UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

MODELAGEM EXECUTÁVEL DE SISTEMAS DISTRIBUÍDOS EM JAVA

AFRÂNIO MANGUEIRA LIMA DE ASSIS

CAMPINA GRANDE – PB
MARÇO DE 2006

Modelagem Executável de Sistemas Distribuídos em Java

Afrânio Mangueira Lima de Assis

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Dalton Dario Serey Guerrero

(Orientador)

Jorge César Abrantes de Figueiredo

(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

A848m Assis, Afrânio Mangueira Lima de
2006 Modelagem executável de sistemas distribuídos em JAVA / Afrânio Mangueira Lima de Assis. — Campina Grande, 2006.
129.: il.

Referências.
Dissertação (Mestrado em Informática) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Dalton Dario S. Guerrero e Jorge César A. de Figueiredo.

1— Engenharia de Software 2 – Modelagem Comportamental 3 – Verificação Formal I — Título

CDU 004.415.22/.5

"MODELAGEM EXECUTÁVEL DE SISTEMAS DISTRIBUÍDOS EM JAVA"

AFRÂNIO MANGUEIRA LIMA DE ASSIS

DISSERTAÇÃO APROVADA EM 27.02.2006

PROF. DALTON DARIO SEREY GUERRERO , D.Sc
Orientador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO , D.Sc
Orientador

PROF. BERNARDO LULA JÚNIOR , Dr.
Examinador

PROFª CHRISTINA VON FLACH GARCIA CHAVEZ, Drª
Examinadora

CAMPINA GRANDE – PB

Resumo

Com o objetivo de minimizar a quantidade de erros existentes em sistemas de computador, desenvolvedores procuram inserir no seu processo de desenvolvimento atividades que contribuam para o aumento da qualidade dos sistemas. Dentre essas atividades encontramos a modelagem comportamental. Essa atividade permite a criação de modelos que refletem o comportamento do sistema em desenvolvimento, sendo, portanto, modelos passíveis de execução.

A execução de um modelo simula a execução do sistema modelado. Essa simulação possibilita a validação do sistema antes mesmo de implementá-lo, através da detecção e correção de erros que possam existir no modelo. A correção desses erros é realizada a um baixo custo, se comparado ao custo de corrigir os mesmos erros após ter implementado o sistema. Caso a linguagem utilizada na modelagem comportamental seja formal, ou seja, definida matematicamente, podemos verificar nossos modelos através do emprego de técnicas de verificação formal.

Este documento apresenta o resultado de nosso trabalho, uma liguagem formal de modelagem comportamental, denominada CROMOL. Ela permite a criação de modelos formais para um sistema, mesmo que o modelador não possua conhecimentos matemáticos avançados. Durante o seu desenvolvimento atacamos vários problemas encontrados nas linguagens formais existentes na literatura e focamos na modelagem de sistemas implementados em Java. Como resultado, obtivemos uma linguagem que oferece suporte à orientação a objetos, a sistemas distribuídos e é estruturalmente próxima à linguagem de programação Java, facilitando o processo de mapeamento entre código e modelo.

Abstract

With the aim to minimize the errors in computer systems, developers try to include in their development process activities which will contribute to the increase in the quality of systems. Among these activities, one finds the behavior modeling, which allows the creation of models that reflect the developing system's behavior, thus being models which can be executed.

The execution of a model simulates the execution of the modeled system. This simulation makes it possible to validate the system even before its implementation through the detection and correction of errors that might exist in the model. The correction of these errors is carried out at low cost if compared to the costs to correct the same errors after having implemented the system. If the language used in the behavior modeling is formal, i.e. mathematically defined, we can verify our models through the use of techniques of formal verification.

This document presents the results of our work: a formal language of behavior modeling called CROMOL, which permits the creation of formal models for a system, even if the designer does not have advanced mathematical knowledge. During its development, we faced many problems found in formal languages which exist in the literature and focused on modeling Java-implemented systems. As a result, we got a language that offers support to the orientation of objects, to distributed systems and is structurally close to the programming language Java, making the mapping process between the code and model easier.

Agradecimentos

Agradeço a toda minha família. Aos meus pais, Demar e Isete, pelo incentivo, ajuda financeira e apoio emocional. A minha esposa Adriane pela compreensão e cuidados. Ao meu irmão Alex, que me incentivou bastante a ingressar no mestrado e sempre teve uma preocupação especial com minha carreira profissional, contribuindo, inclusive financeiramente, para o meu desenvolvimento. E a minha irmã Alinne, por sua amizade.

Agradeço também aos colegas de trabalho. Aos meus orientadores, Jorge e Dalton, pelos conselhos e conversas esclarecedoras. À secretária da COPIN, Aninha, por seu empenho na solução de problemas burocráticos. E aos alunos de mestrado da COPIN que me ajudaram direta ou inderetamente, em especial, Elthon, Rogério e Daniel.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização e Motivação

No mundo é cada vez mais comum o uso de sistemas de computador. Gradativamente, os usuários desses sistemas estão se tornando mais exigentes, cobrando dos desenvolvedores sistemas mais estáveis e confiáveis. Os desenvolvedores reagem a esta cobrança procurando inserir no seu processo de desenvolvimento atividades que contribuam para o aumento da qualidade dos sistemas desenvolvidos. Dentre essas atividades, temos a modelagem comportamental do sistema, que consiste na utilização de uma linguagem de modelagem para a criação de modelos que descrevem o comportamento do sistema a ser desenvolvido. Exemplos de modelos comportamentais simples são os diagramas de estados de UML [Larman, 1999]. Além de facilitar o entendimento do problema, os modelos comportamentais permitem a realização de uma validação inicial do sistema, antes mesmo de implementá-lo. Como resultado, erros são descobertos e corrigidos antecipadamente.

Quando os sistemas que estão sendo desenvolvidos são distribuídos [Tanenbaum e Steen, 2002], a atividade de modelagem comportamental precisa ser bastante rigorosa, pois este tipo de sistema tem natureza complexa e esconde erros sutis. O rigor desejado pode ser alcançado através da utilização de linguagens formais de modelagem comportamental. O termo "formal" aqui significa que a linguagem é fortemente embasada em conceitos matemáticos, isto não implica que o modelador necessite de conhecimentos matemáticos avançados para criar seus modelos. Como benefício, essas linguagens pos-

sibilitam a utilização de técnicas de verificação formal [Clarke, 1999] durante o processo de validação do sistema. Neste caso, como estamos lidando com a validação do sistema a partir de modelos comportamentais, as técnicas às quais nos referimos são técnicas de verificação formal baseadas em modelos [Katoen, 1999]. Essas técnicas consistem na exploração de um conjunto de estados, gerado a partir da execução de um modelo, em busca de comportamentos indesejáveis. Estes comportamentos representam erros que dificilmente são encontrados quando utilizamos uma técnica tradicional de verificação [Monin, 2003], tal como testes.

A utilização de técnicas de verificação formal exige um grande número de iterações e modificações tanto no modelo como código, pois cada erro descoberto deve ser corrigido em ambos os artefatos [Katoen, 1999]. Durante esse processo devemos sempre garantir a conformidade entre código e modelo [Aldrich, 2002; Shaw e Garlan, 1996]. Quando desenvolvemos modelos que possuam uma certa proximidade estrutural com o código, tanto o processo de iteração e modificação é simplificado, como a conformidade é mais facilmente garantida. Outros benefícios decorrentes dessa proximidade entre modelo e implementação são listados a seguir.

- Conclusões obtidas durante a verificação de modelos podem ser mapeadas com maior facilidade para a implementação. Por exemplo, podemos facilmente mapear *traces* resultantes da verificação para o código.
- A tarefa de atualização do modelo em relação à implementação e vice-versa pode ser automatizada.
- Os modelos seriam mais facilmente entendidos pelos implementadores, estimulando-os a participarem do processo de modelagem.
- Construído o modelo do sistema, este poderia ser aproveitado, automaticamente ou não, para gerar parte do código fonte durante a fase de implementação.

Infelizmente, a maioria das linguagens formais de modelagem comportamental tem problemas que distanciam estruturalmente os modelos de suas implementações. A seguir apresentamos os principais problemas encontrados:

Ausência de suporte à orientação a objetos - A orientação a objetos nos permite construir modelos com base nas entidades que exercem algum papel no escopo do sistema. Essa abordagem consiste basicamente na identificação dessas entidades, suas responsabilidades e seus atributos. Como resultado, geralmente, obtemos modelos modularizados e mais fáceis de se compreender. Algumas linguagens formais de modelagem existentes não oferecem suporte à orientação a objetos, contrariando a forte tendência atual de utilização deste paradigma no desenvolvimento de sistemas. Exemplos dessas linguagens são Promela [Bérard, 1999] e a linguagem do SMV [McMillan, 2005]. Outras linguagens são orientadas a objetos, mas o modelo de objetos dessas linguagens não reflete o modelo de objetos comumente usado nas linguagens de programação. Em RPOO [Guerrero, 2002; Santos, 2003], por exemplo, todos objetos executam concorrentemente, ou seja, cada um tem seu próprio fluxo de execução. Enquanto que no modelo utilizado em Java a execução concorrente de objetos é possível apenas utilizando-se o conceito de *thread*.

Alto nível de abstração - Algumas linguagens formais de modelagem são mais concretas, pois oferecem construções que possibilitam uma modelagem mais detalhada e próxima da implementação. Outras linguagens, como RPOO e Dynamic Unity [Zimmerman, 2003], permitem apenas a construção de modelos bastante abstratos que desconsideram muitos detalhes de implementação.

Ausência do conceito de local - Em determinados sistemas de computador, durante sua implantação, alguns processos são executados em locais diferentes. Muitas vezes queremos representar isto explicitamente em nossos modelos, pois algumas características inerentes a este contexto precisam ser analisadas. Por exemplo, mensagens trocadas entre processos que executam em locais diferentes podem ser perdidas. Linguagens formais de modelagem como Actors [Agha, 1983] e CSP [Hoare, 1985] não disponibilizam o conceito de local.

Atualmente, uma parcela significante dos sistemas distribuídos desenvolvidos são implementados utilizando a linguagem de programação Java [Gosling, 2000] e o modelo de distribuição de objetos RMI (*Remote Method Invocation*) [Wollrath e Waldo,

2005]. Entretanto, assim como os demais desenvolvedores de sistemas distribuídos, os desenvolvedores desses sistemas não possuem a sua disposição uma linguagem formal de modelagem comportamental que forneça a proximidade estrutural adequada entre modelo e implementação, ou seja, falta uma linguagem formal para modelar o comportamento de sistemas distribuídos, que possua características próximas às da linguagem Java e ofereça suporte ao modelo de distribuição de objetos RMI.

1.2 Declaração do Problema

Diante do contexto e motivação apresentados na Seção 1.1 o problema a ser atacado por este trabalho é:

> *Suprir as necessidades de modelagem dos desenvolvedores de sistemas distribuídos e orientados a objetos, através da disponibilização de uma linguagem formal de modelagem comportamental que permita criar modelos que sejam estruturalmente próximos de suas implementações e possibilite o emprego de técnicas de verificação formal na validação dos sistemas.*

Com o objetivo de direcionar nossos esforços e conseqüentemente obter uma solução mais concreta, trataremos aqui de sistemas distribuídos e orientados a objetos que são implementados utilizando a linguagem de programação Java e o modelo de comunicação entre objetos distribuídos RMI.

1.3 A Linguagem CROMOL

Com o objetivo de solucionar o problema descrito na Seção 1.2 desenvolvemos a linguagem CROMOL. Ela é uma linguagem formal, utilizada para a modelagem de sistemas, que oferece suporte à execução de modelos, à orientação a objetos, a sistemas distribuídos e que é concreta o suficiente para possibilitar que um modelo do sistema esteja estruturalmente próximo de sua implementação.

Como o foco de nosso trabalho é voltado para modelagem de sistemas que serão implementados utilizando Java e RMI, a sintaxe e a semântica de CROMOL têm como alicerce a especificação da linguagem Java [Gosling, 2000] e o modelo de comunicação

entre objetos distribuídos RMI [Wollrath e Waldo, 2005]. Obviamente, como se trata de uma linguagem de modelagem, alguns dos conceitos existentes em Java e RMI foram abstraídos, de forma a reduzir nosso escopo. Nesse sentido, a especificação de CROMOL pode ser vista como resultado de um processo de abstração da especificação de Java e do modelo de distribuição RMI, conforme podemos observar na Figura 1.1.

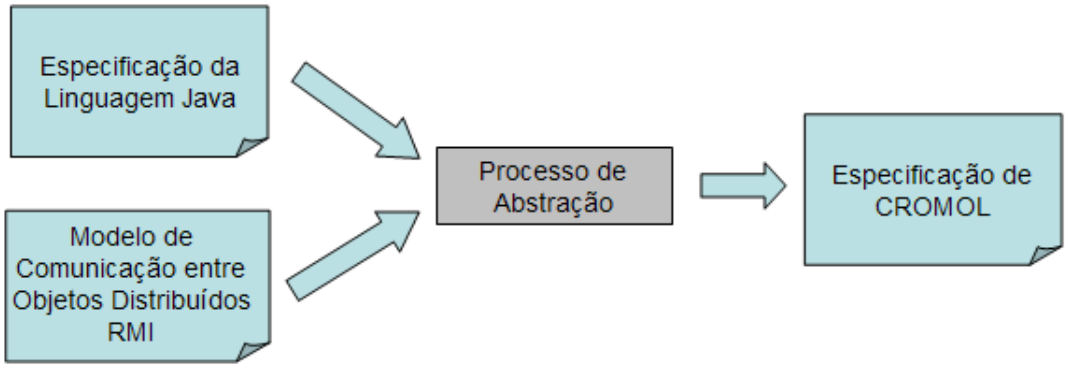

Figura 1.1: A especificação de CROMOL como resultado de um processo de abstração de Java e RMI.

Observe que estamos nos referindo a uma linguagem formal. Portanto, como parte integrante da definição de CROMOL temos a sua formalização, ou seja, definimos matematicamente sua sintaxe e semântica operacional. A formalização é útil àqueles que pretendem desenvolver ferramentas para a linguagem, como também aos que pretendem definir como técnicas de verificação formal podem ser aplicadas aos modelos desenvolvidos.

Durante o processo de desenvolvimento de CROMOL, tivemos uma grande preocupação com relação a sua usabilidade, principalmente porque trata-se de uma linguagem formal [Glass, 2004], assim, tentamos ao máximo simplificá-la, aproximando sua sintaxe à da linguagem de programação Java e não exigindo conhecimentos matemáticos avançados do modelador.

Apesar de CROMOL ter como base Java, ela também pode ser utilizada para modelagem de sistemas que não serão implementados em Java, mas é importante que a linguagem de implementação seja orientada a objetos. Isto é possível porque os conceitos abordados na linguagem são comuns à grande maioria das linguagens de programação orientadas a objetos existentes. Entretanto, o ideal seria uma análise

prévia da linguagem de programação e do modelo de distribuição a serem utilizados durante a fase de implementação, a fim de garantir que exista uma compatibilidade entre esses e CROMOL.

Conforme relatamos anteriormente, um modelo construído em CROMOL pode ser executado, pois estamos nos referindo a modelos comportamentais. Com o propósito de executar automaticamente os modelos criados, desenvolvemos um simulador baseado em modelos. A idéia é que possamos simular o comportamento de um sistema através da execução de seu modelo. Durante a simulação, podemos monitorar o estado dos objetos que compõe o modelo, em busca de comportamentos indesejáveis.

A validação de CROMOL foi realizada através da criação e execução de modelos que representam o comportamento de sistemas reais. No Capítulo 6 mostramos um desses modelos, juntamente com a análise de alguns *traces* obtidos através execução dele no simulador que desenvolvemos. Ainda como forma de contribuir para a validação de nosso trabalho, no Capítulo 2 enumeramos um conjunto de parâmetros que serão utilizados como critérios de validação.

1.4 Contribuições

A partir da execução de todos os possíveis caminhos de um modelo em CROMOL podemos gerar todo o comportamento do modelo, ou seja, o seu espaço de estados. Sobre esse espaço de estados podemos aplicar técnicas de verificação formal com o objetivo de validar o sistema. Como conseqüência, teremos sistemas mais estáveis e confiáveis, beneficiando assim, os desenvolvedores e usuários de sistemas distribuídos e orientados a objetos, principalmente os sistemas desenvolvidos usando Java e RMI. Além deste benefício principal, podemos citar outros benefícios que também são propiciados pela atividade de modelagem usando CROMOL, alguns deles são herdados da tarefa de modelagem não formal:

- A modelagem facilita o entendimento do problema.
- Erros descobertos durante a fase de modelagem são corrigidos a um custo bem menor do que erros descobertos em fases posteriores do processo de desenvolvimento.

- O processo de sincronização entre código e modelo pode ser automatizado.
- Erros descobertos em modelos podem ser facilmente identificados no código.
- Um modelo pode ser usado para gerar código fonte de forma automatizada.

1.5 Estrutura da Dissertação

O restante deste documento está estruturado da seguinte forma:

Capítulo 2 - Conceitos Fundamentais e Trabalhos Relacionados - O objetivo deste capítulo é fornecer embasamento teórico e analisar, de forma crítica, alguns trabalhos que estão relacionados ao nosso, ressaltando as peculiaridades de cada um. A análise é realizada utilizando uma abordagem comparativa, tendo como base um conjunto de parâmetros que nós definimos.

Capítulo 3 - A linguagem CROMOL - Apresentamos a linguagem de modelagem CROMOL. Iniciamos o capítulo familiarizando o leitor com a linguagem através da explicação do comportamento de um simples modelo. Durante a explicação, mostramos o significado das construções do modelo e apresentamos alguns conceitos nos quais a linguagem está fundamentada. Entendido o comportamento desse modelo, nós explicaremos de forma detalhada e informal a sintaxe e a semântica da linguagem.

Capítulo 4 - Formalização de CROMOL - Neste capítulo apresentamos a formalização de CROMOL, ou seja, utilizamos teoria matemática para descrever o significado da linguagem. A idéia é especificar como os modelos devem ser executados. Como passo inicial para a formalização apresentamos a sintaxe formal da linguagem. Em seguida definimos algumas funções auxiliares que tornarão mais simples o processo de formalização. Por último, mostramos a semântica operacional da linguagem.

Capítulo 5 - Simulador CROMOL - Apresentamos o simulador desenvolvido para CROMOL. Iniciamos descrevendo o processo de simulação baseada em modelos

e mostrando como podemos utilizar o Simulador CROMOL neste processo. Em seguida, apresentamos a arquitetura do simulador. Por último, descrevemos o papel do simulador no processo de verificação de modelos.

Capítulo 6 - Estudo de Caso - Nosso estudo de caso consiste na modelagem de um sistema de segurança doméstica chamado *SafeHome*. Esse sistema foi especificado por Pressman no livro intitulado *Engenharia de Software* [Pressman, 2002]. Iniciamos o capítulo apresentando o escopo do *SafeHome*. Em seguida, mostramos o modelo CROMOL que representa o comportamento descrito nesse escopo. Por último, analisamos alguns traces obtidos através do processo de simulação da execução do *SafeHome*.

Capítulo 7 - Conclusões - Este capítulo inicia apresentando as nossas conclusões com relação ao trabalho que desenvolvemos. Em seguida, caracterizamos CROMOL utilizando o mesmo conjunto de parâmetros definidos no Capítulo 2. Por último, apresentamos algumas restrições da linguagem e descrevemos os trabalhos a serem desenvolvidos posteriormente com o objetivo de complementar o trabalho atual e dar continuidade ao nosso projeto de pesquisa.

Apêndice A - Gramática de CROMOL - Mostramos a gramática de nossa linguagem.

Apêndice B - Modelo do Estudo de Caso - Apresentamos por completo o modelo produzido durante nosso estudo de caso.

Capítulo 2

Conceitos Fundamentais e Trabalhos Relacionados

Este capítulo tem como objetivo fornecer embasamento teórico ao leitor e analisar, de forma crítica, trabalhos relacionados ao nosso. Iniciamos o capítulo, descrevendo alguns conceitos que são necessários para a compreensão deste documento. Em seguida, na Seção 2.4, apresentamos os trabalhos relacionados, ressaltando as peculiaridades de cada um. Por último, na Seção 2.5, realizamos uma análise crítica e comparativa destes trabalhos, com base em um conjunto de parâmetros que julgamos ser importantes para a confecção do nosso trabalho.

2.1 Simulação de Sistemas Baseada em Modelos

De acordo com Moreira [Moreira, 2001], o processo de simulação pode ser definido como:

> *O ato de imitar um procedimento real em menor tempo e com menor custo, permitindo um melhor estudo do que vai acontecer e de como consertar erros que gerariam grandes gastos.*

Um modelo é uma abstração de um sistema. Partindo desse princípio, podemos simular (imitar) a execução de um sistema (procedimento real), através da execução

de seu modelo. Analisando a execução desse modelo, podemos descobrir e corrigir erros que, caso contrário, seriam traduzidos para o código do sistema. Assim, estaríamos reduzindo gastos, pois a correção de erros antes da implementação do sistema é realizada a um custo bem menor [Pressman, 2002]. Além disso, falhas causadas durante a operação do sistema quase sempre resultam em danos, muitas vezes irreversíveis, portanto, devem ser evitadas. Por outro lado, mesmo que o sistema já tenha sido implementado, muitas vezes não é viável analisar o seu comportamento em busca de erros. Nestes casos é preferível analisar e corrigir o modelo e atualizar o código de acordo com as correções efetuadas.

2.2 Verificação de Modelos

A verificação de modelos é uma técnica de verificação formal [Katoen, 1999], que tem como princípio verificar a ocorrência de algumas propriedades em um modelo comportamental. Desta forma, podemos verificar o comportamento de um sistema modelado, antes mesmo de implementá-lo. Como se trata de uma técnica de verificação formal, os modelos utilizados devem ser desenvolvidos em uma linguagem com semântica formalmente definida. O arcabouço da técnica de verificação de modelos é ilustrado na Figura 2.1. Um modelo deve ser criado a partir dos requisitos que o sistema precisa atender. Após construir o modelo, é gerado o seu espaço de estados, ou seja, todos os estados possíveis de serem alcançados durante a execução do modelo. Algumas propriedades (comportamentos), cuja existência é desejável no modelo, devem ser elaboradas. O processo de verificação propriamente dito ocorre quando verificamos se o espaço de estados gerado satisfaz às propriedades elaboradas. Como resultado temos uma das duas situações:

- Todas as propriedades foram satisfeitas - O modelo está correto com relação as propriedades especificadas.
- Uma ou mais propriedades não foram satisfeitas - Existem falhas no modelo. Para cada falha é mostrado o *trace* de execução que prova a existência de tal falha.

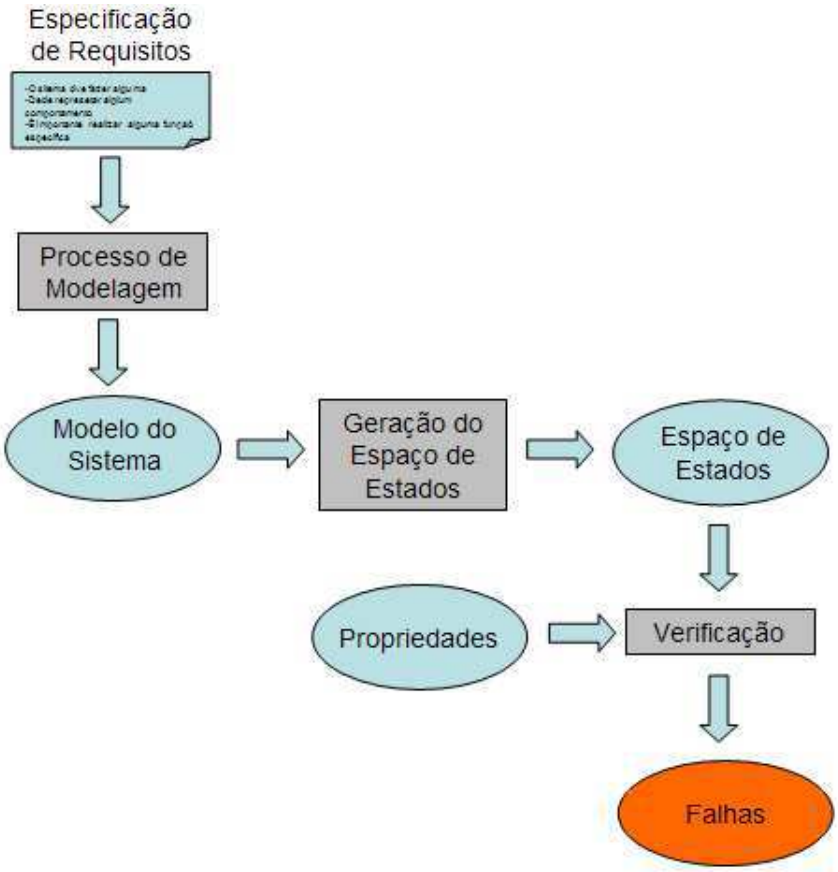

Figura 2.1: Arcabouço da técnica de verificação de modelos.

Observe que a verificação de modelos checa a corretude do modelo criado. Portanto, é importante ter um modelo que reflita com exatidão o comportamento do sistema a ser desenvolvido, ou seja, deve existir uma relação de conformidade entre o modelo e os requisitos do sistema.

2.3 Sistema de Transição

Um sistema de transição [J. W. de Bakker e Zucker, 1985] é composto por regras de transição que especificam todas as mudanças de estado de um sistema, ou seja, o seu comportamento. Quando ocorre uma mudança de estado, dizemos que ocorreu o disparo de uma regra de transição, ou simplesmente, o disparo de uma transição.

Com os estados A, B e C, podemos definir, por exemplo, o seguinte sistema de transição:

$A \rightarrow B$

$B \rightarrow C$

O comportamento especificado por essas regras afirma que a partir do estado A apenas podemos alcançar o estado B e que o estado C apenas pode ser alcançado a partir do B.

2.4 Trabalhos Relacionados

Existem diversas linguagens formais utilizadas na modelagem comportamental de sistemas de computador. Nesta seção, teremos uma visão geral das principais linguagens que influenciaram o desenvolvimento deste trabalho, mostrando as características e peculiaridades de cada linguagem.

2.4.1 Promela

Promela [Dwyer e Hatcliff, 2001] é uma linguagem de modelagem, baseada na sintaxe da linguagem C, utilizada para descrever sistemas distribuídos. Nos modelos criados, os processos são concorrentes, e se comunicam através de variáveis compartilhadas e da passagem de mensagens síncronas ou assíncronas. Modelos Promela podem ser

analisados pelo SPIN (Simple Promela INterpreter). Durante a análise podem ser detectados erros como: *deadlocks*, *race conditions*, *assertion violations*, *safety properties* e *liveness properties*. Os blocos básicos de construção de modelos Promela são [Ireland, 2002]: **processos**, **canais** e **variáveis**.

Processos [Ireland, 2001] são utilizados para modelar o comportamento de componentes concorrentes de nosso sistema, e são por definição, no que se refere a escopo, construções globais. A declaração de um processo contém um identificador, uma lista de parâmetros e uma seqüência de declaração de variáveis locais e *statements*.

Statements podem estar, em um determinado instante, bloqueados ou em condições de execução. Caso mais de um statement possa ser executado em um dado instante, a execução procede de forma não-determinística. Um bloco em Promela pode ser declarado atômico, impedindo assim a intercalação entre processos durante a execução de tal bloco. Todo modelo Promela deve conter o processo *init*. Este processo é o ponto inicial de execução de um modelo, sendo responsável por coordenar a instanciação e a execução de outros processos.

Canais [Ireland, 2002] são utilizados por processos para a troca de mensagens síncronas ou assíncronas, servindo como meios de comunicação. Um canal pode ser definido como local ou global. Cada canal possui um valor associado, que representa a quantidade de mensagens que podem ser armazenadas no *buffer* do canal, ou seja, o tamanho do *buffer*. O tamanho zero indica um canal síncrono, pois, neste caso, qualquer mensagem enviada não pode ser armazenada no canal, e portanto o processo que a enviou deve esperar que esta seja consumida para então prosseguir com sua execução.

Variáveis [Ireland, 2001] são fundamentais para representar o estado de um sistema, e também podem ser utilizadas na comunicação entre processos, uma vez que, além de escopo local a um processo, podem também ter escopo global. As variáveis precisam ser declaradas antes de utilizadas, a sintaxe de declaração segue o estilo da linguagem C. Algumas outras construções herdadas de C também podem ser utilizadas em modelos Promela, tais como: estruturas e *arrays*.

2.4.2 Dynamic UNITY

Dynamic UNITY [Zimmerman, 2003] é um formalismo utilizado para especificação e prova de corretude de sistemas distribuídos dinâmicos. Entende-se por sistemas distribuídos dinâmicos aqueles onde processos podem ser criados ou destruídos e os canais de comunicação entre processos podem mudar ao longo da execução do sistema, ou seja, o sistema não possui uma configuração estática. O Dynamic UNITY é baseado no formalismo UNITY [Ehmety e Paulson, 2003].

O UNITY é utilizado para especificação de sistemas distribuídos estáticos, sendo formado por uma linguagem, um modelo de execução e uma lógica de especificação de propriedades. Um modelo descrito em sua linguagem consiste de um conjunto de variáveis, um conjunto de *assignment statements* de inicialização e um conjunto de múltiplos *assignment statements* protegidos com guardas, representando as transições do sistema.

De acordo com o modelo de execução do UNITY, primeiramente são executados os *statements* de inicialização, o que resulta na inicialização das variáveis do modelo. Em seguida, as transições são repetidamente escolhidas e executadas. A escolha ocorre de forma não-determinística e está sujeita a restrição de que toda transição é escolhida freqüentemente, enquanto durar a execução do modelo.

A lógica de especificação de propriedades do UNITY é temporal linear e consiste de três operadores fundamentais:

- Next - p **next** q significa que todas as transições executadas a partir de um estado onde p é válido resultam em um estado onde q é válido.
- Transient - **transient** p significa que existe uma transição que leva o sistema de um estado onde p é válido para um estado no qual *not* p é válido.
- Initially - **initially** p significa que p é válido em todos os estados iniciais do sistema.

Entretanto, conforme mencionamos anteriormente, modelos especificados usando UNITY devem possuir um conjunto estático de variáveis e transições. Portanto, é in-

conveniente descrever sistemas que tenham comportamento dinâmico. Outro problema é a ausência do conceito de processo.

Foi com o objetivo de solucionar estes problemas que o Dynamic UNITY foi desenvolvido. Assim, podemos vê-lo como uma evolução do UNITY, que passou por mudanças em sua linguagem, na sua lógica e no seu modelo de execução. No entanto, foram impostas algumas restrições aos sistemas que podem ser modelados. Como resultado temos que alguns sistemas que podem ser modelados em um dos formalismo não podem ser modelados no outro.

2.4.3 RPOO

Redes de Petri Orientadas a Objetos (RPOO) [Guerrero, 2002] é uma linguagem formal utilizada para modelagem de sistemas distribuídos. Ela oferece suporte a orientação a objetos e é baseada no formalismo de redes de Petri [Murata, 1989].

Em RPOO o comportamento interno dos objetos de um sistema é modelado utilizando redes de Petri, temos uma rede de Petri para cada classe modelada. Assim, cada instância de uma rede representará um objeto. Isto proporciona um caráter formal e concorrente aos objetos, além de tornar possível uma decomposição orientada a objetos de complexos modelos em redes de Petri.

A comunicação entre objetos ocorre através da troca de mensagens, preservando a encapsulação de informações. Essa interação é especificada nos modelos através de uma linguagem de inscrições. Assim, para construir um modelo RPOO, usamos uma notação gráfica herdada de redes de Petri, com algumas inscrições na forma textual. Essas inscrições são inseridas nas transições do modelo. Quando uma transição dispara as inscrições (ações) associadas àquela transição são executadas. As possíveis ações são:

- Envio de mensagens síncronas
- Envio de mensagens assíncronas
- Recebimento de mensagens
- Criação de novos objetos

Referências a objetos podem ser incluídas no conteúdo de uma mensagem. Ao receber uma referência para um objeto qualquer, o recebedor passa a conhecer este objeto, podendo a partir de então comunicar-se com ele através do envio de mensagens. É importante perceber que a concorrência no modelo ocorre em dois níveis, os objetos são concorrentes entre si e um objeto pode possuir concorrência interna.

2.4.4 Actors

Actors [Agha, 1983] é um modelo de computação concorrente para sistemas distribuídos. Neste modelo, a execução de um sistema é baseada no envio de comunicações e de respostas decorrentes do processamento dessas comunicações. Comunicações e respostas tem como origem e destino um *actor*. Um *actor* é um agente computacional, precursor do objeto, que mapeia cada comunicação de entrada para uma 3-tupla consistindo de:

1. Um conjunto finito de comunicações enviadas para outros *actors*.
2. Um comportamento que será responsável por produzir a resposta para a próxima comunicação de entrada processada.
3. Um conjunto finito de novos *actors*.

Algumas observações aqui são necessárias. Primeiro, o comportamento de um *actor* pode ser influenciado por ações passadas. Segundo, não é presumido nenhuma seqüencialidade nas ações que um *actor* executa e, matematicamente, cada uma das ações é uma função do comportamento do *actor* e de uma comunicação de entrada. Por último, a criação de um *actor* é uma parte integral do modelo computacional.

Um *actor* possui um endereço que identifica sua caixa de entrada de mensagens de forma única dentro do sistema, e quando um novo comportamento é especificado para ele, este endereço permanece inalterado.

Comunicações estão contidas em *tasks*. Uma *task* é uma 3-tupla composta por:

1. Uma *tag* que distingue ela de todas as outras *tasks* no sistema.
2. Um destino que é endereço de *mail* para o qual a comunicação deve ser entregue.

3. Uma comunicação contendo a informação que deve ser disponibilizada para o *actor* no endereço de destino, quando este processar a *task* em questão.

No decorrer de sua execução, um sistemas de *actors* evolui, incluindo novas *tasks* e novos *actors* que são criados como resultado do processamento de *tasks* já existentes no sistema. Todas as *tasks* que já foram processadas e todos os *actors* que não são mais úteis devem ser removidos do sistemas sem afetar o comportamento subseqüente. A configuração de um sistema é definida pelos *actors* que ele contém e pelo conjunto de *tasks* não processadas.

2.4.5 CSP

Communicating Sequential Processes (CSP) [Hoare, 1985] é um formalismo que utiliza uma linguagem, baseada em eventos, para descrever padrões de comportamento. CSP nos permite descrever sistemas como um determinado número de processos que operam de forma independente e se comunicam através de mensagens síncronas enviadas por canais bem definidos. O envio de uma mensagem pode ser visto como um tipo especial de evento e a comunicação de um processo com o seu ambiente é descrita através de uma álgebra de processos.

Durante a modelagem, ou seja, a descrição do comportamento, devemos selecionar quais eventos são importantes para o nosso modelo. Por exemplo, uma máquina de venda de chocolates (MVC) [Hoare, 1985] possui os seguintes eventos:

- *moeda* - a inserção de uma moeda na máquina.
- *chocolate* - a extração de um chocolate da máquina.

O conjunto de nomes de eventos que são considerados importantes em um modelo constituem o alfabeto do modelo. Na MVC o alfabeto seria αMVC = {*moeda*, *chocolate*}. Um modelo apenas contém eventos que estejam em seu alfabeto e a ocorrência de um desses eventos é considerada uma ação instantânea.

O processo com alfabeto A que não responde a nenhum evento é chamado $STOP_A$. O modelo de uma simples máquina de vendas de chocolate (SMVC) que serve dois consumidores e então pára é mostrada a seguir.

(moeda → (chocolate → (moeda → (chocolate → $STOP_{\alpha SMVC}$))))

Neste modelo utilizamos a notação prefixa, ela é útil para representar o comportamento de um processo que parará no futuro. Entretanto, algumas vezes é necessário modelar um comportamento que se repete ao longo do tempo e não tem previsão de término, ou seja, não é possível conhecer a princípio quanto tempo terá o ciclo de vida do processo modelado. Este tipo de modelo é criado utilizando uma notação que permite recursão. Abaixo podemos ver um modelo que utiliza recursão para representar uma máquina de vendas de chocolate (MVC) que serve quantos chocolates forem solicitados.

MVC = (moeda → (chocolate → MVC))

Outro conceito importante de CSP é o de processo composto. Ele é um processo formado a partir de um grupo de outros processos. Neste tipo de processo, as interações entre os elementos do grupo devem ser escondidas, mas interações do grupo com o ambiente devem ser visíveis.

2.5 Análise Comparativa

As linguagens formais apresentadas na seção anterior possuem problemas. Algumas são muito abstratas, distanciando muito o modelo da implementação. Outras não oferecem suporte à orientação a objetos. Existem também linguagens que não permitem o envio de mensagens síncronas, contrariando assim, linguagens de programação freqüentemente utilizadas, como Java. Outro problema comum é a ausência do conceito de local. Nesta seção faremos uma análise crítica e comparativa destas linguagens, evidenciando seus pontos fortes e fracos. A análise será realizada com base nos seguintes parâmetros:

- **Nível de abstração** - As linguagens variam com relação ao nível de abstração. Algumas oferecem construções que possibilitam uma modelagem mais próxima da implementação, ou seja, mais concreta. Outras permitem apenas uma modelagem de forma superficial, não possibilitando a modelagem de alguns detalhes de implementação que tem um papel importante no funcionamento do sistema. Este último tipo de linguagem possui um maior nível de abstração em relação ao

primeiro. Neste trabalho caracterizamos o nível de abstração de uma linguagem como sendo: baixo, intermediário e alto.

- **Tipo de notação da linguagem** - Uma linguagem de modelagem pode ter notação textual ou gráfica. A notação gráfica, geralmente, é assimilada mais facilmente. Por outro lado, uma notação textual simplifica o desenvolvimento da linguagem, como também a implementação de ferramentas de suporte. Outro ponto favorável à notação textual é que a grande maioria das linguagens de programação usam este tipo de notação, e esta similaridade simplifica bastante a tarefa de mapeamento entre modelo e código.

- **Suporte à orientação a objetos** - A orientação a objetos nos permite construir modelos mais próximos do mundo real. Utilizando essa abordagem identificamos entidades, suas responsabilidades e seus atributos. Uma linguagem de modelagem orientada a objetos, além de acompanhar a forte tendência atual da engenharia de software, resulta em modelos de fácil compreensão.

- **Suporte ferramental** - Ferramentas automatizam ou auxiliam diversas tarefas, tais como: criação e execução de modelos, geração de espaços de estados, verificação de propriedades e geração de código. A ausência de ferramentas dificulta o uso prático de uma linguagem. Ao suporte ferramental de um formalismo atribuímos um desses três valores: alto, intermediário e baixo.

- **Suporte a mensagem síncronas** - Na comunicação entre processos, em alguns casos, é preciso aguardar que uma mensagem enviada por um processo seja consumida pelo processo de destino.

- **Suporte a mensagem assíncronas** - Na comunicação entre processos, algumas vezes, não é necessário aguardar que uma mensagem enviada seja consumida.

- **Proximidade a Java** - Conforme mencionado anteriormente, neste trabalho estamos focados na modelagem de sistemas implementados em Java. Portanto, é fundamental analisarmos quais linguagens formais possuem construções próximas as da linguagem de programação Java.

- **Suporte ao conceito de local** - Em sistemas distribuídos, processos executam em locais diferentes. Quando criamos nossos modelos, muitas vezes, queremos representar isto explicitamente, pois algumas peculiaridades relativas a este contexto precisam ser analisadas.

A seguir mostramos os valores desses parâmetros para cada uma das linguagens apresentadas na Seção 2.4. A Tabela 2.1 mostra os valores para Promela, a Tabela 2.2 para Dynamic UNITY, a Tabela 2.3 para RPOO, a Tabela 2.4 para Actors e a Tabela 2.5 para CSP.

Parâmetro	**Valor**
Nível de abstração	Baixo
Tipo de notação da linguagem	Textual
Suporte a orientação a objetos	Não
Suporte ferramental	Alto
Suporte a mensagem síncronas	Sim
Suporte a mensagem assíncronas	Sim
Proximidade a Java	Não
Suporte ao conceito de local	Não

Tabela 2.1: Valores dos parâmetros para Promela

Parâmetro	**Valor**
Nível de abstração	Intermediário
Tipo de notação da linguagem	Textual
Suporte a orientação a objetos	Não
Suporte ferramental	Baixo
Suporte a mensagem síncronas	Não
Suporte a mensagem assíncronas	Sim
Proximidade a Java	Não
Suporte ao conceito de local	Não

Tabela 2.2: Valores dos parâmetros para Dynamic UNITY

Parâmetro	**Valor**
Nível de abstração	Intermediário
Tipo de notação da linguagem	Gráfica
Suporte a orientação a objetos	Sim
Suporte ferramental	Intermediário
Suporte a mensagem síncronas	Sim
Suporte a mensagem assíncronas	Sim
Proximidade a Java	Não
Suporte ao conceito de local	Não

Tabela 2.3: Valores dos parâmetros para RPOO

Parâmetro	**Valor**
Nível de abstração	Alto
Tipo de notação da linguagem	Textual
Suporte a orientação a objetos	Não
Suporte ferramental	Baixo
Suporte a mensagem síncronas	Sim
Suporte a mensagem assíncronas	Sim
Proximidade a Java	Não
Suporte ao conceito de local	Não

Tabela 2.4: Valores dos parâmetros para Actors

Com relação aos valores apresentados é importante destacar que os objetos de RPOO são unicamente concorrentes, ou seja, cada objeto representa uma linha de execução. Este fato distância o conceito de objetos em RPOO daquele conceito que nós conhecemos em linguagens de programação como Java, onde objetos podem se comportar de forma seqüencial. Também é importante destacar que nenhuma das linguagens apresentadas tem proximidade estrutural à linguagem de programação Java e nem suporte ao conceito de local.

Se fôssemos definir, em função dos parâmetros apresentados, uma linguagem ideal para modelagem de sistemas distribuídos a serem implementados em Java, teríamos

Parâmetro	**Valor**
Nível de abstração	Alto
Tipo de notação da linguagem	Textual
Suporte a orientação a objetos	Não
Suporte ferramental	Intermediário
Suporte a mensagem síncronas	Sim
Suporte a mensagem assíncronas	Não
Proximidade a Java	Não
Suporte ao conceito de local	Não

Tabela 2.5: Valores dos parâmetros para CSP

uma linguagem com baixo nível de abstração; notação textual[1]; suporte à orientação a objetos; ampla disponibilidade de ferramentas; mensagens síncronas; mensagens assíncronas; proximidade estrutural à linguagem Java; e suporte explícito ao conceito de local. Comparando as linguagens apresentadas, temos que a que mais se aproxima de satisfazer a esse conjunto de valores é Promela, pois preenche adequadamente cinco dos oito parâmetros. Utilizando outra abordagem, por exemplo, se definirmos que a orientação a objetos é muito mais importante que os demais parâmetros, a linguagem que mais se aproximaria de nosso ideal seria RPOO.

[1]Uma linguagem com notação gráfica que reflita as construções de Java também atenderia as nossas necessidades, entretanto o custo de desenvolvimento tem uma magnitude maior

Capítulo 3

A Linguagem CROMOL

Este capítulo tem como meta aproximar o leitor da linguagem de modelagem CROMOL. O objetivo é que o leitor possa aprender detalhes sintáticos e semânticos da linguagem. Durante todo o capítulo apresentamos a linguagem utilizando uma abordagem informal, mas bastante coesa. A formalização da linguagem é apresentada no Capítulo 4.

Iniciamos o capítulo explicando o comportamento de um modelo bastante simples para que leitor possa ter uma idéia concreta com relação a linguagem. No decorrer da explicação, mostramos o significado das construções do modelo e apresentamos alguns conceitos nos quais a linguagem está fundamentada. Entendido o comportamento desse modelo, iremos nos aprofundar na sintaxe da linguagem, mostrando palavras reservadas, operadores e construções. Por último, mostraremos detalhadamente a semântica da linguagem, explicando o que acontece quando cada uma das suas instruções é executada.

3.1 Compreendendo um modelo em CROMOL

Nesta seção explicaremos um simples modelo em CROMOL, denominado Cliente x Servidor. Durante a explicação, alguns conceitos nos quais CROMOL está fundamentada serão tratados à medida que se tornarem necessários para a compreensão do modelo. Esses conceitos descrevem o modelo de orientação a objetos e o de distribuição de objetos de CROMOL. Conforme antecipamos na Seção 1.3, o modelo de orientação

a objetos tem como alicerce o modelo utilizado na linguagem de programação Java e o modelo de comunicação entre objetos distribuídos é fortemente baseado em RMI (*Remote Method Invocation*).

Antes de iniciar a explicação do modelo Cliente x Servidor, precisamos compreender a organização de um modelo em CROMOL. Um **modelo** em CROMOL é composto por um conjunto de definições de locais e definições de classes. Uma **definição de classe** descreve um conjunto de atributos e operações. Um **objeto** representa uma instância de uma classe. Assim, quando instanciamos um objeto, devemos indicar qual classe define seus atributos (variáveis de instância) e as operações que podem ser realizadas sobre esses atributos. As variáveis de um objeto caracterizam o seu estado em um determinado momento, enquanto que as operações caracterizam o seu comportamento. Uma **definição de local** descreve um conjunto de atributos e uma única operação, chamada *main*, que é executada quando uma instância daquela definição de local é criada. Uma instância de uma definição local é chamada de local e pode ser vista como uma estação de rede ou, fazendo uma analogia direta a Java, como uma Máquina Virtual Java (JVM). Cada objeto pertence, necessariamente, a um determinado local.

O modelo Cliente x Servidor é formado por três definições de classes e duas de locais. As classes são: *Cliente*, *SolicitadorPagina* e *Servidor*, conforme pode ser visto na Figura 3.1. O comportamento modelado por essas classes é ilustrado na Figura 3.2 e descrito a seguir. Um *Cliente* funciona como um *browser*, solicitando páginas *web* a um *Servidor*. Como primeiro passo para solicitar uma página, um *Cliente* cria um *SolicitadorPagina*. Em seguida, o *SolicitadorPagina* é utilizado para iniciar uma nova *thread* que será responsável por requisitar a página desejada a um *Servidor* e aguardar o seu recebimento. Caso a página seja recebida, o *Cliente* é notificado pelo *SolicitadorPagina*. Um cliente pode cancelar a última solicitação de página realizada.

As definições de locais existentes no modelo Cliente x Servidor são: *MaquinaServidor* e *MaquinaCliente*. O método *main* de um local *MaquinaServidor* é responsável por criar um *Servidor* e torná-lo disponível para que possa ser acessado a partir de outros locais do modelo. Enquanto que o método *main* de um local *MaquinaCliente* tem como função criar um *Cliente* e enviar comandos para este cliente. Um objeto pertence ao local no qual ele foi criado. Durante a execução do modelo Cliente x

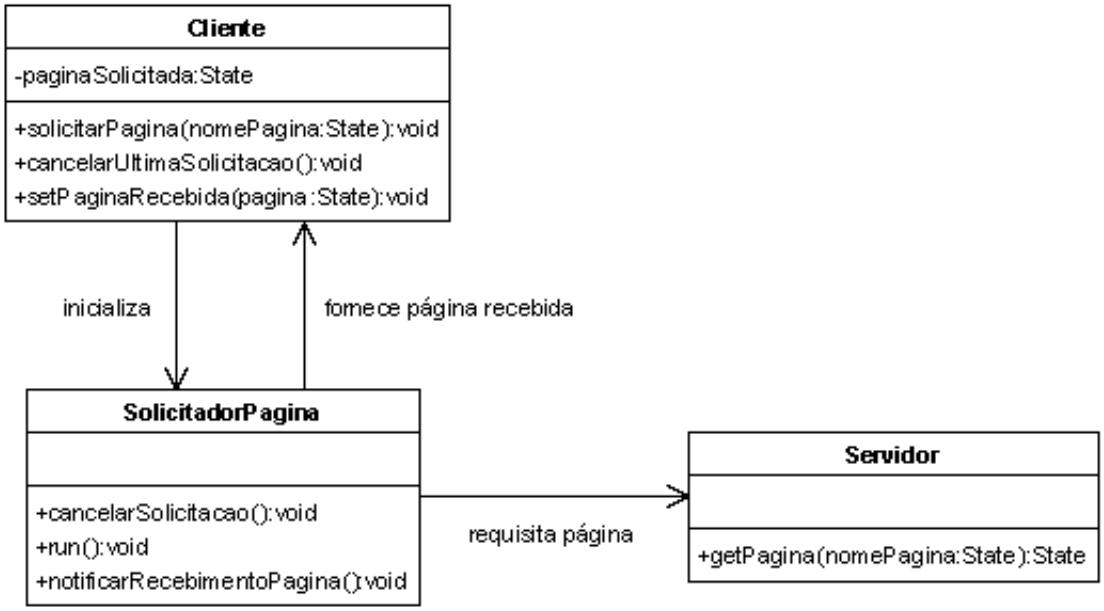

Figura 3.1: Diagrama de classes do modelo Cliente x Servidor.

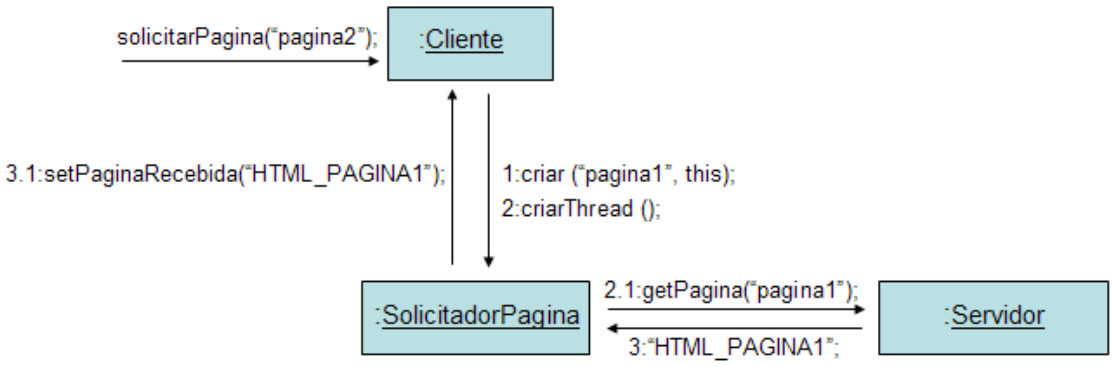

Figura 3.2: Diagrama de colaboração do modelo Cliente x Servidor.

Servidor, objetos *Cliente* e *SolicitadorPagina* apenas são criados a partir de um local *MaquinaCliente*, logo, eles sempre pertencem a um local deste tipo. O mesmo ocorre para objetos *Servidor* e locais do tipo *MaquinaServidor*. A Figura 3.3 ilustra a localização de alguns objetos que foram criados durante a execução do modelo Cliente x Servidor.

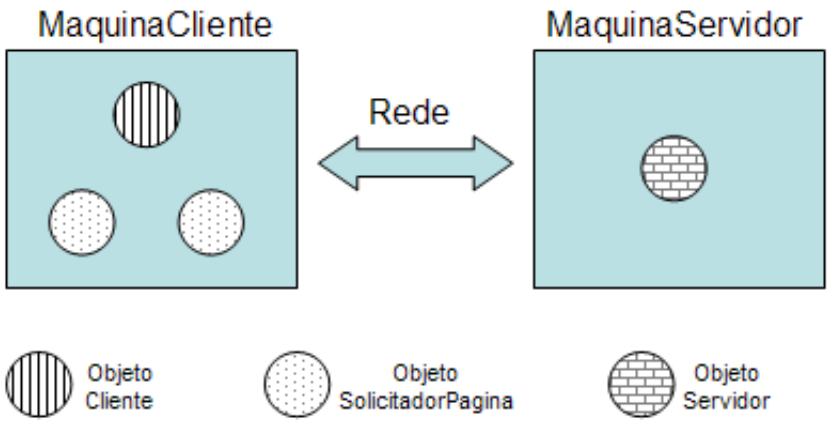

Figura 3.3: Localização de objetos criados durante a execução do modelo Cliente x Servidor.

Agora explicaremos cada uma das definições de classes e locais do modelo, iniciando pela classe *Servidor* que é definida a seguir:

```
class Servidor{
    State getPagina(State nomePagina){
        [nomePagina == "pagina1"]
            return "HTML_PAGINA1";
        [nomePagina == "pagina2"]
            return "HTML_PAGINA2";
        return "PAGINA_ERRO";
    }
}
```

Esta classe pode ser utilizada para criar objetos do tipo *Servidor*. Ela define uma única operação, cuja a assinatura é:

```
State getPagina(State nomePagina)
```

Conforme podemos observar em sua assinatura, esta operação é chamada *getPagina*, retorna um valor do tipo *State* e recebe como parâmetro a variável *nomePagina*, também do tipo *State*. Em CROMOL existem apenas dois **tipos de variáveis**. O primeiro tipo é o das **variáveis de estado**, cujo o valor é representado por uma cadeia de caracteres, simbolizando um estado. O segundo tipo é o das **variáveis de referência**. Uma variável de referência armazena uma ligação para um objeto, esta ligação é denominada referência. Quando um objeto **A** possui uma referência para um objeto **B**, dizemos que **A** referencia **B**. Durante a criação de um objeto todas as suas variáveis são incializadas para valores nulos.

Como podemos ver na operação *getPagina*, uma operação engloba um conjunto de instruções pré-definidas que abstraem uma função. Uma operação pode ser um método ou um construtor. No caso da operação *getPagina* trata-se de um método. A principal diferença, conforme é especificado em Java, é que os construtores são executados durante a criação de um objeto, enquanto que os métodos são utilizados como mecanismo de comunicação entre objetos. Esta comunicação é possível através do uso de uma variável de referência.

O conjunto de instruções do método *getPagina* tem como função testar se a página solicitada é uma daquelas que o objeto *Servidor* hospeda. Em caso afirmativo a página hospedada é retornada, em caso contrário uma página de erro é retornada. Para cada um dos testes a serem realizados utilizamos uma guarda, conforme pode ser visto na instrução:

```
[nomePagina == "pagina1"]
```

Nessa instrução, se a variável *nomePagina* tiver valor igual a "pagina1"a execução prossegue na instrução seguinte, caso contrário a instrução seguinte não é executada e a execução prossegue na instrução subseqüente a ela. Quando uma instrução de retorno é encontrada a execução do método é terminada e possivelmente um valor é retornado.

Entendida a classe *Servidor*, precisamos agora instanciar um servidor e prepará-lo para atender solicitações de páginas. Inicialmente, temos que definir o local onde esse servidor executará.

A execução de um modelo inicia com a criação de um local. Durante a execução,

podemos criar novos locais, cada local criado corresponde a um novo fluxo de execução. Esse novo fluxo de execução inicia executando a primeira instrução do método *main* do local criado. Observe que a execução em um local ocorre paralelamente à execução em qualquer outro local.

Objetos que pertencem a um local **A** podem se comunicar com qualquer outro objeto, independente do local ao qual ele pertença. Os objetos de **A** são chamados objetos locais em relação a **A**. Por outro lado, os objetos pertencentes a um local **B** qualquer são chamados objetos remotos em relação a **A**.

No modelo Cliente x Servidor o local onde um objeto *Servidor* executa é especificado pela definição de local *MaquinaServidor*. O método *main* dessa definição instanciará e disponibilizará um servidor para que ele possa ser acessado como um serviço remoto. A definição de local *MaquinaServidor* é mostrada a seguir:

```
site MaquinaServidor {
    void main(){
        mkrecorder <"registradorServicos">;
        Servidor servidor;
        servidor = new Servidor();
        export <servidor>;
        publish <"registradorServicos", "servidor", servidor>;
        return;
    }
}
```

O método *main* inicia criando um registrador de serviços. Um **registrador de serviços** possui duas funções. A primeira função é publicar objetos, ou seja, tornar um objeto acessível a partir de qualquer local existente na execução de um modelo. A segunda função nos permite obter uma referência para um objeto anteriormente publicado. Um objeto apenas pode ser publicado se ele for um serviço, conforme veremos logo mais. A instrução responsável por criar o registrador de serviços do modelo Cliente x Servidor é:

```
mkrecorder <"registradorServicos">;
```

O endereço dado ao registrador de serviços criado foi "registradorServicos". Este endereço será utilizado posteriormente quando precisarmos acessar este registrador, em busca de algum serviço publicado.

É importante compreender que cada instrução executada muda o estado da execução de um modelo. Ao estado atual da execução de um modelo em um determinado instante damos o nome de **configuração**. Ela inclui objetos, variáveis desses objetos, locais, registradores de serviços e os elementos necessários para controlar a execução de um modelo. Esses elementos serão vistos na Seção 3.3.

Após criar o registrador de serviços, o método *main* declara uma variável de referência para objetos do tipo *Servidor*, cria um *Servidor* e atribui a essa variável.

Agora precisamos transformar o *servidor* em um serviço remoto para que ele possa ser publicado. Um **serviço** é um objeto que disponibiliza um conjunto de funcionalidades a outros elementos de uma configuração. Essas funcionalidades são implementadas através de métodos do serviço. O funcionamento de um serviço consiste, basicamente, em atender requisições, processá-las e retornar os resultados desse processamento. Um **serviço remoto** é um serviço capaz de atender a requisições oriundas de um local diferente daquele que ele se encontra. Para transformar o *servidor* em um serviço remoto temos que exportá-lo através da instrução:

```
export <servidor>;
```

Após transformar o objeto em um serviço remoto, precisamos publicá-lo para que os demais objetos possam encontrá-lo e referenciá-lo remotamente. Um serviço deve ser publicado em um registrador de serviços, para publicar nosso serviço utilizaremos o registrador de serviços criado anteriormente. Na instrução de publicação devemos informar o endereço do registrador onde o serviço deve ser publicado, o nome do serviço e o objeto que representa o serviço, conforme podemos ver na instrução a seguir:

```
publish <"registradorServicos", "servidor", servidor>;
```

O nome dado ao serviço foi "servidor". Este nome será utilizado posteriormente para acessar esse serviço. Resumindo, o processo de disponibilização de um serviço remoto é composto por três passos:

- **Criação do objeto que implementa o serviço** - Ocorre da mesma forma que a criação de objetos comuns.

- **Exportação do objeto** - Depois de criado, o objeto que implementa o serviço deve ser exportado. A exportação consiste em transformar o objeto em um serviço remoto. Assim, após exportado, um objeto estará apto a atender e processar requisições às funcionalidades que ele oferece, sejam elas oriundas do local que ele se encontra ou de um outro local.

- **Publicação do serviço** - Como último passo, o serviço deve ser publicado em um registrador de serviços. Apenas assim, ele poderá ser acessado a partir de um outro local da configuração.

Após realizar o seu trabalho, o método *main* do local *MaquinaServidor* retorna com a instrução:

```
return;
```

Terminada a parte do servidor, explicaremos agora a parte do cliente, começando pela classe *SolicitadorPagina* que é definida a seguir:

```
class SolicitadorPagina{
    State nomePagina;
    State statusSolicitacao;
    Cliente clienteWeb;

    SolicitadorPagina(State nomePag, Cliente cliente){
        clienteWeb = cliente;
        nomePagina = nomePag;
        statusSolicitacao = "ATIVA";
        return;
    }
    void run(){
        Servidor servidor;
```

```
        State pagina;
        reference <"registradorServicos", "servidor", servidor>;
        pagina = servidor.getPagina(nomePagina);
        this.notificarRecebimentoPagina(pagina);
        return;
    }
    synchronized void notificarRecebimentoPagina(State pagina){
        [statusSolicitacao != "CANCELADA"]
            clienteWeb.setPaginaSolicitada(pagina);
        return;
    }
    synchronized void cancelarSolicitacao(){
        statusSolicitacao = "CANCELADA";
        return;
    }
}
```

Um *SolicitadorPagina* é responsável por solicitar uma página a um *Servidor* remoto. A classe define três variáveis. A primeira, chamada *nomePagina*, armazena o nome da página que está sendo solicitada. A segunda, denominada por *statusSolicitacao*, indica o *status* da solicitação em andamento. Por último temos a variável *clienteWeb* que referencia o cliente que iniciou a solicitação.

O construtor de um *SolicitadorPagina* recebe como parâmetro o nome de uma página a ser solicitada e o cliente que a requisitou. Podemos observar isso na assinatura do construtor:

```
SolicitadorPagina(State nomePag, Cliente cliente)
```

O conjunto de instruções do construtor de um *SolicitadorPagina* apenas atribui os parâmetros recebidos às devidas variáveis do objeto e inicializa a variável *statusSolicitacao* com o valor "ATIVA". Construtores não retornam valor.

O método *run* é responsável por adquirir de um servidor a página cujo nome está armazenado na variável *nomePagina*. Devido à existência desse método duas conside-

rações aqui são importantes. A primeira é o conceito de *thread*. Uma ***thread*** é uma parte do modelo que é executada de forma seqüencial, mas concorrentemente a outras *threads*. Cada *thread* está associada a um fluxo de execução do modelo, assim, sempre que uma thread é criada temos um novo fluxo de execução. A segunda consideração é que CROMOL possui dois **tipos de objetos**:

- **Objetos comuns** - Todos os objetos que não possuem um método chamado *run*.
- **Objetos executáveis** - Representam os objetos que possuem um método chamado run. Objetos executáveis são utilizados quando desejamos criar uma *thread*. Assim, em uma instrução de criação de *thread* devemos informar qual objeto executável está sendo utilizado. O novo fluxo de execução decorrente da criação da *thread* inicia no método *run* do objeto executável informado na criação da *thread*.

Como possuem o método *run*, objetos criados a partir da classe *SolicitadorPagina* são objetos executáveis. Portanto, podem ser utilizados em instruções de criação de *threads*.

As duas primeiras instruções do método *run* da classe *SolicitadorPagina* declaram duas variáveis. A variável *servidor* será usada para referenciar o servidor ao qual o *SolicitadorPagina* deve solicitar a página procurada. A variável *pagina* armazena a página retornada pelo servidor. A terceira instrução da operação *run* é:

```
reference <"registradorServicos", "servidor", servidor>;
```

Esta instrução é responsável por adquirir uma referência para o servidor que foi publicado anteriormente como um serviço remoto no registrador de serviços identificado por “registradorServicos”. O nome que identifica o serviço remoto é “servidor”. A variável *servidor* é utilizada para armazenar a referência adquirida. Neste ponto, precisamos definir os **tipos de referências** existentes em CROMOL. Existem dois tipos de referências: **local** e **remota**. O tipo de uma referência está relacionado a forma como obtemos tal referência. Referências podem ser obtidas de cinco formas:

- **Criação de objetos** - A criação de um objeto resulta em uma referência local para o objeto criado.

- **Consulta a um registrador de serviços** - Quando uma referência para um serviço específico é solicitada a um registrador de serviços, ele retorna uma referência remota para tal serviço.

- **Passagem de parâmetros** - Os parâmetros passados para um método podem ser referências locais ou remotas para um objeto qualquer. As referências que são recebidas pelo método invocado mantêm o mesmo tipo das referências passadas.

- **Retorno de métodos** - Referências retornadas por métodos também podem ser locais ou remotas. A referência recebida como retorno mantém o mesmo tipo da referência retornada.

- **Atribuição** - Após a execução de uma atribuição, a referência do lado esquerdo recebe o mesmo tipo da referência do lado direito da atribuição.

A Tabela 3.1 mostra, resumidamente, as formas de se obter uma referência e o tipo da referência obtida.

Formas de obter uma referência	**Tipo da referência**
Criação de objeto	Local
Consulta a um registrador de serviços	Remota
Passagem de parâmetros	Tipo mantido
Retorno de métodos	Tipo mantido
Atribuição	Tipo mantido

Tabela 3.1: Formas de obter uma referência e o tipo da referência obtida

Voltando ao modelo Cliente x Servidor, a referência armazenada na variável *servidor*, como resultado da execução da instrução **reference**, trata-se de uma referência remota, pois ele foi adquirida a partir de um registrador de serviços.

A próxima instrução da operação *run* requisita uma página a um servidor remoto. Isto é realizado através da invocação do método *getPagina* utilizando a referência remota recém-adquirida. Conforme podemos ver a seguir:

```
pagina = servidor.getPagina(nomePagina);
```

A **invocação de um método** tem por objetivo a execução do conjunto de instruções que compõem o corpo do método invocado. A invocação de um método depende do tipo de referência que se tem para o objeto que possui o método sendo invocado.

A invocação de um método utilizando-se uma referência local é ilustrada na Figura 3.4. Neste caso, o fluxo de execução corrente é desviado do ponto que invoca o método para o corpo do método invocado (1). Após o término da execução do método, o fluxo de execução retorna ao ponto a ser executado imediatamente após a chamada do método em questão (2). Com relação aos parâmetros passados temos dois casos. Primeiro, quando os parâmetros são representados por variáveis ou valores do tipo estado, passamos para o método invocado cópias desses parâmetros. Segundo, quando os parâmetros são representados por variáveis do tipo referência, passamos cópias dessas referências. Observe que um cópia de uma referência possui o mesmo valor que a referência original, assim, elas referenciam o mesmo objeto. Caso o método retorne algum valor, as mesmas regras citadas para os parâmetros são válidas para o valor retornado.

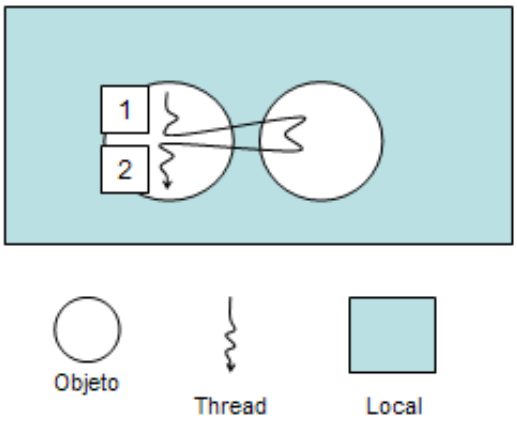

Figura 3.4: Invocação de método usando uma referência local.

A invocação de um método utilizando-se uma referência remota é ilustrada na Figura 3.5. A invocação inicia com o bloqueio do fluxo de execução corrente (1). Em seguida, uma mensagem é enviada (2) para o serviço remoto que possui o método a ser executado. A mensagem enviada trafega através de um meio não confiável, podendo ser perdida. Caso a mensagem chegue ao seu destino, uma *thread* é criada (3) no local do serviço remoto. A thread tem como finalidade executar o método que implementa

o serviço solicitado. Após a execução, o retorno do método é enviado através de uma mensagem (4) para o objeto solicitante do serviço. Novamente, a mensagem trafega através de um meio não confiável. Se a mensagem de retorno chegar ao seu destino, o fluxo de execução anteriormente bloqueado é desbloqueado (5) e a execução continua. Parâmetros do tipo estado se comportam conforme explicado para invocação usando uma referência local. Entretanto, quando os parâmetros são representados por variáveis do tipo referência temos uma modificação. Neste caso, se a referência for local passamos uma referência que aponta para uma cópia do objeto que a referência original apontava. Por outro lado, se a referência for remota passamos uma cópia dessa referência, ou seja, uma nova referência remota que aponta para o mesmo objeto que a referência original. O mesmo ocorre durante o retorno de um método, caso ele retorne algum valor.

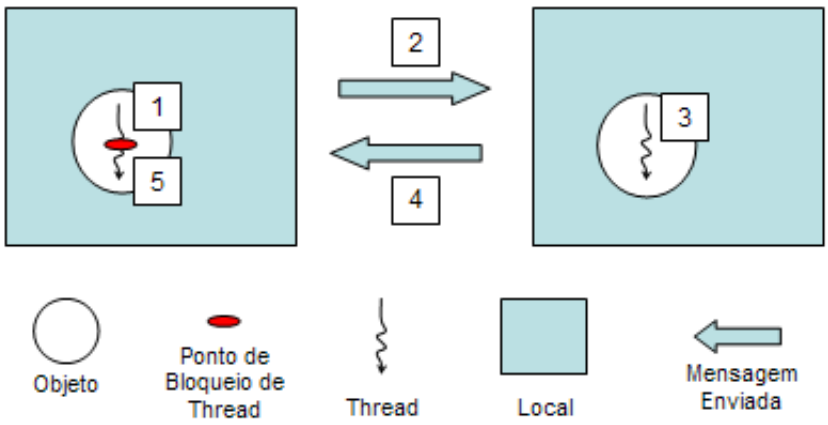

Figura 3.5: Invocação de método usando uma referência remota.

Como a referência utilizada para invocar o método *getPagina* é remota, temos uma invocação de método remoto. Assim, o parâmetro que representa a página desejada é passado como uma cópia do valor original. Da mesma forma, a página retornada e armazenada na variável *pagina* é uma cópia do valor original.

Quando uma página é retornada pelo servidor remoto temos que notificar o cliente através da invocação de método local vista a seguir:

```
this.notificarRecebimentoPagina(pagina);
```

A palavra reservada *this* representa uma referência para o objeto atualmente sendo executado. A última instrução do método *run*, indica que ele deve retornar. Quando

esta instrução é executada, temos o fim da execução da thread criada a partir deste objeto executável.

Os dois últimos métodos da classe *SolicitadorPagina* são:

```
synchronized void notificarRecebimentoPagina(State pagina)
synchronized void cancelarSolicitacao()
```

O método *notificarRecebimentoPagina* verifica se a solicitação atual foi cancelada, caso ela não tenha sido o cliente é comunicado que a página foi recebida. O método *cancelarSolicitacao* cancela a solicitação atual. Observe o uso da palavra reservada *synchronized*. Ele indica que apenas uma *thread* em um dado momento pode entrar em um desses métodos. Assim, não poderemos ter duas *threads* executando os dois métodos simultaneamente, também não poderemos ter duas *threads* executando um destes métodos simultaneamente. Isto evita o problema de um cliente cancelar uma solicitação e depois receber a página referente a essa solicitação.

Agora explicaremos a classe *Cliente* que pode ser vista a seguir:

```
class Cliente{
    State paginaSolicitada;
    SolicitadorPagina solicitador;

    void solicitarPagina(State nomePagina){
        solicitador =  new SolicitadorPagina(nomePagina, this);
        start <solicitador>;
        return;
    }
    void cancelarUltimaSolicitacao(){
        solicitador.cancelarSolicitacao();
        return;
    }
    void setPaginaRecebida(State pagina){
        paginaSolicitada = pagina;
        return;
```

```
    }
}
```

O atributo *solicitador* referencia o objeto criado para efetuar a última solicitação de página. O atributo *paginaSolicitada* armazenará a última página recebida como resposta de uma solicitação.

O método *solicitarPagina* inicia criando um *SolicitadorPagina* e o armazenando na variável *solicitador*. Em seguida, cria uma nova *thread* utilizando o *SolicitadorPagina* recém-criado, como podemos ver na instrução:

```
start <solicitador>;
```

A instrução **start** apenas é válida se o objeto informado como parâmetro for um objeto executável. No caso da instrução acima, o objeto representado pela variável *solicitador* é executável, pois possui o método *run*, conforme vimos anteriormente.

O método *cancelarUltimaSolicitacao* cancela a última solicitação de página que foi efetuada. Para isto, ele invoca o método *cancelarSolicitacao* do último *SolicitadorPagina* criado.

O método *setPaginaRecebida* é invocado por um *SolicitadorPagina* quando este recebe de um *Servidor* uma página solicitada. Este método apenas armazena a página recebida através do parâmetro *pagina* na variável *paginaSolicitada*.

Compreendido o funcionamento da classe *Cliente*, apresentaremos agora o local que inicializa a execução do modelo no lado cliente. Este local é chamado *MaquinaCliente* e sua definição é apresentada a seguir:

```
site MaquinaCliente {
    void main(){
        Cliente cliente;
        cliente = new Cliente();
        cliente.solicitarPagina("pagina1");
        cliente.cancelarUltimaSolicitacao();
        cliente.solicitarPagina("pagina2");
        return;
```

```
    }
}
```

O método *main* do local *MaquinaCliente* inicia criando um cliente e requisitando que o cliente solicite uma página chamada "pagina1". Em seguida, ele cancela a solicitação feita anteriormente e requisita uma nova página, chamada "pagina2". Por último o método retorna, encerrando sua execução.

É importante ressaltar a similaridade comportamental entre modelo apresentado e a linguagem Java. CROMOL absorveu e formalizou parte da semântica definida na especificação de Java, como resultado, conforme pôde ser observado ao longo desta seção, temos uma certa proximidade entre as construções de Java e CROMOL. Da mesma forma, a invocação remota de métodos segue fielmente o modelo RMI.

3.2 Sintaxe de CROMOL

Na Seção 3.1 o leitor pôde aprender através de um exemplo a sintaxe de CROMOL. Nesta seção apresentamos a sintaxe de forma direta e concreta. Uma descrição ainda mais completa da sintaxe da linguagem pode ser vista no Apêndice A, através de sua gramática. A notação utilizada no apêndice é a meta linguagem do gerador de *parser* e analisador léxico *Java Compiler Compiler* (JavaCC) [Sreenivas e Sankar, 2005].

3.2.1 Palavras Reservadas

As palavras reservadas em CROMOL e o contexto no qual elas são utilizadas são mostrados na Tabela 3.2.

3.2.2 Operadores

Em CROMOL existem apenas dois operadores. Na Tabela 3.3 mostramos, para cada operador, o símbolo utilizado para representá-lo, o seu nome e a finalidade para a qual ele é utilizado.

Palavra Reservada	Contexto
site	Declaração de local
class	Declaração de classe
State	Declaração de variáveis de estado
main	Definição do método de um local
new	Criação de um objeto
return	Retorno de uma operação
this	Referência para o objeto que está sendo executado
void	Definição de métodos que não retornam valor
export	Exportação de um objeto
mkrecorder	Criação de um registrador de serviços
publish	Publicação de um serviço
reference	Obtenção de uma referência remota para um serviço
start	Criação de uma *thread*
synchronized	Definição de um método sincronizado

Tabela 3.2: Palavras reservadas em CROMOL e contexto no qual elas são utilizadas

Operadores

Símbolo	Nome	Finalidade
==	igual	Comparar se dois estados são iguais
!=	diferente	Comparar se dois estados são diferentes

Tabela 3.3: Operadores em CROMOL

3.2.3 Construções

Nesta seção utilizamos uma notação simples e intuitiva para apresentar as construções de CROMOL. O objetivo não é mostrar a gramática da linguagem, mas oferecer um guia de consulta rápida aos modeladores. As construções serão apresentadas utilizando uma abordagem *top-down*, ou seja, iniciaremos mostrando as definições de classes e locais e iremos decompondo essas construções até chegar nas instruções da linguagem.

Um modelo é formado por uma seqüência de definições de locais seguida por um seqüência de definições de classes. A estrutura de uma definição de local é mostrada a seguir:

```
site NOME_DO_LOCAL {
    DECLARACAO_DE_VARIAVEIS
```

```
    void main(){
        SEQUENCIA_DE_INSTRUCOES
    }
}
```

No modelo deve existir pelo menos uma definição de local. O item DECLARACAO_DE_VARIAVEIS consiste em uma seqüencia de instruções de declaração de variável, cuja sintaxe será vista posteriormente.

A seguir temos o arcabouço da definição de uma classe:

```
class NOME_DA_CLASSE{
    DECLARACAO_DE_VARIAVEIS
    CONSTRUTOR
    SEQUENCIA_DE_METODOS
}
```

O item CONSTRUTOR é opcional. Sua definição pode ser vista a seguir:

```
NOME_DO_CONSTRUTOR(DECLARACAO_PARAMETROS){
    SEQUENCIA_DE_INSTRUCOES
}
```

O item DECLARACAO_PARAMETROS é formado por um conjunto de pares separados por vírgula. Cada um desses pares indica o tipo e o nome de uma variável. Como podemos ver a seguir:

```
TIPO_DA_VARIAVEL NOME_DA_VARIAVEL
```

A seguir temos a definição de um método:

```
SYNCHRONIZED RETORNO_DO_METODO
NOME_DO_METODO(DECLARACAO_PARAMETROS){
    SEQUENCIA_DE_INSTRUCOES
}
```

O item SYNCHRONIZED é opcional. Ele define se um método é sincronizado ou não. Quando o método for sincronizado o valor deste item deve ser *synchronized*. O valor do item RETORNO_DO_METODO pode ser *void*, informando que o método não retorna nenhum valor, ou pode ser o tipo do valor retornado pelo método.

O conjunto de instruções disponíveis em CROMOL é mostrado na Tabela 3.4.

3.2.4 Condições Sensíveis ao Contexto

Além de obedecer à sintaxe definida na Seção 3.2.3, um modelo CROMOL para estar sintaticamente correto precisa satisfazer a um conjunto de condições sensíveis ao contexto. Na validação da sintaxe de uma construção com relação a uma condição sensível ao contexto, precisamos de informações adicionais que podem ser obtidas através da análise do modelo (contexto) em questão. Como essas condições são dependentes de contexto, elas não são expressas na gramática mostrada no Apêndice A, pois a gramática foi escrita utilizando uma linguagem livre de contexto. Condições deste tipo são checadas por um analisador sintático sensível ao contexto. A seguir enumeramos as condições sensíveis ao contexto existentes em CROMOL:

- Na invocação de uma operação, o número de parâmetros passados deve ser igual ao número de parâmetros definidos na operação. Além disso, os tipos dos parâmetros também devem ser iguais.
- A igualdade de tipos é exigida também nas instruções de atribuição e de retorno com valor. Na última, o tipo de valor retornado deve ser igual ao tipo de retorno informado na declaração do método.
- O nome de um construtor deve ser igual ao nome da classe a qual ele pertence.
- Uma operação deve ser finalizada com uma instrução de retorno.
- Qualquer tipo usado no modelo, com exceção do tipo *State*, precisa ter sido declarado, não necessariamente antes de ser utilizado. Por outro lado, qualquer variável usada, precisa **antes** ter sido declarada.
- Todas as definições de classes de um modelo devem ter nomes diferentes. O mesmo ocorre com os lugares do modelo e os métodos de uma classe.

Instrução	**Sintaxe**
Criar objeto	new NOME_DA_CLASSE(PARAMETROS);
Criar registrador de serviços	mkrecorder <NOME_DO_REGISTRADOR>;
Criar thread	start <VARIAVEL_DE_REFERENCIA>;
Invocar método	REFERENCIA.NOME_DO_METODO(PARAMETROS);
Retornar	return;
Retornar com valor	return VALOR_RETORNADO;
Alterar valor de variável	VARIAVEL = NOVO_VALOR;
Exportar objeto	export <VARIAVEL_DE_REFERENCIA>;
Publicar serviço	publish <NOME_DO_REGISTRADOR, NOME_DO_SERVICO, VARIAVEL_DE_REFERENCIA>;
Obter referência remota	reference <NOME_DO_REGISTRADOR, NOME_DO_SERVICO, VARIAVEL_DE_REFERENCIA>;
Declarar variável	TIPO_DA_VARIAVEL NOME_DA_VARIAVEL;
Testar guarda	[ESTADO OPERADOR ESTADO]

Tabela 3.4: Instruções disponíveis em CROMOL

- O endereço de um registrador de serviços e o nome de um serviço devem sempre ser representados por um valor do tipo *State*.
- O objeto utilizado na invocação de um método deve possuir o método invocado.
- Um serviço remoto apenas pode ser referenciado através de uma variável de referência.
- O objeto utilizado na criação de uma *thread* deve ser executável.
- Os valores utilizados em uma guarda devem ser do tipo *State*.

3.3 Semântica

O entendimento desta seção é imprescindível para que se possa compreender a formalização mostrada no Capítulo 4. Aqui apresentamos a semântica informal de cada uma das ações possíveis de serem realizadas durante a execução de um modelo CROMOL. Essas ações são representadas pelas doze instruções mostradas na Seção 3.2.3 juntamente com a ação responsável pela criação de um local.

Antes de apresentar a semântica de cada uma dessas ações, precisamos definir os elementos de uma configuração que são necessários para controlar a execução de um modelo. Esses elementos são definidos a seguir:

Pilha de Execução

Uma pilha de execução representa um fluxo de execução do modelo. Quando precisamos de um novo fluxo de execução, seja devido a criação de um local ou de uma *thread*, criamos uma pilha de execução para representar este novo fluxo. Assim, durante a execução de um modelo temos um conjunto de pilhas de execução. Cada pilha está associada a uma das instruções que podem ser executadas em um determinado momento. Uma pilha também informa qual a próxima instrução que devemos executar quando retornamos da invocação de uma operação. Essas informações são armazenadas nos elementos da pilhas.

Elemento de uma Pilha de Execução

Podemos empilhar e desempilhar elementos em uma pilha de execução. O elemento do topo de uma pilha indica a localização do fluxo de execução representado pela pilha a qual ele pertence. Para isto o elemento armazena a **operação** sendo executada, o **local ou objeto** proprietário daquela operação, e o **índice da próxima instrução** a ser executada dentro daquela operação. Por exemplo, suponhamos que nós iniciássemos a execução de um modelo com a criação de um local que fosse identificado por *l1*, isto implicaria na criação de uma pilha com um elemento empilhado. Este elemento indicaria que a próxima instrução a ser executada por aquele fluxo é a **primeira** instrução da operação *main* do local *l1*, conforme é ilustrado na Figura 3.6. Suponhamos agora que esta primeira instrução seja executada e que sua execução não implique na invocação de uma operação, neste caso, temos a situação ilustrada na Figura 3.7. Se a segunda instrução for executada e esta implique na invocação de uma operação, por exemplo, o método m do objeto a, um novo elemento é empilhado, conforme ilustra a Figura 3.8. Quando retornamos de um método, o elemento do topo da pilha é desempilhado e o índice que aponta para a próxima instrução é incrementado. Assim, após o retorno do método m do objeto a teremos a pilha mostrada na Figura 3.9.

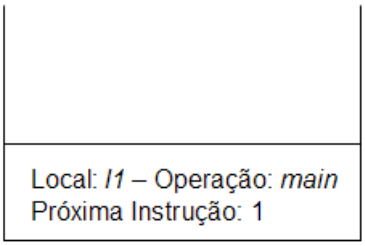

Figura 3.6: Pilha de execução após a criação do local *l1*.

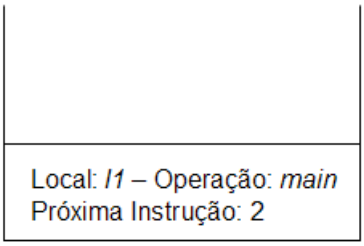

Figura 3.7: Pilha de execução após a execução da primeira instrução.

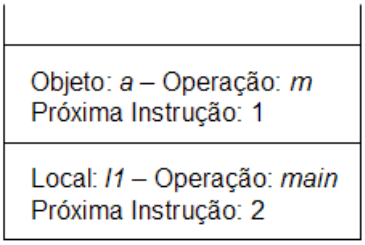

Figura 3.8: Pilha de execução após a execução da segunda instrução.

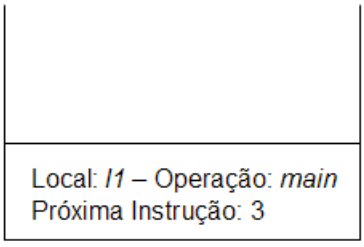

Figura 3.9: Pilha de execução após o retorno do método *m* do objeto *a*.

***Lock* de um objeto**

Todo objeto possui um *lock* associado a ele. Quando uma *thread* invoca um método sincronizado de um determinado objeto, ela deve adquirir o *lock* daquele objeto, antes de proceder na execução do método. Caso o *lock* seja conseguido, a execução do método procede normalmente. Caso contrário, a *thread* que invocou o método é bloqueada e passa aguardar que o *lock* desejado seja liberado. Sempre que uma *thread* termina a execução de um método sincronizado, ela libera o *lock* do objeto proprietário do método e desbloqueia todas as *threads* que aguardavam por esse *lock*.

Entendidos os elementos responsáveis pelo controle da execução de um modelo, concentraremos agora nossos esforços em definir a semântica das treze ações de execução de um modelo CROMOL.

A execução de uma ação implica em uma mudança no estado atual da execução de um modelo, ou seja, uma mudança na configuração atual. Desta forma, a semântica de uma ação pode ser definida através das modificações que a sua execução provoca em uma determinada configuração. Seguindo essa abordagem, definimos a seguir, a semântica de cada uma de nossas ações de execução.

Criar objeto

- Parâmetros:
  - *Variável de referência*
  - *Classe*
  - *Parâmetros do construtor*
- Modificações na Configuração:
  - Um objeto é criado a partir da *classe* informada. Este novo objeto pertencerá ao mesmo local onde se encontra o objeto que executou a ação.
  - As variáveis desse objeto são inicializadas. Variáveis de estado são inicializadas com o estado vazio, representado por “ ”. Variáveis de referência são

inicializadas com o objeto *null*, que pode ser considerado uma instância de todas as classes.

- A *variável de referência* informada passa a armazenar uma referência local para o objeto criado.
- Um novo elemento é empilhado na pilha responsável pela execução dessa instrução. Este novo elemento denota que a próxima instrução a ser executada por esse fluxo de execução será a primeira instrução do construtor do objeto criado.
- Para cada um dos *parâmetros do construtor* é criada uma variável local. Estas variáveis são incializadas com cópias dos valores recebidos como parâmetros. Lembre-se que uma cópia de uma referência aponta para o mesmo objeto que a referência original.

Criar registrador de serviços

- Parâmetros:
  - *Endereço do registrador*
- Modificações na Configuração:
  - Um novo registrador de serviços é criado.
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Criar thread

- Parâmetros:
  - *Objeto executável*
- Modificações na Configuração:
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

  - Uma nova pilha de execução é criada.
  - Um novo elemento é empilhado na pilha recém-criada. Este novo elemento denota que a próxima instrução a ser executada pelo novo fluxo de execução será a primeira instrução do método *run* do *objeto executável* utilizado na criação da *thread*.

Alterar valor de variável

- Parâmetros:
  - *Variável*
  - *Valor*
- Modificações na Configuração:
  - A *variável* informada passa armazenar uma cópia do *valor* fornecido.
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Exportar objeto

- Parâmetros:
  - *Objeto*
- Modificações na Configuração:
  - O *objeto* informado passa a ser um serviço remoto.
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Publicar serviço

- Parâmetros:
  - *Endereço de um registrador de serviços*

  - *Nome do serviço*
  - *Serviço remoto*

- Modificações na Configuração:
  - O registrador de serviços com o *endereço* fornecido passa a ter uma entrada em sua tabela de serviços que mapeia o *nome de serviço* informado para *serviço remoto* correspondente.
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Obter referência remota

- Parâmetros:
  - *Endereço de um registrador de serviços*
  - *Nome do serviço*
  - *Variável de referência*

- Modificações na Configuração:
  - A *variável de referência* informada passa a armazenar uma referência remota para o serviço solicitado.
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Declarar variável

- Parâmetros:
  - *Tipo da Variável*
  - *Nome da Variável*

- Modificações na Configuração:

  - Uma variável é criada no escopo atual, de acordo com o *tipo* e o *nome* informados.
  - A nova variável é inicializada.
  - O índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Testar guarda

- Parâmetros:
  - *Primeiro estado*
  - *Operador*
  - *Segundo estado*
- Modificações na Configuração:
  - Se o resultado do teste for verdadeiro o índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade. Caso o resultado seja falso esse índice é incrementado em duas unidades.

Invocar método

- Parâmetros:
  - *Objeto proprietário do método*
  - *Método*
  - *Parâmetros do método*
  - *Variável para armazenar retorno*
- Modificações na Configuração:
  - Se o *método* não é sincronizado, a execução do *método* é iniciada normalmente. Caso o *método* seja sincronizado e o *lock* do *objeto proprietário do método* encontre-se livre, o *lock* é adquirido e a execução do *método* também

é iniciada normalmente. A execução não será iniciada quando o *método* for sincronizado e o *lock* do *objeto proprietário do método* estiver em uso, neste caso, a pilha de execução responsável pela invocação do método é bloqueada e passa aguardar o evento de liberação daquele *lock*.

– Nos casos em que ocorre o início da execução do *método* temos dois aspectos a considerar. Se o *objeto proprietário do método* é referenciado através de uma referência local (invocação local), um novo elemento é empilhado na pilha de execução. Este novo elemento denota que a próxima instrução a ser executada naquele fluxo é a primeira instrução do *método* invocado. Para cada um dos *parâmetros do método* é criada uma variável local. Estas variáveis são inicializadas com cópias dos valores recebidos como parâmetros. Por outro lado, se o *objeto proprietário do método* é referenciado através de uma referência remota, ou seja, estamos utilizando uma referência remota para acessar um serviço (invocação remota), a pilha de execução responsável pela invocação do *método* é bloqueada e, caso não ocorra perca de mensagens, uma outra pilha de execução é criada no local onde se encontra o serviço solicitado. Após a criação desta pilha, um elemento também é criado e empilhado nela. Este novo elemento denota que a próxima instrução a ser executada é a primeira instrução do *método* invocado remotamente. Com objetivo de termos como saber para onde retornar ao término da execução do *método* remoto, armazenamos um *bind* entre a pilha responsável pela invocação do *método* e pilha criada para executar o método. Para cada um dos *parâmetros do método* é criada uma variável local. Quando o parâmetro recebido é do tipo estado a variável correspondente é inicializada com uma cópia desse estado. No caso do parâmetro ser uma referência local, a variável correspondente é inicializada com uma referência que aponta para uma cópia do objeto que a referência recebida apontava. Por outro lado, se o parâmetro for uma referência remota, a variável correspondente é inicializada com uma cópia dessa referência, ou seja, uma nova referência remota que aponta para o mesmo objeto que a referência original.

Retornar

- Modificações na Configuração:
  - É desempilhado um elemento da pilha de execução responsável pela ação de retorno.
  - Se a pilha de execução ainda possuir pelo menos um elemento (retorno local), o índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade. Por outro lado, caso a pilha não possua mais elementos, ela é removida da configuração.
  - Quando uma pilha é removida e existe algum *bind* entre ela e outra pilha qualquer, temos um retorno remoto. Neste caso, o *bind* é eliminado e, se não ocorrer perca de mensagem, a pilha de execução obtida a partir do *bind* é desbloqueada e o índice que aponta para a próxima instrução a ser executada é incrementado em uma unidade.

Retornar com valor

- Parâmetros:
  - *Valor retornado*
- Modificações na Configuração:
  - Todas as modificações citadas na ação **Retornar** também são válidas aqui. Adicionalmente temos que o valor retornado é armazenado na variável destinada a receber tal valor. Em um retorno local, o valor retornado obedece as mesmas regras descritas para valores passados como parâmetros durante a invocação local de um método. Da mesma forma, para valores retornados remotamente são válidas as regras descritas para valores passados como parâmetros em uma invocação remota.

Criar Local

- Parâmetros:

  - *Definição de local*

- Modificações na Configuração:

  - Um local é criado a partir da *definição de local* informada.
  - Uma pilha de execução é criada. Nesta pilha empilhamos um novo elemento. Este novo elemento denota que a próxima instrução a ser executada pelo novo fluxo de execução, será a primeira instrução do método *main* do local criado.

Concluído o aprendizado da sintaxe e semântica informal de CROMOL, o leitor encontra-se preparado para simular o comportamento de sistemas distribuídos implementados em Java, através da criação e execução de modelos. É conveniente relembrar que a compreensão da semântica aqui apresentada é fundamental para que se possa dar início a leitura do Capítulo 4, que diz respeito a formalização de CROMOL.

Capítulo 4

Formalizando CROMOL

No Capítulo 3, apresentamos CROMOL com uma semântica informal. Neste capítulo formalizamos CROMOL, utilizando teoria matemática para descrever o seu significado. Nosso objetivo é especificar matematicamente como os modelos devem ser executados.

O principal benefício de uma formalização é que ela pode ser utilizada como um **guia formal** para a implementação de ferramentas. Exemplos de ferramentas são: simuladores, máquinas virtuais e verificadores de modelos. Um guia formal é não ambíguo, e resulta em uma implementação única, desde que seja obedecido aquilo especificado na formalização. Assim, apenas podemos afirmar, de um ponto de vista matemático, que duas ferramentas implementadas terão o mesmo comportamento, se ambas tiverem sido desenvolvidas em concordância com o mesmo modelo matemático. Neste caso, o comportamento ao qual nos referimos diz respeito apenas ao comportamento especificado na formalização da linguagem, em outras palavras, se alguma funcionalidade informal[1] for incorporada à linguagem, nada podemos afirmar em termos matemáticos com relação a esta funcionalidade. No Capítulo 5, apresentamos um simulador para a linguagem CROMOL implementado com base na formalização apresentada neste capítulo.

Para simplificar a formalização, realizamos duas abstrações na semântica da linguagem. A primeira é que um objeto não precisa ser exportado para se tornar um serviço remoto. A segunda é que todos os serviços são publicados em um único registrador de serviços. Essas abstrações facilitam o processo de leitura da formalização e

[1]Funcionalidades informais são aquelas que não foram especificadas em um formalismo

não resultam em danos ao nosso projeto.

A formalização está estruturada em três partes. Inicialmente, apresentamos a sintaxe formal da linguagem. Em seguida, definimos algumas funções auxiliares que tornarão mais simples o processo de formalização. Por último, mostramos a semântica operacional [Winskel, 1994] da linguagem. Outras linguagens que também foram formalizadas através da definição de uma semântica operacional são: POOL [Pierre America e Rutten, 1986] e CSP [Plotkin, 1983].

4.1 Conjuntos Sintáticos

Antes de apresentar a semântica operacional de CROMOL, precisamos descrever formalmente sua sintaxe. Faremos isto através da definição dos conjuntos sintáticos associados a nossa linguagem.

Os conjuntos sintáticos serão divididos em dois grupos, de acordo com a notação utilizada para defini-los. O primeiro grupo consiste dos seguintes conjuntos:

Obj – Conjunto de objetos. Este conjunto é definido com o auxílio do conjunto $\mathbb{N}$ dos números naturais:

$$Obj = \{ \ \hat{r} \ | \ r \in \mathbb{N} \ \}$$

Elementos representados por α e o.

ObjUnull – Conjunto formado pela união do conjunto de objetos ao conjunto contendo o elemento *null*, formalmente temos:

$$ObjUnull = Obj \ \cup \ \{null\}$$

O elemento *null* representa o valor nulo.

Elementos representados por β.

State – Conjunto de valores que uma variável do tipo **State** pode assumir, ou seja, é o conjunto formado por estados que podem ser alcançados por um variável deste tipo. Este conjunto é definido com o auxílio do conjunto $\mathbb{N}$ dos números naturais:

$$State = \{ \quad \check{e} \quad | \quad e \ \in \ \mathbb{N} \quad \}$$

O estado vazio, representado em CROMOL por " ", é aqui representado por $\check{0}$.

Elementos representados por ζ e ι.

$Site$ – Conjunto de locais. Este conjunto é definido com o auxílio do conjunto $\mathbb{N}$ dos números naturais:

$$Site = \{ \quad \tilde{t} \quad | \quad t \ \in \ \mathbb{N} \quad \}$$

Elementos representados por ψ e ρ.

$OVar$ – Conjunto de variáveis utilizadas para referenciar objetos.

Elementos representados por x e y.

$SVar$ – Conjunto de variáveis utilizadas para armazenar um estado.

Elementos representados por s.

Var – Conjunto de todas as variáveis. Formado pela união de $OVar$ a $SVar$:

$$Var = \ OVar \ \cup \ SVar$$

Elementos representados por v.

$CName$ – Conjunto de nomes de classes.

Elementos representados por C.

$CCName$ – Conjunto de nomes de construtores.

Elementos representados por a.

$CMName$ – Conjunto de nomes de métodos.

Elementos representados por m.

$SDName$ – Conjunto de nomes de definições de locais.

Elementos representados por K.

Conforme anunciado anteriormente, os conjuntos sintáticos do segundo grupo serão definidos através de uma notação diferente da empregada até o momento. A idéia é definir como os elementos desses conjuntos são construídos, utilizando para isso regras de produção. Para especificar as regras usamos a Backus Normal Form (BNF). Ainda com relação a notação, utilizamos parênteses - " (" e ") " - para delimitar uma seqüência; parênteses angulares - " $\langle$ " e " $\rangle$ " - para delimitar uma tupla; e $A \Leftarrow B$ para indicar que $(A, B) \in R$, sendo R uma relação.

Os conjuntos sintáticos do segundo grupo são:

StateRep – Conjunto de formas para representar um estado.

Elementos representados por *sr*.

sr	::=	*s*	(variável que armazena um estado)
	|	ζ	(estado)

ObjRep – Conjunto de formas para representar um objeto.

Elementos representados por *or*.

or	::=	*x*	(variável que referencia um objeto)
	|	**this**	(objeto proprietário da operação sendo executada)

ObjExp – Conjunto de expressões que retornam um objeto.

Elementos representados por *oe*.

oe	::=	*or*	(representação de um objeto)
	|	**new** $C(\ )$	(criação de um objeto)

RelOp – Conjunto de operadores relacionais. Nossos operadores relacionais são usados na comparação de estados.

Elementos representados por *ro*.

ro	::=	==	(igual)
	|	!=	(diferente)

Statement – Conjunto de instruções. Instruções podem estar presentes no corpo de um método ou construtor.

Elementos representados por *st*.

st	::=	$x = oe;$	(atribuição de um objeto a uma variável)
	|	$s = sr;$	(atribuição de um estado a uma variável)
	|	$or.m(\ );$	(invocação de um método)
	|	**publish** $<sr, or>;$	(publicação de um serviço)
	|	**start** $<or>;$	(criação de uma thread)
	|	$[sr \ \ ro \ \ sr]$	(condição para execução da instrução seguinte)
	|	**reference** $<sr, x>;$	(referenciando um serviço)
	|	**return**;	(retornando de uma operação)

$Model$ – Conjunto de modelos, com cada modelo sendo composto por definições de locais e classes.

Elementos representados por M.

$$M \quad ::= \quad \langle (C_1 \Leftarrow cd_1, \cdots, C_n \Leftarrow cd_n), (K_1 \Leftarrow sd_1, \cdots, K_l \Leftarrow sd_l) \rangle$$

onde:

- C_n representa o nome de uma classe e cd_n sua definição,
- K_l representa o nome de uma definição de local e sd_l a respectiva definição,
- $n \in \mathbb{N}$,
- $l \in \mathbb{N}$ e $l > 0$.

$ClassDef$ – Conjunto de definições de classes.

Elementos representados por cd.

$$cd \quad ::= \quad \langle a \Leftarrow oc, (v_1, \cdots, v_n), (m_1 \Leftarrow om_1, \cdots, m_l \Leftarrow om_l) \rangle$$

onde:

- a representa o nome do construtor da classe e oc a definição dele,
- $v_1, \cdots, v_n$ são variáveis de instância da classe,
- m_l representa o nome de um método e om_l sua definição,
- $n \in \mathbb{N}$,
- $l \in \mathbb{N}$.

$SiteDef$ – Conjunto de definições de locais.

Elementos representados por sd.

$$sd \quad ::= \quad \langle (v_1, \cdots, v_n), (main \Leftarrow om) \rangle$$

onde:

- $v_1, \cdots, v_n$ são variáveis de instância da definição de local,
- om representa a definição do método $main$,
- $n \in \mathbb{N}$.

$OpDef$ – Conjunto de definições de operações. Uma operação pode ser um método ou um construtor.

Elementos representados por op, om e oc.

$$op \quad ::= \quad (st_1, \cdots, st_n)$$

onde:

- $st_1, \cdots, st_n$ são as instruções do corpo da operação,
- $n \in \mathbb{N}$.

Finalizada as definições dos conjuntos sintáticos associados a nossa linguagem, é importante fazer uma última consideração. As condições sensíveis ao contexto, apresentadas informalmente na Seção 3.2.4, devem ser incorporadas à sintaxe formal aqui descrita, ou seja, elas devem ser satisfeitas para que possamos ter um modelo sintaticamente correto. Entretanto, visando à simplificação de nossa formalização, essas condições não serão formalizadas.

4.2 Funções Auxiliares

Nesta seção apresentamos algumas funções que serão utilizadas durante a definição da semântica operacional de CROMOL. Essas funções, denominadas funções auxiliares, têm por objetivo facilitar o processo de formalização da linguagem.

As funções auxiliares são assim definidas:

μ – Sendo M, C e m os seus parâmetros, a função μ seleciona a definição do método m que se encontra na classe chamada C do modelo M. Formalmente:

$\mu \ : \ Model \ \times \ CName \ \times \ CMName \ \rightarrow \ OpDef$

definida por

$\mu(M, C, m) = om$

onde:

- $M = \ \langle(\cdots, C \Leftarrow cd, \ \cdots), (K_1 \Leftarrow sd_1, \cdots, K_l \Leftarrow sd_l)\rangle$,
- $cd = \ \langle a \Leftarrow oc, (v_1, \cdots, v_n), (\cdots, m \Leftarrow om, \ \cdots)\rangle$,
- $om = \ (st_1, \cdots, st_p)$.

κ – Sendo M e K os seus parâmetros, a função κ seleciona a definição do método *main* que se encontra na definição de local chamada K do modelo M. Formalmente:

$\kappa \ : \ Model \ \times \ SDName \ \rightarrow \ OpDef$

definida por

$\kappa(M, K) = om$

onde:

- $M = \ \langle(C_1 \Leftarrow cd_1, \cdots, C_n \Leftarrow cd_n), (\cdots, K \Leftarrow sd, \ \cdots)\rangle$,
- $sd = \ \langle(v_1, \cdots, v_l), (main \Leftarrow om)\rangle$,
- $om = \ (st_1, \cdots, st_p)$.

η – Sendo M e C os seus parâmetros, a função η seleciona a definição do construtor da classe chamada C do modelo M. Formalmente:

$\eta \ : \ Model \ \times \ CName \ \rightarrow \ OpDef$

definida por

$\eta(M, C) = oc$

onde:

- $M = \langle(\cdots, C \Leftarrow cd, \ \cdots), (K_1 \Leftarrow sd_1, \cdots, K_l \Leftarrow sd_l)\rangle$,
- $cd = \langle a \Leftarrow oc, (v_1, \cdots, v_n), (m_1 \Leftarrow om_1, \cdots, m_p \Leftarrow om_p)\rangle$,
- $oc = (st_1, \cdots, st_q)$.

δ – Sendo *op* e n os seus parâmetros, a função δ seleciona a instrução de índice n da operação *op*. Formalmente:

$\delta : OpDef \times \mathbb{N} \rightarrow Statement$

definida por

$\delta(op, n) = st_n$

onde:

- $op = (st_1, \cdots, st_l)$,
- $n \in \mathbb{N}$,
- $l \in \mathbb{N}$,
- $n \leq l$.

4.3 Semântica Operacional

Iniciamos esta seção com a definição formal de uma configuração. Em seguida, apresentamos o nosso sistema de transições, mostrando as regras de transição que definem formalmente como a execução de um modelo evolui de uma configuração para outra. São essas regras que descrevem o comportamento de nossa linguagem e, portanto, sua semântica operacional.

4.3.1 Configuração

Conforme visto no Capítulo 3, uma configuração reflete o estado de execução de um modelo em um dado instante. Nesta subseção concretizamos esta idéia subjetiva, formalizando o conceito de configuração. Entretanto, antes de definir uma configuração precisamos de algumas definições preliminares:

Bool – Conjunto dos valores booleanos.

$$Bool = \{true, false\}$$

Elementos representados por b.

EEnt – Conjunto de entidades de execução. Uma entidade de execução é um componente de nosso modelo que possui uma ou mais operações. Isto possibilita que a entidade possa ser percorrida por um fluxo de execução.

$$EEnt = Obj \ \cup \ Site$$

Elementos representados por en.

ESElem – Conjunto de elementos de pilhas de execução. Formado pelos elementos que podem ser empilhados em uma pilha de execução. Esses elementos indicam, em um determinado momento, a localização do fluxo de execução representado pela pilha a qual ele pertence.

$$ESElem = EEnt \times OpDef \times Statement \times \mathbb{N}$$

Elementos representados por el e ee.

Um elemento corresponde à tupla:

$$\langle en, \ op, \ st, \ ip \rangle$$

onde:

- en é a entidade (objeto ou local) que está sendo percorrida pelo fluxo de execução,
- op é a operação de en que está sendo executada,
- st é a próxima instrução a ser executada,
- ip é o índice da próxima instrução a ser executada.

EStack – Conjunto de pilhas de execução. Uma pilha de execução representa um fluxo de execução.

$$EStack = Bool \times Site \times \text{seq } ESElem$$

onde:

- seq $ESElem$ representa uma seqüência do tipo $(el_1, \cdots, el_n)$.

Elementos representados por es e xs.

Um elemento corresponde a tupla:

$$\langle b,\ \psi,\ (el_1, \cdots, el_n)\rangle$$

onde:

- b é uma *flag* indicando se a pilha está bloqueada ou não.
Caso o seu valor seja $true$, o fluxo de execução representado pela pilha não poderá ter sua próxima instrução executada,
- ψ é o local ao qual pertence a pilha,
- $(el_1, \cdots, el_n)$ é uma seqüência que representa os elementos empilhados. O elemento de índice 1 é a base da pilha, o elemento de índice n é o topo da pilha.

Σ – Conjunto de tuplas. Cada tupla é formada por duas funções. A primeira função recebe como parâmetro um elemento do tipo $EEnt$ e retorna uma outra função que mapeia um elemento em $Ovar$ para um em $ObjUnull$. A segunda função também recebe como parâmetro um elemento do tipo $EEnt$ e retorna uma outra função que mapeia um elemento em $Svar$ para um em $State$. Formalmente:

$$\Sigma = (EEnt \rightarrow (OVar \rightarrow ObjUnull)) \times (EEnt \rightarrow (SVar \rightarrow State))$$

$Type$ – Conjunto de funções que mapeiam um elemento em Obj para um elemento em $CName$. Formalmente:

$$Type = Obj \rightarrow CName$$

Γ – Conjunto de funções que mapeiam um elemento em $State$ para um elemento em Obj. Formalmente:

$$\Gamma = State \rightarrow Obj$$

Ξ – Conjunto de funções que mapeiam um elemento em Obj para um elemento em $Site$. Formalmente:

$$\Xi = Obj \rightarrow Site$$

Θ – Conjunto de funções que mapeiam um elemento em Obj e um elemento em $EEnt$ para um elemento em $Bool$. Formalmente:

$$\Theta = Obj \times EEnt \rightarrow Bool$$

Ω – Conjunto de funções que mapeiam um elemento em $EStack$ para um outro elemento em $EStack$. Formalmente:

$$\Omega = EStack \rightarrow EStack$$

Após essas definições preliminares podemos definir formalmente o conjunto de configurações $Conf$ como sendo:

$$Conf = \mathcal{P}_{fin}(EStack) \times \Sigma \times \mathcal{P}_{fin}(Site) \times Type \times \Gamma \times \Xi \times \Theta \times \Omega \times Model$$

Uma configuração é um elemento do conjunto $Conf$, sendo representada pela tupla:

$$c = \langle X,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle$$

A seguir descrevemos cada componente da tupla.

X – Conjunto finito de pilhas de execução. Cada pilha indica a existência de um fluxo de execução paralelo na configuração.

$$X = \{es_1, \cdots, es_n\}$$

σ – Representa uma tupla contendo duas funções. As funções são utilizadas para se obter o valor de qualquer variável de um modelo em execução. Portanto, este componente é responsável por nos fornecer o estado de variáveis.

$$\sigma = \langle \sigma_1, \sigma_2 \rangle$$

onde:

- σ_1 é uma função que recebe um objeto α e retorna uma outra função. Esta outra função mapeia nomes de variáveis de α que referenciam objetos para seus respectivos valores,
- σ_2 é uma função que recebe um objeto α e retorna uma outra função. Esta outra função mapeia nomes de variáveis de α que armazenam estados para seus respectivos valores.

S – Conjunto de locais criados até um determinado instante durante a execução de um modelo.

τ – Função que mapeia um objeto para sua classe. Trata-se de uma função de tipo.

γ – Função que mapeia o nome de um serviço para o respectivo serviço previamente publicado.

ξ – Função que mapeia um objeto para o local a qual ele pertence.

θ – Função que mapeia um objeto e uma entidade de execução para um valor booleano. O valor *true* indica que o objeto é um serviço remoto para aquela entidade de execução.

ω – Função utilizada durante o retorno de uma invocação remota de um método. A função indica qual pilha de execução deve ser desbloqueada quando o retorno de uma execução remota for executado.

M – Representa o modelo que está sendo executado.

4.3.2 Sistema de Transição

No nosso sistema de transição, o disparo de uma transição corresponde a execução de uma das ações definidas na semântica de CROMOL[2]. Assim, durante a execução de um modelo, quando uma ação é executada o estado do sistema muda, ou seja, saímos de uma configuração para outra. Em decorrência disto, execução de um modelo

[2]Como conseqüência disto, utilizaremos os termos *disparo de uma transição* e *execução de uma ação* indistintamente

pode ser representada como uma seqüência de configurações com transições entre elas. Neste ponto, duas considerações são importantes. Primeiro, o conjunto das possíveis seqüências de execução de um modelo resulta no espaço de estados deste modelo. Segundo, a partir do seu espaço de estados podemos verificar um modelo aplicando técnicas formais de verificação de modelos [Clarke, 1999]. Estas duas considerações mostram como a formalização de uma linguagem facilita a utilização de técnicas formais de verificação de corretude.

Voltando ao nosso sistema de transição, o conjunto das possíveis transições é dado pela função de transição:

$$f : Conf \rightarrow Conf$$

Antes de definir nossa função de transição uma consideração é importante. Durante a execução de um modelo, os valores das variáveis envolvidas em uma instrução são recuperados antes que ela seja executada. A recuperação dos valores é realizada através da função σ de uma configuração. Assim, temos:

- A variável x da entidade de execução en é avaliada para o valor β

  onde:

  - $\sigma = \langle \sigma_1, \sigma_2 \rangle$,
  - $\sigma_1(en)(x) = \beta$.

- A variável s da entidade de execução en é avaliada para o valor ζ

  onde:

  - $\sigma = \langle \sigma_1, \sigma_2 \rangle$,
  - $\sigma_2(en)(s) = \zeta$.

Entretanto a avaliação da variável implícita **this** ocorre de forma diferente. Seja st a instrução onde o **this** ocorre e $\langle \alpha,\ op,\ st,\ ip \rangle$ o elemento onde ela ocorre, **this** será avaliada para α.

A seguir apresentamos as regras de transição (axiomas) que definem as possíveis transições de nosso sistema, e conseqüentemente, definem também a função de transição f e a semântica operacional de nossa linguagem.

Criando um objeto

$\langle X \cup \langle false, \psi, (el_1, \cdots, el_n)\rangle,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M\rangle \rightarrow$

$\langle X \cup \langle false, \psi, (el_1, \cdots, el_n, el_{n+1})\rangle,\ \sigma'',\ S,\ \tau',\ \gamma,\ \xi',\ \theta',\ \omega,\ M\rangle$

onde:

- $el_n = \langle en,\ op,\ st,\ ip\rangle$,
- st é a instrução $y =$ **new**$C(\ )$; ,
- $el_{n+1} = \langle \hat{r},\ \eta(M, C),\ \delta(\eta(M, C), 1),\ 1\rangle$,
- r é o primeiro número que não está sendo usado por um objeto e $\hat{r}$ representa o objeto recém-criado,
- $\tau' = \tau\{C/\hat{r}\}$,
- $\xi' = \xi\{\psi/\hat{r}\}$,
- $\sigma'' = \langle\sigma'_1\{\lambda x.null/\hat{r}\}, \sigma'_2\{\lambda s.\check{0}/\hat{r}\}\rangle$,
- $\sigma' = \langle\sigma_1\{(\sigma_1(en)\{\hat{r}/y\})/en\}, \sigma_2\rangle$,
- $\theta' = \theta\{false/(\hat{r}, en)\}$,
- $n \geq 1$.

O elemento el_{n+1} representa o novo topo da pilha, observe que este elemento indica que o fluxo de execução encontra-se na primeira instrução do construtor de $\hat{r}$. Na notação que nós utilizamos, $\tau\{C/\hat{r}\}$ é definido por:

$$\tau\{C/\hat{r}\} = \begin{cases} \tau(p) & \text{se } p \neq \hat{r}, \\ C & \text{se } p = \hat{r}. \end{cases}$$

A definição de $\xi\{\psi/\hat{r}\}$ ocorre de forma análoga. Na definição de σ'', $\lambda x.null$ representa a função que mapeia qualquer variável x para o valor $null$. Significado semelhante tem $\lambda s.\check{0}$. Observe que a definição de σ'' é dada em função de σ' que por sua vez é definida em função de σ. Nosso objetivo ao utilizar essa abordagem é facilitar o entendimento do leitor. A redefinição da função θ indica que todo objeto é criado localmente.

Mudando o valor de uma variável usada para referenciar um objeto

$\langle X \cup \langle false, \psi, (el_1, \cdots, el_n)\rangle,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M\rangle \rightarrow$

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el'_n)\rangle, \ \sigma', \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle$

onde:

- $el_n = \ \langle en, \ op, \ st, \ ip\rangle$,
- st é a instrução $y = \beta$; ,
- $el'_n = \ \langle en, \ op, \ \delta(op, ip + 1), \ ip + 1\rangle$,
- $\sigma' = \ \langle \sigma_1\{(\sigma_1(en)\{\beta/y\})/en\}, \sigma_2\rangle$,
- $n \geq 1$.

O novo topo da pilha, ou seja, o elemento el'_n, aponta para a próxima instrução a ser executada em *op*. A função de estado σ é redefinida, passando a retornar o valor β para a variável y da entidade de execução *en*.

Mudando o valor de uma variável usada para armazenar um estado

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el_n)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle \ \rightarrow$
$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el'_n)\rangle, \ \sigma', \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle$

onde:

- $el_n = \ \langle en, \ op, \ st, \ ip\rangle$,
- st é a instrução $s = \zeta$; ,
- $el'_n = \ \langle en, \ op, \ \delta(op, ip + 1), \ ip + 1\rangle$,
- $\sigma' = \ \langle \sigma_1, \sigma_2\{(\sigma_2(en)\{\zeta/s\})/en\}\rangle$,
- $n \geq 1$.

Testando uma guarda com o operador igual

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el_n)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle \ \rightarrow$
$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el'_n)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle$

onde:

- $el_n = \ \langle en, \ op, \ st, \ ip\rangle$,
- st é a instrução $[\ \zeta \ == \ \iota \ ]$,
- se $(\ \zeta = \ \iota \ )$ então $el'_n = \ \langle en, \ op, \ \delta(op, ip + 1), \ ip + 1\rangle$
  caso contrário $el'_n = \ \langle en, \ op, \ \delta(op, ip + 2), \ ip + 2\rangle$,
- $n \geq 1$.

Testando uma guarda com o operador diferente

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el_n)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle \ \rightarrow$

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el'_n)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle$

onde:

- $el_n = \ \langle en, \ op, \ st, \ ip\rangle$,
- st é a instrução $[\ \zeta \ != \ \iota \ ]$,
- se $(\ \zeta \neq \ \iota \ )$ então $el'_n = \ \langle en, \ op, \ \delta(op, ip+1), \ ip+1\rangle$
  caso contrário $el'_n = \ \langle en, \ op, \ \delta(op, ip+2), \ ip+2\rangle$,
- $n \geq 1$.

Criando um local definido por K

$\langle X, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle \ \rightarrow$

$\langle X \ \cup \ \langle false, \tilde{t}, (el)\rangle, \ \sigma', \ S', \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle$

onde:

- $el = \ \langle \tilde{t}, \ \kappa(M, K), \ \delta(\kappa(M, K), 1), 1\rangle$,
- $S' = \ S \ \cup \ \{ \ \tilde{t} \ \}$,
- t é o primeiro número que não está sendo usado por um
  local e $\tilde{t}$ representa o local recém-criado,
- $\sigma' = \ \langle \sigma_1\{\lambda x.null/\tilde{t}\}, \sigma_2\{\lambda s.\check{0}/\tilde{t}\}\rangle$,
- $n \geq 1$.

O componente $\langle false, \tilde{t}, (el)\rangle$ representa a nova pilha de execução inserida ao modelo em decorrência da criação do local $\tilde{t}$.

Iniciando uma *thread*

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el_n)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle \ \rightarrow$

$\langle X \ \cup \ \langle false, \psi, (el_1, \cdots, el'_n)\rangle, \langle false, \rho, (ee)\rangle, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle$

onde:

- $el_n = \langle en,\ op,\ st,\ ip \rangle$,
- st é a instrução **start** $< \alpha >$; ,
- $el'_n = \langle en,\ op,\ \delta(op, ip+1), ip+1 \rangle$,
- $ee = \langle en,\ om,\ \delta(om, 1), 1 \rangle$,
- $om = \mu(M, \tau(\alpha), run)$,
- $\rho = \xi(\alpha)$,
- $n \geq 1$.

O componente $\langle false, \rho, (el) \rangle$ representa a nova pilha de execução inserida ao modelo em decorrência da *thread* iniciada. O topo desta pilha, ou seja, o elemento el, aponta para a primeira instrução do método *run* do objeto α. O local da nova pilha é mesmo local onde se encontra o objeto α.

Publicando um serviço

$$\langle X \ \cup\ \langle false, \psi, (el_1, \cdots, el_n) \rangle,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle \ \rightarrow$$
$$\langle X \ \cup\ \langle false, \psi, (el_1, \cdots, el'_n) \rangle,\ \sigma,\ S,\ \tau,\ \gamma',\ \xi,\ \theta,\ \omega,\ M \rangle$$

onde:

- $el_n = \langle en,\ op,\ st,\ ip \rangle$,
- st é a instrução **publish** $< \zeta, \alpha >$; ,
- $el'_n = \langle en,\ op,\ \delta(op, ip+1), ip+1 \rangle$,
- $\gamma' = \gamma\{\alpha/\zeta\}$,
- $n \geq 1$.

A função γ é redefinida, passando a retornar o serviço α quando recebe como parâmetro o endereço representado pelo estado ζ.

Referenciando um serviço remoto

$$\langle X \ \cup\ \langle false, \psi, (el_1, \cdots, el_n) \rangle,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle \ \rightarrow$$
$$\langle X \ \cup\ \langle false, \psi, (el_1, \cdots, el'_n) \rangle,\ \sigma',\ S,\ \tau,\ \gamma,\ \xi,\ \theta',\ \omega,\ M \rangle$$

onde:

- $el_n = \langle en,\ op,\ st,\ ip \rangle$,
- st é a instrução **reference** $< \zeta, x >$; ,
- $el'_n = \langle en,\ op,\ \delta(op, ip+1), ip+1 \rangle$,
- $\theta' = \theta\{true/(\alpha, en)\}$,
- $\sigma' = \langle \sigma_1\{(\sigma_1(en)\{\alpha/x\})/en\}, \sigma_2 \rangle$,
- $\alpha = \gamma(\zeta)$,
- $n \geq 1$.

A chamada de função $\gamma(\zeta)$ retorna o serviço remoto α previamente publicado. A função θ é redefinida, indicando que o objeto α é um serviço remoto para a entidade de execução en. A função σ passa a retornar o valor α para a variável x da entidade de execução en.

Invocando um método local

$\langle X \cup \langle false, \psi, (el_1, \cdots, el_n) \rangle,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle \rightarrow$
$\langle X \cup \langle false, \psi, (el_1, \cdots, el_{n+1}) \rangle,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle$

onde:

- $el_n = \langle en,\ op,\ st,\ ip \rangle$,
- st é a instrução $\alpha.m(\ )$; ,
- $el_{n+1} = \langle \alpha,\ \mu(M, \tau(\alpha), m),\ \delta(\mu(M, \tau(\alpha), m), 1), 1 \rangle$,
- $\theta(\alpha, en) = false$,
- $n \geq 1$.

O retorno da chamada de função $\theta(\alpha, en)$ mostra que o objeto é local, portanto, trata-se de uma invocação de método local. O novo elemento empilhado el_{n+1}, aponta para a primeira instrução do método m do objeto α.

Realizando um retorno com mais de um elemento na pilha

$\langle X \cup \{es\},\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle \rightarrow$
$\langle X \cup \{es'\},\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M \rangle$

onde:

- $es = \langle false, \psi, (el_1, \cdots, el_n)\rangle$,
- $es' = \langle false, \psi, (el_1, \cdots, el'_{n-1})\rangle$,
- $(el_n)_3$ é a instrução **return**; ,
- $el_{n-1} = \langle en,\ op,\ st,\ ip\rangle$,
- $el'_{n-1} = \langle en,\ op,\ \delta(op, ip+1), ip+1\rangle$,
- $n > 1$.

O topo da pilha, ou seja, o elemento el_n, é desempilhado. Como ainda existem elementos na pilha ($n > 1$), o novo topo passa a apontar para a próxima instrução da operação *op*.

Realizando um retorno com um elemento na pilha

$\langle X\ \cup\ \{es\},\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M\rangle\ \rightarrow$
$\langle X,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M\rangle$

onde:

- $es = \langle false, \psi, (el)\rangle$,
- $(el)_3$ é a instrução **return**; ,
- $(es, xs) \notin \omega$.

Como existe um único elemento na pilha, o retorno só será local se não existir um *bind* entre es e outra pilha qualquer. A pilha de execução es é retirada da configuração, pois quando desempilhamos el ela fica sem elementos e conseqüentemente não controla mais um fluxo de execução.

Invocando um método remoto

$\langle X\ \cup\ es,\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega,\ M\rangle\ \rightarrow$
$\langle X\ \cup\ \{es', xs\},\ \sigma,\ S,\ \tau,\ \gamma,\ \xi,\ \theta,\ \omega',\ M\rangle$

onde:

- $es = \ \langle false, \psi, (el_1, \cdots, el_n)\rangle$,
- $es' = \ \langle true, \psi, (el_1, \cdots, el_n)\rangle$,
- $xs = \ \langle false, \rho, (ee)\rangle$,
- $el_n = \ \langle en, \ op, \ st, \ ip\rangle$,
- st é a instrução $\alpha.m(\ )$; ,
- $\xi(\alpha) = \ \rho$,
- $ee = \ \langle \alpha, \ \mu(M, \tau(\alpha), m), \ \delta(\mu(M, \tau(\alpha), m), 1), 1\rangle$,
- $\theta(\alpha, en) = \ true$,
- $\omega' = \ \omega\{es/xs\}$,
- $n \geq 1$.

O retorno da chamada de função $\theta(\alpha, en)$ mostra que o objeto é remoto, portanto, trata-se de uma invocação de método remota. A pilha de execução es é bloqueada e é criada a pilha xs no mesmo local do objeto remoto. O topo da pilha xs aponta para a primeira instrução do método m do objeto α. Um *bind* é criado entre as pilhas es e xs através da redefinição da função ω.

Realizando um retorno remoto

$\langle X \ \cup \ \{es, xs\}, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega, \ M\rangle \ \rightarrow$
$\langle X \ \cup \ \{xs'\}, \ \sigma, \ S, \ \tau, \ \gamma, \ \xi, \ \theta, \ \omega', \ M\rangle$

onde:

- $es = \langle false, \psi, (ee)\rangle$,
- $(ee)_3$ é a instrução **return**; ,
- $xs = \langle true, \rho, (el_1, \cdots, el_n)\rangle$,
- $el_n = \ \langle en, \ op, \ st, \ ip\rangle$,
- $\omega(es) = xs$,
- $\omega' = \omega - \{(es, xs)\}$ (diferença de conjuntos),
- $xs' = \langle false, \rho, (el_1, \cdots, el'_n)\rangle$,
- $el'_n = \ \langle en, \ op, \ \delta(op, ip + 1), ip + 1\rangle$,
- $n \geq 1$.

A pilha xs é desbloqueada e a es retirada da configuração. O *bind* existente entre es e xs é removido com a redefinição de ω.

Capítulo 5

Simulador CROMOL

Neste capítulo, apresentamos o Simulador CROMOL. Inicialmente, mostramos como ocorre o processo de simulação baseada em modelos e como utilizar o Simulador CROMOL neste processo. Em seguida, descrevemos a arquitetura do simulador, explorando os principais componentes e a interação entre esses componentes. Por último, descrevemos o papel do simulador no processo de verificação de modelos. Uma vez que, a verificação de modelos CROMOL é um objetivo a ser alcançado em trabalhos futuros.

5.1 Simulação de Sistemas Baseada em Modelos e o Simulador CROMOL

Com o objetivo de possibilitar a execução de modelos e, conseqüentemente, simular a execução dos sistemas que esses modelos representam, desenvolvemos um simulador baseado em modelos CROMOL. Além do simulador, desenvolvemos também um tradutor de modelos. Ele é responsável por traduzir a representação textual de um modelo CROMOL, para uma representação que possa ser compreendida pelo simulador. Outra função do tradutor é verificar se o modelo está sintaticamente correto. Uma tradução apenas será concluída, se essa verificação for realizada com êxito. Na Figura 5.1 mostramos a interconexão do tradutor com o Simulador CROMOL, fazendo uma analogia com a plataforma Java.

O Simulador CROMOL oferece suporte a dois tipos de simulação. O primeiro

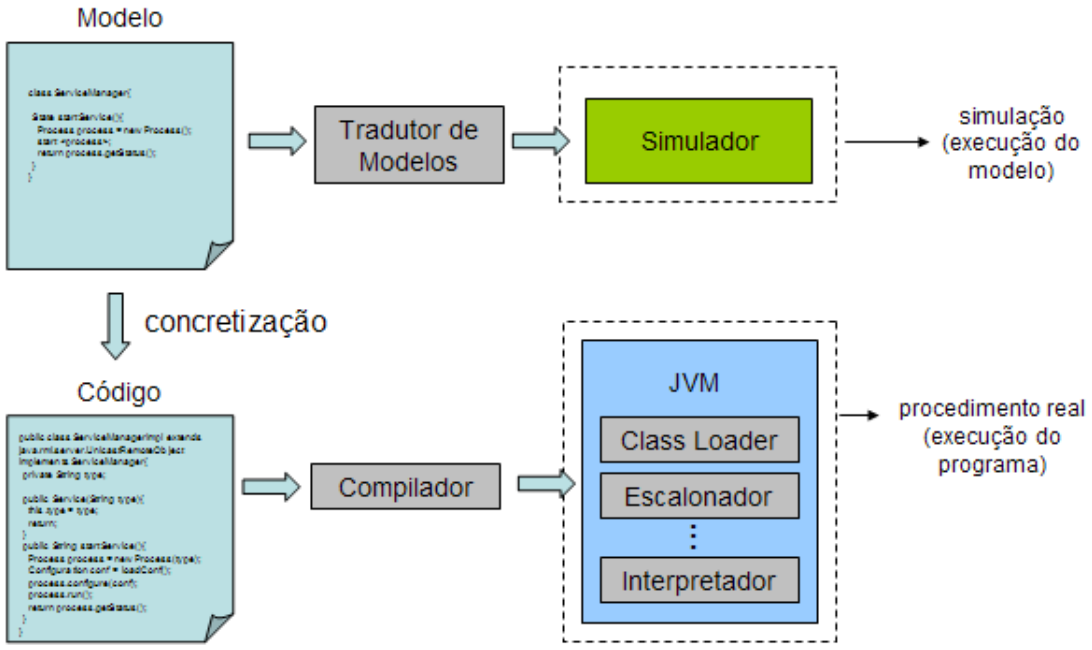

Figura 5.1: Interconexão do tradutor com o Simulador CROMOL, fazendo uma analogia com a plataforma Java.

tipo é chamada simulação automática. Neste tipo de simulação o controle do fluxo de execução do modelo é feito automaticamente, ou seja, a escolha de qual ação deve ser executada em um determinado momento é realizada de forma automática, baseada em uma semente que é fornecida no início da simulação. Para uma mesma semente temos sempre o mesmo *trace* de execução. Caso deseje realizar uma simulação automática, o usuário deve implementar um programa simples e de interface bem definida, que servirá como um guia de execução. Esse programa será responsável por informar ao simulador qual das ações habilitadas em um determinado momento deve ser executada. Esta decisão é tomada com base na semente citada anteriormente. O segundo tipo de simulação é a guiada pelo usuário. Nesse tipo de simulação, quem controla o fluxo de execução do modelo é o usuário, ou seja, ele é quem escolhe qual ação deve ser executada em um determinado momento.

Neste trabalho, focamos na simulação guiada pelo usuário. A justificativa é simples, uma simulação automática apenas faria sentido, caso realizássemos uma verificação formal no decorrer da execução do modelo, ou seja, caso nós implementássemos um modulo responsável por analisar a configuração atual, procurando por comportamentos

indesejáveis. Entretanto isso está fora do escopo de nosso trabalho. Portanto, optamos por uma simulação guiada pelo usuário, onde o próprio usuário poderá analisar a configuração atual, juntamente com os *traces* de execução do modelo, tentando encontrar erros no comportamento modelado.

A fim de facilitar a simulação guiada pelo usuário, desenvolvemos uma interface gráfica para o tradutor e o simulador. Através dela podemos validar a sintaxe dos modelos construídos, como também executá-los, simulando a execução do sistema modelado. Esta interface possui duas janelas principais: a janela para edição de modelos e a janela para execução de modelos. A janela para edição de modelos pode ser vista na Figura 5.2, ela permite:

- Criar, salvar, abrir e editar modelos.
- Traduzir um modelo, validando-o sintaticamente.
- Iniciar a execução de um modelo sintaticamente correto.

A janela para execução de modelos pode ser vista na Figura 5.3. Durante a execução de um modelo, o usuário pode interagir com essa janela, criando novos locais e escolhendo dentre as instruções habilitadas qual deve ser executada em um determinado momento. Cada instrução habilitada é mostrada através da pilha de execução a qual ela está associada. Essa janela também permite que o usuário acesse algumas informações relacionadas à execução do modelo, como ilustra a Figura 5.4. Essas informações serão úteis durante a análise do comportamento modelado.

O *trace de execução* mostra a seqüência de ações executadas desde o início da execução do modelo. O *trace de estado* mostra essa mesma seqüencia de ações, mas intercalada pelo valor das variáveis de estado observado após execução de cada uma das ações. Assim, o formato do *trace de estado* é: **ação** - **variáveis de estado** - **ação**... Isso possibilita ao usuário verificar as mudanças de estado resultantes da execução de uma ação. O *estado atual de execução* pode ser visto através das *pilhas de execução* e das *variáveis de estado*. As *variáveis de estado*, por sua vez, podem ser exibidas por escopo ou em sua totalidade.

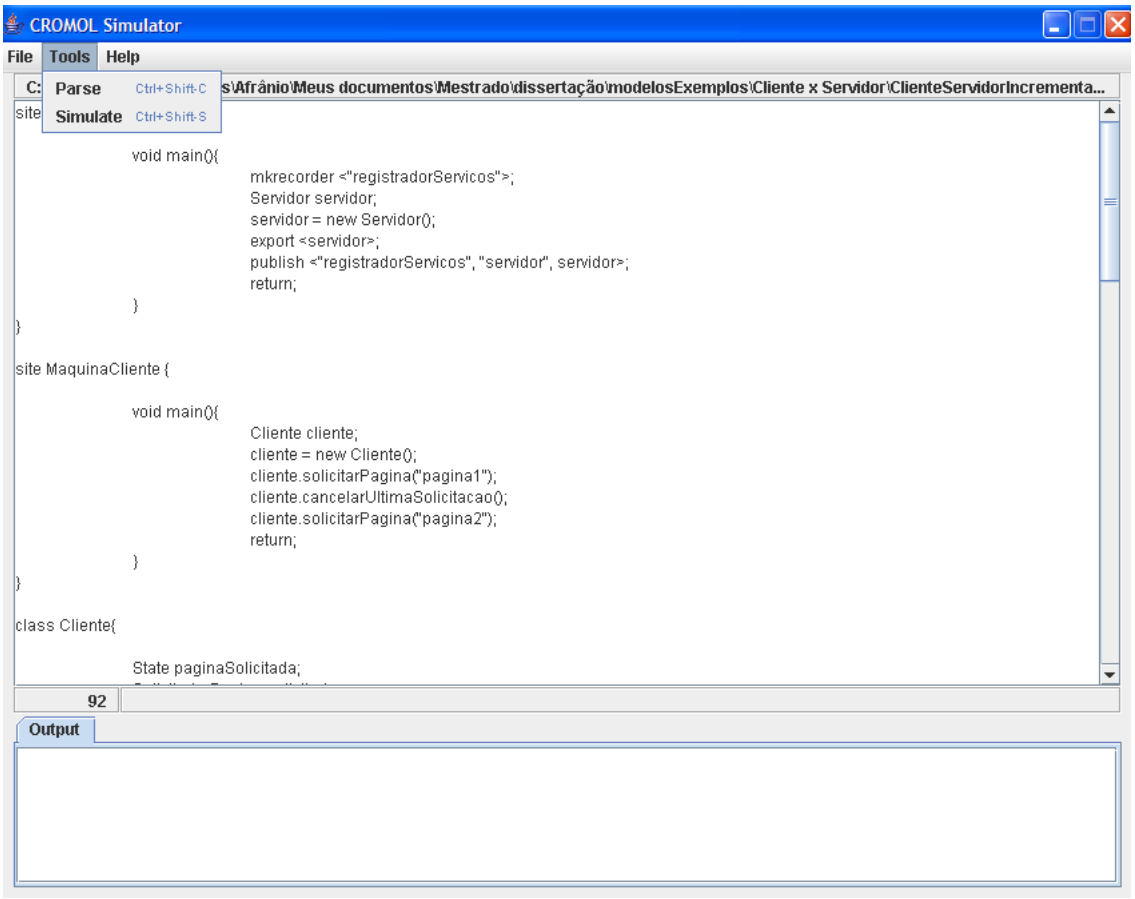

Figura 5.2: Janela para edição de modelos no simulador CROMOL.

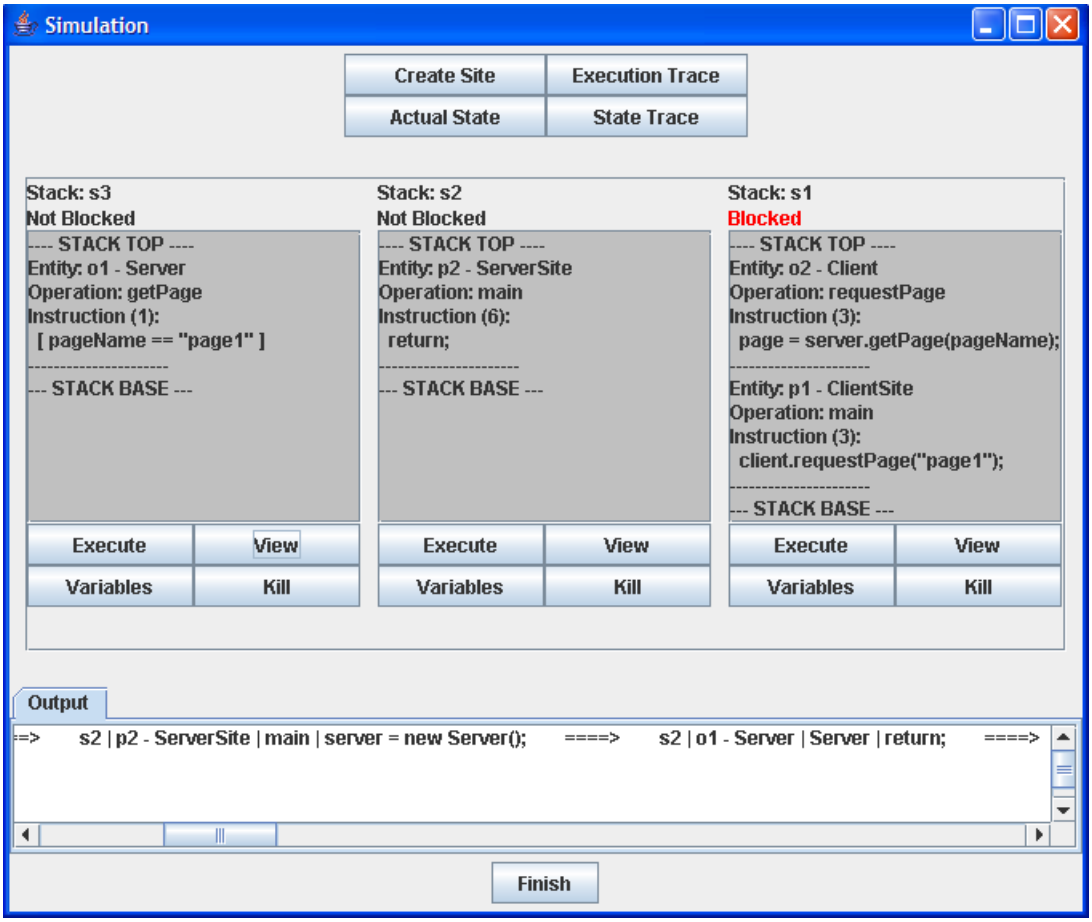

Figura 5.3: Janela para execução de modelos no simulador CROMOL.

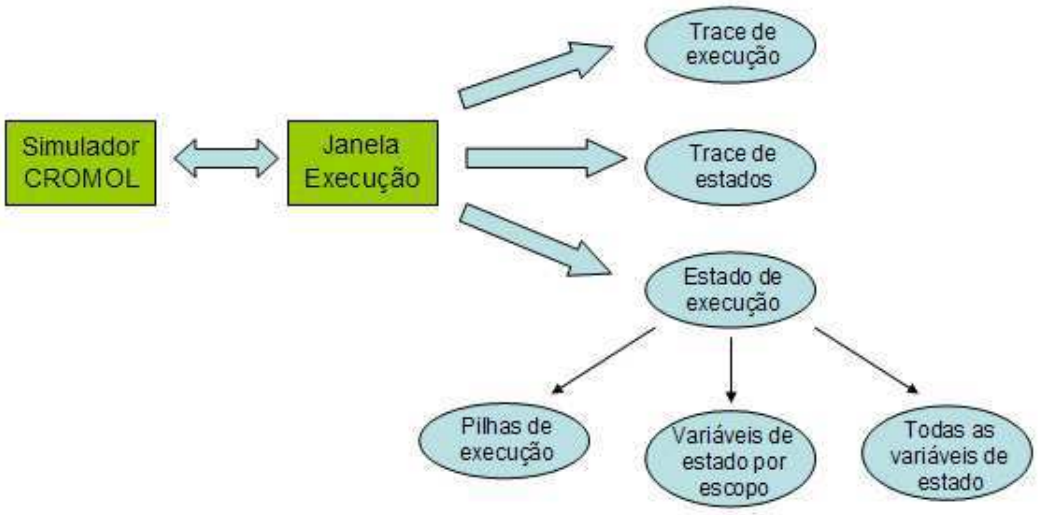

Figura 5.4: Informações que podem ser obtidas durante a execução de um modelo.

5.2 A arquitetura do Simulador CROMOL

Conforme mencionamos na Seção 5.1, o Simulador CROMOL recebe como entrada a representação de um modelo produzida por um tradutor de modelos. Em virtude dessa integração, antes de apresentar a arquitetura do simulador, mostraremos a arquitetura desse tradutor. Tal arquitetura é ilustrada na Figura 5.5.

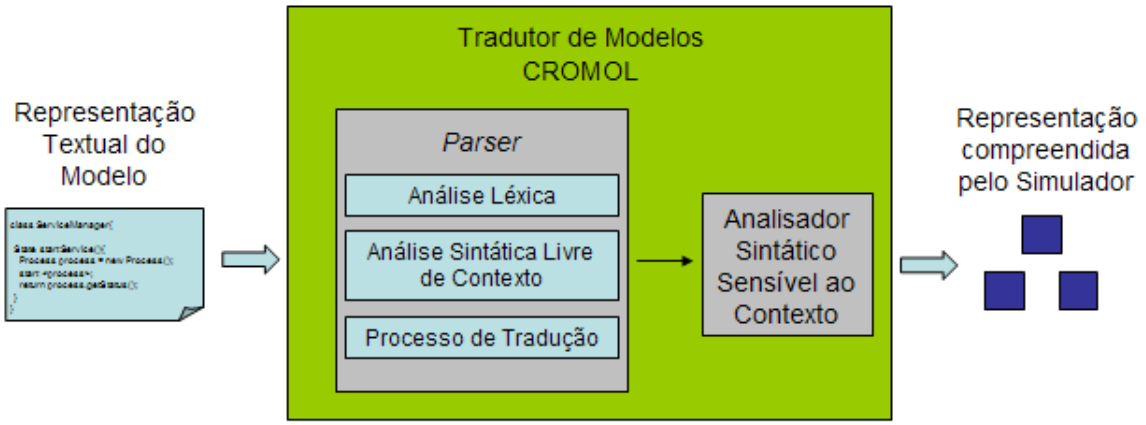

Figura 5.5: Arquitetura do Tradutor de Modelos CROMOL.

O Tradutor de Modelos CROMOL tem dois componentes principais. O primeiro é o *Parser*. Este componente foi gerado automaticamente usando a ferramenta JavaCC [Sreenivas e Sankar, 2005]. Para isto, fornecemos como entrada para essa ferramenta a gramática livre de contexto da linguagem CROMOL e o código relativo ao processo de tradução de modelos. O *Parser* realiza sobre um modelo, uma análise livre de contexto, com o objetivo de verificar se o modelo está sintaticamente correto. Além disso, ele também realiza a tradução da representação textual de um modelo para uma representação que possa ser compreendida pelo simulador. O segundo componente do tradutor é o *Analisador Sintático Sensível ao Contexto*. Este componente é responsável por checar se o modelo, já traduzido, é válido em relação as condições sensíveis ao contexto mostradas na Seção 3.2.4.

A arquitetura do Simulador CROMOL pode ser vista na Figura 5.6. O componente *Gerente de Execução* é responsável por fornecer à *Interface com Usuário* o conjunto de ações habilitadas em um dado momento. Outra função do *Gerente de Execução* é receber da *Interface com Usuário* a ação selecionada pelo usuário e enviar o comando que

executa essa ação ao componente *Executor*. Quando recebe um comando, o *Executor* executa este comando interagindo e modificando o componente *Configuração*.

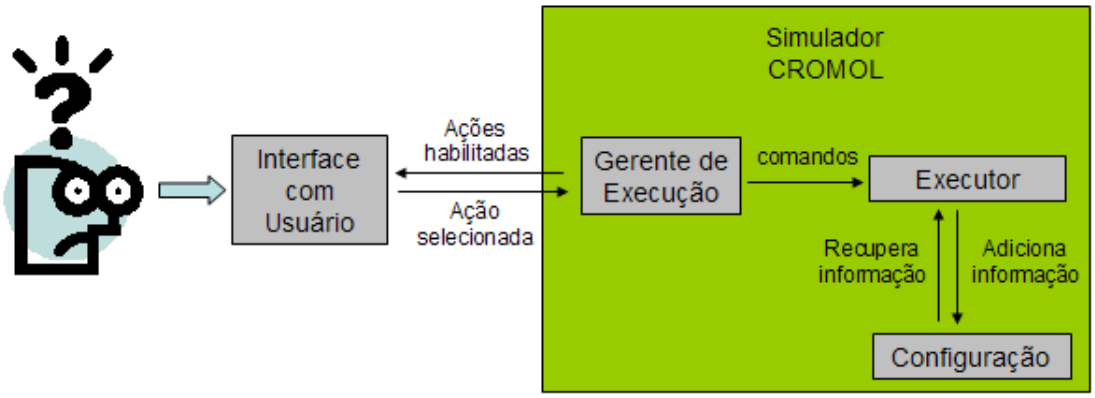

Figura 5.6: Arquitetura do Simulador CROMOL.

5.3 O Simulador CROMOL no Processo de Verificação de Modelos

Como vimos no Capítulo 4, CROMOL possui uma semântica matematicamente definida. Assim, os modelos desenvolvidos em CROMOL podem passar pelo processo de verificação descrito anteriormente. Entretanto, até o término deste trabalho, todas ferramentas necessárias para tornar viável o processo de verificação de modelos CROMOL não haviam sido desenvolvidas. Atualmente possuímos apenas o Simulador CROMOL. Além do simulador, precisamos de um *Gerador de Espaço de Estados* e de um *Verificador de Propriedades*. Na Figura 5.7 mostramos a integração do simulador a essas duas ferramentas.

O *Gerador de Espaço de Estados* interage com o simulador, executando todos os *traces* possíveis de um modelo. Para isso, quando ele chegar em um ponto onde mais de uma ação estiver habilitada, ele deve guardar o estado da execução naquele ponto e escolher um *trace* a seguir. Chegando ao fim do *trace*, o *Gerador de Espaço de Estados* deve restaurar o estado salvo anteriormente e escolher uma outra ação a ser executada, ou seja, uma ação que o leve por outro *trace*. Esse processo continua até que todo o espaço de estados tenha sido gerado. O *Verificador de Propriedades* apenas percorre o

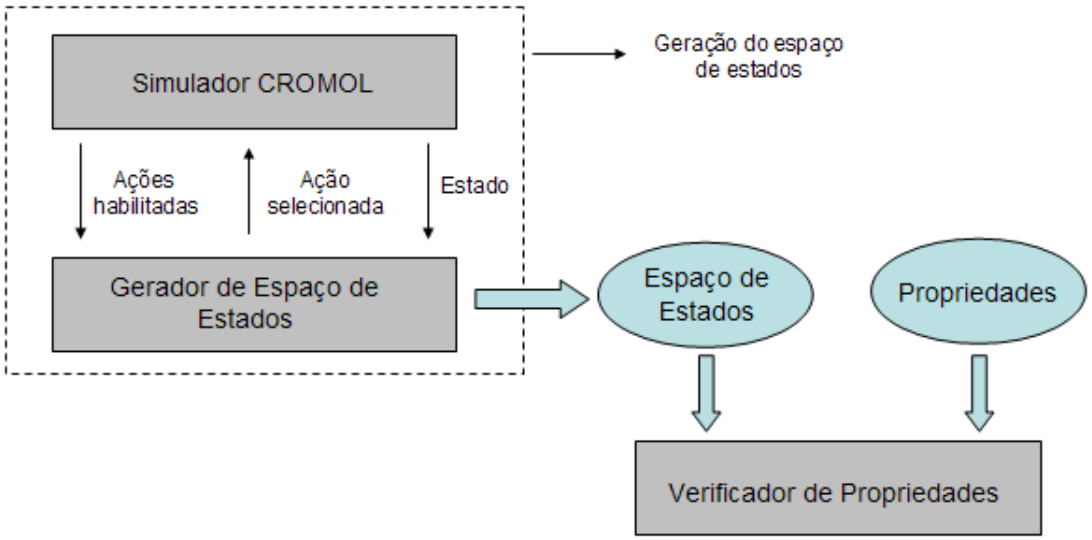

Figura 5.7: Integração das ferramentas necessárias para tornar viável a verificação de modelos CROMOL.

espaço de estados, verificando se este satisfaz as propriedades especificadas.

Capítulo 6

Estudo de Caso

Com o objetivo de contribuir para a validação de nosso trabalho, elaboramos o estudo de caso apresentado neste capítulo. O estudo de caso consiste na modelagem de um sistema de segurança doméstica chamado *SafeHome*. Esse sistema foi especificado por Pressman no livro intitulado *Engenharia de Software* [Pressman, 2002]. Entretanto, neste trabalho efetuamos algumas modificações na especificação original. Essas modificações vêem a tornar o *SafeHome* um pouco mais simples.

O estudo de caso será apresentado de acordo com a ordem descrita a seguir. Primeiro, mostramos o escopo do *SafeHome*, esse escopo já incorpora as modificações citadas anteriormente. Em seguida, mostramos o modelo CROMOL que representa o comportamento descrito nesse escopo. Por último, analisamos alguns traces obtidos através do processo de simulação da execução do *SafeHome*.

6.1 Escopo do *SafeHome*

O *SafeHome* é um sistema de segurança doméstica que visa à proteção de uma residência contra algumas situações de perigo. Exemplos dessas situações são: invasão ilegal, incêndio, arrombamento e etc. Uma vez detectada uma dessas situações, o sistema deve soar um alarme sonoro e comunicar o ocorrido a um serviço de segurança.

As situações de perigo são detectadas por sensores espalhados pela residência. Temos um tipo específico de sensor para cada uma das situações de perigo. Um sensor é identificado de forma única. Através dessa identificação, podemos obter a localização

do sensor em relação a residência e o tipo de situação de perigo que ele detecta.

Para que o sistema possa operar, ele deve antes ser configurado. A configuração, assim como qualquer outra interação com o usuário, é realizada por intermédio de um painel de controle. Durante a configuração devem ser informados a senha de acesso ao sistema e o serviço de segurança que deve ser acionado quando alguma ocorrência (situação de perigo) precisar ser tratada. Após configurado, o usuário pode adicionar sensores ao sistema.

O sistema só pode ser ativado se estiver devidamente configurado. Para ativar o sistema é necessária a senha de acesso cadastrada durante a fase de configuração. Da mesma forma, o sistema apenas pode ser desativado se essa senha de acesso for fornecida. Quando ativado, o sistema irá verificar o ambiente, procurando identificar situações de perigo. Quando temos a ocorrência de uma situação de perigo, o estado do sensor que identificou tal ocorrência é alterado. Percebendo uma mudança em seu estado interno, o sensor registra, em um gerenciador de ocorrências, a ocorrência que provocou essa mudança. O gerenciador de ocorrências, por sua vez, aciona o serviço de segurança informado durante a configuração do sistema e dispara um alarme sonoro. Esse alarme pode ser desligado pelo usuário, desde que seja fornecida a senha de acesso.

6.2 O Modelo CROMOL do *SafeHome*

O modelo CROMOL do *SafeHome* pode ser visto no Apêndice B, ele foi desenvolvido com base no escopo apresentado na Seção 6.1. O diagrama de classes do modelo é mostrado na Figura 6.1. O *PainelControle* é responsável pela interação com o usuário, sendo utilizado para:

- Configurar, ativar e desativar o sistema.
- Adicionar sensores ao sistema.
- Desligar o alarme sonoro.

Cada *Sensor* adicionado ao sistema é responsável por verificar no ambiente a ocorrência de situações de perigo. Identificada uma ocorrência, o *Sensor* responsável

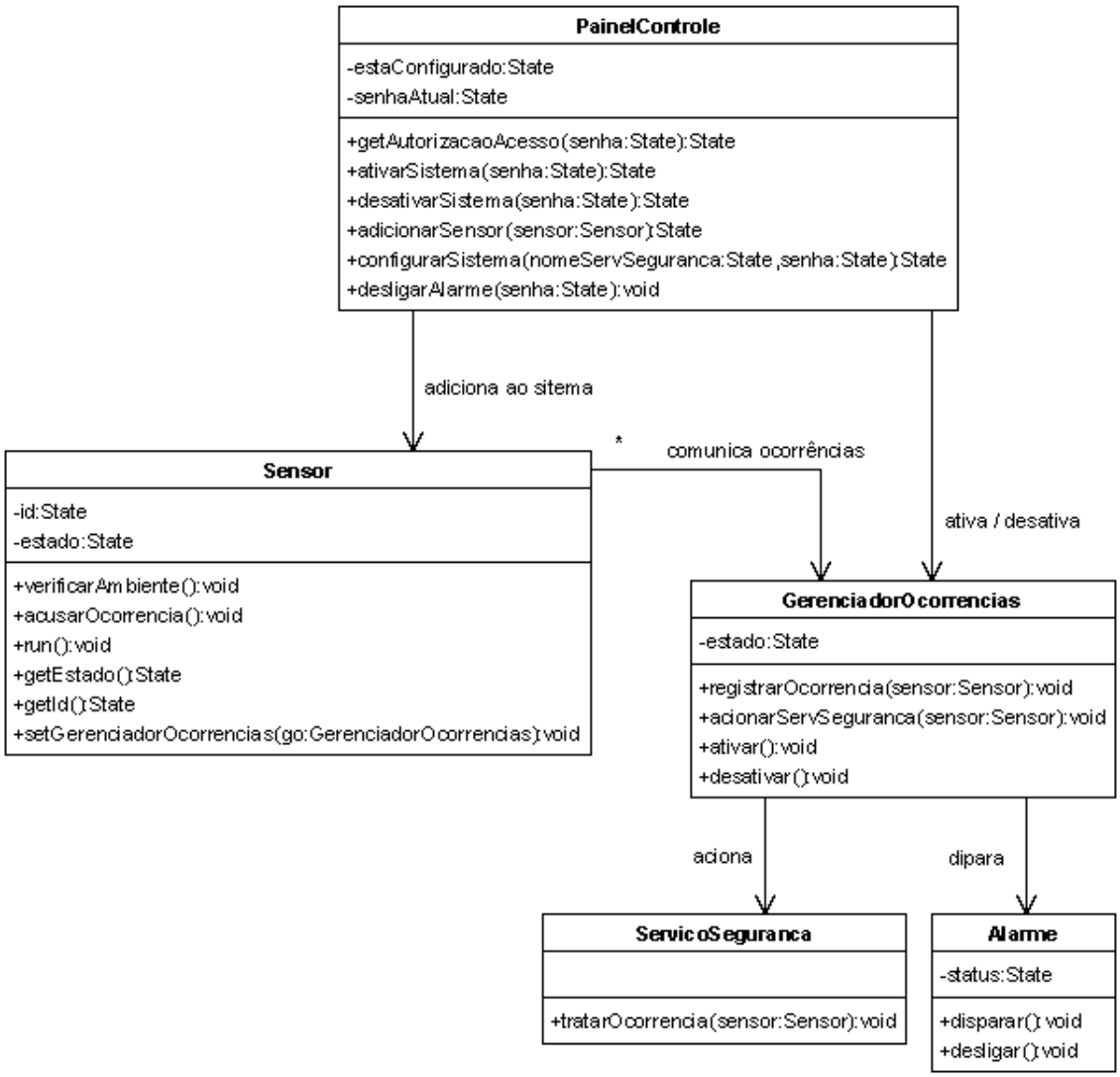

Figura 6.1: Diagrama de classes do *SafeHome*.

registra essa ocorrência em um *GerenciadorOcorrencias*. Em resposta ao registro de uma ocorrência, o *GerenciadorOcorrencias* dispara um *Alarme* e aciona o *ServicoSeguranca* para que a ocorrência possa ter tratada. Como cada *Sensor* deve verificar paralelamente o ambiente, eles devem ter seu próprio fluxo de execução, ou seja, eles precisam ser objetos executáveis. Outro ponto importante é que a comunicação entre o *GerenciadorOcorrencias* e o *ServicoSeguranca* é realizada através de um meio não confiável, pois estes se encontram em locais diferentes.

Além das classes citadas anteriormente, existem no modelo duas definições de locais, a *CasaCliente* e a *EmpresaSeguranca*. Na *CasaCliente* encontraremos objetos do tipo *PainelControle*, *GerenciadorOcorrencias*, *Sensor* e *Alarme*. Na *EmpresaSeguranca* encontraremos um objeto do tipo *ServicoSeguranca*, que poderá ser acessado como um serviço remoto.

O modelo também possui algumas classes auxiliares que servem para simular eventos externos ao *SafeHome*. Essas classes são: *ConfiguradorSistema*, *AdicionadorSensor*, *AtivadorSistema*, *EstimuladorSensor* e *DesativadorSistema*. Durante a execução do modelo, nada é garantido sobre a ordem de acontecimento dos eventos modelados por essas classes. Assim, podemos simular o comportamento do sistema sobre diversas ordens de acontecimento de eventos.

6.3 Analisando *Traces* da Simulação da Execução do *SafeHome*

Usando o Simulador CROMOL podemos obter *traces* da execução do modelo que desenvolvemos para o *SafeHome*. Nesta seção analisaremos alguns *traces* que julgamos terem um papel importante na execução do modelo. O primeiro *trace* a ser analisado representa uma tentativa de ativar o sistema sem que este tenha sido antes configurado. Este *trace* inicia quando um *AtivadorSistema* invoca o método *ativarSistema* do *PainelControle*, conforme podemos observar na instrução:

```
painelControle.ativarSistema(senha);
```

O comportamento do método *ativarSistema* é mostrado a seguir:

```
State ativarSistema(State senha) {
    State autorizacao;
    autorizacao = this.getAutorizacaoAcesso(senha);
    [autorizacao == "ERRO"]
        return "ERRO";
    gerenciadorOcorrencias.ativar();
    return "OK";
}
```

Este método invoca o método *getAutorizacaoAcesso* a fim de verificar se o acesso ao sistema é permitido. O método *getAutorizacaoAcesso* é apresentado a seguir:

```
 State getAutorizacaoAcesso(State senha) {
    [estaConfigurado == "NAO"]
        return "ERRO";
    [senhaAtual != senha]
        return "ERRO";
    return "OK";
}
```

A primeira instrução do método *getAutorizacaoAcesso* é verificar se o sistema está configurado. Como o sistema ainda não está configurado é retornado um erro para o método *ativarSistema*. Quando recebe o erro, o método *ativarSistema* repassa esse erro ao *AtivadorSistema*. Portanto, temos o comportamento desejado, ou seja, quando o sistema não está configurado ele não pode ser ativado.

O segundo *trace* a ser analisado representa uma tentativa de desativar o sistema já configurado, mas com uma senha de acesso inválida. Este *trace* inicia quando um *DesativadorSistema* invoca o método *desativarSistema* do *PainelControle*, como pode ser visto na instrução:

```
painelControle.desativarSistema(senha);
```

O comportamento do método *desativarSistema* é mostrado a seguir:

```
State desativarSistema(State senha) {
    State autorizacao;
    autorizacao = this.getAutorizacaoAcesso(senha);
    [autorizacao == "ERRO"]
        return "ERRO";
    gerenciadorOcorrencias.desativar();
    return "OK";
}
```

Este método invoca o método *getAutorizacaoAcesso* a fim de verificar se o acesso ao sistema é permitido. O método *getAutorizacaoAcesso* inicialmente verifica se o sistema está configurado. Como o sistema já está configurado prosseguimos com a instrução que checa se a senha fornecida corresponde a senha de acesso ao sistema. Ao verificar que a senha fornecida não corresponde a senha de acesso, o método *getAutorizacaoAcesso* retorna um erro para o método *desativarSistema*. Quando recebe o erro, o método *desativarSistema* repassa esse erro ao *DesativadorSistema*. Portanto, temos o comportamento desejado, ou seja, não podemos desativar o sistema com uma senha inválida.

Por último, analisaremos o *trace* desencadeado pela detecção de uma situação de perigo qualquer. Este *trace* inicia quando um *EstimuladorSensor* invoca o método *acusarOcorrencia* de um *Sensor*, conforme podemos observar na instrução:

```
sensor.acusarOcorrencia();
```

O corpo do método *acusarOcorrencia* é mostrado a seguir:

```
void acusarOcorrencia() {
    estado = "OCORRENCIA_DETECTADA";
    return;
}
```

Este método apenas altera o estado do *Sensor*, informando que ele detectou a ocorrência de uma situação de perigo. Quando este *Sensor* for verificar o ambiente, através do método *verificarAmbiente*, ele verá que seu estado interno sinaliza que uma

ocorrência foi detectada. Como conseqüência, ele invocará o método *registrarOcorrencia* do *GerenciadorOcorrencias*. O comportamento do método *verificarAmbiente* pode ser visto a seguir:

```
void verificarAmbiente() {
    [estado != "SEM_OCORRENCIA"]
        gerenciadorOcorrencias.registrarOcorrencia(this);
    return;
}
```

O método *registrarOcorrencia* verifica se o sistema está ativo. Caso esteja, ele dispara o *Alarme* e solicita ao *ServicoSeguranca* que a ocorrência seja tratada. Observe que o *ServicoSeguranca* é um serviço remoto. O método *registrarOcorrencia* é mostrado a seguir:

```
void registrarOcorrencia(Sensor sensor) {
    [estado == "INATIVO"]
        return;
    alarme.disparar();
    this.acionarServSeguranca(sensor);
    return;
}
```

Assim, quando uma situação de perigo é detectada por um *Sensor*, acontece aquilo especificado no escopo do *SafeHome*, ou seja, um *ServicoSeguranca* é acionado e um *Alarme* sonoro é disparado.

Observe que os *traces* analisados refletem, de forma concreta, comportamentos apresentados na Seção 6.1. Além disso, as entidades (objetos) e os locais de execução do sistema estão explicitamente descritos no modelo. Assim, CROMOL demonstra ser uma alternativa viável para a modelagem comportamental e orientada a objetos de sistemas distribuídos.

Capítulo 7

Conclusões

Neste trabalho apresentamos a linguagem CROMOL. Desenvolvemos CROMOL com o objetivo de facilitar a modelagem comportamental e formal de sistemas a serem implementados utilizando a linguagem de programação Java e o modelo de comunicação entre objetos distribuídos RMI. CROMOL tem como principal característica o fato de que os modelos criados são estruturalmente próximos de sua implementação.

Além da linguagem, desenvolvemos também uma ferramenta que, através da execução automática dos modelos produzidos, simula a execução dos sistemas modelados. Isto permite uma validação inicial do sistema, antes mesmo de implementá-lo.

Estes resultados vêm a beneficiar os desenvolvedores de sistemas distribuídos e orientados a objetos, principalmente, aqueles sistemas mais complexos, que exigem muito rigor durante sua validação.

7.1 Caracterização da Linguagem CROMOL

Como forma de contribuir para a validação do nosso trabalho, caracterizamos CROMOL utilizando o mesmo conjunto de parâmetros definidos no Capítulo 2, conforme pode ser visto na Tabela 7.1.

O baixo nível de abstração e a proximidade à linguagem Java aproximam o modelo à implementação, facilitando o mapeamento de erros descobertos no modelo para a implementação e simplificando a sincronização, a navegação e a garantia de conformidade entre modelo e código. Além disso, um modelo mais concreto viabiliza a geração de

Parâmetro	**Valor**
Nível de abstração	Baixo
Tipo de notação da linguagem	Textual
Suporte a orientação a objetos	Sim
Suporte ferramental	Intermediário
Suporte a mensagem síncronas	Sim
Suporte a mensagem assíncronas	Sim
Proximidade a Java	Sim
Suporte ao conceito de local	Sim

Tabela 7.1: Características da Linguagem CROMOL

código a partir do modelo e possibilita uma verificação mais ampla do sistema antes de sua implementação.

A notação textual simplifica o desenvolvimento de ferramentas e permite o aproveitamento de parte da sintaxe de Java. Observe que este aproveitamento resulta em outro ponto positivo, que é uma imediata familiaridade dos programadores Java com CROMOL.

O suporte a orientação a objetos facilita a modelagem e vai de encontro a uma forte tendência atual, permitindo que, no momento de modelar os sistemas, os desenvolvedores possam pensar em termos de entidades e interações entre elas.

Neste trabalho, como suporte ferramental, desenvolvemos para CROMOL um editor e executor de modelos, mas nossa meta é desenvolver ferramentas que viabilizem a verificação dos modelos, conforme podemos observar na Seção 7.3 que se refere a trabalhos futuros.

O envio de uma mensagem em CROMOL é representado pela invocação de um método, portanto, o envio de mensagens é síncrono, indo de encontro a semântica de Java e RMI. Entretanto, o envio de mensagens assíncronas pode ser simulado através da criação de uma *thread* que seria responsável pelo envio da mensagem. Assim, CROMOL oferece suporte a mensagens assíncronas de forma indireta.

O suporte ao conceito de local aproxima a execução do modelo à execução do sistema, possibilitando a verificação de propriedades que levam em consideração o local

onde um objeto executa.

Analisando as características de CROMOL, concluímos que atendemos de forma satisfatória aos critérios estabelecidos na Seção 2.5 durante a definição de uma linguagem ideal para modelagem de sistemas distribuídos em Java. O único parâmetro que não esta em plena consonância com os critérios estabelecidos, diz respeito ao suporte ferramental, pois para CROMOL ele possui valor "Intermediário", enquanto que para a linguagem definida como ideal ele possui valor "Alto". Entretanto, conforme mencionando anteriormente, pretendemos, em trabalhos futuros, evoluir nesse aspecto.

7.2 Limitações

Como todo trabalho recém desenvolvido, CROMOL possui algumas limitações. Essas limitações não são definitivas e devem vir a ser superadas em trabalhos futuros. Como limitação inicial temos o fato de que CROMOL é direcionada a modelagem de sistemas a serem implementados utilizando-se Java e RMI. Caso o modelador deseje utilizar outra linguagem de programação ou outro modelo de comunicação entre objetos distribuídos, ele deverá está certo da conformidade entre a linguagem a ser utilizada e Java ou entre o modelo de comunicação a ser utilizado e RMI. Esta conformidade será de suma importância para que o modelador possa usufruir de todos os benefícios oferecidos por CROMOL.

Outra limitação está relacionada à edição de modelos. Neste trabalho definimos uma representação textual para criação e apresentação dos modelos. Entretanto, entendemos que uma representação gráfica, apesar de ter um maior custo se comparada a textual, pode vir a contribuir para o trabalho, pois facilita a compreensão, edição e execução de modelos. É importante observar que uma mesma linguagem, no caso CROMOL, pode ter mais de uma representação, seja gráfica ou textual, desde que sua semântica operacional permaneça inalterada.

A terceira limitação é ausência de algumas construções que existem na linguagem Java. Por exemplo, não temos como representar, diretamente, laços em CROMOL, enquanto em Java, laços são representados diretamente com construções como o *while* e o *for*. Outro exemplo, é impossibilidade de se modelar listas ou *arrays*. Esta limitação

é de certa foram necessária, pois se todas as construções de Java fossem inclusas na linguagem, teríamos, praticamente, uma outra linguagem de programação. Assim, temos que encontrar um meio termo.

A última limitação identificada é a ausência de ferramentas que possibilitem uma verificação mais profunda dos sistemas modelados, mais especificamente ferramentas que viabilizem o emprego de técnicas de verificação formal de modelos.

7.3 Trabalhos Futuros

A continuidade deste trabalho será direcionada à solução das limitações apresentadas na Seção 7.2. O trabalho futuro prioritário é o desenvolvimento de ferramentas que possibilitem, de forma automática, o emprego de técnicas de verificação formal de modelos. Os demais trabalhos futuros são:

- Desenvolvimento de ramificações de CROMOL que tratariam outras linguagens de programação, como também outros modelos de comunicação entre objetos distribuídos.

- Criação de novas formas de representação de modelos. O objetivo tornar a criação e a edição dos modelos mais simples e intuitiva. É importante observar que novas formas de representação exigem o desenvolvimento de ferramentas que lhes ofereçam suporte.

- Análise do conjunto de ações atual, com a possibilidade de inserção de novas ações e remoção de ações existentes.

Bibliografia

[Agha, 1983] Agha, G. A. (1983). *Actors: A Model of Concurrent Computation in Distributed Systems*. Massachusetts Institute of Technology.

[Aldrich, 2002] Aldrich, J., Chambers, C., e Notkin, D. (2002). *ArchJava: connecting software architecture to implementation*. ICSE '02: Proceedings of the 24th International Conference on Software Engineering, ACM Press, páginas 187-197, New York, NY, USA.

[Bérard, 1999] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., e McKenzie, P. (1999). *Systems and Software Verification*. Springer.

[Clarke, 1999] Clarke, E. M., Grumberg, O., e Peled, D. A. (1999). *Model Checking*. MIT.

[Dwyer e Hatcliff, 2001] Dwyer, M. e Hatcliff, J. (2001). *Specification and Verification of Reactive Systems*. Kansas State University. www.cs.sun.ac.za/~abvdm/model/hatcliff/SPIN-Temporal-Logic.pdf - último acesso em 20/11/2004.

[Ehmety e Paulson, 2003] Ehmety, S. e Paulson, L. C. (2003). *The UNITY Formalism*. http://isabelle.in.tum.de/library/HOL/UNITY/document.pdf - último acesso em 02/10/2004.

[Glass, 2004] Glass, R. L. (2004). *The Mystery of Formal Methods Disuse*. Communications of the ACM.

[Gosling, 2000] Gosling, J., Joy, B., Steele, G., e Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley.

[Guerrero, 2002] Guerrero, D. D. S. (2002). *Redes de Petri Orientadas a Objeto*. Tese de Doutorado - COPELE, UFCG.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

[Ireland, 2001] Ireland, A. (2001). *Distributed Systems Programming F29NM1 - Promela I*. School of Mathematical and Computer Sciences, Heriot-Watt University - Edinburgh. http://www.macs.hw.ac.uk/~air/spin/lectures/lec-3-promela-1.ps - último acesso em 25/10/2004.

[Ireland, 2002] Ireland, A. (2002). *Distributed Systems Programming F29NM1 - Promela II*. School of Mathematical and Computer Sciences, Heriot-Watt University - Edinburgh. http://www.macs.hw.ac.uk/~air/spin/lectures/lec-4-promela-2.ps - último acesso em 28/10/2004.

[J. W. de Bakker e Zucker, 1985] J. W. de Bakker, J. J. Meyer, E. R. O. e Zucker, J. I. (1985). *Transition Systems, Infinitary Languages and the Semantics of Uniform Concurrency*. Proc. 17th ACM Symposium on Theory of Computing.

[Katoen, 1999] Katoen, J. P. (1999). *Concepts, Algorithms, and Tools for Model Checking*. Lecture notes of the course "Mechanised Validation of Parallel Systems" (course number 10359) at the Friedrich-Alexander Universität Erlangen-Nürnberg.

[Larman, 1999] Larman, C. (1999). *Utilizando UML e Padrões*. Prentice Hall.

[McMillan, 2005] McMillan, K. L. (2005). *The SMV Language*. http://www.cis.ksu.edu/santos/smv-doc/language/language.html - último acesso em 05/11/2004.

[Monin, 2003] Monin, J.-F. (2003). *Understanding Formal Methods*. Springer.

[Moreira, 2001] Moreira, C. M. (2001). *Estratégias de Reposição de Estoques em Supermercados: Avaliação por meio de Simulação*. Dissertação de Mestrado - Engenharia de Produção, UFSC.

[Murata, 1989] Murata, T. (1989). *Petri Nets: Properties, Analysis and Applications*. Proceedings of the IEEE, vol 77, No. 04, April 1989, páginas 541-580.

[Pierre America e Rutten, 1986] Pierre America, Jaco de Bakker, J. N. K. e Rutten, J. (1986). *Operational Semantics of Parallel Object-Oriented Language*. ACM-0-89791-175-X-1/86-0194, páginas 194-208.

[Plotkin, 1983] Plotkin, G. D. (1983). *An Operational Semantics for CSP*. Conference on Formal Description of Programming Concepts - II, páginas 199-225.

[Pressman, 2002] Pressman, R. S. (2002). *Engenharia de Software*. Mc Graw Hill.

[Santos, 2003] Santos, J. A. M. (2003). *Suporte a Análise e Verificação de Modelos RPOO*. Dissertação de Mestrado - COPIN, UFCG.

[Shaw e Garlan, 1996] Shaw, M. e Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Sreenivas e Sankar, 2005] Sreenivas, V. e Sankar, S. (2005). *Java Compiler Compiler*.

[Tanenbaum e Steen, 2002] Tanenbaum, A. S. e Steen, M. V. (2002). *Distributed Systems: Principles and Paradigms*. Prentice Hall.

[Winskel, 1994] Winskel, G. (1994). *The Formal Semantics of Programming Languages*. Massachusetts Institute of Technology Press.

[Wollrath e Waldo, 2005] Wollrath, A. e Waldo, J. (2005). *Java Remote Method Invocation*.

[Zimmerman, 2003] Zimmerman, D. M. (2003). *A UNITY-based Formalism for Dynamic Distributed Systems*. California Institute of Technology Pasadena. http://www.cs.unh.edu/~charpov/FMPPTA/2003/01.pdf - último acesso em 10/10/2004.

Apêndice A

Gramática de CROMOL

```
SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT:
       "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT:
       "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT:
       "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}

TOKEN :
{
  < LPAREN: "(" >
```

```
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LCLASP: "[" >
| < RCLASP: "]" >
| < EQUAL: "==" >
| < DIFFERENT: "!=" >
| < SEMICOLON: ";" >
| < DOT: "." >
| < COMMA: "," >
| < ASSIGN: "=" >
| < LT: "<" >
| < GT: ">" >
| < STATE: "State" >
| < CLASS: "class" >
| < MAIN: "main" >
| < NEW: "new" >
| < RETURN: "return" >
| < THIS: "this" >
| < SITE: "site" >
| < VOID: "void" >
| < EXPORT: "export" >
| < MKRECORDER: "mkrecorder" >
| < PUBLISH: "publish" >
| < REFERENCE: "reference" >
| < START: "start" >
| < SYNCHRONIZED: "synchronized" >
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
| < #LETTER:
```

```
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
| < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
```

```
      ]
  >
}

TOKEN : /* LITERALS */
{
< STATE_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

void Start() :
{}
{
    ( SiteDefinition() )+
    ( ClassDeclaration() )*
    <EOF>
}

void SiteDefinition() :
{}
{
    <SITE> Identifier() <LBRACE>
    ( VariableDeclaration() <SEMICOLON> )*
```

```
    <VOID> <MAIN> <LPAREN> <RPAREN> <LBRACE>
    ( Statement() )*
    <RBRACE>
    <RBRACE>
}

void ClassDeclaration() :
{}
{
    <CLASS> Identifier() <LBRACE>
    ( LOOKAHEAD(3) VariableDeclaration() <SEMICOLON> )*
    ( LOOKAHEAD(2) ConstructorDeclaration() )?
    ( LOOKAHEAD(2) MethodDeclaration() )*
    <RBRACE>
}

void ConstructorDeclaration() :
{}
{
    Identifier() <LPAREN> ( FormalParameterList() )? <RPAREN> <LBRACE>
    ( Statement() )*
    <RBRACE>
}

void MethodDeclaration() :
{}
{
    ( <SYNCHRONIZED> )? ( <VOID> | Type() ) Identifier() <LPAREN>
        ( FormalParameterList() )? <RPAREN> <LBRACE>
    ( Statement() )*
    <RBRACE>
}
```

```
void FormalParameterList() :
{}
{
    VariableDeclaration() ( FormalParameterRest() )*
}

void FormalParameterRest() :
{}
{
    <COMMA> VariableDeclaration()
}

void Statement() :
{}
{
    (
        LOOKAHEAD(3)
        VarDeclarationStatement()
    |
        LOOKAHEAD(3)
        ObjectCreationStatement()
    |
        LOOKAHEAD(4)
        AssignmentValueStatement()
    |
        ReturnStatement()
    |
        MessageSendStatement()
    |
        ExportStatement()
    |
        MkrecorderStatement()
    |
```

```
        PublishStatement()
    |
        StartStatement()
    |
        GuardStatement()
    |
        ReferenceStatement()
    )
}

void ReturnStatement() :
{}
{
    <RETURN> ( PrimaryExpression() )? <SEMICOLON>
}

void MessageSendStatement() :
{}
{
    (
        LOOKAHEAD(2)
        Identifier() <ASSIGN> MessageReceptor() <DOT> Identifier()  <LPAREN>
            ( ParametersList() )? <RPAREN> <SEMICOLON>
        |
        LOOKAHEAD(2)
        MessageReceptor() <DOT> Identifier() <LPAREN>
            ( ParametersList() )? <RPAREN> <SEMICOLON>
    )
}

void ObjectCreationStatement() :
{}
{
```

```
    Identifier() <ASSIGN> <NEW> Identifier() <LPAREN>
        ( ParametersList() )? <RPAREN> <SEMICOLON>
}

void AssignmentValueStatement() :
{}
{
    Identifier() <ASSIGN> PrimaryExpression() <SEMICOLON>
}

void VarDeclarationStatement() :
{}
{
    VariableDeclaration() <SEMICOLON>
}

void VariableDeclaration() :
{}
{
    Type() Identifier()
}

void ExportStatement() :
{}
{
    <EXPORT> <LT> ObjectRepresentation() <GT> <SEMICOLON>
}

void GuardStatement() :
{}
{
    (
        LOOKAHEAD(3)
```

```
        <LCLASP> StateRepresentation() <EQUAL> StateRepresentation() <RCLASP>
        |
        LOOKAHEAD(3)
        <LCLASP> StateRepresentation() <DIFFERENT> StateRepresentation() <RCLASP>
    )
}

void MkrecorderStatement() :
{}
{
    <MKRECORDER> <LT> StateRepresentation() <GT> <SEMICOLON>
}

void PublishStatement() :
{}
{
    <PUBLISH> <LT> StateRepresentation() <COMMA> StateRepresentation() <COMMA>
        ObjectRepresentation() <GT> <SEMICOLON>
}

void StartStatement() :
{}
{
    <START> <LT> ObjectRepresentation() <GT> <SEMICOLON>
}

void ReferenceStatement() :
{}
{
    <REFERENCE> <LT> StateRepresentation() <COMMA> StateRepresentation() <COMMA>
        Identifier() <GT> <SEMICOLON>
}
```

```
void MessageReceptor() :
{}
{
    ( Identifier() | ThisExpression() )
}

void ParametersList() :
{}
{
    PrimaryExpression() ( ParametersRest() )*
}

void ParametersRest() :
{}
{
    <COMMA> PrimaryExpression()
}

void PrimaryExpression() :
{}
{
    ( StateLiteral() | Identifier() | ThisExpression() )
}

void StateRepresentation() :
{}
{
    ( StateLiteral() | Identifier() )
}

void ObjectRepresentation() :
{}
{
```

```
    ( ThisExpression() | Identifier() )
}

void Type() :
{}
{
    (Identifier() | StateType() )
}

void StateType() :
{}
{
    <STATE>
}

void Identifier() :
{}
{
    <IDENTIFIER>
}

void StateLiteral() :
{}
{
    <STATE_LITERAL>
}

void ThisExpression() :
{}
{
    <THIS>
}
```

Apêndice B

Modelo do Estudo de Caso

```
site CasaCliente {
    void main(){
        PainelControle painel;
        painel = new PainelControle();
        ConfiguradorSistema configurador;
        configurador = new ConfiguradorSistema(painel, "123");
        AdicionadorSensor adicionador1;
        adicionador1 = new AdicionadorSensor(painel, "sensor1");
        AdicionadorSensor adicionador2;
        adicionador2 = new AdicionadorSensor(painel, "sensor2");
        AtivadorSistema ativador;
        ativador = new AtivadorSistema(painel, "123");
        DesativadorSistema desativadorSenhaErrada;
        desativadorSenhaErrada = new DesativadorSistema(painel, "1234");
        DesativadorSistema desativador;
        desativador = new DesativadorSistema(painel, "123");
        start <configurador>;
        start <adicionador1>;
        start <adicionador2>;
        start <ativador>;
        start <desativadorSenhaErrada>;
        start <desativador>;
```

```
        return;
    }
}

site EmpresaSeguranca{
    void main(){
        ServicoSeguranca serv;
        serv = new ServicoSeguranca();
        export <serv>;
        mkrecorder <"REGISTRADOR_SERVICOS">;
        publish <"REGISTRADOR_SERVICOS", "SERV_SEG", serv>;
        return;
    }
}

class Alarme {
    State estado;
    Alarme(){
        estado = "ALARME_DESLIGADO";
        return;
    }
    void disparar() {
        estado = "ALARME_DISPARADO";
        return;
    }
    void desligar() {
        estado = "ALARME_DESLIGADO";
        return;
    }
 }

class PainelControle {
    State estaConfigurado;
```

```
State senhaAtual;
GerenciadorOcorrencias gerenciadorOcorrencias;
Alarme alarme;
PainelControle(){
    estaConfigurado = "NAO";
    alarme = new Alarme();
    return;
}
State getAutorizacaoAcesso(State senha) {
    [estaConfigurado == "NAO"]
        return "ERRO";
    [senhaAtual != senha]
        return "ERRO";
    return "OK";
}
State ativarSistema(State senha) {
    State autorizacao;
    autorizacao = this.getAutorizacaoAcesso(senha);
    [autorizacao == "ERRO"]
        return "ERRO";
    gerenciadorOcorrencias.ativar();
    return "OK";
}
State desativarSistema(State senha) {
    State autorizacao;
    autorizacao = this.getAutorizacaoAcesso(senha);
    [autorizacao == "ERRO"]
        return "ERRO";
    gerenciadorOcorrencias.desativar();
    return "OK";
}
State adicionarSensor(Sensor sensor) {
    [estaConfigurado == "NAO"]
```

```
            return "ERRO";
        sensor.setGerenciadorOcorrencias(gerenciadorOcorrencias);
        start <sensor>;
        return "OK";
    }
    synchronized State configurarSistema(State nomeServSeguranca, State senha) {
        [estaConfigurado == "SIM"]
            return "ERRO";
        ServicoSeguranca serv;
        reference <"REGISTRADOR_SERVICOS", nomeServSeguranca, serv>;
        gerenciadorOcorrencias = new GerenciadorOcorrencias(serv, alarme);
        senhaAtual = senha;
        estaConfigurado = "SIM";
        return "OK";
    }
    void desligarAlarme(State senha) {
        State autorizacao;
        autorizacao = this.getAutorizacaoAcesso(senha);
        [autorizacao == "ERRO"]
            return;
        alarme.desligar();
        return;
    }
}

class GerenciadorOcorrencias {
    Alarme alarme;
    State estado;
    ServicoSeguranca servicoSeguranca;
    GerenciadorOcorrencias(ServicoSeguranca servico, Alarme al){
        servicoSeguranca = servico;
        alarme = al;
        estado = "INATIVO";
```

```
        return;
    }
    void registrarOcorrencia(Sensor sensor) {
        [estado == "INATIVO"]
            return;
        alarme.disparar();
        this.acionarServSeguranca(sensor);
        return;
    }
    void acionarServSeguranca(Sensor sensor){
        State idSensor;
        idSensor = sensor.getId();
        servicoSeguranca.tratarOcorrencia(idSensor);
        return;
    }
    void ativar() {
        estado = "ATIVO";
        return;
    }
    void desativar() {
        estado = "INATIVO";
        return;
    }
}

class Sensor {
    State id;
    GerenciadorOcorrencias gerenciadorOcorrencias;
    State estado;
    Sensor(State identificador){
        estado = "SEM_OCORRENCIA";
        id = identificador;
        return;
```

```
    }
    void setGerenciadorOcorrencias(GerenciadorOcorrencias gerenciador){
        gerenciadorOcorrencias = gerenciador;
        return;
    }
    void acusarOcorrencia() {
        estado = "OCORRENCIA_DETECTADA";
        return;
    }
    void verificarAmbiente() {
        [estado != "SEM_OCORRENCIA"]
            gerenciadorOcorrencias.registrarOcorrencia(this);
        return;
    }
    void run() {
        this.verificarAmbiente();
        return;
    }
    State getId() {
        return id;
    }
}

class ServicoSeguranca {
    void tratarOcorrencia(State idSensor) {
        //Acionar viaturas
        return;
    }
 }

//Classes Auxiliares
class AtivadorSistema{
    PainelControle painelControle;
```

```
    State senha;
    AtivadorSistema(PainelControle painel, State senhaSistema){
        painelControle = painel;
        senha = senhaSistema;
        return;
    }
    void run(){
        painelControle.ativarSistema(senha);
        return;
    }
}

class DesativadorSistema{
    PainelControle painelControle;
    State senha;
    DesativadorSistema(PainelControle painel, State senhaSistema){
        painelControle = painel;
        senha = senhaSistema;
        return;
    }
    void run(){
        painelControle.desativarSistema(senha);
        return;
    }
}

class ConfiguradorSistema{
    PainelControle painelControle;
    State senha;
    ConfiguradorSistema(PainelControle painel, State senhaSistema){
        painelControle = painel;
        senha = senhaSistema;
        return;
```

```
    }
    void run(){
        painelControle.configurarSistema("SERV_SEG", senha);
        return;
    }
}

class EstimuladorSensor{
    Sensor sensor;
    EstimuladorSensor(Sensor sensorParaEstimular){
        sensor = sensorParaEstimular;
        return;
    }
    void run(){
        sensor.acusarOcorrencia();
        return;
    }
}

class AdicionadorSensor{
    PainelControle painelControle;
    State idSensor;
    AdicionadorSensor(PainelControle painel, State id){
        painelControle = painel;
        idSensor = id;
        return;
    }
    void run(){
        Sensor sensor;
        sensor = new Sensor(idSensor);
        painelControle.adicionarSensor(sensor);
        EstimuladorSensor es;
        es = new EstimuladorSensor(sensor);
```

```
        start <es>;
        return;
    }
}
```