UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

VERIFICAÇÃO DISTRIBUÍDA DE MODELOS: INVESTIGANDO O USO DE GRADES COMPUTACIONAIS

PAULO EDUARDO E SILVA BARBOSA

CAMPINA GRANDE – PB
FEVEREIRO DE 2007

Verificação Distribuída de Modelos: Investigando o Uso de Grades Computacionais

Paulo Eduardo e Silva Barbosa

Dissertação submetida à Coordenação do Curso 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

Jorge César Abrantes de Figueiredo

Dalton Dario Serey Guerrero

(Orientadores)

Campina Grande, Paraíba, Brasil

©Paulo Eduardo e Silva Barbosa, Fevereiro-2007

B240d

2007 Barbosa, Paulo Eduardo e Silva

Verificação Distribuída de Modelos: Investigando o Uso de Grades Computacionais / Paulo Eduardo e Silva Barbosa. - Campina Grande, 2007.

109f: il..

Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.

Orientadores: Jorge César Abrantes de Figueiredo, Dalton Dario Serey Guerrero.

1. Verificação Formal. 2. Métodos Formais. 3. Lógica Temporal. 4. Grades Computacionais. I. Título.

CDU – 004.4’233:004.415.4

"VERIFICAÇÃO DISTRIBUÍDA DE MODELOS: INVESTIGANDO O USO DE GRADES COMPUTACIONAIS"

PAULO EDUARDO E SILVA BARBOSA

DISSERTAÇÃO APROVADA EM 23.02.2007

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

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinadora

PROF. DAVID BORIS PAUL DÉHARBE, Dr.
Examinador

CAMPINA GRANDE – PB

Agradecimentos

Ao longo desta jornada, interagimos com diversas pessoas, que chegaram e partiram de nossas vidas e deixaram sua contribuição de alguma maneira para que este todo fosse construído. Em resumo, deve ficar registrada a gratidão à toda população deste país que financiou este projeto individual que será empregado no bem coletivo a partir de agora.

Agradeço ao Pai, a fonte de energia para todos os momentos e também ao conforto dado pelos meus pais, irmãos e minha esposa Uyara, que permitiram que eu chegasse até aqui.

Este trabalho não teria se desenvolvido se não fosse a orientação exclusiva dos meus mestres Jorge Figueiredo e Dalton Guerrero, que em conjunto com o parceiro Cássio Rodrigues, transmitiram todo o conhecimento e caráter necessário para a produção de conhecimento para esta nação. A eles dedico cada nova descoberta que esta carreira me propocionar.

Além dos professores, tive a satisfação de trabalhar com alunos que dedicaram vários momentos a semear esta idéia. Sem as presenças de Jairson Cabral, Leandro Max, Talita Uchôa e Fábio Jorge, estes frutos também não teriam brotado para esta colheita.

Sinto-me orgulhoso de também poder agradecer pelo apoio a todos que compõem o GMF. Em especial à minha madrinha Ana Emília e meu atual orientador Franklin Ramalho pelas idéias, correções e momentos de descontração dos últimos anos. Meciono aqui também o carinho pela professora Patrícia Machado pelas contribuições durante minha formação, como também ao professor David Dehárbe pelas considerações relevantes de ambos na avaliação do trabalho. E, enfim, às pessoas que se fizeram presentes como irmãos em nosso grupo: Afrânio, Ana Esther, André, Amâncio, Amanda, Daniel Aguiar, Daniel Barbosa, Elthon, Emanuela, Emerson, Erica, Fabrício, Flávio, Helton, Jaime, João, Laísa, Lile, Neto, Rodrigo, Rogério, Taciano e Wilkerson. Meu muito obrigado!

Resumo

Todo programador ou engenheiro de software lida com um problema crônico na concepção de seus sistemas: violações das especificações ou requisitos de projeto. Essas violações necessitam de uma captura imediata, pois geralmente originam falhas que só podem ser descobertas tardiamente, a um custo de reparo bastante elevado.

Nos últimos anos, pesquisadores da ciência da computação estão conseguindo um progresso notável no desenvolvimento de técnicas e ferramentas que verificam automaticamente requisitos e projeto. A abordagem em maior evidência chama-se verificação de modelos (model-checking). Verificação de modelos é uma técnica formal e algorítmica de se fazer verificação de propriedades de sistemas com um espaço de estados finito. Suas principais vantagens são o poder de automação e a qualidade dos resultados produzidos. Porém, esta técnica sofre de um problema fundamental — *a explosão do espaço de estados* — que se deve ao crescimento exponencial na estrutura que representa o comportamento de sistemas e à falta de recursos computacionais disponíveis para lidar com grandes quantidades de informação sobre o comportamento dos sistemas sob verificação.

Este trabalho concentra-se em verificação de modelos utilizando plataformas de distribuição como tentativa de aliviar o problema citado. Mais detalhadamente, investigamos o uso de grades computacionais que rodam aplicações *bag-of-tasks* e formulamos algoritmos específicos para o processo de verificação. Aplicações bag-of-tasks são aplicações paralelas cujas tarefas são independentes entre si. Elas são as aplicações mais apropriadas para grades computacionais por permitirem heterogeneidade dos recursos. Aplicamos ferramentas de grades computacionais como uma camada entre a ferramenta de verificação e os recursos distribuídos compartilhados existentes e comparamos os quesitos desempenho e escala nos sistemas a serem verificados em relação às versões centralizadas de verificadores.

A plataforma empregada na distribuição é muito atrativa no quesito custo, controle e escala. Através do compartilhamento de uma simples máquina, o engenheiro de sistemas ganha acesso a uma comunidade provedora de uma grande quantidade de recursos heterogêneos e

automaticamente gerenciados para se fazer computação paralela seguindo sua filosofia. Durante o trabalho, essas vantagens são comparadas com suas desvantagens, como o alto custo de comunicação e a dificuldade de particionar o processo, por exemplo.

O trabalho envolveu a produção das seguintes ferramentas: uma API genérica para a geração distribuída de grafos que representam o comportamento de sistemas concorrentes sobre plataformas de grades computacionais *bag-of-tasks*, um protótipo de verificação CTL que age de duas maneiras distintas, sendo *on-the-fly* durante a geração do espaço de estados ou sobre esse espaço de estados distribuído representado explicitamente seguindo a mesma filosofia de comunicação e versões simplificadas de simuladores de sistemas concorrentes sob alguns formalismos baseados em redes de Petri. Resultados experimentais sobre a aplicação deste ferramental são apresentados.

Abstract

Every programmer or software engineer deals with a chronic problem during the conception of their systems: violations in the project requirements. These violations need to be discovered early because they generally produce errors that can be discovered later, at a very expensive cost to repair.

In recent years, researchers in computer science are obtaining a notable progress in the development of techniques and tools to automatically verify requirements and designs. The most evidently approach is called model-checking. Model checking is a formal and algorithmic technique to perform properties verification in a finite state space of systems. Its main advantages are the automation power and the quality of the produced results. However this technique suffers from one big and foundamental problem - *the state space explosion* - which is the absense of computational resources available today's to deal with large amounts of information about the behavior of the systems under verification.

This work investigates a solution to verify models in a distributed way using computational grids which runs *bag-of-tasks* applications, alleviating the mentioned problem. Bag-of-tasks applications are those parallel applications which tasks are independent of each other and are the applications most suited for computational grids because they allow heterogeneity between resources. We employ computational grid tools as a layer between between the verification tool and the distributed shared resources.

This verification is performed by adapted CTL algorithms to the *bag-of-tasks* philosophy. So we intend to obtain improvements in speed-up and scalability in the systems to be checked when compared to centralized versions of verifiers produced inside the group. Moreover, the employed middleware in the distribution is very attractive in the cost and control aspects. By sharing a single machine, the system engineer obtains access to a community that provides large amounts of heterogeneous resources automatically managed to perform parallel computation.

This work included the production of the following tools: a generic API to the distributed generation of graphs that describes the behavior of concurrent systems under *bag-of-tasks* computational grids platforms, a prototype to check for CTL properties using on-the-fly

algorithms or iterating over the generated fragments of the distributed graph. We also implemented simplified versions of simulators of concurrent systems following some formalisms based on Petri nets. Experimental results are also presented.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização

Garantia de Confiabilidade em Sistemas Com o advento da tecnologia da informação, os dispositivos eletro-eletrônicos desempenham um papel cada vez mais fundamental em nossa sociedade da informação. Eles são responsáveis por prover os principais meios de comunicação e estão cada vez mais presentes no auxílio a tarefas do nosso cotidiano, acarretando também um maior grau de dependência em sua confiabilidade. Em essência, esses dispositivos são compostos por componentes de hardware e software, estando sujeitos a diversos tipos de falhas, inclusive originárias da sua concepção, tornando técnicas de validação do correto funcionamento desses componentes, requisitos importantes no seu projeto e desenvolvimento.

As características de concorrência e a reatividade dificultam bastante conseguirmos garantir a confiabilidade total de sistemas. Um sistema é concorrente se os elementos que o compõem são independentes e, mesmo sendo seqüenciais, interagem entre si. Já os sistemas reativos apresentam uma interação contínua com o ambiente no qual estão inseridos. Isso é caracterizado pela interação com o seu ambiente por meio de entradas e saídas produzidas pelo sistema [dSM06]. Quando isso acontece, a dificuldade reside no fato dos seus componentes poderem interagir de muitas formas diferentes [God03], e isto geralmente ocasiona um número exponencial de combinações de estados.

Uma alternativa para melhorarmos os projetos destes sistemas complexos é o uso de verificação formal. Para Katoen [Kat99], verificação formal é uma importante abordagem para

se tentar obter uma garantia de que o sistema funcione conforme o esperado. Ao contrário de testes, que geralmente utiliza implementações, também no geral trabalha-se com modelos e diversas técnicas com uma grande fundamentação matemática e lógica para se provar a corretude de um sistema. Outro fator motivante para a adoção dessa abordagem é que erros detectados antes da codificação real do sistema são mais baratos de serem reparados pela equipe de desenvolvimento.

Exploração de espaços de estados é uma das técnicas aplicadas com maior sucesso no processo de verificação formal de sistemas concorrentes e reativos. Consiste em uma exploração sistemática e automática de um grafo de nós ou estados finito que representa o comportamento completo de todos os componentes concorrentes do sistema. A principal limitação desta técnica reside no tamanho do espaço de estados, pois este é incrementado exponencialmente muitas vezes com um aumento linear no tamanho do modelo do sistema. A enumeração de todos os estados alcançáveis de um sistema de escala industrial facilmente excede as capacidades de memória e processamento de uma máquina simples. Este problema é conhecido como *explosão do espaço de estados*.

Segundo Bérard [BBF+01], Clarke [CGP99] e McMillan [McM93], verificação de modelos pode ser descrita como uma técnica formal, quase totalmente automática e algorítmica de se fazer verificação de propriedades em um espaço de estados finito de sistemas. Sua execução se dá através do uso de uma ferramenta chamada verificador de modelos, que a partir do comportamento do sistema e de uma propriedade expressa em algum formalismo lógico, verifica sistematicamente a validade dessa propriedade. Verificação de modelos é uma abordagem adequada a sistemas reativos, com características de serem inerentemente distribuídos, concorrentes, e acima de tudo complexos. É uma abordagem apropriada para sistemas de hardware e software, e devido a isso recebeu uma boa adoção por parte da indústria e de comunidades científicas [Kat99]. Porém, segundo a comunidade, o principal desafio em verificação de modelos, desde sua origem e ainda não resolvido, também é o de encontrar algoritmos e adequar estruturas de dados para amenizar o problema da *explosão de espaço de estados*.

Entre as principais técnicas desenvolvidas em plataformas centralizadas para que verificadores lidem com o problema da *explosão do espaço de estados*, podemos citar:

- Métodos de representação simbólica, tal como BDDs [McM93] e [JEK+90].

- Métodos que exploram simetria [Sch00], [ES97], [SG04] e [BG05].
- Técnicas de redução de ordem parcial [LLEL02], [NP06] e [HP95].
- Combinação com técnicas de interpretação abstrata [DHJ+01].

Nos últimos anos, soluções paralelas e distribuídas de verificação de modelos vêm sendo estudadas com esse intuito [BBS01; CGN98; GMS01; LS99; OvdPE04; SD97]. Um dos principais congressos da área de verificação automática, o *International Workshops on Parallel and Distributed Methods in Verification - PDMC*, demonstra que o interesse se deve à emergência de ambientes computacionais distribuídos de alta disponibilidade e porque as demais técnicas individualmente não apresentaram até hoje resultados que fossem considerados satisfatórios. Além do mais, um grande desafio para esses ambientes é de se tornarem adequados a vários domínios da ciência da computação [SIE05], justificando o esforço por parte de alguns cientistas e desenvolvedores de que suas infraestruturas de computação atendam satisfatoriamente aos requisitos de verificadores formais distribuídos, que geralmente envolvem ganhos em desempenho, escala nos modelos, baixo custo e outros mais.

Porém, o desenvolvimento desses verificadores distribuídos é uma tarefa bastante complexa. Por um lado, as ferramentas devem endereçar requisitos lógicos relacionados à tarefa de verificar, como implementação de algoritmos dos operadores de lógicas temporais, fornecimento de contra-exemplos e outros mais. De outro lado, aspectos distribuídos como compartilhamento de recursos, desempenho de computação, confiabilidade e comunicação são de extrema importância [Jou03]. Não existe uma ferramenta de verificação distribuída que possa ser aplicada por usuários sem conhecimento aprofundado do domínio. A maioria dos trabalhos utilizam soluções configuráveis, considerando requisitos específicos da distribuição [KP04; CGN98; GMS01; Jou03; CP03; SD97], ou então ficam apenas em um nível muito alto de abstração, sem considerar de forma alguma aspectos práticos e implementações nas tecnologias disponíveis no momento. Necessita-se de trabalhos que contribuam com práticas e implementações de algoritmos em protótipos de verificadores, abordando os problemas mencionados.

Aplicação de Grades Computacionais como Plataforma de Apoio à Verificação Distribuída Neste trabalho, investigamos o uso de *grades computacionais* [FK98] na exploração

distribuída de espaços de estados. Grades computacionais estão se tornando efetivas porque elas oferecem uma forma de resolver esses problemas sendo em geral mais escalável e barata do que outras soluções, tais como clusters e super-computadores. Isto é um grande atrativo para se aplicar em nosso contexto. Segundo o grupo de pesquisa responsável pelo desenvolvimento da grade computacional Nordugrid ($www.dcgc.dk$), em uma de suas conferências, declarou-se que usuários de verificadores de modelos podem se adequar a grades porque não possuem o hardware necessário para uma super-computação paralela ou o software que executa essa tarefa é muito difícil de instalar e manter.

Levando em consideração o desenvolvimento de ferramentas de verificação, um dos principais benefícios de se usar grades computacionais, que não é provido por outras plataformas distribuídas, é que requisitos fundamentais relacionados à distribuição são endereçados de uma maneira transparente, isto é, com pouca intervenção do usuário, agregando o vasto conhecimento e ferramental desenvolvido por pesquisadores na área. Então, os custos e o tempo para desenvolver a ferramenta de verificação podem ser consideravelmente reduzidos.

Considerando do ponto de vista da aplicação, grades computacionais podem aumentar o número de máquinas para a verificação distribuída. Assim, afirmamos que é uma plataforma bastante poderosa no quesito de escala dos recursos disponibilizados. Podemos citar seu uso na biologia computacional [STMBSS98] e o projeto SETI (*Search for Extraterrestrial Intelligence*), que é uma área científica engajada na detecção de vida inteligente fora da Terra [ACK+02]. Por exemplo, a rede privada no qual este trabalho foi desenvolvido tem dezessete máquinas. Ao usarmos a grade computacional **Ourgrid** [our05; CBA+04; CPC+03] - um sistema que trabalha como uma comunidade ponto-a-ponto que permite a doação de recursos disponíveis entre seus membros - é possível contarmos com mais de quinhentas máquinas amplamente distribuídas que são afiliadas à comunidade **Ourgrid**. Isto não significa que esses quinhentos recursos estarão disponíveis todo o tempo, mas é possível usar outras máquinas disponíveis que estejam associadas ao **Ourgrid** e o custo disso é muito baixo; a comunidade é paga com a doação de nossos ciclos ociosos de CPU [ABCM04].

Ao usarmos grades computacionais, não se sabe se o desempenho de computação deva ser incrementado. Grades computacionais são promissoras neste cenário, pois provêem heurísticas de escalonamento que lidam com o desempenho computacional a nível de recursos. Isto permite tirar vantagens de recursos que sejam mais rápidos que outros, deixando o usuá-

rio encarregado apenas de definir um particionamento lógico para seus sistemas. Porém há também alguns limites da tecnologia que serão detalhados no decorrer do trabalho.

O trabalho também contribui ao elucidar que o particionamento lógico requisitado nem sempre é trivial para sistemas concorrentes com essa abordagem, gerando alguns empecilhos quando tentamos conseguir ganhos significativos em desempenho. Além do mais, esse ganho também é atenuado pelo alto custo de comunicação entre tarefas. Concluímos então uma alternativa para elucidar estas expectativas de ganhos e perdas é a avaliação experimental. Esta avaliação deve considerar as características dos modelos a serem verificados, implementação dos algoritmos e características de distribuição. Eis a principal justificativa do trabalho apresentado.

1.2 Declaração do Problema e da Solução

Atualmente, as abordagens existentes de verificação de sistemas complexos tentam lidar com dois problemas principais. O primeiro é conhecido como *explosão do espaço de estados* e é inerente da complexidade dos modelos, pois o número de estados de um modelo tende a crescer exponencialmente com o número de componentes do modelo. Isto leva a uma situação onde o comportamento do modelo não cabe na memória disponível dos recursos atuais, ou o tempo de exploração desses estados se torna inaceitável. O outro problema é inerente à complexidade de métodos de verificação distribuída de modelos: os algoritmos geralmente são propostos apenas como prova de conceito, esquecendo os problemas de projeto que um sistema dessa natureza provê, tecnologias de implementação, otimização e uso prático.

O trabalho desenvolvido nessa dissertação é uma investigação do uso de grades computacionais como plataforma para se fazer verificação distribuída de modelos. O método empregado foi o experimental, fazendo-se necessária a implementação de um protótipo com o núcleo da abordagem para observarmos exaustivamente o comportamento de diversos modelos e fundamentar nossas conclusões.

1.2.1 Contribuições do Trabalho

.

Desenvolvemos os seguintes artefatos para verificação distribuída de modelos na plataforma de grades computacionais:

- Método e protótipo de geração distribuída de espaços de estados de sistemas concorrentes.
- Algoritmos e protótipo de verificação distribuída utilizando lógica temporal CTL de espaços de estados de sistemas concorrentes. CTL justifica-se por ser capaz de expressar em tempo ramificado a maioria das propriedades que nos interessam, e está apta a ser paralelizada.
- Algoritmos e protótipo de verificação distribuída de propriedades no modo *on-the-fly*.
- Reuso de algumas idéias definidas em outros trabalhos sobre particionamento dos grafos que descrevem comportamento desses sistemas. Assim, formulamos nossa própria heurística para obtermos um particionamento adequado.
- Resultados experimentais e classificação dos modelos que dizem se são viáveis ou inviáveis para se obter ganho em desempenho quando aplicamos a nossa solução. No geral, apresentamos uma solução escalável.

1.3 Estrutura da Dissertação

O restante do documento está estruturado da seguinte forma:

Capítulo 2: Fundamentação Teórica Este capítulo divide-se em cinco partes fundamentais. Iniciamos a primeira parte tratando dos sistemas concorrentes e reativos, definindo-os e comentando seu comportamento, pois estes são as principais entradas a serem analisadas por nossa solução. Uma ferramenta bastante usada para modelagem desses sistemas é redes de Petri e suas derivações, que também será tratada neste capítulo, juntamente com a técnica de derivação de seu comportamento, que é chamado de grafo de ocorrência.

A segunda parte trata de grades computacionais, apresentando os fundamentos da plataforma e a visão geral do seu modelo de computação distribuída. Lá, apresentamos os principais ambientes de grades computacionais e o Ourgrid, que é uma grade computacional para tarefas bag-of-tasks.

Na terceira parte, o tópico tratado é a técnica de verificação de modelos, os algoritmos de verificação de propriedades em espaços de estados já gerados seguindo alguma lógica proposicional. É discutido o problema da explosão de espaços de estados, que se deve à falta de memória suficiente dos recursos disponíveis para armazenar todo o comportamento. Apresentamos uma idéia dos principais algoritmos de produção de comportamento de sistemas e formas de armazená-los. As lógicas temporais complementam essa parte. Definimos o que são, como se derivam da lógica proposicional e porque são adequadas para expressar propriedades. O foco é em CTL (Computation Tree Logic). Enfatizamos o uso das lógicas CTL e LTL no contexto de verificação de modelos.

Na seção quatro, temos os trabalhos existentes em verificação distribuída, que é o estado da arte no tema central de nosso trabalho. Existem diversos trabalhos propostos para outras plataformas, seguindo outros algoritmos, outras linguagens de especificação de propriedades e outra variáveis mais. Será apresentada uma visão geral do que já foi desenvolvido para melhor situar o leitor sobre o que são nossas contribuições.

Para concluir, apresentamos o cenário de desenvolvimento do trabalho. Devido ao trabalho estar inserido em um grupo de pesquisa e tentar resolver alguns problemas encontrados em pesquisas anteriores, faz-se necessária uma visão geral das ferramentas que nossos algoritmos atendem e alguma fundamentação em conceitos implementados por elas. Lá, detalhamos o processo da geração centralizada, os principais *trade-offs* e problemas que existiam que permitiam simulação apenas de pequenos modelos. Apresentamos também o verificador de modelos *Veritas*.

Capítulo 3: Verificação Distribuída de Modelos Este capítulo formaliza todas as idéias do nosso trabalho. Primeiramente são apresentadas noções de particionamento e o método que empregamos. Logo após, investigamos a geração distribuída de espaços de estados e discutimos a forma como se dá sobre a plataforma de grades computacionais. Defendemos a generalidade da nossa solução com um modelo abstrato. Apresentamos o mecanismo

de comunicação entre as partições e instanciamos uma implementação. Ainda discutimos os algoritmos de verificação distribuída em CTL que também caracterizam nossa solução. Descrevemos os passos que foram necessários para construirmos esses algoritmos também em um modelo abstrato, mas considerando todas as variáveis envolvidas com a distribuição. Detalhes da implementação e tecnologias usadas são apresentados no final.

Capítulo 4: Resultados Experimentais Comparamos o desempenho das nossas ferramentas distribuídas em relação às versões centralizadas das mesmas e modelamos sistemas de protocolos e padrões de agentes móveis e sistemas clássicos, cada um apresentando uma característica particular. Em seguida, apresentamos a validação através de duas etapas: simulação e verificação de modelos. Classificamos esses modelos de acordo com os benefícios obtidos ao aplicarmos nossa solução.

Capítulo 5: Conclusão Por último, apresentamos nossas conclusões baseando-se nos resultados obtidos com as experimentações. Relatamos as experiências adquiridas que pretendemos que sejam transmitidas para a comunidade científica. Fechamos o trabalho enumerando possíveis desdobramentos como trabalhos futuros.

Capítulo 2

Fundamentação Teórica

O capítulo está dividido em cinco partes fundamentais. A primeira parte é sobre sistemas concorrentes e sobre formalismos usados para descrevê-los ou modelá-los. A compreensão do comportamento desses sistemas faz-se bastante útil ao entendimento da dissertação por possuirem características bastante peculiares em relação a sistemas comuns e que dificultam a garantia total de seu correto funcionamento. A segunda seção é uma visão geral sobre plataformas de grades computacionais existentes e principalmente sobre a plataforma adotada em nossas experimentações. A terceira seção é uma breve apresentação de verificação de modelos. Os algoritmos já existentes para geração de espaços de estados são apresentados e algoritmos de verificação para alguns tipos de lógicas temporais na avaliação de propriedades são brevemente discutidos. A quarta parte também se concentra na verificação de modelos, porém apresentando o estado da arte em soluções distribuídas, que são os trabalhos relacionados ao nosso tema de pesquisa. Na quinta e última parte, apresentamos o que já existia no grupo antes do início do trabalho. Antes já enfrentávamos o problema da explosão do espaço de estados com algumas ferramentas para verificação desenvolvidas por nós e que puderam ser re-adaptadas. Lá, detalhamos esses trabalhos.

2.1 Sistemas Concorrentes

Concorrência está relacionada ao fato de algum sistema empregar diversos agentes independentes entre si para computação, compartilharem recursos e existir uma infra-estrutura para a comunicação entre esses agentes. Esses sistemas são chamados de sistemas concorrentes e

este conceito abrange várias arquiteturas computacionais. Podemos mencionar que vai desde arquiteturas fortemente acopladas, a maioria dos sistemas paralelos síncronos até sistemas assíncronos, fracamente acoplados e largamente distribuídos.

A corretude de sistemas concorrentes é intrinsecamente mais difícil de analisar do que sequenciais. Um dos motivos é a dificuldade de expressar as interações dos agentes ou componentes do sistema, gerando questões como *deadlock*, *condições de corrida*, *vivacidade* e outros mais. Além do mais, a maioria dos sistemas concorrentes interagem bastante com seus ambientes, sendo reativos. Assim, esses sistemas geralmente não terminam, ficam eternamente esperando por estímulos do ambiente, o que dificulta verificar corretude através de conjuntos de entradas e saídas.

Entre as principais teorias desenvolvidas para modelar formalmente esses sistemas [mck02], podemos citar:

- **Autômatos** Modela os sistemas em termos de estados e transições entre estados. Formalismos como redes de Petri, álgebras de processos, autômatos de *I/O* derivam-se desse paradigma.

- **Ações** Modela sistemas via estruturas consistindo de ações atômicas. Geralmente são construídos caminhos de execuções que serão comparados com árvores de aceitações e de refutação.

Além do mais, esses modelos de concorrência podem assumir características diferentes em três categorias [Nic87]:

- **Descrições intensionais contra extensionais** Modelos intensionais requerem descrições explícitas dos estados do sistema e mudanças que ocorrem entre eles. Modelos extensionais concentram-se em padrões do que ocorre.

- **Entrelaçamento e concorrência real** Em modelos com entrelaçamento, as observações do comportamento do sistema são sequenciais, tornando assim a execução paralela de ações determinística. Na concorrência real, o comportamento é representado em termos das relações causais entre os eventos executados pelos componentes dos estados distribuídos.

- **Tempo linear contra tempo ramificado** A diferença está nas escolhas que os sistemas terão que fazer durante sua execução. No tempo ramificado, pontos de escolha durante a execução são considerados importantes, enquanto no tempo linear não são.

O nosso interesse aqui em sistemas concorrentes reside na tentativa de garantia de sua corretude. Para isso, existe a *especificação*, que se divide no uso de lógicas e de relações comportamentais para relatar especificações e implementações, e *verificação*, que é uma maneira de checar se um sistema satisfaz uma especificação dada pra ele. Maiores detalhes sobre estas técnicas serão apresentados em seções posteriores.

2.1.1 Redes de Petri

Redes de Petri é uma ferramenta gráfica e matemática para modelagem de sistemas. Devido às suas características, as redes de Petri são fortemente recomendadas para modelagem de sistemas com um alto índice de paralelismo ou concorrência, revelando informações importantes sobre a estrutura e o comportamento dinâmico do sistema modelado.

Definition 2.1.1 *Uma Rede de Petri R= ($\mathcal{P}$, $\mathcal{T}$, I, O) é definida por:*

- *Um conjunto finito $\mathcal{P}$ de lugares ("places");*
- *Um conjunto finito $\mathcal{T}$ de transições;*
- *Uma função de entrada I: $\mathcal{T} \rightarrow 2^{\mathcal{P}}$;*
- *Uma função de saída O: $\mathcal{T} \rightarrow 2^{\mathcal{P}}$.*

Graficamente, lugares são representados por círculos e as transições por barras ou retângulos. Uma rede de Petri pode ser representada por um grafo dirigido bipartido G = (V,E), com $V = \mathcal{P} \cup \mathcal{T}$ e $\mathcal{P} \cap \mathcal{T} = \emptyset$. Qualquer aresta e em E é incidente em um membro de $\mathcal{P}$ e um membro de $\mathcal{T}$.

Seja μ uma função μ: $\mathcal{P} \rightarrow \mathbb{N}$ que mapeia o conjunto de lugares $\mathcal{P}$ em números inteiros não negativos. Cada lugar armazena uma certa quantidade de fichas que é dada por esse número inteiro. Denomina-se marcação numa rede de Petri a tupla M = $\langle$ $\mu(p_1)$, $\mu(p_2)$, ... ,$\mu(p_L)$ $\rangle$, com $\mu(p_i) \in \mathbb{N}$ e L = número de lugares.

Figura 2.1: Exemplo de uma rede de Petri

Uma *rede de Petri marcada* é RM = (R, M), sendo que pelo menos a marcação de um lugar dessa rede seja diferente de zero, ou seja, $\mu(pi) \neq 0$. Na sua representação gráfica, as marcas são representadas por pontos no lugares.

Cada rede possui uma configuração inicial que corresponde à descrição do sistema. Com relação à dinâmica das redes de Petri, dizemos que uma transição está *habilitada* quando todos os seus lugares de entrada estiverem marcados com um número maior ou igual de fichas requisitados pelos arcos. Uma transição *dispara* sempre que estiver habilitada e ocorrer o evento a ela associado. O disparo de uma transição resulta em uma nova marcação da rede, retirando uma marca de cada lugar de entrada da transição que disparou e colocando em cada lugar de saída.

2.1.2 Espaços de Estados

Exploração de espaço de estados é um dos principais meios para a verificação completa do software. Para verificar um modelo, precisamos simulá-lo seguindo todas as suas possibilidades de execução. A simulação de apenas alguns poucos cenários não cobre todas as possibilidades de estados do modelo de um sistema [Sil04].

O espaço de estados de um modelo nada mais é do que o conjunto de todos os estados possíveis que um sistema pode assumir e as transições entre esses estados. Em formalismos baseados em redes de Petri, o espaço de estados corresponde ao conjunto de todas as configurações alcançáveis (possibilitado pelo disparo de um número finito de transições) a partir da configuração inicial, considerando a ocorrência de todos os possíveis eventos. Neste contexto ele é freqüentemente denominado por grafo de alcançabilidade ou grafo de ocorrência

Como exemplo, considere uma instância do modelo do jantar dos filósofos de Dijkstra com dois filósofos e dois garfos. Este modelo está descrito em Redes de Petri Orientadas a Objetos. Uma representação abstrata do espaço de estados se encontra na Figura 2.2 com a rede que descreve cada objeto sendo ocultada. Para esta instância do problema e, considerando a configuração inicial representada na figura pela configuração 1, é possível se chegar a 9 configurações diferentes.

Figura 2.2: Representação gráfica do espaço de estados do jantar dos filósofos

Algoritmos para a geração desse grafo basicamente se resumem à busca em largura e em profundidade. Geralmente o que diferencia o projeto desses algoritmos são os métodos utilizados para a compactação desse grafo durante a geração e também a plataforma empregada para executar esses algoritmos.

2.2 Grades Computacionais

Grades computacionais são um modelo de computação em evidência que fornece um grande poder computacional tirando-se vantagem de múltiplos computadores em rede que modelam uma arquitetura computacional virtual. Esta arquitetura é capaz de distribuir a execução dos processos através de uma infra-estrutura paralela. Ela faz uso de recursos ociosos, tais como ciclos de CPU ou armazenamento em disco, de grandes números de computadores separados geograficamente. O foco da tecnologia de grades é na habilidade de suportar computação através de diferentes conjuntos de domínios administrativos [VCT05]. Esta abordagem difere bastante da computação distribuída tradicional ou de como se faz processamento em clusters.

Grades computacionais motivam o uso ótimo dos recursos tecnológicos dentro de uma organização. O interessante é que pode ser provido um serviço de computação, assim como eletricidade ou água, onde o usuário paga apenas pelo que consumiu. Porém, atualmente várias comunidades provedoras de recursos ainda são de livre adesão.

A computação em grades envolve o compartilhamento de recursos, como diferentes plataformas, arquiteturas de hardware e software e linguagens de programação. Estes recursos estão localizados em lugares diferentes pertencendo a domínios administrativos diferentes sobre uma rede usando padrões de conexão consolidados [FK98].

Deve ficar claro aqui, que existem várias denominações e usos para grades computacionais. Podemos classificar o termo grades das seguintes formas [FKT01]:

- Grades computacionais. O foco principal está em operações computacionais massivas.
- Grades de dados. É o controle e gerência de grandes quantidades de dados distribuídos.
- Grades de equipamentos. É quando algum equipamento complexo, como por exemplo um telescópio, é controlado e analisado remotamente por uma grade de recursos computacionais.

Computação em grades reflete mais em um framework conceitual do que em um recurso físico. A abordagem de grades é utilizada quando queremos resolver uma tarefa computacional em diversos recursos distantes administrativamente. O foco da tecnologia de grades

está associado com os requisitos de fornecimento de recursos computacionais flexíveis além do seu domínio administrativo local.

2.2.1 Desempenho de Grades Computacionais

Uma característica que atualmente distingue a computação em grades da computação distribuída comum é a abstração de um *recurso distribuído* em um *recurso da grade* [MDS+05]. Uma característica dessa abstração é que ela tolera de forma mais estável a substituição de recursos. Porém, esta flexibilidade reflete em um custo de comunicação na camada de middleware e na latência temporal associada ao acesso dos recursos distribuídos na grade. Este custo, especialmente a latência temporal, devem ser avaliados em termos do desempenho computacional quando um recurso da grade é empregado.

2.2.2 Principais Ambientes de Grades Computacionais

Globus Toolkit

Consiste em um conjunto de serviços que facilitam a construção de infra-estruturas para a computação em grades [FK97]. Os serviços Globus são usados para submissão e controle de aplicações, descoberta de recursos, movimentação de dados, segurança na grade e inúmeros outros. Esses serviços são razoavelmente independentes, permitindo que se use apenas parte dos serviços em uma dada solução. Esse fato de ser um conjunto de serviços não faz do Globus uma solução pronta e completa (*plug-and-play*) para a construção de grades. Desenvolvedores, administradores e usuários precisam despender certo esforço para finalizar sua grade. Isto se deve à computação em grades ser muito complexa para possibilitar soluções *plug-and-play*, e geralmente são criadas falsas expectativas com relação ao uso dessa plataforma [CPC+03].

Esta plataforma é a principal responsável pela atual convergência que está acontecendo entre grades computacionais e web services. Isto se deve à introdução do padrão OGSA que promove maior interoperabilidade.

Condor

É um sistema que possui um escopo bem definido e menos genérico que outras soluções de alto desempenho em grades computacionais. Utiliza recursos ociosos da rede, objetivando assim alta vazão e não alto desempenho. Ambos usuários e donos da máquina são representados no sistema por agentes de software e esses agentes interagem diretamente na execução da tarefa depois do casamento de padrão entre tarefa, requisitos da tarefa e atributos da máquina. O Condor utiliza mecanismo de *checkpoint* das tarefas, que permite que tarefas sejam reexecutadas em outras máquinas a partir do ponto que parou. As versões mais atuais do Condor adotam visões mais heterogêneas de grades, chegando a utilizar recursos até via Globus.

2.2.3 Uma Grade Computacional para Aplicações Bag-Of-Tasks

O conjunto de ferramentas relacionado à distribuição em plataformas de grades computacionais empregado em nossa solução chama-se Ourgrid. Esta sub-seção é destinada a descrever esse sistema enfatizando apenas as características básicas que são relevantes para o entendimento do trabalho desenvolvido. Uma descrição mais detalhada sobre grades computacionais está fora do escopo dessa dissertação.

Ourgrid trabalha em uma comunidade peer-to-peer [CPC+03] que permite a doação de recursos disponíveis entre seus membros. Ao rodar uma aplicação na grade, o usuário pode tirar vantagem de recursos que estejam disponíveis em sua rede privada e também de recursos que pertençam a outros membros da comunidade. A visão geral do sistema Ourgrid é descrita na Figura 2.3.

Cada membro acessa a comunidade Ourgrid através do sistema MyGrid. Este sistema é responsável por coordenar a execução de aplicações bag-of-tasks sobre todos os recursos que estejam disponíveis. Aplicações bag-of-tasks são aplicações independentes e que não é permitida a comunicação entre elas. Elas são o tipo de aplicações mais apropriadas para se obter garantia de disponibilidade de uma ampla gama de recursos completamente heterogêneos. Essa característica da aplicação permite escalonar de maneira eficiente sem necessidade de informações sobre o desempenho dos recursos.

Resumindo, a atividade de coordenar inclui o escalonamento de tarefas, implantação e

Figura 2.3: Visão Geral do Sistema Ourgrid

transferência de dados. A máquina no qual o MyGrid roda é chamada *máquina local (home machine)*. As máquinas distribuídas no qual as tarefas são executadas são as *máquinas da grade (grid machines)*. O *Par (Peer)* do OurGrid é o sistema que organiza e provê máquinas da grade que pertençam ao mesmo domínio administrativo. Assim, o Par do OurGrid é um serviço de rede que provê dinâmicamente máquinas da grade para a execução de tarefas.

A solução Ourgrid contrasta com outras soluções. Ela se destaca por ser útil e eficiente, embora execute tarefas de escopo limitado (bag-of-tasks), mas assim consegue abstrair todos os detalhes a nível de recursos.

No intuito de melhorar o desempenho das aplicações que executam na grade, o MyGrid implementa algoritmos de escalonamento eficientes que não requerem a intervenção do usuário. Aqui, escalonar significa atribuir tarefas que compõem a aplicação para máquinas que compõem a grade [CBA+04]. Os algoritmos de escalonamento assumem duas coisas sobre as tarefas:

1. Que cada tarefa é executada em três fases sequenciais: inicial, remota e final. Durante a fase inicial, o ambiente da tarefa é configurado durante a transferência dos dados de entrada para a máquina da grade. Durante a fase remota, a tarefa roda na máquina da grade. A fase final é complementar à fase inicial, pois os resultados produzidos são retornados à máquina local.

2. As tarefas têm que ser idempotentes. Isto significa que deve ser permitido executá-las várias vezes sem efeitos colaterais. Esta suposição torna possível recuperar as tarefas de problemas nas máquinas remotas e escalonar tarefas eficientemente através da criação de múltiplas réplicas da mesma tarefa. Porém, a idempotência deve ser garantida pelo usuário.

A criação de múltiplas réplicas de uma mesma tarefa é útil para garantir que alguma delas irá concluir a computação. Além do mais, considerando-se que a grade é composta de recursos heterogêneos, múltiplas réplicas executando ao mesmo tempo em diferentes máquinas podem melhorar significantemente o desempenho de computação. A réplica que alcançar a fase final tem seus resultados considerados e as demais réplicas dessa tarefa são abortadas.

2.3 Verificação de Modelos

Verificação de modelos é um método de verificar automaticamente sistemas formais. O método é aplicado através de uma ferramenta chamada *verificador de modelos*. As partes fundamentais de seu processo podem ser visualizadas na Figura 2.4.

Figura 2.4: O processo de verificação de modelos

Inicialmente é descrito um modelo formal do sistema e automaticamente é gerada uma representação de todo o seu possível comportamento. Essa estrutura que contém esse com-

portamento geralmente é chamada *espaço de estados* ou *grafo de ocorrência*. Em seguida, são especificadas formalmente todas as propriedades que despertem interesse em investigar sua veracidade. Estas especificações geralmente são expressas em uma lógica temporal proposicional. A ferramenta de verificação executa o processo algoritmicamente e produz um valor verdade como resultado que indique se a especificação foi satisfeita ou não. Caso esta especificação não tenha sido satisfeita, a ferramenta deve fornecer uma lista de estados consecutivos que puderam ser alcançados e que demonstram que a especificação não foi válida nesse modelo. Essa seqüência de estados é chamada de *contra-exemplo*.

O sistema de provas durante o processo é o seguinte:

Φ **S** Ψ

Onde Φ é uma pré-condição, **S** uma expressão do programa e Ψ uma pós-condição.

Dizemos que uma fórmula é correta totalmente se toda computação de **S** que começa em um estado satisfazendo Φ termina em um estado satisfazendo Ψ e que ela é correta parcialmente se qualquer computação que começa em um estado satisfazendo Φ que termine satisfizer Ψ.

Com a introdução do paralelismo chegamos ao não-determinismo, pois sistemas paralelos podem compartilhar variáveis e então seus resultados irão depender da ordem em que as variáveis forem acessadas. Não é suficiente conhecer apenas propriedades dos seus estados iniciais e finais, é necessário verificar também expressões que dizem o que ocorre durante a computação [Kat99]. Para resolver esse problema utilizamos lógica temporal. Lógica temporal refere-se à corretude de comportamentos de sistemas sobre o tempo e não apenas como uma relação de entrada/saída.

Verificação de modelos pode ser aplicada a sistemas reativos, que apresentam uma interação contínua com o ambiente no qual estão inseridos. Eles estão cada vez mais presentes em nosso cotidiano e como exemplos citamos sistemas operacionais, protocolos de redes, micro-controladores e diversos softwares. Estes sistemas geralmente recebem estímulos do ambiente e reagem às entradas recebidas. Podemos afirmar que eles são mais complexos, e apresentam características tais como distribuição, concorrência e não possuem um término de execução. Estas características requerem que as propriedades destes sistemas sejam definidas não apenas em função de valores de entrada e saída, mas também em relação à *ordem* em que os eventos ocorrem.

Lógicas temporais são usadas para prover uma descrição precisa das propriedades temporais desses sistemas. Lógicas temporais suportam formulações do comportamento do sistema ao longo do tempo, porque são capazes de expressar relações de ordem, sem recorrer à noção explícita de tempo propriamente dita.

Estudamos duas interpretações que consideram diferentes relações de mudança nos sistemas e são as mais tradicionalmente empregadas no contexto de desenvolvimento de técnicas e ferramentas:

1. LTL: Lógica temporal linear [Pnu77]. As expressões de propriedades de execução ocorrem de maneira sequencial.

2. CTL: Lógica temporal ramificada [Sif90]. Há expressões de propriedades sobre possíveis sequências de execuções a partir de um estado.

2.3.1 Explosão do Espaço de Estados

Embora algoritmos de verificação tenham complexidade temporal linear ao tamanho do comportamento dos modelos, o tamanho da estrutura que armazena esse comportamento, ou seja, o espaço de estados do sistema geralmente é exponencial em relação ao tamanho da descrição do programa [Nam98]. Registrar todos os comportamentos possíveis de um sistema complexo pode esgotar os recursos de memória de uma máquina, mesmo que o número de estados alcançados pelo sistema seja finito. O principal desafio ao emprego da técnica de verificação de modelos nos sistemas atuais é o de encontrar algoritmos e adequar estruturas para lidar com esse problema, que é chamado de *explosão do espaço de estados*.

Atualmente, as melhores implementações de verificadores podem lidar com programas de apenas algumas centenas de variáveis, o que inviabiliza a adoção desta técnica pela indústria. Muitos trabalhos de pesquisa têm sido realizados com o objetivo de aliviar esse problema e há, atualmente, um número considerável de técnicas para tratar deste problema. Por exemplo, se tomarmos o desenvolvimento de sistemas de hardware, a técnica para a representação simbólica do espaço de estados, permitiu a aplicação de verificação de modelos alguns protocolos IEEE [EOH$^+$93]. Quando tratamos com o desenvolvimento de sistemas de software, este problema torna-se crônico por eles serem intrinsecamente mais complexos e muitos possuem várias características de assincronia. Existem trabalhos que tentam lidar com

esse problema. Entre eles está a introdução de técnicas de interpretação abstrata [DHJ+01] e a proposta de reduções de ordem parcial nos grafos que descrevem comportamento [GPS96; Val98].

2.3.2 Lógica Temporal Linear - LTL

Primeiramente definimos sua sintaxe. Como ponto inicial temos um conjunto chamado AP de proposições atômicas. AP possui as proposições mais básicas que podem ser ditas sobre o sistema sob verificação. Estas proposições mais básicas são sentenças que dizem verdades sobre o sistema, sendo interpretada sua veracidade. Como exemplos podemos expressar se uma variável é igual a zero, ou se o *filósofo n* está comendo no modelo dos filósofos de Dijkstra ou ainda se todos os pacotes foram entregues ao destinatário.

A sintaxe de LTL em BNF é definida da seguinte forma: para p pertencente a AP, o conjunto de fórmulas LTL é definido por:

$\Phi ::= p \mid \sim \Phi \mid \Phi \vee \Phi \mid \mathrm{X}\ \Phi \mid \Phi\ \mathrm{U}\ \Phi$

Por essa lógica temporal ser interpretada formalmente em uma sequência de estados, X ϕ significa que a fórmula ϕ deve valer no próximo estado da seqüência e ϕ U ψ significa que ϕ deve valer até que ψ seja verdadeira. Temos também operadores herdados da lógica proposicional, logo precisam apenas do estado corrente para ser detectada sua veracidade.

Definimos os operadores booleanos $\wedge$ (conjunção), $\Rightarrow$ (implicação) e $\Leftrightarrow$ (equivalência) da seguinte forma:

$\Phi \wedge \Psi \equiv \sim(\sim\Phi \vee \sim\Psi)$

$\Phi \Rightarrow \Psi \equiv \sim \Phi \vee \Psi$

$\Phi \Leftrightarrow \Psi \equiv (\Phi \Rightarrow \Psi) \wedge (\Psi \Rightarrow \Phi)$

A lógica temporal linear complementa-se nos seguintes operadores: G (“sempre” ou “globalmente”) e F (“futuramente” ou “eventualmente”), que são operadores sobre caminhos. Gϕ significa que ϕ deve ser verdadeira no estado atual e em todos estados alcançáveis partindo deste estado. Fϕ significa que ϕ deve ser verdadeira no estado atual ou em algum estado futuro na exploração. Eles são definidos da seguinte forma:

$\mathrm{F}\Phi \equiv \text{true U } \Phi$

$\mathrm{G}\Phi \equiv \sim\mathrm{F}\sim\Phi$

Como *true* é válido em todos os estados, FΦ denota que Φ valerá em algum estado no futuro. E GΦ que não haverá nenhum estado no qual $\sim\Phi$ valerá no futuro.

Para a semântica de LTL, dizemos que o significado formal de suas fórmulas é definido em termos de um modelo.

Um modelo PLTL é uma tripla M = (S, R, Label) onde S é um conjunto não-vazio de estados. R: S $\rightarrow$ S atribui a cada $s \in$ S um único sucessor e Label: S $\rightarrow 2^{AP}$ associa a cada estado s o conjunto das proposições atômicas que são válidas naquele estado.

Para definirmos a semântica LTL, considere p $\in$ AP, M = (S, R, Label) um modelo LTL, $s \in$ S, e Φ, Ψ fórmulas LTL. A relação de satisfação $\models$ é definida por:

$s \models$ p se e somente se p $\in$ Label(s)

$s \models \sim \Phi$ se e somente se não ocorre ($s \models \Phi$)

$s \models \Phi \vee \Psi$ se e somente se ($s \models \Phi$) ou ($s \models \Psi$)

$s \models$ XΦ se e somente se R(s) $\models \Phi$

$s \models \Phi$ U Ψ se e somente se $\exists\, j \geq 0.\ R_j(s) \models \Psi \wedge (\forall\, 0 \leq k < j.\ R_k(s) \models \Phi)$.

Se R(s) = *s'*, o estado *s'* é chamado um sucessor de s. Se M,$s \models \Phi$, nós dizemos que o modelo M satisfaz Φ no estado s.

A interpretação formal dos outros conectivos, como true, false, $\wedge$, $\Rightarrow$, G e F podem ser obtidos com derivações a partir dessa definição.

Agora podemos definir formalmente o problema de verificação de modelos: dado um modelo finito M, um estado s e uma propriedade Φ, temos M,$s \models \Phi$?

Então com a definição dessa lógica, estamos aptos a definir e especificar propriedades em LTL para nossos sistemas. Podemos ditar seu comportamento no decorrer do tempo de uma maneira clara e eficiente, como comprovam algumas implementações [Hol97].

Verificação de Modelos em LTL

Em LTL, quando tratamos do problema de verificação de modelos, especificamos da seguinte forma:

Definição 2.1 (Verificação de modelos em LTL) *Seja o modelo $\mathcal{M}$ como uma estrutura de Kripke $\mathcal{M} = (S, I, R, Label)$ formal e uma fórmula nessa lógica temporal chamada ϕ, te-*

mos:

$$\mathcal{M} \models \phi \; se \; e \; somente \; se \; \forall s \in I, (\forall \sigma \mid \sigma \; \in \; Caminhos(s), \sigma \models \phi)$$

Explicando melhor, podemos afirmar que a propriedade ϕ é satisfeita no modelo $\mathcal{M}$ se e somente se ϕ for verídica em todos os caminhos cuja origem sejam os estados iniciais. Isto significa dizer que a propriedade precisa necessariamente ser satisfeita em todas as execuções do sistema.

Geralmente, quando aplicamos a técnica de verificação de modelos em LTL, as propriedades especificadas são traduzidas ou compiladas como um autômato de *Büchi*. Este autômato é finito, mas aceita palavras infinitas, sendo adequado à validação de propriedades de sistemas reativos em que os seus espaços de estados sejam finitos. Dessa forma, o problema é transferido para o domínio da teoria dos autômatos. Em [Kat99], é descrito o produto síncrono entre autômatos de palavras infinitas, que é o procedimento principal da verificação, mas que os detalhes fogem do escopo deste trabalho.

2.3.3 Lógica Temporal Ramificada - CTL

Vimos que em LTL o tempo é linear: em cada momento do tempo existe apenas um possível estado sucessor e assim apenas um futuro possível. Os operadores temporais X, U, F e G descrevem eventos ao longo de um caminho simples como uma computação simples do sistema.

Lógicas Temporais Ramificadas são baseadas formalmente em modelos onde cada momento pode ter diferentes futuros possíveis. Assim *R(s)* é um conjunto não vazio de estados em vez de um simples estado como em LTL. Fica fácil imaginar semanticamente a lógica CTL como uma árvore ao invés de uma seqüência. Nossas fórmulas temporais agora serão precedidas por quantificadores de caminho.

Na definição da sintaxe, semelhantemente a LTL, o conjunto de proposições atômicas é denotado por AP. Sua sintaxe em BNF é:

Para p $\in$ AP o conjunto de fórmulas CTL é definido por:

Φ ::= p | $\sim \Phi$ | $\Phi \vee \Phi$ | EXΦ | E[ΦUΦ]

Quanto à sua semântica, um modelo CTL é visto como uma tripla M = (S, R, Label), também chamado de estrutura Kripke, onde S é um conjunto não-vazio de estados, R $\subseteq$ S

x S é uma relação total em S, que leva cada s $\subseteq$ S em seus possíveis estados sucessores e Label: S $\longrightarrow 2^{AP}$ atribui a cada estado *s* $\subseteq$ S um conjunto de proposições atômicas válidas em *s*.

Um caminho é uma seqüência infinita de estados $s_0 s_1 s_2$..., tal que (s_i, s_{i+1}) $\in$ R para todo i $\geq$ 0 e o conjunto de caminhos começando em um estado s do modelo M é definido por $P_M(s)$ = {σ[0] = s}.

Seja p $\in$ AP uma proposição atômica, M = (S, R, Label) um modelo CTL, *s* $\in$ S e Φ, Ψ fórmulas CTL. Definimos nossa relação de satisfação $\models$ como:

s $\models$ p se e somente se p $\in$ Label(*s*)

s $\models \sim\Phi$ se e somente se não acontece (*s* $\models \Phi$)

s $\models \Phi \vee \Psi$ se e somente se (*s* $\models \Phi$) ou (*s* $\models \Psi$)

s $\models$ EXΦ se e somente se $\exists\sigma \in P_M(s).\sigma[1] \models \Phi$

s $\models$ E[ΦUΨ] se e somente se $\exists\sigma \in P_M(s).(\exists$j $\geq 0.\sigma$[j] $\models \Psi$ e ($\forall 0 \leq$ k $<$ j.σ[k] $\models \Phi$))

s $\models$ A[ΦUΨ] se e somente se $\forall\sigma \in P_M(s).(\exists$j $\geq 0.\sigma$[j] $\models \Psi$ e ($\forall 0 \leq$ k $<$ j.σ[k] $\models \Phi$))

Quanto à expressividade, LTL e CTL são incomparáveis, mesmo se considerarmos fórmulas LTL (de estado) como fórmulas de caminho (Φ ::= AΨ) [CGP99]. Este fato não influenciou na descrição das propriedades que foram necessárias para as experimentações do trabalho.

Verificação de Modelos em CTL

Queremos determinar se uma fórmula CTL Φ é válida no modelo CTL finito M = (S, R, Label). O algoritmo que verifica isso é baseado em nomear cada estado s $\in$ S com as sub-fórmulas de Φ que são válidas em s. Esse conjunto de sub-fórmulas é denotado por Sub(Φ) e sua definição indutiva junto com o algoritmo pode ser encontrada em [Kat99].

Definimos que uma fórmula Φ é satisfeita no estado *s* do modelo M se e somente se s está rotulado com Φ. Não há problemas em verificar para subfórmulas como proposições atômicas ou lógica proposicional. Os problemas maiores estão na satisfatibilidade de E[ΦUΨ] e A[ΦUΨ] e para resolver esses problemas, escrevemos as funções específicas Sat_{EU} e Sat_{AU} que estão descritos em maiores detalhes no livro [CGP99].

A corretude desses algoritmos é baseada na computação de pontos fixos. A idéia é caracterizar a fórmula como o maior ou menor ponto fixo de uma função e aplicar um algoritmo

iterativo para computar tais pontos fixos. Para isso, definimos um reticulado completo de fórmulas CTL e funções monotônicas tal que E[ΦUΨ] e A[ΦUΨ] sejam caracterizados como menor e maior pontos fixos dessas funções.

Em computações de pontos fixos, precisamos caracterizar a relação de precedência que rege os elementos da função. Neste caso, temos a relação de continência entre os conjuntos de estados que satisfazem a fórmula que serão marcados pelo algoritmo. Um estado do algoritmo precede outro se sua estrutura de dados que armazena os estados encontrados que satifazem a fórmula temporal está contida na do outro estado.

Para uma dada fórmula, tomamos o conjunto de estados que satisfazem essa fórmula e a relação de precedência fica definida como a relação de continência entre conjuntos.

Então pela axiomatização de CTL podemos definir funções que tenham E[ΦUΨ] e A[ΦUΨ] como pontos fixos e iterativamente iremos alcançar o conjunto de estados que satisfaçam a fórmula desejada.

2.4 Verificação Distribuída de Modelos

A maioria dos trabalhos propostos para geração e verificação distribuída de espaços de estados possuem o mesmo princípio básico: cada processador na rede é responsável por uma parte específica do espaço de estados inteiro, de acordo com uma função hash que mapeia estados em processadores. As principais diferenças entre eles reside no benefício esperado ao usarmos uma solução distribuída, o middleware adotado para a distribuição e a estratégia específica para se particionar o espaço de estados.

Os benefícios de se usar uma solução distribuída são ganho em desempenho na execução do trabalho e o uso da memória distribuída com o objetivo de gerar e verificar espaços de estados maiores. A versão paralela da ferramenta Murϕ[SD97], por exemplo, foi desenvolvida porque o cálculo do próximo estado durante uma geração era muito complexo, e a geração do espaço de estados demandava várias horas com os recursos que o grupo de pesquisa tinha disponíveis. Porém, experimentos mostraram que a ferramenta distribuída roda com um ganho de desempenho muito próximo da linearidade quando utiliza-se uma ampla gama de processadores de memória distribuída e redes de estações de trabalho. Em contrapartida, o trabalho de Lerda [LS99] lida com essa limitação de memória física de uma máquina cen-

tralizada, explorando a execução em um ambiente de memória distribuída, tais como redes locais comuns, para aumentar o tamanho dos problemas a serem verificados com sucesso no SPIN. Este trabalho apresenta vários resultados experimentais na verificação de propriedades de segurança (safety properties). É a chamada versão paralela do SPIN.

Bourahla em um trabalho recente [Bou05] apresenta um método astuto de melhorar o desempenho de verificação de modelos usando técnicas de paralelização também em um ambiente de memória distribuída. É o trabalho mais focado em um equilíbrio efetivo e na redução na comunicação entre as partições, sendo inclusive a aplicado a grandes sistemas industriais. O autor considera o espaço de estados como uma estrutura de Kripke ponderada e aplica uma série de abstrações e refinamentos. É um trabalho bastante valoroso na combinação de diversas abordagens de verificação e apresenta resultados experimentais bastante convincentes.

Levando-se em consideração o middleware distribuído, conforme mencionado em [Jou03], a maioria dos trabalhos utilizam apenas definições abstratas das primitivas de comunicação, tais como `SEND` e `RECEIVE`. Assim, a implementação dessas primitivas de comunicação é feita sobre sistemas de passagens de mensagens básicos, tais como Active Message ou MPI. A abordagem proposta nessa dissertação não faz uso desses sistemas diretamente. Aqui, a ferramenta de verificação interage diretamente com a grade computacional. Dessa forma, apresentamos uma solução mais completa porque capturamos uma série de aspectos importantes não tratados por sistemas de passagens de mensagens, tais como escalonamento e alocação ótima dos recursos, por exemplo.

A estratégia de particionamento dos estados usada nesse trabalho segue o mesmo princípio adotado em [CGN98; KP04]. Estes trabalhos usam funções hash no qual os parâmetros são um pequeno subconjunto de lugares mantenedores de dados (veja a Seção 3 para maiores informações).

2.5 O Ambiente de Desenvolvimento do Trabalho

Esta seção explica em detalhes o cenário existente no grupo antes de nossas experimentações. Faz-se necessário esse entendimento para uma argumentação mais concisa sobre os algoritmos que propomos na seção seguinte. Além do mais, participei diretamente da imple-

mentação das ferramentas e protótipos apresentados aqui, o que facilitou bastante estarmos sempre configurando e otimizando essas soluções para conseguirmos desempenhos melhores.

2.5.1 Geração Centralizada do Espaço de Estados

Obter o espaço de estados de um sistema consiste em gerar um grafo de alcançabilidade, ou grafo de ocorrência, onde cada nó corresponde a um estado do sistema e cada transição corresponde à ocorrência de um evento que leva o sistema do estado de origem a um estado de destino.

A Figura 2.5 é um exemplo da representação gráfica de um grafo de ocorrência. Neste grafo, a cada vértice está associado um estado e a cada arco está associado um evento. Um estado é formado por um conjunto de associações atributo-valor, descritos em uma notação interna conveniente. Um arco é uma tripla *(o, e, d)*, onde *o* é o vértice de origem, *e* é o evento executado e *d* é o vértice de destino. Em todo o grafo de ocorrência os elementos são únicos, ou seja, não podem existir dois vértices idênticos, bem como também não há dois arcos idênticos. Nesta figura, temos estados e arcos possuindo identificadores.

O principal problema que dificulta a aplicação de verificação de modelos em sistemas de software concorrentes, conforme já mencionado, é a explosão do espaço de estados, que implica em um crescimento exponencial do tamanho do grafo de ocorrência em função do número de componentes do modelo. Sistemas com alto nível de paralelismo apresentam naturalmente seqüências com muitas intercalações, que decorrem da execução de várias combinações de um mesmo conjunto de eventos. Por isso, em sistemas com muito paralelismo torna-se difícil a obtenção de todo o espaço de estados. Neste cenário, vale mencionar que a técnica de redução de ordem parcial também se mostra promissora, recebendo investigação por parte de várias comunidades.

A geração normalmente é feita em largura (Breadth-First Generation), como descrito no Algoritmo 1. Primeiro, um estado inicial é colocado na fila de estados não explorados e também em uma árvore de busca, cujo papel é armazenar os estados alcançados, provendo uma busca eficiente no espaço de estados. O processo entra então em um laço que só termina quando não há mais estados na fila de não explorados. A cada iteração, o primeiro estado s da fila de não explorados é removido. O simulador é acionado para gerar todos os possíveis

Figura 2.5: Exemplo de um Grafo de Ocorrência

estados alcançáveis a partir de s. Para cada estado alcançável s_0 executa-se uma busca na árvore de busca. Caso s_0 não esteja na árvore, isto indica que s_0 nunca foi alcançado antes, então s_0 é adicionado à fila de não explorados e à árvore de busca. Por fim, um novo arco de s para s_0 é adicionado. A Figura 2.6 ilustra a relação entre os estados não explorados, os já explorados e a posição do próximo estado alcançado.

A ferramenta na qual implementou-se esses algoritmos chama-se J-Mobile. Originalmente ela foi implementada em [Sil04], mas que durante todo o trabalho recebeu manutenção e alguns conceitos tivemos que implementar novamente. A princípio, surgiram muitas dificuldades para se executar esse processo de forma eficiente, pois não houve uma preocupação durante a implementação dessa ferramenta de se obter uma forma eficaz de representação dos estados e transições que compõem o comportamento do modelo. Houve um pequeno refatoramento, para lidar com os estados na memória de forma mais eficiente.

```
Let og be the generated Occurrence Graph
procedure GENERATE(INITIALSTATE)
  unexplored.addLast(initialState)
  og.searchTree.add(initialState)
  while not unexplored.isEmpty() do
    s ← unexplored.removeFirst()
    next ← simulator.generateAll(s)
    for each s' ∈ next do
      if (s' ∉ og.searchTree) then
        unexplored.addLast(s')
        og.searchTree.add(s')
      end if
      og.addArcs(s,s')
    end for
  end while
end
```

Algoritmo 1: Algoritmo de Geração do Espaço de Estados (Breadth-First Generation)

Figura 2.6: Processo de Geração do Espaço de Estados (Breadth-First Generation)

2.5.2 O Veritas - Verificador de Modelos Centralizado

O nosso grupo possui uma ferramenta de verificação de modelos em redes de Petri orientadas a objetos chamada Veritas [RBGdF04]. Esta ferramenta sofria do problema da explosão do

espaço de estados ao ter que verificar sistemas com alguma complexidade considerável, pois utiliza uma única unidade de processamento durante a verificação e não houve um investimento durante o seu projeto em formas de representação mais eficiente do comportamento em memória.

Com o conjunto de requisitos da API de verificação descrito logo adiante na Seção 2.5.2, foi feito um estudo do Veritas inserido no contexto do projeto Divíduo [Cab05], necessitando adaptá-lo para conseguirmos validar os requisitos citados anteriormente. Ele foi escrito na linguagem funcional Moscow ML e sua arquitetura é descrita na Figura 2.7.

Figura 2.7: Arquitetura do Verificador de Modelos VERITAS

O módulo de especificação de propriedades CTL é responsável por dar suporte à especificação de propriedades. Existe no Veritas uma biblioteca com funções para facilitar a construção de expressões booleanas que podem ser entendidas pelo verificador. O funcionamento do Veritas não depende desta biblioteca. Por enquanto, ela existe para ajudar o usuário no processo de especificação.

O módulo de análise do espaço de estados faz a leitura do grafo de ocorrência a partir do arquivo que o armazena em disco e converte-o para uma estrutura de dados na solução que pode ser acessada pelo verificador através de funções específicas.

O módulo de verificação é responsável por explorar o espaço de estados, a partir de um certo estado, avaliando as expressões que compõem a especificação CTL. Ele está imple-

mentado na forma de uma função que deve receber dois parâmetros: um inteiro e um espaço de estados. O primeiro parâmetro é o identificador do estado a partir do qual a expressão booleana deve ser avaliada e o segundo parâmetro é o espaço de estados do sistema.

Conforme já foi mencionado, a API de verificação deve interagir diretamente com a API de simulação e geração de espaços de estados J-Mobile. Isto é retratado também pelos próprios requisitos que foram levantados já prevendo futuras extensões das funcionalidades que foram previstas para esta implementação. Por exemplo, para suportar a verificação *on the fly*, o verificador precisa acessar os estados do comportamento do sistema pedindo diretamente ao simulador conforme a sua necessidade.

Requisitos do Protótipo de Verificação

O grupo busca implementar e utilizar uma ferramenta de verificação de modelos. Após um aprofundamento teórico na técnica, foi feito o levantamento de requisitos para essa solução. Estes requisitos são justificados na flexibilidade provida por eles quanto à forma de implementação, não importando os algoritmos, plataforma ou tecnologia. Eles devem servir como direcionamento para a solução que o grupo necessita. Os principais requisitos que foram levantados estão listados abaixo:

1. Ler e tratar modelos em RPOO segundo a gramática descrita em [Sil04].
2. Ler e tratar especificações CTL segundo o formato descrito em [RBGdF04].
3. Acionar a geração de espaços de estados completa.
4. Acionar a geração de espaços de estados parcial com condição de parada explícita.
5. Acionar a geração de espaços de estados parcial com funções de mapeamento estado/partição.
6. Acionar a geração de espaços de estados sob demanda de propriedades CTL *on-the-fly*.
7. Exportar espaços de estados em formato textual segundo a sintaxe descrita em [Cab04].
8. Ler espaços de estados segundo a sintaxe descrita em [Cab04].

9. Iniciar o processo de verificação sobre o espaço de estados lido.
10. Mostrar progressão do processo de verificação.
11. Exibir relatório final da verificação. O relatório deve conter um valor verdade (*verdadeiro* ou *falso*) e, quando falso, um contra-exemplo que invalida uma dada propriedade.
12. Salvar o *log* de execução resumindo o processo de verificação.
13. Suportar acoplamento de módulos de tratamento do espaço de estados. Isto possibilita uma futura implementação das técnicas de redução do espaço de estados.
14. Poder interagir diretamente com o simulador para se poder implementar verificação *on-the-fly*.

Embora estes requisitos tenham sido definidos após a implementação de um verificador centralizado, conseguimos tornar compatível a implementação com essa especificação e a partir daí desenvolvermos a solução distribuída apresentada no capítulo seguinte.

O caso de uso mais geral para esta ferramenta ilustra como a arquitetura proposta pode satisfazer os requisitos levantados:

1. Existe um modelo RPOO descrito em um ou mais documentos da gramática que foi definida.
2. Este modelo é lido para o simulador do J-Mobile.
3. O verificador solicita ao J-Mobile que gere o espaço de estados.
4. Existe um documento com as propriedades especificadas em CTL.
5. O módulo de análise e especificação de propriedades lê essas propriedades e as analisa.
6. O processo de verificação é iniciado.
7. O resultado é mostrado na tela.
8. Caso o resultado seja falso, o usuário solicita ao simulador para visualizar o contra-exemplo.

9. O simulador mostra em elementos do modelo a evolução do sistema até a especificação ser invalidada.

Infelizmente, o problema da explosão do espaço de estados dificulta bastante a implementação, validação e implantação de ferramentas com nossa arquitetura. O nosso trabalho investiga uma alternativa de distribuição para tentarmos viabilizar o uso dessa arquitetura. O próximo capítulo introduz em detalhes nossa técnica de verificação distribuída de modelos.

Capítulo 3

Verificação Distribuída de Modelos

Neste capítulo, tratamos em detalhes os nossos algoritmos para as experimentações em verificação distribuída de modelos para as experimentações requisitadas pelo trabalho. Aqui estão concentradas as principais contribuições do nosso trabalho: o reúso e adaptações de um método de particionamento para sistemas descritos em redes de Petri para plataformas com a filosofia Bag-of-Tasks de comunicação, um método de geração distribuída de espaços de estados para sistemas concorrentes e algoritmos distribuídos de verificação na lógica temporal CTL. Tudo isto está sintetizado no conjunto de ferramentas denominado Divíduo.

Este trabalho não apresenta provas de corretude e validação formal desses algoritmos de verificação de modelos. Essa prova foge do escopo do trabalho, pois estamos procurando por resultados experimentais que elucidem o emprego da plataforma de grades computacionais na verificação distribuída de modelos. Como não achamos algoritmos específicos já prontos que pudessem ser reusados, tivemos que produzir nossos próprios, o que aumentou bastante a complexidade do trabalho e que demandam por recursos humanos e tecnológicos que no momento não temos disponíveis em nosso grupo de pesquisa.

3.1 Particionamento

A geração e verificação de espaços de estados não é inerentemente uma aplicação do tipo Bag-of-Tasks. Este fato se deve à possibilidade de haver mais de um arco de entrada para um mesmo vértice, pois muitas vezes o número de componentes fortemente conectados é grande e não é possível deixar de haver comunicação entre os fragmentos do grafo quando dividido.

A forma mais natural de quebrar esta aplicação em tarefas consiste em dividir o grafo em regiões ou partições, um para cada tarefa. Contudo, para manter a integridade do grafo como um todo, é preciso preservar todos os seus componentes (nós e arcos), bem como a relação de unicidade entre eles. Nada deve impedir que um vértice alcançado em uma tarefa tenha arcos de entrada que devam ser computados por outra tarefa. Isto gera a dependência entre tarefas, o que não caracteriza uma aplicação Bag-of-Tasks.

Para resolvermos esse problema, a solução assume o espaço de estados como sendo composto por um conjunto finito de partições. Estas partições são definidas estaticamente através de uma função de particionamento $h : S \rightarrow \{0, \ldots, N-1\}$ mapeando todo estado alcançável em um número natural, que representa o identificador da partição. Esta função é específica do domínio, ou seja, projetada especificamente para o modelo sob análise. Em nossa implementação, há um módulo de geração de espaços de estados para cada partição que é responsável pela computação do fragmento do grafo determinado. Na Figura 3.1, um exemplo de um particionamento de um espaço de estados é mostrado. Dado um grafo de ocorrência original, aplicando uma função de particionamento, obtemos esse grafo fragmentado no número de subgrafos escolhidos pelo engenheiro de sistemas. s_i é o estado inicial e as áreas acinzentadas são as partições. A nível de implementação, cada partição é processada por uma tarefa que roda na grade computacional.

Figura 3.1: Um espaço de estados particionado

O desempenho de computação na construção e exploração distribuída de espaços de estados depende do particionamento adotado. Em um particionamento ótimo, três pontos necessitam serem considerados: equilíbrio espacial, localidade e equilíbrio temporal. Equilíbrio significa que o número de estados processados deve ser aproximadamente o mesmo em cada

máquina. Localidade significa que o número de transições que ligam estados em diferentes partições processadas em diferentes máquinas deve ser minimizado tanto quanto para evitar custo de comunicação. A comunicação entre partições é dada através do termo *exportar* um estado, que significa enviar esse estado para sua partição correspondente para que possa ser devidamente computado. Balanço temporal significa que cada partição deve ser processada em paralelo o maior tempo possível.

Para conseguirmos nos aproximar desses pontos mencionados, podemos usar heurísticas que dependam também da intuição do modelador, como a proposta de Ciardo [CGN98; KP04], ou alguma análise prévia que extraia informação relevante para se definir o particionamento. Como exemplo, podemos citar o método proposto por Orzan [OvdPE04].

A heurística proposta por Ciardo sugere funções hash cujos parâmetros sejam conteúdos com dados específicos quem compõem o modelo. No geral, quando uma transição dispara, apenas um pequeno subconjunto dos lugares que mantenham os dados são alterados. Assim, a idéia fundamental dessa abordagem é definir uma função de particionamento sobre o conteúdo de um pequeno subconjunto $\mathcal{P}'$ de todos os lugares que mantenham dados e que é chamado de *conjunto de controle*. Este conjunto é a base de todas as possíveis configurações, porém não é fornecida uma maneira automática de detecção desse conjunto. O disparo de uma transição que não envolva nenhum lugar com dados em $\mathcal{P}'$ significa uma transição de estados entre dois estados atribuídos à mesma partição. O disparo de uma transição deve corresponder a um arco entre partições somente se ele modifica a marcação do conjunto de controle $\mathcal{P}'$.

Orzan utiliza interpretação abstrata para computar pequenas aproximações do espaço de estados e fazer uma predição de conectividade de estados concretos sobre esta aproximação.

A heurística usada neste trabalho é baseada no trabalho de Ciardo. No intuito de reduzir o número de arcos que atravessam partições e tentar conseguir equilíbrio temporal, a noção de transições conflitantes foi considerada. Duas ou mais transições estão em conflito se o disparo de qualquer uma delas desabilita o das demais no próximo estado. Um exemplo de uma transição conflitante é ilustrado na Figura 3.2. Intuitivamente, esta observação significa que caminhos resultantes de transições conflitantes possuem uma grande chance de derivarem estados pertencentes à mesma partição, aumentando assim a localidade do particionamento. Assim, o principal ponto dessa estratégia consiste em usar lugares que mantenham

dados que estejam associados a transições conflitantes para compor o conjunto de controle. As mudanças no conjunto de controle implicam que o estado alcançado deva ser exportado e o caminho que inicia imediatamente nesse estado não irá ser processado imediatamente, aguardando estar em uma outra partição.

Figura 3.2: Situação de conflito em uma transição

Para entender melhor nossa heurística, considere os estados s, s' e r, transições t e t' tal que (s, t, s') e $(s, t', r) \in \Delta$, e t e t' são transições conflitantes. Se o *conjunto de controle* contém os lugares mantenedores de dados que estão associados a t e t', os estados s' e r podem ser exportados para partições diferentes. Resumindo, os caminhos que serão iniciados no estado s' provavelmente não exportarão um estado para a partição que irá começar no estado r e vice versa.

Assim, é possível controlar a localidade e o equilíbrio temporal. Naturalmente, a qualidade do particionamento será melhor se os caminhos iniciando nos estados s' e r forem profundos o suficiente até a próxima exportação. Este fato depende do modelo sob verificação e não existe uma regra geral para predizer isto para todas as classes de modelos. Da mesma forma, o equilíbrio depende da experiência do usuário com o modelo sob verificação. O usuário deve selecionar corretamente todos os lugares que mantenham dados com o objetivo de alcançar o mesmo número de estados em cada partição. O conjunto de ferramentas desenvolvido nesse trabalho não possui um mecanismo que detecte automaticamente

as transições conflitantes. Esta atividade deve ser executada manualmente. Contudo, alguns mecanismos permitem esta atividade por meio de análises estruturais ou sintáticas.

3.2 Arquitetura do Divíduo

Para que pudéssemos empregar o método experimental em nossas investigações, fez-se necessária a implementação de um protótipo para executarmos os algoritmos de verificação nos modelos escolhidos. O desenvolvimento dessa ferramenta é uma tarefa complexa e requisitou um projeto arquitetural. Sua arquitetura é composta em camadas dispostas em cinco níveis abstração, conforme ilustra a Figura 3.3.

Figura 3.3: Arquitetura do Divíduo

A primeira camada é responsável pela interação com o usuário, há módulos para construção do modelo e especificação de propriedades, além da ferramenta de simulação de modelos J-Mobile. A segunda camada apresenta as funcionalidades básicas do Divíduo que geralmente são enviadas para serem executadas nas máquinas remotas e que serão detalhadas mais adiante, que são a geração e verificação distribuída do espaço de estados, além do mé-

todo de particionamento. Estas são as partes destacadas na figura e que foram contribuições providas por nosso trabalho.

A parte mais clara da Figura 3.3 é referente à distribuição e foi reusada da solução Ourgrid. A segunda camada é complementada com a biblioteca de implementação das primitivas do Mygrid. Esta juntamente com as outras camadas provêem a infra-estrutura de distribuição do processo. O Mygrid é a interface que faz escalonamento das tarefas para a comunidade e está na terceira camada, ainda na máquina do usuário. A quarta camada é composta dos serviços automáticos implementados pela solução, que despreocupa o usuário do particionamento a nível de recurso. Esta camada gerencia automaticamente questões como escalonamento eficiente dos recursos, autenticação e autorização, e outras mais necessárias durante o envio e recolhimento das tarefas processadas. Por fim, a quinta camada é a comunidade Ourgrid em si, com seus recursos disponíveis de uma forma completamente heterogênea ao redor do planeta.

3.3 Geração Distribuída de Espaços de Estados

Iniciado o processo de geração, quando uma partição alcança um estado pertencente a uma outra partição, este estado é armazenado e encaminhado apenas uma vez para sua partição correspondente. Assim, cada partição é responsável apenas por explorar os sucessores dos estados pertencentes a ela. Contudo, já sabemos que a comunicação direta entre tarefas não é suportada, pois não há uma primitiva que permita que um estado seja exportado diretamente para outra partição. Para permitirmos essa comunicação, o processo de construção teve que se basear em uma entidade central chamada de *coordenador*. Isto ainda garante que partes do processo de geração sejam independentes. Assim, temos a implementação de um mecanismo de troca de mensagens assíncronas entre tarefas.

O *coordenador* é responsável por colocar todas as tarefas para trabalharem em um ambiente distribuído interagindo a interface, que é o Mygrid, da grade computacional. Além do mais, o coordenador coleta e despacha os estados que devem ser trocados entre as partições. Desta maneira, toda tarefa que é escalonada para executar em uma máquina remota descontinua temporariamente sua execução e retorna ao *coordenador*. Em seguida, o *coordenador* recebe e entrega os estados que foram exportados pelas partições. Uma vez que uma tarefa

tenha estados não explorados, o coordenador irá reescaloná-la até que sua parte do espaço de estados tenha sido totalmente construída. O Algoritmo 2 descreve suscintamente o comportamento dessa entidade. Detalhando mais, inicialmente são alocadas na caixa de mensagens espaços para N partições conforme especificado pelo modelador. Logo após, de fato é criada a partição inicial, com s_0 empilhado para ser explorado inicialmente e é feita a coleta das partições prontas para serem exploradas, que nesse caso será apenas a partição detentora de s_0. A partir daí entramos em um laço que irá ser encerrado apenas quando todas as partições tiverem seus estados como finalizados ou não existirem mais estados a serem explorados, aumentando assim o tamanho do grafo de ocorrência gerado. Dentro desse laço serão criadas tarefas na grade para cada partição pronta e colocadas para rodarem pelo coordenador no Mygrid. Após o cômputo dessas partições, são enviadas mensagens baseando-se em uma tabela de estados que indica quando ocorreu finalização dos processos.

```
procedure COORDINATOR()
  createAllPartitions(N)
  createInitialPartition(s_0)
  collectDispatchStates()
  while not checkForTermination() do
    createTask()
    putToWorkOnGrid()
    getGridResults()
    collectDispatchStates()
  end while
end
```

Algoritmo 2: Algoritmo para o processo de coordenação

3.3.1 O Processo de Geração Distribuída

Partes da aplicação serão executadas somente nas estações da grade, enquanto outras somente na estação coordenadora. Na Figura 3.4, é ilustrado passo a passo o procedimento de geração distribuída de espaços de estados sobre a plataforma adotada. Inicialmente, o desenvolvedor ou engenheiro de sistemas dotado de poucos recursos define o seu modelo em alguma linguagem de formalismos concorrentes conforme explicado anteriormente e define

uma função de particionamento baseada no modelo. Em seguida ele aciona o coordenador, que está representado na Figura como Divíduo por ser o cerne da nossa solução. Tendo em mões o estado inicial do modelo, o Divíduo cria um aplicações remotas dispostas a computar o comportamento do modelo gerado a partir daquele estado inicial. Essas aplicações são escalonadas na solução Ourgrid pela ferramenta Mygrid e enviada para o Par gerenciador dos recursos da comunidade. Em cada recurso alocado, uma aplicação remota é iniciada. Ao término de uma aplicação remota em um dos recursos, os resultados são enviados de volta à máquina coordenadora através do Mygrid. O coordenador então torna-se responsável pela coleta e interpretação desses resultados e poderá re-escalonar outras aplicações remotas sempre que ainda existirem estados a serem computados pelo processo. Ao final, os dados completos do processo de geração são exibidos para o usuário juntamente com o grafo de ocorrência fragmentado gerado.

Figura 3.4: Visão geral do processo de geração distribuída de espaços de estados

Definições Necessárias ao Processo

Primeiramente, o coordenador cria partições vazias que deverão compor o espaço de estados. Naturalmente, o número de partições é definido de acordo com a função de particionamento h. Cada partição P_i é definida da seguinte maneira:

Definition 3.3.1 *$P_i = \{V_i, E_i, OB_i, IB_i, T_i\}$, tal que V_i é o conjunto de estados que foram alcançados pela tarefa, mas não foram completamente processados; E_i é o conjunto de estados que foram alcançados e foram totalmente explorados; OB_i é o conjunto de estados que foram alcançados pela tarefa, mas não pertencem à partição; IB_i é o conjunto de estados que foram alcançados por outra tarefa e encaminhados para a partição correspondente; e finalmente, T_i é o conjunto de transições entre estados que compõem a partição.*

Na prática, OB_i e IB_i existem devido ao modelo de distribuição adotado, no qual não permite a troca direta de mensagens entre tarefas diferentes. Eles podem ser interpretados como uma implementação de canais de comunicação entre tarefas que serão gerenciados pelo coordenador. Eles são uma área de intercâmbio que representam as relações de endereçamento e seus itens são retirados pelo processo coordenador para serem enviados às suas respectivas partições. No espaço de estados distribuído, os estados armazenados nos canais de comunicação representam o conjunto de arcos entre partições χ:

Definition 3.3.2 *For all $i, j \in \{0, \ldots, N-1\}$ tal que $i \neq j$, o conjunto de arcos entre partições χ é $\chi = \{(s, t, s') \mid s \in P_i \wedge s' \in P_j \wedge (s, t, s') \in \Delta\}$.*

O espaço de estados distribuído $\mathbb{S}$ é definido da seguinte forma:

Definition 3.3.3 *Seja $h : S \to \{0, \ldots, N-1\}$ uma função de particionamento mapeando cada estado alcançável em seu identificador de partição correspondente, um espaço de estados distribuído $\mathbb{S}$ é $\mathbb{S} = \bigcup_{i=0}^{n-1} P_i$*

As relações de endereçamento (ou rotas) são arcos cujos estados de origem e destino estão em partições diferentes. Elas são essenciais para a recomposição do grafo de ocorrência que ficou fragmentado após o particionamento. O processo que coordena a distribuição de tarefas também fornece mecanismos para a manutenção da tabela de roteamento que, por sua vez, é distribuída entre as partições. Uma rota pode ser definida como uma tupla ordenada

de identificadores (*p*, *s*, p_0, s_0), onde *p* é a partição de origem, *s* é o estado de origem, p_0 é a partição de destino e s_0 é o estado destino. É importante ressaltar que uma rota só pode ser obtida em duas fases: *p*, *s* e p_0 são obtidos na partição que exporta, enquanto s_0 só pode ser registrado pela partição que importa, uma vez que o identificador s_0 é obtido somente após a importação do estado por p_0. A Figura 3.5 mostra um exemplo do grafo de ocorrência gerado em três partições por nosso protótipo. Cada partição tem seus estados hachurados de uma forma diferente e seguindo a definição de máquinas de estados, se um estado não tiver transição de saída, significa implicitamente que há um auto-laço para esse mesmo estado.

Figura 3.5: Exemplo de grafo de ocorrência particionado junto com a tabela de roteamento

Para finalizar, a Tabela 3.1 apresenta as variáveis globais empregadas no módulo coordenador do processo de geração de espaços de estados da nossa solução.

Iniciando a Geração

Uma vez criadas as partições, o coordenador inicia o primeiro turno. De acordo com a função h, a partição que possui o estado inicial s_0 será processada. O final deste turno, assim como dos outros, é limitado por dois fatores. O primeiro é no tamanho do buffer de saída OB_i.

Identificador	Descrição
MB	*Caixa de mensagens* responsável por armazenar os estados exportados por outras partições que estejam endereçados a uma dada partição. Cada partição corrente terá um campo reservado nessa caixa de mensagens para uma dada fórmula de lógica temporal que contém os estados a serem explorados e o resultado booleano temporário dessa avaliação.

Tabela 3.1: Descrição das principais variáveis utilizadas nos algoritmos de verificação distribuída.

Isto significa que em um turno, toda partição deverá construir o espaço de estados até que o buffer de saída esteja totalmente preenchido. O segundo se dá quando todos os estados alcançáveis neste turno tiverem sido explorados. Neste caso, mesmo que o buffer de saída não esteja totalmente preenchido, o turno será finalizado porque não existem mais estados para serem processados. O procedimento para construirmos a partição inicial é mostrado no Algoritmo 3.

Para prevenirmos que o coordenador se torne um gargalo, esta é a única situação em que o coordenador constrói uma partição. Após isto, o coordenador gerenciará apenas a construção distribuída. O procedimento $createPartition$ é o conjunto de instruções básicas que toda tarefa executa nas máquinas remotas, e por esta razão, este procedimento será apresentado na subseção seguinte, quando a fase remota for discutida.

procedure CREATEINITIALPARTITION(s_0)
  $i \leftarrow h(s_0)$
  $IB_i \leftarrow IB_i \cup s_0$ `{Escreve o estado inicial no buffer de entrada correspondente.}`
  createPartition(i)
end

Algoritmo 3: Iniciando a primeira partição

Em seguida, o coordenador remove todos os estados no buffer de saída OB_i e entrega-os às suas partições de destino. Ao final deste primeiro turno, todas as partições de destino estão no coordenador e a entrega pode ser feita sem problema algum. Contudo, o mesmo não pode ser dito sobre os turnos seguintes. Como as tarefas estão rodando em uma plataforma hete-

rogênea, uma pode demorar mais tempo para executar do que a outra. Assim, para evitarmos que uma partição permaneça ociosa no coordenador durante muito tempo esperando pelas partições que devam ser contactadas, o coordenador armazena os estados em uma estrutura que trabalha como uma caixa de mensagens. Este algoritmo está ilustrado no Algoritmo 4.

```
procedure COLLECTDISPATCHSTATES()
  {Armazene os estados em suas respectivas caixas de mensagens}
  for each partição i executando no coordenador do
    for each estado s ∈ OB[i] do
      dest ← h(s)
      MB[dest] ← MB[dest] ∪ {s}
    end for
    OB[i] ← ∅
  end for
  {Despache o conteúdo da caixa de mensagens}
  for each partição i executando no coordenador do
    IB[i] ← MB[i]
    MB[i] ← ∅
  end for
end
```

Algoritmo 4: O algoritmo para coletar e despachar os estados borda

A Geração Remota

Neste ponto, a construção distribuída pode ser iniciada de fato. Desta forma, uma tarefa é criada para cada partição que deva ser processada. Uma partição P_i deve ser processada se P_i tem estados que foram encontrados mas ainda não foram explorados ($V_i \neq \emptyset$), ou se P_i tem estados armazenados em seu buffer de entrada ($IB_i \neq \emptyset$). No intuito de executarmos estas tarefas em um ambiente distribuído, o coordenador compõe um pacote de tarefas (*job*) com essas tarefas e delega as execuções para a grade computacional.

A grade computacional é responsável por definir o número de máquinas remotas que serão usadas. Seja n o número de partições e m o número de máquinas remotas disponíveis na grade computacional. Três situações podem ocorrer:

1. $m < n$: se existem menos máquinas remotas do que partições, algumas tarefas serão escalonadas sequencialmente. No pior cenário, no qual $m = 0$, o coordenador e as tarefas irão executar na mesma máquina.

2. $m = n$: neste caso, todas as tarefas devem executar em paralelo.

3. $m > n$: para evitarmos perdas de recursos, a ferramenta de verificação pode explorar o sistema de escalonamento provido pela grade computacional, no qual é baseado em replicação [dSCB03]. Pode-se argumentar que a melhor escolha é tentar usar todas as máquinas remotas criando novas partições. Esta escolha depende bastante da estratégia que foi usada para definir o particionamento. Há uma seção especial para tratarmos apenas sobre particionamento nessa dissertação.

```
procedure CREATETASKS()
  for each partição i ∈ P do
    if V[i] ≠ ∅ or IB[i] ≠ ∅ then
      {Cria as tarefas de acordo com as especificações da grade
      computacional}
      createPartition(i)
    end if
  end for
end
```

Algoritmo 5: O algoritmo que constrói o pacote das tarefas.

O algoritmo 6 apresenta o que é executado em cada máquina remota. Conforme mencionado, o trabalho de cada tarefa é construir uma parte específica do espaço de estados. Quando a tarefa alcança um estado que pertence a uma outra partição, essa tarefa deve exportar o estado. Aqui, exportar significa armazenar esse estado em um buffer de saída porque não há uma forma de enviá-lo para uma outra tarefa. Consequentemente, a construção irá executar enquanto existirem estados para serem explorados e o buffer de saída não for totalmente preenchido. Há uma tabela que pode manter pequenas referências de estados que foram alcançados por outra partição. Assim, antes que estados já visitados por outras partições sejam explorados, o coordenador possui o mecanismo de detecção de quem já foi explorado e os descarta de um re-escalonamento.

```
procedure CREATEPARTITION(i)
  for each estado s ∈ IB[i] do
    if (s ∉ V[i] ∪ E[i]) then
      V[i] ← V[i] ∪ {s}
    end if
  end for
  IB[i] ← ∅
  while (not(full(OB[i])) and V[i] ≠ ∅) do
    let s ∈ V[i]
    V[i] ← V[i] \ {s}
    if (h(s) = i) then
      for each s' ∈ successors(s) do
        T[i] ← (s, s')
        if (s' ∉ V[i] ∪ E[i]) then
          V[i] ← V[i] ∪ {s'}
        end if
      end for
    else
      OB[i] ← OB[i] ∪ {s}
    end if
    E[i] ← E[i] ∪ {s}
  end while
end
```

Algoritmo 6: O algoritmo executado pelas tarefas

Uma vez que o trabalho foi delegado à grade, o coordenador deverá esperar por um período de tempo o retorno das tarefas. Após este intervalo, a informação chegada será carregada na memória principal pelo coordenador no intuito de continuarmos com a construção.

Detectando Terminação

O final da geração distribuída é verificado de uma maneira centralizada. Quando todas as tarefas tiverem voltado ao coordenador, e a caixa de mensagem estiver vazia, o final da

construção distribuída é detectado se

$$\bigcup_{i=0}^{n} OB_i = \emptyset \ \wedge \ \bigcup_{i=0}^{n} IB_i = \emptyset \ \wedge \ \bigcup_{i=0}^{n} V_i = \emptyset$$

ocorrer. Estas condições garantem que nenhuma tarefa esteja executando em uma máquina remota e também que todas as tarefas tenham processado suas partes, já que o espaço de estados é finito.

3.3.2 Visão Geral da Arquitetura do Protótipo de Geração Distribuída

Em termos de projeto, o seguinte modelo conceitual na Figura 3.6 ilustra os principais módulos ou pacotes na idéia original do protótipo.

Figura 3.6: Modelo conceitual inicial do protótipo de geração distribuída

Os módulos são classificados da seguinte maneira:

- *Local*. É responsável por lidar com a configuração inicial do modelo sob verificação e enviá-la para ser processada. Neste módulo, também é feito todo o gerenciamento das

partições, tais como instanciação, envio para serem processadas, coleta de resultados e estados exportados. Este módulo também é responsável pelo controle da tabela de roteamento entre partições, que é responsável pela navegabilidade completa no grafo.

- *MyModel*. O modelo representado no formalismo concorrente propriamente dito. A partir desse modelo podemos obter o estado inicial do comportamento para iniciarmos todo o processo.

- *Distributed*. Este módulo é o que vai ser executado nas máquinas remotas da comunidade e é responsável pela computação dos fragmentos do grafo. Para isso ele é basicamente composto de um objeto partição. Uma partição é composta de um identificador, buffers para armazenar estados a serem importados e exportados, um objeto particionador que contenha a função de particionamento definida pelo usuário, lista de estados a serem explorados, um relatório e está associada ao grafo de ocorrência parcialmente computado. Este projeto está mais claro no diagrama de classes desse pacote que é apresentado na Figura 3.7.

Figura 3.7: Diagrama de classes do pacote distributed

3.4 Verificação Distribuída de Espaços de Estados

Em seções anteriores, apresentamos os requisitos que elaboramos de uma solução desejada de verificação de propriedades em sistemas concorrentes independente da sua implementação. Infelizmente, a solução inicialmente proposta pelo grupo [RBGdF04], que era centralizada, sofreu de diversos problemas relativos à falta de memória para término do processo quando necessitávamos verificar sistemas com alguma complexidade. Diversas tentativas de amenizar esse problema foram estudadas pelo grupo. Relatamos a formalização da investigação do uso de grades computacionais como plataforma de apoio à verificação.

Assumindo um espaço de estados distribuído gerado e sua divisão em um conjunto finito de partições, produzimos uma ferramenta de verificação de modelos em CTL distribuída que é capaz de cobrir todo o comportamento dos sistemas sob análise. Esta ferramenta de verificação está desacoplada da ferramenta de geração de espaços de estados, recebendo como entrada apenas o grafo de ocorrência já fragmentado respeitando a sintaxe estabelecida para representação. Maiores detalhes sobre os modelos que foram experimentados nessa ferramenta, sua implementação e representação gramatical de seu comportamento serão apresentados na seção de resultados experimentais.

3.4.1 A Idéia do Verificador

A idéia principal também é similar a vários algoritmos distribuídos: o espaço de estados é particionado entre as máquinas da grade, isto é, cada máquina da grade verifica um subconjunto do espaço de estados. Isto é feito aplicando-se algoritmos CTL centralizados originalmente descritos em [RBGdF04] para cada partição e sincronizando seus resultados. Assim, adotamos os seguintes operadores básicos: *AP*, *NOT*, *AND*, *EX*, *EG* e *EU*, ficando os demais operadores CTL sendo originados da composição desses operadores básicos. Como neste trabalho apresentamos apenas um protótipo, e não tivemos tempo e nem recursos suficientes para implementação, limitamo-nos apenas em alguns operadores que cobrissem nossos interesses para os resultados experimentais. Identificamos aqui a necessidade de um trabalho de validação formal desses algoritmos posteriormente.

Iniciando a Verificação

Primeiramente, assim como em toda ferramenta de verificação de modelos, o modelador deve especificar as propriedades do sistema em observação e os objetivos de verificação desejados. Isto pode ser expresso da seguinte maneira:

Definition 3.4.1 *A tupla (s_0,P_i,ϕ) é a entrada do verificador, tal que s_0 é o estado inicial e $s_0 \in E_i$, P_i é um identificador para a partição do grafo $P_i = \{E_i, OB_i, subOP\}$, que serão explicados adiante, e ϕ é uma fórmula CTL a ser checada.*

No geral, o processo de verificação distribuída é coordenado por uma pilha global denotada por Γ. No caso, inicialmente empilhamos em Γ a tupla (P_i,ϕ) para ser decomposta e verificada através das partições.

As principais variáveis utilizadas nos algoritmos estão ilustradas na Tabela 3.2. Estas variáveis são úteis para o gerenciamento global do processo de verificação das partições na máquina local do usuário.

A seguir, o que está empilhado na cabeça é retirado e escalonado na grade computacional. O procedimento para se fazer isso chama-se *dispatchAll* e está descrito a seguir:

A implementação do procedimento *dispatch* depende da plataforma de distribuição que se esteja usando. Neste trabalho, enfatizando mais uma vez, procuramos desacoplar o máximo possível a atuação da ferramenta de verificação e da plataforma de distribuição empregada. Ilustrando brevemente como se dá na plataforma adotada, a comunicação entre o coordenador e as máquinas da grade computacional se dá através de serialização e desserialização de arquivos. No procedimento *dispatch* específico do Ourgrid, serializamos cada partição em um arquivo e criamos uma thread que represente essa partição trabalhando. Ao término dessa execução na máquina remota, essa thread recebe uma notificação que indica que os resultados produzidos pela máquina remota devem ser coletados.

A Verificação Remota

A arquitetura proposta para verificador distribuído segue a mesma idéia do gerador de espaços de estados distribuído. Detalhes da construção do modelo do sistema concorrente sob verificação, o método de particionamento e a comunidade Ourgrid já foram explicados nesse

Identificador	Descrição
Γ	*Pilha* global para armazenar os contextos de verificação das partições. Em cada nível são armazenadas listas de identificadores das partições que tiveram sua execução abortada por limitação do buffer de comunicação ou por temporariamente não possuir mais estados a explorar. O resultado de computação dessas partições geralmente é dado como indefinido, pois estão dependendo de resultados lógicos computados por outras partições para poderem retornar o seu processamento.
ST	*Tabela* com o status momentâneo de cada partição. Classificamos os estados como *pronto* (*READY*), *rodando* (*RUNNING*), *falho* (*FAILED*), *finalizado* (*FINISHED*), *cancelado* (*CANCELED*) e *empilhado* (*QUEUED*), que é quando esta partição encontra-se em Γ aguardando ser realocada pelo coordenador. Ao ser alterado algum campo desta tabela, há o disparo de um evento que notifica os observadores cadastrados dela, sinalizando mudança de estado na grade computacional.
toDispatch	*Lista* de identificadores de partições que estavam armazenadas na cabeça de Γ e que acabaram de ser desempilhadas. Para essa lista poder voltar a ser computada na grade computacional, deve passar por alguns testes sobre seu status nos algoritmos que serão apresentados adiante.

Tabela 3.2: Descrição das principais variáveis utilizadas nos algoritmos de verificação distribuída.

trabalho. Nesta seção, nós discutimos detalhes sobre o processo coordenador e a aplicação remota que executa algoritmos de verificação na lógica temporal CTL, na qual são os conceitos chave da nossa solução.

Cada máquina que roda na grade computacional é responsável por computar os resultados nos pares (P_n,ϕ_k), onde P_n é a partição corrente e $\phi_k \in$ *Sub(ϕ)* é uma sub-fórmula da fórmula original a ser verificada. Em nosso trabalho, cada par (P_n,ϕ_k) é chamado de *partição de verificação*. Cada máquina na grade é responsável por executar algoritmos de verificação CTL que podem acessar apenas uma parte do sistema inteiro. Dependendo do tipo do algoritmo e da relação de estados exportados, ela se comunica com outros nós para

```
procedure DISPATCHALL()
  if Γ = ∅ then
    return
  end if
  OPs ← pop(Γ)
  for each op ∈ OPs do
    if ST.status(op) ∈ {QUEUED, READY} or ST.status(op) = FINALIZED and
    MB.get(op).OB_i ≠ ∅ then
      toDispatch ← toDispatch ∪ MB.get(op)
    end if
  end for
  if toDispatch ≠ ∅ then
    dispatch(toDispatch) {Serializa cada partição e cria uma thread de acordo
    com a filosofia da plataforma}
  else
    dispatchAll() {Recomeça o processo até que a pilha global esteja vazia}
  end if
end
```

Algoritmo 7: Colocando as partições que estão empilhadas para trabalhar na grade

alcançar o resultado global requerido.

Analogamente ao processo de geração, que é baseado em primitivas Bag-of-Tasks da grade computacional, a comunicação é dada através de buffers para exportar e importar resultados. Um resultado de avaliação $R_{\phi n}$ no processo de verificação consiste na seguinte tupla:

Definition 3.4.2 *$R_{\phi n}$ = (s_i, β, π, Ψ) é um resultado de avaliação contido na partição de verificação (P_n,ϕ_k), aonde s_i é o estado a ser checado, β é o valor verdade sobre a avaliação, π é uma lista de estados que provam a veracidade de β como um testemunho ou contra-exemplo e Ψ é um conjunto de identificadores das partições de avaliação que exportaram s_i para (P_n,ϕ_k).*

Os significados das variáveis globais que compõem as partições são ilustradas na Tabela 3.3. Essas variáveis estarão presentes nos algoritmos CTL a seguir e sem o correto entendi-

mento dessas, esses algoritmos tornam-se ilegíveis.

Identificador	**Descrição**
Pid	*Inteiro* que serve como identificador da partição do espaço de estados a ser verificado.
E_i	*Conjunto* dos estados que já foram visitados pelo algoritmo de verificação. Não necessariamente, teremos um resultado definitivo sobre a avaliação dessa fórmula, podendo estar com um valor indefinido também, conforme mencionamos. No início, esse conjunto deve ser vazio.
$subOP$	*Operador CTL* que compõe a fórmula CTL da partição. Assim, por exemplo, na fórmula CTL *AGEGp*, temos que *EGp* é o sub-operador de *AG*. Em nossa solução, as fórmulas CTL são construídas por composição. Seus resultados lógicos de avaliação são combinados também. Ainda mais, se o operador for do tipo *EU* ou *AU*, teremos dois sub-operadores para cada um destes operadores.
$property$	*Propriedade* a ser satisfeita em um determinado estado. O atributo é usado na classe que contém as proposições atômicas.
OB_i	*Buffer* de saída com os estados que não puderam ser completamente avaliados porque ainda dependem de resultados lógicos processados por outras partições.

Tabela 3.3: Descrição das principais variáveis utilizadas nos algoritmos de verificação distribuída.

Ilustramos no Algoritmo 8 o procedimento para avaliarmos a satisfação de uma fórmula booleana em um dado estado. Este é o operador mais básico da lógica temporal CTL, e é denotado por AP (de *atomic proposition*). Observe que diferentemente de outros algoritmos, ao encontrarmos um valor que valide essa proposição atômica, não inserimos o estado avaliado nos contra-exemplos ou testemunhos do operador. Deixamos esta responsabilidade para a classe de operador que for composta dessa proposição atômica.

O Algoritmo 9 avalia fórmulas com o operador *NOT*, que inverte o valor verdade da expressão avaliada.

O próximo algoritmo a ser apresentado é do operador AND no Algoritmo 10 que trata de

procedure AP(s_i, β_i, π_i, Ψ) (E_i, T_i)
`{Essa é a assinatura padrão da avaliação das fórmulas}`
if $property \in \{TRUE \mid FALSE\}$ **then**
$\beta_i \leftarrow property$
else
$\beta_i \leftarrow property \in s_i$.label
end if
end

Algoritmo 8: Algoritmo para checar o operador mais básico da lógica CTL, que são proposições atômicas. Chamado de AP.

procedure NOT(s_i, β_i, π_i, Ψ) (E_i, T_i)
(s_i, β_i, π_i, Ψ) $\leftarrow subOP$.evaluate((s_i, β_i, π_i, Ψ), (E_i, T_i))
if $property \in \{TRUE|FALSE\}$ **then**
$\beta_i \leftarrow property$
else
$OB_i \leftarrow subOP.OB_i$
for each $\beta_k \in E_i$ **do**
$E_i.s_i \leftarrow inverse(\beta_k)$
end for
end if
end

Algoritmo 9: Algoritmo para checar o operador NOT

conjunções entre duas fórmulas temporais.

O Algoritmo 13 ilustra um exemplo do que é computado em cada máquina da grade. Este algoritmo computa o operador *CTL EG*. *EG* ϕ é chamado de "potencialmente teremos sempre ϕ" e significa que existe ao menos um caminho, iniciando de um estado específico, onde ϕ vale globalmente. Cada partição (P_n,ϕ_k) roda em paralelo em um processo separado *n* checando a satisfação da fórmula CTL ϕ_k usando algoritmos de verificação de modelos explícitos adaptados apresentados em [RBGdF04]. Além do mais, ela tem um buffer para armazenar resultados parciais computados e comunicá-los às respectivas partições. Para lidarmos com o grafo fragmentado, a semântica padrão CTL foi adaptada da idéia apresentada em [BZ03]. Este algoritmo se completa com os Procedimentos 11 e 12 que são auxiliares

```
procedure AND(s_i, β_i, π_i, Ψ) (E_i, T_i)
  (s_1, β_1, π_1, Ψ_1) ← subOP_1.evaluate((s_i, β_i, π_i, Ψ), (E_i, T_i))
  if β_1 = FALSE then
    β_i ← FALSE
    return
  end if
  (s_2, β_2, π_2, Ψ_2) ← subOP_2.evaluate((s_i, β_i, π_i, Ψ), (E_i, T_i))
  if β_2 = FALSE then
    β_i ← FALSE
    return
  end if
  if β_1 = UNDEFINED or β_2 = UNDEFINED then
    β_i ← UNDEFINED
    OB_i ← OB_i ∪ s_i
    return
  end if
  β_i = TRUE
  pi_i ← pi_i ∪ s_i
end
```

Algoritmo 10: Algoritmo para checar o operador AND

e usados para, em caso de um resultado positivo ser encontrado, atualizar os resultados indefinidos deixados em outras partições durante sua exploração por não se saber qual seria o resultado da avaliação e testar se o estado deve ser explorando na partição corrente ou exportado para a sua respectiva.

O Algoritmo 14 para computar o operador *CTL EU* segue a mesma filosofia do que foi apresentado anteriormente, porém agora é requisitada a avaliação de duas fórmulas com dependências, o que torna a geração do contra-exemplo mais complexa.

O Algoritmo 15 computa o operador CTL EX. Este algoritmo não depende de computação paralela por possuirmos a imagem dos estados borda em outras partições que tenham rotas com a partição corrente. Reusamos este algoritmo de [RdFG04], visto que lá foi apresentada uma solução centralizada para verificação de modelos. Em linhas gerais, este algoritmo possui apenas um laço para visita de todos os sucessores do estado atual. Caso algum

```
procedure CHECKPREVIOUSEVALUATION(s_i, β_i, π_i, Ψ)
  if s_i ∈ E_i then
    β_i ← E_i.s_i
    if β_i = UNDEFINED or β_i = TRUE then
      return TRUE
    end if
  end if
  return FALSE
end
```

Algoritmo 11: Verificando as avaliações anteriores

```
procedure CHECKPARTITION(s_i, β_i, π_i, Ψ) (E_i, T_i)
  if i ≠ Pid then
    OB_i ← OB_i ∪ s_i
    β_i ← UNDEFINED;
    E_i.s_i ← UNDEFINED
    return FALSE
  end if
  return TRUE
end
```

Algoritmo 12: Verificando se um estado deve ser processado nessa partição

estado satisfaça a propriedade sendo checada, o procedimento é abortado e este pequeno caminho serve como prova do operador EX.

Classicamente, os demais operadores CTL podem ser construídos pela composição desses apresentados. Em nossa solução, maiores cuidados precisam ser tomados por serem algoritmos distribuídos e assíncronos. A construção de contra-exemplos, por exemplo, torna-se mais complexa em nossa solução do que em soluções clássicas centralizadas. Maiores detalhes sobre nossa implementação fogem do escopo da dissertação, porque o intuito aqui era apenas a obtenção de resultados experimentais em alguns modelos.

O Coordenador

Além do mais, para se permitir a comunicação entre algoritmos de verificação rodando em partições diferentes, o processo de verificação global é gerenciado por uma entidade central

procedure EG(s_i, β_i, π_i, Ψ) (E_i, T_i)
  if checkPreviousEvaluation(s_i, β_i, π_i, Ψ) **then**
    return (s_i, β_i, π_i, Ψ)
  end if
  $E_i.s_i \leftarrow$ *FALSE* {Supomos que EG é falsa em s_i.}
  (s_i, β_i, π_i, Ψ) $\leftarrow$ *subOP*.evaluate((s_i, β_i, π_i, Ψ), (E_i, T_i))
  if β_i = *UNDEFINED* **then**
    $\beta_i \leftarrow$ *UNDEFINED*; $\pi_i \leftarrow \pi_i \cup s_i$
    return (s_i, β_i, π_i, Ψ)
  end if
  if β_i = *TRUE* **then**
    checkPartition(s_i, β_i, π_i, Ψ) (E_i, T_i)
    $\pi_i \leftarrow \pi_i \cup s_i$
    $E_i.s_i \leftarrow$ *TRUE* {Supomos que EG é válida}
    for all s_r such that s_r *in* E_i and R(s_i,s_r) **do**
      (s_r, β_r, π_r, Ψ) = evaluate((s_r, β_r, π_r, Ψ), (E_i, T_i))
      if β_r = *TRUE* **then**
        $E_i.s_i \leftarrow$ *TRUE*
        $\pi_i \leftarrow \pi_i \cup s_i$; $\pi_i \leftarrow \pi_i \cup \pi_r$; $\beta_i \leftarrow$ *TRUE*
      else
        if β_r = *UNDEFINED* **then**
          $E_i.s_i \leftarrow$ *UNDEFINED*; $\beta_i \leftarrow$ *UNDEFINED*
          return (s_i, β_i, π_i, Ψ)
        end if
      end if
    end for
  else
    return (s_i, β_i, π_i, Ψ) {O resultado β_i é falso}
  end if
end

Algoritmo 13: O algoritmo distribuído CTL EG

procedure EU(s_i, β_i, π_i, Ψ) (E_i, T_i)
  if checkPreviousEvaluation(s_i, β_i, π_i, Ψ) **then**
    return (s_i, β_i, π_i, Ψ)
  end if
  $E_i.s_i \leftarrow$ *TRUE* {Supomos que EU é verdade em s_i.}
  (s_i, β_i, π_i, Ψ) $\leftarrow subOP_2$.evaluate((s_i, β_i, π_i, Ψ), (E_i, T_i))
  if β_i = *UNDEFINED* **then**
    $\beta_i \leftarrow$ *UNDEFINED*; $\pi_i \leftarrow \pi_i \cup s_i$
    return (s_i, β_i, π_i, Ψ)
  end if
  if β_i = *TRUE* **then**
    checkPartition(s_i, β_i, π_i, Ψ) (E_i, T_i)
    if β_r = *TRUE* **then**
      $E_i.s_i \leftarrow$ *TRUE*
      $\pi_i \leftarrow \pi_i \cup s_i$; $\pi_i \leftarrow \pi_i \cup \pi_r$; $\beta_i \leftarrow$ *TRUE*
    else
      (s_i, β_i, π_i, Ψ) $\leftarrow subOP_2$.evaluate((s_i, β_i, π_i, Ψ), (E_i, T_i))
      if β_i = *TRUE* **then**
        for all s_r such that s_r *in* E_i and R(s_i,s_r) **do**
          (s_r, β_r, π_r, Ψ) = evaluate((s_r, β_r, π_r, Ψ), (E_i, T_i))
          if β_r = *TRUE* **then**
            $E_i.s_i \leftarrow$ *TRUE*
            $\pi_i \leftarrow \pi_i \cup s_i$; $\pi_i \leftarrow \pi_i \cup \pi_r$; $\beta_i \leftarrow$ *TRUE*
          else
            if β_r = *UNDEFINED* **then**
              $E_i.s_i \leftarrow$ *UNDEFINED*; $\beta_i \leftarrow$ *UNDEFINED*
              return (s_i, β_i, π_i, Ψ)
            end if
          end if
        end for
      end if
    end if
    return (s_i, β_i, π_i, Ψ) {O resultado β_i é falso}
  end if
end

Algoritmo 14: O algoritmo distribuído CTL EU

procedure EX(s_i, β_i, π_i, Ψ) (E_i, T_i)
  $E_i.s_i \leftarrow$ *TRUE* {Supomos que EX é válida em s_i.}
  for all s_r such that s_r in E_i and R(s_i,s_r) **do**
    (s_r, β_r, π_r, Ψ) = evaluate((s_r, β_r, π_r, Ψ), (E_i, T_i))
    if β_r = *TRUE* **then**
      $E_i.s_i \leftarrow$ *TRUE*
      $\pi_i \leftarrow \pi_i \cup s_i$; $\pi_i \leftarrow \pi_i \cup \pi_r$; $\beta_i \leftarrow$ *TRUE*
    end if
  end for
  return (s_i, β_i, π_i, Ψ) {O resultado β_i é falso}
end

Algoritmo 15: O algoritmo distribuído CTL EX

chamada *coordenador*. Sua idéia principal é fazer uso da pilha global Γ para controlar o escalonamento e sincronização entre as partições. O Algoritmo 16 coordena esse trabalho:

O algoritmo segue o padrão de projeto Observer [GVJH98]. A função *waitForPartition* espera por uma notificação das partições no qual o estado seja *FINALIZADO* na grade computacional. Quando elas chegam no coordenador, seus resultados são intercalados levando-se em consideração os resultados prévios. Após isto, todos os estados exportados são encaminhados para a partição de verificação correspondente. Resultados de avaliação são criados durante este processo e as partições pendentes são colocadas em Γ para serem exploradas depois. Ao final, todas as partições remanescentes devem ter sido despachadas para executarem seus algoritmos. Este processo é repetido até que a condição de terminação seja satisfeita.

Detectando a Terminação da Verificação

O processo de verificação pode também ser feito de uma maneira centralizada. Isto ocorre quando todas as sub-fórmulas geradas da fórmula original ϕ tenham seu estado finalizado de acordo com sua partição do grafo.

Definition 3.4.3 $\forall P_i \in S$ *e* $\forall \phi_j \in Sub(\phi)$ *temos (*P_i*,*ϕ_j*).status = FINALIZADO.*

```
procedure VERIFICATIONMANAGER()
  (P_0,φ) ← forward(s_0, φ)
  push(Γ, (P_0,φ))
  dispatch(pop(Γ))
  while not checkForTermination() do
    waitForPartition()
    for all (P_n,φ_n) such that (P_n,φ_n).status = FINISHED do
      mergeResults(P_n,φ_n)
      if (P_n,φ_n).exported ≠ ∅ then
        push(Γ, (P_0,φ))
      end if
      Q ←
      for all s_i ∈ (P_n,φ_n).exported do
        (P_i,φ_n) ← forward(s_i, φ_n)
        insert(Q,P_i,φ_n)
      end for
      push(Γ, Q)
    end for
    Q ← pop(Γ)
    for all (P_n,φ_n) ∈ Q such that (P_n,φ_n).status = FINISHED do
      dispatch((P_n,φ_n))
    end for
  end while
end
```

Algoritmo 16: Gerenciando partições de verificação em uma entidade central

Estes algoritmos podem tirar mais vantagens de um ambiente distribuído se cada partição executa os turnos ao mesmo tempo. Quando o algoritmo alcança os estados que pertencem às bordas entre partições, a avaliação pára e retorna ao coordenador. Então, as partições donas dos estados borda são escalonadas para continuarem a avaliação. Naturalmente, existe uma ordem específica para avaliarmos as sub-fórmulas que compõem uma propriedade CTL. Problemas com comunicação excessiva também foram reportados. Estes e outros detalhes estão em análise e algoritmos baseados na teoria do ponto fixo como os apresentados em

[Mat87] são sugestões para serem analisadas.

3.4.2 Checando Propriedades

A implementação atual do protótipo também permite aos usuários verificarem a existência de propriedades de deadlock, segurança e alcançabilidade de uma maneira *on-the-fly*, ou seja, durante a construção do espaço de estados.

Definimos genericamente essas propriedades para um melhor entendimento do que será experimentado. Uma propriedade de deadlock ocorre quando uma partição explora um estado no qual não existe uma transição habilitada, ocorrendo um travamento global do sistema. Uma propriedade p é de segurança se p é um predicado que deva ser satisfeito em todos os estados alcançáveis. Isto é equivalente a verificar se a negação de p não é alcançável no sistema. Assim, quando um estado s, no qual a negação de p vale, é calculado, s é reportado como um estado de contra-exemplo. Uma propriedade q é de alcançabilidade se q é um predicado que deva ser satisfeito ao menos em um estado alcançável do sistema. Então, o primeiro estado alcançado que satisfaz q é reportado como um estado de testemunho.

Os estados reportados são usados para compor caminhos que provam os resultados da verificação. Por exemplo, quando um deadlock é reportado, a ferramenta constrói um caminho do estado inicial até o estado em deadlock. No intuito de se permitir a composição de caminhos através de diferentes partições que compõem o espaço de estados completo, um esquema específico para identificação dos estados é mantido. Sempre quando um estado é exportado, ele leva seu identificador para a partição destino. Este identificador será mantido juntamente com o novo identificador que é atribuído pela partição original. Esta abordagem difere do método que é mais empregado em trabalhos similares, pois eles fazem uso de *backtracking* para compor os contra-exemplos [GMS01].

Considerando o trabalho executado por cada tarefa, o processo de verificação requer $O(|V| + |E|)$ operações mais o custo imposto pelo processo de coordenação, no qual é $O(|\chi|)$. Para a verificação *on-the-fly*, a construção do caminho ainda não é feita em um ambiente distribuído. Ela é feita após o espaço de estados ter sido calculado, na mesma máquina na qual o coordenador executa. Uma possível maneira de construir os caminhos em uma abordagem distribuída é criar uma tarefa para cada estado reportado. O tempo para a construção dos caminhos poderia ser reduzido se cada estado reportado fosse processado por

uma máquina remota em paralelo.

3.5 Conclusões da Verificação Distribuída

Aqui, explanamos o que foi produzido em termos de verificação distribuída de modelos sobre plataformas de grades computacionais. Embora os algoritmos sejam apresentados de uma forma genérica, o que realmente é nossa intenção, eles foram implementados para serem executados na plataforma Ourgrid e seus resultados são apresentados na próxima seção.

O grupo agora possui um núcleo de geração de espaços de estados e verificação de modelos distribuído, que foram avaliados experimentalmente. Uma tendência é que esses algoritmos e as lições aprendidas sejam reusadas em soluções mais robustas com propósitos científicos e talvez industriais.

Capítulo 4

Resultados Experimentais

Esta seção apresenta os resultados experimentais produzidos nesta pesquisa. No intuito de avaliar os prós e os contras da abordagem proposta, alguns modelos foram analisados. Com uma versão estável do protótipo definida, observamos o seu comportamento e características quando abordamos diferentes problemas de modelos concorrentes com variadas complexidades. Com isso, temos o intuito de testarmos exaustivamente a ferramenta para identificarmos seus limites e concluirmos se estamos trilhando o caminho correto rumo a uma solução eficiente que amenize o problema da explosão de espaço de estados para o modelador de sistemas que possui poucos recursos computacionais.

Classificamos as execuções experimentais em três modos, que foram executados para todos os modelos:

- Experimentos Centralizados. Foram realizados em uma estação de trabalho *AMD Athlon(tm) XP 1500+* com CPU de 1350 MHz, 1GBytes de memória total e tendo como sistema operacional a distribuição Linux FEDORA CORE 5. Foram observadas propriedades de representação de grafos em memória, tempo de processamento de uma única CPU, uso de memória virtual e outros mais.

- Experimentos Distribuídos Restritos ao Laboratório. Utilizamos a solução Ourgrid, porém restringimos os recursos utilizados pela solução ao domínio do laboratório do nosso grupo de pesquisa, o Laboratório de Métodos Formais da UFCG, que conta com dezessete máquinas com configurações aproximadas à apresentada nos experimentos centralizados. Esse tipo de experimento visou apenas a obtenção de ganho de

desempenho (*speed-up*) com relação aos experimentos centralizados. O particionamento deve ser bem direcionado, de forma que aproveite de maneira eficaz os recursos disponíveis internamente e evite gargalos, pois, a depender da aplicação, pode ocorrer de existirem mais tarefas pendentes do que máquinas disponíveis para execução, adicionando atrasos consideráveis ao tempo global de resolução do problema.

- Experimentos Distribuídos na Comunidade. Podemos contar com até aproximadamente quinhentos recursos distribuídos por todo o planeta com configurações bastante heterogêneas. A disponibilidade desses recursos está relacionada com a ociosidade deles, então não temos nenhuma garantia de acesso a eles, principalmente em horários de expediente nos centros de pesquisa possuidores desses recursos. Neste caso, o principal objetivo é ganho em escala. A comunicação entre domínios diferentes não nos dá garantia das qualidades desejadas. A falta de garantia de acesso aos recursos faz com que experimentos diferentes executem com recursos provavelmente diferentes também.

O principal objetivo desta atividade é mapear todas as características da solução, em termos de tempo de processamento, consumo de memória, tempo de comunicação, alocação de recursos e modo de particionamento. Nesta etapa, nos concentramos na construção de modelos em formalismos baseados em redes de Petri, tais como redes de Petri lugar/transição, redes de Petri coloridas e redes de Petri orientadas a objetos. Cada experimento foi executado várias vezes, porém nesta dissertação filtramos informações desnecessárias apresentando apenas a média geral destes experimentos. Elaboramos funções de particionamento baseando-nos em algum conhecimento prévio dos modelos que possibilitassem o maior paralelismo possível entre as partições.

Os modelos escolhidos para essa bateria de testes inicial da ferramenta foram: protocolos Stop-and-Wait [GB06] e IP móvel [GS03], padrão Itinerary para agentes móveis [Lim04] e o jantar dos filósofos de Dijkstra [Dij02]. Cada modelo desses apresenta características peculiares em seu comportamento, possibilitando estudos diferentes relacionados ao particionamento do grafo de ocorrência que o descreve. As informações foram devidamente registradas usando o gerador de relatórios da ferramenta, que fornece valiosas informações sobre o processo, tais como tempo de geração de cada partição, tempo de geração global do

sistema, número de nós e arestas gerados por cada partição e somatório desses componentes a nível global, além de fornecer a quantidade de comunicações entre as partições (processo de exportação e importação de estados). Em determinados experimentos, estivemos preocupados em registrar quão grandes são os modelos que podemos verificar usando recursos distribuídos. Para tanto, realizamos experimentos nos quais o número de itens do modelo muda em conjunto com o número de máquinas utilizadas.

As Tabelas 4.1 e 4.2 ilustram, para cada modo, o número de experimentos realizados com geração e verificação do espaço de estados de cada modelo em diferentes escalas.

Modelo/Modo	Centralizado	Distribuído Lab	Distribuído Comunidade	Total
Stop-and-Wait	35	55	70	160
IP Móvel	35	45	55	135
IP Retorno	55	55	55	165
Filósofos	35	45	50	130
Itinerário	40	40	40	120
Total	200	240	270	710

Tabela 4.1: Total de experimentos de geração de espaços de estados para cada modelo e modo

Modelo/Modo	Centralizado	Distribuído Lab	Distribuído Comunidade	Total
Stop-and-Wait	30	60	80	170
IP Móvel	25	20	30	75
IP Retorno	10	10	12	32
Filósofos	40	60	60	160
Itinerário	5	5	5	15
Total	110	155	187	452

Tabela 4.2: Total de experimentos de verificação para cada modelo e modo

A Tabela 4.3 provê para cada modelo a maior soma de nós e arcos possibilitados de lidar durante o processo de geração dos espaços de estados. Esta tabela é importante para explicitar como a solução escala em relação à versão centralizada existente. Este ganho, conforme mencionado, é o principal benefício da solução.

Os resultados, de uma maneira geral, são apresentados em tabelas e gráficos ilustrativos. Cada tabela deve prover informações sobre o número de componentes concorrentes do sis-

tema e partições, quantidade de nós e arcos do grafo de ocorrência trabalhado e uma média geral do total de execuções do mesmo experimento em um dado modelo. Com relação aos gráficos, eles evoluem no eixo X de acordo com o incremente do número de componentes concorrentes nos modelos, que implica em modelos diferentes sendo avaliados e no eixo Y de acordo com o tempo gasto no processo para cada modelo. Lembrando que nenhum resultado apresentado aqui pode ser comparado com alguma ferramenta escolhida como estado da arte em verificação de modelos, pelos limites já discutidos.

Modelo/Modo	Centralizado	Distribuído Lab	Distribuído Comunidade
Stop-and-Wait	450.031	2.250.151	9.000.100
IP Móvel	2.158.993	11.251.107	22.502.225
IP Retorno	1.242.241	2.912.217	2.912.217
Filósofos	95.997	1.284.639	4.710.088
Itinerário	1.206.064	1.206.064	1.206.064

Tabela 4.3: Maior número de nós e arcos que a solução conseguiu lidar para cada modelo

4.1 Modelo do Protocolo Stop-and-Wait

4.1.1 Descrição

A transmissão Stop-and-Wait é bastante simples do ponto de vista de confiabilidade e é adequada a protocolos de comunicação não muito complexos. Transmitem-se Unidades de Dados do Protocolo (UDP) e é aguardada uma resposta. Um ACK é enviado pelo receptor no caso da UDP ter sido recebida corretamente.

Aproveitamos o fato de haver paralelismo explícito entre os diversos transmissores que compõem a rede para servir como nosso primeiro modelo experimental. Para uma maior simplicidade, embora dispuséssemos de uma linguagem tipada e outra até com propriedades da orientação a objetos para modelar o problema, resolvemos implementá-lo como uma rede de Petri lugar/transição que dita todo o comportamento do sistema apenas por questões de simplicidade e ser possível de expressar as propriedades que já desejávamos. O sistema modelado está representado na Figura 4.1:

Essas características foram incluídas nesse modelo pretendendo resolver o problema do

particionamento, de modo que o equilíbrio temporal e espacial, além da localidade, sejam endereçados de uma maneira ótima. Assim, embora simples, este modelo previne que a avaliação desta aplicação através do uso de grades computacionais sofra por causa dos problemas enfrentados com particionamento.

Figura 4.1: Rede responsável pela descrição do comportamento do protocolo Stop-and-Wait

4.1.2 Particionamento e Características do Grafo de Ocorrência

A coloração diferente em algumas regiões da rede na Figura 4.1 indicam os componentes concorrentes identificados por nossa função de particionamento. Assim, certamente os lugares pertencentes a essas regiões estarão disputando o disparo de uma transição conflitante, permitindo sempre que apenas um desses componentes esteja ativo em todos os momentos.

Esta característica do modelo origina o grafo apresentado na Figura 4.2 do lado esquerdo e nos permite gerar uma função de particionamento que aplicando-a a esse grafo, divida-o em partições de acordo com as regiões acinzentadas apresentadas no lado direito ainda da Figura 4.2. Aqui, estamos fazendo análise do melhor caso, porém, isso não implica que o uso ou combinação de outras técnicas não possam identificar resultados mais satisfatórios. Como exemplo, para este grafo, devido à similaridade entre suas partições, poderia ser feita uma exploração das partições por equivalência.

Figura 4.2: Visão geral grafo de ocorrência do Stop and Wait antes e depois do particionamento

4.1.3 Experimentação com Geração Centralizada

Iniciamos as experimentações simulando e gerando espaços de estados com a versão centralizada da ferramenta, que utiliza uma única CPU. Os resultados estão descritos na Tabela 4.4.

Transmissores	Nº Estados	Nº Arcos	Tempo Médio
2	30003	30002	5s
3	45004	45003	9s
4	60005	60004	14s
6	90007	90006	28s
10	150011	150010	63s
15	225016	225015	937s
20	x	x	x

Tabela 4.4: Execução centralizada do Stop-and-Wait

Observamos que o incremento em tempo relacionado ao número de nós do grafo gerado é quase linear. Porém, não conseguiremos progresso com uma CPU quando precisarmos tratar a maioria dos sistemas adotados hoje pela indústria. Os resultados mostram que quando ultrapassamos a faixa dos quinhentos mil componentes do grafo de ocorrência neste exemplo, fica impossível progredir, ocorrendo parada do processo de geração.

4.1.4 Experimentação com Geração Distribuída no Laboratório

A Tabela 4.5 indica resultados de distribuição utilizando apenas máquinas internas do laboratório, com número de partições sempre iguais ao número de transmissores.

Transmissores	Nº Estados	Nº Arcos	Tempo Médio
2	30003	30002	10s
3	45004	45003	12s
4	60005	60004	18s
6	90007	90006	20s
10	150011	150010	35s
15	225016	225015	52s
20	300021	300020	79s
30	450031	450030	140s
40	600041	600040	223s
50	750051	750050	314s
75	1125076	1125075	635s

Tabela 4.5: Execução distribuída interna ao laboratório do Stop-and-Wait

Observe que conseguimos gerar grafos dezenas de vezes maiores do que os que a versão centralizada apresentada anteriormente conseguiu e em um tempo aceitável. Problemas de falta de recursos para resolver todas as tarefas pendentes em um dado momento existem e não temos como alocar réplicas para rodarem em paralelo com computadores considerados lentos em relação aos demais no intuito de melhorar desempenho. Ultrapassamos sem problemas a barreira dos dois milhões de componentes do grafo de ocorrência e esse processo só não prosseguiu porque o coordenador tornou-se um gargalo na responsabilidade de criar tarefas e interpretar resultados gerados pelas máquinas da grade. Otimizações no intuito de aliviar essa sobrecarga do mestre ainda devem ser implementadas.

A Tabela 4.6 detalha melhor a execução do experimento com seis transmissores. Ilustramos para cada partição o número de estados exportados, o número de vezes que foi executada, os estados e os arcos processados e o tempo total das execuções. Quando afirmamos que esse modelo é ideal, nos referimos ao fato de a maioria das partições processarem o mesmo número de nós e arcos, o que caracteriza equilíbrio espacial, e possuir um baixo número de estados exportados, que é o princípio da localidade.

Partições	Exportados	Nº Execuções	Nº Estados	Nº Arcos	Tempo Médio
0	0	1	15001	15000	2.808s
1	0	1	15001	15000	2.186s
2	0	1	15001	15000	5.277s
3	0	1	15001	15000	2.440s
4	0	1	15001	15000	2.199s
5	0	1	15001	15000	2.245s
6	6	1	7	6	0.029s

Tabela 4.6: Equilíbrio entre partições com 6 transmissores

O histograma na Figura 4.3 ilustra graficamente o equilíbrio obtido com nossa função de particionamento durante as execuções.

Figura 4.3: Número de estados processados por cada partição no Stop-and-Wait

4.1.5 Experimentação com Geração Distribuída na Comunidade

O experimento que se na Tabela 4.7 segue utiliza todos os recursos da comunidade buscando obter a maior escala possível em termos do grafo gerado. Variamos o número de transmissores com 15000 UDPs e usando o buffer do tamanho que consideramos ideal a depender

da situação do sistema. Este resultado dá uma idéia inicial do quão poderoso pode ser nosso método. Nos aproximamos dos 10 milhões de estados sem necessidades de cuidados extras. Novos experimentos poderiam ser realizados visando obtermos um número maior ainda de componentes gerados, já que a ferramenta não apresentou qualquer sintoma de incapacidade de lidar com modelos maiores.

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	1	90004	90002	36s
3	2	135006	135003	42s
4	∞	180008	180004	46s
6	∞	270012	270006	106s
8	∞	360016	360008	128s
10	∞	450020	450010	149s
20	∞	900020	900000	443s
30	10	1350030	1350000	1598s
50	10	2250000	2250050	2570s
100	10	4500000	4500100	6337s

Tabela 4.7: Execução distribuída na comunidade do Stop-and-Wait

Observe que à medida que o modelo vai ficando maior, faz-se necessário o uso de um buffer de limite que evite que partições gastem muito tempo com a geração correndo riscos de estouro de memória nas máquinas da grade e que não sejam produzidos grafos além da capacidade da rede, para que possam ser transportados entre máquinas da grade e a máquina local.

A Tabela 4.8 fixa o modelo em 20 transmissores de 5000 UDPs, variando-se apenas o número de partições do experimento buscando obter um ganho de desempenho por linha.

Estes resultados mostram que a escolha de um número inconsistente de partições e de tamanho de buffer podem levar a resultados bastante inconsistentes. Idealmente devemos ter um número de partições aproximando-se do número de componentes concorrentes identificados no sistema, além de um buffer que permita estressar a máquina ao seu limite sem comprometer a transmissão do grafo de ocorrência pela rede.

O gráfico na Figura 4.4 ilustra uma comparação entre as duas tabelas apresentadas previamente sobre execução dos experimentos. Este primeiro exemplo objetiva ilustrar um caso ótimo, que foram bastante evidenciados por esse gráfico nos quesitos desempenho e escala.

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	1	300021	300020	1880s
4	3	300021	300020	1448s
6	5	300021	300020	3356s
8	4	300021	300020	3241s
10	4	300021	300020	2315s
20	4	300021	300020	1106s
20	9	300021	300020	948s

Tabela 4.8: Variação de buffer no Stop-and-Wait com 20 transmissores

Há ganhos significativos em escala nos experimentos distribuídos em relação ao centralizado.

Figura 4.4: Comparação entre experimentos centralizados e distribuídos no Stop-and-Wait

4.1.6 Experimentação com Verificação Distribuída na Comunidade

As Figuras 4.5, 4.6 e as Tabelas 4.9 e 4.10 ilustram alguns resultados da verificação de propriedades no verificador de modelos CTL distribuído também aderindo à comunidade Ourgrid. No gráfico na Figura 4.5, estamos usando um quantificador universal para checarmos se eventualmente todos os pacotes unitários (UDP - Unit Data Protocol) são entregues ao seu destino. Isto, em um nível mais próximo do modelo, significa que utilizamos o operador AF

de CTL e a propriedade se dá verificando se a soma das fichas dos lugares *Received*, que armazena pacotes entregues, e *Dt1*, que armazena os pacotes a serem transmitidos, se iguala ao número de pacotes que seriam transmitidos pelo sistema. O gráfico apresentado na Figura 4.6 representa os resultado de quando aplicamos o operador EG à propriedade de um transmissor ficar ocioso sem transmitir. Estamos procurando saber se algum processo fica receber recursos (starvation), ou seja, checando se existe um emissor que nunca pode operar. Esse exemplo torna-se mais longo, consumindo mais tempo, porque é testada exaustivamente a construção de um exemplo da maior profundidade possível do grafo.

Partições	Nº Estados	Nº Arcos	Tempo Centralizado	Tempo Distribuído
2	30003	30002	2.8s	10s
3	45004	45003	6.1s	17s
4	60005	60004	12.3s	18s
6	90007	90006	32.1s	27s
10	150011	150010	230.9s	32s
15	225016	225015	467.5s	39s
20	300021	300020	X	49s
30	450031	450030	X	69s
40	600041	600040	X	101s
50	750051	750050	X	135s
75	1125076	1125075	X	251s
100	1500101	1500100	X	372s

Tabela 4.9: Execução centralizada e distribuída da entrega dos UDP's no Stop-and-Wait

4.1.7 Conclusões sobre Stop-and-Wait

Estas tabelas e gráficos explicitam as vantagens adquiridas por essa classe de modelos ao adotar nossa solução. Os ganhos de desempenho e escala são de uma ordem bastante elevada também na verificação. Isto se deve ao particionamento ótimo proporcionado pelo modelo, ideal para uma plataforma com filosofia Bag-of-Tasks de comunicação.

Figura 4.5: Checando se todos os UDPs são entregues ao seu destino

Figura 4.6: Procurando por uma condição de eqüidade, se todos os transmissores executam

4.2 Modelo Protocolo IP Móvel

4.2.1 Descrição

O segundo modelo é um cenário um pouco mais realístico do protocolo IP Móvel, que é um padrão que provê mecanismos para manter a conectividade de hosts móveis quando eles mi-

Partições	Nº Estados	Nº Arcos	Tempo Centralizado	Tempo Distribuído
2	30003	30002	8s	19s
3	45004	45003	16s	27s
4	60005	60004	24s	33s
6	90007	90006	521s	69s
10	150011	150010	2000s	111s
15	225016	225015	8000s	161s
20	300021	300020	∞	179s
30	450031	450030	∞	211s
40	600041	600040	∞	223s
50	750051	750050	∞	314s
75	1125076	1125075	∞	635s
100	1500101	1500100	∞	1048s

Tabela 4.10: Execução centralizada e distribuída da busca por eqüidade no Stop-and-Wait

gram através de redes diferentes. Estudos sobre o protocolo IP Móvel podem ser encontrados em [RGdF+04; DK99].

O protocolo do IP Móvel foi projetado com o intuito de manter hosts móveis com conectividade IP, mesmo quando eles estiverem fora da sua rede local. O protocolo utiliza-se de dois endereços para manter a conectividade: o endereço da rede original e o novo endereço da rede estrangeira que é obtido quando ele migra para uma nova rede.

No modelo apresentado na Figura 4.7, inicialmente o dispositivo está na sua rede local, lugar InLocalHome, e ele pode receber diversos UDPs através do disparo da transição RecLoc. Ao migrar para uma rede externa, ele passa pelos processos de requisitar um registro (*AskREg*) e receber o registro (*RecReg*), estando então no agente estrangeiro. Ao receber mais dados, ele passa por um processo de tunelamento (*Tunn*) e destunelamento (*DeTunn*) do dado até ele ser corretamente recebido pelo agente da rede estrangeira (*RecFA*).

4.2.2 Particionamento e Características do Grafo de Ocorrência

A idéia geral do particionamento segue o que desenvolvemos no protocolo Stop-and-Wait. O que é diferenciado nesse modelo é o tempo seqüencial antes da criação das partições e um número maior de comunicação entre as partições. Porém, o grafo de ocorrência desse modelo continua sendo ideal para uma grade de filosofia *bag-of-tasks* de comunicação.

Figura 4.7: Rede lugar/transição responsável pela descrição do comportamento do protocolo IP Móvel

4.2.3 Experimentação com Geração Centralizada

Nos centralizados, não obtivemos o crescimento linear do tempo em relação ao número de componentes do grafo de ocorrência como tinha acontecido no modelo anterior. Isso se deve ao fato de haver uma complexidade maior no número de componentes fortemente conectados do grafo de ocorrência. Freqüentemente são gerados estados que já foram explorados e sua consulta à tabela de nós já explorados torna-se mais cara à medida que o grafo é gerado. Isso implica também em uma maior complexidade na função de particionamento durante os experimentos distribuídos. Seus resultados estão descritos na Tabela 4.11.

4.2.4 Experimentação com Geração Distribuída no Laboratório

Na Tabela 4.12 estão os dados da geração do espaço de estados usando a plataforma de grades computacionais apenas com as máquinas do laboratório:

Os resulados da segunda tabela são satisfatórios, pois, embora estejamos com um modelo de maior complexidade em mãos, continuamos sem as dificuldades de lidar com grafos de grandes ordens e o processo de geração continua apresentando números aceitáveis de tempo.

Qtd Agentes	Nº Estados	Nº Arcos	Tempo Médio
2	107415	342678	81s
3	161007	513807	127s
4	214599	684936	177s
6	321783	1027194	4134s
8	428967	1369452	48383s
10	536151	1622842	226122s
15	X	X	X

Tabela 4.11: Geração centralizada do IP Móvel

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	∞	107877	342678	82s
3	∞	161388	513807	135s
4	∞	215523	684936	194s
6	∞	323169	1027194	252s
8	∞	430815	1369452	440s
10	∞	538461	1711710	997s
20	∞	1076691	3423000	1513s
30	∞	1614921	5126940	2262s
50	∞	2692425	8558682	5788s

Tabela 4.12: Geração distribuída no laboratório do IP Móvel

Um detalhe a ser observado, é que nunca precisamos limitar o tamanho do buffer para alguma quantidade finita, pois como o particionamento não apresenta um nível tão alto de localidade, temos que concluir prematuramente tarefas nas máquinas da grade por não ser possível a continuação da geração sem haver exportação dos estados a serem explorados em outras partições.

O próximo experimento, descrito na Tabela 4.13, mostra que durante o particionamento, quanto mais pedaços disjuntos do grafo pudermos gerar sem comunicação, melhor será o nosso desempenho do tempo global de geração. Buffers pequenos não são recomendados por sempre implicarem em pouca localidade. Para esse modelo do IP Móvel, observamos que um buffer pequeno combinado com um grande número de partições exige muita comunicação (retorno ao coordenador e criação de novas tarefas delegadas às máquinas da grade), proporcionando grandes tempos de geração, o que não é um resultado muito satisfatório em nosso caso. O ideal é que tenhamos um número de partições razoável com o máximo de equilíbrio e localidade possível com um buffer de tamanho maior o possível, já que sabemos que este modelo possui estados que tendem a se comunicar naturalmente com outras partições sem a necessidade de um buffer para limitar o processo de geração em uma partição.

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	1	108690	165817	10480s
4	3	61575	161465	18506s
6	5	62997	165070	8579s
8	4	61803	163099	10328s
10	4	61845	153915	20540s
20	4	50762	138322	2527s
20	9	33709	94474	1387s

Tabela 4.13: Geração distribuída do IP Móvel variando-se buffer

4.2.5 Experimentação com Geração Distribuída na Comunidade

Complementando os experimentos com geração distribuída do espaço de estados do IP Móvel, temos a Tabela 4.14 que ilustra a execução com máquinas da comunidade.

O gráfico na Figura 4.8 ilustra explicitamente as observações mencionadas anteriormente. Este gráfico está com uma escala maior que as demais, mas percebe-se que ini-

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	∞	107877	342678	97s
3	∞	161007	513807	146s
4	∞	215523	684936	221s
6	∞	323169	1027194	294s
8	∞	430815	1369452	540s
10	∞	538461	1711710	1348s
20	∞	1076691	3423000	2001s
30	∞	1614921	5126940	4162s
50	∞	2692425	8558682	10832s
100	∞	5384856	17117369	19641s

Tabela 4.14: Geração distribuída na comunidade do IP Móvel

cialmente as gerações distribuídas não conseguem serem mais velozes que a geração centralizada. Porém, devido aos problemas de falta de memória, o processo centralizado não prossegue, perdendo em muito no quesito escalabilidade para a gerações que utilizam nossa solução. Em resumo, conseguimos ir bem além da geração interna ao laboratório no quesito escala, porém sempre estivemos abaixo em desempenho.

Figura 4.8: Comparação da experimentação centralizada contra distribuída no IP Móvel

4.2.6 Verificação On-the-Fly de Propriedades no Procolo IP Móvel

Os gráficos nas figuras a seguir ilustram resumidamente a execução de algumas buscas por propriedades comportamentais típicas de modelos de sistemas concorrentes descritos em redes de Petri.

Segurança

A propriedade de segurança em um lugar p_i em um rede de Petri, refere-se a saber se $p_i \in P$, então $\forall M \in Alcancaveis(M_0)$, temos $M(p_i) \leq 1$. Nesse caso, nós procuramos saber se o lugar *InForeignAgent* é seguro. Essa é uma outra maneira de testarmos se ocorreu a migração de mais de um nó.

Alcançabilidade

Essa propriedade indica a possibilidade de uma determinada marcação M' ser atingida após o disparo de um número finito de transições a partir da marcação inicial. Dizemos que M' é alcançável se e somente se pertencer ao conjunto das marcações acessíveis a partir de M.

No protocolo IP Móvel nós procuramos por uma situação em que mais de um nó tenha migrado, que é falsa em nosso modelo, e significa que o número de lugares iniciando por *ForAg* com marcação nula seja maior que um. Também procuramos por uma marcação em que todas as migrações tenham sido bloqueadas, permanecendo todos os agentes nas suas redes de origem, nesse caso, esse estado existe e pôde ser alcançado.

Na experimentação da Figura 4.9, percebemos que os tempos de busca são muito altos, quase comparáveis com a geração. Isto se deve à necessidade de uma busca completa no grafo de ocorrência que estava sendo gerado. Naturalmente, a busca centralizada para no limite de geração do grafo que já foi apresentado.

A verificação na Figura 4.10 não exige a busca completa no grafo e geralmente apenas uma partição acha essa propriedade. Então, a princípio temos uma elevação de tempo na busca distribuída devido aos custos de criar e coletar as tarefas. A partir de um certo ponto, esse tempo de busca praticamente se estabiliza não importando o tamanho do modelo sob verificação. Este é um tipo de verificação bastante adequado para nossa solução.

Figura 4.9: Procurando situação em que mais de um nó migre: verificação centralizada contra distribuída

Figura 4.10: Procurando situação em que todas migrações sejam bloqueadas: verificação centralizada contra distribuída

4.2.7 Modelo Protocolo IP Móvel com Retorno

A Figura 4.11 ilustra uma pequena alteração no modelo no intuito de dificultar sua função de particionamento. É inserido o conceito de retorno dos agentes em uma rede estrangeira para sua rede local. Assim, nos distanciamos mais ainda de um grafo de ocorrência propício a ser explorado por aplicações Bag-of-Tasks, pois esse retorno implica em um aninhamento maior entre as partições identificadas previamente. Obviamente, nosso ganho de desempenho sofre um pequeno decremento em relação ao modelo anterior do mesmo protocolo. Na Figura 4.12 é apresentada a complexidade que é inserida no grafo de ocorrência do sistema após a inserção deste conceito. Observa-se que o número de arcos que atravessam partições é aumentado significativamente.

Figura 4.11: Descrição do protocolo IP Móvel adicionando-se o conceito de retorno à rede local

Nos experimentos com este modelo, fixamos o número de fichas no *CorrespondentNode* em dez. Isto significa que para que cada nó em sua rede local pode combinar um número de dez migrações para a rede estrangeira com os demais nós. O conceito de retorno faz esse número de combinações crescer em uma ordem muito maior de acordo com o número de nós, o que dificulta bastante o particionamento. Estes experimentos foram realizados utilizando-se apenas máquinas do laboratório.

Figura 4.12: Grafo de ocorrência do IP Móvel com o conceito de retorno e seu particionamento

A função de particionamento intuitiva seria de acordo com as regiões coloridas apresentadas na Figura 4.11, pois foi a estratégia adotada no IP Móvel anterior. Aqui, não conseguimos de maneira alguma viabilizar a paralelização dos algoritmos e devido a isso somos obrigados a mostrar uma outra característica do nosso método quando nos deparamos com essa situação. Quando não for possível obtermos uma função de particionamento satisfatória, podemos ao menos simular uma geração seqüencial em disco. Isto foi feito neste modelo da seguinte maneira: guiamos nosso particionamento de acordo com o número de fichas no lugar *CorrespondentNode* seqüencialmente. Assim, à medida que o número de fichas for decrescendo nesse lugar, estaremos mudando de partição. Isso faz com que partes do grafo de ocorrência sejam computadas seqüencialmente e integradas aos resultados em disco, evitando assim problemas imediatos de estouro de memória.

As Tabelas 4.15 e 4.16 ilustram os dados dos experimentos centralizados e o experimento distribuído com recursos apenas do laboratório, onde tentamos usar o método de particionamento sugerido por nossa solução.

Como se pode perceber, não conseguimos de forma alguma viabilizar o uso de nossa solução nesse modelo. Assim, tivemos que recorrer à simulação da geração seqüencial em disco, que é exposta na tabela seguinte.

A Tabela 4.17 e o Histograma 4.13 apresentam um pequeno resumo da qualidade do nosso processo seqüencial. Tomamos o modelo com dez agentes e como se pode observar, não houve equilíbrio espacial, pois as partições variam de 341 até 68541 estados. A variação

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
1	∞	12012	40040	5s
2	∞	21021	72072	10s
3	∞	30030	104104	20s
4	∞	39039	136136	42s
6	∞	57057	200200	54s
8	∞	75075	264264	87s
10	∞	93093	328328	132s
12	∞	111111	392392	198s
15	∞	138138	488488	296s
20	∞	183183	648648	492s
30	∞	273273	968968	7855s

Tabela 4.15: Geração centralizada do IP Móvel com retorno

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
1	∞	18018	40040	23s
2	∞	36476	72072	46s
3	∞	52651	104108	67s
4	∞	81730	136136	101s
6	∞	323169	200200	322s
8	∞	430815	264264	929s
10	∞	108727	328328	4526s
20	∞	195993	648648	13099s
30	∞	292383	968968	22401s
40	∞	1614921	1297296	372721s

Tabela 4.16: Geração distribuída no laboratório do IP Móvel com retorno

do número de estados exportados também foi alta e cada partição foi executada apenas uma vez seqüencialmente de acordo com a tabela. O tempo total de geração havia sido 4526.681 segundos e o tempo total de processamento nas partições somou-se apenas 1682.63 segundos. Percebemos que houve um tempo muito grande gasto com o custo de comunicação de 2844.051 segundos, que é inerente da nossa solução. Lembrando que essa função de particionamento poderia ter sido melhorada com uma investigação maior do modelo.

Partições	Exportados	Nº Execuções	Nº Estados	Nº Arcos	Tempo Médio
1	155	1	341	499	0.5s
2	1085	1	2635	5010	17s
3	3906	1	9982	21581	205s
4	10230	1	26970	62868	1400s
5	0	1	68541	238370	58s

Tabela 4.17: Equilíbrio entre partições no IP Móvel com retorno

Figura 4.13: Número de estados processados por cada partição

4.2.8 Conclusões sobre IP Móvel

A primeira versão do modelo apresentada aumentou em um nível a complexidade envolvida nos experimentos, e já nos aproximamos do perfil genérico que apresentarão os experimentos de agora em diante: raríssimos ganhos em desempenho enquanto a complexidade do modelo permanece aceitável para uma única CPU. A partir daí, o processo centralizado *trava*, nos restando apenas a solução distribuída que permite lidar com modelos mais complexos, porém com tempos de processamento bastante elevados.

As soluções distribuídas também se diferenciaram. Houve um pequeno ganho em desempenho quando utilizávamos apenas máquinas do laboratório, devido ao custo de comu-

nicação e um pequeno ganho em escala para a comunidade por proporcionar mais recursos para a geração.

No experimento com retorno, os níveis de perda de desempenho ficaram bem mais acentuados e praticamente não houve ganho em escala quando tentamos usar nossos métodos padrões de particionamento, possuindo este modelo as piores características para este tipo de solução. Assim, tivemos que recorrer à simulação de geração em disco distribuída para amenizar essas deficiências.

4.3 Modelo do Jantar dos Filósofos de Dijkstra

4.3.1 Descrição

Nosso próximo modelo a ser analisado é o clássico Jantar dos Filósofos de Dijkstra. Nesse problema, temos ao redor dessa mesa vários filósofos e cada filósofo assume dois estados: pensando ou comendo. Ao lado de cada filósofo tem um garfo que é compartilhado com seu vizinho, ou seja, cada filósofo tem acesso a um garfo esquerdo e a um garfo direito. Um garfo pode estar livre ou em uso e, para passar a comer um filósofo precisa dos dois garfos. Depois de algum tempo não-determinístico, o filósofo para de comer e libera os garfos. Seus componentes estão dispostos ao redor de um mesa circular na Figura 4.14, com P para filósofo e F para garfo.

Figura 4.14: Visão geral do modelo com 5 filósofos e 5 garfos

A Figura 4.15 ilustra o modelo geral do sistema. Temos duas entidades no sistema,

Filósofo e Garfo, e um Filósofo está associado a dois Garfos, que no caso é distinto por garfo direito e garfo esquerdo.

Figura 4.15: Diagrama de classes do sistema do jantar dos filósofos

Figura 4.16: Rede de Petri colorida que descreve o comportamento do filósofo

Utilizamos a ferramenta J-Mobile [Sil04] para implementar as classes e as redes de Petri desse modelo RPOO conforme proposto por Guerrero em sua tese [Gue02]. Esta ferramenta não possuía muitas sofisticações relacionadas à formas eficientes de representação, o que implicou em nós do grafo que consumiam muita memória para serem representados. As classes Filósofo e Garfo estão ilustradas nas Figuras 4.16 e 4.17 respectivamente. Resumidamente,

Figura 4.17: Rede de Petri colorida que descreve o comportamento do garfo

um garfo (chopstick) pode estar nos estados *disponível* e *ocupado* e um filósofo assume os estados *pensando* ou *comendo*.

O modelo foi construído de uma forma genérica e abstrata para que a quantidade de componentes no sistema pudesse ser alterada facilmente para observarmos questões envolvendo escala. Devido à enorme complexidade do processo de verificação da corretude desse grafo por um ser humano, o validamos aproveitando-se do fato de existir uma equivalência entre modelos RPOO e modelos em redes de Petri coloridas (CPN) no qual garante que o número de estados e arcos do espaço de estados gerado na ferramenta Design/CPN [Jen92] deve ser exatamente igual ao do modelo no J-Mobile.

4.3.2 Particionamento e Características do Grafo de Ocorrência

A seguir ilustramos os nove estados que foram gerados pela nossa ferramenta para o modelo com dois filósofos. Temos que cada filósofo é representado por *P* (do inglês philosopher)

acrescido de seu índice no modelo. O mesmo vale para os garfos com a letra *F* (do inglês fork).

```
1: P0(thinking{0} eating{1})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |  |
P0:F0.releasing & P0:F1.releasing>3
2: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{0} eating{1})[F1 F0] + F1(idle{0} busy{1})[] |  |
P1:F1.releasing & P1:F0.releasing>4
3: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P0:F0.releasing() + P0:F1.releasing() | F1?releasing>5
F0?releasing>6
4: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P1:F1.releasing() + P1:F0.releasing() | F1?releasing>7
F0?releasing>8
5: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |
P0:F0.releasing() | F0?releasing>0
7: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |
P1:F0.releasing() | F0?releasing>0
6: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P0:F1.releasing() | F1?releasing>0
8: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P1:F1.releasing() | F1?releasing>0
0: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |  |
P0:F0!taking & P0:F1!taking>1  P1:F1!taking & P1:F0!taking>2
```

A Figura 4.18 representa graficamente o espaço de estados completo do modelo quando temos dois garfos e dois filósfos. Neste caso, aparentemente podemos construir uma função de particionamento eficiente, dividindo em nós do lado direito e lado esquerdo. Porém, quando aumentamos o número de componentes do sistema, os nós do grafo de ocorrência tornam-se mais acoplados, dificultando uma função de particionamento eficiente. Isso será refletido nos resultados experimentais.

Em nosso caso, implementamos ele de uma forma síncrona, ou seja, um filósofo ao comer pega os dois garfos instantâneamente, decidimos que ao termos um filósofo n comendo, este estado será computado também pela partição n.

Figura 4.18: Representação gráfica do espaço de estados do jantar dos filósofos

Abaixo apresentamos as duas partições produzidas pelo gerador distribuído de espaços de estados. Perceba que o estado 2 para partição 0 é uma imagem ou réplica do estado 0 na partição 1. Da mesma forma acontece com o estado 4 da partição 1 e o estado 0 da partição 0. Isto se deve à forma como escolhemos de criar rotas entre partições, pois os estados borda são replicados para que se possa trafegar entre essas partições.

```
Partition 0:
1: P0(thinking{0} eating{1})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |  |
P0:F0.releasing & P0:F1.releasing>3
2: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{0} eating{1})[F1 F0] + F1(idle{0} busy{1})[] |  |
3: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P0:F0.releasing() + P0:F1.releasing() | F1?releasing>4
F0?releasing>5
4: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |
P0:F0.releasing() | F0?releasing>0
5: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P0:F1.releasing() | F1?releasing>0
0: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |  |
P0:F0!taking & P0:F1!taking>1  P1:F1!taking & P1:F0!taking>2{P1}

Partition 1:
0: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{0} eating{1})[F1 F0] + F1(idle{0} busy{1})[] |  |
P1:F1.releasing & P1:F0.releasing>1
1: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
```

```
P1:F1.releasing() + P1:F0.releasing() | F1?releasing>2
F0?releasing>3
2: P0(thinking{1} eating{0})[F0 F1] + F0(idle{0} busy{1})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |
P1:F0.releasing() | F0?releasing>4{P0}
3: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{0} busy{1})[] |
P1:F1.releasing() | F1?releasing>4
4: P0(thinking{1} eating{0})[F0 F1] + F0(idle{1} busy{0})[] +
P1(thinking{1} eating{0})[F1 F0] + F1(idle{1} busy{0})[] |  |
```

Embora nessa instância do modelo exista um equilíbrio aparente com 6 e 5 estados, verificaremos mais adiante nos resultados que isso não foi sempre possível pelo método de particionamento adotado.

4.3.3 Experimentação com Geração Centralizada

Como podemos observar na Tabela 4.18, com a evolução do número de componentes, que é a soma do número de arcos e nós do grafo de ocorrência, temos um aumento de ordem exponencial no tempo de geração de todo o grafo. Pior ainda, quando o sistema ultrapassa os cem mil componentes, em todos os computadores do laboratório onde foram realizados os experimentos (1.8Ghz e 1Gb de memória RAM) há um travamento geral do sistema por falta de memória - mesmo considerando que houve uma alocação prévia, máxima possível, de memória virtual e *heap* concedida pela Java Virtual Machine na chamada ao gerador. Concluímos que a ferramenta J-Mobile quando aplicada a um problema sem requisitos de escala era bastante funcional e satisfazia aos objetivos que foram propostos para ela. Ela apresentava um tempo de geração aceitável para pequenos modelos, porém a desconsideração de formas de compactação de representação de seus modelos durante seu projeto arquitetural comprometeram bastante essa ferramenta no quesito eficiência de geração.

Nº Filósofos	Nº Estados	Nº Arcos	1ª Exec	2ª Exec	3ª Exec	Tempo Médio
2	9	12	0.002 min	0.0009 min	0.0011 min	0.001 min
3	34	72	0.003 min	0.004 min	0.004 min	0.003 min
4	117	336	0.00985 min	0.0102 min	0.009 min	0.009 min
5	391	1410	0.026 min	0.027 min	0.027 min	0.027 min
6	1296	5616	0.14 min	0.13 min	0.13 min	0.13 min
7	4285	21672	1.43 min	1.41 min	1.35 min	1.4 min
8	14157	81840	1hs 46 min	1hs 44 min	1hs 58 min	1hs 49 min

Tabela 4.18: Geração centralizada do Jantar dos Filósofos

4.3.4 Experimentação com Geração Distribuída no Laboratório

Mesmo com um ferramenta de geração de próximos estados mais complexa, ainda é possível observar-se na Tabela 4.19 que fomos bem além no tamanho do grafo de ocorrência gerado. Essa solução distribuída com as máquinas internas do laboratório também apresenta bom desempenho de computação, com tempos de geração satisfatórios, chegando a ultrapassar a solução centralizada no modelo com 8 filósofos.

Nº Filósofos	Nº Estados	Nº Arcos	1ª Exec	2ª Exec	3ª Exec	Tempo Médio
2	9	12	0.063 min	0.066 min	0.077 min	0.068 min
3	34	72	0.19 min	0.18 min	0.18 min	0.183 min
4	117	336	0.243 min	0.25 min	0.29 min	0.261 min
5	391	1410	0.42 min	0.38 min	0.38 min	0.39 min
6	1296	5616	0.91 min	0.94 min	0.87 min	0.9 min
7	4285	21672	4.93 min	5.70 min	5.24 min	5.29 min
8	14157	81840	16.59 min	15.08 min	19.32 min	17 min
9	46762	304128	2hs 33 min	1hs 59 min	2hs 46 min	2hs 28 min
10	154460	1130179	13hs 21 min	15hs 55min	12hs 03min	13hs 59 min

Tabela 4.19: Geração distribuída no laboratório do Jantar dos Filósofos

Este resultado foi bastante animador em termos de pesquisa, pois achávamos que com um modelo mais complexo e um simulador menos eficiente seria muito difícil mantermos os mesmos patamares de vantagens que estávamos obtendo com as soluções anteriores.

Na Tabela 4.20, detalhamos melhor os dados para a geração com cinco filósofos. De acordo também com o histograma apresentado percebemos que o número de estados e arcos

processados pelas partições variam bastante e o número de estados exportados também é bastante elevado considerando-se o tamanho do espaço de estados. Devido a esses problemas, consideramos então esse modelo bastante adequado para obtermos resultados exaustivos com essa solução.

Partição	Exportados	Nº Execuções	Nº Estados	Nº Arcos	Tempo Médio
0	74	5	119	190	1s
1	73	7	141	221	1.5s
2	100	5	199	337	1s
3	150	5	374	835	2.5s
4	199	5	545	1337	3s

Tabela 4.20: Equilíbrio entre partições no Jantar dos Filósofos

Figura 4.19: Número de estados processados e estados exportados por cada partição

4.3.5 Experimentação com Geração Distribuída na Comunidade

A Tabela 4.21 apresenta dados do processo de geração distribuída utilizanda a comunidade Ourgrid.

Nº Filósofos	Nº Estados	Nº Arcos	1ª Exec	2ª Exec	3ª Exec	Tempo Médio
2	9	12	0.084s	0.078s	0.072s	0.078s
3	34	72	0.229s	0.229s	0.228s	0.22s
4	117	336	0.65s	0.65s	0.63s	0.64s
5	391	1410	2.999s	3.090s	2.990s	2.99s
6	1296	5616	31.67s	31.93s	31.72s	31.74s
7	4285	21672	8.59 min	8.27 min	8.24 min	8.42 min
8	14157	81840	15.27 min	15.5 min	15.55 min	15.44 min
9	46762	304128	4 hs 35 min	4 hs 52 min	4 hs 32 min	4 hs 39 min
10	154460	1130179	17 hs 44 min	19 hs 21 min	X	18 hs 32 min
11	510197	4199891	49 hs 28 min	X	X	39 hs 28 min

Tabela 4.21: Geração distribuída na comunidade do Jantar dos Filósofos

Os seguintes gráficos comparam essas tabelas. O gráfico na Figura 4.20 ilustra os resultados da maneira que foram coletados e percebemos que só temos diferenças visíveis a partir do modelo com sete filósofos. Porém, na Figura 4.21 observamos mais detalhadamente que até esse modelo a geração centralizada prevalecia em tempo de desempenho, sendo ultrapassada depois, bem próximo de seu travamento. A geração com máquinas do laboratório apresenta um comportamento de crescimento linear, porém infelizmente pela falta de recursos para no modelo com dez filósofos. O topo da escala fica por conta da geração na comunidade.

4.3.6 Verificação On-the-Fly de Propriedades no Procolo IP Móvel

Nesta seção, teremos mais verificação de propriedades comportamentais. Estas propriedades agora são específicas para o modelo dos filósofos.

Espera Circular

A situação de espera circular ou *deadlock*, caracteriza-se como uma situação em um sistema no qual ocorre um impasse e este fica impedido de continuar suas atividades normais indefinidamente. Este problema é bastante estudado no contexto de sistemas concorrentes, pois é inerente da própria natureza desses sistemas.

No contexto do modelo do Jantar dos Filósofos, se todos os filósofos retirarem seu garfo esquerdo e esperarem que o garfo direito fique disponível, o sistema entrará em espera cir-

Figura 4.20: Comparação entre experimentos centralizados e distribuídos para o modelo dos filósofos

Figura 4.21: Comparação entre experimentos centralizados e distribuídos com Y em escala logarítmica para o modelo dos filósofos

cular. A computação não poderá avançar porque os filósofos nunca conseguirão pegar seu segundo garfo. Na implementação de nosso modelo, evitamos essa situação implementando uma sincronização entre as ações de cada filósofo de pegar os garfos. Nesse caso, cada filósofo ao passar do estado *pensando* para *comendo*, retirará atomicamente os dois garfos da mesa. Para a devolução dos garfos, não há essa preocupação.

Embora de antemão já estejamos informados que essa condição não ocorre, o seguinte gráfico ilustra a busca por um estado sem sucessor em todo o espaço de estados distribuído durante a própria geração. Seus resultados se aproximam bastante ao processo de geração em si.

Figura 4.22: Procura por uma situação de deadlock

Fome de um Processo

Fome, ou do inglês *starvation*, é um problema também bastante estudado na ciência da computação, no qual um processo é negado perpetuamente de acessar os recursos necessários. É uma situação parecida com deadlock, que evita o progresso normal do programa. Isso geralmente ocorre quando um processo detém recursos que os outros necessitam, mas não pretende liberá-los.

No modelo dos filósofos, isso ocorre se não é permitido que um filósofo adquira os dois garfos em algum período específico de tempo. Existem várias modificações no modelo que evitam esse tipo de execução, como por exemplo determinar tempos específicos para cada filósofo pegar e devolver os garfos.

No modelo que foi contruído, essa situação não foi evitada, e existem execuções possíveis em que cada um dos filósofos fique sem acesso a ambos os garfos. No gráfico da Figura 4.23, é ilustrada essa busca para o caso do filósofo de índice um ficar com fome. A busca por esse caminho de execução exige que apenas uma pequena parte do grafo seja gerada, produzindo assim ganhos substanciais em relação à versão centralizada, pois uma determinada partição sempre acha antes das demais, abortando assim o processo.

Figura 4.23: Procura por uma situação de fome de um filósofo

4.3.7 Conclusões sobre Modelo dos Filósofos

Concluímos deste experimento que nossa solução não oferece ganhos tão significativos em escala quando a ferramenta de descrição do sistema concorrente utiliza muita memória para representação dos estados sem alguma preocupação. Nesse caso, há uma sobrecarga muito acentuada no módulo *coordenador*, dificultando muito o gerenciamento das partições e a

leitura/escrita dos grafos de ocorrência computados. Além do mais, este modelo não possibilitou uma função de particionamento que considerássemos satisfatória, gerando também muitos arcos que atravessam partições.

4.4 Modelo do Padrão Itinerary para Agentes Móveis

4.4.1 Descrição

Este padrão de projeto para agentes móveis provê uma maneira de executar a migração de um agente, que será responsável por realizar uma dada tarefa em máquinas remotas. O agente recebe um itinerário na agência de origem, indicando a seqüência de agências que deve visitar. Uma vez que o agente estiver em uma agência, ele deve executar a tarefa localmente e continuar seu itinerário. Após visitar a última agência, o agente sabe que concluiu sua tarefa designada. Este padrão é uma boa solução para agentes que precisam executar tarefas seqüenciais. Em nosso modelo, descrito em uma versão de redes de Petri coloridas suportada por nosso protótipo, inserimos a complexidade de vários agentes migrarem ao mesmo tempo. Isto dificulta bastante nossa função de particionamento, pois praticamente o grafo todo torna-se um componente fortemente conectado.

A Figura 4.24 descreve a rede que desenvolvi para modelar esse padrão. O lugar Migrating é responsável por guardar as fichas do tipo Agente, representando que eles estão prontos e com número de passos possíveis de sua rota definidos para migrarem da sua agência de origem.

4.4.2 Particionamento e Características do Grafo de Ocorrência

As regiões de cores diversificadas representam as agências, e é considerando elas que calculamos o particionamento. Assim, por prioridade, quando a primeira agência estiver com algum agente executando nela, preferencialmente estaremos na partição de número um, se isso não ocorrer, mas tivermos agentes executando na segunda partição, estaremos na partição dois e assim por diante.

A Figura 4.25 que segue logo abaixo, apresenta a idéia geral do grafo de ocorrência que descreve o comportamento desse protocolo. Observamos que seu particionamento, do lado

Figura 4.24: Rede de Petri colorida que descreve o padrão Itinerary

direito da figura, é bem mais complicado, gerando vários arcos que atravessam partições (*cross-arcs*). Devido a isso, também não conseguimos manter o equilíbrio entre o número de nós explorados pelas partições. Isto se deve à prioridade dada aos agentes por ordem decrescente. Assim, se tivermos uma configuração com *m* agentes migrando, estaremos na partição MIN(*id*($agente_1$),..., *id*($agente_m$)).

4.4.3 Experimentação com Geração Centralizada

Nos seguintes experimentos, que estão descritos na Tabela 4.22, fixamos a quantidade de agências para três, o número máximo de passos no itinerário para três e o trabalho executado em cada agência pelos agentes como dez fichas para o disparo da transição *Workload*. O que variamos foram apenas a quantidade de agentes, porém isso não implicava em uma mudança linear no comportamento desse sistema. Verificamos a ocorrência de um crescimento exponencial devido às combinações de visita de um agente às várias agências.

Figura 4.25: Grafo de ocorrência do padrão Itinerary antes e depois do particionamento

Agências	Nº Estados	Nº Arcos	Tempo Médio
2	2692	8748	2s
3	7570	27702	4s
4	16552	64152	15s
5	30934	123930	31s
6	52012	212868	67s
8	119440	501552	182s
10	229204	976860	415s
15	X	X	X

Tabela 4.22: Geração centralizada do Padrão Itinerário

4.4.4 Experimentação com Geração Distribuída no Laboratório

De acordo com a função de particionamento apresentada, de antemão sabíamos que não seria possível obtermos um balanceamento regular, e esse era nosso intuito, pois precisaríamos testar o limite da solução nesse quesito. A Tabela 4.23 apresenta os dados da geração interna no laboratório.

Observando o gráfico acima, percebemos que houve um incremento bastante acentuado no número de estados replicados na solução distribuída. Isso é característico de soluções com grande número de comunicações e pouco equilíbrio. Além do mais, percebemos a inviabilidade de desempenho causada no modelo, esse é o extremo negativo da solução.

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	∞	3790	8748	37s
3	∞	12088	27702	164s
4	∞	26362	64152	880s
6	∞	82630	212868	11303s
8	∞	185653	501552	41235s
10	∞	417128	1181744	77841s

Tabela 4.23: Geração distribuída no laboratório do Padrão Itinerário

Figura 4.26: Comparação de equilíbrio entre nós e arcos em uma execução

4.4.5 Experimentação com Geração Distribuída na Comunidade

Nesse modelo, somando-se algum custo de comunicação, os tempos de geração na comunidade se assemelharam bastante à geração com máquinas do laboratório. Os resultados estão descritos na Tabela 4.24.

Partições	Buffer	Nº Estados	Nº Arcos	Tempo Médio
2	∞	3790	8748	41s
3	∞	12088	27702	186s
4	∞	26362	64152	902s
6	∞	82630	212868	11512s
8	∞	185653	501552	42729s
10	∞	417128	1181744	83858s

Tabela 4.24: Geração distribuída na comunidade do Padrão Itinerário

4.4.6 Conclusões sobre Padrão Itinerary

Conforme pode-se observar no gráfico da Figura 4.27, que está em escala logaritmica, obtemos melhores tempos do que a geração centralizada nesse modelo. Isto se deve ao pouco equilíbrio obtido com nossa função de particionamento. Provavelmente, obteríamos tempo bem mais satisfatórios se novamente utilizássemos a geração em disco distribuída do protocolo do IP Móvel com retorno, porém, o intuito aqui é provarmos que nosso método permite pelo menos obter os resultados das gerações seqüenciais, por pior que seja a função de particionamento e não ocorram sobrecargas nas comunicações.

Figura 4.27: Experimentação centralizada contra a distribuída no padrão Itinerary

O número de estados a serem computados nesse modelo cresce a uma ordem bastante

alta com a complexidade do modelo, e sua representação também demanda muita memória. Assim, considero esse resultado bastante satisfatório observando-se os problemas envolvidos.

4.5 Conclusões das Experimentações

No geral, os benefícios de usarmos soluções distribuídas para a exploração do espaço de estados é o ganho em desempenho e o uso de memória distribuída no intuito de gerarmos maiores espaços de estados. Com relação ao ganho em desempenho, como a grade computacional adotada requer comunicação através de diferentes domínios, não é possível garantirmos esse ganho sempre porque em alguns casos o custo de comunicação é maior do que o ganho resultante do processamento paralelo. Assim, o principal benefício esperado é o aumento na escala dos modelos sob verificação.

Nossos principais objetivos com esse trabalho foram identificação das vantagens e desvantagens fornecidas pela solução, em termos de tempo de processamento, consumo de memória e limites de escala. Estávamos interessados principalmente em soluções que lidassem com grafos de ocorrência do maior tamanho possível. Aqui, focamos na construção de modelos de sistemas concorrentes e na elaboração de funções de particionamento que possibilitassem o maior paralelismo possível entre as partições. As conclusões mais importantes obtidas com essa atividade foram:

- É preciso ter um conhecimento prévio do aspecto geral do espaço de estados para elaborar uma boa função de particionamento.
- Uma boa função de particionamento é aquela que minimiza o número de rotas entre as partições e que identifica o maior número possível de componentes concorrentes no modelo.
- O tempo gasto com comunicação é proporcional ao número de rotas.
- O número de ciclos de geração é proporcional ao número de rotas.
- O tempo total de geração é proporcional ao número de ciclos de geração.
- Quanto mais balanceadas forem as partições, maior será o paralelismo entre elas.

- Quanto maior for a média de duração dos ciclos de geração, maior será o ganho de desempenho com relação à geração centralizada.

- Há um incremento no ganho à medida em que aumenta o tamanho do espaço de estados em casos de particionamentos aceitáveis.

- Quando inserido em uma comunidade, não devemos esperar muito ganho em relação ao quesito desempenho, pois a disponibilidade ótima dos recursos nem sempre é garantida. Por outro lado, temos a certeza que poderemos trabalhar com modelos de uma ordem milhares de vezes maior do que os obtidos com versões centralizadas do sistema ou em uma rede local de poucas máquinas.

- No pior caso, onde o tempo distribuído esteja muito acima do tempo centralizado e não haja possibilidade de escalar em relação ao modelo centralizado, deve-se recorrer às gerações seqüenciais com várias réplicas providas pela grade computacional.

Capítulo 5

Conclusão

Durante o desenvolvimento desta dissertação, usamos métodos experimentais para obtermos uma análise sobre a viabilidade do emprego de grades computacionais na tarefa de verificação de modelos distribuída na tentativa de aliviar o problema da explosão do espaço de estados. Esta investigação justificou-se na atratividade que a plataforma de grades disponibiliza em termos de ser de baixo custo, auto-gerenciável e extremamente escalável.

Nos preocupamos em produzir um protótipo o mais robusto possível ao invés de apenas produzirmos algoritmos e idéias abstratas de algoritmos de distribuição baseando-se em algumas primitivas genéricas de comunicação. Foi intuito desse protótipo reproduzir fielmente alguns benefícios e limites de quando precisamos adotar alguma tecnologia para implementar ferramentas baseadas em técnicas formais. Explicitamos os benefícios do desacoplamento da ferramenta de geração e verificação do espaço de estados da plataforma de distribuição. Como, por exemplo, o fato do particionamento ficar apenas a nível de modelo, pois a grade é gerenciadora do nível de recursos.

Os algoritmos de verificação são explícitos e para a lógica temporal CTL. O uso destes algoritmos justicam-se por ser adequada para expressar propriedades de sistemas concorrentes, eficiente e já termos tido um trabalho que os implementasse no nosso grupo de pesquisa.

Apresentamos duas ferramentas específicas, MyGrid e J-Mobile. Nós não amarramos a abordagem proposta a elas. Outros middlewares de grades computacionais podem ser usados considerando-se o algoritmo de coordenação. Da mesma forma que ocorre com o MyGrid, usuários podem conseguir outras soluções também grátis para problemas cruciais de sistemas distribuídos. Naturalmente, muitos aspectos precisam serem revistos, como o

mecanismo de transferência de dados. O mesmo pode ser dito sobre os formalismos para descreverem os modelos dos sistemas.concorrentes. Formalismos que permitam a construção e verificação de espaços de estados em uma maneira enumerativa podem ser usados, como redes de Petri Coloridas [Jen92] por exemplo. Do ponto de vista de verificação, os algoritmos CTL distribuídos permanecem intactos qual seja a arquitetura da grade.

Durante a aplicação experimental do protótipo, obtivemos alguns resultados interessantes. Conforme prometido, conseguimos lidar com espaços de estados maiores dos sistemas em comparação com versões centralizadas da ferramenta. Porém, esse ganho em escala nem sempre implica em ganho de desempenho durante o processo. Em alguns modelos ainda não temos um método eficaz de particionamento devido termos poucas transições conflitantes e isso dificulta bastante dividirmos o trabalho em sub-tarefas eficientes, além da plataforma prover um custo muito alto de comunicação quando necessitamos simular comunicação entre as tarefas.

A conclusão que obtemos é que dá pra usar grades como plataforma de distribuição, porém devemos ter um conhecimento prévio do comportamento do modelo para que se construa uma função de particionamento aceitável ou recorra para geração em seqüencial em disco na plataforma. Esse conhecimento prévio muitas vezes já pode ter sido adquirido por experiências frustradas com outros métodos antes de se recorrer a este. É importante frisar, que muitas vezes essa será a última opção de alguns engenheiros devido à escassez de seus recursos e seus limites financeiros, o que deve aumentar a tolerância às dificuldades apresentadas pela solução.

5.1 Contribuições

Antes das publicações dos resultados dessa dissertação em congressos científicos, a aplicação da infra-estrutura de grades computacionais em verificação de modelos era uma grande incógnita. Mais especificamente, não se conhecia nada a respeito de como se tirar proveito da filosofia de comunicação bag-of-tasks usufruindo de suas potencialidades e contornando suas limitações. Os resultados experimentais com os modelos cuidadosamente selecionados exemplificam o que estamos contribuindo para a área. Lá, mostramos que embora estejamos ainda distantes de uma solução definitiva para o problema da explosão do espaço de estados,

é possível geralmente, fazendo uso de alguma heurística, contornar esse problema de uma maneira escalável e bastante viável economicamente.

Este trabalho é um pontapé inicial em um novo engajamento em pesquisa dentro do grupo, que é a investigação de técnicas que amenizem o problema da explosão do espaço de estados. O objetivo inicial, que era de prover um framework, foi alcançado. Ele pode ser utilizado por outros grupos de pesquisa ou engenheiros que possuam problemas ao tentarem verificar o comportamento de seus sistemas concorrentes. A depender do interesse de usuários, esse núcleo de verificação pode ser expandido provendo outras funcionalidades, maior robustez e uma melhor usabilidade.

Embora não apresentemos aqui novos algoritmos de verificação em CTL, a proposta de reuso dos algoritmos já existentes e suas adaptações para uma plataforma em bastante emergência já caracteriza-se como uma contribuição relevante. Isso é importante também para garantirmos que existe um desacoplamento entre o verificador e os recursos distribuídos compartilhados existentes. Na abordagem atual, os algoritmos dependem da representação explícita do espaço de estados do sistema ou propriedades podem ser verificadas *on-the-fly*, ou seja, durante a geração do espaço de estados. Nada impede que outras abordagens sejam tentadas, como por exemplo uma abordagem simbólica baseada em algoritmos de ponto fixo, técnicas de interpretação abstrata, relações de ordem parcial, simetrias e outros mais. Aqui não assumimos que nossa técnica se sobressaia com relação às demais em todas as situações, mas que uma combinação, principalmente de técnicas que produzem uma representação eficiente do espaço de estados com o poder computacional fornecido por grades atualmente.

5.2 Trabalhos Futuros

A primeira sugestão de um próximo trabalho é transformar o protótipo desenvolvido aqui em uma ferramenta robusta. Devido ao pouco tempo e poucos recursos, alguns aspectos ainda não foram implementados eficientemente e necessitamos de um refatoramento no protótipo, uma interface amigável e tutorial no que é exigido de configuração.

Ainda não estamos satisfeitos com todos os experimentos que foram realizados. Em breve estaremos inserindo mais modelos para nossa base de aplicações e alterando ou adici-

onando variáveis de observação também.

Um outro problema que também não foi atacado devidamente é a validação formal da solução. Os algoritmos que foram reusados [RdFG04] já foram validados [MOBR04] em um contexto sem distribuição e estavam implementados em linguagem funcional. Aqui, ainda não temos a garantia da preservação de semântica do que foi traduzido de código funcional para Java e muito menos os aspectos de distribuição foram devidamente testados.

A partir de agora precisamos experimentar mais a abordagem. Devemos procurar grupos de pesquisa com interesse em reusar a experiência que nós adquirimos e dessa forma, também contribuam com o amadurecimento da abordagem.

Bibliografia

[ABCM04] Nazareno Andrade, Francisco Vilar Brasileiro, Walfredo Cirne, and Miranda Mowbray. Discouraging free riding in a peer-to-peer cpu-sharing grid. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 129–137, 2004.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[BBF+01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. Mckenzie. *Systems and Software Verification - Model-Checking Techniques and Tools*, volume 1. Springer, 2001.

[BBS01] Jiri Barnat, Lubos Brim, and Jitka Stříbrná. Distributed LTL model-checking in SPIN. *Lecture Notes in Computer Science*, 2057:200+, 2001.

[BG05] Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. *Form. Methods Syst. Des.*, 27(1/2):29–66, 2005.

[Bou05] M. Bourahla. Distributed ctl model checking. In *IEEE - Software – December 2005*, volume 152, Issue 6, pages 297–308, 2005.

[BZ03] Luboš Brim and Jitka Zidkova. Using assumptions to distribute alternation free μ-calculus model checking. *Electronic Notes in Theoretical Computer Science*, 89(1):17–32, 2003.

[Cab04] J. M. Cabral. Geração de espaços de estados para modelos rpoo. *Relatório de Iniciação Científica da UFCG*, 2004.

[Cab05] J. M. Cabral. Verificação de modelos rpoo. *Relatório de Defesa de Estágio da UFCG*, 2005.

[CBA+04] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Daniel Paranhos, Elizeu Santos-Neto, César De Rose, Tiago Ferreto, Miranda Mowbray, Roque Scheer, and João Jornada. Scheduling in Bag-of-Task Grids: The PAU´A Case. In *SBAC-PAD 2004 16th Symposium on Computer Architecture and High Performance Computing*, October 2004.

[CGN98] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS J. on Computing*, 10(1):82–93, 1998.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CP03] I. Cerna and R. Pelanek. Distributed explicit fair cycle detection. In *10th International SPIN Workshop on Model Checking of Software, Portland, Oregon, 2003. 16*, LNCS, pages 49–73, 2003.

[CPC+03] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. A. B da Silva, C. O. Barros, and C. Silveira. Running bag-of-task applications on computational grids: the MyGrid approach. In *Proceedings of the ICCP 2003 - International Conference on Parallel Processing*, 2003.

[DHJ+01] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd international conference on Software engineering*, pages 177–187. IEEE Computer Society, 2001.

[Dij02] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. pages 198–227, 2002.

[DK99] Zhe Dang and Richard A. Kemmerer. Using the astral model checker to analyze mobile ip. *ACM*, pages 132–141, 1999.

[dSCB03] Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference*, pages 169 – 180. Springer-Verlag GmbH, 2003.

[dSM06] D. A. da Silva and P. D. L. Machado. Verificação de modelos em redes de petri orientadas a objetos. *Dissertação de Mestrado, UFCG*, Maio 2006.

[EOH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[ES97] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, July 1997.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[FK98] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, volume 1. Morgan Kaufmann, 1998.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.

[GB06] Guy Edward Gallasch and Jonathan Billington. A parametric state space for the analysis of the infinite class of stop-and-wait protocols. In *SPIN*, pages 201–218, 2006.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state

space construction for model-checking. *Lecture Notes in Computer Science*, 2057:217+, 2001.

[God03] P. Godefroid. Software model checking: the verisoft approach, 2003.

[GPS96] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *Proceedings of the 1996 international symposium on Software testing and analysis*, pages 261–269. ACM Press, 1996.

[GS03] F. V. Guerra and T. M. Silva. Modelagem do Protocolo IP móvel. Relatório da disciplina Modelagem e Validação de Sistemas usando de Redes de Petri, oferecida pela COPIN - UFPB, Maio 2003.

[Gue02] D. D. S. Guerrero. Redes de petri orientadas a objeto. *Tese de Doutorado - COPELE, UFCG*, 2002.

[GVJH98] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns CD: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[HP95] Gerard Holzmann and Doron Peled. Partial order reduction of the state space. In *First SPIN Workshop*, Montrèal, Quebec, 1995.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.

[Jou03] Christophe Joubert. Distributed model checking: From abstract algorithms to concrete implementations. In *Proc. of the Parallel and Distributed Model Checking 2003*, 2003.

[Kat99] J. P. Katoen. *Concepts, Algorithms, and Tools for Model Checking*. Lectures Notes of the Course "Mechanised Validation of Parallel Systems. Friedrich-Alexander Universitat Erlangen-Nurnberg, 1999.

[KP04] Lars M. Kristensen and Laure Petrucci. An approach to distributed state space exploration for coloured petri nets. In *Proceedings of ICATPN 2004, Bologna, Italy, June 21-25, 2004 — Volume 3099 of LNCS / Cortadella, Reisig (Eds.)*, pages 474–483. Springer-Verlag, September 2004.

[Lim04] E. F. A. Lima. Formalização e análise de padrões de projeto para agentes móveis. *Dissertação de mestrado - COPIN, UFCG*, 2004.

[LLEL02] Alberto Lluch-Lafuente, Stefan Edelkamp, and Stefan Leue. Partial order reduction in directed model checking. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 112–127, London, UK, 2002. Springer-Verlag.

[LS99] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[Mat87] F. Mattern. Algorithms for distributed termination detection. pages 161–175, 1987.

[mck02] Model checking kit, 2002. http://wwwbrauer.in.tum.de/gruppen/theorie/KIT/.

[McM93] K. L. McMillan. Symbolic model checking: An approach to the state explosion problem. *Kluwer Academic Publishers*, 1993.

[MDS+05] Kaoutar El Maghraoui, Travis Desell, Boleslaw K. Szymanski, James D. Teresco, and Carlos A. Varela. Towards a middleware framework for dynamically reconfigurable scientific computing. In L. Grandinetti, editor, *Grid*

Computing and New Frontiers of High Performance Processing, volume 14 of *Advances in Parallel Computing*, pages 275–301. Elsevier, 2005.

[MOBR04] P. D.L̃. Machado, E. A.S̃. Oliveira, P. E.S̃. Barbosa, and C.L̃. Rodrigues. Algebraic specification-based testing: The veritas case study. In *Proc. of Brazilian Symposium on Formal Methods (SBMF2004)*, 2004.

[Nam98] K. Namjoshi. Ameliorating the state space explosion problem, 1998.

[Nic87] R De Nicola. Extensional equivalence for transition systems. *Acta Inf.*, 24(2):211–237, 1987.

[NP06] Peter Niebert and Doron Peled. Efficient model checking for ltl with partial order snapshots. In *TACAS*, pages 272–286, 2006.

[our05] OurGrid Project, 2005. www.ourgrid.org.

[OvdPE04] Simona Orzan, Jaco van de Pol, and Miguel Valero Espada. A state space distribution policy based on abstract interpretation. In *Proc. of the Parallel and Distributed Model Checking 2004*, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[RBGdF04] C. L. Rodrigues, P. E. S. Barbosa, D. D. S. Guerrero, and J. C. A. de Figueiredo. Rpoo model checker. In *Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas*, Brasília - Brasil, Outubro 2004.

[RdFG04] C. L. Rodrigues, J. C. A. de Figueiredo, and D. D. S. Guerrero. Verificação de modelos em redes de petri orientadas a objetos. *Dissertação de Mestrado, UFCG*, Outubro 2004.

[RGdF+04] C. L. Rodrigues, F. V. Guerra, J. C. A. de Figueiredo, D. D. S. Guerrero, and T. S. Morais. Modeling and verification of mobility issues using object-oriented petri nets. In *Proc. of 3rd International Information and Telecommunication Technologies Symposium (I2TS2004)*, 2004.

[Sch00] K. Schmidt. LoLA: A low level analyser. In Nielsen, M. and Simpson, D., editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, volume 1825, pages 465–474. Springer-Verlag, 2000.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. In *9th International Conference on Computer Aided Verification*, 1997.

[SG04] A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004.

[SIE05] D. Sahoo S. Iyer, J. Jain and E. Emerson. Under-approximation heuristics for grid-based bmc. In *International Workshops on Parallel and Distributed Methods in Verification - PDMC 2005*, pages 6–21, 2005.

[Sif90] J. Sifakis, editor. *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*. Springer, 1990.

[Sil04] T. M. Silva. Simulação automática e geração de espaç de estados de modelos rpoo. *Dissertação de mestrado - COPIN, UFCG*, 2004.

[STMBSS98] Joel R. Stiles, Jr. Thomas M. Bartol, Edwin E. Salpeter, and Miriam M. Salpeter. Monte carlo simulation of neuro-transmitter release using mcell, a general simulator of cellular physiological processes. In *CNS '97: Proceedings of the sixth annual conference on Computational neuroscience : trends in research, 1998*, pages 279–284, New York, NY, USA, 1998. Plenum Press.

[Val98] A. Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[VCT05] Carlos A. Varela, Paolo Ciancarini, and Kenjiro Taura. Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments. *Scientific Programming Journal*, 13(4):255–263, December 2005. Guest Editorial.