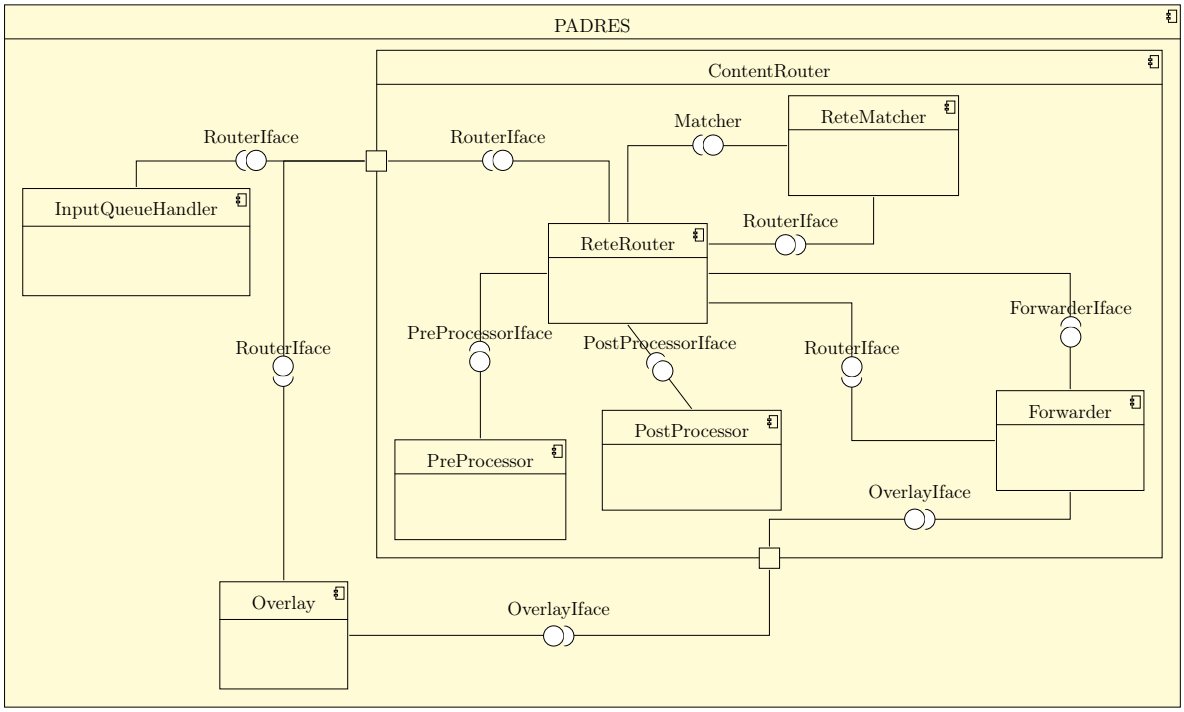


Figura 5.4: Diagrama do Componente Composto `ContentRouter`.

5.3 Cenário de Uso

O PADRES refatorado foi utilizado para integrar dois simuladores para treinamento, utilizados para instrução na Marinha do Brasil, a fim de permitir a troca de informações entre os simuladores. Este estudo de caso, com o uso nos simuladores, servirá de base para a integração dos simuladores ao sistema computacional de apoio ao comando e controle (C2) na Marinha do Brasil, denominado de Sistema Naval de Comando de Controle (SisNC2). O estudo também irá contribuir para, num futuro próximo, integrar os sistemas computacionais de C2 das Forças com o Ministério da Defesa.

O primeiro simulador de treinamento é usado para o treinamento de manobras num passadiço de um navio, num ambiente virtual que representa tanto o navio quanto o cenário onde será feita a manobra. Este simulador contém, na sua arquitetura, um *broker* de Pub/Sub com 5 clientes:

1. Módulo do movimento do navio: possui um modelo matemático da física do movimento de um navio e gera o comportamento do navio;

2. Módulo Radar: apresenta os alvos (outros navios e aeronaves) e o contorno da linha de terra para apoiar a navegação;

3. Módulo Instrumentos: mostra uma série de instrumentos virtuais montando uma console de equipamentos para a manobra do navio;

4. Módulo Porta Serial: representa equipamentos cuja interface de recepção é uma porta serial;

5. Módulo do instrutor: gera os cenários de um treinamento e coleta as informações para posterior avaliação da manobra efetuada pelos alunos;

Cada um destes clientes possui a sua lista de assinaturas e publicações, com frequência variada de publicação, sendo que, a frequência mais alta é de 30Hz.

O simulador a ser integrado com o simulador de manobras anteriormente descrito é utilizado em jogos construtivos e apoia o planejamento e execução de uma operação, que inclui um ou mais navios. Este segundo simulador não possui na sua arquitetura um *broker* de Pub/Sub, mas ele já gera eventos que são armazenados num banco de dados e também pode consumir e apresentar estes eventos, provenientes deste mesmo banco de dados para apresentá-los numa interface georeferenciada. Estes eventos são estruturados num arquivo em XML e que obedece a um esquema (XSD) para permitir a troca entre os simuladores, deste mesmo tipo, empregados nas demais forças (Exército e Força Aérea). A integração proposta entre o simulador de manobras e o simulador de jogos construtivos visa possibilitar que, um ou mais

navios do simulador de manobras sejam utilizados num cenário do jogo. Adicionalmente, os navios utilizados no simulador de jogos devem ser apresentados no cenário virtual do simulador de manobras.

No caso desta integração, decidiu-se que a configuração objetivo do *broker* deverá ser capaz de receber e enviar mensagens no formato XML já utilizado pelo simulador de jogos. Esta capacidade será obtida por meio de uma interface baseada em *web services* e com uma especificação de interfaces definida num WSDL. Originalmente, o *broker* só possui interfaces para troca de mensagens no protocolo de serialização de objetos Java, seja por meio de uma conexão *Socket* ou por Java *Remote Method Invocation* (RMI).

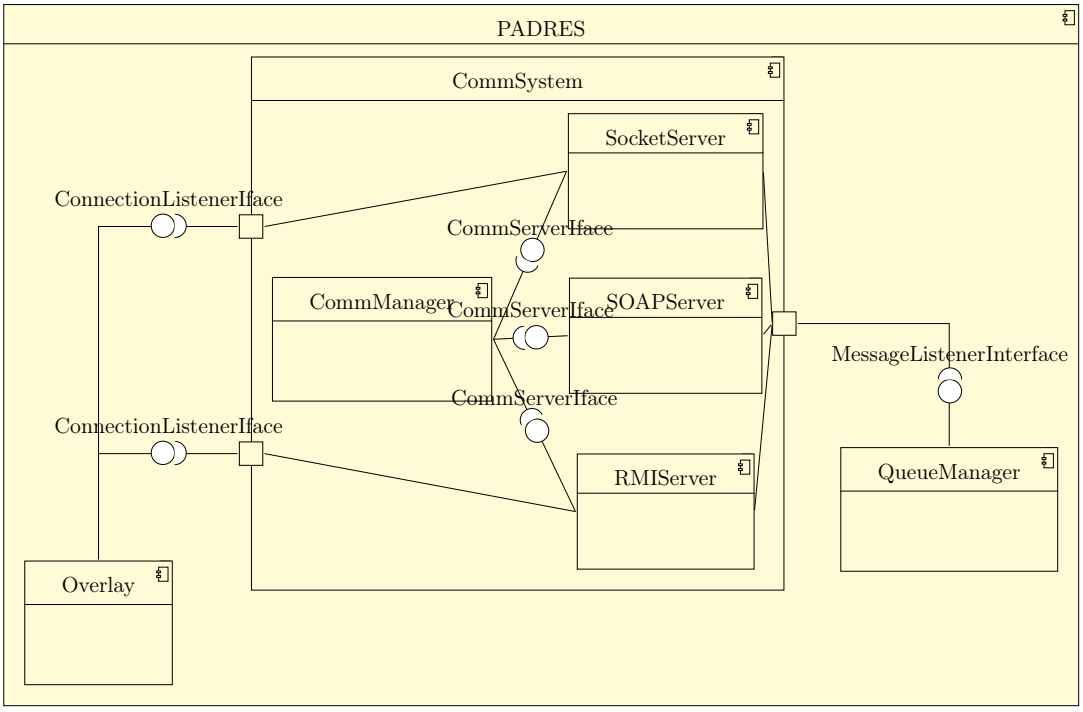


Figura 5.5: Trecho da Configuração Objetivo do Componente Composto `CommSystem`.

A parte que deverá ser alterada para obter a configuração objetivo está ilustrada na Figura 5.5. Nela, o componente composto `CommSystem` receberá um novo subcomponente composto denominado `SOAPServer`, ilustrado na Figura 5.6.

Este componente permitirá receber e enviar anúncios, publicações e as notificações correspondentes às assinaturas no protocolo SOAP. Na nova configuração, as mensagens neste protocolo serão recebidas numa outra porta de rede, pois a arquitetura do *broker* prevê uma lista de protocolos. Depois da mensagem recebida, ela é traduzida para o formato interno previsto na gramática das mensagens e segue o seu caminho normal. Como este é um componente novo na configuração, optou-se por colocar o componente `Logger` que fará o *log* necessário.

Para o componente `SOAPServer`, não foi provida a interface

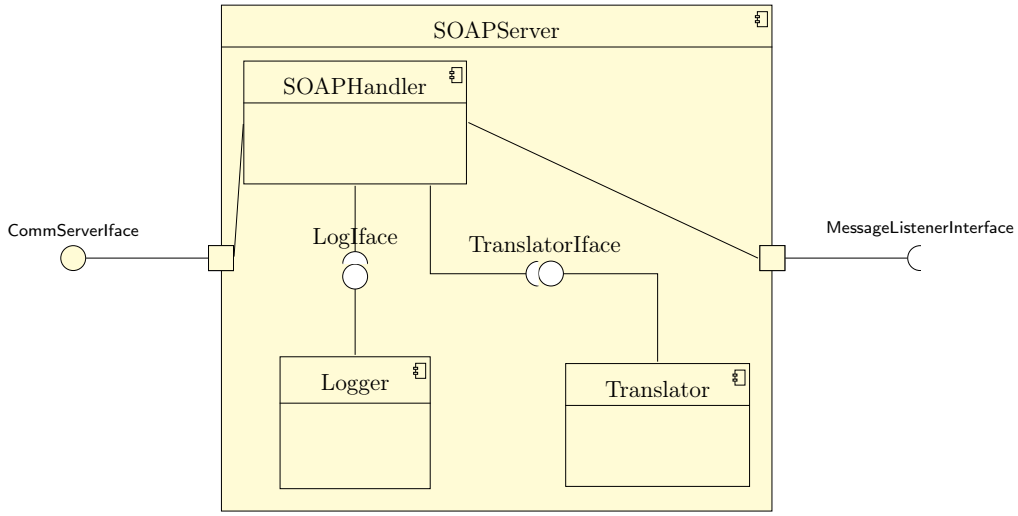


Figura 5.6: Diagrama do componente `SOAPServer`.

`ConnectionListenerIface` pois esta interface está relacionada a conexão entre *broker* e não é necessário que eles efetuem comunicação neste protocolo também, além do *Socket* e *RMI*.

A iniciação do *broker* PADRES e a reconfiguração anteriormente descrita foram especificadas no Anexo C. Neste anexo, há o problema de planejamento que representa as duas situações, o plano gerado e a respectiva tradução na DSL do FRACTAL para permitir a sua execução.

5.4 Considerações

A primeira aplicação, o servidor HTTP Comanche, serviu como ponto de partida para o emprego do planejador na geração das reconfigurações. Em seguida, a refatoração do *broker* PADRES acrescentou alguns aspectos relevantes para a modelagem do domínio de planejamento e posterior execução.

Na iniciação de um componente, identificou-se a necessidade de que houvesse a chamada de um outro componente, para permitir que o componente que chama atingisse o estado *iniciado*. Este tipo de situação também ocorre na especificação SCA (OASIS, 2011), onde há um conjunto de anotações que indicam o método que será chamado pelo contêiner de injeção de dependências na iniciação de um componente.

Um outro aspecto que teve que ser tratado foi o comportamento da membrana num componente composto em relação ao ciclo de vida. A semântica natural é que o estado dos subcomponentes deve ser igual ao estado do seu supercomponente direto. Esta semântica se reflete no comportamento do controlador do ciclo de vida de um componente composto, que irá parar e iniciar todos os seus subcomponentes toda ver que ele for parado e iniciado respectivamente. Entretanto, como foi descrito na

Seção 4.1, se não for obedecida uma ordem isso pode causar um travamento.

Em relação à iniciação dos componentes, o próprio domínio se encarrega de fazê-lo de baixo para cima na hierarquia de componentes e então, um componente composto ao ser iniciado, já terá os seus subcomponentes iniciados. Para a parada dos componentes, a solução utilizada foi alterar o comportamento do controlador do ciclo de vida a fim de que os subcomponentes não tenham o seu estado alterado como consequência da alteração no estado do supercomponente. Isto foi possível pois o FRACTAL permite que os controladores (*controllers*) possam ser gerados pelo programador.

Capítulo 6

Conclusões

Este trabalho de pesquisa tratou sobre o tema de reconfiguração dinâmica da arquitetura de sistemas baseados em componentes. Neste contexto, o trabalho mostrou a necessidade da geração de uma reconfiguração ser realizada no momento em que se precisa dela, e não previamente.

Após um levantamento das soluções de reconfiguração existentes para o mesmo problema desta pesquisa, foi identificado que as soluções não podiam ser empregadas em modelos de componentes mais gerais que são objeto desta pesquisa, nem nas aplicações de interesse, como os sistemas de apoio ao comando e controle. Dentre as razões para isso, destacam-se: o uso de modelos de componentes altamente especializados o que dificulta o seu uso para modelos mais gerais; adoção de especificações de um modelo de componentes sem considerar uma implementação concreta; não tratar dos aspectos relacionados a colocação num estado seguro para reconfigurar e, posteriormente, reiniciar a aplicação, o que compromete o funcionamento da aplicação; e a possibilidade de parada de um grande número de componentes na aplicação não envolvidos diretamente numa reconfiguração, ferindo o requisito dinâmico do ajuste arquitetural.

Para resolver a questão de pesquisa, adotou-se a visão de agente inteligente, da área de Inteligência Artificial, e o respectivo agente baseado em objetivos, materializado pela técnica de planejamento automatizado. Como plataforma escolhida para a implementação das aplicações de interesse foi escolhido o modelo de componentes FRACTAL, e a sua implementação em Java, denominada JULIA. Sua escolha se deve à maturidade da plataforma, ser um modelo de componentes geral, possuir capacidade reflexiva e ser utilizado em projetos na indústria e amplamente empregado em pesquisas na academia. Escolhida a plataforma, produziu-se o respectivo domínio de planejamento para ser empregado pelo planejador JSHOP2 na geração dos planos de reconfiguração. Para garantir que os planos gerados permitissem uma reconfiguração controlada e confiável, o domínio foi validado perante o modelo de componentes representado e aos aspectos de reconfiguração dinâmica.

Por fim, para se empregar a solução proposta, foram utilizados dois sistemas de software baseados em componentes: o servidor de HTTP *Comanche* e o *broker* de *Publish/Subscribe* **PADRES**. O *Comanche* serviu para mostrar o impacto de diferentes reconfigurações na geração dos respectivos planos. Já o **PADRES**, é uma aplicação voltada para a integração de sistemas, conectando-os por meio de um barramento de comunicação orientado a mensagens, que recebe diferentes tipos de mensagens e as envia apenas aos sistemas interessados em partes do conteúdo das mensagens ou na combinação destas partes. Este *broker* foi refatorado e implementado como um sistema baseado em componentes, utilizando-se o FRACTAL e aplicado na integração de sistemas de simulação para treinamento já em produção.

6.1 Contribuições

Uma das principais motivações para este trabalho foi a falta de estudos sobre o tipo de reconfiguração dinâmica descrito na seção anterior, voltado para aplicações computacionais de uso comum e cujo desenvolvimento é baseado em modelos de componentes gerais e reflexivos. Este trabalho visa suprir essa deficiência por meio das seguintes contribuições principais:

- Desenvolvimento de um modelo (domínio de planejamento) que representa a capacidade reflexiva de um modelo de componentes para emprego na geração da reconfiguração dinâmica.
- Validação do modelo desenvolvido em relação à capacidade reflexiva e a aspectos de reconfiguração dinâmica.
- Emprego da abordagem em apoio à reconfigurações controladas e confiáveis num sistema baseado em componentes.

Secundariamente, a abordagem desenvolvida pode ser empregada como um "banco de ensaios" (*test bed*) para reconfigurações. Com isso, pode-se observar se há uma reconfiguração possível, qual o impacto dela sobre a aplicação como a quantidade de componentes que irão parar e verificar se há diferentes maneiras de se reconfigurar uma aplicação.

Ao longo do desenvolvimento desta pesquisa, a proposta foi discutida no Simpósio de Doutorado em Componentes e Arquitetura em DI BENEDITTO (2012). Em seguida, um primeiro desenvolvimento da abordagem voltada para o modelo de componentes FRASCATI (SEINTURIER *et al.*, 2012), foi apresentada em DI BENEDITTO e WERNER (2012). Por fim, a abordagem completa desta tese foi apresentada em DI BENEDITTO e WERNER (2014).

6.2 Limitações

A partir da abordagem desenvolvida nesta pesquisa, um caminho natural seria empregá-la em outros modelos de componentes. Entretanto, a construção do modelo (domínio de planejamento) e a sua posterior validação são trabalhos manuais. Na construção do modelo não se visualiza outra forma, pois não há um formalismo único para a representação da estrutura de um modelo de componentes nem da sua capacidade reflexiva. Entretanto, quanto à validação deste modelo construído, há propostas na literatura, como a de RAIMONDI *et al.* (2009), que pretendem facilitar a geração de testes empregando o critério *Modified Condition/Decision Coverage* (MC/DC) de um domínio de planejamento descrito em *Planning Domain Definition Language* (PDDL) MCDERMOTT *et al.* (1998). Caberia agora, propor uma reescrita do domínio nesta linguagem para se tentar aplicar ferramentas deste tipo e melhorar a qualidade dos testes de validação do domínio.

Uma estrutura mais confiável para a geração e execução de um plano de reconfiguração, deveria ser realizada de maneira *online*, onde o planejador observa o ambiente a cada ação executada e efetua um replanejamento, caso seja necessário. Neste pesquisa, adotou-se a maneira *offline*, onde o planejador recebe o estado inicial e gera o respectivo plano olhar para o ambiente.

Um planejador pode empregar um algoritmo que é completo. Em outras palavras, se há um plano capaz de levar um estado inicial até um estado objetivo, então o planejador encontrará este plano. Considerando que há um plano, mas ele não foi gerado pelo planejador, isto pode significar que:

- não foi encontrado no domínio uma ou mais ações que efetuariam uma transição de estado necessária; ou
- o problema de planejamento não permite ao planejador gerar o plano.

Para o primeiro caso, seria interessante que houvesse algum mecanismo capaz de indicar o que "está faltando" para ajudar ao engenheiro a entender a causa de inexistência de um plano e possivelmente aperfeiçoar o domínio de planejamento.

Já no segundo, pode-se empregar uma etapa de verificação do problema de planejamento, onde uma série de invariantes seriam verificados, a fim de identificar alguma impropriedade na descrição do problema. Este segundo caso foi observado frequentemente durante esta pesquisa, onde alguns problemas não geravam um plano. Quando isto ocorria, e a causa não era do domínio, estas situações eram decorrentes de problemas de planejamento incompletos ou inconsistentes, como componentes simples contendo componentes compostos ou a ausência de enlaces em interfaces obrigatórias. Uma maneira de se realizar isso seria usar a mesma estrutura de busca

do planejador HTN empregado e aplicar uma série de métodos que, por meio das precondições de cada decomposição, verificariam se as invariantes são aplicáveis.

Quando mais de um plano é gerado, seria bom poder avaliá-los e ordená-los segundo algum critério, permitindo-se saber qual o melhor plano dentro do critério estabelecido. Alguns valores poderiam ser atribuídos a algumas ações de reconfiguração, como a questão do tempo utilizada por ARSHAD *et al.* (2007), permitindo-se obter um plano que teria o menor tempo de execução. Estes valores poderiam ser obtidos da própria aplicação, coletando-se os dados necessários para gerar estes valores.

6.3 Trabalhos Futuros

Além das limitações descritas anteriormente, que podem gerar outros trabalhos, seria pertinente aplicar a abordagem para outros modelos de componentes com capacidade reflexiva como o FraSCAti e o iPOJO. No caso do iPOJO, num trabalho conjunto a esta pesquisa publicado em FONSECA *et al.* (2012), ele foi ampliado para permitir o apoio a ações de reconfiguração dinâmica provenientes de um plano de reconfiguração arquitetural. Cabe agora gerar o respectivo domínio de planejamento, a fim de poder efetuar as reconfigurações geradas.

A geração de outros domínios de planejamento para diferentes modelos de componentes poderia aumentar a capacidade de reconfiguração em algumas aplicações. Por exemplo, o FraSCAti permite desenvolver uma aplicação por meio de componentes de software implementados em diferentes modelos de componentes. Entretanto, para o FraSCAti este componente é uma caixa preta do ponto de vista da sua arquitetura. Assim, a existência de mais domínios de planejamento possibilitaria gerar planos para reconfigurar a arquitetura destes outros componentes.

Seria pertinente integrar o mecanismo desta pesquisa com propostas que tratem da decisão sobre a configuração objetivo, como as citadas por GOLDSBY *et al.* (2008) e LIASKOS *et al.* (2010), onde um modelo de objetivos (*goal model*) é empregado para se decidir a configuração de software que atenderá aos novos requisitos.

Referências Bibliográficas

KEPHART, J. O., CHESS, D. M. "The Vision of Autonomic Computing", *Computer*, v. 36, pp. 41–50, January 2003. ISSN: 0018-9162. doi: http://dx.doi.org/10.1109/MC.2003.1160055. Disponível em: <`http://dx.doi.org/10.1109/MC.2003.1160055`>.

NAU, D., GHALLAB, M., TRAVERSO, P. *Automated Planning: Theory & Practice*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608567.

KRAMER, J., MAGEE, J. "The evolving philosophers problem: dynamic change management", *Software Engineering, IEEE Transactions on*, v. 16, n. 11, pp. 1293 –1306, nov 1990. ISSN: 0098-5589. doi: 10.1109/32.60317.

SYKES, D. *Autonomous Architectural Assembly And Adaptation*. PhD in computing, Imperial College of Science, Technology and Medicine - Department of Computing, February 2010. Disponível em: <`http://www.doc.ic.ac.uk/~das05/phd.pdf`>.

TAJALLI, H., GARCIA, J., EDWARDS, G., MEDVIDOVIC, N. "PLASMA: a plan-based layered architecture for software model-driven adaptation". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pp. 467–476, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0116-9. doi: http://doi.acm.org/10.1145/1858996.1859092.

BOYER, F., GRUBER, O., POUS, D. "Robust Reconfigurations of Component Assemblies". In: *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 13–22, Piscataway, NJ, USA, 2013. IEEE Press. ISBN: 978-1-4673-3076-3. Disponível em: <`http://dl.acm.org/citation.cfm?id=2486788.2486791`>.

LÉGER, M. *Fiabilité des reconfigurations dynamiques dans les architectures à composants.* Doctorat informatique temps réel, robotique et automatique,

Ecole des mines de Nantes, Paristech, May 2009. Disponível em: <http://bib.rilk.com/5308>.

OREIZY, P., MEDVIDOVIC, N., TAYLOR, R. N. "Architecture-based runtime software evolution". In: *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pp. 177–186, Washington, DC, USA, 1998. IEEE Computer Society. ISBN: 0-8186-8368-6. Disponível em: <http://dl.acm.org/citation.cfm?id=302163.302181>.

CHENG, B. H. C., DE LEMOS, R., GIESE, H., et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap." In: Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (Eds.), *Software Engineering for Self-Adaptive Systems*, v. 5525, *Lecture Notes in Computer Science*, pp. 1–26. Springer, 2009. ISBN: 978-3-642-02160-2. Disponível em: <http://dblp.uni-trier.de/db/conf/dagstuhl/adaptive2009.html>.

GARLAN, D., CHENG, S.-W., HUANG, A.-C., SCHMERL, B., STEENKISTE, P. "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure", *Computer*, v. 37, pp. 46–54, 2004. ISSN: 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2004.175.

GEORGAS, J. C., TAYLOR, R. N. "Policy-based Self-adaptive Architectures: A Feasibility Study in the Robotics Domain". In: *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pp. 105–112, New York, NY, USA, 2008. ACM. ISBN: 978-1-60558-037-1. doi: 10.1145/1370018.1370038. Disponível em: <http://doi.acm.org/10.1145/1370018.1370038>.

CHENG, S.-W., GARLAN, D. "Stitch: A language for architecture-based self-adaptation", *Journal of Systems and Software*, v. 85, n. 12, pp. 2860 – 2875, 2012. ISSN: 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2012.02.060. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121212000714>. Self-Adaptive Systems.

HUBER, N., VAN HOORN, A., KOZIOLEK, A., BROSIG, F., KOUNEV, S. "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments", *Service Oriented Computing and Applications*, v. 8, n. 1, pp. 73–89, 2014. ISSN: 1863-2386. doi: 10.1007/s11761-013-0144-4. Disponível em: <http://dx.doi.org/10.1007/s11761-013-0144-4>.

SALEHIE, M., TAHVILDARI, L. "Self-Adaptive Software: Landscape and Research Challenges", *ACM Trans. Auton. Adapt. Syst.*, v. 4, n. 2, pp. 1–42, 2009. ISSN: 1556-4665. doi: http://doi.acm.org/10.1145/1516533.1516538.

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMBIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D. S., WOLF, A. L. "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, v. 14, pp. 54–62, May 1999. ISSN: 1541-1672.

KRAMER, J., MAGEE, J. "Self-Managed Systems: an Architectural Challenge". In: *2007 Future of Software Engineering*, FOSE '07, pp. 259–268, Washington, DC, USA, 2007. IEEE Computer Society. ISBN: 0-7695-2829-5.

MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., CHENG, B. H. C. "Composing Adaptive Software", *Computer*, v. 37, pp. 56–64, July 2004. ISSN: 0018-9162. doi: http://dx.doi.org/10.1109/MC.2004.48. Disponível em: <http://dx.doi.org/10.1109/MC.2004.48>.

SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. segunda edição ed. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

COUNCILL, B., HEINEMAN, G. T. "Definition of a Software Component and Its Elements". cap. 1, pp. 5–19, Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., June 2001. ISBN: 0-201-70485-4. Disponível em: <http://dl.acm.org/citation.cfm?id=379381.379438>.

CRNKOVIC, I., SENTILLES, S., VULGARAKIS, A., CHAUDRON, M. "A Classification Framework for Software Component Models", *IEEE Transaction of Software Engineering*, v. 37, n. 5, pp. 593–615, October 2011. Disponível em: <http://www.es.mdh.se/publications/1784->.

BRUNETON, E., COUPAYE, T., LECLERCQ, M., QUÉMA, V., STEFANI, J.-B. "The FRACTAL component model and its support in Java", *Softw. Pract. Exper.*, v. 36, pp. 1257–1284, September 2006. ISSN: 0038-0644. doi: 10.1002/spe.v36:11/12. Disponível em: <http://dl.acm.org/citation.cfm?id=1152333.1152345>.

ESCOFFIER, C., HALL, R., LALANDA, P. "iPOJO: an Extensible Service-Oriented Component Framework". In: *Services Computing, 2007. SCC*

2007. IEEE International Conference on, pp. 474–481, 2007. doi: 10.1109/SCC.2007.74.

COULSON, G., BLAIR, G., GRACE, P., TAIANI, F., JOOLIA, A., LEE, K., UEYAMA, J., SIVAHARAN, T. "A Generic Component Model for Building Systems Software", *ACM Trans. Comput. Syst.*, v. 26, n. 1, pp. 1:1–1:42, mar. 2008. ISSN: 0734-2071. doi: 10.1145/1328671.1328672. Disponível em: <http://doi.acm.org/10.1145/1328671.1328672>.

ADAMS, N., FIELD, M., GELENBE, E., HAND, D., JENNINGS, N., LESLIE, D., NICHOLSON, D., RAMCHURN, S., ROGERS, A. "The Aladdin Project: Intelligent Agents for Disaster Management". In: *Proceedings of the EURON/IARP International Workshop on Robotics for Risky Interventions and Surveillance of the Environment*, Benicassim, Spain, January 2008.

SYKES, D., HEAVEN, W., MAGEE, J., KRAMER, J. "From goals to components: a combined approach to self-management". In: *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pp. 1–8, New York, NY, USA, 2008. ACM. ISBN: 978-1-60558-037-1. doi: http://doi.acm.org/10.1145/1370018.1370020. Disponível em: <http://doi.acm.org/10.1145/1370018.1370020>.

RUSSELL, S. J., NORVIG, P. *Artificial Intelligence - A Modern Approach*. Pearson Education, 2010. ISBN: 978-0-13-207148-2.

GEFFNER, H. "The Model-Based Approach to Autonomous Behavior: A Personal View". In: Fox, M., Poole, D. (Eds.), *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence, AAAI, Atlanta, Georgia, USA*. AAAI Press, July 2010.

DOWLING, J., CAHILL, V. "Self-managed decentralised systems using K-components and collaborative reinforcement learning". In: *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, WOSS '04, pp. 39–43, New York, NY, USA, 2004. ACM. ISBN: 1-58113-989-6. doi: http://doi.acm.org/10.1145/1075405.1075413. Disponível em: <http://doi.acm.org/10.1145/1075405.1075413>.

ROTHENBERG, J. "The Nature of Modeling". In: Widman, L. E., Loparo, K. A., Nielsen, N. R. (Eds.), *Artificial Intelligence, Simulation and Modeling*, pp. 75–92. John Wiley & Sons, 1989.

MÜLLER, H., PEZZÈ, M., SHAW, M. "Visibility of control in adaptive systems". In: *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ULSSIS '08, pp. 23–26, New York, NY, USA, 2008. ACM. ISBN: 978-1-60558-026-5. doi: http://doi.acm.org/10.1145/1370700.1370707. Disponível em: <http://doi.acm.org/10.1145/1370700.1370707>.

DOBSON, S., DENAZIS, S., FERNÁNDEZ, A., GAÏTI, D., GELENBE, E., MASSACCI, F., NIXON, P., SAFFRE, F., SCHMIDT, N., ZAMBONELLI, F. "A Survey of Autonomic Communications", *ACM Trans. Auton. Adapt. Syst.*, v. 1, pp. 223–259, December 2006. ISSN: 1556-4665. doi: http://doi.acm.org/10.1145/1186778.1186782. Disponível em: <http://doi.acm.org/10.1145/1186778.1186782>.

DEAN, T. L., WELLMAN, M. P. *Planning and control*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 1991. ISBN: 1-55860-209-7.

ASTROM, K. J., MURRAY, R. M. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton, NJ, USA, Princeton University Press, 2008. ISBN: 0691135762, 9780691135762.

OGATA, K. *Modern Control Engineering*. Prentice Hall, 2010. ISBN: 9780136156734.

DIAO, Y., HELLERSTEIN, J., PAREKH, S., GRIFFITH, R., KAISER, G., PHUNG, D. "A control theory foundation for self-managing computing systems", *Selected Areas in Communications, IEEE Journal on*, v. 23, n. 12, pp. 2213–2222, Dec 2005. ISSN: 0733-8716. doi: 10.1109/JSAC.2005.857206.

WELD, D. S. "Recent Advances in AI Planning", *AI MAGAZINE*, v. 20, pp. 93–123, 1999.

ARENTOFT, M. M., FUCHS, J. J., PARROD, Y., GASQUET, A., STADER, J., STOKES, I., VADON, H. "OPTIMUM-AIV: A Planning and Scheduling System for Spacecraft AIV", *Future Gener. Comput. Syst.*, v. 7, n. 4, pp. 403–412, maio 1992. ISSN: 0167-739X. doi: 10.1016/0167-739X(92)90055-G. Disponível em: <http://dx.doi.org/10.1016/0167-739X(92)90055-G>.

SIRIN, E., PARSIA, B., WU, D., HENDLER, J., NAU, D. "HTN planning for Web Service composition using SHOP2", *Web Semant.*, v. 1, pp. 377–396, October 2004. ISSN: 1570-8268.

KELLY, J.-P., BOTEA, A., KOENIG, S. "Planning with hierarchical task networks in video games". In: *Proceedings of the ICAPS-07 Workshop on Planning in Games*, 2007.

BENSALEM, S., HAVELUND, K., ORLANDINI, A. "Verification and validation meet planning and scheduling", *International Journal on Software Tools for Technology Transfer*, v. 16, n. 1, pp. 1–12, 2014. ISSN: 1433-2779. doi: 10.1007/s10009-013-0294-x. Disponível em: <http://dx.doi.org/10.1007/s10009-013-0294-x>.

TAYLOR, R., MEDVIDOVIC, N., DASHOFY, E. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009. ISBN: 0470167742.

MCILROY, M. D. "Mass Produced Software Components". In: Naur, P., Randell, B. (Eds.), *Software Engineering Concepts and Techniques*, v. Proc. NATO Conf. on Software Engineering, pp. 138–155, NATO Science Affairs Division, 1968. Disponível em: <http://www-st.inf.tu-dresden.de/Lehre/SS04/st/slides/16b-transconsistent-composition.pdf>.

LAU, K.-K., WANG, Z. "Software Component Models", *IEEE Trans. Softw. Eng.*, v. 33, pp. 709–724, October 2007. ISSN: 0098-5589. doi: 10.1109/TSE.2007.70726. Disponível em: <http://dl.acm.org/citation.cfm?id=1314033.1314048>.

WEINREICH, R., SAMETINGER, J. "Component models and component services: concepts and principles". pp. 33–48, Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-70485-4. Disponível em: <http://dl.acm.org/citation.cfm?id=379381.379482>.

LÉGER, M., LEDOUX, T., COUPAYE, T. "Reliable Dynamic Reconfigurations in a Reflective Component Model". In: *CBSE*, pp. 74–92, 2010.

ARSHAD, N., HEIMBIGNER, D., WOLF, A. L. "Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems". In: *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '03, pp. 39–46, Washington, DC, USA, 2003. IEEE Computer Society. ISBN: 0-7695-2038-3. Disponível em: <http://dl.acm.org/citation.cfm?id=951951.952269>.

ARSHAD, N., HEIMBIGNER, D., WOLF, A. L. "Deployment and dynamic reconfiguration planning for distributed software systems", *Software Quality Control*, v. 15, pp. 265–281, September 2007. ISSN: 0963-9314.

doi: 10.1007/s11219-007-9019-2. Disponível em: <http://dl.acm.org/citation.cfm?id=1286061.1286070>.

FOX, M., LONG, D. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains", *J. Artif. Intell. Res. (JAIR)*, v. 20, pp. 61–124, 2003.

SYKES, D., HEAVEN, W., MAGEE, J., KRAMER, J. "Plan-directed architectural change for autonomous systems". In: *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pp. 15–21, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-721-6. doi: http://doi.acm.org/10.1145/1292316.1292318. Disponível em: <http://doi.acm.org/10.1145/1292316.1292318>.

BERTOLI, P., CIMATTI, A., DAL LAGO, U., PISTORE, M. "Extending PDDL to nondeterminism, limited sensing and iterative conditional plans". In: *Proceedings of ICAPS'03 Workshop on PDDL*, Citeseer, 2003.

DAVID, P.-C., LEDOUX, T., LÉGER, M., COUPAYE, T. "FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures", *Annals of Telecommunications*, v. 64, pp. 45–63, 2009. ISSN: 0003-4347. 10.1007/s12243-008-0073-y.

NAU, D., NOZ AVILA, H. M., CAO, Y., LOTEM, A., MITCHELL, S. "Total-order planning with partially ordered subtasks". In: *In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pp. 425–430, 2001.

OASIS. "Service Component Architecture SCA-J Common Annotations and APIs Specification Version 1.1". http://docs.oasis-open.org/opencsa/sca-j/sca-javacaa-1.1-spec-csprd04.html, 2011. [Online; accessed 1-July-2014].

EUGSTER, P. T., FELBER, P. A., GUERRAOUI, R., KERMARREC, A.-M. "The Many Faces of Publish/Subscribe", *ACM Comput. Surv.*, v. 35, n. 2, pp. 114–131, jun. 2003. ISSN: 0360-0300. doi: 10.1145/857076.857078. Disponível em: <http://doi.acm.org/10.1145/857076.857078>.

FIDLER, E., A. JACOBSEN, H., LI, G., MANKOVSKI, S. "The padres distributed publish/subscribe system". In: *In 8th International Conference on Feature Interactions in Telecommunications and Software Systems*, pp. 12–30, 2005.

ROUVOY, R., MERLE, P. "Leveraging component-based software engineering with Fraclet", *annals of telecommunications - annales des télécommunications*, v. 64, n. 1-2, pp. 65–79, 2009. ISSN: 0003-4347. doi: 10.1007/s12243-008-0072-z. Disponível em: <http://dx.doi.org/10.1007/s12243-008-0072-z>.

FORGY, C. L. "Expert systems". IEEE Computer Society Press, cap. Rete: a fast algorithm for the many pattern/many object pattern match problem, pp. 324–341, Los Alamitos, CA, USA, 1990. ISBN: 0-8186-8904-8. Disponível em: <http://portal.acm.org/citation.cfm?id=115710.115736>.

DI BENEDITTO, M. E. M. "Automating the Reconfiguration for Self-adaptable Software". In: *Proceedings of the 17th International Doctoral Symposium on Components and Architecture*, WCOP '12, pp. 13–18, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1348-3. doi: 10.1145/2304676.2304680. Disponível em: <http://doi.acm.org/10.1145/2304676.2304680>.

SEINTURIER, L., MERLE, P., ROUVOY, R., ROMERO, D., SCHIAVONI, V., STEFANI, J.-B. "A component-based middleware platform for reconfigurable service-oriented architectures", *Softw. Pract. Exper.*, v. 42, n. 5, pp. 559–583, maio 2012. ISSN: 0038-0644. doi: 10.1002/spe.1077. Disponível em: <http://dx.doi.org/10.1002/spe.1077>.

DI BENEDITTO, M. E. M., WERNER, C. M. L. "A Declarative Approach for Software Compositional Reconfiguration". In: *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware*, ARM '12, pp. 7:1–7:6, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1609-5. doi: 10.1145/2405679.2405686. Disponível em: <http://doi.acm.org/10.1145/2405679.2405686>.

DI BENEDITTO, M. E. M., WERNER, C. M. L. "Using a Model to Generate Reconfiguration Plans at Runtime". In: *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pp. 65–74, New York, NY, USA, 2014. ACM. ISBN: 978-1-4503-2577-6. doi: 10.1145/2602458.2602466. Disponível em: <http://doi.acm.org/10.1145/2602458.2602466>.

RAIMONDI, F., PECHEUR, C., BRAT, G. "PDVer, a Tool to Verify PDDL Planning Domains". In: *In Proc. Workshop on Verification and Validation of Planning and Scheduling Systems, ICAPS*, 2009.

MCDERMOTT, D., GHALLAB, M., HOWE, A., KNOBLOCK, C., RAM, A., VELOSO, M., WELD, D., WILKINS, D. *PDDL – The Planning Domain Definition Language – Version 1.2.* Relatório técnico, Yale Center for Computational Vision and Control, 1998.

FONSECA, F. L., DI BENEDITTO, M. E. M., WERNER, C. M. L. "Um mecanismo extensível para a execução de um plano de reconfiguração arquitetural sob o framework OSGi+iPOJO". In: *III Workshop sobre Sistemas de Software Autônomos (Autosoft)/III Congresso Brasileiro de Software: Teoria e Prática*, pp. 1–10, Natal, RN, 2012.

GOLDSBY, H., SAWYER, P., BENCOMO, N., CHENG, B. H. C., HUGHES, D. "Goal-Based Modeling of Dynamically Adaptive System Requirements". In: *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pp. 36–45, March 2008. doi: 10.1109/ECBS.2008.22.

LIASKOS, S., MCILRAITH, S., SOHRABI, S., MYLOPOULOS, J. "Integrating Preferences into Goal Models for Requirements Engineering". In: *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pp. 135–144, Sept 2010. doi: 10.1109/RE.2010.26.

Apêndice A

Operadores de Planejamento do FRACTAL

Formalmente, seja $\mathcal{L}$ uma linguagem de primeira ordem na qual há um número finito de *símbolos predicativos*, A o conjunto de ações e $a \in A$ uma ação descrita por uma tripla de subconjuntos de $\mathcal{L}$, descrita como $a = (precond(a), \quad efeitos^{-}(a), efeitos^{+}(a))$. Assim as ações formam a função de transição de estados $\gamma : S \times A \rightarrow S$, tal que $\gamma(s, a) = (s - efeitos^{-}(a)) \quad \cup \quad efeitos^{+}(a)$.

Para cada operador, será descrito o conjunto de precondições (**PRECOND**) e efeitos (**EFEITOS**$^{+}$ e **EFEITOS**$^{-}$). Na sintaxe do domínio do planejador empregado, um operador é precedido do sinal de exclamação (!) e uma variável é precedida do sinal de interrogação (?). A precondição é descrita por uma expressão lógica das proposições e com os conectivos: **e** ($\wedge$), **ou** ($\vee$) e **não** ($\neg$).

o_1 - (!`instantiateNew` ?`comp`)

PRECOND : (`component` ?`comp` ?`var`) $\wedge$ $\neg$(`active` ?`comp`) $\wedge$ $\neg$(`activated` ?`comp`)

EFEITOS^{+} : (`activated` ?`comp`)

EFEITOS^{-} : $\varnothing$

o_2 - (!`instantiateOld` ?`comp`)

PRECOND : (`component` ?`comp` ?`var`) $\wedge$ (`active` ?`comp`) $\wedge$ $\neg$(`activated` ?`comp`)

EFEITOS^{+} : (`activated` ?`comp`)

EFEITOS^{-} : $\varnothing$

o_3 - (!`removeComponent` ?`comp`)

PRECOND : (`component` ?`comp` ?`var`) $\wedge$ $\neg$(`contain` ?`compFather` ?`comp`) $\wedge$
$\neg$(`bind` ?`comp` ?`iface1` ?`compServer` ?`ifaceServer`) $\wedge$
$\neg$(`bind` ?`compRef` ?`ifaceRef` ?`comp` ?`iface2`) $\wedge$

EFEITOS^{+} : $\varnothing$

EFEITOS^{-} : (active ?comp)

o_4 - (!addChildNew ?compFather ?comp)

PRECOND : (component ?comp ?var) $\wedge$ (activated ?comp) $\wedge$
(component ?compFather composite) $\wedge$ (activated ?compFather) $\wedge$
(to − contain ?compFather ?comp) $\wedge$ $\neg$(contain ?compFather ?comp) $\wedge$
$\neg$(child ?comp ?compFather) $\wedge$ $\neg$(call = ?compFather ?comp)

EFEITOS^{+} : (contained ?compFather ?comp)

EFEITOS^{-} : $\varnothing$

o_5 - (!addChildOld ?compFather ?comp)

PRECOND : (component ?comp ?var) $\wedge$ (activated ?comp) $\wedge$
(component ?compFather composite) $\wedge$ (activated ?compFather) $\wedge$
(to − contain ?compFather ?comp) $\wedge$ (contain ?compFather ?comp) $\wedge$
$\neg$(child ?comp ?compFather) $\wedge$ $\neg$(call = ?compFather ?comp)

EFEITOS^{+} : (contained ?compFather ?comp)

EFEITOS^{-} : $\varnothing$

o_6 - (!removeChild ?compFather ?comp)

PRECOND :
(component ?comp ?var) $\wedge$ (component ?compFather composite) $\wedge$
(contain ?compFather ?comp) $\wedge$ $\neg$(started ?compFather) $\wedge$
($\neg$(bind ?comp ?ifaceRef ?compServer ?ifaceServer) $\vee$
($\neg$(call = ?compFather ?compServer) $\wedge$ $\neg$(contain ?compFather ?comp)))

EFEITOS^{+} : $\varnothing$

EFEITOS^{-} : (contain ?compFather ?comp)

o_7 - (!bindNew ?compRef ?ifaceRef ?compServer ?ifaceServer)

PRECOND : (component ?compRef ?var) $\wedge$
(activated ?compRef) $\wedge$ (interface ?ifaceRef ?typeIface) $\wedge$
(component ?compServer ?var) $\wedge$ (activated ?compRef) $\wedge$
(interface ?ifaceServer ?typeIface) $\wedge$
(reference ?compRef ?ifaceRef ?cardinality ?contingency) $\wedge$
(service ?compServer ?ifaceServer) $\wedge$
$\neg$(bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
(to − bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
$\neg$(binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS^{+} : (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS^{-} : $\varnothing$

o_8 - (!bindOld ?compRef ?ifaceRef ?compServer ?ifaceServer)

PRECOND : (component ?compRef ?var) $\wedge$
(activated ?compRef) $\wedge$ (interface ?ifaceRef ?typeIface) $\wedge$
(component ?compServer ?var) $\wedge$ (activated ?compRef) $\wedge$
(interface ?ifaceServer ?typeIface) $\wedge$
(reference ?compRef ?ifaceRef ?cardinality ?contingency) $\wedge$
(service ?compServer ?ifaceServer) $\wedge$
(bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
(to − bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
$\neg$(binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS^{+} : (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS^{-} : $\varnothing$

o_9 - (!bindInternalNew ?compRef ?ifaceRef ?compServer ?ifaceServer)

PRECOND : (component ?compRef ?var) $\wedge$ (activated ?compRef) $\wedge$
(interface ?ifaceRef ?typeIface) $\wedge$
(component ?compServer ?var) $\wedge$ (activated ?compRef) $\wedge$
(interface ?ifaceServer ?typeIface) $\wedge$
(((reference ?compRef ?ifaceRef ?cardinality ?contingency) $\wedge$
(internalservice ?compServer ?ifaceServer)) $\vee$
((internalreference ?compRef ?ifaceRef) $\wedge$
(service ?compServer ?ifaceServer))) $\wedge$
$\neg$(bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
(to − bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
$\neg$(binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS^{+} : (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS^{-} : $\varnothing$

o_{10} - (!bindInternalOld ?compRef ?ifaceRef ?compServer ?ifaceServer)

PRECOND : (component ?compRef ?var) $\wedge$ (activated ?compRef) $\wedge$
(interface ?ifaceRef ?typeIface) $\wedge$
(component ?compServer ?var) $\wedge$ (activated ?compRef) $\wedge$
(interface ?ifaceServer ?typeIface) $\wedge$
(((reference ?compRef ?ifaceRef ?cardinality ?contingency) $\wedge$
(internalservice ?compServer ?ifaceServer)) $\vee$
((internalreference ?compRef ?ifaceRef) $\wedge$
(service ?compServer ?ifaceServer))) $\wedge$
(bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
(to − bind ?compRef ?ifaceRef ?compServer ?ifaceServer) $\wedge$
$\neg$(binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS$^+$: (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS$^-$: $\varnothing$

o_{11} - (!unbind ?compRef ?ifaceRef ?compServer ?ifaceServer)

PRECOND : (component ?compRef ?var) $\wedge$ $\neg$(started ?compRef) $\wedge$
(reference ?compRef ?ifaceRef ?cardinality ?contingency) $\wedge$
(component ?compServer ?var) $\wedge$ (service ?compServer ?ifaceServer) $\wedge$
(bind ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS$^+$: $\varnothing$

EFEITOS$^-$: (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)

o_{12} - (!unbindInternal ?compRef ?ifaceRef ?compServer ?ifaceServer)

PRECOND : (component ?compRef ?var1) $\wedge$ $\neg$(started ?compRef) $\wedge$
(interface ?ifaceRef ?typeIface) $\wedge$
(component ?compServer ?var2) $\wedge$ (interface ?ifaceServer ?typeIface) $\wedge$
(((reference ?compRef ?ifaceRef ?cardinality ?contingency) $\wedge$
(internalservice ?compServer ?ifaceServer)) $\vee$
((internalreference ?compRef ?ifaceRef) $\wedge$
(service ?compServer ?ifaceServer))) $\wedge$
(bind ?compRef ?ifaceRef ?compServer ?ifaceServer)

EFEITOS$^+$: $\varnothing$

EFEITOS$^-$: (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)

o_{13} - (!start ?comp)

PRECOND : (component ?comp ?var) $\wedge$ (activated ?comp) $\wedge$
($\neg$(reference ?comp ?iface ?cardinality mandatory) $\vee$
(binded ?comp ?iface ?compServer ?ifaceServer))

EFEITOS$^+$: (started ?comp)

EFEITOS$^-$: $\varnothing$

o_{14} - (!stop ?comp)

PRECOND : (component ?comp ?var) $\wedge$ (started ?comp)

EFEITOS$^+$: $\varnothing$

EFEITOS$^-$: (started ?comp)

Apêndice B

Domínio de Planejamento do Fractal

```
(defdomain domainpadres (

(:method (configure)

    _0configure
    ; PRE
    (
      (component ?comp ?var1)
    )
    ; SUBTASKS
    (
      (destroySingle)
      (reconfigureHierarchy)
      (removeBind)
      (makeBind)
      (destroyComposite)
      (startComponents)
    )
); END (:method (configure)

(:method (startComponents)

  _0startSingle
  ; PRE start an activated component
  (
    (component ?comp ?var)
    (activated ?comp)
    (not (started ?comp))
  )
  ; SUBTASKS
  (
```

```
    (startComponent ?comp)
    (startComponents)
  )

  _9startComponent
  ; PRE no more components to be started
  (
    (not (
      (component ?comp ?var)
      (activated ?comp)
      (not (started ?comp)))
    )
  )
  ; SUBTASKS
  ()
); END (:method (startComponents)

(:method (startComponent ?comp)

 _0startSingleComponentWithoutDependencies
  ; PRE start an activated single component without dependencies
  (
    (component ?comp single)
    (activated ?comp)
    (not (started ?comp))
    (not (reference ?comp ?ifaceName ?card ?contingency))
  )
  ; SUBTASKS
  (
    (!start ?comp)
  )

 _1startCompositeComponentWithoutDependencies
  ; PRE start an activated composite component without dependencies
  (
    (component ?comp composite)
    (activated ?comp)
    (not (started ?comp))
    (not (
      (component ?compChild ?var)
      (contained ?comp ?compChild)
      (not (started ?compChild)))
    )
    (not (reference ?comp ?ifaceName ?card ?contingency))
  )
  ; SUBTASKS
  (
```

```
  (!start ?comp)
)

_2startSingleComponentDepBindedStarted
; PRE start an activated single component with all dependencies binded and start
(
  (component ?comp single)
  (activated ?comp)
  (not (started ?comp))
  (not (
    (reference ?comp ?iface ?card mandatory)
    (not (binded ?comp ?iface ?compServer ?ifaceServer)))
  )
  (forall
    (?ifaceName ?compServer1 ?ifaceServer1 ?card1)
    (
      (binded ?comp ?ifaceName ?compServer1 ?ifaceServer1)
      (reference ?comp ?ifaceName ?card1 mandatory)
    )
    (started ?compServer1)
  )
)
; SUBTASKS
(
  (!start ?comp)
)

_3startCompositeComponentDepBindedStarted
; PRE start an activated composite component with all dependencies binded and st
(
  (component ?comp composite)
  (activated ?comp)
  (not (started ?comp))
  (not (
    (component ?compChild ?var)
    (contained ?comp ?compChild)
    (not (started ?compChild)))
  )
  (not (
    (reference ?comp ?iface ?card mandatory)
    (not (binded ?comp ?iface ?compServer ?ifaceServer)))
  )
  (forall
    (?ifaceName ?compServer1 ?ifaceServer1 ?card1)
    (
      (binded ?comp ?ifaceName ?compServer1 ?ifaceServer1)
      (reference ?comp ?ifaceName ?card1 mandatory)
```

```
    )
    (started ?compServer1)
  )
)
; SUBTASKS
(
  (!start ?comp)
)

_4startSingleComponentDepBindedNotStartedNotCallAtStart
; PRE start an activated single component with all dependencies binded and those
(
  (component ?comp single)
  (activated ?comp)
  (not (started ?comp))
  (not (
    (reference ?comp ?iface ?card mandatory)
    (not (binded ?comp ?iface ?compServer ?ifaceServer)))
  )
  (forall
    (?ifaceName ?compServer1 ?ifaceServer1 ?card1)
    (
      (binded ?comp ?ifaceName ?compServer1 ?ifaceServer1)
      (reference ?comp ?ifaceName ?card1 mandatory)
      (not (started ?compServer1))
    )
    (not (callatstart ?comp ?ifaceName))
  )
)
; SUBTASKS
(
  (!start ?comp)
)

_5startCompositeComponentDepBindedNotStartedNotCallAtStart
; PRE start an activated composite component with all dependencies binded and th
(
  (component ?comp composite)
  (activated ?comp)
  (not (started ?comp))
  (not (
    (component ?compChild ?var)
    (contained ?comp ?compChild)
    (not (started ?compChild)))
  )
  (not (
    (reference ?comp ?iface ?card mandatory)
```

```
      (not (binded ?comp ?iface ?compServer ?ifaceServer)))
    )
    (forall
      (?ifaceName ?compServer1 ?ifaceServer1 ?card1)
      (
        (binded ?comp ?ifaceName ?compServer1 ?ifaceServer1)
        (reference ?comp ?ifaceName ?card1 mandatory)
        (not (started ?compServer1))
      )
      (not (callatstart ?comp ?ifaceName))
    )
  )
  ; SUBTASKS
  (
    (!start ?comp)
  )
); END :method (startComponent ?comp)

(:method (destroyComposite)

  _0destroyComposite
  ; PRE composite component not present in goal configuration
  (
    (component ?comp composite)
    (active ?comp)
    (not (to-active ?comp))
  )
  ; TASK
  (
    (removeBindingTo ?comp)
    (removeBindingFrom ?comp)
    (removeComponentFromAllParent ?comp)
    (!removeComponent ?comp)
    (destroyComposite)
  )

  _9destroyComposite
  ; PRE no more composite components to be removed
  (
    (not (
      (component ?comp composite)
      (active ?comp)
      (not (to-active ?comp))
    ))
  )
  ; SUBTASKS
  ()
```

```
); END (:method (destroyComposite)

(:method (removeComponentFromAllParent ?comp)

    _0removeComponentFromAllParent
    ; PRE The component is inside a compFather
    (
      (component ?comp ?var)
      (component ?compFather composite)
      (contain ?compFather ?comp)
      (not (contained ?compFather ?comp))
    )
    ; SUBTASKS
    (
      (removeCompFromFather ?compFather ?comp)
      (removeComponentFromAllParent ?comp)
    )

    _9removeComponentFromAllParent
    ; PRE
    (
      (not (
        (component ?comp ?var)
        (component ?compFather composite)
        (contain ?compFather ?comp)
        (not (contained ?compFather ?comp)))
      )
    )
    ; SUBTASKS
    ()
); END (:method (removeComponentFromAllParent ?comp)

(:method (makeBind)

  _0makeBind
  ; PRE an activated component and a bind to be made
  (
    (component ?comp ?var)
    (activated ?comp)
    (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
  ; TASK
  (
    (makeBindFrom ?comp)
    (makeBind)
  )
```

```
  _9makeBind
  ; PRE no more bind to be made
  (
    (not (
      (component ?comp ?var)
      (activated ?comp)
      (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
      (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer)))
    )
  )
  ; TASK
  ()
); END :method (makeBind)

(:method (makeBindFrom ?comp)

  _0makeBindFrom_ToSibling
  ; PRE a bind from component to a sibling component
  (
    (component ?comp ?var)
    (activated ?comp)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compServer ?var2)
    (activated ?compServer)
    (service ?compServer ?ifaceServer)
    (component ?compFather composite)
    (activated ?compFather)
    (contained ?compFather ?comp)
    (contained ?compFather ?compServer)
    (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
  ; SUBTASKS
  (
    (reconfigureBindToSibling ?comp)
    (makeBindFrom ?comp)
  )

 _1makeBindFrom_ToParent
  ; PRE a bind from child component to a parent
  (
    (component ?comp ?var)
    (activated ?comp)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compFather composite)
    (activated ?compFather)
```

```
    (contained ?compFather ?comp)
    (internalservice ?compFather ?ifaceServer)
    (to-bind ?comp ?ifaceRef ?compFather ?ifaceServer)
    (not (binded ?comp ?ifaceRef ?compFather ?ifaceServer))
  )
  ; SUBTASKS
  (
    (reconfigureBindToParent ?comp)
    (makeBindFrom ?comp)
  )

  _2makeBindFrom_ToChild
  ; PRE a bind from parent component to child
  (
    (component ?comp composite)
    (activated ?comp)
    (internalreference ?comp ?ifaceRef)
    (component ?compChild ?var2)
    (activated ?compChild)
    (contained ?comp ?compChild)
    (service ?compChild ?ifaceServer)
    (to-bind ?comp ?ifaceRef ?compChild ?ifaceServer)
    (not (binded ?comp ?ifaceRef ?compChild ?ifaceServer))
 )
  ; SUBTASKS
  (
    (reconfigureBindToChild ?comp)
    (makeBindFrom ?comp)
  )

  _9makeBindFrom
  ; PRE no more bind to do
  (
    (not (
      (component ?comp ?var)
      (activated ?comp)
      (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
      (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer)))
    )
  )
  ; SUBTASKS
  ()

); END (:method (makeBindFrom)

(:method (reconfigureBindToChild ?comp)
```

```
_0reconfigureBindToChild_Old
; PRE a bind from parent to child component in initial and goal configuration
(
  (component ?comp composite)
  (internalreference ?comp ?ifaceRef)
  (service ?comp ?ifaceRef)
  (component ?compChild ?var2)
  (contained ?comp ?compChild)
  (service ?compChild ?ifaceServer)
  (to-bind ?comp ?ifaceRef ?compChild ?ifaceServer)
  (bind ?comp ?ifaceRef ?compChild ?ifaceServer)
  (not (binded ?comp ?ifaceRef ?compChild ?ifaceServer))
)
; SUBTASKS
(
  (!bindInternalOld ?comp ?ifaceRef ?compChild ?ifaceServer)
  (reconfigureBindToChild ?comp)
)

_1reconfigureBindToChild_New
; PRE a bind from parent component to a child
(
  (component ?comp composite)
  (internalreference ?comp ?ifaceRef)
  (service ?comp ?ifaceRef)
  (component ?compChild ?var2)
  (contained ?comp ?compChild)
  (service ?compChild ?ifaceServer)
  (to-bind ?comp ?ifaceRef ?compChild ?ifaceServer)
  (not (bind ?comp ?ifaceRef ?compChild ?ifaceServer))
  (not (binded ?comp ?ifaceRef ?compChild ?ifaceServer))
)
; SUBTASKS
(
  (!bindInternalNew ?comp ?ifaceRef ?compChild ?ifaceServer)
  (reconfigureBindToChild ?comp)
)

_9reconfigureBindToChild
; PRE no more bind
(
  (not (
    (component ?comp composite)
    (internalreference ?comp ?ifaceRef)
    (service ?comp ?ifaceRef)
    (component ?compChild ?var2)
    (contained ?comp ?compChild)
```

```
      (service ?compChild ?ifaceServer)
      (to-bind ?comp ?ifaceRef ?compChild ?ifaceServer)
      (not (binded ?comp ?ifaceRef ?compChild ?ifaceServer)))
    )
  )
  ; SUBTASKS
  ()

); END (:method (reconfigureBindToChild)

(:method (reconfigureBindToParent ?comp)

  _0reconfigureBindToParent_Old
  ; PRE a bind from single component to a parent in initial and goal configuratio
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compFather composite)
    (contained ?compFather ?comp)
    (internalservice ?compFather ?ifaceServer)
    (reference ?compFather ?ifaceServer ?cardinality ?contingency)
    (to-bind ?comp ?ifaceRef ?compFather ?ifaceServer)
    (bind ?comp ?ifaceRef ?compFather ?ifaceServer)
    (not (binded ?comp ?ifaceRef ?compFather ?ifaceServer))
  )
  ; SUBTASKS
  (
    (!bindInternalOld ?comp ?ifaceRef ?compFather ?ifaceServer)
    (reconfigureBindToParent ?comp)
  )

  _1reconfigureBindToParent_New
  ; PRE a bind from single component to a parent
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compFather composite)
    (contained ?compFather ?comp)
    (internalservice ?compFather ?ifaceServer)
    (reference ?compFather ?ifaceServer ?cardinality ?contingency)
    (to-bind ?comp ?ifaceRef ?compFather ?ifaceServer)
    (not (bind ?comp ?ifaceRef ?compFather ?ifaceServer))
    (not (binded ?comp ?ifaceRef ?compFather ?ifaceServer))
  )
  ; TASK
  (
    (!bindInternalNew ?comp ?ifaceRef ?compFather ?ifaceServer)
```

```
    (reconfigureBindToParent ?comp)
  )

  _9reconfigureBindToParent
  ; PRE no more bind
  (
    (not (
       (component ?comp ?var)
       (reference ?comp ?ifaceRef ?cardinality ?contingency)
       (component ?compFather composite)
       (contained ?compFather ?comp)
       (internalservice ?compFather ?ifaceServer)
       (reference ?compFather ?ifaceServer ?cardinality ?contingency)
       (to-bind ?comp ?ifaceRef ?compFather ?ifaceServer)
       (not (binded ?comp ?ifaceRef ?compFather ?ifaceServer)))
      )
  )
  ; SUBTASKS
  ()

); END (:method (reconfigureBindToParent)

(:method (reconfigureBindToSibling ?comp)

  _0reconfigureBindToSibling_Old
  ; PRE a bind from component to a sibling in initial and goal configuration
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compServer ?var2)
    (service ?compServer ?ifaceServer)
    (component ?compFather composite)
    (contained ?compFather ?comp)
    (contained ?compFather ?compServer)
    (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
  ; SUBTASKS
  (
    (!bindOld ?comp ?ifaceRef ?compServer ?ifaceServer)
    (reconfigureBindToSibling ?comp)
  )

  _1reconfigureBindToSibling_New
  ; PRE a bind from single component to a sibling
  (
```

```
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compServer ?var2)
    (service ?compServer ?ifaceServer)
    (component ?compFather composite)
    (contained ?compFather ?comp)
    (contained ?compFather ?compServer)
    (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (bind ?comp ?ifaceRef ?compServer ?ifaceServer))
    (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
  ; SUBTASKS
  (
    (!bindNew ?comp ?ifaceRef ?compServer ?ifaceServer)
    (reconfigureBindToSibling ?comp)
  )

  _9reconfigureBindToSibling
  ; PRE no more bind
  (
    (not (
      (component ?comp ?var)
      (reference ?comp ?ifaceRef ?cardinality ?contingency)
      (component ?compServer ?var2)
      (service ?compServer ?ifaceServer)
      (component ?compFather composite)
      (contained ?compFather ?comp)
      (contained ?compFather ?compServer)
      (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)
      (not (binded ?comp ?ifaceRef ?compServer ?ifaceServer)))
    )
  )
  ; TASK
  ()
); END (:method (reconfigureBindToSibling)

(:method (removeBind)

  _0removeBind
  ; PRE a component already activated and a bind to be removed
  (
    (component ?comp ?var)
    (active ?comp)
    (activated ?comp)
    (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
```

```
  ; TASK
  (
    (removeBindingFrom ?comp)
    (removeBind)
  )

  _1removeBind
  ; PRE no more binds to be removed
  (
    (not (
      (component ?comp ?var)
      (active ?comp)
      (activated ?comp)
      (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
      (not (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)))
    )
  )
  ; TASK
  ()
); END :method (removeBind)

(:method (removeBindingFrom ?comp)

  _0removeBindingFrom_Sibling
  ; PRE A bind component client with a component sibling
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compServer ?var2)
    (service ?compServer ?ifaceServer)
    (component ?compParent composite)
    (contain ?compParent ?comp)
    (contain ?compParent ?compServer)
    (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
  ; TASK
  (
    (removeBindingComp ?comp ?ifaceRef ?compServer ?ifaceServer)
    (removeBindingFrom ?comp)
  )

  _1removeBindingFrom_ChildInternal
  ; PRE An internal binding from child component client to parent
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
```

```
  (component ?compParent composite)
  (contain ?compParent ?comp)
  (internalservice ?compParent ?ifaceServer)
  (reference ?compParent ?ifaceServer ?cardinality ?contingency)
  (bind ?comp ?ifaceRef ?compParent ?ifaceServer)
  (not (to-bind ?comp ?ifaceRef ?compParent ?ifaceServer))
)
; TASK
(
  (removeBindingInternal ?comp ?ifaceRef ?compParent ?ifaceServer)
  (removeBindingFrom ?comp)
)

_2removeBindingFrom_ParentInternal
; PRE an internal bind from parent component to child
(
  (component ?comp composite)
  (active ?comp)
  (internalreference ?comp ?ifaceRef)
  (component ?compChild ?var2)
  (active ?compChild)
  (contain ?comp ?compChild)
  (service ?compChild ?ifaceServer)
  (bind ?comp ?ifaceRef ?compChild ?ifaceServer)
  (not (to-bind ?comp ?ifaceRef ?compChild ?ifaceServer))
)
; TASK
(
  (removeBindingInternal ?comp ?ifaceRef ?compChild ?ifaceServer)
  (removeBindingFrom ?comp)
)

_9removeBindingFrom
; PRE No more bindings from component to another component: sibling, father, ch
(
  (not (
    (component ?compServer ?var)
    (service ?compServer ?ifaceServer)
    (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer)))
  )
  (not (
    (component ?compParent composite)
    (contain ?compParent ?comp)
    (internalservice ?compParent ?ifaceServer)
    (bind ?comp ?ifaceRef ?compParent ?ifaceServer)
    (not (to-bind ?comp ?ifaceRef ?compParent ?ifaceServer)))
```

```
    )
    (not (
      (component ?comp composite)
      (contain ?comp ?compChild)
      (internalreference ?comp ?ifaceRef)
      (bind ?comp ?ifaceRef ?compChild ?ifaceServer)
      (not (to-bind ?comp ?ifaceRef ?compChild ?ifaceServer)))
    )
  )
  ; TASK
  ()
); END :method (removeBindingFrom ?comp)

(:method (reconfigureHierarchy)

  _0reconfigureHierarchyHighestComposite
  ; PRE first instantiate the highest composite in initial and goal configuration
  (
    (component ?compHigh composite)
    (not (contain ?compFather ?compHigh))
    (to-active ?compHigh)
    (active ?compHigh)
    (not (activated ?compHigh))
  )
  ; TASK
  (
    (activate ?compHigh)
    (reconfigureCompositeHierarchy)
    (reconfigureSingleHierarchy)
  )
); END :method (reconfigureHierarchy)

(:method (reconfigureSingleHierarchy)

  _0reconfigureSingleHierarchy
  ; PRE all father of a single component are active
  (
    (component ?comp single)
    (to-active ?comp)
    (not (activated ?comp))
    (forall
      (?compFather)
      (to-contain ?compFather ?comp)
      (activated ?compFather)
    );all father already activated
  )
  ; TASK
```

```
  (
    (activate ?comp)
    (insertSingleInAllFather ?comp);insert single in ALL father
    (removeSingleFromAllFather ?comp)
    (reconfigureSingleHierarchy);ALL composites
  )

  _9reconfigureSingleHierarchy
  ; PRE no more single to activate
  (
    (not (
      (component ?comp single)
      (to-active ?comp)
      (not (activated ?comp))
    ))
  )
  ; TASK
  ()
); END (:method (reconfigureSingleHierarchy)

(:method (insertSingleInAllFather ?comp)

  _0insertSingleInAllFather
  ; PRE an activated father of an activated single component
  (
    (component ?comp single)
    (to-active ?comp)
    (activated ?comp)
    (component ?compFather composite)
    (to-contain ?compFather ?comp)
    (activated ?compFather); parent already activated
    (not (contained ?compFather ?comp));not yet a child
  )
  ; TASK
  (
    (addChild ?compFather ?comp)
    (insertSingleInAllFather ?comp)
  )

  _9insertSingleInAllFather
  ; PRE no more father for this single
  (
    (not (
      (component ?compFather composite)
      (to-contain ?compFather ?comp)
      (activated ?compFather)
      (not (contained ?compFather ?comp))
```

```
    ))
  )
  ; TASK
  ()
); END (:method (insertSingleInAllFather)

(:method (removeSingleFromAllFather ?comp)

  _0removeSingleFromAllFather
  ; PRE the composite father is not in the goal
  (
    (component ?comp single)
    (to-active ?comp)
    (activated ?comp)
    (component ?compFather composite)
    (contain ?compFather ?comp);condition to be removed
    (not (to-contain ?compFather ?comp))
  )
  ; TASK
  (
    (removeBindingSiblingToComp ?compFather ?comp)
    (removeBindingFatherToComp ?compFather ?comp)
    (removeBindingFromCompToFather ?compFather ?comp)
    (removeBindingFromCompToSibling ?compFather ?comp)
    (removeCompFromFather ?compFather ?comp)
    (removeSingleFromAllFather ?comp)
  )

  _9removeSingleFromAllFather
  ; PRE no more father
  (
    (not (
      (component ?compFather composite)
      (contain ?compFather ?comp)
      (not (to-contain ?compFather ?comp))
    ))
  )
  ; TASK
  ()
); END (:method (removeSingleFromAllFather)

(:method (reconfigureCompositeHierarchy)

  _0reconfigureCompositeHierarchy
  ; PRE all father of a composite component are active
  (
    (component ?comp composite)
```

```
    (to-active ?comp)
    (not (activated ?comp))
    (forall
      (?compFather)
      (to-contain ?compFather ?comp)
      (activated ?compFather)
    );all father already activated
  )
  ; TASK
  (
    (activate ?comp)
    (insertCompositeInAllFather ?comp);insert composite in ALL father
    (removeCompositeFromAllFather ?comp)
    (reconfigureCompositeHierarchy);ALL composites
  )

  _9reconfigureCompositeHierarchy
  ; PRE no more composite to activate
  (
    (not (
      (component ?comp composite)
      (to-active ?comp)
      (not (activated ?comp))
    ))
  )
  ; TASK
  ()
); END (:method (reconfigureCompositeHierarchy)

(:method (insertCompositeInAllFather ?comp)

  _0insertCompositeInAllFather
  ; PRE the father is already active and the child is already active
  (
    (component ?comp composite)
    (to-active ?comp)
    (activated ?comp)
    (component ?compFather composite)
    (to-contain ?compFather ?comp)
    (activated ?compFather); parent already activated
    (not (contained ?compFather ?comp));not yet a child
  )
  ; TASK
  (
    (addChild ?compFather ?comp)
    (insertCompositeInAllFather ?comp)
  )
```

```
  _9insertCompositeInAllFather
  ; PRE no more father for this composite
  (
    (not (
      (component ?compFather composite)
      (to-contain ?compFather ?comp)
      (not (contained ?compFather ?comp))
    ))
  )
  ; TASK
  ()
); END (:method (insertCompositeInAllFather)

(:method (removeCompositeFromAllFather ?comp)

  _0removeCompositeFromAllFather
  ; PRE the composite father is not in the goal
  (
    (component ?comp composite)
    (to-active ?comp)
    (activated ?comp)
    (component ?compFather composite)
    (contain ?compFather ?comp);condition to be removed
    (not (to-contain ?compFather ?comp))
  )
  ; TASK
  (
    (removeBindingSiblingToComp ?compFather ?comp)
    (removeBindingFatherToComp ?compFather ?comp)
    (removeBindingFromCompToFather ?compFather ?comp)
    (removeBindingFromCompToSibling ?compFather ?comp)
    (removeCompFromFather ?compFather ?comp)
    (removeCompositeFromAllFather ?comp)
  )

  _9removeCompositeFromAllFather
  ; PRE no more father
  (
    (not (
      (component ?compFather composite)
      (contain ?compFather ?comp)
      (not (to-contain ?compFather ?comp))
    ))
  )
  ; TASK
  ()
```

```
); END (:method (removeCompositeFromAllFather)

(:method (removeBindingFromCompToSibling ?compFather ?comp)

  _0removeBindingFromCompToSibling
  ; PRE A component client bind with a component sibling
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compServer ?var2)
    (service ?compServer ?ifaceServer)
    (component ?compFather composite)
    (contain ?compFather ?comp)
    (contain ?compFather ?compServer)
    (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
    (not (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer))
  )
  ; TASK
  (
    (removeBindingComp ?comp ?ifaceRef ?compServer ?ifaceServer)
    (removeBindingFromCompToSibling ?compFather ?comp)
  )

  _9removeBindingFromCompToSibling
  ; PRE No more bindings from component to another component
  (
    (component ?compFather composite)
    (component ?comp ?var)
    (contain ?compFather ?comp)
    (component ?compServer ?var2)
    (contain ?compFather ?compServer);sibling
    (service ?compServer ?ifaceServer)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (not (
      (bind ?comp ?ifaceRef ?compServer ?ifaceServer)
      (not (to-bind ?comp ?ifaceRef ?compServer ?ifaceServer))
    ))
  )
  ; TASK
  ()
); END :method (removeBindingFromCompToSibling ?compFather ?comp)

(:method (removeBindingFatherToComp ?compFather ?comp)

  _0removeBindingFatherToComp
  ; PRE remove father component reference bind in a service interface
  (
```

```
    (component ?comp ?var)
    (service ?comp ?ifaceServer)
    (component ?compFather composite)
    (contain ?compFather ?comp)
    (internalreference ?compFather ?ifaceRef)
    (bind ?compFather ?ifaceRef ?comp ?ifaceServer)
    (not (to-bind ?compFather ?ifaceRef ?comp ?ifaceServer))
  )
  ; TASK
  (
    (removeBindingInternal ?compFather ?ifaceRef ?comp ?ifaceServer)
    (removeBindingFatherToComp ?compFather ?comp)
  )

  _9removeBindingFatherToComp
  ; PRE No more father bindings
  (
    (component ?comp ?var)
    (component ?compFather composite)
    (contain ?compFather ?comp)
    (not (bind ?compFather ?ifaceRef ?comp ?ifaceServer))
  )
  ; TASK
  ()
); END (:method (removeBindingFatherToComp ?compFather ?comp)

(:method (removeBindingFromCompToFather ?compFather ?comp)

  _0removeBindingFromCompToFather
  ; PRE A child component client with parent internal binding
  (
    (component ?comp ?var)
    (reference ?comp ?ifaceRef ?cardinality ?contingency)
    (component ?compFather composite)
    (contain ?compFather ?comp)
    (internalservice ?compFather ?ifaceServer)
    (bind ?comp ?ifaceRef ?compFather ?ifaceServer)
    (not (to-bind ?comp ?ifaceRef ?compFather ?ifaceServer))
  )
  ; TASK
  (
    (removeBindingInternal ?comp ?ifaceRef ?compFather ?ifaceServer)
    (removeBindingFromCompToFather ?compFather ?comp)
  )

  _9removeBindingFromCompToFather
  ; PRE No more bindings from component to father
```

```
  (
    (component ?comp ?var)
    (component ?compFather composite)
    (contain ?compFather ?comp)
    (not (bind ?comp ?ifaceRef ?compFather ?ifaceServer))
  )
  ; TASK
  ()
); END :method (removeBindingFromCompToFather ?compFather ?comp)

(:method (removeBindingSiblingToComp ?compFather ?comp)

    _0removeBindingSiblingToComp
    ; PRE A sibling component reference bind in a service interface bind1
    (
      (component ?comp ?var)
      (service ?comp ?ifaceServer)
      (component ?compReference ?var2)
      (reference ?compReference ?ifaceRef ?cardinality ?contingency)
      (component ?compFather composite)
      (contain ?compFather ?comp)
      (contain ?compFather ?compReference)
      (bind ?compReference ?ifaceRef ?comp ?ifaceServer)
      (not (to-bind ?compReference ?ifaceRef ?comp ?ifaceServer))
    )
    ; TASK
    (
      (removeBindingComp ?compReference ?ifaceRef ?comp ?ifaceServer)
      (removeBindingSiblingToComp ?compFather ?comp)
    )

    _9removeBindingSiblingToComp
    ; PRE No more siblings bindings to remove
    (
      (component ?compFather composite)
      (component ?comp ?var)
      (contain ?compFather ?comp)
      (component ?compReference ?var2)
      (contain ?compFather ?compReference)
      (service ?comp ?ifaceServer)
      (reference ?compReference ?ifaceRef ?cardinality ?contingency)
      (not (
        (bind ?compReference ?ifaceRef ?comp ?ifaceServer)
        (not (to-bind ?compReference ?ifaceRef ?comp ?ifaceServer))
      ))
      )
    ; TASK
```

```
    ()
); END (:method (removeBindingSiblingToComp ?compFather ?comp)

(:method (removeCompFromFather ?compFather ?comp)

    _0removeCompFromFather_AllStarted
    ; PRE The component is started inside a started compFather and there is no mor
    (
      (component ?compFather composite)
      (component ?comp ?var)
      (contain ?compFather ?comp)
      (not (
        (component ?compSibling ?var1)
        (contain ?compFather ?compSibling)
        (or
          (bind ?comp ?iface ?compSibling ?iface1)
          (bind ?compSibling ?iface3 ?comp ?iface2)
        )
      ))
      (not (or (bind ?comp ?iface4 ?compFather ?iface5)
               (bind ?compFather ?iface6 ?comp ?iface7)
           )
      )
      (started ?compFather)
      (started ?comp)
    )
    ; TASK
    (
      (!stop ?comp)
      (!stop ?compFather)
      (!removeChild ?compFather ?comp)
    )

    _1removeCompFromFather_FatherStarted
    ; PRE The component is stopped inside a started compFather and there is no mor
    (
      (component ?compFather composite)
      (component ?comp ?var)
      (contain ?compFather ?comp)
      (not (
        (component ?compSibling ?var1)
        (contain ?compFather ?compSibling)
        (or
          (bind ?comp ?iface ?compSibling ?iface1)
          (bind ?compSibling ?iface3 ?comp ?iface2)
        )
      ))
```

```
  (not (or (bind ?comp ?iface4 ?compFather ?iface5)
           (bind ?compFather ?iface6 ?comp ?iface7)
       )
  )
  (started ?compFather)
  (not (started ?comp))
)
; TASK
(
  (!stop ?compFather)
  (!removeChild ?compFather ?comp)
)

_2removeCompFromFather_ChildStarted
; PRE The component is started, empty inside a stopped compFather and there is
(
  (component ?compFather composite)
  (component ?comp ?var)
  (contain ?compFather ?comp)
  (not (
    (component ?compSibling ?var1)
    (contain ?compFather ?compSibling)
    (or
      (bind ?comp ?iface ?compSibling ?iface1)
      (bind ?compSibling ?iface3 ?comp ?iface2)
    )
  ))
  (not (or (bind ?comp ?iface4 ?compFather ?iface5)
           (bind ?compFather ?iface6 ?comp ?iface7)
       )
  )
  (not (started ?compFather))
  (started ?comp)
)
; TASK
(
  (!stop ?comp)
  (!removeChild ?compFather ?comp)
)

_3removeCompFromFather_AllStopped
; PRE The component is stopped inside a stopped compFather and there is no mor
(
  (component ?compFather composite)
  (component ?comp ?var)
  (contain ?compFather ?comp)
  (not (
```

```
        (component ?compSibling ?var1)
        (contain ?compFather ?compSibling)
        (or
          (bind ?comp ?iface ?compSibling ?iface1)
          (bind ?compSibling ?iface3 ?comp ?iface2)
        )
      ))
      (not (or (bind ?comp ?iface4 ?compFather ?iface5)
               (bind ?compFather ?iface6 ?comp ?iface7)
           )
      )
      (not (started ?compFather))
      (not (started ?comp))
    )
    ; TASK
    (
      (!removeChild ?compFather ?comp)
    )
); END :method (removeCompFromFather ?compFather ?comp)

(:method (activate ?comp)

    _0activateOld
    ; PRE The component is in initial and goal configuration
    (
      (component ?comp ?var)
      (to-active ?comp)
      (active ?comp)
      (not (activated ?comp))
    )
    ; TASK
    (
      (!instantiateOld ?comp)
    )
    _1activateNew
    ; PRE The component is only in goal configuration
    (
      (component ?comp ?var)
      (to-active ?comp)
      (not (active ?comp))
      (not (activated ?comp))
    )
    ; TASK
    (
      (!instantiateNew ?comp)
    )
); END (:method (activate ?comp)
```

```
(:method (destroySingle)

  _0destroySingle
  ; PRE single component not present in goal configuration
  (
    (component ?comp single)
    (active ?comp)
    (not (to-active ?comp))
  )
  ; TASK
  (
    (removeBindingTo ?comp)
    (removeBindingFrom ?comp)
    (removeComponentFromAllParent ?comp)
    (!removeComponent ?comp)
    (destroySingle)
  )

  _9destroySingle
  ; PRE no more single components to be removed
  (
    (not (
      (component ?comp single)
      (active ?comp)
      (not (to-active ?comp))
    ))
  )
  ; TASK
  ()
); END (:method (destroySingle)

(:method (removeBindingTo ?comp)

    _0removeBindingTo_Sibling
    ; PRE A sibling component reference bind in a service interface (iFace1 type)
    (
      (component ?compReference ?var)
      (reference ?compReference ?ifaceRef ?cardinality ?contingency)
      (component ?compParent composite)
      (contain ?compParent ?comp)
      (contain ?compParent ?compReference)
      (service ?comp ?ifaceServer)
      (bind ?compReference ?ifaceRef ?comp ?ifaceServer)
      (not (to-bind ?compReference ?ifaceRef ?comp ?ifaceServer))
    )
    ; TASK
```

```
    (
      (removeBindingComp ?compReference ?ifaceRef ?comp ?ifaceServer)
      (removeBindingTo ?comp)
    )
    _1removeBindingTo_InternalParent
    ; PRE A father composite component internal bind in a promoted service interfa
    (
      (component ?compParent composite)
      (contain ?compParent ?comp)
      (service ?compParent ?iFaceParent)
      (internalreference ?compParent ?iFaceParent)
      (service ?comp ?ifaceServer)
      (bind ?compParent ?iFaceParent ?comp ?ifaceServer)
      (not (to-bind ?compParent ?iFaceParent ?comp ?ifaceServer))
    )
    ; TASK
    (
      (removeBindingInternal ?compParent ?iFaceParent ?comp ?ifaceServer)
      (removeBindingTo ?comp)
    )
    _9removeBindingTo
    ; PRE No more bindings from siblings and father to this comp
    (
      (not (
        (component ?compReference ?var)
        (reference ?compReference ?ifaceRef ?cardinality ?contingency)
        (component ?compParent composite)
        (contain ?compParent ?comp)
        (contain ?compParent ?compReference)
        (service ?comp ?ifaceServer)
        (bind ?compReference ?ifaceRef ?comp ?ifaceServer)
        (not (to-bind ?compReference ?ifaceRef ?comp ?ifaceServer)))
      )
      (not (
        (component ?compParent composite)
        (contain ?compParent ?comp)
        (service ?compParent ?iFaceParent)
        (internalreference ?compParent ?iFaceParent)
        (service ?comp ?ifaceServer)
        (bind ?compParent ?iFaceParent ?comp ?ifaceServer)
        (not (to-bind ?compParent ?iFaceParent ?comp ?ifaceServer)))
      )
    )
    ; TASK
      ()
); END :method (removeBindingTo ?comp)
```

```
(:method (removeBindingInternal ?compRef ?ifaceRef ?compServer ?ifaceServer)

  _0removeBindingInternal_RefStart
  ; PRE The component client is started
  (
    (component ?compRef ?var)
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (started ?compRef)
  )
  ; TASK
  (
    (!stop ?compRef)
    (!unbindInternal ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )

  _1removeBindingInternal_RefStop
  ; PRE The component client is stopped
  (
    (component ?compRef ?var)
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (started ?compRef))
  )
  ; TASK
  (
    (!unbindInternal ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )
); END (:method (removeBindingInternal ?compRef ?ifaceRef ?compServer ?ifaceServer

(:method (removeBindingComp ?compReference ?ifaceRef ?comp ?ifaceServer)

    _0removeBindingToComp_RefStart
    ; PRE a started component binding
    (
      (component ?compReference ?var)
      (reference ?compReference ?ifaceRef ?cardinality ?contingency)
      (component ?comp ?var2)
      (service ?comp ?ifaceServer)
      (bind ?compReference ?ifaceRef ?comp ?ifaceServer)
      (started ?compReference)
    )
    ; TASK
    (
      (!stop ?compReference)
      (!unbind ?compReference ?ifaceRef ?comp ?ifaceServer)
    )

    _1removeBindingToComp_RefStop
```

```
    ; PRE  a stopped component binding
    (
      (component ?compReference ?var)
      (reference ?compReference ?ifaceRef ?cardinality ?contingency)
      (component ?comp ?var2)
      (service ?comp ?ifaceServer)
      (bind ?compReference ?ifaceRef ?comp ?ifaceServer)
      (not (started ?compReference))
    )
    ; TASK
    (
      (!unbind ?compReference ?ifaceRef ?comp ?ifaceServer)
    )
); END :method (removeBindingComp ?compReference ?ifaceRef ?comp ?ifaceServer)

(:method (addChild ?compFather ?compChild)

    _0addChildNew
    ; PRE The father has a new child
    (
      (component ?compFather composite)
      (activated ?compFather)
      (component ?compChild ?var)
      (activated ?compChild)
      (not (contain ?compFather ?compChild))
    )
    ; TASK
    (
      (!addChildNew ?compFather ?compChild)
    )
    _1addChildOld
    ; PRE The father has an old child
    (
      (component ?compFather composite)
      (activated ?compFather)
      (component ?compChild ?var)
      (activated ?compChild)
      (contain ?compFather ?compChild)
    )
    ; TASK
    (
      (!addChildOld ?compFather ?compChild)
    )
);END (:method (addChild ?compFather ?compChild)


;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
;AXIOMS_AXIOMS
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

(:- (child ?compFather ?compChild)
        childDirectGoalConf
        (contained ?compFather ?compChild)
        childIndirectGoalConf
        ((contained ?compFather ?compAnyChild) (child ?compAnyChild ?compChild))
)

;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
;OPERATORS_OPERATORS_OPERATORS_OPERATORS
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

; instantiate a new software component
(:operator (!instantiateNew ?comp)
; PRE
  (
    (component ?comp ?var)
    (not (active ?comp))
    (not (activated ?comp))
  )
; DEL
  ()
; ADD
  (
    (activated ?comp)
  )
)

; instantiate a software component already active
(:operator (!instantiateOld ?comp)
; PRE
  (
    (component ?comp ?var)
    (active ?comp)
    (not (activated ?comp))
  )
; DEL
  ()
; ADD
  (
    (activated ?comp)
  )
)

; removes a component from the application
```

```
; the component is not inside any composite in initial configuration
; the component does not have bind from it nor bind to it
(:operator (!removeComponent ?comp)
  ; PRE
  (
    (component ?comp ?var)
    (not (
      (component ?compFather composite)
      (contain ?compFather ?comp))
    )
    (not (bind ?comp ?iface1 ?compServer ?ifaceServer))
    (not (bind ?compRef ?ifaceRef ?comp ?iface2))
  )
  ; DEL
  (
    (active ?comp)
  )
  ; ADD
  ()
)

; add a child component
(:operator (!addChildNew ?compFather ?comp)
  ; PRE
  (
    (component ?comp ?var1)
    (activated ?comp)
    (component ?compFather composite)
    (activated ?compFather)
    (to-contain ?compFather ?comp)
    (not (contain ?compFather ?comp))
    (not (call = ?compFather ?comp))

;    (not (child ?comp ?compFather));compFather is not a child of comp

  )
  ; DEL
  ()
  ; ADD
  (
    (contained ?compFather ?comp)
  )
)

; add a child component already a subcomponent
(:operator (!addChildOld ?compFather ?comp)
  ; PRE
```

```
  (
    (component ?comp ?var1)
    (activated ?comp)
    (component ?compFather composite)
    (activated ?compFather)
    (to-contain ?compFather ?comp)
    (contain ?compFather ?comp)
    (not (call = ?compFather ?comp))

;    (not (child ?comp ?compFather));compFather is not a child of comp

  )
  ; DEL
  ()
  ; ADD
  (
    (contained ?compFather ?comp)
  )
)

; removes a subcomponent
(:operator (!removeChild ?compFather ?comp)
  ; PRE
  (
    (component ?compFather composite)
    (contain ?compFather ?comp)
    (not (started ?compFather))
    (or
      (not (bind ?comp ?ifaceRef ?compServer ?ifaceServer))
      ((not (call = ?compFather ?compServer)) (not (contain ?compFather ?compServe
    )
  )
  ; DEL
  (
    (contain ?compFather ?comp)
  )
  ; ADD
  ()
)

(:operator (!bindNew ?compRef ?ifaceRef ?compServer ?ifaceServer)
; PRE bind component interface client to a component interface server
  (
    (component ?compRef ?var1)
    (activated ?compRef)
    (reference ?compRef ?ifaceRef ?cardinality ?contingency)
    (interface ?ifaceRef ?typeIface)
```

```
    (component ?compServer ?var2)
    (activated ?compServer)
    (service ?compServer ?ifaceServer)
    (interface ?ifaceServer ?typeIface)
    (not (bind ?compRef ?ifaceRef ?compServer ?ifaceServer))
    (to-bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?compRef ?ifaceRef ?compServer ?ifaceServer))
  )
  ; DEL
  ()
  ; ADD
  (
    (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )
)

(:operator (!bindOld ?compRef ?ifaceRef ?compServer ?ifaceServer)
; PRE an interface bind in the initial and goal configuration
  (
    (component ?compRef ?var1)
    (activated ?compRef)
    (reference ?compRef ?ifaceRef ?cardinality ?contingency)
    (interface ?ifaceRef ?typeIface)
    (component ?compServer ?var2)
    (activated ?compServer)
    (service ?compServer ?ifaceServer)
    (interface ?ifaceServer ?typeIface)
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (to-bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?compRef ?ifaceRef ?compServer ?ifaceServer))
  )
  ; DEL
  ()
  ; ADD
  (
    (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )
)

(:operator (!bindInternalNew ?compRef ?ifaceRef ?compServer ?ifaceServer)
; PRE an internal interface bind
  (
    (component ?compRef ?var1)
    (activated ?compRef)
    (interface ?ifaceRef ?typeIface)
    (component ?compServer ?var2)
    (activated ?compServer)
```

```
    (interface ?ifaceServer ?typeIface)
    (or
      (
        (reference ?compRef ?ifaceRef ?cardinality ?contingency)
        (internalservice ?compServer ?ifaceServer)
      )
      (
        (internalreference ?compRef ?ifaceRef)
        (service ?compServer ?ifaceServer)
      )
    )
    (not (bind ?compRef ?ifaceRef ?compServer ?ifaceServer))
    (to-bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?compRef ?ifaceRef ?compServer ?ifaceServer))
  )
  ; DEL
  ()
  ; ADD
  (
    (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )
)

(:operator (!bindInternalOld ?compRef ?ifaceRef ?compServer ?ifaceServer)
; PRE an internal interface bind in the initial and goal configuration
  (
    (component ?compRef ?var1)
    (activated ?compRef)
    (interface ?ifaceRef ?typeIface)
    (component ?compServer ?var2)
    (activated ?compServer)
    (interface ?ifaceServer ?typeIface)
    (or
      (
        (reference ?compRef ?ifaceRef ?cardinality ?contingency)
        (internalservice ?compServer ?ifaceServer)
      )
      (
        (internalreference ?compRef ?ifaceRef)
        (service ?compServer ?ifaceServer)
      )
    )
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (to-bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (binded ?compRef ?ifaceRef ?compServer ?ifaceServer))
  )
  ; DEL
```

```
  ()
  ; ADD
  (
    (binded ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )
)

; unbind a component required interface
(:operator (!unbind ?compRef ?ifaceRef ?compServer ?ifaceServer)
  ; PRE
  (
    (component ?compRef ?var1)
    (component ?compServer ?var2)
    (reference ?compRef ?ifaceRef ?card ?contingency)
    (service ?compServer ?ifaceServer)
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (started ?compRef))
  )
  ; DEL
  (
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
  )
  ; ADD
  ()
)

(:operator (!unbindInternal ?compRef ?ifaceRef ?compServer ?ifaceServer)
  ; PRE
  (
    (component ?compRef ?var1)
    (component ?compServer ?var2)
    (or
      (
        (reference ?compRef ?ifaceRef ?cardinality ?contingency)
        (internalservice ?compServer ?ifaceServer)
      )
      (
        (internalreference ?compRef ?ifaceRef)
        (service ?compServer ?ifaceServer)
      )
    )
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
    (not (started ?compRef))
  )
  ; DEL
  (
    (bind ?compRef ?ifaceRef ?compServer ?ifaceServer)
```

```
  )
  ; ADD
  ()
)

; stop a software component
(:operator (!stop ?comp)
  ; PRE
  (
    (component ?comp ?var)
    (started ?comp)
  )
  ; DEL
  (
    (started ?comp)
  )
  ; ADD
  ()
)

; start a software component
(:operator (!start ?comp)
  ; PRE all required mandatory interfaces are binded
  (
    (component ?comp ?var)
    (activated ?comp)
    (or
      (not (reference ?comp ?iface ?cardinality mandatory))
      (binded ?comp ?iface ?compServer ?ifaceServer)
    )
  )
  ; DEL
  ()
  ; ADD
  (
    (started ?comp)
  )
)

 )
)
```

Apêndice C

As Reconfigurações no *broker* PADRES

Neste anexo, serão listados os seguintes cenários que demandam uma reconfiguração. O primeiro cenário descrito na Seção C.1 é a iniciação do *broker*, onde a configuração inicial é vazia e a configuração objetivo é a aplicação completa e com seus componentes no estado iniciado. A segunda reconfiguração descrita na Seção C.2 é a inclusão de um componente composto que irá prover novas funcionalidades.

C.1 Iniciação

A iniciação é obtida considerando-se a configuração inicial vazia e a configuração objetivo a aplicação completa. Como descrito na Seção 4.1.3, a configuração inicial *vazia* deve conter, ao menos, um único componente que é o componente mais alto na hierarquia, ou seja, o predicado `(active padres)`.

A iniciação é descrita pelo seguinte problema de planejamento:

```
(defproblem padres domainpadres
 (
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; building blocks
;; the initial configuration is empty - an initialization plan
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (component padres composite)

  (component queuemanager single)
  (component inputqueuehandler single)
  (component overlay single)

  (component commsystem composite)
  (component commmanager single)
```

```
  (component socketserver single)
  (component rmiserver single)

  (component contentrouter composite)
  (component reterouter single)
  (component retematcher single)
  (component forwarder single)
  (component postprocessor single)
  (component preprocessor single)

  (component soapserver composite)
  (component logger single)
  (component translator single)
  (component soaphandler single)

  (interface messagequeueiface messagequeueiface)
  (interface queuemanageriface queuemanageriface)

  (interface commserveriface commserveriface)
  (interface messagelisteneriface messagelisteneriface)
  (interface connectionlisteneriface connectionlisteneriface)

  (interface routeriface routeriface)
  (interface matcher matcher)
  (interface preprocessoriface preprocessoriface)
  (interface postprocessoriface postprocessoriface)

  (interface overlayiface overlayiface)
  (interface forwarderiface forwarderiface)

  (interface logiface logiface)
  (interface translatoriface translatoriface)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; blocks dependency
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (reference commsystem messagelisteneriface single mandatory)
  (internalservice commsystem messagelisteneriface)

  (reference queuemanager messagequeueiface single mandatory)
  (reference inputqueuehandler queuemanageriface single mandatory)
  (reference inputqueuehandler routeriface single mandatory)

  (callatstart inputqueuehandler routeriface)

  (service queuemanager messagelisteneriface)
```

```
  (service queuemanager queuemanageriface)
  (service inputqueuehandler messagequeueiface)
  (service contentrouter routeriface)

  (reference commsystem connectionlisteneriface single mandatory)
  (internalservice commsystem connectionlisteneriface)
  (reference commmanager commserveriface collection mandatory)
  (reference socketserver messagelisteneriface single mandatory)
  (reference rmiserver messagelisteneriface single mandatory)
  (reference socketserver connectionlisteneriface single mandatory)
  (reference rmiserver connectionlisteneriface single mandatory)

  (service socketserver commserveriface)
  (service rmiserver commserveriface)

  (reference overlay routeriface single mandatory)
  (service overlay connectionlisteneriface)
  (service overlay overlayiface)

  (reference contentrouter overlayiface single mandatory)
  (internalservice contentrouter overlayiface)
  (service contentrouter routeriface)
  (internalreference contentrouter routeriface)

  (reference reterouter preprocessoriface single mandatory)
  (reference reterouter postprocessoriface single mandatory)
  (reference reterouter forwarderiface single mandatory)
  (reference reterouter matcher single mandatory)
  (service reterouter routeriface)

  (callatstart reterouter postprocessoriface)

  (reference retematcher routeriface single mandatory)
  (service retematcher matcher)

  (reference forwarder routeriface single mandatory)
  (reference forwarder overlayiface single mandatory)
  (service forwarder forwarderiface)

  (service postprocessor postprocessoriface)
  (service preprocessor preprocessoriface)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; initial state software architecture
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (active padres)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; goal state software architecture
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (to-active padres)

  (to-active queuemanager)
  (to-active inputqueuehandler)
  (to-active overlay)

  (to-active commsystem)
  (to-active commmanager)
  (to-active socketserver)
  (to-active rmiserver)

  (to-active contentrouter)
  (to-active reterouter)
  (to-active retematcher)
  (to-active forwarder)
  (to-active postprocessor)
  (to-active preprocessor)

  (to-contain padres commsystem)
  (to-contain padres queuemanager)
  (to-contain padres inputqueuehandler)
  (to-contain padres contentrouter)
  (to-contain padres overlay)

  (to-contain commsystem socketserver)
  (to-contain commsystem rmiserver)
  (to-contain commsystem commmanager)

  (to-contain contentrouter reterouter)
  (to-contain contentrouter retematcher)
  (to-contain contentrouter forwarder)
  (to-contain contentrouter postprocessor)
  (to-contain contentrouter preprocessor)

  (to-bind commsystem messagelisteneriface queuemanager messagelisteneriface)
  (to-bind commsystem connectionlisteneriface overlay connectionlisteneriface)

  (to-bind queuemanager messagequeueiface inputqueuehandler messagequeueiface)

  (to-bind inputqueuehandler queuemanageriface queuemanager queuemanageriface)
  (to-bind inputqueuehandler routeriface contentrouter routeriface)
```

```
  (to-bind commmanager commserveriface socketserver commserveriface)
  (to-bind commmanager commserveriface rmiserver commserveriface)

  (to-bind socketserver messagelisteneriface commsystem messagelisteneriface)
  (to-bind socketserver connectionlisteneriface commsystem connectionlisteneriface

  (to-bind rmiserver messagelisteneriface commsystem messagelisteneriface)
  (to-bind rmiserver connectionlisteneriface commsystem connectionlisteneriface)

  (to-bind contentrouter overlayiface overlay overlayiface)
  (to-bind overlay routeriface contentrouter routeriface)

  (to-bind contentrouter routeriface reterouter routeriface)
  (to-bind retematcher routeriface reterouter routeriface)
  (to-bind forwarder routeriface reterouter routeriface)
  (to-bind forwarder overlayiface contentrouter overlayiface)

  (to-bind reterouter matcher retematcher matcher)
  (to-bind reterouter forwarderiface forwarder forwarderiface)
  (to-bind reterouter preprocessoriface preprocessor preprocessoriface)
  (to-bind reterouter postprocessoriface postprocessor postprocessoriface)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; requirement to component
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; goals - can be a list of methods
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (
  (configure)
 )

)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Dado este problema, o planejador o recebe e busca por um plano. O plano gerado está listado a seguir. Ele é uma cópia da saída do planejador JSHOP2. No caso da iniciação, a única ação que deve ser alterada é a `(!instantiateold padres)`, pois

este componente também deve ser iniciado apesar de possuir o sufixo `Old`.

```
1 plan(s) were found:

Plan #1:
Plan cost: 62.0

(!instantiateold padres)
(!instantiatenew commsystem)
(!addchildnew padres commsystem)
(!instantiatenew contentrouter)
(!addchildnew padres contentrouter)
(!instantiatenew queuemanager)
(!addchildnew padres queuemanager)
(!instantiatenew inputqueuehandler)
(!addchildnew padres inputqueuehandler)
(!instantiatenew overlay)
(!addchildnew padres overlay)
(!instantiatenew commmanager)
(!addchildnew commsystem commmanager)
(!instantiatenew socketserver)
(!addchildnew commsystem socketserver)
(!instantiatenew rmiserver)
(!addchildnew commsystem rmiserver)
(!instantiatenew reterouter)
(!addchildnew contentrouter reterouter)
(!instantiatenew retematcher)
(!addchildnew contentrouter retematcher)
(!instantiatenew forwarder)
(!addchildnew contentrouter forwarder)
(!instantiatenew postprocessor)
(!addchildnew contentrouter postprocessor)
(!instantiatenew preprocessor)
(!addchildnew contentrouter preprocessor)
(!bindnew queuemanager messagequeueiface inputqueuehandler messagequeueiface)
(!bindnew inputqueuehandler queuemanageriface queuemanager queuemanageriface)
(!bindnew inputqueuehandler routeriface contentrouter routeriface)
(!bindnew overlay routeriface contentrouter routeriface)
(!bindnew commsystem messagelisteneriface queuemanager messagelisteneriface)
(!bindnew commsystem connectionlisteneriface overlay connectionlisteneriface)
(!bindnew commmanager commserveriface socketserver commserveriface)
(!bindnew commmanager commserveriface rmiserver commserveriface)
(!bindinternalnew socketserver messagelisteneriface commsystem messagelisteneriface
(!bindinternalnew socketserver connectionlisteneriface commsystem connectionlister
(!bindinternalnew rmiserver messagelisteneriface commsystem messagelisteneriface)
(!bindinternalnew rmiserver connectionlisteneriface commsystem connectionlisteneri
(!bindnew contentrouter overlayiface overlay overlayiface)
```

```
(!bindinternalnew contentrouter routeriface reterouter routeriface)
(!bindnew reterouter preprocessoriface preprocessor preprocessoriface)
(!bindnew reterouter postprocessoriface postprocessor postprocessoriface)
(!bindnew reterouter forwarderiface forwarder forwarderiface)
(!bindnew reterouter matcher retematcher matcher)
(!bindnew retematcher routeriface reterouter routeriface)
(!bindnew forwarder routeriface reterouter routeriface)
(!bindinternalnew forwarder overlayiface contentrouter overlayiface)
(!start queuemanager)
(!start overlay)
(!start commmanager)
(!start socketserver)
(!start rmiserver)
(!start commsystem)
(!start retematcher)
(!start forwarder)
(!start postprocessor)
(!start reterouter)
(!start preprocessor)
(!start contentrouter)
(!start inputqueuehandler)
(!start padres)
---------------------

Time Used = 0.151
```

C.2 Reconfiguração

A reconfiguração listada a seguir, se refere ao problema descrito na Seção 5.3, e irá inserir o componente composto `SOAPServer`. Este componente está descrito pelos predicados nas linha: 27, 28, 29 e 30 da listagem a seguir, e a inserção destes componentes na configuração objetivo está descrita nas linhas 224, 225, 226 e 227.

```
1
2(defproblem padres domainpadres
3 (
4;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
5;; building blocks
6;; the goal configuration is the soap server composite component
7;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
8
9  (component padres composite)
10
11  (component queuemanager single)
```

```
12  (component inputqueuehandler single)
13  (component overlay single)
14
15  (component commsystem composite)
16  (component commmanager single)
17  (component socketserver single)
18  (component rmiserver single)
19
20  (component contentrouter composite)
21  (component reterouter single)
22  (component retematcher single)
23  (component forwarder single)
24  (component postprocessor single)
25  (component preprocessor single)
26
27  (component soapserver composite)
28  (component logger single)
29  (component translator single)
30  (component soaphandler single)
31
32  (interface messagequeueiface messagequeueiface)
33  (interface queuemanageriface queuemanageriface)
34
35  (interface commserveriface commserveriface)
36  (interface messagelisteneriface messagelisteneriface)
37  (interface connectionlisteneriface connectionlisteneriface)
38
39  (interface routeriface routeriface)
40  (interface matcher matcher)
41  (interface preprocessoriface preprocessoriface)
42  (interface postprocessoriface postprocessoriface)
43
44  (interface overlayiface overlayiface)
45  (interface forwarderiface forwarderiface)
46
47  (interface logiface logiface)
48  (interface translatoriface translatoriface)
49
50;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
51;; blocks dependency
52;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
53
54  (reference commsystem messagelisteneriface single mandatory)
55  (internalservice commsystem messagelisteneriface)
56
57  (reference queuemanager messagequeueiface single mandatory)
58  (reference inputqueuehandler queuemanageriface single mandatory)
```

```
59 (reference inputqueuehandler routeriface single mandatory)
60
61 (service queuemanager messagelisteneriface)
62 (service queuemanager queuemanageriface)
63 (service inputqueuehandler messagequeueiface)
64 (service contentrouter routeriface)
65
66 (reference commsystem connectionlisteneriface single mandatory)
67 (internalservice commsystem connectionlisteneriface)
68 (reference commmanager commserveriface collection mandatory)
69 (reference socketserver messagelisteneriface single mandatory)
70 (reference rmiserver messagelisteneriface single mandatory)
71 (reference soapserver messagelisteneriface single mandatory)
72 (internalservice soapserver messagelisteneriface)
73
74 (reference socketserver connectionlisteneriface single mandatory)
75 (reference rmiserver connectionlisteneriface single mandatory)
76
77 (service socketserver commserveriface)
78 (service rmiserver commserveriface)
79 (service soapserver commserveriface)
80 (internalreference soapserver commserveriface)
81
82 (reference soaphandler messagelisteneriface single mandatory)
83 (reference soaphandler logiface single mandatory)
84 (reference soaphandler translatoriface single mandatory)
85
86 (service soaphandler commserveriface)
87
88 (reference overlay routeriface single mandatory)
89 (service overlay connectionlisteneriface)
90 (service overlay overlayiface)
91
92 (reference contentrouter overlayiface single mandatory)
93 (internalservice contentrouter overlayiface)
94 (service contentrouter routeriface)
95 (internalreference contentrouter routeriface)
96
97 (reference reterouter preprocessoriface single mandatory)
98 (reference reterouter postprocessoriface single mandatory)
99 (reference reterouter forwarderiface single mandatory)
100 (reference reterouter matcher single mandatory)
101 (service reterouter routeriface)
102
103 (reference retematcher routeriface single mandatory)
104 (service retematcher matcher)
105
```

```
106  (reference forwarder routeriface single mandatory)
107  (reference forwarder overlayiface single mandatory)
108  (service forwarder forwarderiface)
109
110  (service postprocessor postprocessoriface)
111  (service preprocessor preprocessoriface)
112
113  (service logger logiface)
114  (service translator translatoriface)
115
116;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
117;; initial state software architecture
118;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
119
120  (active padres)
121
122  (active queuemanager)
123  (active inputqueuehandler)
124  (active overlay)
125
126  (active commsystem)
127  (active commmanager)
128  (active socketserver)
129  (active rmiserver)
130
131  (active contentrouter)
132  (active reterouter)
133  (active retematcher)
134  (active forwarder)
135  (active postprocessor)
136  (active preprocessor)
137
138  (contain padres commsystem)
139  (contain padres queuemanager)
140  (contain padres inputqueuehandler)
141  (contain padres contentrouter)
142  (contain padres overlay)
143
144  (contain commsystem socketserver)
145  (contain commsystem rmiserver)
146  (contain commsystem commmanager)
147
148  (contain contentrouter reterouter)
149  (contain contentrouter retematcher)
150  (contain contentrouter forwarder)
151  (contain contentrouter postprocessor)
152  (contain contentrouter preprocessor)
```

```
153
154 (bind commsystem messagelisteneriface queuemanager messagelisteneriface)
155 (bind commsystem connectionlisteneriface overlay connectionlisteneriface)
156
157 (bind queuemanager messagequeueiface inputqueuehandler messagequeueiface)
158
159 (bind inputqueuehandler queuemanageriface queuemanager queuemanageriface)
160 (bind inputqueuehandler routeriface contentrouter routeriface)
161
162 (bind commmanager commserveriface socketserver commserveriface)
163 (bind commmanager commserveriface rmiserver commserveriface)
164
165 (bind socketserver messagelisteneriface commsystem messagelisteneriface)
166 (bind socketserver connectionlisteneriface commsystem connectionlisteneriface)
167
168 (bind rmiserver messagelisteneriface commsystem messagelisteneriface)
169 (bind rmiserver connectionlisteneriface commsystem connectionlisteneriface)
170
171 (bind contentrouter overlayiface overlay overlayiface)
172 (bind overlay routeriface contentrouter routeriface)
173
174 (bind contentrouter routeriface reterouter routeriface)
175 (bind retematcher routeriface reterouter routeriface)
176 (bind forwarder routeriface reterouter routeriface)
177 (bind forwarder overlayiface contentrouter overlayiface)
178
179 (bind reterouter matcher retematcher matcher)
180 (bind reterouter forwarderiface forwarder forwarderiface)
181 (bind reterouter preprocessoriface preprocessor preprocessoriface)
182 (bind reterouter postprocessoriface postprocessor postprocessoriface)
183
184 (started padres)
185
186 (started queuemanager)
187 (started inputqueuehandler)
188 (started overlay)
189
190 (started commsystem)
191 (started commmanager)
192 (started socketserver)
193 (started rmiserver)
194
195 (started contentrouter)
196 (started reterouter)
197 (started retematcher)
198 (started forwarder)
199 (started postprocessor)
```

```
200  (started preprocessor)
201
202;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
203;; goal state software architecture
204;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
205
206  (to-active padres)
207
208  (to-active queuemanager)
209  (to-active inputqueuehandler)
210  (to-active overlay)
211
212  (to-active commsystem)
213  (to-active commmanager)
214  (to-active socketserver)
215  (to-active rmiserver)
216
217  (to-active contentrouter)
218  (to-active reterouter)
219  (to-active retematcher)
220  (to-active forwarder)
221  (to-active postprocessor)
222  (to-active preprocessor)
223
224  (to-active soapserver)
225  (to-active soaphandler)
226  (to-active logger)
227  (to-active translator)
228
229  (to-contain padres commsystem)
230  (to-contain padres queuemanager)
231  (to-contain padres inputqueuehandler)
232  (to-contain padres contentrouter)
233  (to-contain padres overlay)
234
235  (to-contain commsystem socketserver)
236  (to-contain commsystem rmiserver)
237  (to-contain commsystem soapserver)
238  (to-contain commsystem commmanager)
239
240  (to-contain contentrouter reterouter)
241  (to-contain contentrouter retematcher)
242  (to-contain contentrouter forwarder)
243  (to-contain contentrouter postprocessor)
244  (to-contain contentrouter preprocessor)
245
246  (to-contain soapserver logger)
```

```
247  (to-contain soapserver translator)
248  (to-contain soapserver soaphandler)
249
250  (to-bind commsystem messagelisteneriface queuemanager messagelisteneriface)
251  (to-bind commsystem connectionlisteneriface overlay connectionlisteneriface)
252
253  (to-bind queuemanager messagequeueiface inputqueuehandler messagequeueiface)
254
255  (to-bind inputqueuehandler queuemanageriface queuemanager queuemanageriface)
256  (to-bind inputqueuehandler routeriface contentrouter routeriface)
257
258  (to-bind commmanager commserveriface socketserver commserveriface)
259  (to-bind commmanager commserveriface rmiserver commserveriface)
260  (to-bind commmanager commserveriface soapserver commserveriface)
261
262  (to-bind socketserver messagelisteneriface commsystem messagelisteneriface)
263  (to-bind socketserver connectionlisteneriface commsystem connectionlisteneriface
264
265  (to-bind rmiserver messagelisteneriface commsystem messagelisteneriface)
266  (to-bind rmiserver connectionlisteneriface commsystem connectionlisteneriface)
267
268  (to-bind soapserver messagelisteneriface commsystem messagelisteneriface)
269
270  (to-bind soapserver commserveriface soaphandler commserveriface)
271  (to-bind soaphandler logiface logger logiface)
272  (to-bind soaphandler translatoriface translator translatoriface)
273  (to-bind soaphandler messagelisteneriface soapserver messagelisteneriface)
274
275  (to-bind contentrouter overlayiface overlay overlayiface)
276  (to-bind overlay routeriface contentrouter routeriface)
277
278  (to-bind contentrouter routeriface reterouter routeriface)
279  (to-bind retematcher routeriface reterouter routeriface)
280  (to-bind forwarder routeriface reterouter routeriface)
281  (to-bind forwarder overlayiface contentrouter overlayiface)
282
283  (to-bind reterouter matcher retematcher matcher)
284  (to-bind reterouter forwarderiface forwarder forwarderiface)
285  (to-bind reterouter preprocessoriface preprocessor preprocessoriface)
286  (to-bind reterouter postprocessoriface postprocessor postprocessoriface)
287
288;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
289;; requirement to component
290;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
291
292;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
293
```

```
294 )
295;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
296;; goals - can be a list of methods
297;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
298
299 (
300  (configure)
301 )
302
303)
304;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
305;; end
306;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

O plano gerado para se obter esta nova configuração, já com as ações com sufixo `Old` removidas, está na listagem a seguir.

```
1(!instantiatenew soapserver)
2(!addchildnew commsystem soapserver)
3(!instantiatenew logger)
4(!addchildnew soapserver logger)
5(!instantiatenew translator)
6(!addchildnew soapserver translator)
7(!instantiatenew soaphandler)
8(!addchildnew soapserver soaphandler)
9(!bindnew commmanager commserveriface soapserver commserveriface)
10(!bindinternalnew soapserver messagelisteneriface commsystem messagelisteneriface)
11(!bindinternalnew soapserver commserveriface soaphandler commserveriface)
12(!bindnew soaphandler logiface logger logiface)
13(!bindnew soaphandler translatoriface translator translatoriface)
14(!bindinternalnew soaphandler messagelisteneriface soapserver messagelisteneriface
15(!start logger)
16(!start translator)
17(!start soaphandler)
18(!start soapserver)
```

Por fim, a tradução deste plano para a linguagem FPath/FScript está descrita na próxima tabela.

Nr	FPath/FScript
1	soapserver = adl-new('file:///src/resources/soapSrv')
2	add($commsystem, $soapserver)
3	logger = adl-new('file:///src/resources/logger')
4	add($soapserver, $logger)
5	translator = adl-new('file:///src/resources/translator')
6	add($soapserver, $translator)
7	soaphandler = adl-new('file:///src/resources/soapHandler')
8	add($soapserver, $soaphandler)
9	bind($commmanager/interface::commserveriface, $soapserver/interface::commserveriface)
10	bind($soapserver/interface::messagelisteneriface, $commsystem/internal-interface::messagelisteneriface)
11	bind($soapserver/internal-interface::commserveriface, $soaphandler/interface::commserveriface)
12	bind($soaphandler/interface::logiface, $logger/interface::logiface)
13	bind($soaphandler/interface::translatoriface, $translator/interface::translatoriface)
14	bind($soaphandler/interface::messagelisteneriface, $soapserver/internal-interface::messagelisteneriface)
15	start($logger)
16	start($translator)
17	start($soaphandler)
18	start($soapserver)

Tabela C.1: O plano de reconfiguração para inserção do protocolo SOAP em FPath/FScript.