Universidade Federal do Rio de Janeiro

Instituto Alberto Luiz Coimbra
de Pós-Graduação e Pesquisa de Engenharia - (COPPE)

Exame de Qualificação

Reconfiguração de Software Automatizada

Marco Eugênio Madeira Di Beneditto

Orientador: Dra. Cláudia Maria Lima Werner

29 de março de 2012

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste Capítulo são apresentados o contexto e a motivação do trabalho, a hipótese de pesquisa, os objetivos a serem alcançados, a metodologia de pesquisa empregada e a organização do trabalho nos demais capítulos.

1.1 Contexto do trabalho

Ao longo do tempo, alguns sistemas devem ser ajustados a fim de atender a um novo conjunto de objetivos e, em alguns casos, estes ajustes precisam ser feitos durante a execução do software, sem que o mesmo seja parado e depois reiniciado. Um exemplo deste tipo de sistema são os empregados no apoio ao comando e controle, seja na área militar ou na área de defesa civil. Durante a operação em curso apoiada por um sistema computacional, o usuário pode dar diferentes prioridades aos objetivos deste sistema, privilegiando um subconjunto dos objetivos em relação a outros. Esta nova atribuição de prioridades, que reflete o contexto, pode ter como consequência uma configuração de software diferente daquela em execução. Desta forma, as necessidades do usuário podem variar de acordo com as variações contextuais e, como consequência, o sistema precisa ser ajustado para poder continuar a apoiar a operação em andamento.

Para este problema, pode-se pensar numa solução que possua um comportamento *adaptativo* ou *adaptável*. Segundo Müller et al. (2008), uma solução é adaptativa quando o seu projeto incorpora mecanismos para instrumentar e observar a solução e o seu ambiente e para modificar o comportamento do sistema em resposta às mudanças observadas. Desta definição fica claro que as mudanças sofridas pelo software são uma consequência das observações feitas sobre o ambiente e o software em si, ou seja, há um processo de realimentação que, a partir das observações, decide o que deve ser alterado no comportamento do software, a fim de controlá-lo.

Esta solução adaptativa possui bastante semelhança com a definição de *agente*

inteligente na área de Inteligência Artificial, em que um *agente* percebe o seu *ambiente* por meio de sensores e age sobre este *ambiente* por meio de atuadores (RUSSELL; NORVIG, 2010). No caso do software adaptável, o agente seria o mecanismo responsável por decidir o que deve ser ajustado e o ambiente seria o software a ser ajustado. Este processo se repete inúmeras vezes ao longo do tempo, onde o agente emprega um mecanismo de tomada de decisão que determina que ações executar a partir dos dados dos sensores.

Cheng et al. (2009) destacam que sistemas auto-adaptáveis devem possuir um mecanismo de realimentação como o descrito anteriormente. Na literatura, este mecanismo foi instanciado de várias formas como o ciclo Monitorar-Analisar-Planejar-Executar (KEPHART; CHESS, 2003), também conhecido pelo acrônimo **MAPE-K**, e o ciclo Coletar-Analisar-Decidir-Agir (DOBSON *et al.*, 2006). Ambos os ciclos têm como objetivo ajustar o software a partir dos dados obtidos pela *Coleta* ou *Monitoração*. Em relação ao ciclo **MAPE-K**, empregado na literatura de sistemas autonômicos e ilustrado na Figura 1.1, ele compõe o *Gerente Autonômico*, que pode ser visto como o *Agente Iteligente* anterior, e é responsável por acompanhar e ajustar o *Elemento Gerenciado* que é o software a ser adaptado.

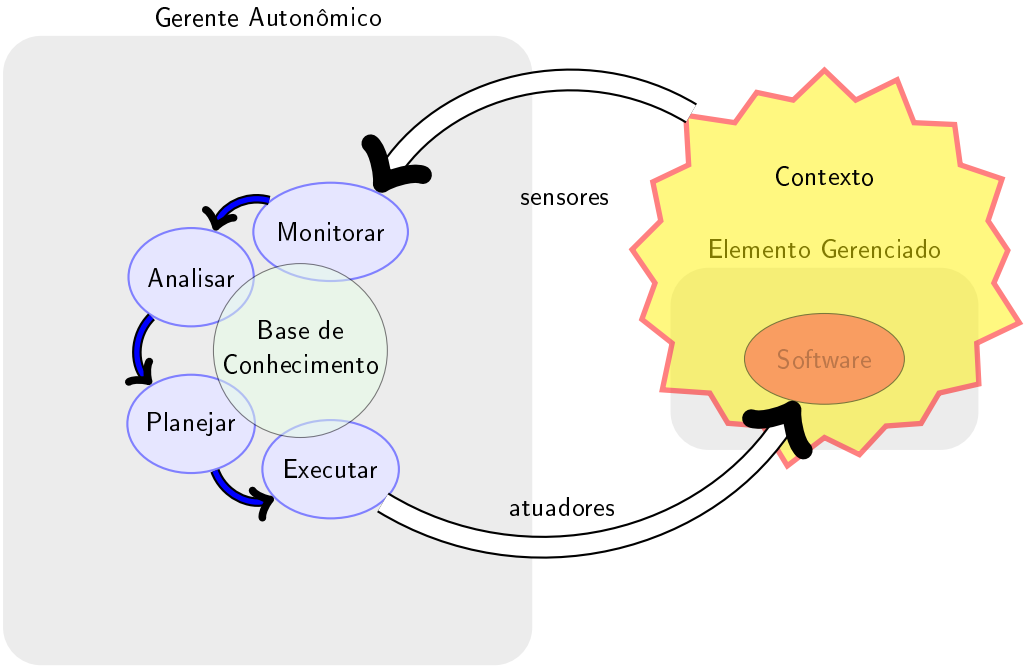


Figura 1.1: Atividades do ciclo de decisão `MAPE-K` (adaptado de (KEPHART; CHESS, 2003)).

Uma maneira de executar os ajustes necessários seria manipular o software a partir do nível algorítmico ou da linguagem de programação, por exemplo, estendendo o comportamento de uma classe ou introduzindo uma nova versão de uma classe. Porém, se a decisão for feita a partir deste nível, o Gerente Autonômico 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 adaptativas. Uma forma de resolver isto é considerar o software num nível de abstração mais elevado, como no nível arquitetural, de forma que os ajustes determinados neste nível possam ser traduzidos para os níveis mais baixos, quando necessário. 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. Em (KRAMER; MAGEE, 2007), os autores mencionam uma série de benefícios em se considerar a adaptação no nível arquitetural, como:

- 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.

A abordagem em nível arquitetural vem ao encontro de uma das formas de adaptação de um software. Segundo McKinley *et al.* (2004), há duas abordagens para 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. A maior flexibilidade permitida pela *Adaptação composicional* - ou *Adaptação estrutural*, segundo Buckley *et al.* (2005) - implica numa maior complexidade para a sua realização, pois algumas perguntas devem ser respondidas:

1. Como obter as ações que irão reconfigurar um software, dado que este software já está numa determinada configuração e em execução ?
2. Como garantir que a nova configuração atende às dependências entre componentes, interfaces requeridas e providas ou estilos arquiteturais ?
3. Como efetuar os ajustes no software em tempo de execução e quais as consequências deste ajuste no funcionamento do sistema ?

Na abordagem composicional um software auto-adaptável pode, ao longo do tempo, assumir diferentes configurações, dado que algum aspecto do contexto cause uma alteração entre configurações. Deste modo, este software pode ser representado por meio de um Diagrama de Estados, em que:

- Estado - representa uma determinada configuração do software.
- Transição - representa uma transição entre estados, ou seja, entre configurações do software.

Este diagrama representa as adaptações possíveis de um software, isto é, as possíveis configurações que um software pode assumir, e a evolução deste diagrama representa adição ou remoção de estados ou transições. Por exemplo, se um componente é removido de um software, deixando de fazer parte do mesmo em qualquer situação, os estados onde este componente existia devem ser removidos do diagrama, bem como as respectivas transições. Detalhando um pouco mais este diagrama, a descrição de um estado pode ser feita por meio de alguma linguagem de representação e as transições podem representar diversos conceitos, como: as *informações do contexto* que causam uma transição entre configurações, ou as *ações* que alteram uma configuração de um software.

Um outro aspecto importante sobre o mecanismo responsável pela reconfiguração é a autonomia com que ele planeja e executa os ajustes necessários em relação ao contexto. Em (RUSSELL; NORVIG, 2010), os autores dizem que: "um sistema é autônomo na medida em que o seu comportamento é determinado pela sua própria experiência" sendo que um agente não possui autonomia se ele não precisa prestar atenção às suas percepções e as suas ações são baseadas somente no conhecimento embarcado. Neste último caso, caberia ao desenvolvedor prover este conhecimento embarcado, de forma que o software pudesse utilizá-lo nas ações de reconfiguração. Olhando para o diagrama de estados, pode-se observar que quanto mais estados e transições este diagrama contiver, mais informação o desenvolvedor deverá inserir no sistema, a fim de que ele efetue todas as reconfigurações possíveis, o que seria uma tarefa inexequível para diagramas de tamanho razoáveis, visto que o número de transições entre os estados cresce com o quadrado do número de estados.

O trabalho aqui proposto se encaixa na área de engenharia de software para sistemas auto-adaptáveis que empregam a adaptação composicional no nível arquitetural. Neste contexto, esta pesquisa irá investigar as atividades de **Planejar** e **Executar** do ciclo ilustrado na Figura 1.1, a fim de obter uma solução que trate da complexidade em se reconfigurar um sistema em execução e que necessite de pouca intervenção humana, ou seja, tenha um alto grau de autonomia, permitindo a execução das validações necessárias nesta atividade.

1.2 Motivação

Um sistema computacional auto-adaptável é uma solução para um software que precise atender a diferentes contextos ao longo do tempo e, como consequência

desta mudança contextual, deve ser ajustado para poder continuar atendendo aos seus objetivos. Como exemplo destes sistemas, no âmbito da Marinha do Brasil há o **Sistema Naval de Comando de Controle** (SisNC2), que interage com uma série de outros sistemas, em especial, com os relacionados ao acompanhamento do trafego marítimo, e que possui algumas características que variam de acordo com a operação em andamento e a situação em que ela se encontra, como diferentes prioridades entre os objetivos do sistema. Por exemplo, em alguns casos o sistema pode priorizar o tráfego de informações entre alguns centros de controle, ou mesmo precisar aumentar o seu desempenho. Deste modo, a reconfiguração do sistema pode ser necessária para poder atender a esta nova atribuição de prioridades.

Como mencionado anteriormente, o ciclo de decisão **MAPE-K** cumpre a função de um *controlador*, que ajusta o software quando for necessário, e também pode ser visto como o ciclo de decisão empregado em Agentes Inteligentes na área de Inteligência Artificial. 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. No âmbito desta proposta de pesquisa, o **objetivo** a ser alcançado é uma nova configuração arquitetural e a **decisão** é como obter esta nova configuração, ou seja, qual a sequência de ações que precisam ser executadas para se reconfigurar o software. O interesse recai sobre o processo de como obter esta nova configuração, dado que o objetivo tenha sido fornecido e, se esta decisão for possível, executá-la sobre o software. Em outras palavras, esta pesquisa irá tratar de como automatizar o processo de reconfiguração de um software, dado que este mesmo software está em execução e uma nova configuração é desejada.

Segundo Geffner (2010), há três diferentes abordagens para se resolver este problema de tomada de decisão:

1. Abordagem baseada em programação: aqui, a decisão de qual será a próxima ação é determinada pelo programador. O problema é resolvido pelo programador, codificado no agente e a solução é expressa por um programa de alto nível.

2. Abordagem baseada em modelo: o controlador não é obtido pelo aprendizado mas sim automaticamente derivado de um modelo de ações, sensores e objetivos. Uma solução para o modelo é a decisão do que deve ser feito.

3. Abordagem baseada em aprendizado: o controlador não é fornecido pelo programador e sim pela experiência do agente, por meio de um aprendizado por reforço ou pela experiência de um professor, num esquema supervisionado.

Na literatura, encontram-se diferentes soluções para o mecanismo de con-

trole que se baseiam em uma destas três diferentes abordagens. A mais utilizada é a abordagem programática e nela as transições entre diferentes configurações de software são descritas pelo projetista por meio de um conjunto de regras, as quais são utilizadas ao longo da execução do software. A 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 anterior, 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 deve decidir, por meio de algum critério, qual delas disparar.

A abordagem baseada em modelo possui uma menor complexidade do ponto de vista da especificação, porém transfere a complexidade para o mecanismo de decisão. Apesar de parecer fácil buscar um caminho num diagrama de estados, este mesmo diagrama pode se tornar tão grande que a busca será intratável. Recentemente esta abordagem vem sendo empregada com o objetivo de reconfigurar software no nível arquitetural, como em (SYKES *et al.*, 2008) e (TAJALLI *et al.*, 2010). Em (SYKES *et al.*, 2008), o algoritmo que gera as ações de reconfiguração emprega uma busca em profundidade e, em (TAJALLI *et al.*, 2010), é utilizado um mecanismo de planejamento baseado em verificação de modelos, cuja ideia é gerar planos por meio da verificação da corretude de uma fórmula em relação ao modelo.

Na abordagem por aprendizado, o mecanismo tem que "aprender" o modelo que será utilizado para gerar 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. A maior flexibilidade deste abordagem vem acompanhada do maior desafio computacional, pois agora o controlador deve aprender o modelo e depois empregá-lo para a reconfiguração do software.

Desta forma, a abordagem baseada em modelos pode ser uma solução mais promissora que as demais, pois aumenta as possibilidades de reconfigurações, reduz a complexidade do lado do desenvolvedor e não possui os desafios de uma abordagem baseada em aprendizado. Entretanto, o mecanismo de decisão tem agora uma tarefa computacionalmente complexa e quanto mais completo for o modelo, isto é, quanto mais próximo da realidade ele for, mais complexa é a solução para este modelo. Uma das maneiras de reduzir a complexidade é aumentar o nível de abstração do modelo, o que pode ser obtido empregando-se o nível arquitetural da aplicação ou paradigmas de desenvolvimento de mais alto

nível, como o desenvolvimento baseado em componentes. Mesmo neste nível, um sistema a ser reconfigurado pode demandar uma série de restrições que devem ser respeitadas pelas ações de reconfiguração como: dependências de recursos do sistema operacional, bibliotecas empregadas pelos componentes e controle do ciclo de vida dos artefatos de software.

Se o modelo não representar os aspectos citados anteriormente, o mecanismo responsável pela execução do plano deverá fazê-lo, a fim de poder efetivar a reconfiguração desejada. Este mecanismo de execução também deve lidar com uma série de aspectos inerentes ao modelo de programação utilizado pelo software adaptável e deve prover meios de uma reconfiguração ser desfeita ou mesmo refeita. Diante disso, a solução para a automatização do processo de reconfiguração que será pesquisado neste trabalho deve:

- possuir a capacidade de ser aplicada a diferentes modelos de componentes.
- assegurar que uma determinada reconfiguração, é correta perante um modelo que descreve como este software pode ser reconfigurado.
- prover mecanismos para que uma reconfiguração possa ser desfeita ou refeita.

1.3 Hipótese

Considerando que a abordagem baseada em modelos é uma solução promissora para um agente responsável por ajustar um software, mas que precisa ser refinada para permitir verificações e validações e também considerar aspectos de nível inferior ao arquitetural, o objetivo desta proposta de pesquisa é avaliar a seguinte hipótese:

> Uma abordagem baseada em modelos pode solucionar o problema de geração de um plano de reconfiguração e posterior execução sobre um software adaptável.

A partir deste hipótese, algumas questões de pesquisa foram consideradas:

- Como automatizar o processo de planejamento e execução da reconfiguração de um software ?
- Como garantir que uma reconfiguração é consistente com o paradigma de programação subjacente ?
- A partir de uma solução de reconfiguração automatizada baseada em modelos, como empregar esta solução para softwares desenvolvidos em diferentes paradigmas de programação ?

- O que deve ser feito caso uma reconfiguração não possa ser executada ou não termine ?

1.4 Objetivos

A partir da importância do emprego de sistemas auto-adaptáveis como forma de amenizar ou até mesmo solucionar os problemas advindos do desenvolvimento de sistemas cada vez mais complexos e que precisam ser ajustados em tempo de execução, esta pesquisa tem como propósito fornecer uma solução para a tomada de decisão e ajuste da configuração do software. Este propósito pode ser decomposto nos seguintes objetivos:

1. **Desenvolver um mecanismo para decisão de adaptação de software**: dado que o software precise ser reconfigurado, este mecanismo irá decidir que ações devem ser tomadas - plano de reconfiguração - para se obter a configuração desejada.
2. **Prover mecanismos de explicação sobre a reconfiguração**: dependendo da situação, o mecanismo de decisão pode não encontrar um plano e é importante fornecer ao engenheiro de software um meio dele poder investigar o motivo da não existência deste plano.
3. **Desenvolver um mecanismo para o acompanhamento e execução do ajuste de um software**: este mecanismo receberá o plano de reconfiguração e irá executá-lo sobre o software a ser ajustado, acompanhando a execução de cada etapa do plano e informando caso um plano não possa ser concluído.

1.5 Metodologia de Pesquisa

A metodologia de pesquisa empregada neste trabalho foi dividida em duas etapas:

- elaboração da proposta de tese: que define o conjunto de atividades que foram realizadas para elaborar este proposta de tese.
- desenvolvimento da proposta de tese: que define o conjunto de atividades a serem realizadas para permitir o desenvolvimento da tese.

Esta proposta de tese foi elaborada de acordo com a sequência de atividades apresentadas na Figura 1.2 e descritas a seguir:

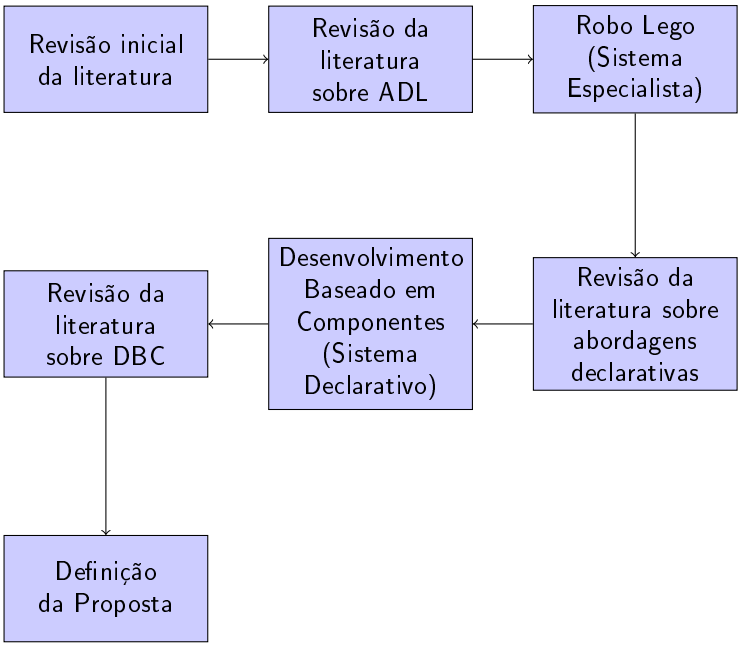


Figura 1.2: Metodologia de pesquisa adotada para a proposta

- Revisão inicial da literatura e ADL: nesta atividade realizou-se uma revisão *ad hoc* da literatura sobre Engenharia de Software para sistemas autoadaptáveis. Este revisão começou com foco em Linha de Produtos de Software Dinâmico e depois passou a considerar outros sistemas que empregassem mecanismos de autoadaptação no nível arquitetural e uma ADL como forma de descrever as configurações de software.

- Implementação sobre um robô com apoio de um sistema especialista: nesta atividade foram iniciadas as primeiras implementações de um sistema autoadaptável sobre uma plataforma robótica. Nesta implementação foi empregada a abordagem programática, apoiada por um sistema especialista, que possuía um conjunto de regras que descrevia a reconfiguração a ser realizada, no nível arquitetural, sobre um software que executava num robô.

- Revisão da literatura sobre abordagens declarativas: após terem sido identificadas uma série de limitações sobre o emprego de um sistema especialista na tarefa de gerar o plano de reconfiguração, foi feita uma nova revisão sobre o emprego de abordagens declarativas. Nesta revisão, a técnica declarativa estudada foi o Planejamento Automatizado.

- Implementação sobre um servidor web com apoio de um sistema declarativo: foi empregada uma aplicação de exemplo para um servidor web baseada num modelo de componentes com capacidade reflexiva. O mecanismo de decisão utilizado foi apoiado por um sistema declarativo que empregava o planejador JSHOP2. Este planejador era responsável por gerar um plano de reconfiguração do software no nível arquitetural.

- Revisão da literatura sobre Desenvolvimento Baseado em Componentes:

após os trabalhos com o robô e com o servidor web foi feita uma revisão da literatura sobre o Desenvolvimento Baseado em Componentes como forma de generalizar as soluções utilizadas.

No Capítulo 5 serão listadas as tarefas que devem ser realizadas para a conclusão desta pesquisa. Deste modo, as contribuições esperadas para este trabalho são:

1. A disponibilização de um sistema declarativo para a geração de planos de reconfiguração.

2. A criação de um mecanismo para a execução dos planos de reconfiguração.

1.6 Organização do Trabalho

Este capítulo apresentou os fatores motivadores para a proposta desta pesquisa, a hipótese de pesquisa, suas perguntas e a solução proposta. Estes tópicos serão mais detalhados nos próximos capítulos e a organização do texto desta proposta é descrita a seguir:

No Capítulo 2, serão abordados os aspectos relacionados a Engenharia para Software Auto-Adaptável, bem como outras áreas que podem auxiliar no desenvolvimento deste tipo de sistema.

No Capítulo 3, serão descritos os Trabalhos Relacionados com esta pesquisa, ou seja, aqueles que tratam de adaptação de software.

No Capítulo 4, a Abordagem proposta será apresentada, descrevendo como se pretende resolver os problemas levantados.

No Capítulo 5, as Considerações Finais sobre esta proposta de pesquisa são descritas, apresentando os resultados já obtidos, a metodologia de pesquisa a ser seguida para o desenvolvimento desta pesquisa e o cronograma tentativo para a sua realização.

Capítulo 2

Engenharia para Software Autoadaptável

Segundo o dicionário Michaelis, *Adaptar* significa: ajustar a novas condições, alterar para tornar adequado, tornar apto (WEISZFLOG, 1998). No domínio de software, seria adequar o software a mudanças que ocorreriam no seu ambiente operacional ou contexto, aqui o contexto inclui o próprio software, e auto-adaptar é efetuar o ajuste por si próprio.

Uma das primeiras definições sobre o que é um software auto-adaptável foi mencionada pela *Defense Advanced Research Projects Agency* (DARPA), em dezembro de 1997, por meio do *Broad Agency Announcement on Self Adaptive Software* (BAA-98-12). Neste documento segundo Laddaga, em (LADDAGA, 2006), mencionava-se que:

> Um software auto-adaptável avalia seu próprio comportamento e muda o comportamento quando: a) a avaliação indica que ele não está atendendo ao que se destina a fazer; ou b) um desempenho ou funcionalidade melhores são possíveis.

Dois anos mais tarde em (OREIZY *et al.*, 1999), o aspecto causador da mudança foi generalizado e o propósito da adaptação foi retirado, resultando na seguinte definição:

> Um software auto-adaptável modifica seu próprio comportamento em resposta a mudanças em seu ambiente operacional. Por ambiente operacional entende-se algo observável pelo software, tais como entradas de um usuário, dispositivos de hardware e sensores ou instrumentação de programas.

Nas definições acima está explícita a capacidade de observar o ambiente e a ele se ajustar, ou seja, há um ciclo de decisão como o mencionado anterior-

mente, o qual também é confirmado nas definições mais atuais para sistemas autoadaptáveis, como:

> Sistemas capazes de ajustar o seu comportamento em resposta a sua percepção do ambiente e de si mesmo (CHENG *et al.*, 2009).

Destas definições, pode-se identificar uma série de aspectos ligados a engenharia de um sistema deste tipo como, por exemplo, a taxonomia de adaptação proposta em (SALEHIE; TAHVILDARI, 2009) que se baseia nas perguntas de **quando**, **o que**, **como**, e **onde**, para levantar estes aspectos de uma série de abordagens propostas na literatura.

Nesta proposta de pesquisa, o interesse é sobre alguns destes aspectos, mais especificamente na geração da reconfiguração e posterior execução sobre o software, ou seja, não há interesse nos fatores causadores de uma mudança nem em como a nova configuração é escolhida. A premissa deste trabalho é que a configuração que deve ser assumida pelo software, isto é, a configuração desejada ou objetivo, já foi decidida por algum outro mecanismo, seja ele manual, automático ou uma combinação dos dois.

Este capítulo trata dos fundamentos teóricos que embasam a geração de um plano de reconfiguração e a posterior execução deste plano sobre o software. Cabe ressaltar que esta geração será feita por meio de uma abordagem baseada em modelos e descrita num nível de abstração elevado. Na Seção 2.1, são tratados os aspectos relacionados ao ciclo de decisão responsável pelos ajustes e algumas maneiras de ver este ciclo. Na Seção 2.2 é vista a teoria de planejamento automatizado, uma técnica que se baseia em modelos, e que irá compor o ciclo de decisão para ser empregada como o mecanismo de geração das ações de reconfiguração. E, por fim, na Seção 2.3, são tratadas as questões relativas ao emprego de um nível de abstração elevado para a geração a reconfiguração, bem como a posterior execução deste plano sobre um software.

2.1 O ciclo de decisão

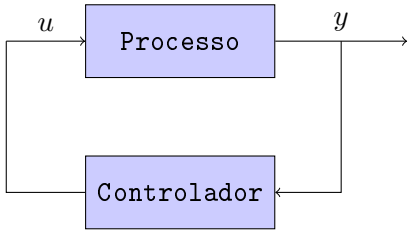
Como destacado em (MÜLLER; PEZZÈ; SHAW, 2008), o mecanismo responsável por ajustar um software deve ser explicitado em termos de projeto ou ser claramente observável na implementação. Este mecanismo tem sido visto de duas formas: a) por meio da teoria de controle; e b) do ponto de vista da teoria de agentes inteligentes. O uso de uma malha de controle fechada já foi destacado na literatura, em (DOBSON *et al.*, 2006), (CHENG *et al.*, 2009) e (LEMOS *et al.*, 2011), como um elemento importante na engenharia de software autoadaptável.

A literatura também reconhece que este mecanismo não pode deixar de observar o contexto, ou melhor, o ambiente e o software a ser ajustado, formando

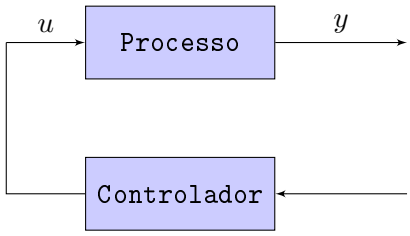
um ciclo ou uma malha fechada. Nesta Seção, é abordado com mais profundidade o ciclo de decisão descrito na Seção 1.1. Este ciclo é tratado sob o ponto de vista da Teoria de Controle e de Agentes Inteligentes, sendo que este último é visto com mais detalhes por se tratar da abordagem empregada nesta pesquisa.

2.1.1 Engenharia de Controle

Segundo Dean e Wellman em (DEAN; 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*. 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; MURRAY, 2008).



(a) Malha Fechada - *Closed Loop*



(b) Malha Aberta - *Open Loop*

Figura 2.1: Malhas de Controle

Um **Sistema de Controle a Malha Fechada** ou **Sistema de Controle com Retroação**, ou ainda **Realimentação**, como o ilustrado na Figura 2.1a, 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 (OGATA, 2010). 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*.

Já na Figura 2.1b, é ilustrado um **Sistema de Controle a Malha Aberta**, onde o **Controlador** não recebe informações do **Processo**, ou seja, um controla-

dor malha aberta não observa a saída do processo controlado como, por exemplo, um sinal de trânsito que abre e fecha o semáforo de acordo com o tempo.

O propósito de um sistema de controle com realimentação é manter a saída do processo controlado numa dada **referência**, ou muito próximo dela. Para processos físicos no mundo real, é difícil executar o controle baseado num conjunto pré-definido, como o utilizado numa malha aberta, ilustrado na Figura 2.1b. Ao invés disso, geralmente é necessário monitorar o processo e ajustá-lo na faixa desejada e a **realimentação** permite que o sistema de controle se adapte a circunstâncias variadas.

Considerando os sistemas autoadaptáveis, em (VILLEGAS *et al.*, 2011), os autores propõem um modelo de caracterização para sistemas autoadaptáveis. Neste modelo, uma das dimensões para a caracterização é a estrutura do sistema composta do **controlador** e do **sistema gerenciado**. Em relação ao controlador, os autores o classificam como:

a. *fixo*, quando as funções de transferência do controlador não variam.

b. *adaptativo*, quando o controlador pode ser ajustado.

c. *reconfigurável*, quando os algoritmos do controlador podem ser alterados.

As **funções de transferência** são utilizadas para caracterizar as relações de entrada e saída de componentes ou sistemas que podem ser descritos por equações diferenciais lineares invariantes no tempo (OGATA, 2010). No caso ilustrado na Figura 2.1, estas relações se dariam entre a variável controlada y e a variável manipulada u. Em geral, esta função é obtida pela razão entre a transformada de Laplace da saída sobre a transformada de Laplace da entrada e o emprego da transformada de Laplace visa facilitar a manipulação das equações diferenciais que são transformadas em equações algébricas mais simples.

Há processos a serem controlados em que um modelo apropriado é conhecido, mas os seus parâmetros não, e.g., um processo pode ser modelado usando uma equação de ordem-k, mas não se sabe qual será o valor de k. O **controle adaptativo** é o método utilizado por um controlador que deve se ajustar ao sistema gerenciado o qual possui parâmetros que podem variar ou são inicialmente desconhecidos (DEAN; WELLMAN, 1991). Por exemplo, um foguete enquanto em vôo consome seu combustível e varia o seu peso. Neste caso, é possível achar valores para os parâmetros do modelo, empregando uma amostragem na entrada e na saída, o que é denominado de *identificação do sistema* e compõe o controle adaptativo.

Quanto ao controlador ser **reconfigurável**, os autores em (VILLEGAS *et al.*, 2011) não tecem maiores detalhes nem exemplificam como poderia ser feita a troca dos algoritmos. As abordagens citadas pelos autores como exemplos de

controladores reconfiguráveis empregam uma abordagem programática, como será vista na Seção 2.1.2. Nesta abordagem, o que poderia ser trocado seriam as regras, e não o mecanismo de seleção do que deve ser feito. Trocar as regras significa ser possível introduzir novas opções para decidir o que deve ser feito numa determinada situação.

Para a estrutura do **sistema gerenciado**, as opções são:

a. *estrutura não modificável*, ou monolítica; e

b. *estrutura modificável* com ou sem capacidade reflexiva.

Cabe ressaltar que nem todas as possíveis combinações entre o controlador e o sistema gerenciado são possíveis para a estrutura do sistema.

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. 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 são vistos na próxima Seção, onde é tratada a visão de Agentes Inteligentes.

2.1.2 Agentes Inteligentes

Na área de Inteligência Artificial, um *agente* é algo que age num ambiente. Em (RUSSELL; NORVIG, 2010), os autores definem um agente como uma entidade que pode ser vista como: percebendo o seu ambiente por meio de sensores e agindo sobre este ambiente por meio de atuadores. Este agente é *racional* quando ele faz a coisa certa, isto é, a ação é apropriada para seus objetivos e circunstâncias.

Percepções	**Ações**	**Objetivos**	**Ambiente**
a) Configuração de software atual b) Regras de composição dos componentes c) Estilos arquiteturais d) Componentes disponíveis	a) Ativar ou Desativar componente b) Carregar ou Descarregar componente	a) Otimizar b) Atender a requisitos	a) Software

Tabela 2.1: Descrição de um agente que realiza adaptação de software no nível arquitetural.

Por exemplo, considerando um agente responsável por controlar um software, na Tabela 2.1, pode-se observar quais seriam as **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 ele toma as decisões, isto é, que o agente decide e executa um conjunto de ações. Deste modo, para se desenvolver este agente, precisa-se implementar o mapeamento da sequência de percepções para as suas ações. 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 uma tarefa intratável, devido ao grande número de possibilidades, e o agente não tem capacidade de se adaptar a mudanças pois, neste último caso, caso uma dada percepção não esteja mapeada, o agente não irá desempenhar qualquer ação.

O modo como este *mapeamento* pode ser feito foi classificado, segundo (RUSSELL; NORVIG, 2010), 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 irá chegar até este estado por meio das ações descritas nas regras. Isto ficará mais claro no exemplo descrito abaixo, inspirado no projeto Rainbow [1], descrito em (GARLAN *et al.*, 2004), e que emprega uma abordagem baseada em reflexo.

Considerando um determinado sistema, organizado num agrupamento de servidores que devem atender a requisições de clientes externos, o desenvolvedor deve descrever o que deve ser feito na arquitetura para um determinado estado. Por exemplo, dado o evento abaixo:

> O TEMPO DE RESPOSTA ÀS REQUISIÇÕES DO AGRUPAMENTO DE SERVIDORES **1** ESTÁ NO VALOR MÁXIMO ACEITÁVEL.

No sistema em questão, este evento indica que os clientes que efetuam uma requisição ao agrupamento de servidores estão esperando muito tempo para receber uma resposta e o serviço poderá ficar comprometido. Deste modo, uma condição que poderia ser observada seria:

> O SERVIDOR **A** ESTÁ EM EXECUÇÃO, OCIOSO E NÃO PERTENCENTE A QUALQUER AGRUPAMENTO DE SERVIDORES.

[1] http://rainbow.self-adapt.org/

Este evento e condição representam um estado denominado de ESTADO 1. Quando este estado for verdadeiro o controlador deve executar a seguinte ação, denominada de **AÇÃO X**:

DESIGNAR O SERVIDOR **A** PARA O AGRUPAMENTO DE SERVIDORES **1**.

Este é o modo como um desenvolvedor descreve o mecanismo de controle na abordagem programática: quando o **ESTADO 1** for verdadeiro então executar a **AÇÃO X**. Esta descrição é feita por meio de regras denominadas de Evento-Condição-Ação (ECA) que são introduzidas num sistema especialista que mantém estas regras e, quando os fatos existentes na memória de trabalho tornam uma ou mais regras verdadeiras, isto é, possíveis de serem executadas, o sistema especialista escolhe uma delas para execução.

Como o evento e a condição representam um determinado estado no diagrama de estados, a regra irá dizer o que deve ser feito sempre que este estado for atingido e a execução desta ação leva a um outro estado, isto é, a um estado objetivo ou desejado. Considerando uma adaptação no nível arquitetural, uma alteração de estado promovida pelo controlador representa uma reconfiguração arquitetural de um software, pois é desta forma que ele controla o software. O controlador também espera que esta nova configuração leve o software a um comportamento desejado, ou seja, que as variáveis controladas voltem aos valores desejados e representem um estado objetivo.

A execução da **AÇÃO X** irá alterar o estado pois o software irá assumir uma nova configuração arquitetural. Esta semântica de um diagrama de estados pode ser vista na Figura 2.2.

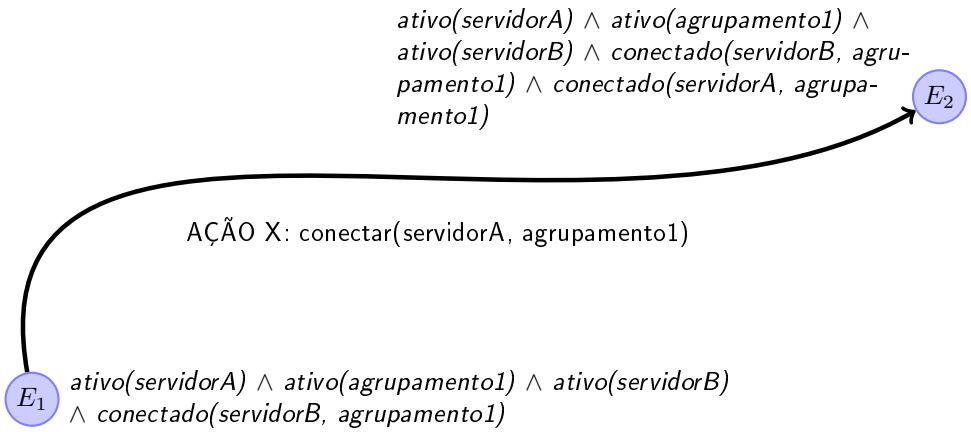


Figura 2.2: Diagrama de transição de estados, com os estados E_1 e E_2, referente a **AÇÃO X**.

Neste diagrama, tem-se:

- E_1: é o estado inicial, denominado de **ESTADO 1**, e representa a configuração arquitetural atual, no qual o servidor **B** está ativo e conectado ao agru-

pamento de servidores **1** e o servidor **A** está ativo mas não está conectado a qualquer grupamento.

- E_2: é o estado objetivo, denominado de **ESTADO 2**, que é obtido após a execução da **AÇÃO X**, e onde os servidores **A** e **B** estão conectados ao agrupamento de servidores **1**.

Como foi dito anteriormente, apesar de não estar explícito, o desenvolvedor sabe qual a configuração de software para aquele contexto, isto é, o *estado objetivo*, e ele descreve o que deve ser feito para obtê-la. Deste modo, há duas questões sendo respondidas:

1. Qual a configuração arquitetural de um software que é adequada para um dado contexto ?
2. Dado que há configuração, isto é, que há uma resposta para a primeira pergunta, como obter esta configuração ?

Esta pesquisa está interessada em como responder à segunda pergunta de forma mais autônoma em relação ao desenvolvedor e com capacidade de fornecer mecanismos de verificação sobre o processo de decisão empregado. Para isto, pretende-se obter uma solução mais declarativa, como num agente baseado em objetivo, na qual o desenvolvedor não tenha que descrever as ações. Na Seção 2.2, a área de planejamento em IA será abordada com mais detalhes, pois ela é um dos campos de IA dedicados a achar a sequência de ações que permitem ao agente atingir os seus objetivos, ou seja, o planejamento será empregado na geração da reconfiguração de um software.

2.1.3 Computação Autonômica

Como citado em (HUEBSCHER; MCCANN, 2008), a visão de agente inteligente da área de Inteligência Artificial deve ter servido de inspiração para o conceito de **Elemento Autonômico** empregado na Computação Autonômica. Em (KEPHART; CHESS, 2003), os autores propõem que o mecanismo de controle seja executado por um **Gerente Autonômico** que é composto por uma malha denominada de MAPE-K (*Monitor-Analyze-Plan-Execute-Knowledge*). Este Gerente, que desempenha o papel de controlador, juntamente com o **Elemento Gerenciado**, que é o sistema a ser controlado, compõem o **Elemento Autonômico**. O Gerente Autonômico monitora o Elemento Gerenciado e o seu ambiente externo e constrói e executa os planos que ajustarão o Elemento Gerenciado, baseando-se na análise desta informação e na base de conhecimento.

O ciclo do Gerente Autonômico é composto pelas seguintes atividades: a) Monitorar; b) Analisar; c) Planejar; e d) Executar. Além das atividades, uma

base de conhecimento está disponível para ser utilizada por qualquer uma delas. O processo de **monitorar** é responsável por coletar e correlacionar os dados provenientes de sensores, convertendo-os no formato esperado para serem analisados. O processo de **analisar** recebe os dados e é responsável pela detecção de uma mudança no contexto para a qual seja necessária uma adaptação. A **decisão** determina a configuração objetivo, o que precisa ser modificado para se obtê-la e como esta modificação deve ser feita a fim atingir a configuração objetivo, a partir da configuração atual. Por fim, o plano anteriormente estabelecido deve ser **executado** no software, promovendo a sua reconfiguração até atingir a configuração objetivo.

O **Objetivo da Adaptação** é a principal razão para o emprego de um software auto-adaptável e, em geral, ele é definido por meio de uma ou mais das propriedades auto-* (*self-**). Na Computação Autonômica (KEPHART; CHESS, 2003), os autores definem um Sistema Autonômico como um tipo de sistema computacional com capacidade de poder gerenciar a si próprio, dado que objetivos de alto nível tenham sido fornecidos por administradores deste tipo de sistema. A essência de um sistema autonômico é a sua capacidade de *Auto-Gerenciamento* que é descrita por quatro propriedades, também conhecidas como propriedades auto-* (*self-**):

1. Auto-Configuração (Self-Configuring): é a capacidade de reconfiguração automática por meio da instalação, atualização, integração e composição/decomposição das entidades de software.

2. Auto-Otimização (Self-Optimizing): é a capacidade de gerenciamento do desempenho e alocação de recursos a fim de satisfazer os requisitos.

3. Auto-Cura (Self-Healing): capacidade de descobrir, diagnosticar e reagir a perturbações. Também pode antecipar problemas potenciais e tomar ações para prevenir falhas.

4. Auto-Proteção (Self-Protecting): é a capacidade de detectar brechas de segurança e se recuperar de seus efeitos.

As quatro propriedades auto-* mencionadas acima não são as únicas encontradas na literatura. Em (SALEHIE; TAHVILDARI, 2009), os autores descrevem além destas outras como, Auto-Organização, Auto-Controle e Auto-Manutenção.

Apesar dos exemplos de aplicação empregados nesta pesquisa, que serão vistos mais adiante, serem ligados a propriedade de *Auto-Configuração*, esta pesquisa visa buscar uma solução que poderá ser empregada para apoiar as outras propriedades auto-*, pois ela visa estudar a geração da reconfiguração, a partir de uma configuração atual e uma desejada, e a sua execução sobre um sistema. Simplificadamente, o interesse desta pesquisa é: dado um objetivo em termos de

uma configuração arquitetural, obter este objetivo. A tarefa de gerar este objetivo, que no caso é a configuração desejada, não está no escopo desta pesquisa.

2.2 Planejamento Automatizado

Para Dean e Wellman, *planejamento* é o problema de formular uma sequência de ações ao longo do tempo para se atingir um determinado objetivo. Em (NAU; GHALLAB; TRAVERSO, 2004), *Planejamento* é o processo de decidir uma escolha de ações a fim de atingir algum objetivo. Segundo Weld em (WELD, 1999), numa formulação *clássica* e simples, o problema de planejamento pode ser definido por três entradas: 1) uma descrição do estado inicial do mundo em alguma linguagem formal; 2) uma descrição da meta do agente, i.e., qual o comportamento desejado, em alguma linguagem formal; e 3) uma descrição das ações possíveis que podem ser realizadas, novamente em alguma linguagem formal. Cada ação possui algumas precondições e alguns efeitos. As precondições são o que deve ser válido no estado atual 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, o mecanismo de controle pode prever o que ocorrerá se ele tomar uma determinada ação, sem precisar fazê-la e depois observar o estado do processo para recuperar o estado alcançado.

2.2.1 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. **Σ finito** - O número de estados é finito.
2. **Σ 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. **Σ 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. **Σ 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.

2.2.2 A Representação Clássica

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* para denotar relacionamentos entre objetos, *constantes* para denotar objetos específicos, e sem 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:

- $ativo(x)$, lê-se "x está ativo";
- $conectado(x, y, z)$, lê-se "x está conectado a y na interface z";
- $requer(x, y)$, lê-se "x requer a interface y";
- $prove(x, y)$, lê-se "x provê a interface y";

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:

- $nome(o)$, 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 $efeitos(o)$, a *precondição* e os *efeitos* de o, respectivamente, são conjuntos de átomos.

Para o problema de reconfiguração de um software, pode-se ter os seguintes operadores descritos na Tabela 2.2

NOME	**DESCRIÇÃO**	**PRECOND**	**EFEITOS**
$ativar(x)$	ATIVA O COMPONENTE x	$\neg ativo(x)$	$ativo(x)$
$desativar(x)$	DESATIVA O COMPONENTE x	$ativo(x)$	$\neg ativo(x)$
$conectar(x, y, z)$	CONECTA OS COMPONENTES x E y POR MEIO DA INTERFACE z	$ativo(x)$ $ativo(y)$ $prove(y, z)$ $requer(x, z)$	$conectado(x, y, z)$
$desconectar(x, y, z)$	DESCONECTA OS COMPONENTES x E y NA INTERFACE z	$conectado(x, y, z)$	$\neg conectado(x, y, z)$

Tabela 2.2: Operadores de planejamento para o domínio de reconfiguração

Na Tabela 2.2, o operador $conectar(x, y, z)$ tem como precondição: os dois componentes estarem ativos e a interface requerida pelo componente x ser provida pelo componente y. Esta precondição é descrita pelos predicados apresentados na tabela e a execução deste operador, instanciado por uma ação como $conectar(c_1, c_2, i_3)$, irá gerar um outro estado onde o predicado $conectado(c_1, c_2, i_3)$ é verdadeiro. Já o operador $desconectar(x, y, z)$ tem como precondição: os dois componentes estarem conectados. Esta precondição é descrita pelo predicado apresentado na tabela e a execução deste operador, instanciado por uma ação como $desconectar(c_1, c_2, i_3)$, irá gerar um outro estado onde o predicado $conectado(c_1, c_2, i_3)$ é falso, ou na notação $\neg conectado(c_1, c_2, i_3)$, onde $\neg$ significa a negação.

Um domínio de planejamento pode ser descrito por meio de um grafo orientado e rotulado, denominado grafo de transições de estados. Nesse grafo, os vértices representam todos os possíveis estados do mundo enquanto as arestas, rotuladas com as ações (os operadores de planejamento instanciados), representam todas as possíveis transições de estados no mundo. Na Figura 2.3, o estado s_0 possui dois componentes de software ativos, o estado s_2 possui os mesmos componentes que s_0 só que agora conectados. Esta conexão entre os dois componentes ocorre devido a ação $conectar(c_1, c_2, i_3)$, que conecta os dois componentes por meio da interface i_3.

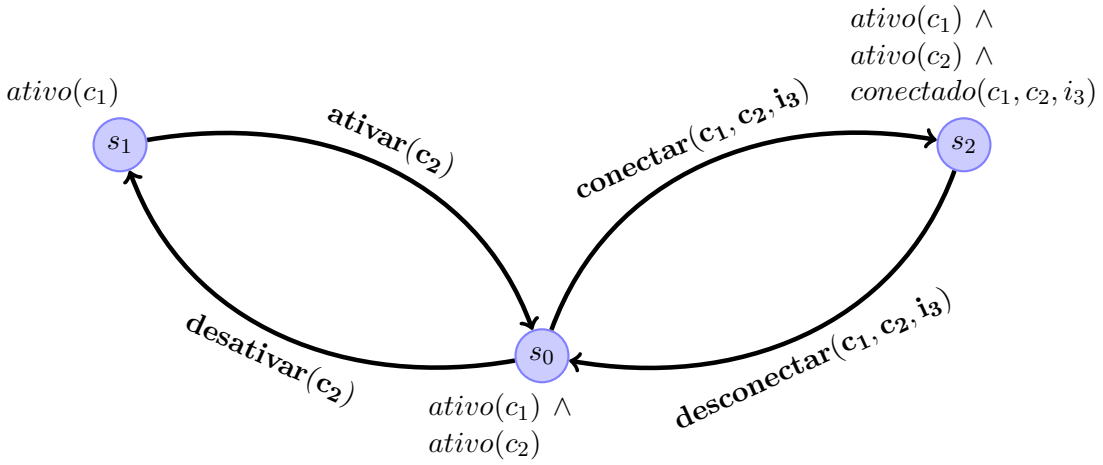


Figura 2.3: Um grafo de transições de estado rotulado correspondente a um domínio de planejamento.

2.2.3 Geração do plano

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 entre os estados s_0 e s_1, por meio da ação a_1 e com origem em s_0. No exemplo anterior, há uma transição entre s_0 e s_2, assim esta transição significa que $s_2 \in \gamma(s_0, a_2)$, onde $a_2 = conectar(c_1, c_2, i_3)$. Dado um problema de planejamento clássico, um *planejador* consiste num procedimento que efetua uma busca no grafo de transições de estados correspondente ao domínio de planejamento, tentando encontrar um caminho no grafo 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. Este caso restrito pode parecer simples de ser resolvido, porém o grafo de um domínio $\mathbf{\Sigma}$ pode ser grande o bastante para tornar inviável sua representação explícita. Deste modo, os *planejadores* necessitam de representações compactas e que facilitem a busca.

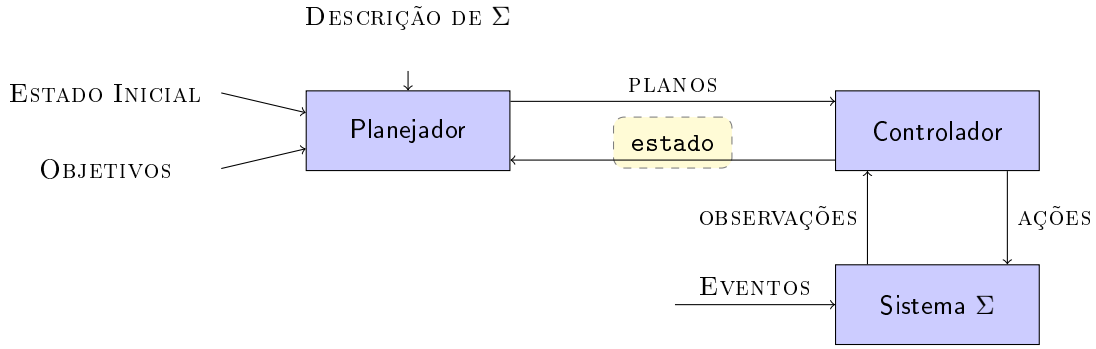


Figura 2.4: Um modelo conceitual simples para planejamento - adaptado de (NAU; GHALLAB; TRAVERSO, 2004)

Um modelo conceitual para o planejamento pode ser visto na Figura 2.4. Um planejamento *offline* é obtido pela seguinte interação entre os três componentes:

- Um PLANEJADOR recebe como entrada a descrição do sistema Σ, um estado inicial, algum OBJETIVO e sintetiza um plano para o CONTROLADOR, a fim de atingir o OBJETIVO.
- Um CONTROLADOR recebe como entrada s, o estado atual do sistema, e fornece como saída uma ação a de acordo com o plano. A saída `estado` não é considerada no planejamento *offline* e será explicada mais adiante.
- Um sistema de transição de estados 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 pela saída `estado`, onde o CONTROLADOR realimenta o PLANEJADOR. Esta realimentação retorna ao planejador o estado da execução do plano, permitindo o planejamento dinâmico.

Na prática, algumas das premissas anteriormente citadas 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, e que causam transições de estado, assim como as ações, mas que não são controlados pelo executor do plano, por exemplo, a falha repentina de um componente de software; e 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.

2.2.4 O Controlador

No modelo conceitual ilustrado na Figura 2.4, o CONTROLADOR é o componente responsável por executar o plano decidido pelo PLANEJADOR. Nesta execução, o controlador deve levar o Sistema Σ, ou melhor, o software ajustável, até a configuração objetivo, ou seja, levar do estado inicial até o estado final. Esta transição pode envolver uma série de estados intermediários, como pode ser visto na Seção 2.2. Cabe ao controlador verificar estas transições intermediárias por meio da verificação dos estados alcançados após a execução de cada uma das ações, o que é feito com as pré-condições de cada uma das ações contidas no plano, ou seja, antes de executar uma ação, o controlador verifica se ela pode ser executada naquele estado. Uma ação não é instantânea, ou seja, a premissa de que as ações não têm duração é falsa e cabe ao controlador verificar se o estado esperado foi alcançado. O controlador pode ativamente verificar se o estado esperado foi atingido ou esperar até que ele seja verdadeiro e isto pode depender do modelo

de programação subjacente, por exemplo, se um determinado componente deve ser colocado num determinado estado antes de ser reconfigurado, o componente pode informar ao controlador o estado alcançado ou o controlador pode ter que verificar se o estado foi alcançado, pois o componente só informa um estado se lhe for perguntado.

A não obtenção de um estado desejado pode impedir que o controlador continue com a execução da próxima ação. Este estado não desejado pode ocorrer devido ao Sistema Σ ser não determinístico, isto é, a ação contida no plano pode levar a mais de um estado ou ao Sistema Σ ser dinâmico, onde algum evento externo diferente de uma ação, pode alterar um estado. Para o primeiro caso, há algoritmos de planejamento que consideram que o domínio é não determinístico, gerando um plano que é denominado de **política**. Uma política é um conjunto de pares ordenados (s, a) tal que: $s \in S$, i.e. s é um estado, e $a \in A(s)$, a é uma ação aplicável no estado s. Uma política é executada por um controlador que observa o estado e seleciona a ação que deve ser executada neste estado, ou seja, dado uma política π, o controlador identifica um estado s e seleciona uma ação a tal que $(s, a) \in \pi$.

No caso do sistema Sistema Σ ser dinâmico, cabe ao projetista do controlador decidir o que será feito. Pode-se pensar nas seguintes situações:

1. tentar replanejar para atingir novamente o estado objetivo. Um novo estado inicial é enviado ao planejador para que ele possa gerar um novo plano a partir deste novo estado mantendo o estado objetivo anterior.

2. tentar retornar ao estado inicial que foi utilizado para a geração do plano, desfazendo as alterações efetuadas até o momento, em outras palavras, efetuar um *rollback*. Neste caso o planejador também pode ser empregado para levar a aplicação do estado atual para o estado inicial anterior, ou seja, o estado inicial será o estado objetivo a ser atingido, retornando a aplicação ao estado anterior a aplicação do plano.

2.3 Adaptação arquitetural

Abstração é uma das maneiras de se lidar com complexidade. Em (ROTHENBERG, 1989), o autor cita que, em sentido amplo, *Modelagem* é o uso mais eficaz, em termos de custos, de algo no lugar de alguma outra coisa para algum propósito cognitivo. Um modelo permite representar a realidade para um dado propósito; o modelo é uma abstração da realidade no sentido em que ele não pode representar todos os aspectos da realidade. Do ponto de vista dos sistemas autoadaptáveis, o emprego de modelos do sistema permite com que o problema seja tratado de uma forma simplificada, ou seja, reduz a quantidade de artefatos que

o projetista deve especificar no desenvolvimento para poder executar o software.

Como já mencionado na Seção 1.1 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 um 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.

Por isso, considerar as adaptações de software num nível de abstração mais elevado, como o nível arquitetural, possibilita a generalização da solução permitindo que ela seja empregada em diferentes domínios. A arquitetura de software descreve o conjunto principal de decisões de projeto feitas sobre um sistema (TAYLOR; MEDVIDOVIC; DASHOFY, 2009). 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.

Em relação ao ajuste de um software, se o nível de abstração considerado para decidir sobre a adaptação fosse de mais baixo nível, como o algorítmico ou a linguagem de programação, o modelo seria mais complexo e, neste nível, 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.

Considerando o emprego de Planejamento Automatizado, descrito na Seção 2.2, para efetuar a geração do plano de adaptação arquitetural, faz-se necessário descrever:

- o domínio Σ; e
- o estado inicial da aplicação s_0.

O domínio deve conter as ações que alteram uma configuração arquitetural e estas irão depender dos conceitos empregados na linguagem de representação. Por exemplo, se os atributos e o ciclo de vida de um componente são representados e se as respectivas alterações representam diferentes configurações, então as ações que representam estas alterações devem pertencer ao domínio Σ. A lista abaixo poderia ser um conjunto de ações possíveis de serem executadas

para alterar a configuração de um software, isto é, transitar entre dois estados adjacentes do diagrama de transição.

- instanciação e destruição de componentes e conectores.
- adição e remoção de componentes e conectores.
- associação e desassociação (*binding* e *unbinding*) de componentes e conectores.
- controle do ciclo de vida de um componente: início e parada.
- ajuste dos atributos de componentes e conectores.

Se for considerada a possibilidade de composição de componentes, isto é, componentes compostos, então pode-se acrescentar as ações de recomposição de componentes e a reassociação (*rebind*) entre interfaces de componentes compostos com os componentes que o compõem.

Está claro que quanto mais "detalhada" for a linguagem de representação e quanto mais detalhes este modelo contiver, maior será o número de possíveis estados e mais complexo será um problema neste domínio. Este é justamente o propósito de se representar a realidade por meio de um modelo, ou seja, reduzir a complexidade, restando decidir qual o nível de abstração deste modelo. Além disso, este nível deve ser o suficiente para permitir que um software seja efetivamente ajustado, mesmo que por meio de uma "tradução" entre o modelo empregado para a tomada de decisão sobre os ajustes e o ajuste do software em si.

Nos tópicos abaixo serão descritos os conceitos que irão apoiar a representação da reconfiguração de um software como um problema de planejamento, bem como a execução do plano sobre o modelo de programação subjacente.

2.3.1 Linguagens para a Descrição Arquitetural

Uma Linguagem de Descrição Arquitetural (*Architectural Description Language* - ADL) é uma notação para modelar uma arquitetura de software (TAYLOR; MEDVIDOVIC; DASHOFY, 2009). Estes mesmo autores consideram que uma ADL deve possuir os seguintes elementos: Componentes, Conectores, Interface e Configurações. Um quinto elemento importante quando se trata de arquitetura de software é a razão de um determinado projeto, também chamada de *Rationale*. Este quinto elemento é voltado para comunicar informações e intenções entre os envolvidos num projeto e, por isso, é geralmente expresso por meio de linguagem natural.

Deste modo, a ADL é uma linguagem que pode ser empregada para descrever um software, ou seja, um determinado estado num diagrama de transição de estados. Na literatura há várias propostas de ADL e cada uma tem um nível de abstração diferente pois elas se destinam a modelar aspectos diferentes de uma arquitetura de software como, por exemplo, a linguagem SADEL descrita em (TAJALLI *et al.*, 2010) e que possui conceitos para representar variáveis de estado pertencentes a componentes. Apesar de haver ADLs com capacidade de descrever arquiteturas dinâmicas como a π-ADL (OQUENDO, 2004), onde as mudanças na arquitetura dependem dos dados de entrada, nesta e em outras ADL com a mesma capacidade, a descrição de uma modificação é feita em tempo de projeto, ou seja, o desenvolvedor "programa" como uma arquitetura pode transitar entre diferentes estados do diagrama de transição.

Porém, quando se trata de obter um plano de reconfiguração para transitar entre duas configurações quaisquer, o mecanismo de decisão deve descobrir quais as mudanças que devem ser feitas, o que é diferente do modelo programático anterior. A descoberta do plano de reconfiguração é feita a partir do domínio Σ e faz-e necessário descrever este domínio por meio de alguma linguagem. Além disso, uma série de restrições e invariantes a estas alterações podem ser necessárias e devem ser consideradas na geração do plano de reconfiguração. A necessidade de descrever o domínio já foi reconhecida na literatura, mas não sobre a ótica do problema de planejamento, mas sim do ponto de vista arquitetural. Em (OREIZY; MEDVIDOVIC; TAYLOR, 1998), os autores mencionam que as ADLs não são suficientes para apoiar arquiteturas dinâmicas, sendo necessárias extensões ou outras linguagens que devem: a) descrever as modificações - ou, como definido pelos autores, uma linguagem de modificação arquitetural (*Architecture Modification Language* - AML); e b) uma linguagem para descrever as restrições sob as quais estas modificações devem ser executadas, denominada de *Architecture Constraint Language* (ACL). Assim, considerando o diagrama de transição de estados, estas linguagens poderiam ser empregadas da seguinte forma:

- ADL: descrever um estado, ou seja, uma configuração do software.
- AML: descrever as ações que alteram um estado, isto é, que alteram a configuração.
- ACL: descrever as restrições arquiteturais que restringem as configurações e as ações de reconfiguração.

Assim, teríamos os estados inicial e o objetivo descritos por meio de uma ADL, e o planejador empregaria um modelo de ações - descrito por meio de uma AML e uma ACL - para descobrir um plano de reconfiguração descrito em AML. Após o planejador determinar um plano de reconfiguração, este deverá ser executado sobre o software e, como este plano está no nível arquitetural, torna-se necessário traduzi-lo para o nível da implementação. O paradigma de desenvolvimento

que mais se aproxima do nível de abstração arquitetural é o *Desenvolvimento Baseado em Componentes* (DBC), que será visto a seguir. Um exemplo de AML para este paradigma são as linguagens FScript e FPath, descritas em (DAVID *et al.*, 2009), empregadas juntamente com o modelo de componentes Fractal (BRUNETON *et al.*, 2006). No trabalho, os autores propõem a FPath como forma de navegar numa determinada configuração de um software, isto é, poder introspectar sobre a sua arquitetura. Já a FScript, é empregada para executar ações de reconfiguração no nível do modelo de componentes Fractal.

2.3.2 Desenvolvimento baseado em componentes

A partir da decisão tomada no nível arquitetural, cabe agora traduzi-la para o nível de implementação do software, ou seja, executar as ações do plano de reconfiguração. Deste forma, considerando ações de adaptação como adicionar e remover um componente ou associar e desassociar componentes e conectores, a plataforma de implementação do sistema deve ser capaz de executá-las. Nesta pesquisa, tem-se interesse em como executar os ajustes decididos num nível mais elevado do que a implementação e que características o software deve possuir para permitir a execução destes ajustes.

Considerando a arquitetura de um software, em (TAYLOR; MEDVIDOVIC; DASHOFY, 2009), os autores citam que relacionar arquitetura a 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, este *arcabouço* possui muitas similaridades com o conceito de *middleware* ou *modelo de componentes*. Os autores mencionam

que a similaridade é decorrente da influência mútua entre um estilo arquitetural e um *middleware*. Esta influência foi citada em (NITTO; ROSENBLUM, 1999), onde argumenta-se que o emprego de um *middleware* pode influenciar a arquitetura de um sistema desenvolvido, bem como as escolhas arquiteturais restringem a seleção do *middleware* subjacente utilizado na fase de implementação.

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), mas segundo Szyperski este modelo não teve sucesso como um meio de tornar mais eficiente o desenvolvimento de sistemas. Do ponto de vista de adaptação de sofware, o Desenvolvimento Baseado em Componentes (DBC) é uma alternativa para apoiar a adaptação composicional, como citado em (MCKINLEY *et al.*, 2004), podendo ser empregado para a composição dinâmica, onde o desenvolvedor pode adicionar, remover ou reconfigurar os componentes em tempo de execução. Segundo Lau e Wang em (LAU; 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; 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, Councill e Heineman, em (COUNCILL; HEINEMAN, 2001), 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."

Assim, o DBC é uma forma de efetuar este mapeamento entre ajustes arquiteturais e a plataforma de implementação. Porém, o modelo de componentes deve ser capaz de fornecer os mecanismos para que estes ajustes sejam feitos em tempo de execução, ou seja, deve permitir reconfigurações dinâmicas. Isto leva ao emprego de mecanismos de apoio ao modelo de componentes que forneçam a capacidade de *reflexão computacional* que, segundo Kon, em (KON *et al.*, 2002), 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 por meio dos sensores, ou seja, as suas *percepções* e execute as ações de reconfiguração decididas por ele por meio dos *atuadores*.

2.3.3 Consistência da reconfiguração

No caso de uma reconfiguração de um software, cabe definir o que vem a ser consistência. Em (CHENG *et al.*, 2009), os autores tratam de garantia ou *assurance*, que significa prover evidências de que o conjunto de propriedades funcionais e não-funcionais são satisfeitas durante a operação do sistema. Para os autores, o software e o contexto em que ele está inserido se modificam ao longo do tempo e, a verificação destas propriedades se baseia na existência de modelos. Num dado instante de tempo, um modelo deve refletir uma determinada configuração do software e contexto a fim de que as verificações necessárias possam ser realizadas. Entretanto, os autores não tratam da verificação da transição entre os diferentes estados ao longo do tempo.

Em (LÉGER; LEDOUX; COUPAYE, 2010), os autores consideram que a consistência de uma reconfiguração se baseia em dois elementos: 1) *restrições de integridade*, que representam combinações de elementos arquiteturais e estado de componentes e, 2) *pré* e *poscondições* das transições. Este modelo de restrições é dividido em três 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.

Assim, os autores argumentam que a verificação de uma reconfiguração pode ser feita verificando-se as pré e poscondições das operações em relação as restrições de integridade, garantindo que as operações não violam estas restrições. No trabalho, as verificações são feitas traduzido-se as restrições e operações para uma linguagem baseada em lógica utilizada pela ferramenta Alloy Analyzer [2]. Deste modo, as operações de reconfiguração podem ser simuladas e o desenvolvedor pode saber se um determinado conjunto de operações é válido sobre o modelo de restrições, garantindo que a sequência de operações é válida.

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

Entretanto, este processo é manual e o desenvolvedor precisa verificar o conjunto de operações de reconfiguração sobre estas restrições que agora são apoiadas por uma ferramenta. Ou seja, ainda é o desenvolvedor que gera o plano de reconfiguração e ele precisa verificar este plano sobre as restrições. Num processo de reconfiguração mais autônomo e automatizado, a garantia de que o plano é consistente deve ser feita sem a participação humana. Em (SYKES, 2010), o autor propõe um mecanismo automatizado em que estas restrições são verificadas após a geração do plano de reconfiguração. Se o plano gerado não atender às restrições ele é descartado ou pode ser refinado a fim de se obter um plano válido perante as restrições. No trabalho, o refinamento é feito a partir do plano original e são feitas recombinações das ações deste plano para verificar se algum dos novos planos gerados atende às restrições. Entretanto, estas recombinações não possuem qualquer heurística que leva a geração de planos válidos e o número de possíveis combinações cresce com o comprimento do plano gerado.

Este mesma verificação de pré e poscondições é realizada na geração do plano feita pelo algoritmo de planejamento, conforme descrito anteriormente na Seção 2.2.3. Deste modo, a geração de um plano feita por um algoritmo de planejamento está de acordo com o conceito de consistência de reconfiguração de software proposto em (LÉGER; LEDOUX; COUPAYE, 2010), e que será adoto nesta pesquisa.

2.3.4 Dependências

Um componente de software pode depender de outros componentes, de recursos do sistema operacional, do *middleware* subjacente ou mesmo de outras bibliotecas. Por exemplo, um componente que precise expor ou consumir um serviço web, provavelmente irá depender de bibliotecas como a Apache Axis [3], que implementa o protocolo SOAP, pois este componente que requer esta biblioteca dificilmente terá todas as suas dependências resolvidas internamente. Deste modo, o mecanismo responsável pelo ajuste do software deve considerar que esta dependência deve estar resolvida antes de efetuar a reconfiguração propriamente dita.

Para isto, deve-se decidir em que etapa esta verificação de dependência deve ser executada e como resolvê-la. Se considerarmos no momento do planejamento, o planejador deve:

1. possuir a informação de que um componente depende de um determinado recurso; e
2. ter capacidade deve verificar se esta dependência é verdadeira, pois ela será uma precondição para que um componente possa ser ativado.

[3] http://ws.apache.org/axis/

No caso 1, este informação deve estar disponível junto com o problema e codificada na linguagem empregada pelo planejador. Se o planejador puder fazer requisições a algum outro sistema durante o planejamento, então pode-se pensar num repositório onde ele fará estas consultas e receberá como resposta o que deve ser verificado. De posse da informação obtida no item 1, o planejador deve verificar se ela é satisfeita, ou seja, se a dependência está disponível. Esta verificação também deve ser feita no controlador, que é o componente responsável por efetivar as mudanças no software. Neste caso, o planejador deverá incluir uma ação que sirva para verificar se a dependência está satisfeita, ou seja, uma *ação de sentir*, que não altera um estado, mas serve para identificar um estado no qual a dependência está ou não satisfeita. No caso dela não estar satisfeita, o controlador pode retornar um erro, ou ter como solicitar esta biblioteca para que ela esteja disponível para o componente a ser ativado.

2.4 Considerações

Neste capítulo foram revistas as maneiras de se entender o mecanismo de controle responsável por ajustar a configuração arquitetural de um software. Este mecanismo pode ser generalizado pela visão de agentes inteligentes e pelos diferentes modos em se construir um agente que irá atuar como o controlador do software, agindo sobre a sua configuração. Esta visão de agente engloba o sistema de controle com realimentação, bem como serviu de inspiração para a visão de computação autonômica. Dentre os diferentes tipos de agentes, o agente baseado em modelos pode ser uma solução ao problema de geração de um plano de reconfiguração e que considere os aspectos relacionados ao modelo de programação subjacente, gerando um plano correto do ponto de vista do modelo.

Em relação a execução do plano em si, ou seja, da efetiva reconfiguração do software, o modelo de programação subjacente deve ser capaz de representar os conceitos que compõem o plano de reconfiguração, bem como prover capacidade reflexiva para executar este plano. Em outras palavras, o plano deve ser traduzido e executado no nível do modelo de programação. O DBC é um dos modelos de programação que apoiam a adaptação composicional do ponto de vista dos conceitos empregados para a geração do plano e, para a execução do plano, o modelo de componentes deve prover uma capacidade reflexiva.

Na Seção 3, serão vistos trabalhos que empregam agentes baseados em modelos na tarefa de reconfigurar um software, mas que não consideram uma série de aspectos diretamente relacionados com o modelo de componentes empregado. Também nesta mesma Seção, serão vistos alguns trabalhos que tratam de fornecer ao modelo de programação subjacente capacidade reflexiva, permitindo que o mesmo possa ser adaptado em tempo de execução.

Capítulo 3

Trabalhos relacionados

Depois de rever alguns conceitos ligados a Engenharia de Software para sistemas autoadaptáveis, neste Capítulo, serão descritas as abordagens identificadas na literatura que empregam a adaptação composicional. Conforme dito anteriormente, o interesse reside nas soluções que prescindem da participação do desenvolvedor nesta tarefa, permitindo que o mecanismo de geração desta reconfiguração tenha um maior grau de autonomia.

As propostas identificadas empregam a abordagem baseada em modelos - que na visão de agentes da Seção 2.1.2 são os agentes baseados em objetivos ou utilidade - para decidir sobre o plano de reconfiguração. Como poderá ser visto, cada abordagem representa o modelo de uma forma diferente, seja pelo nível de detalhes ou pelas características que devem ser levadas em consideração numa alteração de configuração e o plano de reconfiguração é expresso no nível do modelo de componentes subjacente, ou seja, os elementos arquiteturais empregados na descrição do modelo são os mesmos que o modelo de componentes utiliza, não havendo um mapeamento ou tradução entre os diferentes níveis de abstração.

Decidido qual o plano, a etapa seguinte é executá-lo sobre o software, ou seja, efetivar os ajustes descritos no plano. Das três abordagens identificadas, apenas duas efetuam os ajustes e, dessas duas, ambas empregam modelos de componentes próprios, onde controlam o ciclo de vida dos componentes e efetuam as conexões necessárias, bem como cada modelo tem a sua própria semântica de componente e linguagem de implementação próprias. É no nível do modelo de componentes que as alterações são feitas e o emprego de modelos próprios dificulta a tarefa de utilizar as soluções em aplicações que não se baseiam nestes mesmos modelos, pois o sistema de transição de estados é derivado do modelo subjacente, ou seja, as ações que alteram uma configuração são decorrentes do modelo, pois ele deve permitir que ela seja executada.

As abordagens de adaptação arquitetural empregam uma visão de cima para

baixo, onde decisões no nível arquitetural são mapeadas para decisões de mais baixo nível. Seguindo na direção inversa, ou seja, de baixo para cima, há propostas de modelos de componentes que provêm apoio a reconfiguração dinâmica. Deste modo, também serão analisadas propostas sobre estes modelos de componentes com o objetivo de diminuir a lacuna entre a adaptação composicional no nível arquitetural e a sua execução sobre o software.

Os seguintes aspectos sobre as abordagens a serem revisadas foram destacados:

- na descrição do sistema de transição de estados $\Sigma = (S, A, E, \gamma)$;
- no estado inicial;
- nos objetivos;
- no plano que será executado pelo controlador; e
- no controlador que executa o plano.

Neste capítulo, são revisadas abordagens identificadas na literatura que empregam a adaptação composicional no nível arquitetural. Destas abordagens, o interesse é sobre o mecanismo que gera a reconfiguração do software bem como a sua posterior execução. Na Seção 3.1, são descritas as abordagens baseadas em modelos encontradas e na Seção 3.2 são revisados os conceitos sobre modelos de componentes que apoiam a reconfiguração dinâmica. E, por fim, na Seção 3.3, são feitas as considerações sobre problemas ainda não resolvidos pelas soluções encontradas.

3.1 A Atividade de Planejar na Adaptação Arquitetural

Segundo (HUEBSCHER; MCCANN, 2008), em sentido amplo, a atividade *Planejar* envolve considerar os dados monitorados a partir dos sensores, a fim de produzir uma série de mudanças a serem executadas sobre o Elemento Gerenciado. É nesta atividade que são tomadas as decisões sobre o que deve ser ajustado na configuração arquitetural do software, com o objetivo de que ele venha a atender às alterações contextuais.

As abordagens descritas a seguir empregam um modelo, ou melhor, um Sistema de Transição de Estados (Σ), sobre como o software pode ser reconfigurado. Com este modelo, um algoritmo é utilizado para buscar por um plano de reconfiguração que leve o software à configuração arquitetural objetivo.

3.1.1 Reconfiguração Dinâmica para Sistemas Distribuídos

Apesar desta proposta não efetuar a execução do plano, ela foi a primeira proposta identificada na literatura que emprega a abordagem baseada em modelos. Em (ARSHAD; HEIMBIGNER; WOLF, 2003) e (ARSHAD; HEIMBIGNER; WOLF, 2007), Arshad et al. propõem uma abordagem para implantar (*deploy*) e reconfigurar um sistema distribuído. O trabalho se preocupa em gerar um plano, sem executá-lo sobre um sistema, considerando os tempos de execução das respectivas ações.

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 A deve estar conectado sem especificar com o que este componente estará conectado. A configuração explícita descreve explicitamente os artefatos e suas configurações, empregando predicados em pares para descrevê-la. Por exemplo, um componente que está conectado a um conector é descrito por dois predicados: um que representa que o componente está conectado e um outro que significa dizer a quem este componente está conectado.

Definição do domínio

O domínio é descrito em PDDL 2.1 (FOX; 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. Aliás, a busca por um plano ótimo reside naquele cuja execução se dará no menor tempo, de acordo com as funções utilidades descritas no domínio, as quais descrevem o tempo consumido em cada operação, e permitem identificar qual o melhor plano. As ações alteram os estados de componentes, conectores e máquinas. Um conector ou componente pode estar: INATIVO, ATIVO, CONECTADO ou DESLIGADO (*killed*). Já a máquina pode estar: EM BAIXA (*down*), EM CIMA (*up*) ou DESLIGADA (*killed*).

Uma configuração arquitetural e o estado dos elementos que formam uma configuração são descritas 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.

Definição do problema

Na abordagem, um elemento de software denominado de *Planit* planeja novas configurações de software em duas ocasiões: a) para a implantação inicial dos componentes e conectores nas respectivas máquinas; e b) quando ocorre algum problema no sistema e o mesmo deve ser dinamicamente reconfigurado. Em ambos os casos, o objetivo é descrito numa linguagem no nível proposicional.

No primeiro caso, o estado inicial descreve a lista de artefatos e os fatos sobre estes artefatos relacionados ao fator tempo, como tempo para iniciar, tempo para conectar e tempo para uma máquina estar rodando. 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. Este estado pode ser dado de forma explícita ou implícita e, caso alguma informação não seja fornecida, o mecanismo emprega uma configuração implícita.

No segundo caso, quando ocorre 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, que irá gerar uma série de planos cada vez "melhores", segundo as métricas tamanho do plano e tempo total de execução.

Na geração deste estado objetivo, alguns artefatos específicos possuem um *plano de contingência* que descreve como e onde reiniciar este artefato. Os planos de contingência estão previamente armazenados num mecanismo próprio e não podem ser modificados ao longo da operação do sistema e, para os artefatos que o possuem, o estado objetivo incluirá a configuração explícita fornecida pela contingência.

Geração do plano de reconfiguração

A geração do plano de implantação e reconfiguração é feita por meio de um planejador da área de IA - 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. Estas funções consideram aspectos temporais relacionados a aplicação, como o intervalo de tempo gasto para conectar um componente a um conector que estão na mesma máquina, e o plano gerado descreve quanto tempo cada uma das ações a ser executada consome, bem como o tempo total para a execução do plano.

3.1.2 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. 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 superior - *Camada de gerenciamento de objetivos*.

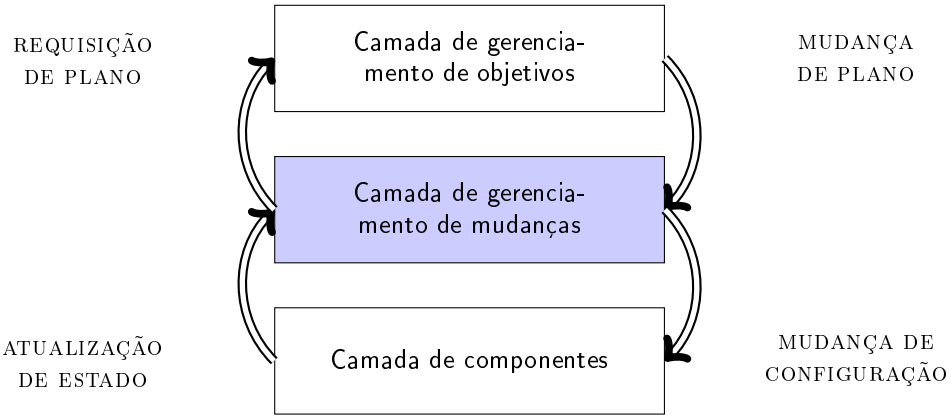


Figura 3.1: Modelo de referência em três camadas

Definição do domínio

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.

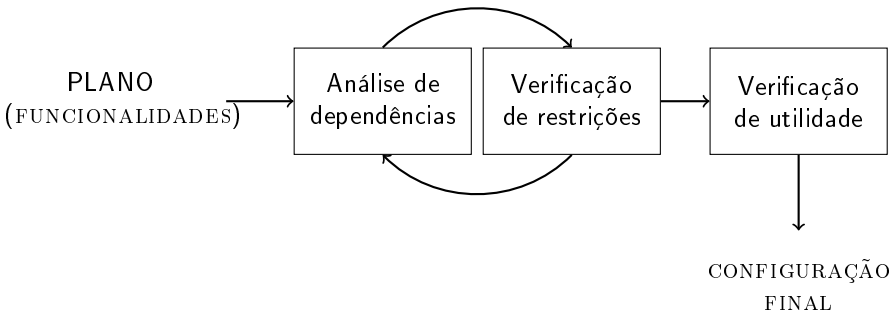


Figura 3.2: Processo de geração agregado proposto em (SYKES, 2010).

Definição do problema

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 no plano representam as funcionalidades que o software a ser configurado deve possuir para atender aos requisitos da aplicação. O processo de montagem de uma configuração - ilustrado na Figura 3.2 - começa na determinação destas funcionalidades.

Cada ação deste plano possui um mapeamento para uma interface de componente de software, ou seja, a interface determina qual o componente de software que pode ser utilizado para satisfazer um requisito funcional. Deste modo, o plano recebido da *Camada de Gerenciamento de Objetivos* constitui o estado objetivo a ser obtido.

Geração do plano de reconfiguração

A partir do estado objetivo descrito no parágrafo anterior, as dependências entre os componentes - descritas por meio de interfaces requeridas - 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. Esta verificação é similar a verificação de consistência tratada na Seção 2.3.3.

A verificação de restrições é feita programando-se na linguagem Prolog, ou melhor, o desenvolvedor deve descrever um programa em Prolog que verifique as restrições que ele deseja. Se uma determinada configuração não é válida perante uma restrição, o mecanismo tenta estender a configuração para tentar torná-la válida antes de descartá-la e verificar a próxima candidata. Depois disso, para cada configuração, 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, a que possuir o menor número de componentes será a escolhida.

3.1.3 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*.

No *Planejador da Aplicação*, o desenvolvedor fornece um *Problema* descrito

na linguagem NPDDL (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.

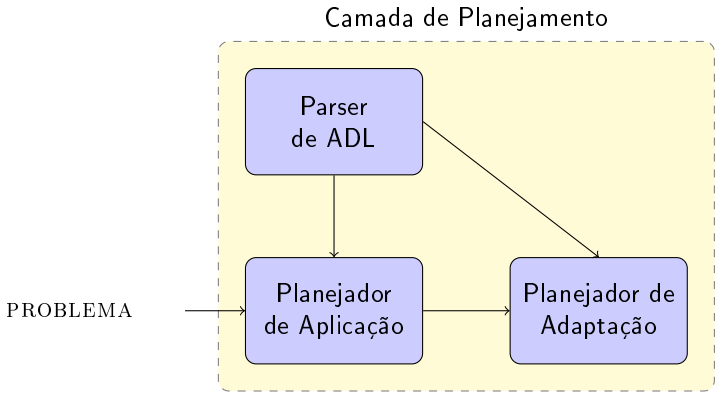


Figura 3.3: Camada de Planejamento - adaptado de (TAJALLI *et al.*, 2010)

O estilo arquitetural empregado pelo software a ser executado no robô, ilustrado na Figura 3.4, é similar a uma arquitetura orientada a serviços, 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.

Definição do domínio

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 es-

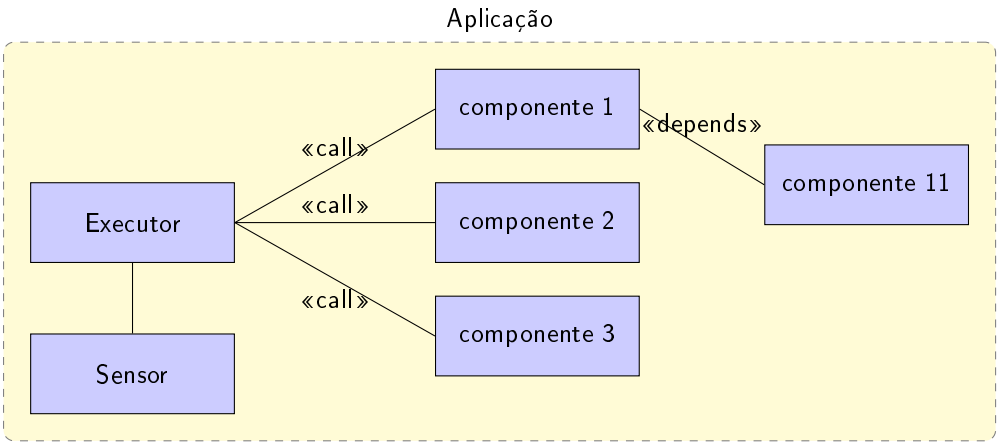


Figura 3.4: Arquitetura da aplicação a ser executada no robô - adaptado de (TAJALLI *et al.*, 2010)

pecifica 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

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 do plano de 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 a falha de algum componente da aplicação, que irá comprometer a execução do plano pelo *Executor*. Neste caso, a falha 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.

3.2 A Atividade de Executar na Adaptação Arquitetural

Gerado o plano de reconfiguração de um software, cabe agora executar este plano. Em (SALEHIE; TAHVILDARI, 2009), os autores citam que a atividade de *executar* é a responsável por aplicar as ações determinadas na tomada de decisão. Na visão de agente inteligente, este age sobre o ambiente por meio de atuadores e cabe agora levantar as técnicas para implementar um atuador num software adaptável. Em (SALEHIE; TAHVILDARI, 2009), os autores listam uma série de soluções como: padrões de projeto, protocolos de meta-objeto, orientação a aspectos, ponteiros de função e *middleware*. Já em (MCKINLEY *et al.*, 2004), os autores mencionam que nas abordagens composicionais, as tecnologias que apoiam a adaptação contêm um nível de indireção para interceptar e redirecionar interações entre entidades do programa. Destas tecnologias, os autores consideram que: reflexão computacional, DBC e separação de interesses (*Aspect Oriented Programming (AOP) - separation of concerns*) são tecnologias chave para o projeto de software reconfigurável.

Como foi visto antes, o DBC por si só não garante que uma determinada configuração possa ser alterada em tempo de execução e as características da reflexão computacional complementam o DBC nesta tarefa. Deste modo, modelos de componentes que tenha capacidade de reflexão computacional, só que agora no nível do modelo de componentes, podem facilitar a tarefa de desenvolver sistemas reconfiguráveis. Assim, do ponto de vista do modelo de componentes faz-se necessário determinar os atributos que representam uma configuração neste modelo que será obtida por meio da *introspecção* e as operações que permitirão alterar uma configuração, ou seja, a *reconfiguração*. No contexto de uma aplicação com capacidade de auto-cura ou auto-reparo (*self-repair*), em (SICARD; BOYER; PALMA, 2008), os autores argumentam que um modelo de componentes deve possuir cinco abstrações, ou meta-objetos, para apoiar de maneira eficiente uma aplicação com esta capacidade, a saber:

1. Atributos - os atributos de um componente representam as suas propriedades configuráveis, por exemplo, a porta onde um servidor que recebe requisições via *socket* ficará escutando.

2. Ciclo de vida - o estado do ciclo de vida de um componente, ou seja, iniciado,

implantado, parado, etc.

3. Interfaces - as interfaces representam as dependências funcionais do componente em relação a outros componentes, e.g., dois componentes que possuam cada um uma interface provida e requerida de um tipo compatível podem ser conectados entre si.

4. Enlace (*Binding*) - representam as conexões entre componentes que possuem dependências funcionais.

5. Composição - representam as relações de composição entre componentes.

A partir das abstrações e da sua semântica, os autores mencionam que dois tipos de meta-operações são necessárias, as quais possuem uma direta correlação com as características de reflexão computacional:

1. Introspecção - quando o modelo de componentes provê meios de recuperar as abstrações acima, ou seja, seu estado atual.

2. Reconfiguração - quando o meta-dado dos componentes, representado pelas abstrações anteriores, pode ser modificado em tempo de execução, permitindo alterar o estado da aplicação.

Desta forma, um modelos de componentes deve possuir meta-objetos que representem o seu estado e meta-operações que atuam sobre os meta-objetos. Dentre as abordagens revisadas na Seção 3.1, apenas as citadas por (SYKES, 2010) e (TAJALLI *et al.*, 2010) executam o plano gerado e os detalhes sobre esta execução são aqui tratados. Além disso, também é descrita uma abordagem de adaptação arquitetural que, apesar de empregar uma abordagem programática para a geração do plano, considera uma série de aspectos a serem observados durante a execução do plano similares as descritas para o controlador tratado na Seção 2.2.4.

3.2.1 Adaptação e Montagem Arquitetural Autônoma

Nesta abordagem, os meta-objetos empregados descrevem as interfaces de cada componente e não há representação dos atributos dos componentes. Quanto ao ciclo de vida, este é considerado no protocolo que efetua a remoção e troca de componentes, denominado na abordagem de *tranquilidade*. Este protocolo pode *pausar* um componente aguardando o momento em que as mudanças possam ser aplicadas sem comprometer a segurança da aplicação. Apesar dos atributos dos componentes não serem representados, o trabalho menciona que o estado de um componente pode ser enviado a um outro componente quando há uma troca entre componentes.

Quanto aos enlaces, eles estão implicitamente discriminados pois considera-se que as interfaces são únicas e, por isso, se dois componentes ocorrem numa configuração e eles possuem interfaces requeridas e providas de tipo compatível, então eles devem ser conectados. Segundo o autor, esta simplificação na forma de discriminar os enlaces serve para reduzir a complexidade do algoritmo que gera o plano de reconfiguração. Componentes compostos podem ser descritos no repositório mas, ao considerar estes componentes, o algoritmo de montagem "desmonta" o componente composto nos seus constituintes, "achatando" a hierarquia de componentes e considerando apenas os constituintes e não mais o composto.

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

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.

3.2.3 RAINBOW

Apesar de ser uma abordagem baseada em regras, o arcabouço RAINBOW se preocupa com as tarefas de execução de uma adaptação e é um bom exemplo a ser tratado aqui sobre a atividade de execução da reconfiguração. O arcabouço RAINBOW, proposto em (GARLAN *et al.*, 2004), é uma malha de controle para a auto-adaptação de um software. Ele usa um modelo arquitetural da aplicação para acompanhar as propriedades de execução de um sistema, avaliar se está ocorrendo alguma violação das restrições e, caso um problema ocorra, efetuar adaptações globais ou no nível de módulos no sistema em execução. Estas adaptações estão descritas por meio de estratégias e táticas que na verdade são regras

utilizadas numa abordagem programática.

Neste arcabouço, há uma camada de *infraestrutura de sistema* que contêm três elementos: *atuadores*, *sensores* e *descoberta de recursos*. Esta camada fornece uma interface reutilizável para acessar os atuadores, pois as operações de reconfiguração enviadas são traduzidas para o atuador específico do tipo do componente. Os *atuadores* são específicos dos componentes, pois um atuador desempenha operações de adaptação em um tipo de componente, como um gateway de videoconferência, onde o atuador pode ativar e desativar este gateway. Os sensores podem medir o tempo de resposta, carga e banda de vários componentes do sistema e monitoram a informação que é usada nos níveis mais altos da infraestrutura para atualizar as propriedades do modelo, pois as estratégias e táticas podem fazer uso destas informações para decidir sobre o que deve ser feito, ou seja, a regra a ser disparada. O mecanismo de adaptação precisa de um serviço de descoberta de recursos para poder determinar que componentes estão disponíveis para substituir outros componentes, conforme for determinado pelas estratégias de adaptação. Este serviço de descoberta permite descobrir recursos baseados no tipo de um componente e nos seus atributos, pois algumas estratégias empregam aspectos não-funcionais do sistema.

As ações de reconfiguração são denominadas de operadores e uma *tática* contém uma série destes operadores formando uma pacote de mudanças ou os passos de uma adaptação. Uma tática define uma série de *efeitos* que devem ser observados após ela ser executada. Se estes efeitos forem verdadeiros, o sistema considera que a tática foi realizada com sucesso, ou seja, o ajuste a ser feito no sistema em execução foi realizado. Do ponto de vista do controlador estes efeitos servem para informar que o estado desejado após a execução deste pacote de operadores foi atingido. Apesar das táticas poderem ser aninhadas, o autor cita que este uso pode gerar ciclos e comprometer a avaliação das condições de aplicabilidade da táticas bem como dos seus efeitos. Assim, a sequencia dos operadores contida numa tática pode ser vista como o plano de reconfiguração que é executado pelo controlador e, após este identificar o estado desejado descrito nos efeitos, ele considera que a reconfiguração foi feita com sucesso. Caso contrário, o mecanismo informa ao sistema uma falha na adaptação.

3.2.4 Reconfiguração no Modelo Fractal

As abordagens que tratam das reconfigurações no nível arquitetural deixam de considerar aspectos de mais baixo nível do modelo de programação subjacente. Desta forma, nos trabalhos que tratam de modelos de componentes reflexivos pode-se encontrar uma série de aspectos ligados a execução das reconfigurações e em especial da confiabilidade destas sobre o software adaptável. O modelo de componentes Fractal possui uma linguagem específica para realizar as reconfi-

gurações e o mecanismo de execução desta linguagem apoia a confiabilidade da reconfiguração. Em (DAVID *et al.*, 2009), os autores descrevem as propriedades Atomicidade, Consistência, Isolamento e Durabilidade, conhecidas também pelo acrônimo **ACID**, para a reconfiguração de software.

- Atomicidade - ou o sistema é reconfigurado e a transação da reconfiguração é efetivada (*commit*), quando todas as operações que formam a transação são executadas, ou ele não é e a transação é abortada. Se uma transação de reconfiguração falha, o sistema retorna a um estado consistente prévio como se a transação nunca tivesse ocorrido.
- Consistência - uma transação de reconfiguração é uma transformação válida de um estado de sistema, i.e., ela leva o sistema de um estado consistente a um outro estado também consistente. Um estado é consistente se e somente se ele está em conformidade com um critério: não viola o modelo de componentes e possivelmente outras restrições mais específicas.
- Isolamento - as transações de reconfiguração são executadas como se elas fossem independentes, Os resultados das operações de reconfiguração dentro de uma transação não efetivada não são visíveis de outras transações até que a transação seja efetivada, ou nunca serão caso seja abortada.
- Durabilidade - os resultados de uma transação de reconfiguração efetivada são permanentes: uma vez que a transação de reconfiguração seja efetivada, o novo estado do sistema é persistido de forma que ele possa ser recuperado em caso de falhas mais graves, e.g., falhas de hardware.

Estas propriedades são consideradas pelo Fractal e em especial, no mecanismo de execução da linguagem FScript. A *Atomicidade* é garantida pela FScript, pois todas as ações ou funções (grupo de ações) demarcam uma transação ao serem chamadas. A *Consistência* é obtida por meio das validações descritas no item 2.3.3. O *Isolamento* é obtido pelo bloqueio de algumas operações ao se iniciar uma transação, e o bloqueio é liberado quando esta transação é efetivada. A *Durabilidade* é feita tanto para as transações de reconfiguração, quanto para a aplicação como um todo. Cada transação é registrada num log e segue o formato das linguagens FScript e FPath e, frequentemente, o estado da aplicação é registrado, formado pela configuração de software e pelos atributos dos componentes.

Estas propriedades estão intimamente ligadas ao mecanismo responsável por executar a reconfiguração sobre o software. Porém, com relação a Consistência esta também pode ser considerada na geração do plano de reconfiguração. Por exemplo, se for adotada a abordagem programática, as ações descritas nas regras podem ser verificadas sobre um modelo do sistema e, se for adotada a abordagem baseada em modelos, o próprio modelo pode conter as restrições de consistência

que devem ser obedecidas ao se gerar um plano. As demais propriedades devem ser atributos do mecanismo **Controlador**, ilustrado na 2.4.

3.3 Considerações

As abordagens anteriores, descritas na Seção 3.1, tratavam o problema de adaptação num nível mais elevado, mas deixaram de considerar aspectos ligados ao modelo de programação no qual a aplicação foi desenvolvida, como os citados em algumas abordagens da Seção 3.2. Estes aspectos podem não ter sido considerados devido ao próprio domínio de aplicação empregado, que foi o de veículos autônomos tanto em (SYKES, 2010) quanto em (TAJALLI *et al.*, 2010) ou ao próprio modelo de componentes utilizado. Além da simplificação proveniente do domínio, o problema de reconfiguração também possuía uma complexidade menor pois os modelos de componentes empregados permitiam abstrair uma série de questões como:

- prescindir do controle do ciclo de vida.
- ausência da capacidade de composição de componentes.
- poucas ou até mesmo ausência de restrições arquiteturais durante a geração do plano.
- simplificação das atividades relacionadas ao controlador.
- um mesmo tipo de conexão entre componentes.

Porém, estas simplificações podem não estar presentes nos modelos de componentes empregados ou mesmo nos domínios das aplicações a serem adaptadas, como as empregadas em comando e controle, onde conexões podem ter diferentes protocolos e serem síncronas ou assíncronas. Como será visto no Capítulo 4, o uso de um modelo de componentes reflexivo e com características mais complexas do que as apresentadas nas abordagens tratadas até aqui, representa um aumento na complexidade na reconfiguração a ser feita. Além disso, a efetiva execução do plano de reconfiguração, levanta uma série de questões como a confiabilidade da reconfiguração, expressas pelas propriedades **ACID** descritas anteriormente para a reconfiguração de um software.

Capítulo 4

Reconfiguração de Software Automatizada

Neste Capítulo, é descrito como esta pesquisa propõe empregar os fundamentos descritos no Capítulo 2 para automatizar o processo de geração de um plano de reconfiguração e posterior execução deste plano sobre o software, dado que uma nova configuração é necessária. Devido a complexidade existente em algumas aplicações, decorrente do grande número de componentes que podem constituí-la bem como do modelo de programação subjacente, empregar uma abordagem programática irá restringir as possíveis adaptações a serem consideradas pelo desenvolvedor. Deste modo, para lidar com a complexidade inerente do problema e ainda evitar restrições na adaptação, esta automatização será obtida empregando-se uma abordagem baseada em modelos e que será apoiada pelo emprego de um planejador que deve gerar planos corretos em relação ao modelo do domínio. Uma outra premissa é o emprego de uma visão de mais alto nível que o da linguagem de programação e a capacidade de reflexão computacional e isto pode ser obtido por meio de um modelo de componentes reflexivo.

Antes de passar para descrição da proposta de solução propriamente dita, na Seção 4.1, é descrito um trabalho realizado no âmbito desta pesquisa que trata da adaptação de um software empregando a abordagem programática, com o intuito de mostrar as limitações em se empregar esta abordagem para a reconfiguração de um software. Na Seção 4.2, é descrita a proposta de solução desta pesquisa para automatizar a geração do plano de reconfiguração e posterior execução do mesmo sobre o software a ser ajustado. Para esta solução, é descrito o fluxo de execução entre os seus componentes e algumas exceções que podem ocorrer. Na Seção 4.2.1, é apresentada a modelagem de domínio e de problema para o emprego de um planejador, a geração dos planos e sua execução sobre um software que emprega o paradigma de DBC sobre um modelo de componentes reflexivo. Por fim, na Seção 4.3, são feitas as considerações sobre os problemas de reconfiguração executados na Seção 4.2.1 e alguns problemas que podem ser

levantados a partir disso.

4.1 Uso de uma abordagem programática para configuração arquitetural

A primeira experiência empregando um sistema autoadaptável foi feita no domínio da robótica e a plataforma robótica escolhida foi o conjunto LEGO MINDSTORMS NXT 2.0[1]. Cada conjunto contém um grupo de peças para a montagem de robôs, tais como: servo motores, sensor ultrassônico, sensor de toque e sensor de luz e cor. Além disso, outros sensores estão disponíveis para aquisição como bússola, som, Sistema de Posicionamento Global (GPS) e acelerômetro. O robô é controlado por uma unidade micro-processada, denomidada de *brick* NXT, onde até três motores e quatro sensores podem ser conectados e controlados. O *brick* também possui capacidade de conectividade por meio de uma conexão USB 2.0 e uma conexão sem fio Bluetooth. A fim de permitir maior flexibilidade no desenvolvimento de aplicações com o robô, o *firmware* original foi substituído pelo firmware[2] LEJOS NXJ 0.85 , o qual fornece uma máquina virtual Java miniaturizada e permite a implantação e execução de código Java previamente compilado num computador pessoal.

Neste domínio, o experimento foi feito sobre um robô, configurado como um veículo, com duas rodas na frente e duas atrás, dois servo motores, onde cada um controla uma das rodas da frente e, para realizar uma curva, um motor pode permanecer parado, enquanto o outro está em movimento, ou os dois motores ficam em movimento mas com direções opostas, fazendo o robô girar num ponto. Um terceiro motor foi conectado ao sensor de ultrassom, que tem capacidade de medir distâncias de até 2,5 metros, e o motor permite que este sensor possa ser apontado num arco de 290º, ou seja, se o sensor for alinhado com o eixo longitudinal do robô, ele pode ser girado 145º para cada um dos lados. Os servo motores fornecem a informação de ângulo e, por isso, quando acoplado ao ultrassom, pode-se obter a direção de algum objeto detectado por ele. Dois sensores de toque foram posicionados na frente do robô e eles tem a função de detectar a colisão com algum obstáculo. Um sensor de cor foi instalado voltado para baixo e faz a leitura da cor da superfície diretamente abaixo do robô e a comunicação com o PC é feita por meio do Bluetooth. O aspecto do robô pode ser visto na Figura 4.1.

Neste projeto, o agente que contém o ciclo **MAPE-K** é executado num PC e se comunica via Bluetooth com o robô. Esta comunicação é feita em ambos os sentidos, do PC para o robô e do robô para o PC. O robô envia periodicamente a

[1] http://mindstorms.lego.com/en-us/default.aspx
[2] http://lejos.sourceforge.net/nxj.php

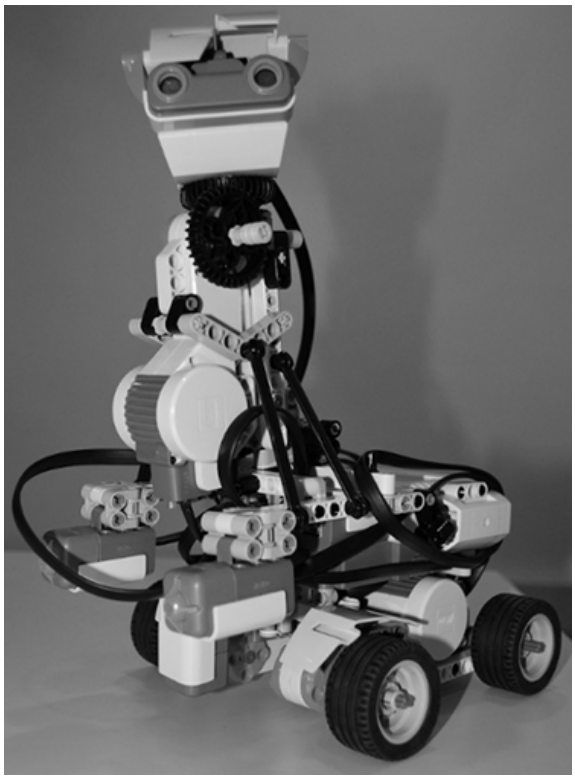


Figura 4.1: A configuração do robô.

sua posição - em coordenadas x e y - a distância e marcação lidos pelo sensor de ultrassom e a cor do piso que se encontra abaixo do robô. O agente pode solicitar a configuração de software atual, que representa as classes instanciadas e respectivas conexões entre as mesmas. O agente é responsável pela configuração do software que executa no robô e, esta configuração, possui dois momentos:

1. momento de iniciação - quando o robô não está executando o software e receberá a configuração correspondente a intenção do usuário.
2. momento de execução - quando mudanças no contexto impedem o robô de cumprir a intenção do usuário e ele deve ser reconfigurado.

A intenção do usuário é realizada por meio da seleção das características que o robô deve possuir. O diagrama de características, descrito na Figura 4.2, mostra as características e os diferentes relacionamentos existentes entre elas. A partir deste diagrama, é possível obter as diferentes combinações de características e, consequentemente, configurações de software por meio de um mapeamento, como proposto em (TRINIDAD *et al.*, 2007), onde as características no nível das folhas, ou seja, características que não são decompostas, são mapeadas de/para componentes de software num relacionamento um para um.

A escolha do usuário é feita em níveis mais altos do diagrama e as características que podem ser escolhidas estão assinaladas no diagrama da Figura 4.2,

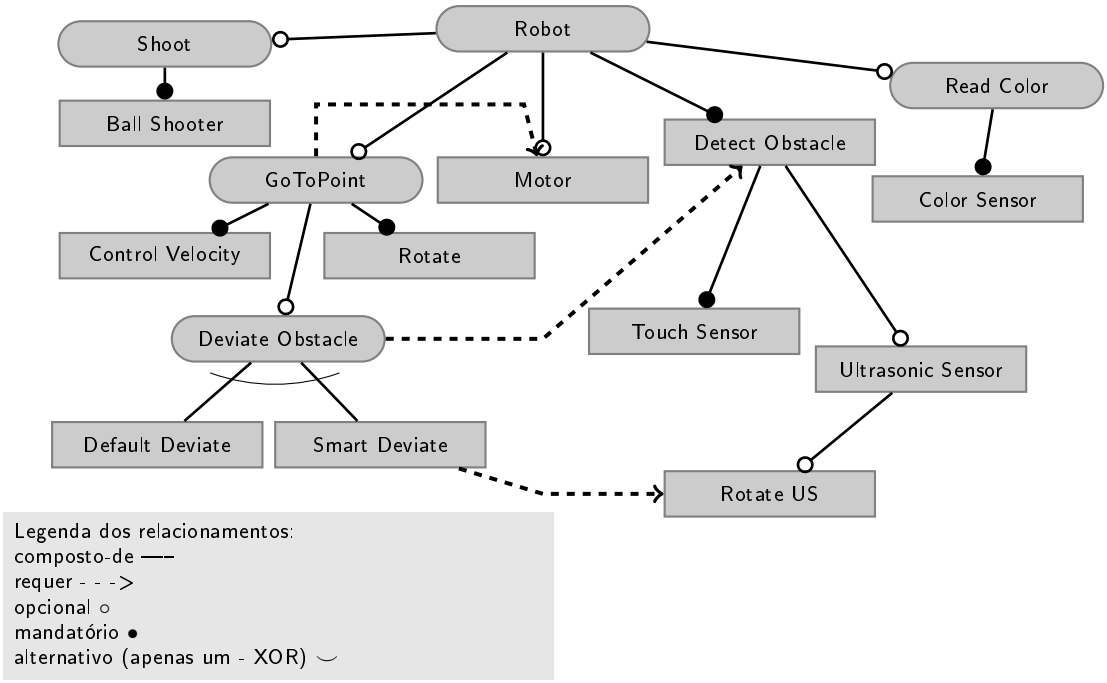


Figura 4.2: Diagrama de características para o robô.

por meio de retângulos arredondados. São elas que o usuário pode escolher para gerar a configuração inicial do robô e as dependências entre as características de mais baixo nível, sejam elas composicionais ou funcionais, serão resolvidas pelo mecanismo de tomada de decisão descrito a seguir. A vantagem de se empregar um nível mais alto, é que o usuário tem menos restrições para trabalhar e não precisa lidar com os relacionamentos de mais baixo nível, reduzindo a complexidade desta tarefa, porém ele tem menos possibilidades de escolha. Feita a escolha das características, este conjunto é passado para um sistema especialista, onde um conjunto de regras é armazenado para descrever os relacionamentos entre as características. Estas regras servem para aliviar a complexidade do desenvolvedor, pois elas resolvem as dependências de mais baixo nível.

O sistema especialista empregado foi o Drools[3], também conhecido como JBoss Rules. Os relacionamentos descritos por meio de regras são obtidos a partir do diagrama de características e servem para sanar as dependências entre as características, pois, como foi dito antes, o usuário escolhe características de alto nível para não ter que se ocupar de resolver as dependências de mais baixo nível.

Neste experimento, duas formas de empregar o sistema especialista foram consideradas, mas apenas a segunda delas foi empregada. Na primeira, cada possível configuração do robô seria descrita por meio de uma regra e o sistema deveria identificar qual a regra correspondente a configuração escolhida. Identificada a regra - cujo antecedente seria verdadeiro para a configuração escolhida

[3]http://www.jboss.org/drools

- o seu consequente iria discriminar quais as dependências de mais baixo nível que deveriam ser sanadas. Para isto, seria necessário gerar todas as possíveis configurações e depois obter as regras correspondentes, o que poderia ser feito por algum procedimento automático, caso o conjunto de características tivesse um tamanho razoável. Neste caso, se o diagrama sofresse alguma alteração nas características todo o conjunto de regras deveria ser gerado novamente. Além disso, se fosse considerado a possibilidade de que um novo componente de software pudesse ser disponibilizado, as regras também deveriam ser novamente geradas, pois este novo componente deveria ser mapeado para as características que ele representa, bem como critérios de seleção de um componente deveriam também ser descritos, para permitir a decisão de qual componente será empregado caso uma característica mapeada para mais de um componente seja selecionada.

Na segunda forma, que foi a empregada, cada relacionamento do diagrama foi descrito por uma regra. Por exemplo, na regra **1**, abaixo ilustrada na Figura 4.3, e utilizando a sintaxe Drools, caso o usuário tenha selecionado a característica *Smart Deviate*, o sistema irá selecionar a característica requerida *Rotate Ultrasonic Sensor*, ativando-a. Esta ativação irá disparar a regra **2**, que irá ativar o sensor ultrassônico. Para o robô, estas duas características envolvem um sensor, no caso o ultrassônico, e um motor acoplado ao sensor ultrassônico para girá-lo, o que será necessário para o algoritmo de desvio empregado pelo componente mapeado para a característica *SmartDeviate*.

```
1 rule "Verifica requer SmartDeviate e RotateUltrasonicSensor"
  when
      Caracteristica (nome == "smartDeviate", ativada == true)
      AND caracRotateUS : Caracteristica (nome == "rotateUS",
          ativada == false)
  then
      modify(caracRotateUS) { setAtivada(true) }
end

2 rule "Verifica compoe UltrasonicSensor e RotateUltrasonicSensor"
  when
      Caracteristica (nome == "rotateUS", ativada == true)
      AND caracUltrasonicSensor : Caracteristica (
          nome == "UltrasonicSensor", ativada == false)
  then
      modify(caracUltrasonicSensor) { setAtivada(true) }
end
```

Figura 4.3: Regras **1** e **2**.

Assim, as regras vão sendo disparadas uma a uma até que não haja mais regras para serem disparadas e este conjunto, composto do antecedente de cada

regra disparada, formará o plano de configuração do robô. Cabe aqui descrever como um sistema especialista dispara uma regra e quais as implicações disto: a verificação se a condição de uma regra é verdadeira é feita sobre os **fatos**, os quais estão armazenados na **memória de trabalho**. A ação da regra também é realizada sobre estes fatos, ou seja, o disparo de uma regra altera a memória de trabalho e esta alteração pode causar o disparo de outras regras. Isto pode levar a duas situações: a **parada**, onde não há mais regras a serem disparadas, ou um **laço infinito**, quando sempre há uma regra a ser disparada. Para evitar o segundo caso, foi empregado um recurso do sistema especialista utilizado, que deixa de disparar uma regra mais de um vez durante uma chamada ao sistema, evitando o ciclo infinito, porém este recurso pode não estar disponível no sistema especialista empregado, pois ele não é um recurso padrão para todos os sistemas deste tipo.

Depois de gerado o plano, a memória de trabalho deve ser atualizada, para refletir o estado atual da aplicação, pois ela foi utilizada como um modelo de predição sobre o que deve ser feito e, ao final, ela não representa a realidade e sim um estado futuro a ser alcançado após a reconfiguração.

Como dito anteriormente, há dois momentos em que uma configuração de software é gerada: na iniciação e em execução. Na iniciação, após o robô ser ligado, ele informa ao PC quais os sensores e motores disponíveis na sua configuração física, i.e., qual o hardware que está instalado nele. Com isto, o PC atualiza a memória de trabalho representando os itens de hardware disponíveis. Após o usuário efetuar a sua configuração, o PC irá determinar a configuração de software correspondente e informar ao robô as classes que devem ser instanciadas e respectivas referências. No robô há uma classe responsável por "montar" uma configuração de software e, como a API do robô não permite carregamento dinâmico de classes, todas as possíveis classes devem estar implantadas no robô, num único arquivo de extensão JAR, e o "montador" instancia os objetos destas classe sob demanda e realiza os enlaces entre as classes.

Já na execução, a reconfiguração acontece quando o robô não consegue cumprir uma determinada tarefa, como chegar num determinado ponto ou localizar algum objeto. Assim, o robô informa para o PC esta situação e um outro conjunto de regras é empregado, onde são descritas situações e as respectivas características que devem ser ativadas para se cumprir o objetivo. Ou seja, estas regras dizem quais as características que devem estar ativadas para um determinado contexto. Um exemplo é quando o robô deve ir a um determinado ponto e ler a cor que se encontra abaixo dele. Se ele encontra um obstáculo, e não consegue chegar até o ponto de destino, ele informa isto ao PC e a regra **10**, ilustrada na Figura 4.4, cuja condição é obstáculo encontrado e desvio inteligente não selecionado é disparada. Novamente, o disparo desta regra altera a memória de trabalho e faz disparar o conjunto de regras de dependências entre as características, gerando

uma nova configuração de software.

```
10 rule "Não consegue atingir destino e desvio smart não selecionado"
  when
      DestinationPoint (nome == "destinationPoint", arrived == false)
      AND Caracteristica (nome == "smartDeviate", ativada == false)
  then
      modify(smartDeviate) { setAtivada(true) }
end
```

Figura 4.4: Regra **10**, indicando a situação do contexto em que uma reconfiguração deve ser executada.

A geração da nova configuração é feita pelo componente de software **montador** executado no robô, por meio da diferença entre conjuntos, igual a abordagem proposta em (SYKES, 2010), onde:

componentes criados = configuração nova \ configuração atual
componentes removidos = configuração atual \ configuração nova

Os enlaces entre os componentes não são explicitados, pois o **montador** já possui um grafo de dependências entre os componentes e este grafo não é alterado durante a execução do software.

4.1.1 Considerações sobre o emprego da abordagem programática

Neste exemplo desenvolvido, algumas premissas foram feitas para simplificar o problema a ser resolvido, bem como outros problemas deixaram de ser tratados. Estes aspectos são descritos a seguir, a fim de identificar problemas a serem resolvidos pela abordagem proposta nesta pesquisa.

Configuração de software em alto nível

O uso do diagrama de características permite que o usuário escolha funcionalidades que irão compor o software num nível de abstração mais elevado, deixando de se preocupar com detalhes de mais baixo nível, que apesar de evitar erros de configuração, reduzem as possíveis combinações. Porém, ao trabalhar com conceitos de mais alto nível, algumas escolhas devem já estar resolvidas, como é o caso de escolhas **padrão** ou **default**. Este conceito é empregado em Linha de Produtos de Software quando há características que são obrigatórias em qualquer produto de uma família ou, num determinado ponto de variação, há variantes que são escolhidos por padrão, caso o arquiteto não faça a escolha por um outro variante diferente. Estas escolhas **padrão** também podem ser empregadas em componentes compostos que podem conter diferentes componentes na sua composição e, caso o arquiteto não detalhe o conjunto de componentes de uma

determinada composição, o mecanismo responsável por gerar uma configuração pode escolher os componentes **default** desta composição.

Algumas escolhas em alto nível podem resultar numa configuração inexistente. Por exemplo, as escolhas padrão em níveis mais baixos da hierarquia podem resultar em componentes que são mutuamente exclusivos, ou seja, não podem pertencer a uma mesma configuração de software. Assim, o mecanismo de geração da configuração deve ter a capacidade de mostrar ao arquiteto o motivo de uma determinada configuração não poder ser gerada.

O emprego do sistema especialista

O sistema especialista foi empregado em duas tarefas neste trabalho com o robô: a) na identificação do momento em que uma reconfiguração deve ser feita; e b) na determinação da reconfiguração em si. Na primeira tarefa, o arquiteto identifica possíveis *problemas*, e a respectiva *solução*, que podem surgir ao longo da execução do robô e as representa numa regra. O *problema* é o não cumprimento de algum requisito do sistema e a *solução* é uma *característica* que precisa ser ativada. A ativação desta característica leva a execução da segunda tarefa, que é resolver as dependências decorrentes da ativação da característica. Neste caso, ao invés de gerar uma regra para cada possível configuração de software, as relações de dependência funcional e de composição foram separadas e cada regra representam uma relação binária entre componentes. Para isto funcionar, as regras foram sendo disparadas uma a uma, por meio da alteração da memória de trabalho do sistema especialista. Deste modo, a semântica de uma regra poderia ser vista como a função de transição de estados γ do domínio de planejamento, mas com a ressalva de que os estados não podem ser descritos por meio de variáveis, pois o algoritmo empregado num sistema especialista não funciona com variáveis. Assim, se for desejado representar todas as transições, um número grande de regras deve ser descrito, o que torna a tarefa impraticável para um ser humano. A alteração da memória de trabalho ao longo da geração do plano de reconfiguração irá levar esta mesma memória a um estado diferente do atual. Assim, é preciso sincronizar a memória de trabalho com o estado atual da aplicação, após gerar o plano ou após executá-lo, refletindo a realidade da aplicação.

Dependências de outros recursos

Não foi considerado que os componentes de software, representados por classes Java na aplicação, dependem de outros recursos, além daqueles já disponíveis no pacote de implantação. Porém, poderia ser necessário que um determinado componente necessitasse utilizar uma biblioteca, por exemplo, comunicação USB, que não faz parte da aplicação manuseada pelo arquiteto, mas que deve estar disponível e ser fornecida pelo sistema operacional ou o *middleware* executado no robô. Quanto mais relacionamentos forem necessários representar

e verificar, maior o número de regras a serem inseridas no sistema especialista, tornando a tarefa do arquiteto de gerar as regras cada vez mais complexa.

Enlaces

Um enlace entre componentes pode ser mais complexo do que apenas uma referência entre os objetos que implementam um determinado componente. Em geral, estes mesmos enlaces representam os *conectores* existentes em algumas ADL, os quais podem definir diferentes protocolos de comunicação, requisitos de segurança ou mesmo se as chamadas serão síncronas ou assíncronas. Deste modo, considerar enlaces mais complexos também aumenta a complexidade das possíveis configurações de software que um conjunto de componentes pode gerar.

Conclusões sobre a abordagem programática

A abordagem programática baseada num sistema especialista facilita a representação do conhecimento do ponto de vista do desenvolvedor, pois se baseia na observação de que especialistas humanos parecem raciocinar a partir de regras de ouro ou *rules of thumb* para desempenhar as suas tarefas (BRACHMAN; LEVESQUE, 2004). Porém, quando a quantidade de informação necessária para este raciocínio cresce, o que acontece quando o modelo que representa o software possui mais detalhes, fica difícil manter a base de regras, bem como descrever tudo o que pode ser considerado. Além disso, não há como garantir que as regras sejam disparadas considerando-se uma relação de ordem entre elas, como uma regra que altera o estado de um componente antes de uma outra efetuar a troca de uma conexão deste componente.

Assim, ao invés de empregar uma abordagem programática na qual o desenvolvedor precise descrever o mecanismo que ajusta um software, esta pesquisa propõe empregar uma abordagem baseada em modelos. Nesta abordagem, o desenvolvedor não precisa dizer o que tem que ser feito para se atingir um objetivo, ele tem que dizer qual é o objetivo e o que tem que ser feito é a solução para o modelo.

4.2 A Solução Proposta

Inicialmente, a solução proposta nesta pesquisa possui a arquitetura descrita na Figura 4.5.

Nesta arquitetura, as entradas são a configuração objetivo, o estado inicial, as restrições e a descrição de Σ que, juntamente, formam o domínio de planejamento. O fluxo de execução ocorre da seguinte forma:

O **planejador** possui a descrição do domínio do software adaptável, que pode incluir o estilo arquitetural ou parâmetros de configuração. Estas restrições tam-

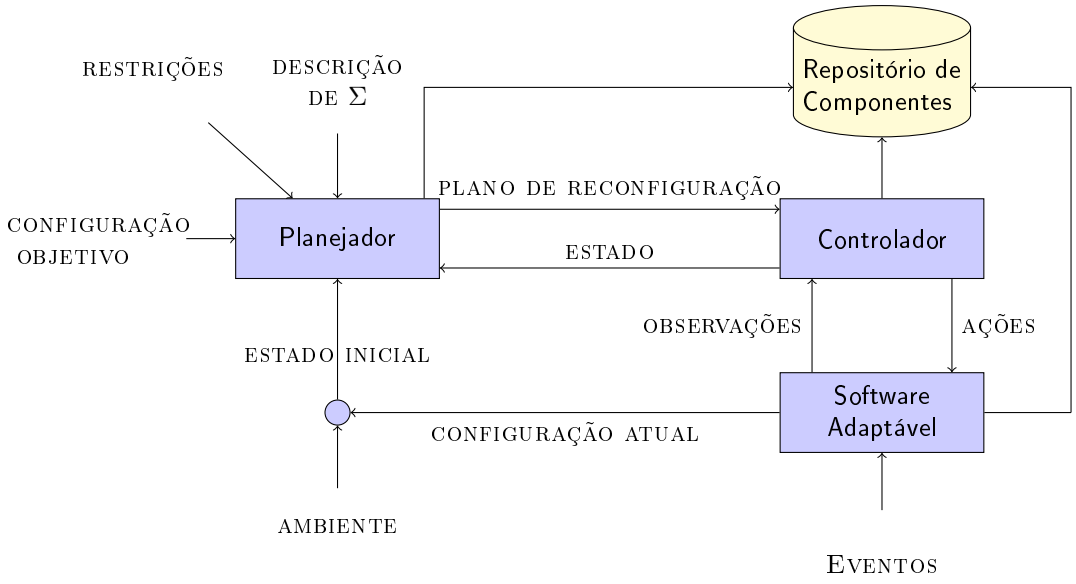


Figura 4.5: Arquitetura da solução

bém podem estar descritas no problema de planejamento que é composto do **estado inicial** e da **configuração objetivo**. Ao receber uma configuração objetivo, o **planejador** obtém o estado inicial, que é composto da configuração atual da aplicação e de dados do ambiente - como algum indicador de desempenho ou consumo de recursos - que possam interessar ao planejamento da reconfiguração. A seguir, ele gera o plano de reconfiguração, caso exista, e envia o mesmo para o **controlador**, que é o elemento responsável por atuar sobre o software, ajustando-o. Pode ser que haja mais de um plano e, neste caso, o **planejador** deve possuir algum critério para escolher qual deles será enviado ao **controlador**.

O **controlador** deve alterar o software a partir do plano recebido, executando as ações e coletando observações. Esta execução deve considerar algum protocolo de mudança, que pode já estar embutido no plano de reconfiguração, como o gerenciamento do ciclo de vida dos componentes, e as observações servem para indicar que a próxima ação do plano já pode ser executada, pois o estado em que as precondições para a ação a ser executada foi obtido. Além disso, o **controlador** deve considerar que as ações possuem uma duração, ou seja, não são instantâneas, e que há uma relação de ordem entre elas, onde uma ação só pode ser executada após uma outra ter sido executada previamente e com sucesso. Aqui, sucesso significa atingir o estado esperado no qual as precondições da próxima ação estão satisfeitas. O **repositório de componentes** contém os componentes de software e a sua descrição por meio de alguma meta-linguagem que deve informar: as interfaces requeridas e providas, atributos, se houver, se for um componente composto por quem ele é composto e que interfaces deverão ser expostas e dependências de outros recursos como bibliotecas e dispositivos físicos (hardware). Estas descrições podem ser recuperadas pelo **planejador**,

controlador e o **software**, ou seja, um novo componente de software no repositório pode ser considerado na hora de gerar o plano de reconfiguração e, neste último caso, esta informação deve fazer parte do estado inicial. Do ponto de vista do **software**, os componentes do repositório podem ser recuperados pelo modelo de componentes do software, o que permite implantá-los na configuração do software adaptável.

Durante este ciclo, alguns eventos podem acontecer e pode ser necessário efetuar fluxos alternativos. Após o **controlador** receber um plano, se o estado inicial para o qual ele foi gerado estiver diferente, o plano pode deixar de ser executável sobre aquele estado, necessitando ser refeito. Ao iniciar a execução da reconfiguração, o **controlador** pode se deparar com algum erro e ter de corrigí-lo. Aqui algumas possibilidades podem ser consideradas e nelas o estado inicial da aplicação é o estado atual, onde ele está inconsistente ou com um erro, por exemplo:

1. o planejador poderia ser executado novamente, mas agora a configuração objetivo seria a configuração inicial considerada na geração do plano, ou seja, o planejador deve reconfigurar o software do estado atual para o estado anterior a execução do plano;
2. o planejador poderia ser executado novamente, considerando a mesma configuração objetivo anterior, ou seja, uma nova tentativa de reconfiguração para o mesmo objetivo seria executada.

O mesmo **controlador** pode nunca conseguir executar o plano, ou seja, ele fica esperando a aplicação chegar num determinado estado para poder executar uma das ações do plano, mas este estado nunca é obtido. Por isso, deve haver algum meio de o **controlador** forçar que a aplicação vá para um determinado estado, ou que ele aborte a execução do plano.

Deste modo, esta solução pretende:

- Gerar planos de reconfiguração que considerem as restrições de consistência descritas na Seção 2.3.3, ou seja, estilos arquiteturais, restrições do modelo de componentes e da aplicação.
- Permitir que a adição de novos componentes possa ser considerada pelo planejador, aumentando as possibilidades de reconfigurações.
- Fornecer um meio de desfazer reconfigurações que não tenham terminado ou causado algum erro.

O componente **planejador** é o responsável por gerar o plano de reconfiguração da aplicação. O algoritmo de planejamento utilizado na arquitetura da

abordagem foi o JSHOP2 (ILGHAMI; NAU, 2003), uma implementação em Java do planejador SHOP2 (NAU *et al.*, 2001). Este planejador é independente de domínio, ou seja, o algoritmo empregado por ele não é vinculado a um domínio específico, e utiliza a semântica de rede de tarefas hierárquicas, ou *Hierarchical Task Network* (HTN), que emprega a decomposição para reduzir o tamanho do espaço de busca por um plano. Esta técnica foi utilizada neste trabalho devido: a característica hierárquica de uma aplicação em DBC, ao sucesso dos seus resultados em competições de planejamento, seu mecanismo de representação do domínio, o seu amplo uso em diferentes aplicações de planejamento e por ser um método correto para a geração de plano, como citado em (NAU; GHALLAB; TRAVERSO, 2004), ou seja, o plano é correto em relação ao domínio.

Para resolver problemas de planejamento num determinado domínio, o planejador JSHOP2 recebe um conjunto de métodos específicos do domínio, onde cada um fornece um meio de decompor uma determinada tarefa num conjunto parcialmente ordenado de subtarefas, dado que as precondições sejam satisfeitas. Para criar os planos, o JSHOP2 reduz o problema da seguinte forma: ele decompõe as tarefas recursivamente em subtarefas até obter uma tarefa primitiva que pode ser executada diretamente por meio de operadores de planejamento. Como mencionado na Seção 2.2.2, um operador de planejamento instanciado, i.e., com as variáveis substituídas, é uma ação de reconfiguração no software. Outro aspecto deste planejador é a premissa de mundo fechado ou *closed-world assumption*, que significa: um átomo que não esteja explicitamente especificado num determinado estado, não vale para aquele estado, ou seja, no caso da reconfiguração do software, se um predicado como `(started comanche)` - que significa que o componente *comanche* está iniciado - não existir num determinado estado significa que `(started comanche)` é falso, i.e., o componente *comanche* está parado.

4.2.1 O problema de planejamento

Nesta pesquisa, considera-se que o **software adaptável** foi desenvolvido sobre um modelo de componentes e este modelo deve ser representado tanto no problema de planejamento quanto no domínio. Inicialmente, nesta pesquisa o problema de planejamento é descrito por meio dos seguintes predicados, onde as palavras precedidas por **?** são variáveis a serem instanciadas por ocasião da busca por um plano:

- `(component ?comp single)` - um componente primitivo, ou seja, que não contém outros componentes. O componente representado pela variável `?comp` é um componente primitivo.
- `(component ?comp composite)` - um componente composto. O componente

representado pela variável `?comp` é um componente composto.

- `(can-contain ?compP ?comp)` - a relação de composição entre dois componentes, o componente representado pela variável `?compP` pode conter o componente `?comp`.
- `(reference ?comp ?iface single)` - uma interface requerida por um componente. O componente representado pela variável `?comp`, requer uma interface de nome `?iface` e esta interface é singela, ou seja, `?comp` pode se conectar a um e somente um outro componente por meio desta interface.
- `(reference ?comp ?iface multiple)` - uma interface requerida por um componente cuja cardinalidade é maior ou igual a um, ou seja, um ou mais componentes podem ser conectados a `?comp` por meio desta interface.
- `(service ?comp ?iface)` - uma interface provida por um componente. O componente representado pela variável `?comp`, provê uma interface de nome `?iface`.
- `(started ?comp)` - o estado do ciclo de vida de um componente, representado pela variável `?comp`, e significa que este componente está iniciado. O predicado `(not (started ?comp))` significa que o componente `?comp` está parado.
- `(active ?comp)` - significa que o componente, representado pela variável `?comp`, está disponível para a aplicação.
- `(bind ?compP ?comp ?iface)` - um enlace ou conexão entre dois componentes. O componente representado pela variável `?compP` está conectado ao componente `?comp`, por meio da interface `?iface`. Um enlace é direcionado, ou seja, a conexão é de `?compP` para `?comp`, pois a pré-condição para que este enlace exista é `?compP` requerer uma interface provida por `?comp`.
- `(promote ?subComp ?ifaceSub ?compComposite ?ifaceComp)` - um enlace entre um componente composto e um subcomponente seu. A interface `?ifaceSub` do componente representado pela variável `?subComp` está conectada a interface `?ifaceComp` do componente `?compComposite`, que é composto. Desta forma, a interface provida ou requerida pelo componente composto está vinculada a uma outra interface provida ou requerida por um subcomponente seu.
- `(contain ?compP ?comp)` - a relação de composição entre dois componentes, o componente representado pela variável `?compP` contém o componente `?comp`.

Um aspecto que foi observado, durante as execuções dos diferentes problemas de reconfiguração, foi a necessidade de efetuar uma verificação sobre a descrição do problema. Esta verificação serve para garantir que os predicados não

descrevem uma configuração objetivo inconsistente com o modelo de componentes, por exemplo, as seguintes inconsistências não podem ocorrer no arquivo do problema:

- interfaces requeridas que não estejam satisfeitas;
- uma interface requerida simples ligada a duas outras interfaces providas;
- um componente composto sem subcomponentes;
- componentes fora da hierarquia de componentes;
- componente na posição errada dentro da hierarquia;
- componente composto com uma interface e sem o respectivo enlace com a interface de um subcomponente;

Neste primeiro experimento, estas verificações foram feitas manualmente, mas faz-se necessário que elas sejam verificadas automaticamente, devido ao elevado número de predicados empregados na especificação. Também não foram representadas as dependências de recursos do sistema, como bibliotecas ou conexões de rede.

4.2.2 Modelagem do domínio

Num planejador baseado em HTN, o objetivo não é obter um conjunto de metas mas realizar um conjunto de tarefas que, se executadas, levarão o sistema ao estado objetivo. O domínio do planejador inclui um conjunto de operadores e também um conjunto de métodos. Um método é uma prescrição de como decompor alguma tarefa em algum conjunto de subtarefas. O planejamento prossegue decompondo as tarefas não-primitivas, i.e., aquelas que possuem uma decomposição, recursivamente, em subtarefas cada vez menores, até atingir as tarefas primitivas que são formadas por um conjunto de operadores de planejamento. Aqui cabe lembrar que uma ação, que efetivamente altera o estado, é um operador de planejamento instanciado.

Esta semântica de rede de tarefas pode ser vista como uma maneira conveniente de se descrever pequenas “receitas” de como se resolver um problema. Para o domínio do planejador empregado, um método é declarado na seguinte sintaxe `(:method h [nome1 ] L1 T1 [nome2 ] L2 T2 . . . [nomen ] Ln Tn )` e tem como propósito especificar:

- Se o estado atual do mundo satisfaz um conjunto de precondições **L1**, então este método pode ser realizado por meio da execução das tarefas em **T1** na ordem dada;

- Caso contrário, se o estado atual do mundo satisfaz um conjunto de pre-condições **L2**, então este método pode ser realizado por meio da execução das tarefas em **T2** na ordem dada;
- Caso contrário, se o estado atual do mundo satisfaz um conjunto de pre-condições **Ln**, então este método pode ser realizado por meio da execução das tarefas em **Tn** na ordem dada;

O domínio de planejamento desenvolvido nesta pesquisa, reflete o modelo de componentes Fractal, no qual há componentes compostos e um componente composto pode prover e requerer as interfaces dos componentes que o compõe. Na Figura 4.6, é apresentado o método **configure** e a decomposição deste método, a qual é feita da seguinte forma: execução do método **destroyComponent** e depois execução do método **reconfigure**.

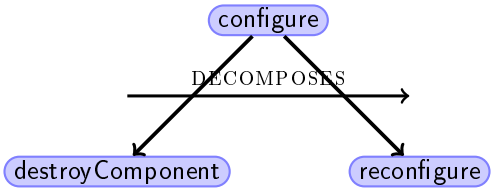


Figura 4.6: Decomposição do método `configure`.

O método **destroyComponent**, cuida de remover os componentes da configuração atual e que não pertencem a configuração objetivo. Neste caso, o componente deverá ser removido e, recursivamente, todos os componentes nesta situação também devem ser removidos. Quando não houver mais componentes a serem removidos, o método **reconfigure** é chamado, e os componentes da configuração objetivo são configurados.

Na Figura 4.7, é apresentado o método **reconfigure**. A decomposição do método é feita nos métodos folha na sequência em que eles estão descritos, ou seja, da esquerda para a direita.

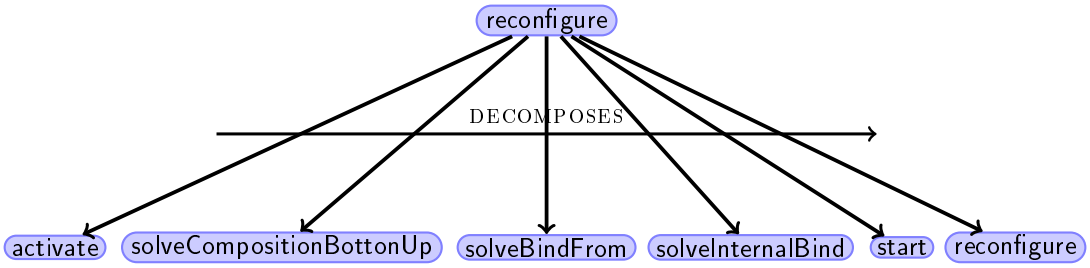


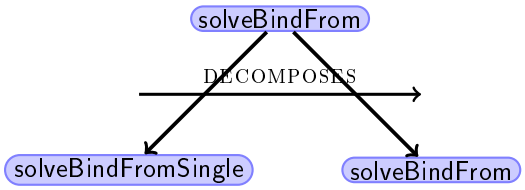
Figura 4.7: O grafo de decomposição do método **reconfigure**.

Assim, seguindo a decomposição ilustrada na Figura 4.7, para se reconfigurar um componente da configuração objetivo, precisa-se:

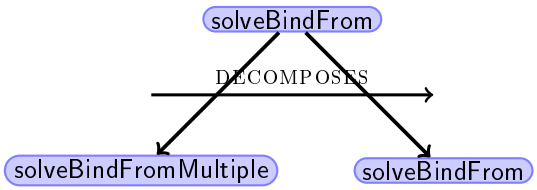
1. ativar o componente;

2. resolver a relação de composição dele com o componente que o contém, isto é, o seu pai;
3. resolver as dependências funcionais do componente, ou seja, as suas interfaces requeridas;
4. resolver os enlaces do componente com as interfaces do componente no qual ele está contido;
5. iniciar o componente; e
6. recursivamente reconfigurar os demais componentes.

Em seguida, na Figura 4.8, é apresentado o método **solveBindFrom**. Dependendo do estado em que o planejador esteja, este método pode ser decomposto de duas diferentes maneiras. A primeira, ilustrada na Figura 4.8a, se refere aos componentes que possuem uma interface requerida simples, ou seja, o componente requer uma e apenas uma outra interface. Já na Figura 4.8b, se refere aos componentes que possuem uma interface requerida múltipla, que requerem uma ou mais interfaces.



(a) Decomposição para enlaces simples.



(b) Decomposição para enlaces múltiplos.

Figura 4.8: O grafo de decomposições do método **solveBindFrom**.

Assim, as tarefas dos métodos vão sendo encadeadas e formam uma hierarquia até que o planejador ache um plano. Um exemplo desta hierarquia pode ser visto na Figura 4.9, contendo apenas o método **reconfigure** acrescido do método **solveBindFrom**.

Como descrito anteriormente, um método pode ter diferentes formas de ser decomposto, dependendo do estado em que ele se encontre e este estado é estabelecido por um conjunto de precondições. Na Figura 4.10, foi reproduzida a precondição para decompor o método **solveBindFrom**. As precondições são predicados descritos no problema de planejamento (ver Seção 4.2.1) e as variáveis são nomes precedidos do símbolo **?**. Então, para este método, a precondição para se decompô-lo como descrito na Figura 4.8a é:

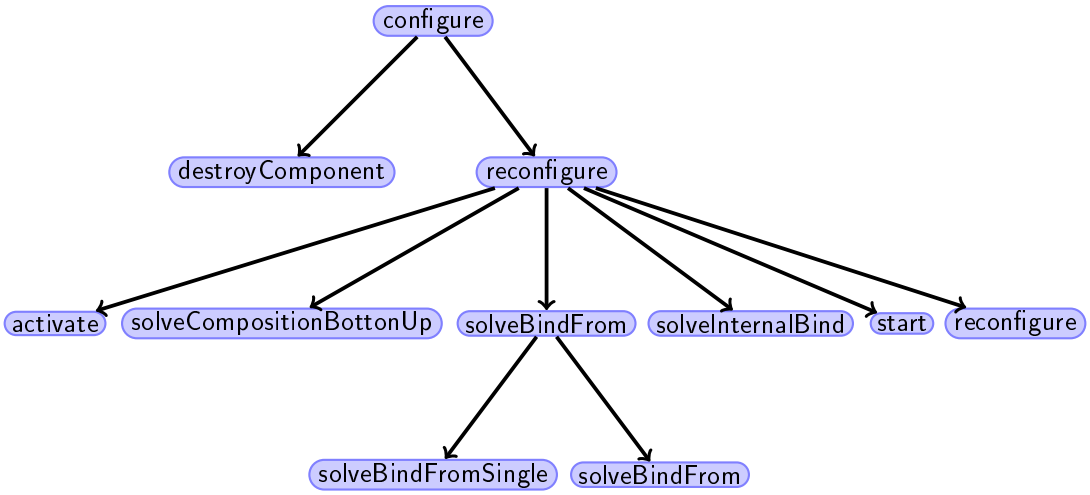


Figura 4.9: O grafo da hierarquia de tarefas.

1. verificar se a variável **?comp1** é um componente;
2. se **?comp1** precisa ser conectado a um outro componente **?compServ**, por meio da interface simples requerida **?iface**; e
3. se o componente **?compServ** já está ativado e se ele oferece a interface **?iface**.

```
(:method (solveBindFrom ?comp1)

 _1solveBindFromSingleReference
  ; PRE Component single reference
  (
    (component ?comp1 ?var1)
    (reference ?comp1 ?iface single)
    (to-bind ?comp1 ?compServ ?iface)
    (component ?compServ ?var2)
    (active ?compServ)
    (service ?compServ ?iface)
  )
  ; TASK
  (
    (solveBindFromSingle ?comp1 ?compServ ?iface)
    (solveBindFrom ?comp1)
  )
)
```

Figura 4.10: As precondições para aplicar uma decomposição do método **solveBindFrom**.

4.2.3 A aplicação Comanche

O software adaptável escolhido para ser utilizado neste exemplo foi a aplicação Comanche[4], que é um servidor web simplificado. Este servidor web foi desenvolvido empregando-se a OW2 FraSCAti[5], que é uma implementação de código aberto da especificação do padrão *Service Component Architecture*[6] (SCA). Nesta especificação, há um arquivo em XML que descreve a arquitetura da aplicação como os componentes, as relações de composição, as interfaces e respectivos enlaces, as classes que implementam um determinado componente e os enlaces entre as interfaces de componentes compostos e seus subcomponentes. O FraSCAti foi desenvolvido sobre o modelo de componentes Fractal[7], que é um modelo independente de linguagem de programação e possui como uma das principais características a capacidade reflexiva, provendo apoio a introspecção e reconfiguração em tempo de execução. Apesar da especificação SCA não prever a capacidade reflexiva, esta é provida por meio do Fractal.

O grafo de dependências funcionais, das relações de composição e de interfaces está ilustrado na Figura 4.11.

Os vértices na forma de elipse representam os componentes de software compostos, que contém outros componentes, e os retângulos com bordas arredondadas são os componentes primitivos, i.e., que não contém nenhum outro componente. Os vértices circulares, representam as interfaces providas e os quadrados as interfaces requeridas e as arestas com rótulo *bind* representam as dependências funcionais dos componentes, ou seja, os enlaces entre componentes por meio das interfaces. As arestas rotuladas com a palavra *child* representam as relações de composição entre os componentes, onde no modelo de componentes utilizado esta relação é bidirecional, ou seja, se um componente A é filho de um componente B, então também há a relação componente B é pai de componente A. As interfaces providas e requeridas por um componente composto são na verdade, interfaces providas e requeridas por componentes que estão contidos neste componente composto. Assim, as arestas tracejadas e rotuladas com a palavra *promote* representam a relação entre uma interface de um componente composto e a respectiva interface do componente que ele contém, e.g., a interface *r* do componente **comanche**, é a interface *r* do componente **frontend** que, por sua vez, é a interface *runnable* do componente primitivo **request receiver**.

O fluxo de funcionamento deste servidor é o seguinte: ao iniciar, o componente **request receiver** abre uma conexão soquete numa determinada porta da máquina onde ele está sendo executado e fica aguardando receber uma requisição. Ao receber uma requisição, ele chama o componente **backend** por meio da

[4]http://fractal.ow2.org/tutorials/comanche.html
[5]http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti
[6]http://www.oasis-opencsa.org/sca
[7]http://fractal.ow2.org/

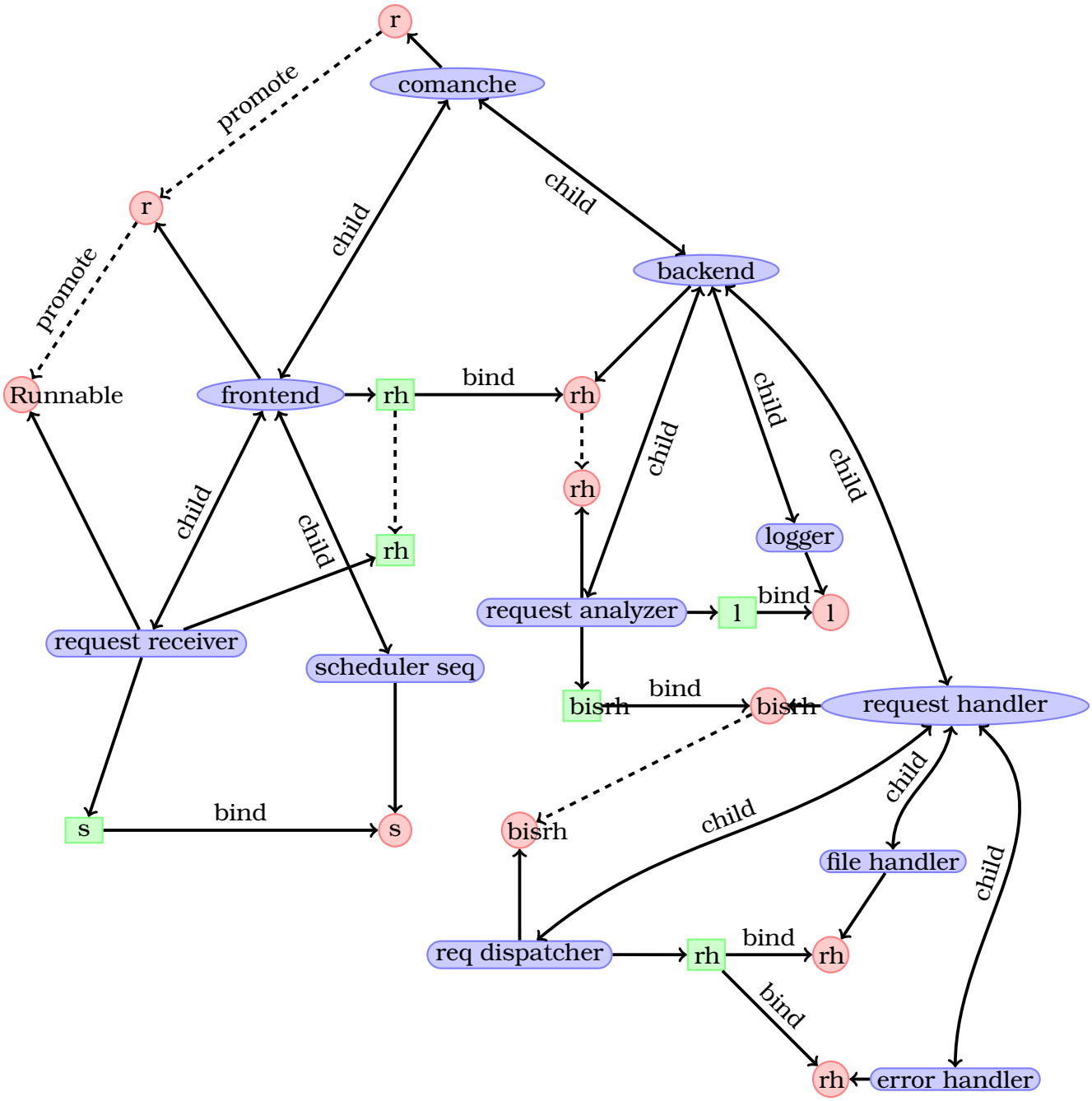


Figura 4.11: O grafo que representa a arquitetura da aplicação Comanche no nível dos componentes de software.

interface *rh*, e deixa que esta chamada seja escalonada pelo **scheduler**. Na configuração padrão, o **scheduler** emprega um escalonador sequencial, onde cada requisição é tratada e, somente após esta terminar, ele irá tratar a próxima requisição. No componente **backend**, o **request analyzer** recebe a requisição, chama o **logger** e o **request handler** pela interface *bisrh*. No **request handler**, o componente **request dispatcher** trata a requisição, buscando um componente que possa tratar aquele tipo de requisição. Caso não haja este componente, a requisição é tratada pelo **error handler**.

4.2.4 Geração da reconfiguração

Depois da modelagem do domínio de planejamento, baseada no modelo de componentes Fractal, e do problema de planejamento da aplicação Comanche, serão consideradas algumas reconfigurações a serem obtidas por meio do planejador, ou seja, uma configuração objetivo será entregue ao planejador e ele deverá fornecer um plano de reconfiguração.

Primeira reconfiguração

Na primeira reconfiguração, o objetivo representa uma troca entre componentes. Esta troca será feita no componente responsável por atender as requisições e que será substituído por um outro. Na configuração inicial, este componente é o **scheduler Seq**, que atende às requisições sequencialmente e, na configuração objetivo, este componente será substituído pelo componente **schedulerMultiThread**, cujo escalonador emprega *multithread* para atender as requisições. A descrição da configuração de software objetivo deve conter todos os componentes e respectivas relações entre eles, os quais são representados por meio de predicados, conforme apresentado na Figura 4.12.

Os predicados precedidos por um asterisco destacam o novo componente, ou seja, os predicados indicam:

- o componente que deve ser ativado, que no caso é o **schedulerMultiThread**;
- a localização na hierarquia dos componentes, pois ele deve ser inserido no componente composto **frontend**; e
- a dependência funcional, pois o serviço **s**, provido por este componente, irá atender ao serviço requerido pelo componente **request receiver**.

Para efetuar a transição entre as duas configurações, o planejador deve resolver os seguintes aspectos:

- Remover as dependências - isto depende do modelo de componentes subjacente. No modelo de componentes utilizado, para se remover arestas *bind* e *promote*, o componente que é a origem da aresta, no caso **request receiver**, deve estar no estado *parado*. Assim, para desfazer o enlace entre as interfaces **s** dos componentes **request receiver** e **scheduler seq**, é necessário *parar* o componente **request receiver** antes de ter o enlace da interface *s* efetivamente removido.
- Remover as relações de composição - ao se remover um componente, este deve ser retirado da hierarquia correspondente. Assim, o **scheduler seq** deve ser removido de dentro do **frontend**.

```
(to-active comanche)
(to-contain comanche frontend )
(to-contain comanche backend )
(to-promote frontend r comanche r)
(to-promote requestAnalyzer rh backend rh)

(to-active frontend)
(to-active backend)
(to-active requestReceiver)
(to-active logger)
(to-active requestAnalyzer)
(to-active requestHandler)
(to-active requestDispatcher)
(to-active errorHandler)
(to-active fileHandler)
*(to-active schedulerMultiThread)

*(to-contain frontend schedulerMultiThread)
(to-contain frontend requestReceiver)
(to-contain backend logger)
(to-contain backend requestAnalyzer)
(to-contain backend requestHandler)
(to-contain requestHandler requestDispatcher)
(to-contain requestHandler errorHandler)
(to-contain requestHandler fileHandler)

(to-promote requestReceiver runnable frontend r)
(to-promote requestReceiver rh frontend rh)
(to-promote requestAnalyzer rh backend rh)
(to-promote requestDispatcher bisrh requestHandler bisrh)

*(to-bind requestReceiver schedulerMultiThread s)
(to-bind frontend backend rh)
(to-bind requestAnalyzer logger l)
(to-bind requestAnalyzer requestHandler bisrh)
(to-bind requestDispatcher errorHandler rh)
(to-bind requestDispatcher fileHandler rh)
```

Figura 4.12: Predicados que descrevem a configuração de software objetivo.

- Resolver as relações de composição - para a inserção de um novo componente, este deve ser colocado na respectiva hierarquia, Assim, o **schedulerMultiThread** deve ser inserido dentro do **frontend**.
- Resolver as dependências funcionais - no caso em questão, o componente **schedulerMultiThread** não requer uma interface, mas o componente **request receiver** requer o serviço **s** provido por **schedulerMultiThread**.
- Reiniciar a aplicação - aqui, no modelo de componentes utilizado, significa

alterar o estado dos componentes afetados, ou seja, alterar do estado *parado* para o estado *iniciado*.

O saída do planejador foi reproduzida na Tabela 4.1, e o plano contém as ações, ou seja, a instanciação dos operadores de planejamento e a sequência com que elas devem ser executadas.

Nr	**Ação**
1	(!stop schedulerseq)
2	(!stop requestreceiver)
3	(!unbind requestreceiver schedulerseq s)
4	(!removechild frontend schedulerseq)
5	(!remove schedulerseq)
6	(!addnew schedulermt)
7	(!start schedulermt)
8	(!addchildnew frontend schedulermt)
9	(!bindnew requestreceiver schedulermt s)
10	(!start requestreceiver)

Tabela 4.1: Plano de reconfiguração de software contendo as ações.

Após gerar o plano, este foi traduzido para as ações de introspecção e reconfiguração fornecidas pelo FraSCAti. Esta tradução é uma relação de um para um das ações de reconfiguração para a linguagem FScript e FPath. Esta tradução está descrita na Tabela 4.2. No primeiro passo, o componente a ser removido **schedulerSeq** é colocado no estado *parado*. Em seguida, o componente **requestReceiver** cuja interface requerida **s** é provida pelo **schedulerSeq** também é colocado no estado *parado*. Depois o enlace entre estes dois componentes é removido. O novo componente **schedulerMT** é criado a partir do arquivo contendo a descrição do componente e a variável scHMT aponta para este componente. Este componente é colocado no estado *iniciado* e em seguida ele é inserido na hierarquia de componentes, compondo o componente **frontend**. O enlace entre o **requestReceiver** e o **schedulerMT** é realizado e em seguida o **requestReceiver** é colocado no estado *iniciado*.

Nr	**Ação Plano**	**FScript**
1	(!stop schedulerseq)	stop(frontend/scachild::schedulerSeq)
2	(!stop requestreceiver)	stop(frontend/scachild::requestReceiver)
3	(!unbind requestreceiver schedulerseq s)	remove-scawire(frontend/scachild::requestReceiver/scareference::s, frontend/scachild::schedulerSeq/scaservice::Scheduler)
4	(!addnew schedulermt)	schMT = sca-new('file:///src/resources/schedulerMT')
5	(!start schedulermt)	start(schMT/scachild::schedulerMT)
6	(!addchildnew frontend schedulermt)	add-scachild(frontend,schMT/scachild::schedulerMT)
7	(!bindnew requestreceiver schedulermt s)	add-scawire(frontend/scachild::requestReceiver/scareference::s, frontend/scachild::schedulerMT/scaservice::Scheduler)
8	(!start requestreceiver)	start(frontend/scadescendant::requestReceiver)

Tabela 4.2: Ações de reconfiguração do Comanche em FScript e FPath.

Segunda reconfiguração

Na segunda configuração objetivo, um número maior de componentes será

envolvido na reconfiguração do software. Haverá a troca de três componentes além da inserção de mais um novo componente. Além disso, também serão feitas reconfigurações que envolvem interfaces requeridas e providas nos componentes compostos, ou seja, as ações do plano também devem resolver as relações *promote*. Esta segunda reconfiguração servirá para mostrar que o número de ações que uma reconfiguração pode envolver torna o problema complexo de ser resolvido pelo desenvolvedor, fazendo-se necessário um apoio computacional. Os componentes envolvidos nesta reconfiguração são: a troca do **request receiver** pelo **reqRx**, a troca do **scheduler Seq** pelo **schedulerMT**, a troca do **logger** pelo **logQueue** e a adição de um componente responsável por responder a requisições de imagens, denominado de **image handler**.

A configuração inicial e a final estão ilustradas na Figura 4.13 e a saída do planejador foi reproduzida na Tabela 4.3.

Nr	**Ação**
1	(!stop requestreceiver)
2	(!unbind requestreceiver schedulerseq s)
3	(!stop frontend)
4	(!unbindinternal requestreceiver runnable frontend r)
5	(!unbindinternal requestreceiver rh frontend rh)
6	(!removechild frontend requestreceiver)
7	(!remove requestreceiver)
8	(!stop schedulerseq)
9	(!removechild frontend schedulerseq)
10	(!remove schedulerseq)
11	(!stop logger)
12	(!stop requestanalyzer)
13	(!unbind requestanalyzer logger l)
14	(!removechild backend logger)
15	(!remove logger)
16	(!addnew schedulermt)
17	(!addchildnew frontend schedulermt)
18	(!start schedulermt)
19	(!addnew reqrx)
20	(!addchildnew frontend reqrx)
21	(!bindnew reqrx schedulermt s)
22	(!start reqrx)
23	(!addnew logqueue)
24	(!addchildnew backend logqueue)
25	(!start logqueue)
26	(!bindnew requestanalyzer logqueue l)
27	(!start requestanalyzer)
28	(!addnew imagehandler)
29	(!addchildnew requesthandler imagehandler)
30	(!start imagehandler)
31	(!stop requestdispatcher)
32	(!bindnew requestdispatcher imagehandler rh)
33	(!start requestdispatcher)
34	(!bindinternalnew reqrx runnable frontend r)
35	(!bindinternalnew reqrx rh frontend rh)
36	(!start frontend)

Tabela 4.3: Ações de reconfiguração correspondentes a aplicação **Comanche**.

4.3 Considerações

Como já havia sido mencionado na Seção 2.1.2, a abordagem programática representada pelo emprego do sistema especialista restringe a tarefa de descrever o diagrama de estados, principalmente quando o número de possíveis estados é grande. A Figura 4.11 já fornece uma ideia das possíveis alterações de configuração que uma aplicação pode sofrer e fica claro que empregar a abordagem programática só é viável se as alterações consideradas forem pequenas.

A quantidade de relações de dependência entre os componentes que compõe uma configuração arquitetural de um software encontrada em aplicações do mundo real tende a crescer. Por exemplo, se o servidor Comanche tivesse que fazer uma requisição para um outro serviço externo exposto como um serviço web, e que para efetuar esta requisição ele faça uso de bibliotecas de terceiros. Então haveria uma dependência deste componente para outros recursos que não apenas os apresentados no grafo de composição ilustrado na Figura 4.11. Estas e outras dependências irão aumentar o modelo a ser considerado durante uma tarefa de reconfiguração, pois agora estas bibliotecas deverão estar disponíveis para serem utilizadas pelos componentes da aplicação.

Um aspecto que não foi considerado nas reconfigurações foi a transferência do estado da aplicação. Quando há uma troca de dois componentes, por exemplo, quando o componente A será trocado pelo componente B, pode-se pensar que o componente B irá desempenhar o mesmo papel que o componente A e, para isso, ele deve receber o estado de A. Entretanto, a não ser que seja explicitamente dito que haverá uma troca entre dois componentes, que é o caso da abordagem programática, seria necessário identificar o que significa uma troca entre as duas configurações de software. Um maneira seria identificar se uma interface requerida que está sendo retirada da configuração inicial será provida na configuração objetivo. Caso afirmativo, este novo componente com a interface provida seria o destinatário do estado. Porém, quando um número maior de componentes está envolvido numa reconfiguração, ou quando não há um mapeamento de um para um entre os componentes removidos e os adicionados, a identificação dos destinatários do estado da aplicação fica prejudicada. Talvez fosse necessário discriminar na nova configuração explicitamente a origem e o destino do estado dos componentes e assim, o planejador incluiria as ações que efetuam esta troca de estado, desde que elas estivessem no domínio.

Com o intuito de levantar o crescimento da complexidade envolvida na geração do plano de reconfiguração, foram feitas três diferentes execuções. Na primeira execução, que corresponde a primeira reconfiguração descrita anteriormente, foi considerada uma aplicação com poucos componentes e apenas uma troca de componentes. Na segunda, foi considerada a aplicação **Comanche** completa mas a mesma troca de componentes realizada na primeira execução. Na

terceira e última, a aplicação **Comanche** completa foi usada e uma reconfiguração envolvendo quatro componentes foi realizada. O domínio permaneceu o mesmo e apenas o problema foi alterado e os resultados obtidos foram descritos na Tabela 4.4.

Execução	**Tamanho da reconfiguração e da aplicação**	**Nr de passos**
1	troca de um componente - **Comanche** com quatro componentes	153
2	troca de um componente - **Comanche** completo	329
3	troca de três componentes e acréscimo de um - **Comanche** completo	620

Tabela 4.4: As diferentes execuções das reconfigurações.

Nela, pode-se observar o crescimento no número de passos efetuados pelo planejador JSHOP2 a cada execução. Entre as execuções **1** e **2**, o número de passos aumenta devido ao maior número de possibilidades a serem consideradas pelo algoritmo na busca por um plano pois, na execução **2**, a aplicação possui um número maior de componentes. Entre as execuções **2** e **3**, o número de passos aumenta devido ao plano ser maior, pois apesar da aplicação ter o mesmo tamanho, o número de componentes envolvidos na reconfiguração desejada é maior.

É claro que o tamanho da reconfiguração também depende das relações que um componente possui, isto é, um componente com apenas uma interface provida e nenhuma requerida ou exposta pelo seu supercomponente, é menos complexo que um componente com interfaces requeridas, providas e expostas pelo seu supercomponente. A intenção não é levantar uma curva de crescimento da complexidade da reconfiguração dado uma variável de entrada, mas sim mostrar que aplicações mais complexas ou com um número maior de elementos considerados numa reconfiguração levam mais tempo para serem solucionadas.

Outro ponto que foi observado, é que a maneira como se descreve o domínio pode trazer grandes consequências no tempo consumido para se descobrir um plano. Numa das versões do domínio, a reconfiguração era feita para qualquer tipo de componente, ou seja, não havia uma separação entre os componentes simples e os compostos e, neste caso, uma execução igual a **3**, levava 719 passos para ser executada. Isto se deve ao domínio considerar a adição de um componente ao seu pai antes de efetuar os demais relacionamentos, ou seja, quando este componente era composto e os filhos ainda não tinham sido incluídos, ele era descartado e apenas quando os filhos já tinham sido incluídos era que o pai podia ser finalizado. Deste modo, executando-se a reconfiguração primeiro nos componentes simples, já sanava as precondições exigidas pelos componentes compostos, reduzindo as possibilidades consideradas pelo planejador e levando 620 passos para ser finalizada.

Também foi notado que domínios de planejamento com métodos que possuem

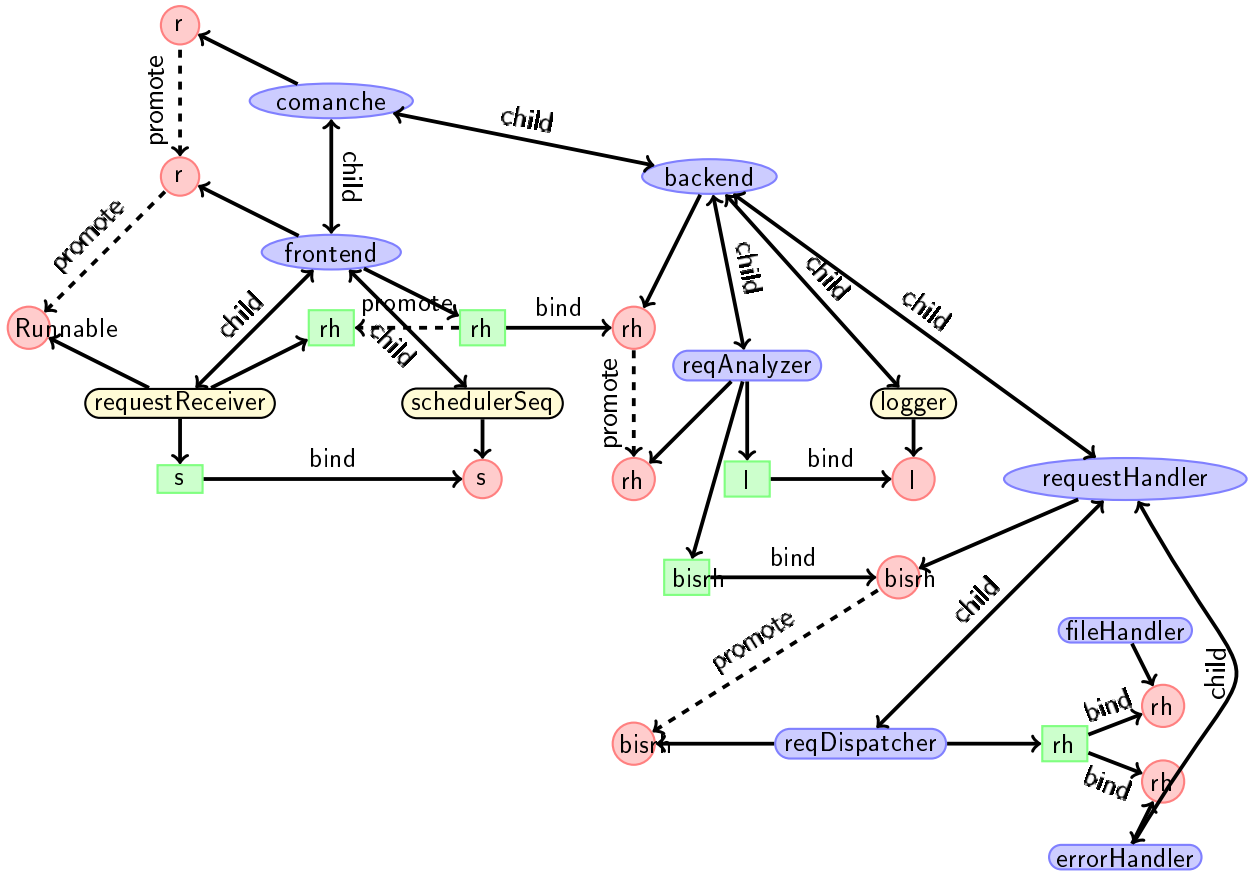
várias formas de serem decompostos, dependendo do estado em que se encontram, são mais complexos de serem resolvidos do que poucas formas de decomposição e hierarquias mais profundas. Esta observação foi constatada quando da realização da reescrita da decomposição dos métodos, reduzindo as possíveis decomposições e aprofundando a hierarquia de tarefas. Antes desta reescrita, o plano correspondente a execução **3** não conseguia ser determinado pelo planejador, pois ele acabava com toda a memória alocada para a máquina virtual Java (JVM), que no caso era de 4 gigabytes. Após a rescrita, o plano foi determinado em aproximadamente 0.06 segundos e consumiu 13 megabytes de memória.

Apesar de não ter sido totalmente implementado até o momento, o **controlador** pode-se deparar com uma série de questões. Por exemplo, se ao executar uma ação o software adaptável devolver um erro, o controlador deve ser capaz de gerar um *dump* da configuração arquitetural vigente. Como foi dito antes, o algoritmo de planejamento é correto perante o modelo, mas nada garante a corretude do modelo e este *dump* pode ajudar na depuração do modelo, identificando erros no modelo, ou mesmo conceitos que devem ser representados no modelo. No modelo de componentes Fractal, o erro gerado pela execução de uma ação em FScript é descrito no nível da linguagem de programação, que no caso foi a linguagem Java e, neste nível, fica difícil descobrir o que pode ter causado a exceção. Assim, do mesmo modo que o plano de reconfiguração, descrito numa linguagem de alto nível, reconfigura um software, faze-se necessário que haja também uma outra linguagem para fornecer o retorno quando uma reconfiguração não ocorre como o esperado. Uma outra situação é a impossibilidade de executar um determinado plano, seja por conta de algum evento exógeno ou por algum estado que não pode ser obtido. Cabe lembrar que o plano foi feito baseando-se num modelo da realidade e, ao ser executado, a realidade pode ser diferente pois o modelo pode não estar representando tudo o que é necessário.

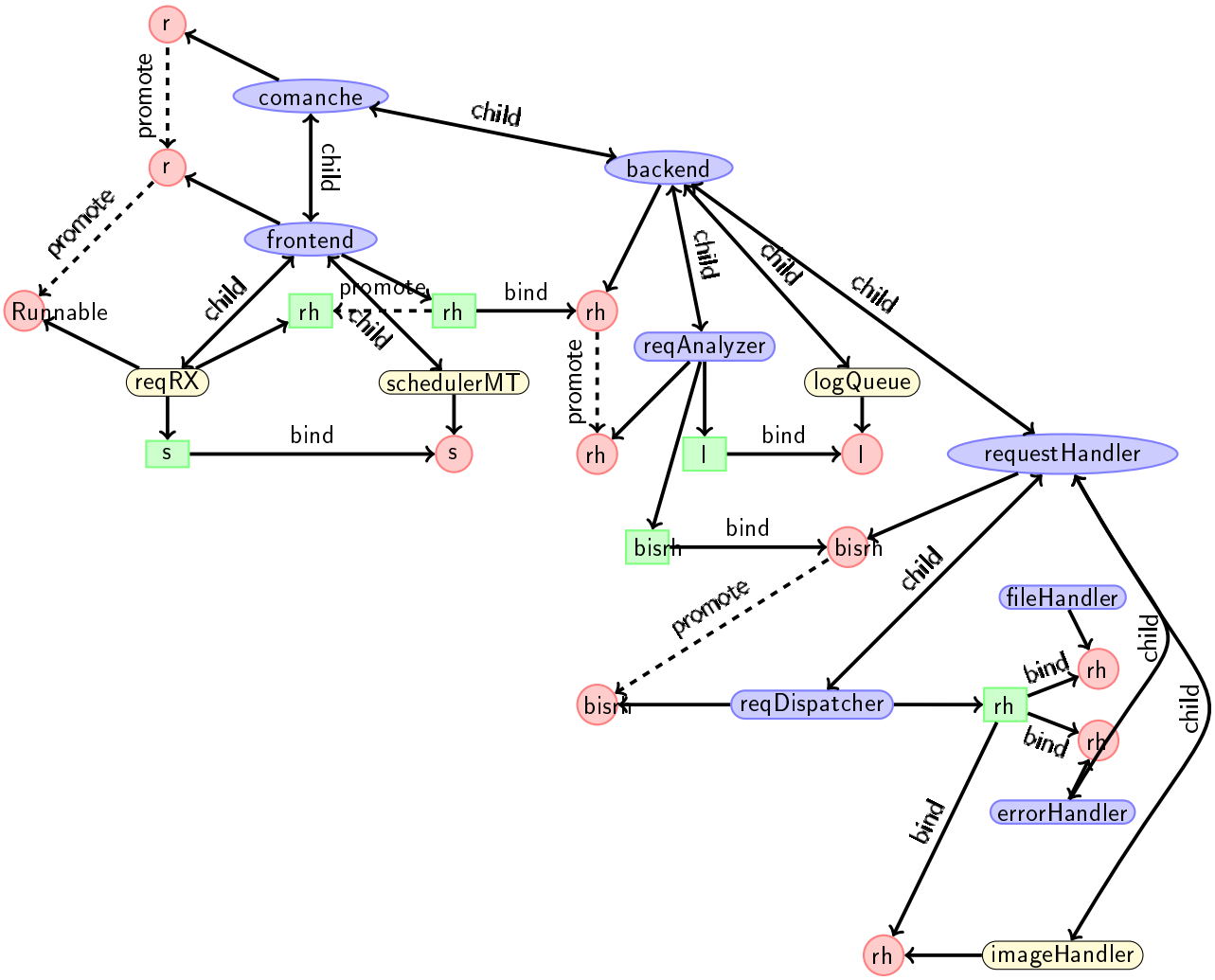
Nos trabalhos apresentados no Capítulo 3, apesar de alguns empregarem a abordagem baseada em modelos, eles não trataram de uma série de questões relacionadas com o modelo de componentes subjacente. Com isso, o domínio de planejamento é mais simples e não é a toa que, em uma delas, um algoritmo de busca em profundidade, sem empregar qualquer heurística na geração dos estados no espaço de busca, foi o suficiente para determinar uma reconfiguração. Eles também não consideram os aspectos relacionados com o componente que executa as ações de reconfiguração, chamado de **controlador** neste proposta de pesquisa, que é importante para manter um software consistente do ponto de vista funcional.

Assim, esta proposta de pesquisa pretende automatizar a reconfiguração de um software, cuja implementação se baseia num modelo de componentes reflexivo e de uso comum. Esta automatização deve se preocupar em: a) gerar um plano de reconfiguração consistente, isto é, que esteja de acordo com o meca-

nismo reflexivo do modelo de componentes; e b) manter a aplicação num estado consistente, do ponto de vista arquitetural e do modelo de componentes subjacente. Esta automatização também deve ser capaz de fornecer ao desenvolvedor explicações no nível do modelo que o permitam entender as decisões e levantar possíveis causas para os erros porventura encontrados, pois estes erros podem ser provenientes de uma modelagem errada ou incompleta da realidade que se deseja representar.



(a) Configuração Inicial



(b) Configuração Objetivo

Figura 4.13: A reconfiguração correspondente a execução 3.

Capítulo 5

Considerações Finais

Um software adaptável pode ser visto como uma solução para o problema de sistemas que precisam ser ajustados em relação ao contexto. Para que este sistema faça estes ajustes com autonomia, ele precisa ter capacidade de tomar decisões. Numa visão de alto nível, um agente inteligente ou um ciclo MAPE-K podem prover os meios para esta tomada de decisão porém, olhando mais detalhadamente para este agente, pode-se identificar várias decisões sendo tomadas, como: identificar em que contexto o sistema está, que configuração de software será a melhor para aquele contexto ou a reconfiguração necessária para alterar o software.

Nesta proposta de pesquisa, o interesse recai em uma das etapas de um ciclo como o MAPE-K, mais especificamente em como decidir por um plano de reconfiguração a partir de uma configuração objetivo. Para isto, esta decisão será feita por um agente inteligente que emprega uma abordagem baseada em modelos. Diferentemente de outras abordagens baseadas em modelos, este pesquisa considera toda a complexidade de um modelo de componentes reflexivo, bem como a necessidade de que as reconfigurações sejam corretas e consistentes em relação a um conjunto de restrições. Também serão considerados os aspectos relacionados ao componente responsável por executar este plano de reconfiguração sobre o software a ser ajustado, a fim de que a solução proposta possa ser efetivamente empregada.

Neste Capítulo, são feitas as considerações sobre os resultados preliminares obtidos e os esperados, descritos na Seção 5.1. Na Seção 5.2, o domínio de aplicação da solução a ser desenvolvida é especificado e, em seguida, na Seção 5.3, são descritas as tarefas relacionadas com o desenvolvimento da tese e respectivo cronograma de atividades.

5.1 Resultados e contribuições

Dentre os resultados preliminares já obtidos, pode-se destacar, a modelagem do domínio de planejamento para o modelo de componentes reflexivo Fractal, considerando todos as restrições do modelo, e posterior execução do plano de reconfiguração gerado sobre uma aplicação, por meio da API fornecida pelo próprio Fractal.

Dentre os resultados esperados com a continuidade desta pesquisa, tem-se: o desenvolvimento de um mecanismo de reconfiguração automatizado para um modelo de componentes. Os modelos de componentes nos quais este mecanismo será empregado são: o Fractal, usado neste trabalho e o iPOJO[1], que é baseado no OSGi[2], e que está dentro do escopo de um outro trabalho de mestrado do Grupo de Reutilização de Software da COPPE/UFRJ.

Como principal contribuição, espera-se que o mecanismo proposto neste pesquisa seja capaz de apoiar a reconfiguração de um software em tempo de execução, de maneira automatizada e correta, reduzindo a intervenção humana e aumentando a capacidade de reconfiguração de um software. Nesta proposta de pesquisa, considera-se que o software adaptável possui:

- restrições advindas do estilo arquitetural empregado, do domínio da aplicação e do modelo de componentes.
- necessidade de manutenção da consistência de uma reconfiguração.
- dependências de recursos externos a aplicação como bibliotecas e dispositivos físicos.
- possibilidade de desfazer uma reconfiguração caso isto seja necessário.

Apesar de estar fora do escopo desta proposta, a abordagem a ser desenvolvida poderá ser integrada em outras soluções que tratem de responder a outras perguntas relativas a adaptação de software como: quando uma adaptação deve ser ajustada e qual a configuração arquitetural objetivo.

5.2 Domínio da aplicação da abordagem

O domínio das aplicações na qual se pretende empregar uma solução adaptável é o Domínio de Controle de Tráfego Marítimo. O Brasil, sendo signatário das Convenções Internacionais para a Salvaguarda da Vida Humana no Mar e de Busca e

[1] http://felix.apache.org/site/apache-felix-ipojo.html
[2] http://www.osgi.org/

Salvamento Marítimo - *Search and Rescue* (SAR), assumiu o compromisso de mobilizar os meios disponíveis para auxiliar os navios, de qualquer nacionalidade, em situação de emergência dentro da área marítima de sua responsabilidade, conhecida como "Área SAR brasileira".

Esta área cobre uma grande parte do Oceano Atlântico Sul e um pequeno trecho do Atlântico Norte, como pode ser visto na Figura 5.1, onde a área em questão é delimitada pela linha de costa do Brasil e um polígono em negrito sobre o oceano, e que indica os limites externos da área.

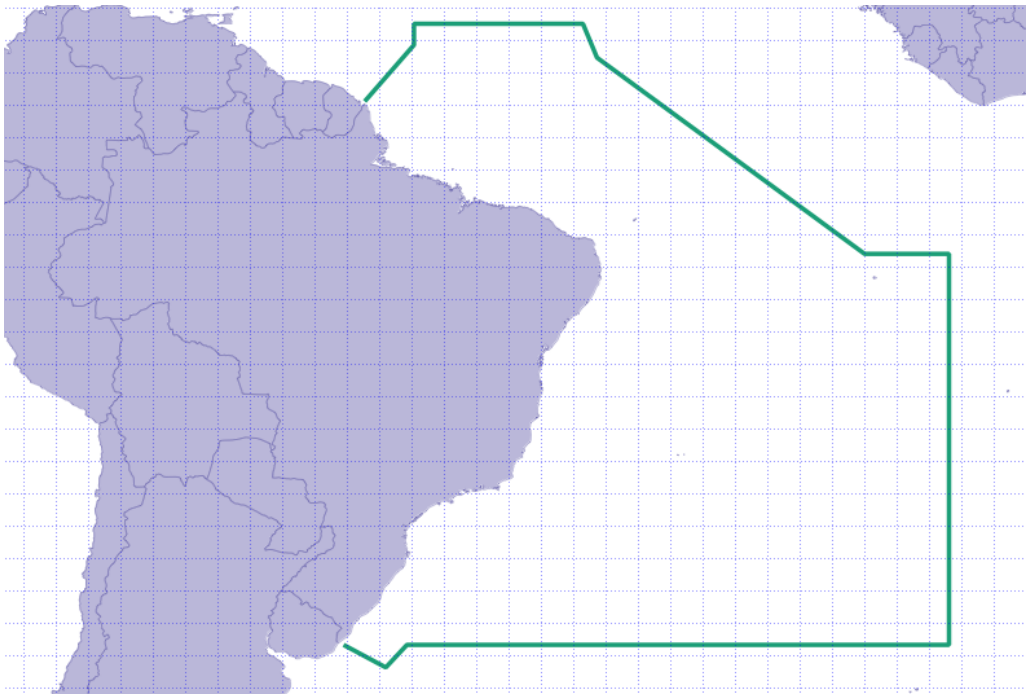


Figura 5.1: Área SAR sob responsabilidade do Brasil.

5.2.1 Sistemas de Informação

Para o cumprimento deste compromisso, a Marinha do Brasil possui vários sistemas de informação que tratam do acompanhamento dos navios navegando na área SAR Brasileira, estejam os navios se dirigindo a algum porto brasileiro ou não. Os sistemas que tratam deste tipo de informação são:

1. Sistema de Controle de Tráfego Marítimo (SISTRAM III);
2. Sistema de Acompanhamento e Identificação de Navios à Longa Distância *Long Range Identification and Tracking* (LRIT);
3. Programa Nacional de Rastreamento de Embarcações Pesqueiras por Satélite (PREPS);
4. Sistema de Apresentação Gráfica e Banco de Dados (SAGBD).

Juntos estes sistemas apresentam as embarcações que: trafegam na área SAR Brasileira e que tenham aderido ao sistema de controle brasileiro, os navios mercantes que estão saindo ou tem como destino algum porto brasileiro, as embarcações pesqueiras de bandeira brasileira e as embarcações da Marinha. Isso significa, em média 1500, embarcações em acompanhamento a qualquer instante de tempo.

Há dois principais sistemas que são clientes destas informações:

1. **Sistema de Apoio à Decisão para Busca e Salvamento** (SAD-SAR) - Ele é o sistema responsável por apoiar o planejamento das buscas a alguma embarcação que tenha declarado emergência. A partir de um ponto geográfico, o sistema fornece o melhor plano de busca a ser efetuado, isto é, o plano que tem a maior probabilidade de encontrar a embarcação desaparecida. Para isto, o sistema utiliza os dados ambientais da área, como correntes marinhas, direção e velocidade do vento, que estão armazenados nele próprio, a partir de dados históricos. Após este primeiro plano de busca, o mesmo é refinado depois do recebimento das condições ambientais atualizadas pelo centro responsável por esta tarefa, e um novo plano é gerado. Com este plano, as embarcações que estejam navegando nas proximidades podem ser acionadas para prestar socorro ou ajudar nas buscas. Para acionar as embarcações, o SAD-SAR realiza uma consulta aos sistemas de informação anteriores, considerando o instante de tempo em que ocorreu a emergência e os trajetos das embarcações, a fim de determinar que embarcação está mais próxima do local onde possivelmente se encontra a embarcação em emergência.

2. **Sistema Naval de Comando de Controle** (SisNC2) - Nas operações de patrulha ou controle de área marítima, ele recebe todas as informações de tráfego sobre uma determinada área geográfica de interesse e por um determinado intervalo de tempo, a fim de distribuir estas informações para os meios envolvidos no controle desta área e aos órgãos de interesse.

Ao longo do tempo, diferentes necessidades de informação e respectivas prioridades podem estar ativas e uma solução adaptável pode contribuir em diversos aspectos não-funcionais como desempenho, confiabilidade e robustez.

5.2.2 A Solução Adaptável

Os sistemas que fornecem as informações de tráfego marítimo listados na Seção 5.2.1, possuem diferentes taxas de atualização e modos de funcionamento. Por exemplo, no PREPS, as informações sobre os pesqueiros são enviadas a cada cinco minutos, por meio de um canal de comunicações satelital. Este canal ape-

nas transmite a informação do barco para o sistema e, além de posição e velocidade, o barco pode enviar um sinal de emergência. Os pesqueiros cadastrados também possuem dados sobre o responsável pela embarcação, caso seja necessário contactá-los. Já no SITRAM III, ele recebe informações de outros sistemas como o Sistema de Identificação Automática ou *Automatic Identification System* (AIS), que é um sistema de acompanhamento automático empregado em navios para a identificação e localização por meio da troca de dados eletrônicos com outros navios próximos e bases costeiras AIS. Este sistema tem como uma das principais tarefas evitar a colisão de navios no mar e um navio equipado com AIS transmite seus dados em intervalos que variam de 2 a 10 segundos, dependendo da sua velocidade na superfície. O sistema LRIT permite a comunicação em ambos os sentidos, ou seja, do navio para terra e vice-versa. Um determinado navio pode ter a sua posição solicitada a qualquer momento e em qualquer localização, pois a sua cobertura é global. A frequência com que esta posição é fornecida também pode ser alterada e há diversas situações em que um navio deve enviar a sua posição a um determinado país, por exemplo, quando um navio se destina a algum porto sob a jurisdição deste país.

A solução adaptável servirá para apoiar os clientes - **SAD-SAR** e **SisNC2** - no uso das informações provenientes dos sistemas anteriormente citados. Em especial, o **SisNC2** possui algumas características que podem variar de acordo com a operação em andamento e a situação em que ela se encontra, como diferentes prioridades entre os objetivos do sistema. A solução observará o desenvolvimento baseado em componentes e, para isto, empregará um modelo de componentes reflexivo e que tenha independência de linguagem de implementação.

5.2.3 Validação da Solução

A validação do mecanismo a ser desenvolvido neste pesquisa é composta das seguintes avaliações:

- Validar a obtenção do estado inicial, isto é, se os predicados do problema de planejamento representam os conceitos existentes;
- Validar o plano de reconfiguração por meio da execução do mesmo sobre a aplicação e verificar se o estado atingido corresponde à configuração objetivo;
- Gerar planos de reconfiguração com erros decorrentes da não satisfação das precondições de uma ação, com o intuito de recuperar a saída do controlador e verificar se ela está condizente com a causa do erro;
- Efetuar a reconfiguração da aplicação após ser obtido um erro, ou seja, efetuar um *rollback* ou tentar atingir a configuração objetivo novamente.

- Gerar situações que devem ser tratadas pelo controlador como planos que nunca terminam e mais de um plano para ser executado.

5.3 Tarefas

As tarefas a serem realizadas para validar a solução proposta estão relacionadas a arquitetura descrita na Seção 4.2.

Nr	Descrição
1	Implementação do Planejador
2	Implementação do Controlador e integração com Planejador
3	Implementação do Software Adaptável e integração com Controlador
4	Implementação do Repositório de Componentes
5	Integração da Arquitetura

Tabela 5.1: Tarefas

As macro tarefas a serem realizadas estão descritas na Tabela 5.1. Delas, cabe ressaltar que atualmente em relação ao planejador, na Tarefa 1, o domínio de planejamento vem sendo modelado, refinado e, posteriormente, uma versão modificada do planejador JSHOP2 será implementada para permitir a obtenção do estado inicial e geração do plano de reconfiguração sem utilizar arquivos textuais, como é feito atualmente.

O cronograma das atividades está descrito na Figura 5.2. As siglas 1T, 2T, 3T e 4T representam o primeiro, segundo, terceiro e quarto trimestres, respectivamente.

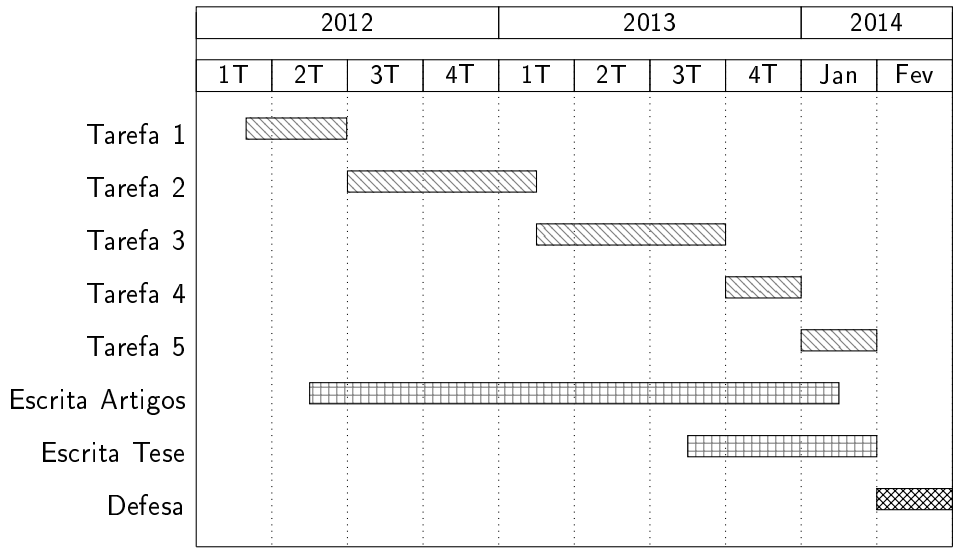


Figura 5.2: Cronograma de atividades

Durante esta pesquisa, serão redigidos artigos científicos ou relatos de experiências, sempre que alguma contribuição bem definida for detectada. Esses

artigos serão direcionados preferencialmente para os seguintes eventos:

1. Simpósio Brasileiro de Engenharia de Software (SBES);
2. Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS);
3. International Conference on Autonomic Computing (ICAC);
4. International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE);
5. International Workshop on Computational Intelligence in Software Engineering (CISE);
6. Workshop on Adaptive and Reflective Middleware (ARM);
7. IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO);
8. International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).

Referências Bibliográficas

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*. Washington, DC, EUA: IEEE Computer Society, 2003. (ICTAI '03), p. 39–. 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*, Kluwer Academic Publishers, Hingham, MA, EUA, v. 15, p. 265–281, September 2007. ISSN 0963-9314. Disponível em: <http://dl.acm.org/citation.cfm?id=1286061.1286070>.

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

BERTOLI, P.; CIMATTI, A.; LAGO, U. D.; PISTORE, M. Extending pddl to nondeterminism, limited sensing and iterative conditional plans. In: _____. *Proceedings of ICAPS'03 Workshop on PDDL*. [S.l.]: Citeseer, 2003.

BRACHMAN, R.; LEVESQUE, H. *Knowledge Representation and Reasoning (The Morgan Kaufmann Series in Artificial Intelligence)*. [S.l.]: Morgan Kaufmann, 2004. Hardcover. ISBN 1558609326.

BRUNETON, E.; COUPAYE, T.; LECLERCQ, M.; QUéMA, V.; STEFANI, J.-B. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, EUA, v. 36, p. 1257–1284, September 2006. ISSN 0038-0644. Disponível em: <http://dl.acm.org/citation.cfm?id=1152333.1152345>.

BUCKLEY, J.; MENS, T.; ZENGER, M.; RASHID, A.; KNIESEL, G. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, v. 17, n. 5, p. 309–332, 2005. ISSN 1532-0618.

CHENG, B. H. C.; LEMOS, R. de; GIESE, H.; INVERARDI, P.; MAGEE, J.; ANDERSSON, J.; BECKER, B.; BENCOMO, N.; BRUN, Y.; CUKIC, B.;

SERUGENDO, G. D. M.; DUSTDAR, S.; FINKELSTEIN, A.; GACEK, C.; GEIHS, K.; GRASSI, V.; KARSAI, G.; KIENLE, H. M.; KRAMER, J.; LITOIU, M.; MALEK, S.; MIRANDOLA, R.; MÜLLER, H. A.; PARK, S.; SHAW, M.; TICHY, M.; TIVOLI, M.; WEYNS, D.; WHITTLE, J. Software engineering for self-adaptive systems: A research roadmap. In: CHENG, B. H. C.; LEMOS, R. de; GIESE, H.; INVERARDI, P.; MAGEE, J. (Ed.). *Software Engineering for Self-Adaptive Systems*. Springer, 2009. (Lecture Notes in Computer Science, v. 5525), p. 1–26. ISBN 978-3-642-02160-2. Disponível em: <http://dblp.uni-trier.de/db/conf/dagstuhl/adaptive2009.html>.

COUNCILL, B.; HEINEMAN, G. T. Definition of a software component and its elements. In: _____. Boston, MA, EUA: Addison-Wesley Longman Publishing Co., Inc., 2001. p. 5–19. ISBN 0-201-70485-4. Disponível em: <http://dl.acm.org/citation.cfm?id=379381.379438>.

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*, Springer Paris, v. 64, p. 45–63, 2009. ISSN 0003-4347. 10.1007/s12243-008-0073-y. Disponível em: <http://dx.doi.org/10.1007/s12243-008-0073-y>.

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

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.*, ACM, New York, NY, EUA, v. 1, p. 223–259, December 2006. ISSN 1556-4665. Disponível em: <http://doi.acm.org/10.1145/1186778.1186782>.

FOX, M.; LONG, D. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, v. 20, p. 61–124, 2003.

GARLAN, D.; CHENG, S.-W.; HUANG, A.-C.; SCHMERL, B.; STEENKISTE, P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, IEEE Computer Society, Los Alamitos, CA, EUA, v. 37, p. 46–54, 2004. ISSN 0018-9162.

GEFFNER, H. The model-based approach to autonomous behavior: A personal view. In: FOX, M.; POOLE, D. (Ed.). *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence, AAAI, Atlanta, Georgia, EUA*. [S.l.]: AAAI Press, 2010.

HUEBSCHER, M. C.; MCCANN, J. A. A survey of autonomic computing degrees, models, and applications. *ACM Comput. Surv.*, ACM, New York, NY, EUA, v. 40, p. 7:1–7:28, August 2008. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1380584.1380585>.

ILGHAMI, O.; NAU, D. S. *A General Approach to Synthesize Problem-Specific Planners*. [S.l.], 2003.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, EUA, v. 36, p. 41–50, January 2003. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2003.1160055>.

KON, F.; COSTA, F.; BLAIR, G.; CAMPBELL, R. H. The case for reflective middleware. *Commun. ACM*, ACM, New York, NY, EUA, v. 45, p. 33–38, June 2002. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/508448.508470>.

KRAMER, J.; MAGEE, J. Self-managed systems: an architectural challenge. In: *2007 Future of Software Engineering*. Washington, DC, EUA: IEEE Computer Society, 2007. (FOSE '07), p. 259–268. ISBN 0-7695-2829-5. Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.19>.

LADDAGA, R. Self adaptive software problems and projects. In: *Proceedings of the Second International IEEE Workshop on Software Evolvability*. Washington, DC, EUA: IEEE Computer Society, 2006. p. 3–10. ISBN 0-7695-2698-5. Disponível em: <http://portal.acm.org/citation.cfm?id=1173703.1174301>.

LAU, K.-K.; WANG, Z. Software component models. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, EUA, v. 33, p. 709–724, October 2007. ISSN 0098-5589. Disponível em: <http://dl.acm.org/citation.cfm?id=1314033.1314048>.

LÉGER, M.; LEDOUX, T.; COUPAYE, T. Reliable dynamic reconfigurations in a reflective component model. In: *CBSE*. [S.l.: s.n.], 2010. p. 74–92.

LEMOS, R. de; GIESE, H.; MÜLLER, H.; SHAW, M.; ANDERSSON, J.; BARESI, L.; BECKER, B.; BENCOMO, N.; BRUN, Y.; CIKIC, B.; DESMARAIS, R.; DUSTDAR, S.; ENGELS, G.; GEIHS, K.; GOESCHKA, K. M.; GORLA, A.; GRASSI, V.; INVERARDI, P.; KARSAI, G.; KRAMER, J.; LITOIU, M.; LOPES, A.; MAGEE, J.; MALEK, S.; MANKOVSKII, S.; MIRANDOLA, R.; MYLOPOULOS, J.; NIERSTRASZ, O.; PEZZÈ, M.; PREHOFER, C.; SCHÄFER, W.; SCHLICHTING, W.; SCHMERL, B.; SMITH, D. B.; SOUSA, J. P.; TAMURA, G.; TAHVILDARI, L.; VILLEGAS, N. M.; VOGEL, T.; WEYNS, D.; WONG, K.; WUTTKE, J. Software engineering for self-adpaptive systems: A second research roadmap. In: LEMOS, R. de; GIESE, H.; MÜLLER, H.; SHAW, M. (Ed.). *Software Engineering for Self-Adaptive Systems*. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. (Dagstuhl Seminar Proceedings, 10431). ISSN 1862-4405. Disponível em: <http://drops.dagstuhl.de/opus/volltexte/2011/3156>.

MCILROY, M. D. Mass produced software components. In: _____. *Software Engineering Concepts and Techniques*. NATO Science Affairs Division, 1968.

Proc. NATO Conf. on Software Engineering, n. October 1968, p. 138–155. Disponível em: <http://www-st.inf.tu-dresden.de/Lehre/SS04/st/slides/16b-transconsistent-composition.pdf>.

MCKINLEY, P. K.; SADJADI, S. M.; KASTEN, E. P.; CHENG, B. H. C. Composing adaptive software. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, EUA, v. 37, p. 56–64, July 2004. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2004.48>.

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*. New York, NY, EUA: ACM, 2008. (ULSSIS '08), p. 23–26. ISBN 978-1-60558-026-5. Disponível em: <http://doi.acm.org/10.1145/1370700.1370707>.

NAU, D.; AVILA, H. M. noz; 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*. [S.l.: s.n.], 2001. p. 425–430.

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

NITTO, E. D.; ROSENBLUM, D. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In: *Proceedings of the 21st international conference on Software engineering*. New York, NY, EUA: ACM, 1999. (ICSE '99), p. 13–22. ISBN 1-58113-074-0. Disponível em: <http://doi.acm.org/10.1145/302405.302406>.

OGATA, K. *Modern Control Engineering*. [S.l.]: Prentice Hall, 2010. ISBN 9780136156734.

OQUENDO, F. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, EUA, v. 29, p. 1–14, May 2004. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/986710.986728>.

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*, IEEE Educational Activities Department, Piscataway, NJ, EUA, v. 14, p. 54–62, May 1999. ISSN 1541-1672.

OREIZY, P.; MEDVIDOVIC, N.; TAYLOR, R. N. Architecture-based runtime software evolution. In: *Proceedings of the 20th international conference*

on Software engineering. Washington, DC, EUA: IEEE Computer Society, 1998. (ICSE '98), p. 177–186. ISBN 0-8186-8368-6. Disponível em: <http://dl.acm.org/citation.cfm?id=302163.302181>.

ROTHENBERG, J. The nature of modeling. In: WIDMAN, L. E.; LOPARO, K. A.; NIELSEN, N. R. (Ed.). *Artificial Intelligence, Simulation and Modeling*. [S.l.]: John Wiley & Sons, 1989. p. 75–92.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. [S.l.]: Pearson Education, 2010. I-XVIII, 1-1132 p. ISBN 978-0-13-207148-2.

SALEHIE, M.; TAHVILDARI, L. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, ACM, New York, NY, EUA, v. 4, n. 2, p. 1–42, 2009. ISSN 1556-4665.

SICARD, S.; BOYER, F.; PALMA, N. D. Using components for architecture-based management: the self-repair case. In: *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE '08), p. 101–110. ISBN 978-1-60558-079-1. Disponível em: <http://doi.acm.org/10.1145/1368088.1368103>.

SYKES, D. *Autonomous Architectural Assembly And Adaptation*. Tese (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>.

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*. New York, NY, EUA: ACM, 2007. (SAVCBS '07), p. 15–21. ISBN 978-1-59593-721-6. Disponível em: <http://doi.acm.org/10.1145/1292316.1292318>.

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*. New York, NY, EUA: ACM, 2008. (SEAMS '08), p. 1–8. ISBN 978-1-60558-037-1. Disponível em: <http://doi.acm.org/10.1145/1370018.1370020>.

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

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. New York, NY, EUA: ACM, 2010. (ASE '10), p. 467–476. ISBN 978-1-4503-0116-9. Disponível em: <http://doi.acm.org/10.1145/1858996.1859092>.

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

TRINIDAD, P.; RUIZ-CORTÉS, A.; NA, J. P.; BENAVIDES, D. Mapping feature models onto component models to build dynamic software product lines. *International Workshop on Dynamic Software Product Line*, 2007.

VILLEGAS, N.; MÜLLER, H.; TAMURA, G.; DUCHIEN, L.; CASALLAS, R. A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems. In: ACM SIGSOFT / IEEE TCSE. *SEAMS 2011*. Honolulu, Hawaii, États-Unis: ACM, 2011. (SEAMS '11, v. 1), p. 80–89. Disponível em: <http://hal.inria.fr/inria-00578337/en/>.

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

WEISZFLOG, W. (Ed.). *MICHAELIS: Moderno Dicionário da Língua Portuguesa.* [S.l.]: Companhia Melhoramentos, 1998.

WELD, D. S. Recent advances in AI planning. *AI MAGAZINE*, v. 20, p. 93–123, 1999.