**Universidade Federal de Minas Gerais**
**Instituto de Ciências Exatas**
**Pós-Graduação em Ciência da Computação**

# Um Algoritmo Distribuído para Verificação de Modelos com Fronteiras

*Dissertação submetida à Universidade Federal de Minas Gerais como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.*

Hugo Valentim Barros

Orientador:
Sérgio Vale Aguiar Campos

Belo Horizonte
Novembro de 2004

## Resumo

Este trabalho apresenta um algoritmo distribuído para o problema de Verificação de Modelos com Fronteiras. O modelo proposto divide o problema a ser resolvido em um solucionador primário e varios secundários, com o objetivo de executar cada um em um computador diferente. Esta arquitetura permite explorar a simetria inerente do problema de Verificação de Modelos com Fronteiras devido ao fato de que cada transição do modelo ser semelhante as outras. Os resultados obtidos demonstram que esta divisao de tarefas é eficiente e obtém ganhos significativos. No decorrer deste texto apresentamos exemplos em que obtivemos ganhos de até um ordem de magnitude no tempo de resposta e na utilização de memória.

## Abstract

This work introduce a distributed algorithm to the bounded model checking problem. The proposed model distribute the problem between a primary solver and several secondary solvers with the objective of execute each one in a different computer. This architecture explore the inherent simmetry of the bounded model cheking problem, due to the fact that each transition in the model is similar to the other ones. The results obtained show that this task distribution is efficient. In the course of the text we present examples where we achieved gains up to one order of magnitude in time and memory utilization.

# Agradecimentos

Ao professor Sérgio Campos pelos excelentes cinco anos que trabalhei sob sua orientação, desde a iniciação científica, passando pela gerência de projetos de tecnologia até a conclusão deste mestrado. Anos que espero no futuro ter a oportunidade de repetir. Dentre as inúmeras qualidades do professor Sérgio gostaria de destacar uma em particular. Essa qualidade é para mim a perfeita capacidade de dosar a quantidade de liberdade e de pressão que se deve oferecer e fazer a um orientado. No meu caso serviu como uma luva.

Aos meus pais não só pela altíssima qualidade de vida que eles me ofereceram, mas principalmente pelo apoio e carinho constantes. Pessoas sem as quais eu dificilmente teria atingido este grau de qualificação.

A minha querida Amanda por todo amor, carinho e paciência durante nossos quase oito anos juntos. Ela foi peça fundamental pois, mais do que ninguém, acompanhou de perto e me apoiou muito durante toda minha vida acadêmica.

As minhas irmãs porque sem elas eu nunca poderia ser o filho preferido. Brincandeiras à parte, agradeço a elas por sempre tentarem, de maneiras nem sempre ortodoxas, a me convencer que a vida não são só números.

Aos meus amigos do DCC, em especial do Speed, aos meus amigos do CSA e aos parentes que, tenho certeza, estiveram sempre torcendo por mim.

E por fim ao Google por estar sempre presente quando a gente mais precisa dele e, é claro, por não cobrar nada pelos seus serviços.

# Sumário

# Capítulo 1

# Introdução

## 1.1 Motivação

Um dos maiores obstáculos para a utilização de computadores para a realização de tarefas sensíveis e críticas é a limitada capacidade humana em modelar e implementar sistemas mantendo um alto nível de confiança e correção. Falhas são uma constante em sistemas de computação.

As técnicas mais comuns que tentam reduzir as falhas de um sistema são *testes* e *simulações* [1]. Embora estas técnicas consigam detectar muitas falhas, erros mais sutis dificilmente são encontrados.

*Verificação de Modelos* [2] é uma abordagem complementar ao problema de detecção de falhas em um sistema. Enquanto que testes e simulações exploram apenas algumas possibilidades de cenários em um sistema, essa técnica faz uma exploração exaustiva de todos os cenários possíveis.

Nessa técnica uma propriedade comportamental é verificada sobre um modelo formal do sistema através da enumeração exaustiva de todos os estados alcançáveis e dos comportamentos encontrados na passagem sobre estes estados. Comparada a outras técnicas a Verificação de Modelos apresenta duas vantagens notáveis:

1. A técnica é completamente automática e sua aplicação não precisa da supervisão de nenhum especialista.

2. Quando uma falha é encontrada no modelo um *contra-exemplo* é sempre demonstrado buscando mostrar como alcançar essa falha.

Porém a Verificação de Modelos sofre de uma grave limitação. Dependendo do sistema a ser verificado o modelo gerado será tão grande que não será possível representá-lo na memória de um computador.

O aparecimento da técnica Verificação Simbólica de Modelos [3, 4], que representa os estados através de funções lógicas, tornou possível a representação de um número astronômico de estados. Nessa técnica o modelo é tradicionalmente implementado através de diagramas de decisão binários (*BDDs* [5]).

No entanto a capacidade de representação dos BDDs não atingiu níveis de modelagens reais, aquelas necessárias pela indústria. Técnicas como abstração e raciocínio composicional em conjunto com otimizações dos algoritmos possibilitaram aos BDDs ingressar em ferramentas comerciais, mas ainda em um nível abaixo do desejado pela indústria.

Um outro método de verificação baseado em representações simbólicas surgiu no final da década de 90 [6, 7]. Este método foi chamado de *Verificação de Modelos com Fronteiras* (*Bounded Model Checking - BMC*) e é baseado em algoritmos de solução de problemas de satisfiabilidade (*SAT* [8, 9]). Quando aplicado, uma fórmula proposicional é criada para representar o modelo e as propriedades que este deve atender. Essa fórmula proposicional é entregue a um algoritmo SAT. Se a fórmula não for satisfazível então o modelo está correto. Caso contrário o modelo tem alguma falha e as atribuições para a falha são o contra-exemplo. BMC baseado em SAT conseguiu detectar falhas em modelos de tamanho adequado para aplicações industriais. Mas, como as outras soluções, também sofre com o problema de explosão de estados. Para alguns modelos, ele executará infinitamente em busca de uma atribuição para a fórmula lógica correspondente. Outros modelos não poderão nem ser representados na memória do computador.

Uma alternativa natural para melhorar o método é a paralelização do mesmo. A utilização de vários computadores para a solução de um mesmo problema sempre trouxe ganhos nas mais variadas áreas da ciência da computação.

O trabalho desenvolvido se encaixa nesse contexto. Criamos um novo algoritmo distribuído para a verificação simbólica de modelos baseado em procedimentos SAT e no paradigma de passagem de mensagens.

## 1.2 Justificativa

Existem dois principais motivos que justificam essa pesquisa:

1. O desenvolvimento de um algoritmo de Verificação Simbólica de Modelos distribuído torna possível a verificação de sistemas cada vez mais complexos. A solução dos erros encontrados diminui custos e, em casos extremos, salvam vidas.

2. O custo de um cluster computacional muitas vezes excede a capacidade financeira da maioria das instituições de pesquisas e de empresas. A utilização dos próprios laboratórios de desenvolvimento e de pesquisa como cluster é uma solução viável e muitas vezes a única disponível para essas pessoas.

## 1.3 Objetivos

Este trabalho tem como objetivo contribuir com o desenvolvimento de um dos métodos mais eficientes de detecção de falhas em sistemas, a Verificação Simbólica de Modelos baseada em algoritmos SAT. A contribuição planejada envolve a aplicação de conceitos de sistemas distribuídos na solução do problema.

O primeiro objetivo do trabalho será desenvolver um algoritmo de Verificação Simbólica de Modelos distribuído. O algoritmo será desenvolvido visando usufruir ao máximo dos recursos oferecidos por um cluster de computadores.

O segundo objetivo será implementar este algoritmo em um verificador de modelos simbólico utilizado em larga escala. O terceiro objetivo será avaliar este algoritmo fazendo a verificação de sistemas complexos. O quarto e último objetivo será comparar estes resultados com soluções alternativas ao mesmo problema.

## 1.4 Organização

Esta dissertação está organizada em sete capítulos que podem ser divididos em dois grupos. O primeiro grupo discute o problema, a abordagem escolhida para tratá-lo e os fundamentos. O segundo grupo discute o algoritmo proposto, sua implementação e os resultados de experimentos realizados com o mesmo.

Os primeiro capítulo introduz o tema da dissertação. Os capítulos seguintes apresentam uma visão geral sobre as técnicas em que nosso algoritmo se fundamenta: solucionadores SAT baseadas no algoritmo Davis-Putman [10] e BMC baseado em métodos SAT. Em seguida descrevemos com mais detalhes o problema de explosão de estados e qual foi a abordagem escolhida para tratá-lo.

O segundo grupo começa no quinto capítulo, onde descrevemos com detalhes como funciona o algoritmo e o comparamos com soluções apresentadas

por outros pesquisadores. Em seguida, discutimos a implementação e os experimentos realizados. Por fim, no último capítulo concluímos o trabalho.

# Capítulo 2

# SAT

## 2.1 Introdução

Satisfiabilidade Proposicional (SAT) é o problema de decidir se existe uma atribuição para variáveis em uma fórmula proposicional que faz com que a fórmula seja verdadeira.

Embora o problema de satisfiabilidade proposicional seja um problema bastante simples de se descrever ele é um marco na teoria da complexidade computacional. Ele foi o primeiro problema que provaram ser NP-completo [11]. Dada essa classificação é bem improvável que exista algum algoritmo polinomial para SAT. No entanto, uma grande quantidade de pesquisas no últimos anos mostrou que algumas instâncias do problema de satisfiabilidade podem ser resolvidas eficientemente na prática.

Muitos problemas podem ser codificados em SAT, inclusive Verificação de Modelos com Fronteiras. As fórmulas SAT geradas a partir desses problemas são entregues a solucionadores SAT, normalmente no formato CNF. Estes por sua vez fazem a busca por atribuições que satisfaçam as fórmulas.

Existem muitos solucionadores SAT disponíveis ao público (e.g. GRASP [12, 13], POSIT [14], WalkSAT [15]). Esses algoritmos ou são *completos* ou são métodos *estocásticos*. Para uma dada instância SAT, algoritmos completos ou encontram uma solução ou provam que não existe nenhuma solução. Métodos estocásticos, por outro lado, não podem provar que uma determinada instância não é satisfazível, mas podem encontrar soluções para determinadas instâncias rapidamente. No caso de BMC, onde queremos provar a insatisfiabilidade das instâncias, métodos completos são um requisito.

Nos últimos anos, algoritmos baseados no procedimento Davis-Putman-Logemann-Loveland (DPLL) [16] vêm emergindo como alguns do mais eficientes solucionadores SAT completos. Na última década, em particular,

pudemos presenciar um aumento significante na pesquisa de solucionadores SAT baseados nesse procedimento. Uma das principais razões para isso foi a descoberta de aplicações práticas, como a Verificação de Modelos, que levaram os solucionadores existentes ao limite, fornecendo motivação para a busca de algoritmos ainda mais eficientes. Uma nova geração de solucionadores SAT surgiu, entre os quais podemos citar o BerkMin [17], o SATO [18] e o Chaff [19]. Essa nova geração pode resolver instâncias do SAT geradas a partir de aplicações industriais com dezenas de milhares ou até milhões de variáveis.

Maiores detalhes sobre solucionadores SAT podem ser encontrados em relatórios que oferecem um visão geral sobre o assunto. Destaque para os relatórios [9, 20, 21].

Na próxima seção descrevemos o que é uma fórmula CNF, formato usado pela maioria dos solucionadores SAT. Em seguida, descrevemos como funciona um solucionador genérico baseado no procedimento DPLL. Para concluir, citamos a competição SAT que mede a performance dos mais variados solucionadores SAT existentes, quando defrontados com os mais diversos tipos problemas.

## 2.2 CNF

Normalmente solucionadores SAT trabalham com fórmulas na forma normal conjuntiva (CNF). Isso não é uma limitação, já que existem algoritmos polinomiais (e.g. [22]) que transformam qualquer fórmula proposicional para uma fórmula equivalente no formato CNF.

**Definição 1** *Uma fórmula está em CNF se e somente se ela é uma conjunção de cláusulas. E uma cláusula é uma disjunção de literais, onde um literal é uma das* ***fases*** *de uma variável lógica. As fases de uma variável lógica $x$ são a* ***fase negativa*** *$\neg x$ e* ***fase positiva*** *$x$.*

Uma fórmula CNF é também chamada de **banco de cláusulas**.
Algumas cláusulas recebem nomes especiais:

1. Uma cláusula contendo apenas um literal sem valor atribuído é chamada de **cláusula unitária**.

2. Uma cláusula sem literais é chamada uma **cláusula vazia** e é interpretada como falsa.

3. Uma cláusula que tem todos os seus literais atribuídos com valor 0 é chamada de **cláusula conflitante**.

Um exemplo de fórmula CNF:

$$(a \vee d) \wedge (a \vee \neg c \vee \neg h) \wedge (a \vee h \vee l) \wedge (b \vee k)$$

Dada a estrutura de uma fórmula em CNF podemos notar que basta conseguirmos uma atribuição que satisfaça pelo menos um literal em cada cláusula.

No restante do texto vamos assumir que as fórmulas proposicionais sempre vão estar no formato CNF.

## 2.3 O arcabouço básico DPLL

Um solucionador SAT baseado em DPLL é normalmente um software com poucas linhas. A maioria desses solucionadores têm cerca de mil linhas de código, a maioria das vezes escritas em C ou C++ por razões de eficiência.

A base do DPLL é um algoritmo de busca com *retrocesso*. Esse algoritmo percorre o espaço de $2^n$ atribuições possíveis para as $n$ variáveis da instância SAT, organizando a busca através da manutenção de uma *árvore de decisão*. Cada nodo dessa árvore de decisão representa uma atribuição a uma variável, tais atribuições são conhecidas como *atribuições de decisão* e as variáveis são *variáveis de decisão*. Um *nível de decisão* é associado com cada atribuição de decisão para indicar a profundidade dessa atribuição na árvore de decisão. A primeira atribuição está no nível de decisão 1.

Em linhas gerais, o funcionamento segue os seguintes passos. Em cada nodo dessa árvore de busca, o algoritmo decide por uma atribuição de valor a uma variável. Em seguida, computa as implicações imediatas através da aplicação iterativa da *regra da cláusula unitária*. Por exemplo, se a atribuição escolhida for $x_1 = 1$, então a cláusula $(\neg x_1 \vee x_2)$ implica imediatamente que $x_2 = 1$. Essa atribuição, por sua vez, pode implicar em outras. A aplicação iterativa da regra da cláusula unitária é conhecida como Propagação de Restrição Lógica (*Boolean Constraint Propagation - BCP*).

Um resultado comum de BCP é que podemos chegar a um ponto em que as implicações geram um *conflito* (uma ou mais cláusulas insatisfeitas), deixando a fórmula insatisfazível. Neste caso, o procedimento tem que realizar o retrocesso. Por exemplo, se a fórmula que descrevemos no exemplo anterior também contiver a cláusula $(\neg x_1 \vee \neg x2)$ então claramente a decisão $x_1 = 1$ tem que ser alterada e as implicações da nova decisão devem ser recomputadas. Ao fazer isso, o algoritmo "poda" parte da árvore de busca. Se no momento em que um conflito for detectado ainda tivermos $n$ variáveis sem atribuição, uma sub-árvore de tamanho $2^n$ é "podada". Esse mecanismo é uma das principais razões da eficiência desse procedimento.

A estrutura geral de um procedimento baseado no DPLL pode ser encontrada na página 14.

O procedimento apresentado consiste de quatro funções principais:

**Decide()** que escolhe uma atribuição em cada estágio do processo de busca. Os procedimentos de decisão são baseados em heurísticas. Maiores detalhes sobre essas heurísticas podem ser vistos na seção 2.3.1.

**Deduz()** que implementa o BCP e mantém um grafo de implicações. Maiores detalhes sobre a implementação do BCP podem ser vistos na seção 2.3.2.

**Diagnostica()** que identifica as causas dos conflitos e pode acrescentar implicações adicionais ao banco de cláusulas, mecanismo que é chamado de *aprendizado direcionado por conflito* e é descrito na seção 2.3.3.

**Apaga()** que apaga a atribuição feita no nível atual de decisão e suas respectivas implicações.

Na literatura os procedimentos *Decide()*, *Deduz()* e *Diagnostica()* são considerados os três motores de um solucionador SAT. Diferentes implementações desses motores geram algoritmos SAT diferentes.

O funcionamento geral do procedimento *SAT()* é simples. A complexidade fica por conta da implementação dos motores, que é ignorada nessa seção. Seguindo o procedimento, podemos notar que a primeira ação, em um determinado nível de decisão $d$, é a escolha de uma atribuição de variável pelo motor *Decide()*. Se todas as variáveis já tiverem sidos atribuídas (indicado por *TODAS-ATRIBUÍDAS*), temos um conjunto de atribuições que satisfazem a fórmula proposicional e o solucionador devolve *SATISFATÍVEL*. Caso contrário, o motor *Deduz()* realiza o BCP. Se o BCP terminar sem a indicação de conflito, o procedimento é chamado recursivamente em um nível de decisão mais alto. Senão, o motor *Diagnostica()* analisa o conflito e decide qual o próximo passo a ser tomado. Se a variável tiver sido atribuída apenas uma vez, o motor inverte a atribuição e o motor *Deduz()* é executado novamente. Se a atribuição inversa também falhar, isso significa que a atribuição escolhida não é responsável pelo conflito. Neste caso, *Diagnostica()* identifica a atribuição que causou o conflito e o nível de decisão $\beta$ ($\beta$ é uma variável global que só pode ser alterada pelo motor *Diagnostica()*) para o qual *SAT()* deve retroceder. O procedimento vai retroceder $d - \beta$ vezes, executando a função *Apaga()* em cada vez.

Nas próximas subseções descrevemos os principais componentes (motores) do arcabouço DPLL.

**Algoritmo 1** Solucionador SAT Genérico

```
1: Entradas: d - nível de decisão atual
2:
3: Saídas:
4:     SAT():          {SATISFAZÍVEL,INSATISFAZÍVEL}
5:     Decide():       {ATRIBUÍ, TODAS-ATRIBUÍDAS}
6:     Deduz():        {OK, CONFLITO}
7:     Diagnostica(): {INVERTE, RETROCEDE}
8:
9: Variável global: β - nível de decisão para retrocesso
10:
11: procedure SAT(d)
12:
13:     if (Decide(d) == TODAS-ATRIBUÍDAS) then
14:         return SATISFAZíVEL;
15:     end if
16:
17:     while (TRUE) do
18:         if (Deduz(d) != CONFLITO) then
19:             if (SAT(d + 1) == SATISFAZíVEL) then
20:                 return SATISFAZíVEL;
21:             else if (β < d || d == 0) then
22:                 Apaga(d);
23:                 return(INSATISFAZÍVEL);
24:             end if
25:         end if
26:
27:         if (Diagnostica(d) == RETROCEDER) then
28:             return INSATISFAZÍVEL;
29:         end if
30:     end while
31:
32: end procedure
```

### 2.3.1 Heurísticas de Decisão

Um dos pontos críticos no desempenho do solucionador SAT são as heurísticas de decisão utilizadas no motor *Decide()*, ou em outras palavras, a estratégia de escolha da próxima variável a ser atribuída e qual o valor deve ser atribuído a essa variável. A ordem de escolha pode ser estática (predeterminada) ou decidida dinamicamente de acordo com o atual estado da busca.

No decorrer dos anos várias heurísticas diferentes foram propostas por diferentes pesquisadores. Naturalmente surgiram estudos comparativos [23, 24].

As primeiras heurísticas como a de Bohm [25], a MOM (que busca a variável com o máximo de ocorrências em cláusulas de tamanho mínimo)(e.g. [14]) e a Jeroslow-Wang [26] são consideradas heurísticas gulosas porque tentam gerar o maior número de implicações ou satisfazer a maioria das cláusulas. Elas são úteis para instâncias aleatórias do SAT mas normalmente não tem um desempenho destacado em instâncias estruturadas (que são geralmente o caso de aplicações do mundo real).

Em [23], o autor propôs o uso de heurísticas baseadas na contagem de literais. Essas heurísticas contam o número de cláusulas não resolvidas em que uma dada variável em qualquer uma de suas fases aparece. A heurística é chamada de Maior Soma Dinâmica Combinada (*Dynamic Largest Combined Sum - DLIS*). A contagem é dependente do estado atual da busca, ou seja, diferentes atribuições levam a diferentes contagens. Dessa forma toda vez que o motor *Deduz()* é chamado, as contagens são recalculadas, o que gera sobrecarga no processamento.

Em [19], o autor propôs uma heurística chamada Soma Decadente Independente do Estado da Variável (*Variable State Independent Decaying Sum - VSIDS*). VSIDS mantém uma pontuação para cada fase da variável. Inicialmente a pontuação é simplesmente o número de ocorrências do literal na fórmula. Na medida que novas cláusulas que contém uma variável são adicionadas ao banco de cláusulas (pelo mecanismo de *Aprendizado* descrito na seção 2.3.3) o VSIDS aumenta a pontuação dessa variável por uma constante. Além disso, na medida que a busca progride as pontuações são divididas por uma constante. A conseqüência desse mecanismo é que as variáveis com maior pontuação tendem a ser aquelas presentes nas cláusulas adicionadas mais recentemente. A variável com maior pontuação é a escolhida pelo motor *Deduz()*. Por ser independente do estado das variáveis o mecanismo é mais barato de se manter. Experimentos mostraram não só que o mecanismo é bem competitivo mas também mostraram que o procedimento usa apenas uma pequena porcentagem do tempo total de processamento. A implemen-

tação dessa heurística se mostrou mais rápida, em uma ordem de magnitude, na média, se comparada a DLIS. Essa heurística é usada pelo moderno solucionador Chaff e por sua eficiente implementação, o ZChaff.

Mais recentemente, [17] propôs um evolução no esquema proposto pelo VSIDS. Assim como o esse último, a estratégia é buscar por variáveis que foram "ativas" recentemente. A diferença entre os dois esquemas é que enquanto VSIDS cria sua pontuação baseada nas ocorrências das variáveis o novo esquema faz sua pontuação baseada nos conflitos. Da seguinte maneira: quando um conflito ocorre, todos os literais que foram responsáveis pelo conflito tem sua pontuação aumentada. A pontuação nesse esquema também é divida por uma constante no decorrer do tempo, privilegiando também as variáveis presentes nos conflitos mais recentes. E vai além, o motor *Deduz()* exige que a variável esteja na cláusula mais recentemente adicionada que não foi ainda resolvida. Os experimentos mostraram que esse esquema é mais robusto que o VSIDS. Essa heurística é usada por outro moderno solucionador, o BerkMin.

### 2.3.2 Mecanismos de Dedução

O motor *Deduz()* "poda" o espaço de buscas através de implicações geradas a partir da atribuição de uma variável.

Através de vários anos vários mecanismos de dedução foram propostos. Entretanto, parece que o mecanismo mais eficiente é a regra da cláusula unitária [16]. Esse mecanismo requer relativamente pouco poder computacional e pode "podar" grandes árvores de busca. Essa regra determina que o literal restante na cláusula unitária deve ser atribuído com o valor 1, o que é essencial para satisfazer a instância SAT. Como já foi dito anteriormente a aplicação iterativa da regra da cláusula unitária, em busca de novas cláusulas unitárias ou de conflito, é conhecida como BCP. Todos os solucionadores SAT modernos incluem essa regra no seu motor de dedução.

O BCP normalmente é a etapa que ocupa maior parte do tempo de processamento. Portanto sua implementação influi diretamente na eficiência do algoritmo.

Uma implementação simples e intuitiva para o BCP é manter contadores para cada cláusula. O GRASP [12] usa essa idéia. Nesse solucionador cada cláusula mantém dois contadores, um para o número de literais que receberam o valor 1 e outro para contar os literais que receberam o valor 0. Cada variável mantém duas listas: uma com as cláusulas em que ela aparece como um literal positivo e outra com as cláusulas em que ela aparece como um literal negativo. Quando uma variável é atribuída todas as cláusulas que contém essa variável em uma das suas fases tem seus contadores atualizados.

Se o contador de literais de valor 0 de uma cláusula conter um valor igual ao número total de literais nessa cláusula temos um conflito. Se esse valor for menor em uma unidade se comparado ao número total de literais temos uma cláusula unitária. Esse mecanismo é fácil de entender e implementar mas não é o mais eficiente.

Uma implementação interessante de BCP foi descrita pelos criadores do Chaff em [19]. Essa implementação foi chamada de *vigilância por dois literais*. A idéia é baseada no seguinte princípio: se um cláusula tem $n$ literais, então só precisamos nos preocupar com ela quando $n-1$ forem atribuídos com 0. Ou seja, só vamos aplicar a regra da cláusula unitária quando o número de literais atribuídos com 0 passar de $n-2$ para $n-1$.

Na implementação, cada cláusula tem dois literais especiais que são denominados *literais de vigilância*. Inicialmente nenhum literal de vigilância está atribuído. Quando um desses literais é atribuído com 0 uma das seguintes condições tem que valer:

1. Existe um literal na cláusula que não foi atribuído com 0 e não é o outro literal de vigilância. Então simplesmente substituí-se o literal de vigilância.

2. Apenas o outro literal de vigilância não está atribuído. Então a cláusula é unitária.

3. Apenas o outro literal de vigilância não está atribuído com 0. Mas, está atribuído com 1. Então não precisamos fazer nada pois, a cláusula já foi satisfeita.

4. Não há outro literal que não tenha sido atribuído com 0. Temos um conflito.

Esse mecanismo apresenta duas grandes vantagens:

- Precisamos apenas examinar cláusulas onde os literais de vigilância recebem o valor 0.

- Não precisamos alterar os literais de vigilância durante um eventual retrocesso. Já que por terem sido os últimos a serem atribuídos com 0, ou eles perderão a atribuição ou receberão 1 como valor.

Vamos ilustrar esse mecanismo com um exemplo. Dada a seguinte fórmula proposicional:

1. $(\mathbf{b} \vee \mathbf{c} \vee a \vee d \vee e) \wedge$
2. $(\mathbf{a} \vee \mathbf{b} \vee \neg\, c) \wedge$
3. $(\mathbf{a} \vee \neg\, \mathbf{b}) \wedge$
4. $(\neg\, \mathbf{a} \vee \mathbf{d}) \wedge$
5. $(\neg\, a)$

Os literais em negrito são os literais de vigilância de cada cláusula. Observe que a última cláusula é uma cláusula unitária e portanto não pode ter dois literais de vigilância. Mas sendo uma cláusula unitária ela não é um problema e sim parte da solução. A partir dela já sabemos que a atribuição $a = 0$ é necessária.

Temos que encontrar as cláusulas que tem fase positiva de $a$ como literais de vigilância já que estes serão atribuídos com 0. No exemplo são as cláusula 2 e 3.

Na cláusula 2 basta substituirmos o literal de vigilância $a$ pelo literal $\neg c$:

1. $(\mathbf{b} \vee \mathbf{c} \vee a \vee d \vee e) \wedge$
2. $(\neg\, \mathbf{c} \vee \mathbf{b} \vee a) \wedge$
3. $(\mathbf{a} \vee \neg\, \mathbf{b}) \wedge$
4. $(\neg\, \mathbf{a} \vee \mathbf{d}) \wedge$
5. $(\neg\, a)$

Atribuições: $\{a = 0\}$

Na cláusula 3 o único literal disponível é o outro literal de vigilância, que ainda não foi atribuído. Portanto temos agora uma cláusula unitária. E a seguinte implicação: $b = 0$.

Buscamos então por cláusulas que tem fase positiva de $b$ como literais de vigilância já que estes serão atribuídos com 0. No exemplo são as cláusula 1 e 2.

Na cláusula 1 basta substituirmos o literal de vigilância $b$ pelo literal $d$, já que esse não foi atribuído com 0.

1. $(\mathbf{d} \vee \mathbf{c} \vee a \vee b \vee e) \wedge$
2. $(\neg\, \mathbf{c} \vee \mathbf{b} \vee a) \wedge$
3. $(\mathbf{a} \vee \neg\, \mathbf{b}) \wedge$
4. $(\neg\, \mathbf{a} \vee \mathbf{d}) \wedge$
5. $(\neg\, a)$

Atribuições: $\{a = 0, b = 0\}$

Na cláusula 2 o único literal disponível é o outro literal de vigilância, que ainda não foi atribuído. Portanto temos agora uma cláusula unitária, e a seguinte implicação: $c = 0$.

Buscamos então por cláusulas que tem fase positiva de $c$ como literais de vigilância já que estes serão atribuídos com 0. No exemplo é apenas a cláusula 1.

Na cláusula 1 basta substituirmos o literal de vigilância $c$ pelo literal $e$, já que esse não foi atribuído com 0.

1. $(\mathbf{e} \vee \mathbf{c} \vee a \vee b \vee d) \wedge$
2. $(\neg\, \mathbf{c} \vee \mathbf{b} \vee a) \wedge$
3. $(\mathbf{a} \vee \neg\, \mathbf{b}) \wedge$
4. $(\neg\, \mathbf{a} \vee \mathbf{d}) \wedge$
5. $(\neg\, a)$

Atribuições: $\{a = 0, b = 0, c = 0\}$

Não temos nenhuma implicação pendente. O solucionador tem que decidir por uma atribuição. Suponhamos que ele escolha por atribuir $d = 1$.

Buscamos então por cláusulas que tem fase negativa de $d$ como literais de vigilância já que estes serão atribuídos com 0. No exemplo não temos nenhuma cláusula que atende a essa característica.

Novamente não temos nenhuma implicação pendente. O solucionador tem que decidir por uma nova atribuição. Suponhamos que ele escolha por atribuir $e = 0$.

Buscamos então por cláusulas que tem fase positiva de $e$ como literais de vigilância já que estes serão atribuídos com 0. No exemplo é apenas a cláusula 1.

Na cláusula 1 o único literal disponível é o outro literal de vigilância, que já foi atribuído anteriormente. Portanto não temos que fazer nada.

Temos então a seguinte solução para o exemplo:

1. $(\mathbf{e} \vee \mathbf{c} \vee a \vee b \vee d) \wedge$
2. $(\neg\, \mathbf{c} \vee \mathbf{b} \vee a) \wedge$
3. $(\mathbf{a} \vee \neg\, \mathbf{b}) \wedge$
4. $(\neg\, \mathbf{a} \vee \mathbf{d}) \wedge$
5. $(\neg\, a)$

Atribuições: $\{a = 0, b = 0, c = 0, d = 1, e = 0\}$

### Grafo de Implicações

As implicações geradas durante o BCP podem ser representadas por um grafo de implicações. Um grafo de implicações típico é mostrado na figura 2.1.

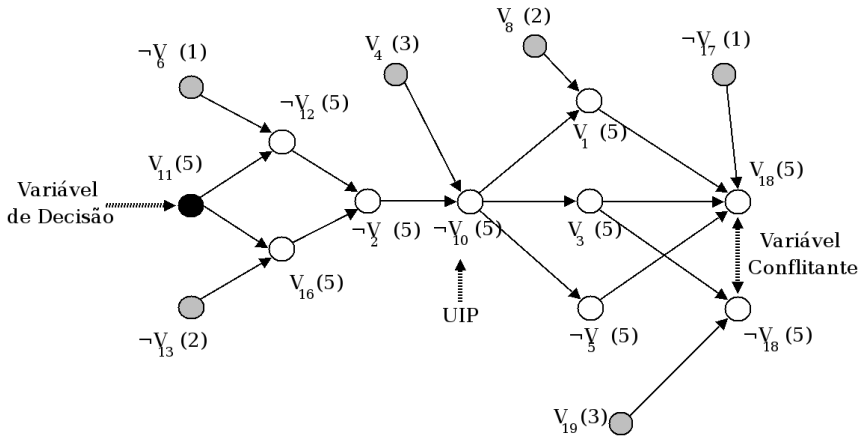


Figura 2.1: Um grafo de implicações típico

Um grafo de implicações é um grafo direcionado acíclico. Cada nodo representa uma atribuição de variável. A fase positiva significa que a atribuição foi feita com o valor 1, a fase negativa indica o valor 0.

Os arcos direcionados mostram as implicações. Variáveis presentes em nodos sem antecedentes são variáveis de decisão (nodos cinza). Cada variável tem um nível de decisão associado, indicado com um número entre parênteses.

Um grafo de implicações sem conflitos tem no máximo um nodo para cada variável. Um conflito ocorre quando há tanto a atribuição 0 quanto a atribuição 1 para uma variável. Essa variável é conhecida como *variável conflitante*. Na figura 2.1 a variável $V_{18}$ é a variável conflitante.

No grafo de implicações um nodo $a$ domina um nodo $b$ se e somente se qualquer caminho do nodo da variável de decisão do nível de decisão de $a$ para o nodo $b$ passa pelo nodo $a$. Um **Ponto de Implicação Único** (*Unique Implication Point - UIP*) [12] é um nodo no nível atual de decisão que domina ambos os nodos da variável conflitante. Em outras palavras, o UIP é a razão única que leva ao conflito em um determinado nível de decisão.

Na figura 2.1 por exemplo, no nível de decisão 5, $V_{10}$ domina $\neg V_{18}$ e $V_{18}$, e portanto é um UIP. Uma variável de decisão é sempre um UIP e podem existir mais de um UIP para um conflito. No exemplo existem três UIPs: $V_{10}$, $V_2$ e $V_{11}$.

Grafos de implicações são usados no entendimento do funcionamento do motor *Diagnostica()* e do mecanismo de Aprendizado.

### 2.3.3 Diagnóstico, Retrocesso e Aprendizado

Sempre que um conflito é encontrado o solucionador precisa encontrar a causa dele e tentar resolvê-lo. O motor que realiza essa tarefa é o *Diagnostica()* que depois de identificar a razão do conflito indica como deve ser feito o retrocesso na busca.

O algoritmo DPLL original propôs um método de análise de conflito bastante simples. Para cada variável decidida, o solucionador mantém um sinal indicando se já foram tentadas as duas fases dessa variável. Quando um conflito ocorre, o algoritmo procura pela variável decidida com maior nível de decisão em que apenas uma fase foi tentada. O algoritmo então marca o sinal indicando que as duas fases foram tentadas, retrocede a busca até o nível de decisão dessa variável e então inverte o valor dela. Esse método é chamado de *retrocesso cronológico*. Esse tipo de retrocesso funciona bem para instâncias randômicas de SAT.

Para instâncias SAT estruturadas (que são geralmente o caso de aplicações do mundo real), o retrocesso cronológico não é suficiente. Motores mais avançados de diagnóstico buscam encontrar e entender a causa direta para o conflito. Esses motores normalmente retrocedem o processo de busca a níveis de decisão anteriores aos do retrocesso cronológico. E por isso o método que usam é chamado de *retrocesso não-cronológico* e foi proposto inicialmente no domínio dos problemas de satisfação de restrições (*Constraint Satisfaction Problem - CSP*, e.g. [27]) .

Durante o diagnóstico do conflito, informação sobre o conflito pode ser adicionada ao banco de cláusulas. As cláusulas adicionadas, embora sejam redundantes no sentido que não mudam a satisfiabilidade do problema original, impedem que o mesmo erro aconteça no futuro, "podando" a árvore. Esse mecanismo é chamado de *aprendizado direcionado por conflito*. As cláusulas adicionadas são chamadas de cláusula de *conflito*.

O retrocesso não-cronológico em conjunto com o aprendizado direcionado por conflito, foi incorporado em solucionadores SAT pela primeira vez em [12, 28]. Esse foi um dos principais avanços que permitiram que esses procedimentos tratassem instâncias do problema SAT com dezenas de milhares de variáveis.

De maneira geral podermos descrever a geração de uma cláusula de conflito da seguinte maneira. O grafo de implicações é dividido por um corte em duas partições. Uma partição que tem todas as variáveis de decisão (chamada de *lado da decisão*) e a outra que tem todas as variáveis conflitantes

(chamada de *lado do conflito*). Todos os nodos no lado de decisão que tem pelo menos um arco direcionado para o outro lado fazem parte da causa do conflito. Um grafo de implicações biparticionado pode ser visto na figura 2.2.

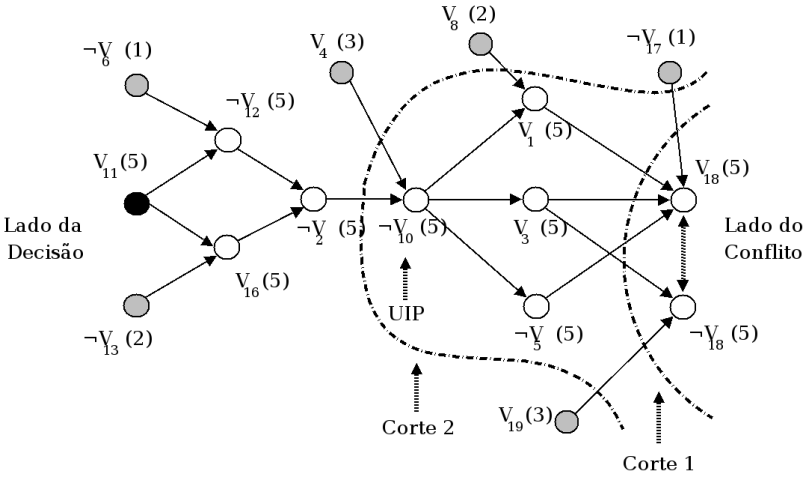


Figura 2.2: Grafo de Implicações Bipartido com diferentes cortes

Para formar uma bipartição temos que definir o corte. Cortes diferentes correspondem a diferentes esquemas de aprendizado. Na figura 2.2 a partir do corte 1 temos que a seguinte atribuição nos levou a um conflito:

$$\{V_1 = 1, V_3 = 1, V_5 = 0, V_{17} = 0, V_{19} = 1\}$$

A partir dessa atribuição podemos gerar a seguinte cláusula de conflito:

$$\begin{aligned} \text{Conflito} &= (V_1 \wedge V_3 \wedge \neg V_5 \wedge \neg V_{17} \wedge V_{19}) \\ \neg \text{ Conflito} &= \neg(V_1 \wedge V_3 \wedge \neg V_5 \wedge \neg V_{17} \wedge V_{19}) \\ \text{Cláusula de Conflito} &= (\neg V_1 \vee \neg V_3 \vee V_5 \vee V_{17} \vee \neg V_{19}) \end{aligned}$$

Semelhantemente, o corte 2 corresponde a cláusula:

$$(V_2 \vee \neg V_4 \vee \neg V_8 \vee V_{17} \vee \neg V_{19})$$

Existem muitos outros esquemas de aprendizado direcionado por conflito, muitos deles explorando o conceito de UIP [12]. Os detalhes desses diferentes esquemas podem ser encontrados em [29, 21].

## 2.4 Competição SAT

A competição SAT [30] é um concurso que vem sendo organizado desde 2002 e ocorre em conjunto com o Simpósio Internacional de Teoria e Aplicação de Provas de Satisfiabilidade.

O propósito dessa competição é identificar novos *benchmarks* desafiadores e promover novos solucionadores SAT assim como compará-los com os solucionadores considerados no estado-da-arte.

A competição é toda realizada utilizando o sistema SAT-Ex [31].

Em 8 de maio de 2003 encerrou-se a segunda competição SAT que havia começado em 14 de fevereiro do mesmo ano. Os organizadores temiam que passado apenas um ano grandes progressos não seriam alcançados. Mas para a surpresa deles vários competidores foram capazes de resolver em quinze minutos instâncias SAT que não eram resolvidas em menos de seis horas pelos melhores colocados da primeira competição.

A plataforma de testes foram dois clusters homogêneos de máquinas Linux. O primeiro cluster, localizada no *Laboratoire de Recherche en Informatique* (LRI, Orsay, França), era composto de quinze computadores, todos com processadores Athlon 1800+ com 1 Gb de memória RAM. O segundo, localizado no *Dipartimento di Informatica Sistemica e Telematica* (DIST Genoa, Itália), era composto de oito computadores, todos com processadores Pentium IV 2.4 GHz com 1 Gb de memória RAM. Ambos usavam o sistema operacional Red Hat Linux 7.2.

Os problemas a serem resolvidos foram divididos em três categorias:

**Problemas Industriais** criados a partir de aplicações industriais reais.

**Problemas Aleatórios** criados partir de algum critério mais geral, como um número fixo de variáveis e de cláusulas.

**Problemas Preparados Manualmente** criados cuidadosamente buscando explorar características específicas das fórmulas.

Por causa dos limites de solucionadores estocásticos cada categoria foi dividida ainda em duas: problemas satisfazíveis e problemas insatisfazíveis. Mas como esse tipo de solucionadores não são importantes para BMC vamos ignorar essa classificação.

A idéia inicial era premiar o melhor solucionador genérico, que teria a melhor performance nas três categorias. Entretanto para os organizadores não fazia sentido comparar solucionadores voltados para resolver instâncias aleatórias com outros preparados para lidar com instâncias geradas a partir de problemas da indústria. Passou-se a premiar por categoria.

A competição de 2003 foi então dividida em três fases:

1. A primeira fase é uma etapa preparatória tanto para os competidores quanto para os organizadores. Os competidores podiam testar seus solucionadores com os padrões exigidos pela competição e com os sistemas operacionais onde eles seriam executados. Os organizadores por sua vez testaram os solucionadores em benchmarks que não participariam da competição. Os solucionadores que não funcionaram corretamente foram devolvidos aos seus desenvolvedores que tinham uma chance de reparar os erros. Os organizadores realizaram os testes uma segunda vez eliminando os solucionadores que persistiram em erros. Outra tarefa dos solucionadores foi a avaliação dos benchmarks e a separação dos mesmos nas categorias.

2. Na segunda fase todos os solucionadores foram testados com todos os benchmarks. Se o solucionador não dava uma resposta dentro de um tempo limitado (que foi definido de acordo com o número de participantes) ele era interrompido.

3. Na terceira fase os organizadores selecionavam os melhores solucionadores de cada categoria e então executavam eles com as menores instâncias não resolvidas de cada categoria. Nessa fase o tempo de execução era substancialmente maior.

Os benchmarks desenvolvidos a partir de problemas de Verificação de Modelos estão nessa categoria. Dessa forma apenas os resultados dessa categoria interessam para esta dissertação.

| Problemas Industriais | |
|---|---|
| Solucionador | Problemas Solucionados |
| forklift | 12 |
| berkmin561 | 11 |
| satzoo1 | 5 |
| jerusat1b | 5 |
| satnik | 4 |
| zchaff | 4 |
| funex | 3 |
| oepir | 1 |
| jquest2 | 0 |
| limmat | 0 |

Tabela 2.1: Resultado Final da Competição SAT 2003 na Categoria de Problemas Industriais

Como podemos observar na Tabela 2.1, dos dez solucionadores que passaram à terceira fase na categoria de problemas industriais, oito conseguiram resolver pelo menos uma instância.

Os seis primeiros solucionadores podem ser descritos da seguinte forma:

1. **forklift** O grande vencedor pode ser visto como uma extensão do solucionador BerkMin62. Infelizmente não sabemos as melhorias implementadas no BerkMin62 e no ForkLift, pois estes trabalhos não foram publicados e não tem o seu código disponibilizado. Sabemos apenas que os autores consideram essas versões superiores ao BerkMin original.

2. **berkmin561** O segundo colocado também é um implementação do BerkMin. Porém, tanto o trabalho [17] quanto o código [32] estão disponíveis publicamente.

3. **satzoo1** O terceiro colocado é um solucionador similar ao Chaff [19]. Ele foi desenvolvido com o propósito principal de ser capaz de se integrar com novas técnicas de Verificação de Modelos. As técnicas usadas por esse solucionador estão descritas em [33]. Seu código fonte está disponível em [34].

4. **jerusat1b** Neste solucionador o autor escreveu um algoritmo novo baseado no DPLL. Também está disponível para o público em [35].

5. **satnik** Este é outro solucionador baseado no Chaff. Suas inovações também estão descritas em [33]. Informações sobre o solucionador e trabalhos do autor podem ser encontrados em [36]. O código não está disponível para download mas talvez possa ser obtido através de um pedido direto para o autor.

6. **zchaff** Este é o vencedor da competição de 2002. O zChaff é um implementação do Chaff com foco em performance e capacidade. Se código pode ser obtido em [37].

Dados os resultados e as descrições podemos deduzir que, para *Bounded Model Checking*, os trabalhos mais interessantes sobre o problema SAT são aqueles baseados no BerkMin [17] e no Chaff [19].

# Capítulo 3

# *Bounded Model Checking* com solucionadores SAT

## 3.1 Introdução

A Verificação de Modelos foi proposta pela primeira vez como uma técnica de verificação cerca de vinte anos atrás. Essa técnica engloba algoritmos para a verificação de propriedades de sistemas através da busca no espaço de estados desses sistemas. As propriedades a serem verificadas são escritas em lógica temporal, um formalismo para o raciocínio sobre a ordenação de eventos no tempo sem a introdução explícita de medidas de tempo. Em lógica temporal pode se, por exemplo, expressar uma propriedade que não é verdadeira no presente mas pode eventualmente ser no futuro. Ou, por exemplo, pode se expressar que essa propriedade inevitavelmente vai ser tornar verdadeira no futuro. Um linguagem de especificação rica combinada com um alto nível de automação fez da Verificação de Modelos uma técnica muita atraente para a indústria.

As primeiras implementações de verificadores de modelos aparecerem no começo da década de 80 e usavam representações explícitas dos grafos de transições de estados e tentavam explorá-las usando técnicas eficientes de caminhamento em grafos. Entretanto, o problema de explosão de estados, onde o tamanho do espaço de estados crescia exponencialmente com o número de componentes do sistemas, geralmente limitava essas técnicas a modelos com menos de um milhão de estados. O que tornava o método inadequado para aplicações industriais.

Na começo da década de 1990 surgiram técnicas que usavam a exploração simbólica de estados. Na Verificação de Modelos Simbólica [3, 4] uma busca em largura do espaço de estados era efetuada através do uso de diagramas

de decisão binários. (BDDs [5]). Os BDDs representavam as funções características do conjunto de estados e permitiam a computação das transições entre conjuntos de estado ao invés de estados individuais.

O primeiro verificador simbólico baseado em BDDs foi capaz de verificar modelos de significativa complexidade, como o protocolo de verificação de consistência de cache Futurebus+ [38]. No entanto, enquanto essas técnicas aumentaram em uma ordem de magnitude o tamanho dos modelos verificáveis, esse tamanho apenas se igualou ao dos menores componentes ditos "interessantes" industrialmente. Durante a década de 1990 a capacidade dos verificadores baseados em BDDs aumentou, através de melhorias na implementação dos BDDs e no progresso no uso de técnicas como abstração e raciocínio composicional. Esses avanços permitiram que os verificadores se tornassem ferramentas CAD comerciais. Mas ainda assim não deram a eles poder suficiente para representar modelos do tamanho que um usuário típico de indústria gostaria.

No final da década de 1990 uma nova técnica, *Bounded Model Checking* (BMC) com solucionadores SAT [39, 6, 7], trouxe resultados promissores. Esse método pode ser aplicado para propriedades de sobrevivência (*liveness*) e segurança (*safety*), onde a verificação de propriedades de segurança envolve verificar se um dado conjunto de estados é alcançável e a verificação de propriedades de sobrevivência envolve a busca por laços no grafo de transições de estados. Informalmente, uma propriedade de segurança estipula que "coisas ruins" não aconteçam durante a execução de um programa e propriedade de sobrevivência estipulam que "coisas boas" acontecem (eventualmente) [40].

Um exemplo simples de propriedade de segurança é um invariante, que é uma propriedade que simplesmente tem que ser verdadeira em todos os estados alcançáveis do modelo. Obviamente se existe uma seqüência de estados em que algum estado tem um suposto invariante falso, a propriedade não pode ser chamada de invariante. Experimentos mostraram que buscas por contra-exemplos desse tipo são feitas com notável eficiência através de BMC, mesmo em modelos que seriam difíceis de serem tratados com BDDs. Outra vantagem comparativa é que BMC requer pouca intervenção manual enquanto BDDs exigem a ordenação de variáveis e o uso de algumas abstrações manuais.

A robustez e a escalabilidade de BMC fez essa técnica muita atrativa para a indústria. Além das vantagens já citadas existe ainda o fato de que os solucionadores SAT como o GRASP [12, 13], o SATO [18], o Chaff [19] e o algoritmo de Stålmarck [41] raramente precisam de espaço exponencial, o oposto do comportamento dos BDDs.

O BMC não é um método completo pois na maioria das vezes não verifica o espaço de estados total. A outra falha do BMC são os poucos tipos de

propriedades que podem ser verificadas.

Na próxima seção vamos explicar um pouco mais sobre o funcionamento da Verificação de Modelos com foco na lógica temporal. Na última seção descrevemos com detalhes o que é e como funciona o BMC.

## 3.2 Lógicas Temporais

Muitos projetos, especialmente projetos de hardware, podem ser modelados como sistemas de transição de estados com a finalidade da verificação da sua corretude. A Verificação de Modelos oferece um meio atraente para se fazer questionamentos sobre um sistema de transição de estados. Nessa técnica, alguém descreve uma propriedade de um sistema em *lógica temporal* [2] e então executa o procedimento que vai percorrer o grafo de transição de estados e determinar onde a propriedade é atendida. O tipo de procedimento que percorre o grafo de transição vai variar com a lógica temporal e o tipo de fórmula utilizada pela propriedade.

Lógicas temporais se mostraram ao longo do tempo bastante úteis para a especificação de sistemas. Elas foram originalmente desenvolvidas por filósofos para a investigação da maneira com a qual o tempo é usado em argumentos de linguagem natural [42]. Embora um bom número de lógicas temporais foi estudado, a maioria tem um operador como $\mathbf{G}f$ que é verdadeiro no presente se $f$ for sempre verdadeiro no futuro (i.e., se $f$ é **g**lobalmente verdadeiro). As lógicas são geralmente classificadas de acordo com a estrutura que é assumida para o tempo, que varia entre *linear* ou *ramificada*. Na lógica temporal linear os operadores descrevem eventos sobre um único caminho. Na lógica temporal ramificada os operadores quantificam sobre os caminhos que são possíveis a partir de um determinado estado.

Nesse trabalho vamos nos concentrar na chamada lógica temporal linear (LTL) [43]. O algoritmo de BMC original trabalha com essa lógica [6]. Existem trabalhos que mostram como usar BMC com outras lógicas como CTL [44]. Como LTL foi pioneira com BMC é razoável pensarmos que os algoritmos que vamos encontrar para eles estejam mais maduros do que as versões para outras lógicas, o que justifica a escolha.

LTL, quando comparada com outras lógicas, é normalmente aclamada como mais intuitiva, embora ninguém tenha provado tal fato. No entanto é certo que a Verificação de Modelos com fórmulas LTL tem alta complexidade [45, 46, 47]. Por muitos anos a viabilidade de um verificador LTL era questionada até que em 1982 foi publicado um algoritmo LTL que era exponencial no tamanho da fórmula, mas linear no tamanho do modelo [48]. Desde então diversas otimizações vem mostrando que a verificação com LTL

é viável. BMC mostrou que modelos reais podem ser verificados com LTL.

Maiores informações sobre lógicas temporais e como Verificação de Modelos é realizada com essas lógicas podem ser encontradas em [49, 50, 51, 52, 53, 54, 55] e [2, 56] respectivamente.

Nas próximas subseções vamos descrever o que é LTL.

### 3.2.1 Sintaxe das Fórmulas *LTL*

A sintaxe das fórmulas LTL é definida pelas seguinte regras:

1. Cada *proposição atômica* $p$ é uma fórmula (com as quais as propriedades dos estados são representadas).
2. Se $f$ e $g$ são fórmulas então $\neg f$, $f \wedge g$ e $f \vee g$ são fórmulas.
3. Se $f$ é uma fórmula então $\mathbf{X}\, f$ é uma fórmula.
4. Se $f$ é uma fórmula então $\mathbf{F}\, f$ é uma fórmula.
5. Se $f$ é uma fórmula então $\mathbf{G}\, f$ é uma fórmula.
6. Se $f$ e $g$ são fórmulas então $f\, \mathbf{U}\, g$ é uma fórmula.
7. Se $f$ e $g$ são fórmulas então $f\, \mathbf{R}\, g$ é uma fórmula.

Adiantando parte da semântica:

- O operador **X** requer que a propriedade que o sucede seja verdadeira no próximo estado. A letra X é oriunda da palavra inglesa *next*.
- O operador **F** requer que a propriedade que o sucede seja verdadeira em algum estado futuro. A letra F é oriunda da palavra inglesa *future*.
- O operador **G** requer que a propriedade que o sucede seja verdadeira em todos os próximos estados, inclusive o estado atual. A letra G é oriunda da palavra inglesa *global*.
- O operador **U** é usado para combinar duas propriedades. Ele requer que a segunda propriedade seja verdadeira em um determinado estado futuro ou presente e em todos os estados entre o presente e esse estado a primeira propriedade seja verdadeira. A letra U é oriunda da palavra inglesa *until*.

- O operador **R** também é usado para combinar duas propriedades. Ele requer que a segunda propriedade seja sempre verdadeira no caminho até um estado em que a primeira seja verdadeira, inclusive. Entretanto a primeira propriedade não é obrigada a ser verdadeira, deixando como única alternativa manter a segunda propriedade sempre verdadeira. A letra R é oriunda da palavra inglesa *release*.

### 3.2.2 Estruturas *Kripke*

Em Verificação de Modelos o significado de uma fórmula de lógica temporal é sempre determinado com respeito a um grafo de transição de estados rotulado, por razões históricas, estruturas desse tipo são chamadas *Kripke* [42].

**Definição 2** *Seja AP um conjunto de proposições atômicas. Uma estrutura* Kripke *M sobre AP é uma tupla* $M = (S, I, T, L)$ *onde:*

1. $S$ *é um conjunto finito de estados.*
2. $I \subseteq S$ *é o conjunto de estados iniciais.*
3. $T \subseteq S \times S$ *é a relação de transição que tem que ser total, isto é, para cada estado* $s \in S$ *tem que existir um estado* $s' \in S$ *tal que* $T(s, s')$.
4. $L : S \rightarrow 2^{AP}$ *é a função que rotula cada estado com o conjunto de proposições atômicas que são verdadeiras naquele estado.*

*Obs.: Para* $(s, t) \in T$ *nós também podemos escrever* $s \rightarrow t$.

As fórmulas LTL descrevem eventos sobre caminhos. Definimos um *caminho* sobre uma estrutura *Kripke* da seguinte forma:

**Definição 3** *Seja M uma estrutura* Kripke *e* $\pi$ *um* caminho*:*

1. *Um caminho em M a partir de um estado s é uma seqüência infinita de estados* $\pi = s_0 s_1 s_2 \ldots$ *tal que* $s_0 = s$ *e* $T(s_i, s_{i+1})$ *existe para qualquer* $i \geq 0$.
2. $\pi(i) = s_i$
3. $\pi^i = (s_i, s_{i+1}, \ldots)$

### 3.2.3 Semântica das Fórmulas *LTL*

A semântica LTL é definida pela relação $\pi \models f$:

**Definição 4** *Seja $M$ uma estrutura* Kripke, *$\pi$ um caminho em $M$ e $f$ e $g$ fórmulas LTL. Então a relação $\pi \models f$ (a fórmula $f$ é válida no caminho $\pi$) é definida da seguinte maneira:*

$\pi \models p$ *iff* $p \in l(\pi(0))$
*(O primeiro estado de $\pi$ é rotulado com $p$)*

$\pi \models \neg f$ *iff* $f \notin l(\pi(0))$
*($f$ não é válida em $\pi$)*

$\pi \models f \wedge g$ *iff* $\pi \models f$ *e* $\pi \models g$
*($f$ e $g$ são válidas em $\pi$)*

$\pi \models f \vee g$ *iff* $\pi \models f$ *ou* $\pi \models g$
*($f$ ou $g$ são válidas em $\pi$)*

$\pi \models \boldsymbol{G}\, f$ *iff* $\forall i.\pi^i \models f$
*($f$ é válida em todo sufixo de $\pi$)*

$\pi \models \boldsymbol{F}\, f$ *iff* $\exists i.\pi^i \models f$
*($f$ é válida em algum sufixo de $\pi$)*

$\pi \models \boldsymbol{X}\, f$ *iff* $\pi^1 \models f$
*($f$ é válida no segundo estado de $\pi$)*

$\pi \models f\ \boldsymbol{U}\, g$ *iff* $\exists (i \geq 0)[\pi^i \models g\, e\, \forall j, j < i.\pi^j \models f]$
*($f$ é válida até o estado em que $g$ seja válida)*

$\pi \models f\ \boldsymbol{R}\, g$ *iff* $\forall (i \geq 0)[\pi^i \models g\, ou\, \exists j, j < i.\pi^j \models f]$
*(Para todos os estados do caminho ou $g$ é valida ou existe um estado antecessor em que $f$ foi válida)*

### 3.2.4 Exemplos de Fórmulas *LTL*

Alguns exemplos simples de fórmulas LTL:

- Exclusão Mútua:

$$\mathbf{G}\neg(critical_1 \wedge critical_2)$$

- No máximo uma requisição é respondida:

$$\mathbf{G} \bigwedge_{i<j} \neg(ack_i \wedge ack_j)$$

- Sempre vou ter minha vez.:

$$\mathbf{GF}\, minhaVez$$

- Entrar na região *tentativa* vai necessariamente levar à região *crítica*:

$$\mathbf{G}(tentativa \rightarrow \mathbf{F}\, crítica)$$

- Para toda requisição vou necessariamente uma resposta:

$$\mathbf{G}(req \rightarrow \mathbf{F}\, ack)$$

- Após a inicialização, o sistema permanecerá inicializado:

$$\mathbf{FG}\, inicializado$$

## 3.3 *Bounded Model Checking*

Nessa seção detalhamos a técnica de verificação simbólica de modelos baseada em solucionadores SAT. Nessa técnica, batizada de *bounded model checking*, construímos uma fórmula proposicional que é satisfazível se e somente se o modelo verificado tem um seqüência finita de estados que alcance estados de interesse. Essa busca é feita limitando o tamanho máximo dessa seqüência em $k$. Se para um determinado $k$ não for encontrado um contra-exemplo, aumenta-se o valor de $k$.

O limite $k$ pode ser interpretado também como o tamanho máximo do contra-exemplo.

As maiores vantagens dessa técnicas são as seguintes:

1. Contra-exemplos são encontrados velozmente.
2. Os contra-exemplos encontrados são de tamanho mínimo, que são mais fáceis de entender.
3. Solucionadores SAT modernos não requerem espaço exponencial de memória.
4. Solucionadores SAT dependem de pouca intervenção manual se comparado a outras técnicas.

Nas subseções que se seguem além de exemplos práticos mostramos um pouco da semântica por trás do método. A semântica nos permitirá demonstrar como BMC para LTL pode ser reduzido para SAT em tempo polinomial.

### 3.3.1 Semântica Limitada

Primeiramente vamos definir o conceito de validade de uma fórmula LTL:

**Definição 5** *Uma fórmula LTL $f$ é* universalmente *válida em uma estrutura* Kripke *$M$ (em símbolos $M \models \boldsymbol{A}\, f$) se e somente se $\pi \models f$ para todos os caminhos $\pi$ em $M$ com $\pi(0) \in I$. Uma fórmula LTL é existencialmente válida em uma estrutura* Kripke *$M$ (em símbolos $M \models \boldsymbol{E}\, f$) se e somente se existe um caminho $\pi$ em $M$ com $\pi \models f$ e $\pi(0) \in I$.*

Determinar se um fórmula LTL é universalmente válida em um estrutura *Kripke* é chamado *problema de verificação de modelos universal*. Analogamente temos o *problema de verificação de modelos existencial*.

É claro que uma fórmula LTL $f$ é universalmente válida em uma estrutura *Kripke* $M$ se e somente se $\neg f$ não é existencialmente válida. Portanto para resolver o *problema de verificação de modelos universal* nós mostramos que o *problema de verificação de modelos existencial* para a fórmula negada não tem solução. Intuitivamente dessa forma estamos procurando por um *contra-exemplo* e se não obtivermos sucesso a fórmula é universalmente válida. Vamos então levar em consideração apenas o *problema de verificação de modelos existencial*.

A *idéia básica* de *bounded model checking* é considerar um prefixo finito de um caminho que pode ser a solução para o *problema de verificação de modelos existencial*. Nós restringimos o tamanho do prefixo por um limite $k$. Na prática nós progressivamente aumentamos o limite em busca de contra-exemplos maiores e maiores, ou até que alcancemos o limite do hardware.

Uma observação crítica é que, embora o prefixo seja finito, ele ainda pode representar um caminho infinito. Para isso é necessário que exista uma transição do último estado do prefixo para um dos estados posteriores. Dessa forma apenas prefixos desse tipo podem ser "testemunhas" para $\mathbf{G}\, f$.

**Definição 6** *Nós chamamos um caminho $\pi$ de laço-k se existe um $l \in \mathbb{N}$ com $l \leq k$ para o qual $\pi(k) \rightarrow \pi(l)$ e $\pi = u.v^{\omega}$ onde $u = (\pi(0), \ldots, \pi(l-1))$, $v = (\pi(l), \ldots, \pi(k))$ e $w$ é .*

A semântica LTL muda quando lidamos com BMC. Basicamente a semântica é dividida em dois casos. No primeiro caso o prefixo do caminho em questão é um *laço-k*. Nessa situação a semântica LTL original é mantida, pois toda a informação sobre o caminho está contida no prefixo.

**Definição 7** *Seja $k \in \mathbb{N}$ e $\pi$ um laço-k. Então a fórmula LTL $f$ é válida no caminho $\pi$ com limite $k$ (em símbolos $\pi \models_k f$) se e somente se $\pi \models f$.*

O segundo caso é quando o prefixo do caminho não é um laço-k. Dessa forma não podemos definir a semântica recursivamente sobre sufixos de $\pi$. Isso implica, por exemplo, que a fórmula $\mathbf{G}f$ é sempre falsa. Na definição da semântica é usado o operador $\models_k^i$. O parâmetro $i$ indica a posição corrente no prefixo.

**Definição 8** *Seja $k \in \mathbb{N}$ e seja $\pi$ um caminho que não é um laço-k. Então uma fórmula LTL é válida sobre $\pi$ com limite $k$ (em símbolos $\pi \models_k f$) se e somente se $\pi \models_k^0 f$ onde:*

$\pi \models_k^i p$ *iff* $p \in l(\pi(i))$ $\quad$ $\pi \models_k^i \neg f$ *iff* $f \notin l(\pi(i))$

*(Onde $p$ é uma proposição atômica.)*

$\pi \models_k^i f \wedge g$ *iff* $\pi \models_k^i f$ *e* $\pi \models_k^i g$

*($f$ e $g$ são válidas em $\pi$.)*

$\pi \models_k^i f \vee g$ *iff* $\pi \models_k^i f$ *ou* $\pi \models_k^i g$

*($f$ ou $g$ são válidas em $\pi$.)*

$\pi \models_k^i \boldsymbol{G}\, f$ *é sempre falso*

*(Pois o caminho não é um laço-k.)*

$\pi \models_k^i \boldsymbol{F}\, f$ *iff* $\exists j, i \leq j \leq k . \pi \models_k^j f$

*($f$ é válida em $\pi$ se somente se existir um estado $\pi(j)$, tal que $i \leq j \leq k$, onde $f$ é válida.)*

$\pi \models_k^i \boldsymbol{X}\, f$ *iff* $i < k$ *e* $\pi \models_k^{i+1} f$

*($f$ é válida em $\pi$ se e somente se $i < k$ e se $f$ for válida no estado $\pi(i+1)$.)*

$\pi \models_k^i f\ \boldsymbol{U}\, g$ *iff* $\exists j, i \leq j \leq k\ [\pi \models_k^j g$ *e* $\forall n, i \leq n < j . \pi \models_k^n f]$

*(A fórmula dada é válida no caminho se existir um estado $\pi(j)$, tal que $i \leq j \leq k$, onde $g$ é verdadeira e para todos os estados do caminho antecessores a $\pi(j)$, $f$ é verdadeira.)*

$\pi \models_k^i f\ \boldsymbol{R}\, g$ *iff* $\exists j, i \leq j \leq k\ [\pi \models_k^j f$ *e* $\forall n, i \leq n \leq j . \pi \models_k^n g]$

*(A fórmula dada é válida no caminho se existir um estado $\pi(j)$, tal que $i \leq j \leq k$, onde $f$ e $g$ são verdadeiras e para todos os estados do caminho antecessores a $\pi(j)$, $g$ é verdadeira.)*

Note que o caso em que $f\ \mathbf{R}\ g$ quando $g$ é sempre verdadeira foi eliminado. Pela mesma razão que culminou na invalidez de $\mathbf{G}f$.

Para concluir a parte semântica da LTL BMC é necessário fazer referência a dois lemas cujas as provas estão em [6].

**Lema 1** *Seja $h$ uma fórmula LTL e $\pi$ um caminho, então $\pi \models_k h \Rightarrow \pi \models h$.*

**Lema 2** *Seja $f$ uma fórmula LTL e $M$ uma estrutura* Kripke. *Se $M \models \boldsymbol{E}f$ então existe um $k \in \mathbb{N}$ com $M \models_k \boldsymbol{E}f$*

A grosso modo estes dois lemas reduzem o problema de verificação modelos existencial ao problema de verificação limitada de modelos existencial (BMC). Ou, que se levarmos em conta todos os valores possíveis para o limite, as semânticas original e limitada são equivalentes.

### 3.3.2 Tradução em SAT

Uma vez definida a semântica de BMC temos que definir como será feita a tradução para uma fórmula proposicional, de maneira que possamos empregar solucionadores SAT para verificação de modelos.

Dados:

- Uma estrutura *Kripke* $M$;
- Uma fórmula de lógica temporal $f$;
- Um limite $k$;

Contruímos a fórmula proposicional $[\![M, f]\!]_k$ que vai ser satisfazível se e somente se a fórmula $f$ é válida em algum caminho de $M$.

Para contruí-la temos antes que definir uma fórmula proposicional $[\![M]\!]_k$, que representa os estados alcançáveis até $k$ transições.

**Definição 9** *Para uma estrutura* Kripke $M$ *e $k \in \mathbb{N}$:*

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

*Onde:*

- *$I(s_0)$ é a função característica do conjunto de estados iniciais;*
- *$T(s_i, s_{i+1})$ é a função característica da relação de transição.*

Para traduzir a fórmula $f$ temos que levar em consideração se o caminho envolvido for um laço-$k$ ou não.

Primeiramente o caso em que o caminho não tem um laço.

**Definição 10** *Para as fórmulas LTL $f$, $g$, proposição atômica $p$ e $k, i \in \mathbb{N}$, com $i \leq k$:*

$$
\begin{aligned}
[\![p]\!]_k^i &:= p(s_i), \text{ onde } p(s_i) \text{ é a representação} \\
&\quad \text{da proposição } p \text{ no estado } s_i. \\
[\![\neg p]\!]_k^i &:= \neg p(s_i) \\
[\![f \wedge g]\!]_k^i &:= [\![f]\!]_k^i \wedge [\![g]\!]_k^i \\
[\![f \vee g]\!]_k^i &:= [\![f]\!]_k^i \vee [\![g]\!]_k^i \\
[\![\boldsymbol{G} f]\!]_k^i &:= falso \\
[\![\boldsymbol{F} f]\!]_k^i &:= \bigvee_{j=i}^{k} [\![f]\!]_k^j \\
[\![\boldsymbol{X} f]\!]_k^i &:= \text{se } i < k \text{ então } [\![f]\!]_k^{i+1} \text{ senão é } falso \\
[\![f \, \boldsymbol{U} g]\!]_k^i &:= \bigvee_{j=i}^{k} \left( [\![g]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![f]\!]_k^n \right) \\
[\![f \, \boldsymbol{R} g]\!]_k^i &:= \bigvee_{j=i}^{k} \left( [\![f]\!]_k^j \wedge \bigwedge_{n=i}^{j} [\![g]\!]_k^n \right)
\end{aligned}
$$

Para o caso em que o caminho é um laço-k a tradução da fórmula LTL depende da posição corrente $i$, do tamanho do prefixo $k$ e da posição onde o laço começa que é denotada por $l$.

Antes temos que definir o operador sucessor (em símbolos *succ*) em um laço.

**Definição 11** *Seja $k, l, i \in \mathbb{N}$ com $i \leq k$, O sucessor $succ(i)$ de $i$ em um laço-k é definido como $succ(i) := i + 1$ para $i < k$ e $succ(i) = l$ para $i = k$.*

Com a ajuda desse operador podemos definir a tradução de uma fórmula LTL para um laço-k.

**Definição 12** *Seja $f$ e $g$ fórmulas LTL, $p$ uma proposição atômica e $k, l, i \in \mathbb{N}$, com $l, i \leq k$:*

$$
\begin{aligned}
{}_l[\![p]\!]^i_k &:= p(s_i) \\
{}_l[\![\neg p]\!]^i_k &:= \neg p(s_i) \\
{}_l[\![f \wedge g]\!]^i_k &:= {}_l[\![f]\!]^i_k \wedge_l [\![g]\!]^i_k \\
{}_l[\![f \vee g]\!]^i_k &:= {}_l[\![f]\!]^i_k \vee_l [\![g]\!]^i_k \\
{}_l[\![\boldsymbol{G}\, f]\!]^i_k &:= \textstyle\bigwedge_{j=min(i,l)}^{k} {}_l[\![f]\!]^j_k \\
{}_l[\![\boldsymbol{F}\, f]\!]^i_k &:= \textstyle\bigvee_{j=min(i,l)}^{k} {}_l[\![f]\!]^j_k \\
{}_l[\![\boldsymbol{X}\, f]\!]^i_k &:= {}_l[\![f]\!]^{succ(i)}_k \\
{}_l[\![f\ \boldsymbol{U}\, g]\!]^i_k &:= \textstyle\bigvee_{j=i}^{k} \left( {}_l[\![g]\!]^j_k \wedge \bigwedge_{n=i}^{j-1} {}_l[\![f]\!]^n_k \right) \vee \\
&\quad \textstyle\bigvee_{j=l}^{i-1} \left( {}_l[\![g]\!]^j_k \wedge \bigwedge_{n=i}^{k} {}_l[\![f]\!]^n_k \wedge \bigwedge_{n=l}^{j-1} {}_l[\![f]\!]^n_k \right) \\
{}_l[\![f\ \boldsymbol{R}\, g]\!]^i_k &:= \textstyle\bigwedge_{j=min(i,l)}^{k} {}_l[\![g]\!]^j_k \vee \\
&\quad \textstyle\bigvee_{j=i}^{k} \left( {}_l[\![f]\!]^j_k \wedge \bigwedge_{n=i}^{j} {}_l[\![g]\!]^n_k \right) \vee \\
&\quad \textstyle\bigvee_{j=l}^{i-1} \left( {}_l[\![f]\!]^j_k \wedge \bigwedge_{n=i}^{k} {}_l[\![g]\!]^n_k \wedge \bigwedge_{n=l}^{j} {}_l[\![g]\!]^n_k \right)
\end{aligned}
$$

Baseando-se nos dois operadores dados podemos chegar a fórmula da tradução geral $[\![M, f]\!]_k$. Mas para isso temos que definir uma condição para diferenciar um laço-k de um caminho comum.

**Definição 13** *Para $k, l \in \mathbb{N}$, seja ${}_lL_k := T(s_k, s_l), L_k := \bigvee_{l=0}^{k} {}_lL_k$*

Podemos então definir a fórmula de tradução geral:

**Definição 14** *Seja $f$ uma fórmula LTL, $M$ uma estrutura* Kripke *e $k \in \mathbb{N}$:*

$$
[\![M, f]\!]_k := [\![M]\!]_k \wedge \left( \left( \neg L_k \wedge [\![f]\!]^0_k \right) \vee \bigvee_{l=0}^{k} \left( {}_lL_k \ \wedge\ {}_l[\![f]\!]^0_k \right) \right)
$$

O lado esquerdo da disjunção é o caso onde não existe o laço e a tradução sem o laço é usada. No lado direito todos os possíveis inícios do laço são

tentados e a tradução com o laço é usada em conjunção com a condição de laço.

O entendimento dessa fórmula é fundamental para o entendimento do algoritmo distribuído descrito no capítulo 5.

# Capítulo 4

# O Problema e a Abordagem

## 4.1 Introdução

Desde a introdução de BMC vários grupos independentes publicaram resultados experimentais dessa técnica. Na maioria das vezes comparando-a a BDDs [7, 57, 58, 59]. Todos esses experimentos chegam sempre a mesma conclusão: BMC baseado em solucionadores SAT normalmente encontra erros mais rapidamente que técnicas baseadas em BDDs. Mas BMC tem limitações. Nos experimentos mostrou-se que tomando como base solucionadores SAT no estado-da-arte e modelos de hardware típicos, a técnica dificilmente alcançava erros além de 80 ciclos em um tempo razoável de execução.

Buscamos neste trabalho aumentar a profundidade máxima que o algoritmo de BMC consegue atingir para um dado problema. Mas como fazer isso? Pfister destacou em [60] que existem três maneiras de se melhorar a performance de alguma coisa:

1. Trabalhar mais,
2. Trabalhar melhor, e
3. Conseguir ajuda.

Fazendo uma analogia a tecnologias de computação, trabalhar mais seria usar um hardware melhor, trabalhar melhor seria utilizar melhores algoritmos e conseguir ajuda seria usar mais de um computador para resolver a mesma tarefa.

Usar melhores de componentes de hardware além de ser uma medida que não envolve nenhum esforço adicional exige que um hardware superior esteja disponível. Podemos então melhorar os algoritmos de BMC ou usar mais

de um computador para resolver o problema. O uso de algoritmos distribuídos em Verificação de Modelos é uma área muito menos explorada que a otimização dos algoritmos seqüenciais. Portanto, neste trabalhos optamos por explorar o domínio da *computação distribuída* para melhorar a técnica de BMC baseada em solucionadores SAT.

## 4.2 Explosão de Estados

O principal desafio para a verificação de modelos é o problema de explosão de estados que ocorre normalmente quando o sistema a ser verificado tem muitos componentes que fazem transições em paralelo. Por exemplo, um sistema simples composto de quatro máquinas de estados ou processos concorrentes, cada um com três variáveis com cinco valores possíveis cada, resulta em um espaço de estados total de aproximadamente 250 milhões de estados.

As primeiras técnicas baseadas em BDDs conseguiram verificar modelos com até $10^{20}$ estados. Posteriormente pesquisadores aprimoram essas técnicas e conseguiram verificar sistemas com até $10^{120}$ estados. BMC com sua abordagem baseada em procedimentos SAT também conseguiu avanços no tamanho do modelo verificado. No decorrer dos anos é esperado que os novos limites sejam batidos.

No entanto, é certo que no decorrer dos anos, com o avanço da computação, sistemas cada vez mais complexos sejam desenvolvidos, aumentando o número de estados que devam ser verificados. Fazendo com que o problema de explosão de estados esteja sempre presente.

Nossa aposta é que a utilização de computação distribuída aliada ao avanço das técnicas de verificação possam permitir um avanço significativo no tamanho dos modelos verificados.

## 4.3 Computação Paralela ou Distribuída?

Informalmente podemos dizer que a maior diferença entre a computação distribuída e a computação paralela é que na primeira a computação é feita em vários computadores, enquanto que na segunda a computação é feita em vários processadores.

Computação Distribuída é o processo de executar uma tarefa computacional simples em mais de um computador distinto. Computação Paralela é a execução simultânea da mesma tarefa em múltiplos processadores de maneira que possamos obter resultados mais rapidamente.

Note que por essas definições toda computação distribuída é um tipo de computação paralela, mas a recíproca não é verdadeira.

No passado, a computação paralela era dominante. Soluções caras como sistemas de Processamento Paralelo Massivo (*Massive Parallel Processing - MPP*) e Processadores Vetorias eram desenvolvidas. A partir dos anos 90 a necessidade desses supercomputadores vem diminuindo e conseqüentemente o paradigma da computação paralela tem efetivamente mudado para o paradigma distribuído.

Esse declínio não é devido a uma queda na necessidade de processamento paralelo mas sim no aumento na procura por soluções mais baratas, escaláveis e que possam cuidar de uma variação maior de problemas paralelos. Uma das forças por trás dessa mudança foi o surgimento de componentes de hardware de alta performance a preços de *commodities*. Outro fator importante para o avanço da computação distribuída é a padronização de muitas ferramentas e utilitários como, por exemplo, a biblioteca de passagem de mensagens MPI [61] e o ambiente de programação PVM [62].

Essa mudança de paradigma justifica a opção pela computação distribuída frente a computação paralela para o desenvolvimento desse trabalho.

## 4.4 Clusters

As duas arquiteturas para computação distribuída mais conhecidas são a computação em *cluster* [63, 60, 64] e a computação em grade [65, 66]. A computação em cluster é normalmente realizada em redes locais (*Local Area Network* - LAN). A computação em grade é realizada em redes geograficamente distribuídas (*Wide Area Network* - WAN) que envolvem áreas maiores como um campus universitário, uma cidade, uma país ou até mesmo o mundo. A Internet é um WAN.

Clusters são uma coleção de computadores poderosos compostos por commodities que são interconectados por uma rede local. Grades de computadores permitem o compartilhamentos e a agregação de recursos computacionais (como supercomputadores, sistemas de armazenamento, fontes de dados e até mesmo clusters) distribuídos geograficamente. Essas duas arquiteturas são usadas como plataformas para computação distribuída com um alto custo-benefício.

Podemos citar como exemplos de aplicações industriais e científicas que usam essas plataformas a simulação de sistemas, a bioinformática, a previsão do tempo, a modelagem de automóveis, a engenharia estrutural e o estudo de abalos sísmicos.

Na área comercial podemos citar vários exemplos que usufruem dessas

arquiteturas: servidores *Web* (e.g. *hotmail.com*, *yahoo.com*), máquinas de busca (e.g. o *google.com* usa um cluster com mais de 4500 nodos), SGBDs (e.g. *Oracle*), modelagem financeira, redes par-a-par (e.g. *Napster*, *FastTrack*) e distribuição de conteúdo (e.g. a *Akamai* tem uma rede mundial de clusters para distribuir o conteúdo de grandes sítios como o *cnn.com*).

Mas dessas arquiteturas qual é a mais adequada para um Verificador de Modelos Distribuído? Grades disponibilizam um poder de computação muito maior que um único cluster, mas isso a um custo de novos desafios como autenticação, autorização, baixa conectividade, heterogenidade e segurança. Clusters são mais fáceis de se trabalhar pois existem em um mesmo espaço físico e estão sob uma mesma administração. Da mesma forma clusters são mais freqüentes de se encontrar do que grades. Uma das diretrizes desse trabalho é utilizar os componentes de hardware e software mais comuns tanto no meio acadêmico quanto no meio industrial. Nesse contexto um cluster é uma opção natural. A utilização de computação em grade é considerada um objetivo futuro.

Um arquitetura típica de um cluster pode ser vista na figura 4.1.

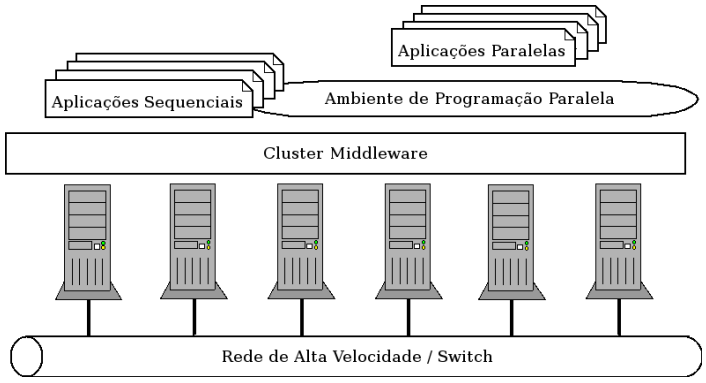


Figura 4.1: Arquitetura Típica de um Cluster

Os componentes dessa arquitetura típica são:

- Nodos do Cluster: compostos por PCs, Estações de Trabalho ou Computadores Multiprocessados. Todos com sistema operacional e recursos de E/S.

- Redes de Alta Velocidade como a Gigabit Ethernet e a Myrinet. Muitas vezes a FastEthernet é suficiente.

- Cluster Middleware: componente responsável por fazer os nodos do cluster trabalharem coletivamente. Pode ser implementado via hard-

ware especial ou software. No caso do software pode ser uma modificação no kernel.

- Ambientes de Programação Paralela: os exemplos clássicos desse componente são a MPI e o PVM.

Clusters apresentam uma série de vantagens se comparados com computadores paralelos. Entre as quais se destacam:

- Estações de Trabalho individuais estão se tornando incrivelmente poderosas.
- Novas tecnologias e protocolos de comunicação para LANs vêm surgindo aumentando a largura de banda e diminuindo a latência.
- Clusters são mais fáceis de integrar em redes existentes do que computadores paralelos.
- A utilização da capacidade de processamento é baixa na maioria das estações de trabalho. O processador passa muito tempo ocioso.
- As ferramentas de desenvolvimento para estações de trabalho são mais maduras se comparadas com as soluções, muitas vezes proprietárias, para computadores paralelos. Isso se deve principalmente a não padronização de muitos sistemas paralelos.
- Estações de trabalho são baratas se comparadas as computadores paralelos.
- Clusters podem crescer facilmente.
- A capacidade de um nodo do cluster pode ser aumentada facilmente.
- Estações de Trabalho são encontradas em quase todas as empresas e centros acadêmicos.

Clusters são classificados de muitas formas. Entre elas destacam-se a aplicação alvo, o tipo de hardware, a disponibilidade do hardware, o sistema operacional e a configuração dos nodos. Detalhes sobre essas classificações podem ser encontrados em [63].

De acordo com nosso objetivo descrito em 1.3 o cluster que procuramos deve ter a seguinte classificação:

1. Aplicação Alvo: nossa aplicação alvo é um Verificador de Modelos Distribuído. Dessa forma o cluster que procuramos deve oferecer desempenho. Um Cluster desse tipo é chamado de Cluster de Alta Performance (*High Performance Cluster* - HPC).

2. Disponibilidade do Hardware: cluster que tem nodos dedicados naturalmente oferecem um melhor desempenho. No entanto clusters que possuem nodos não-dedicados são muito mais comuns já que podem ser formados a partir do hardware já existente em uma empresa ou universidade sem a perda de pontos de trabalho. Por isso projetamos nosso verificador distribuído levando em consideração nodos não-dedicados.

3. Sistema Operacional: nosso trabalho vai ser todo desenvolvido sobre um sistema operacional no padrão GNU/Linux [67]. Sistemas deste tipo oferecem compatibilidade com uma série de equipamentos de hardware e uma enorme quantidade de componentes de software. E como se isso já não fosse suficiente é possível encontrar muitas versões gratuitas do Linux.

4. Tipo de Hardware: o trabalho foi desenvolvido na expectativa de se encontrar computadores monoprocessados baseados na arquitetura x86. Esse tipo de hardware é o mais comum em empresas e centros acadêmicos [68].

5. Configuração dos Nodos: o trabalho foi desenvolvido para funcionar em um cluster homogêneo, onde todos os nodos têm a arquitetura e o mesmo sistema operacional.

No capítulo 6 definimos com precisão como configuramos um cluster para a utilização em nossos experimentos.

# Capítulo 5

# O Projeto

## 5.1 Introdução

Nesse capítulo vamos descrever o desenvolvimento do algoritmo para verificação de modelos distribuído baseado em *Bounded Model Checking* com solucionadores SAT.

O esqueleto do algoritmo é baseado em uma importante observação feita em [69]. Nesse artigo os autores descrevem que um problema BMC gerado para partir do desenrolar de um modelo em diferentes quadros de tempo, provê um particionamento natural onde as partições são organizadas em uma topologia linear. A figura 5.1 ilustra esse particionamento.

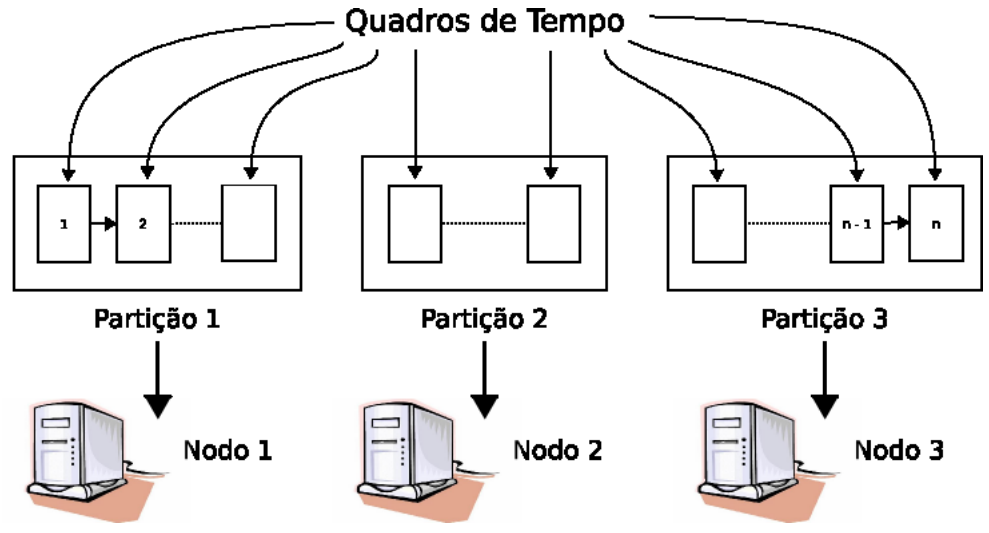


Figura 5.1: Um Possível Particionamento por Quadros de Tempo

Uma vez particionado o problema poderíamos explorar cada partição in-

dependentemente em busca de uma solução local ou parcial. Depois teríamos que sincronizar os resultados até que encontrassemos uma solução para todas as partições ou a indicação de insatisfiabilidade.

Esta abordagem no contexto de BMC não é interessante. Vamos supor, por exemplo, que estamos explorando a terceira partição sem tomar como partida nenhum valor obtido na partição anterior. Corremos o risco nesse caso de atribuir valores a variáveis que só seriam possíveis em estados inalcançáveis. Esse problema foi detectado pela primeira vez em [57]. O autor desse artigo sugere que para solucionar esse problema o solucionador SAT deva escolher suas variáveis de decisão baseando-se em um grafo de dependência de variáveis (ou até mesmo por uma ordenação estática das variáveis). Essa sugestão visava um algoritmo seqüencial. No nosso algoritmo implementamos essa mesma idéia de uma forma diferente. Ao invés de adotarmos uma ordem para escolha das variáveis de decisão em cada partição, exploramos as partições em uma ordem seqüencial.

O funcionamento é simples. Iniciamos explorando a primeira partição, se obtivermos uma atribuição satisfazível para essa partição passamos a explorar a segunda partição a partir dos valores gerados na primeira, se encontrarmos uma atribuição para essa partição repetimos o mesmo procedimento para a terceira e assim por diante até encontrarmos o limite $k$ estipulado previamente.

Porém, se o algoritmo funcionar dessa forma não haveria paralelismo algum já que as partições nunca seriam exploradas ao mesmo tempo. Para podermos explorar as partições de maneira concorrente a exploração da primeira partição deverá ser contínua. Por exemplo: um primeiro processo explora a primeira partição e encontra uma solução parcial. Um segundo processo passa a explorar a segunda partição a partir desta solução parcial. Enquanto isso o primeiro processo procura outra solução para a primeira partição e caso tenha sucesso um terceiro processo passará a explorar a essa outra solução parcial. Se usássemos um processo para explorar cada partição chegaríamos a uma solução similar a descrita na figura 5.2.

No entanto, a forma como particionamos o problema nos permite usar um mesmo conjunto de cláusulas para explorar várias partições. Mais precisamente podemos usar um mesmo processo para explorar da segunda partição até a partição $k$. A figura 5.3 mostra um esboço do diagrama de funcionamento do algoritmo. Maiores detalhes sobre as características que nos permitem explorar de tal forma estão descritos na próximas seções.

As vantagens de se fazer a exploração da forma como nosso algoritmo propõe são claras. Primeiramente diminuimos o uso da rede já que usamos um mesmo processo para explorar da segunda partição até a partição $k$. Em segundo lugar, ao usarmos o mesmo conjunto de cláusulas para várias

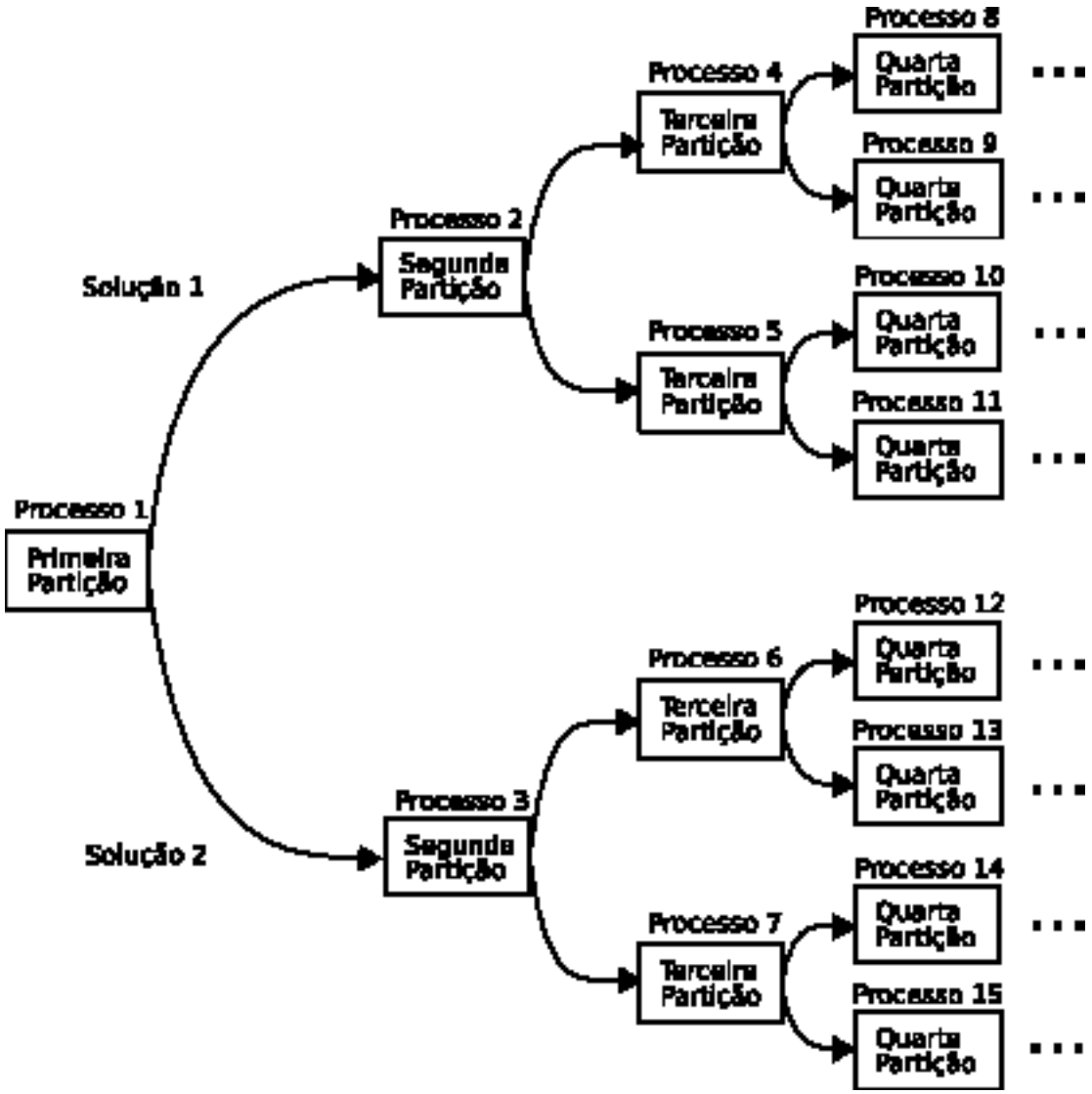


Figura 5.2: Exploração Seqüencial das Partições

partições garantimos uma menor utilização de memória para representar o problema. E finalmente, podemos explorar o poder computacional de mais de um computador para resolver o problema.

O restante do capítulo se divide da seguinte forma. A segunda seção descreve como será feito o particionamento do problema. A terceira seção descreve como funciona o algoritmo sequencial que pode explorar esse particionamento. Em seguida descrevemos o verificador como um todo. Na última seção descrevemos os trabalhos relacionados.

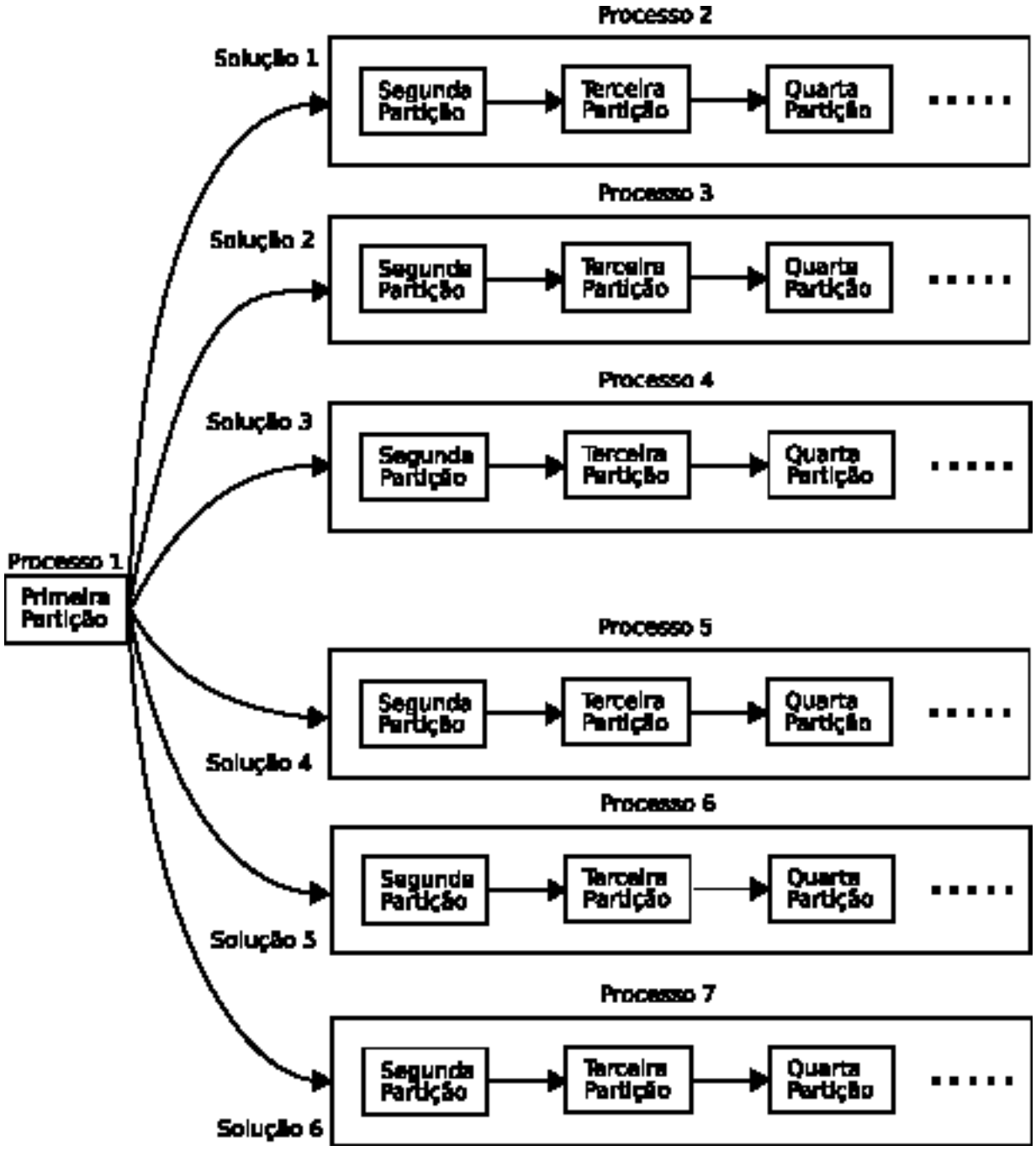


Figura 5.3: Exploração Seqüencial das Partições explorando características especiais de cláusulas geradas a partir de BMC.

## 5.2 Particionamento

Nesta seção descrevemos como optamos por particionar as claúsulas do problema SAT gerado a partir de BMC.

**Definição 15** *Uma* partição *do problema de BMC com SAT é um subconjunto do conjunto total de cláusulas CNF que formam o problema completo.*

O particionamento proposto divide as claúsulas buscando:

- Minimizar a quantidade de variáveis compartilhadas entre as partições. A razão para isso é a simples observação de que menos variáveis compartilhadas implica em uma menor quantidade de comunicação entre os processos envolvidos na solução do problema.
- Separar as partes simétricas do modelo a as partes assimétricas em partições diferentes. Esta separação nos permitirá explorar propriedades que as partes simétricas (que envolvem basicamente as transições no modelo) oferecem.

Nas próximas duas subseções vamos explicar como é feito o particionamento e quais são as principais propriedades apresentadas pelas partes simétricas.

### Particionamento das Claúsulas

Para explicar como funciona este particionamento temos que voltar ao capítulo 3, onde mostramos como uma fórmula proposicional é gerada a partir de um problema de BMC. Primeiramente vamos lembrar como traduzimos uma estrutura *Kripke* em uma fórmula proposicional:

$$[\![M]\!]_k := I(s_0) \wedge \left( \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \tag{5.1}$$

Onde:

$I(s_0)$ é o conjunto de estados iniciais.

$T(s_i, s_{i+1})$ é a transição entre os estados $s_i$ e $s_{i+1}$.

$k$ é o limite máximo para a busca.

Como já foi dito na seção anterior, [69] descreve que um problema BMC gerado para partir do desenrolar dessa fórmula em diferentes quadros de tempo, provê um particionamento natural. Como a fórmula 5.1 não é nada mais do que uma série de conjunções podemos observar isso claramente.

Por exemplo: $I(s_0)$ corresponderia a primeira partição, $T(s_0, s_1)$ a segunda, $T(s_1, s_2)$ a terceira e assim por diante. Observe que as partições só compartilham variáveis dos estados onde são feitos os cortes para o particionamento. No exemplo um dos cortes foi feito sobre estado $s_1$ e as variáveis desse estado são compartilhadas entre a segunda e a terceira partição.

Sabemos também que a fórmula proposicional de BMC só fica completa após incluirmos as propriedades a serem verificadas. Descrevemos a tradução completa como:

$$[\![M, P]\!]_k := I(s_0) \wedge \left( \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge [\![P]\!]_k^0 \tag{5.2}$$

Onde:

$[\![P]\!]_k^0$ é a tradução da propriedade a ser verificada. (Detalhes no capítulo 3).

Fazendo uma pequena manipulação em $[\![M, P]\!]_k$ temos:

$$[\![M, P]\!]_k := I(s_0) \wedge [\![P]\!]_k^0 \wedge \left( \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \tag{5.3}$$

Vamos expandir 5.3 para um $k = 6$ para exemplificar:

$$\begin{array}{lcl}
[\![M, P]\!]_6 & := & I(s_0) \wedge [\![P]\!]_6^0 \wedge \left( \bigwedge_{i=0}^{6-1} T(s_i, s_{i+1}) \right) \\
[\![M, P]\!]_6 & := & I(s_0) \wedge [\![P]\!]_6^0 \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3) \wedge T(s_3, s_4) \wedge \\
 & & T(s_4, s_5) \wedge T(s_5, s_6) \\
[\![M, P]\!]_6 & := & \underbrace{I(s_0) \wedge [\![P]\!]_6^0}_{P1} \wedge \underbrace{T(s_0, s_1)}_{P2} \wedge \underbrace{T(s_1, s_2)}_{P3} \wedge \underbrace{T(s_2, s_3)}_{P4} \wedge \underbrace{T(s_3, s_4)}_{P5} \\
 & & \wedge \underbrace{T(s_4, s_5)}_{P6} \wedge \underbrace{T(s_5, s_6)}_{P7}
\end{array}$$

Note como a fórmula foi dividida em sete partições $P1$, $P2$, $P3$, $P4$, $P5$, $P6$ e $P7$. A primeira partição será composta de $I(s_0) \wedge [\![P]\!]_k^0$ e nos referimos a ela como a *partição primária*. As partições restantes são chamadas de *partições secundárias*.

Esse particionamento é interessante pois nos permite explorar a simetria das partições secundárias (no exemplo $P2$, $P3$, $P4$, $P5$, $P6$ e $P7$). A simetria é clara, basta observarmos que a única diferença da fórmula que representa $T(s_1, s_2)$ da fórmula de $T(s_2, s_3)$ são os índices.

Exemplificando:

$$
\begin{aligned}
T(s_i, s_{i+1}): &\quad (a_{i+1} \leftrightarrow \neg a_i) \wedge (b_{i+1} \leftrightarrow a_i \bigoplus b_i) \\
T(s_1, s_2): &\quad (a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow a_1 \bigoplus b_1) \\
T(s_2, s_3): &\quad (a_3 \leftrightarrow \neg a_2) \wedge (b_3 \leftrightarrow a_2 \bigoplus b_2)
\end{aligned}
$$

**Simetria das Partições Secundárias**

A exploração da simetria das partições secundárias nos trazem vantagens significativas:

1. Nos permite usar um mesmo conjunto de cláusulas para explorar várias partições, já que a única diferença entre as fórmulas são os índices das variáveis.

2. Nos permite compartilhar e replicar cláusulas conflito entre as partições.

A idéias de compartilhamento e replicação de cláusulas conflito foram apresentadas pela primeira vez em [57]. O autor desse trabalho no entanto não sugeriu um particionamento do problema que permitisse a geração direta dessas cláusulas, ele por outro lado tentava contornar a parte não assimétrica do problema.

O mecanismo de compartilhamento é baseado na seguinte idéia. Uma cláusula conflito gerada a partir da fórmula de $T(s_1, s_2)$ pode ser usada na busca pela solução da fórmula $T(s_2, s_3)$ com uma simples mudança de índices. Ou seja, uma vez que uma partição secundária obtém uma cláusula conflito esta pode ser compartilhada com todas as outras partições secundárias.

O mecanismo de replicação expande a idéia do compartilhamento. A idéia pode ser ilustrada por um exemplo. Suponha que uma partição secundária encontrou a cláusula conflito $\pi = (\neg x_4 \vee y_7 \vee z_5)$. Podemos afirmar que, dada a simetria da partição, a cláusula $\pi = (\neg x_3 \vee y_6 \vee z_4)$ também vai ser uma cláusula conflito, chamamos essa cláusula conflito de cláusula replicada. Podemos replicar a cláusula conflito várias vezes desde que o índice de cada variável sempre seja maior que 0 ou menor ou igual a $k$.

É importantíssimo notar que todas as cláusulas conflito geradas, inclusive as replicadas, podem ser incluídas no problema representado pela primeira partição. Isso permite uma poda da árvore de busca SAT muito mais eficiente.

## 5.3 Algoritmo Sequencial

Nesta seção vamos explicar como se deve implementar um algoritmo sequencial que explorasse o particionamento descrito na seção 5.2. A intenção

dessa explicação é um entedimento mais claro de algumas otimizações implementadas. Retiramos então nesse caso tudo o que se refere a paralelismo.

Vamos começar por um diagrama que resume o funcionamento do algoritmo sequencial. A figura 5.4 contém esse diagrama.

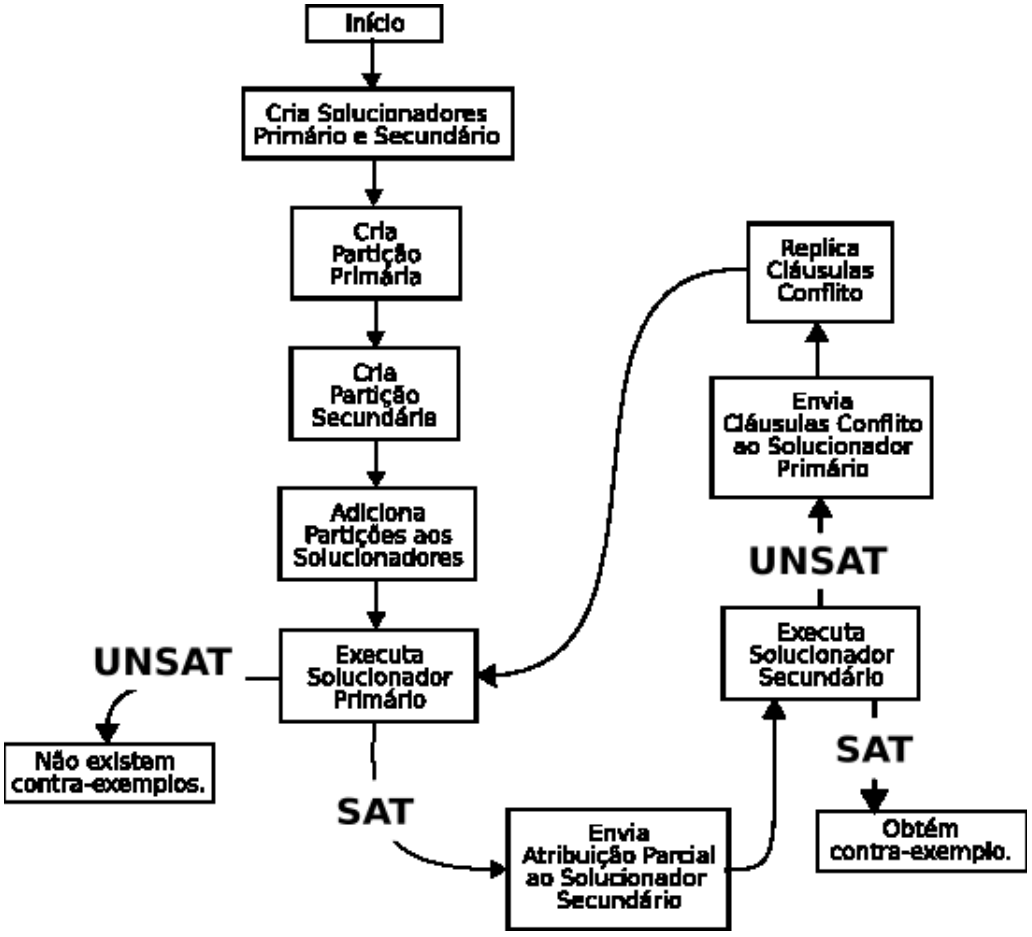


Figura 5.4: Fluxograma do Algoritmo Sequencial

O diagrama será todo explicado com o auxílio de trechos de código fonte. O código fonte dos exemplos é escrito na linguagem C e contém chamadas de funções reais do verificador NuSMV [70]. Este foi o verificador usado para a implementação do algoritmo distribuído e vai facilitar o entedimento posterior deste.

Voltando ao diagrama temos como primeiro passo a criação dos solucionadores. Esse passo é simples e se resume na criação de solucionadores SAT sem cláusulas. Pelo código e pelo diagrama teremos um solucionador primário e um solucionador secundário, que exploram a partição primária e as

```
/* sat solvers */
SatZchaff_ptr primary_zchaff, secondary_zchaff;

/* Primary Solver Construction: */
primary_zchaff =
  SAT_ZCHAFF(Sat_CreateIncSolver(get_sat_solver(options)));

/* Secondary Solver Construction: */
secondary_zchaff =
  SAT_ZCHAFF(Sat_CreateIncSolver(get_sat_solver(options)));
```

Código 1: Trecho de Código para Criação de Solucionadores

partições secundárias respectivamente. A Listagem de Código 1 ilustra esse passo.

O único detalhe a ser destacado nesse código é o uso da função Sat_CreateIncSolver que cria um solucionador incremental. Um solucionador desse tipo permite a inclusão de cláusulas a um grupo anterior de cláusulas sem que tenhamos que recriar o solucionador. O mesmo vale para a remoção de cláusulas. Esta propriedade tem que ser oferecida pelo solucionador SAT utilizado.

O próximo passo envolve a criação da partição primária. Neste trecho de código já incluímos também a adição da partição primária ao solucionador primário. A Listagem de Código 2 ilustra esse passo.

Nesse código exibido podemos notar que a montagem da partição primária é feita em duas etapas. Na primeira etapa buscamos os estados iniciais e adicionamos ao solucionador primário. Na segunda etapa, que é dependente de cada nível de profundidade $k$, adicionamos as cláusulas que representam as fórmulas ao solucionador primário. Note que antes de passarmos para o próximo nível $k$ essas cláusulas são removidas.

Seguindo o diagrama chegamos a criação da partição secundária. Neste trecho de código já incluímos também a adição da partição secundária ao solucionador secundário. A Listagem de Código 3 ilustra esse passo.

A partição secundária é mais simples de criar que a partição primária. Ela só recebe as cláusulas necessárias para representar uma transição. O conteúdo dessa partição não depende do nível $k$ que estamos explorando.

A seguir no diagrama temos os passos que contemplam a exploração das partições.

Inicialmente o solucionador primário é executado. Se não obtivermos uma solução podemos concluir que o problema todo é insatisfazível. Caso contrário, se obtivermos uma solução passamos esta para o solucionador secundário

```
/* Primary Partition */
be_ptr primary_beProb;

/* Initiating Primary Partition Problem */
primary_beProb = Bmc_Model_GetInit0(be_fsm);
bmc_add_be_into_solver_positively(
             SAT_SOLVER(primary_zchaff),
             SatSolver_get_permanent_group(SAT_SOLVER(primary_zchaff)),
             primary_beProb,
             vars_mgr);

...

/* Start problems generations: */
for (increasingK = k_min; (increasingK <= k_max && !found_solution);
     ++increasingK) {
   SatSolverGroup additionalGroup = SatIncSolver_create_group(
                                          SAT_INC_SOLVER(primary_zchaff));
   be_ptr properties_beProb;   /* Primary Partition problem in BE format */

   ...

   /* Adding Properties Formulae to Primary Partition Problem */
   /* Add LTL tableau to an additional group of the secondary solver */
   properties_beProb = Bmc_Tableau_GetLtlTableau(be_fsm, bltlspec,
                                          increasingK, l);
   bmc_add_be_into_solver_positively(
                  SAT_SOLVER(primary_zchaff),
                  additionalGroup,
                  properties_beProb,
                  vars_mgr);

   ...

   SatIncSolver_destroy_group(SAT_INC_SOLVER(primary_zchaff), additionalGroup);

   ...
}
```

Código 2: Trecho de Código para Criação da Partição Primária

```
/* Secondary Partition */
be_ptr secondary_beProb;

/* Initiating Secondary Partition Problem */
/* Unroll the transition relation to the fixed frame 1 */
secondary_beProb = Bmc_Model_GetUnrolling(be_fsm,0,1);
bmc_add_be_into_solver_positively(
             SAT_SOLVER(secondary_zchaff),
             SatSolver_get_permanent_group(SAT_SOLVER(secondary_zchaff)),
             secondary_beProb,
             vars_mgr);
```

Código 3: Trecho de Código para Criação da Partição Secundaria

na forma de uma atribuição parcial de variáveis.

**Definição 16** *Uma* Atribuição Parcial *é uma atribuição a um subconjunto de variáveis do conjunto total de variáveis do modelo.*

O solucionador secundário é então executado usando os valores da atribuição parcial como heurística de decisão. Em outras palavras, os literais cujos valores são determinados pela atribuição parcial formam o conjunto inicial de decisões do solucionador secundário, sobrepondo a heurística de decisão que seria usada normalmente. Este tipo de recurso deve ser oferecido pelo solucionador SAT, caso contrário deverá ser implementado. O solucionador secundário pode então retornar uma solução.

O código que realiza essa exploração é apresentado na Listagem de Código 4.

Nesse trecho de código as atribuições parciais não são passadas explicitamente para o solucionador secundário, o que é passado é o próprio solucionador primário cuja solução forma a atribuição parcial.

A maneira como fazemos a chamada dos solucionadores primário e secundário será explicada nas próximas subseções.

Para finalizar o diagrama temos o caso em que o solucionador secundário encontra uma solução. Neste caso esta solução é o contra-exemplo para o problema.

### 5.3.1 Execução do Solucionador Primário

O objetivo do solucionador primário é encontrar uma solução para a partição primária.

```
while (1) {

  primary_solver_result = BmcSolver_SolvePrimaryProblem(vars_mgr,
                                                        primary_zchaff,k);

  if (primary_solver_result == 1) {
    secondary_solver_result = BmcSolver_SolveSecondaryProblem(
                                    vars_mgr,primary_zchaff,
                                    secondary_zchaff,increasingK);

    if (secondary_solver_result != NULL) break;
  }
  else {
    /* No solution for primary partition means no solution for the problem. */
    ...
    break;
  }
}
```

Código 4: Trecho de Código para Exploração das Partições

```
SatSolverResult satResult;

/* Solve Primary Partition Problem */
satResult = bmc_sat_zchaff_solve_all_groups(SAT_SOLVER(primary_zchaff),
                                            NULL,0);
```

Código 5: Execução do Solucionador Primário

A execução do solucionador primário se resume a nada mais do que uma chamada ao solucionador SAT. A Listagem de Código 5 exemplifica esse processo.

### 5.3.2 Execução do Solucionador Secundário

O objetivo do solucionador secundário é encontrar uma solução para a partição secundária à partir das atribuições de variáveis definidas na atribuição parcial.

A execução do solucionador secundário exige uma codificação mais elaborada. A questão neste caso é que as cláusulas do solucionador secundário representam apenas a primeira transição ($T(s_0, s_1)$). Dessa forma temos que fazer ajustes de índices de variáveis para usarmos esse mesmo conjunto de

```
lsList start_assignment = lsCreate();

start_assignment = Be_CnfModelToBeModel(
                          BmcVarsMgr_GetBeMgr(vars_mgr),
                          SatSolver_get_model(SAT_SOLVER(primary_zchaff)));
```

Código 6: Obtenção da Atribuição Parcial

cláusulas para resolver problemas que envolvem transições diferentes da primeira. Nesta subseção vamos explicar esse processo passo a passo.

O primeiro passo do solucionador secundário é obter a atribuição parcial. Na nossa codificação a atribuição parcial é a solução do solucionador primário e deve ser obtida diretamente dele. A Listagem de Código 6 mostra como isso pode ser feito.

Se nível $k$ de exploração for maior do que 1 a atribuição parcial vai conter variáveis que não são representadas na primeira transição. Como só temos $T(s_0, s_1)$ representada nas cláusulas vamos ter que solucionar transição por transição levando em conta as variáveis compartilhadas entre as transições.

Por exemplo: iniciamos solucionando a transição $T(s_0, s_1)$, para isso obtemos da atribuição parcial os valores das variáveis que fazem parte dessa transição e usamos como decisões iniciais do solucionador SAT. Se encontrarmos uma solução passamos a próxima partição, $T(s_1, s_2)$. Nesta transição além de buscarmos as variáveis da atribuição parcial temos que buscar as variáveis atribuídas pela transição anterior. Repete-se o processo. Se para uma das transições não encontrarmos uma solução devemos voltar a transição anterior. Se essa transição for $T(s_0, s_1)$ o problema é insatisfazível. Se encontrarmos uma solução para a transição $T(s_{k-1}, s_k)$ o problema é satisfazível.

Para impedir que ao solucionarmos uma transição pela segunda vez encontremos uma mesma solução usamos um recurso chamado cláusulas induzidas. Uma cláusula induzida é uma cláusula conflito gerada à partir da solução encontrada. Adicionamos essa cláusulas ao solucionador SAT quando formos solucionar novamente a transição em questão.

O pseudo-código mostrado na Listagem de Código 7 mostra como podemos solucionar várias transições usando o mesmo conjunto de cláusulas. Não vamos mostrar a códificação na linguagem "C"para esse caso pois ela é muito extensa.

Nesse trecho de código destacamos que as cláusulas conflito que efetivamente podam a busca tanto para solucionador primário quanto para o solucionador secundário são geradas por esse código. Isso pode ser observado

```
nivel_da_transicao = 1;

while (1) {

  1. Adiciona cláusulas induzidas ao solucionador secundário
     caso elas existam.

  2. Cria atribuição local
     - Obtém variáveis compartilhadas pertencentes a atribuição parcial.
     - Obtém variáveis compartilhadas pertencentes a solução da partição
       anterior caso não seja a primeira partição.
     - Ajusta índices das variáveis para que elas reflitam seus pares na
       primeira partição.

  3. Executa Solucionador SAT com atribuição local.

  4. Se o solucionador retornou SAT
     - Obtém solução.
     - Ajusta índices da solução para índices compatíveis com o nível da
       transição explorada.
     - Se o nível da transição igual a "k" então foi encontrada uma solução
       para o problema de BMC. Termina "while".
     - Senão:
       - Gera cláusula induzida para que essa solução não possa ser
         encontrada novamente.
       - Passa-se a transição seguinte.

  5. Se o solucionador retornou UNSAT
     - Obtém causa do conflito.
     - Se não foi acrescentada nenhuma cláusula induzida a essa transição:
       - Adiciona cláusula conflito ao solucionador secundário.
       - Replica cláusula conflito até o nível "k" e adiciona cláusulas
         replicadas ao solucionador primário.
     - Se o nível da transição é igual a "1" então chegamos a UNSAT.
       Termina "while".
     - Senão voltamos ao nível de transição anterior.

  6. Remove cláusulas induzidas adicionadas ao solucionador secundário.
}
```

Código 7: Pseudo-código do Solucionador Secundário

no quinto passo.

Outro ponto de destaque é que as cláusulas conflito são geradas à partir de um mecanismo simples implementado diretamente no solucionador SAT. As cláusulas conflito obtidas no quinto passo são compostas apenas das variáveis das atribuições locais. Assim podemos saber quais variáveis da nossa atribuição parcial geraram o conflito.

## 5.4 O Verificador de Modelos Distribuído

Nesta seção vamos descrever cada componente do verificador de modelos distribuído.

A figura 5.5 descreve a organização dos componentes do verificador. Note que alguns componentes estão circulados por uma caixa com borda tracejada. A caixa tracejada indica que seu conteúdo está sendo executado em um processo independente. Componentes dentro de uma mesma caixa pertencem ao mesmo processo. A *Nuvem de Solucionadores Secundários* indica a presença de mais de um *Solucionador Secundário*. As setas indicam o sentido da comunicação.

As próximas subseções descrevem os detalhes de cada componente.

### 5.4.1 Tradutor

O *Tradutor* converte o par estrutura *Kripke* e propriedades a serem verificadas em uma fórmula proposicional. Nosso algoritmo não interfere em nada na maneira como funciona essa tradução.

### 5.4.2 Particionador

A função do *Particionador* é criar a partição primária, a partição secundária, instanciar os solucionadores e enviar a eles suas respectivas partições.

Para a criação das partições este componente executa o processo descrito na seção 5.2.

No nosso algoritmo temos apenas um solucionador primário e vários solucionadores secundários. O número de solucionadores secundários é um parâmetro do algoritmo. A sugestão é de que tenhamos um solucionador por nodo do cluster.

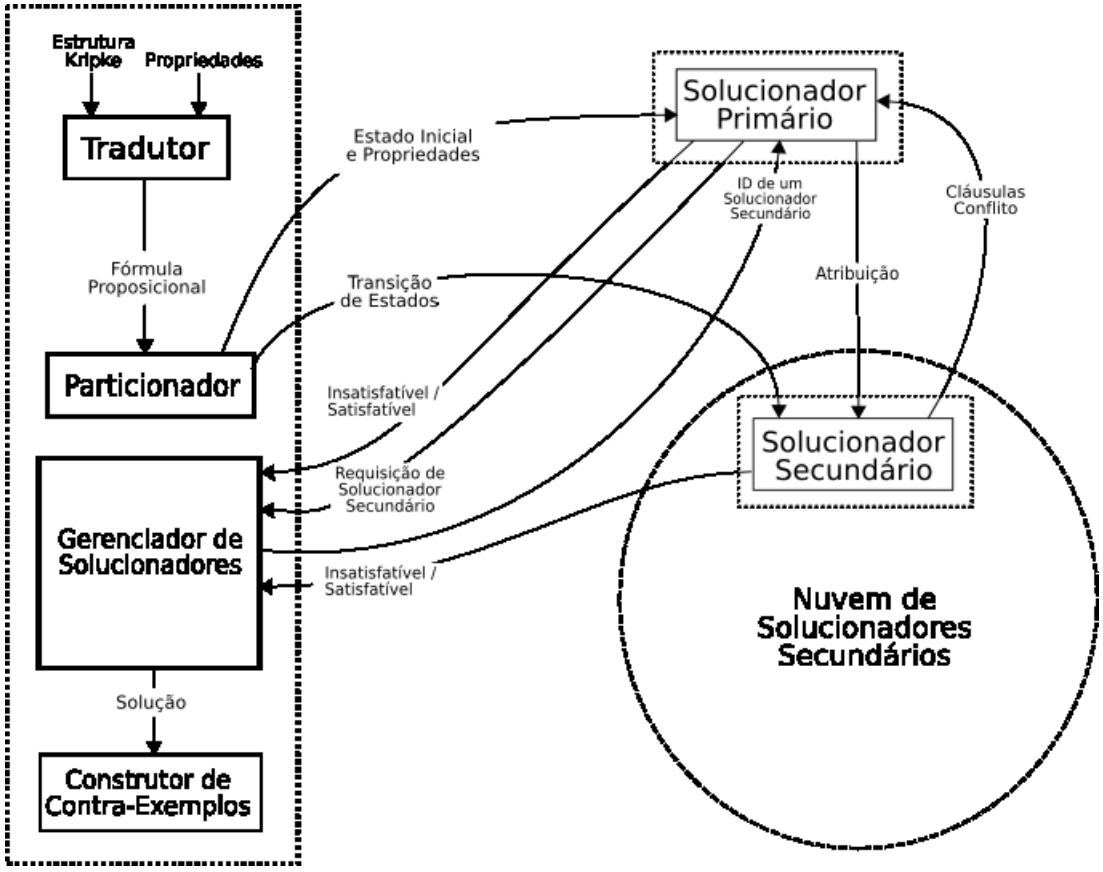


Figura 5.5: Visão Geral do Verificador de Modelos Distribuído

### 5.4.3 Construtor de Contra-Exemplos

O Construtor de Contra-Exemplos tem como única função traduzir a solução encontrada, que está na forma de uma atribuição para uma fórmula proposicional para um caminho no modelo.

Esse caminho ao ser percorrido vai demonstrar onde a propriedade especificada não foi atendida no modelo.

Nosso algoritmo é indiferente à forma como é implementado o construtor de contra-exemplos.

### 5.4.4 Gerenciador de Solucionadores

O *Gerenciador de Solucionadores* tem duas funções. A primeira é atender as requisições de solucionadores secundários livres feitas pelo solucionador primário. A segunda é detectar a terminação do algoritmo.

Para executar estas duas funções este componente deve manter uma lista atualizada dos estados de cada solucionador. Cada entrada dessa lista vai conter o par: solucionador e estado. Os solucionadores podem estar: ocupados, ociosos, aguardando.

A execução do Gerenciador de Solucionadores é orientada de acordo com as mensagens que ele recebe dos solucionadores. São quatro tipos de mensagens possíveis:

1. *Solucionador Primário encontrou solução para sua partição.* Neste caso o Gerenciador deve procurar por um Solucionador Secundário ocioso. Se encontrar ele deve enviar o identificador deste último ao Solucionador Primário. Senão ele deve colocar o Solucionador Primário no estado de espera e aguardar por uma nova mensagem.

2. *Solucionador Primário não encontrou solução para sua partição.* Neste caso o Gerenciador deve marcar o Solucionador Primário como ocioso. Depois deve verificar se todos os solucionadores estão ociosos. Caso todos estiverem o problema é insatisfazível. Senão ele deve aguarda uma nova mensagem.

3. *Solucionador Secundário não encontrou solução para sua partição* Neste caso o Gerenciador deve primeiro verificar se o Solucionador Primário está em estado de espera. Se estiver ele deve enviar o identificador do Solucionador Secundário que enviou a mensagem para o Solucionador Primário. Senão ele deve marcar o Solucionador Secundário como ocioso e depois consultar se todos os solucionadores estão ociosos. Se estiverem então o problem é insatisfazível. Senão o Gerenciador deve aguardar uma nova mensagem.

4. *Solucionador Secundário encontrou solução para sua partição* Neste caso o problem é satisfazível. O Gerenciador deve enviar a solução ao Construtor de Contra-Exemplos.

A terminação do algoritmo é portanto dada quando todos os solucionadores estão ociosos ou quando uma solução é encontrada por um solucionador secundário.

A figura 5.6 ilustra o funcionamento do Gerenciador de Solucionadores.

### 5.4.5 Solucionador Primário

O Solucionador Primário é o componente que inicia a busca por uma atribuição satisfazível. Ele explora o espaço de atribuições da fórmula $I(s_0) \wedge [\![P]\!]^0_k$, a partição primária.

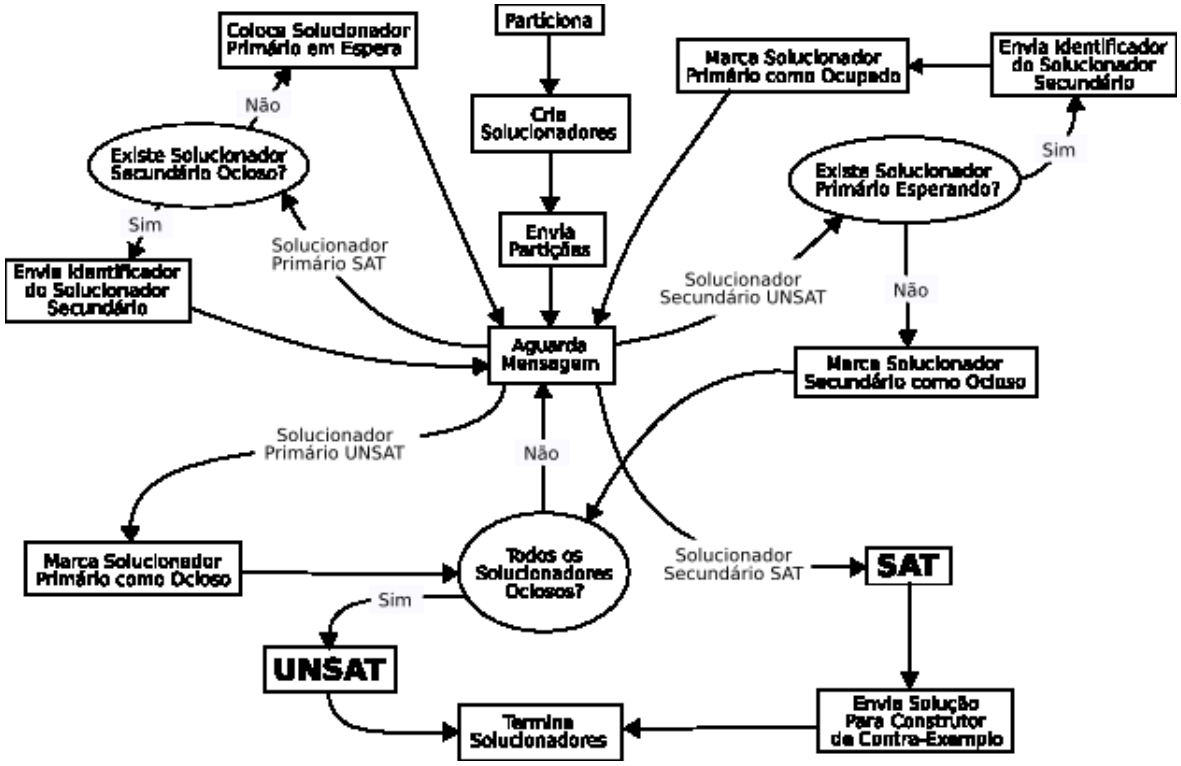


Figura 5.6: Fluxograma do Particionador e do Gerenciador de Solucionadores

Basicamente executa-se um solucionador SAT sobre as cláusulas que representam esta partição, mas ao invés de interromper a busca ao encontrar uma solução a busca prossegue em busca de soluções alternativas. Para evitar que a mesma solução seja encontrada, uma cláusula conflito é gerada a partir da solução encontrada e adicionada ao banco de cláusulas.

Este componente recebe três tipos de mensagens:

1. *Partição.* A mensagem contém as cláusulas que representam a partição primária.

2. *Atribuição Conflitante.* A mensagem contém uma atribuição conflitante identificada por um solucionador secundário. A partir dessa atribuição o Solucionador Primário poderá gerar uma ou mais cláusulas conflito e diminuir o tamanho da árvore de busca.

3. *Identificador de Solucionador Secundário.* A mensagem contém um identificador de um solucionador secundário. O Solucionador Primário usará este identificador para enviar uma solução encontrada a um solucionador secundário.

E envia três tipos de mensagens:

1. *Insatisfazível.* Esta mensagem é enviada ao gerenciador de conflitos indicando que não foi possível encontrar mais uma solução para a partição primária. Lembre-se que outras soluções já podem ter sido encontradas e isso portanto não caracteriza que o problema é insatisfazível.

2. *Requisição de Solucionador Secundário.* Esta mensagem é enviada ao Gerenciador de Solucionadores sempre que o Solucionador Primário precisa de um solucionador secundário para verificar se uma solução encontrada para a partição primária atende a todas as outras partições.

3. *Solução.* Esta mensagem contém uma solução encontrada para a partição primária e é enviada a um solucionador secundário ocioso.

O fluxograma desse componente está apresentado na figura 5.7.

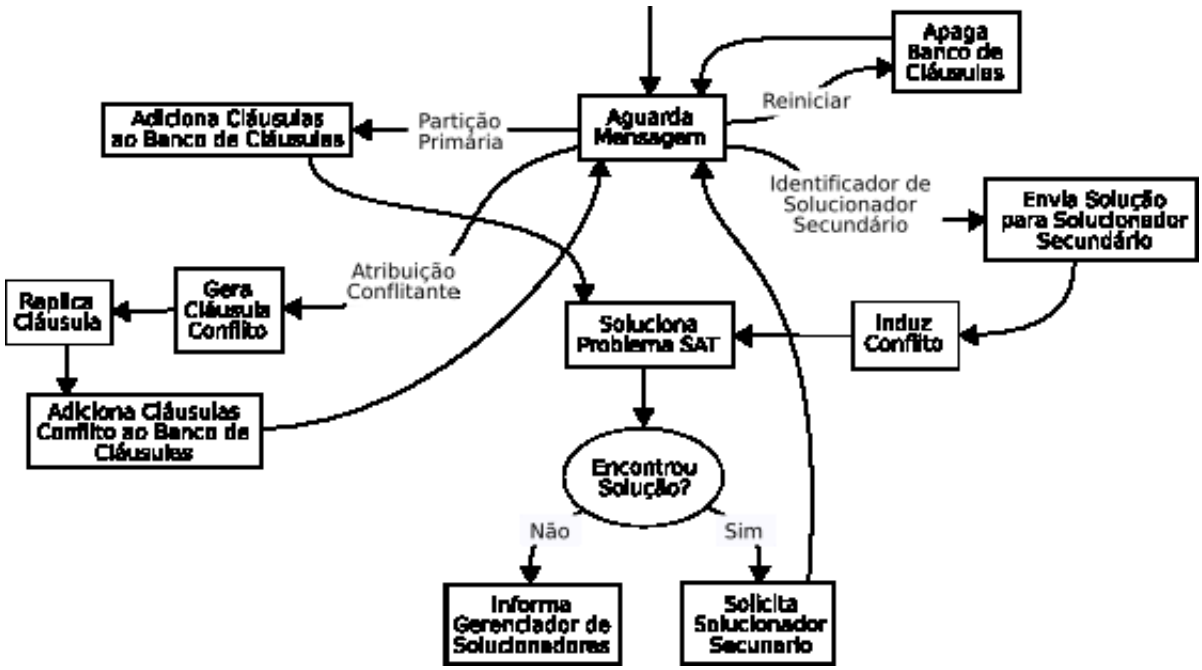


Figura 5.7: Fluxograma do Solucionador Primário

### 5.4.6 Solucionador Secundário

O Solucionador Secundário é o componente que explora todas as partições posteriores à partição primária. A exploração que ele realiza inicia-se a partir de uma solução encontrada pelo Solucionador Primário.

As partições são exploradas seqüencialmente de acordo com sua organização temporal (raciocínio explicado na seção 5.2). O fluxograma desse componente está apresentado na figura 5.8. Vamos usá-lo para explicar o funcionamento do Solucionador Secundário.

O solucionador secundário usa três estruturas de dados principalmente:

1. *Banco de Cláusulas*: é a estrutura de dados que contém as cláusulas que representam o problema SAT a ser explorado. A implementação dessa estrutura é responsabilidade do solucionador SAT.

2. *Atribuição Inicial*: é uma estrutura de dados que contém pares de variáveis e valores. Essa informação é transformada em cláusulas unitárias que são usadas para direcionar a exploração do problema SAT.

3. *Atribuição Global*: é uma estrutura de dados que também contém pares de variáveis e valores. A diferença é que a Atribuição Inicial é definida pelo Solucionador Primário e pode não contiver todas as variáveis do modelo. A Atribuição Global não é uma estrutura de conteúdo fixo e quando uma solução é encontrada ela contém todas as variáveis do modelo. Esta estrutura é enviada ao Construtor de Contra-Exemplos quando o problema é satisfazível.

Pode receber quatro tipos de mensagens:

1. *Partição*. A mensagem contém as cláusulas que representam as partições secundárias.

2. *Atribuição Inicial*. A mensagem contém uma solução enviada pelo Solucionador Primário, a partir da qual o Solucionador Secundário iniciará a exploração das partições secundárias.

3. *Reiniciar*. A mensagem contém uma ordem para reiniciar o solucionador. Este procedimento consiste em apagar o banco de cláusulas e as atribuições global e inicial. O solucionador volta para seu estado inicial.

4. *Finalizar*. A mensagem contém uma ordem de finalização do solucionador. Este procedimento consiste em finalizar o solucionador, descartando qualquer processamento em andamento.

E envia três tipos de mensagens:

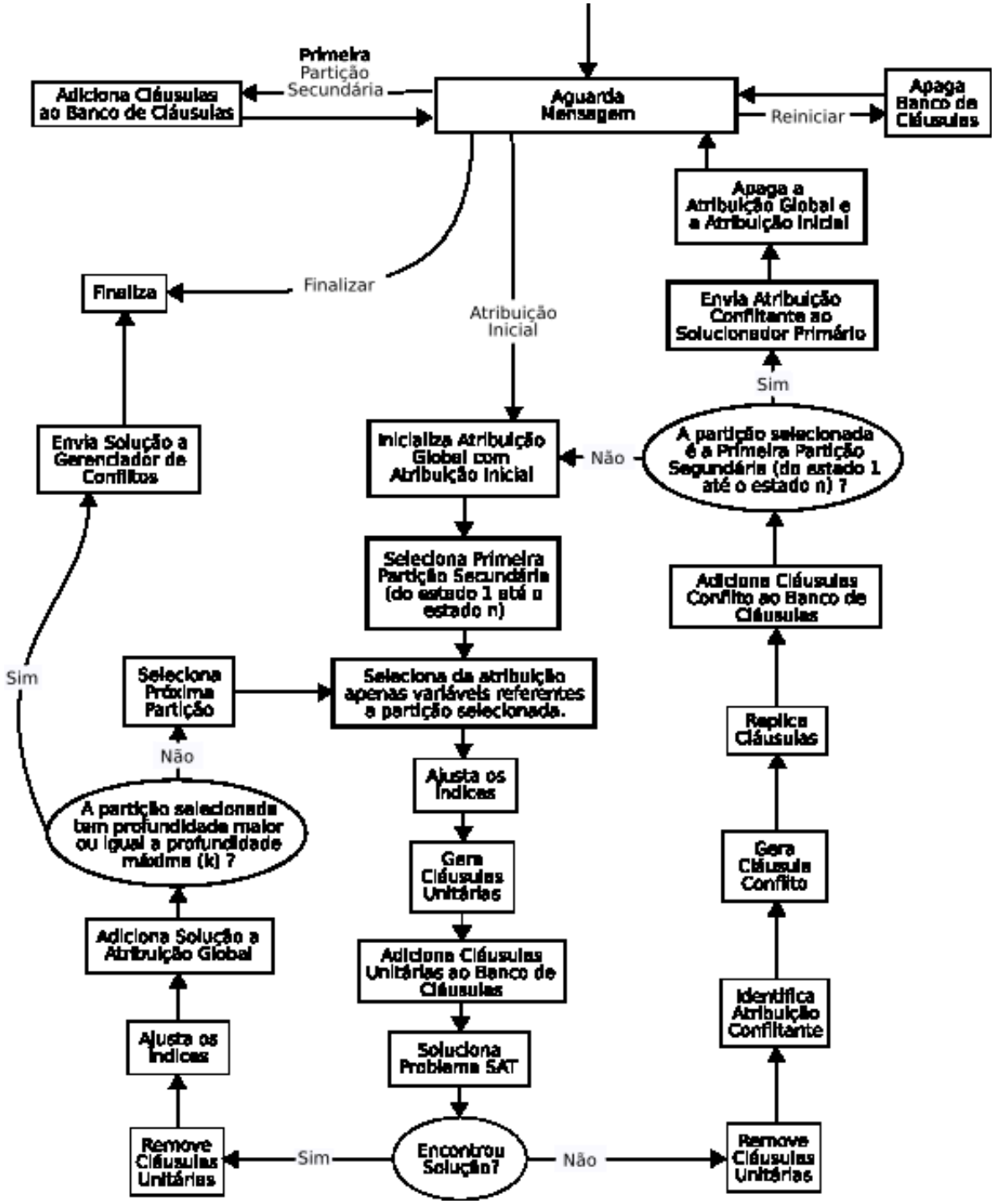


Figura 5.8: Fluxograma do Solucionador Primário

1. *Insatisfazível*. Essa mensagem é enviada ao Gerenciador de Conflitos indicando que não foi possível encontrar uma solução para seu problema.

2. *Atribuição Conflitante*. Esta mensagem é enviada ao Solucionador Primário quando um conflito é encontrado. Ela contém as atribuições que geraram um conflito nas partições secundárias. Esta informação é necessária para a geração de cláusulas conflito.

3. *Satisfazível*. Esta mensagem contém uma solução encontrada para as partições primária e secundária é enviada ao Gerenciador de Solucionadores. Seu envio significa que uma solução para o problema completo foi encontrada.

O interessante nesse solucionador é como ele explora várias partições com apenas um banco de cláusulas compartilhando automaticamente todas cláusulas conflito encontradas.

Seguindo no fluxograma observamos que inicializamos a Atribuição Global com a Atribuição Inicial. Selecionamos então a primeira partição secundária para ser explorada. Da Atribuição Global selecionamos apenas os pares cujas variáveis estão presentes na partição selecionada.

O próximo passo é fazer os ajustes dos índices das variáveis selecionadas. Este ajuste basicamente diminui os índices das variáveis escolhidas para que essas sejam mapeadas no banco de cláusulas. Esse passo é importantíssimo pois é ele que nos permite usar um mesmo grupo de cláusulas para explorar várias partições. Note que ele não é necessário quando estamos lidando com a primeira partição secundária porque são as cláusulas dessa partição que compõem o banco de cláusulas.

Em seguida converte-se os pares selecionados em cláusulas unitárias que são adicionadas ao banco de cláusulas para direcionar a busca. E finalmente executa-se o solucionador SAT.

Dependendo da resposta do solucionador SAT dois caminhos podem ser tomados. O primeiro é quanto o solucionador encontra uma solução para a partição explorada. Neste caso removemos as cláusulas unitárias que havíamos incluído do banco de cláusulas. Ajustamos os índices da solução para que ela reflita a partição explorada e depois adicionamos a solução encontrada a Atribuição Global. Se a partição explorada conter o estado k, que é o limite estipulado para BMC é sinal que temos uma solução para o problema, senão temos que partir para a exploração da próxima partição. Repete-se o procedimento a partir da seleção dos pares na atribuição global.

A outra possibilidade é o solucionador SAT não encontrar uma solução para a partição explorada. Neste caso removemos as cláusulas unitárias e

partimos para a identificação da atribuição conflitante. Uma vez identificada a atribuição conflitante geramos a cláusula conflito e a replicamos se possível. Se a partição selecionada for a primeira é sinal de que o conflito foi originado pela atribuição inicial, que deve ser descartada, e o Solucionador Primário informado. Senão devemos reiniciar a busca pela primeira partição secundária.

Note que todas as cláusulas conflitos são geradas para um mesmo conjunto de cláusulas, ou seja, são compartilhadas automaticamente entre as partições. Poderíamos compartilhar as cláusulas conflito entre os solucionadores secundários, mas optamos por não fazer isso para diminuir a comunicação.

## 5.5 Trabalhos Relacionados

A investigação sobre algoritmos paralelos ou distribuídos para BMC é uma área ainda pouco explorada. Por outro lado existem muitos trabalhos para paralelizar solucionadores SAT. A maioria deles divide a busca e não as cláusulas, embora haja ganho na velocidade não há ganhos na utilização de memória.

Existe um trabalho [71] que distribuí as cláusulas igualmente entre vários processos, objetivando escalabilidade. No entanto, eles pecaram por não cuidar da separação das variáveis. Dessa forma entre as partições existem muitas variáveis compartilhadas. Aumentando muito a comunicação. Os próprios autores notaram que 90% das mensagens eram de difusão (*broadcast*).

Para BMC existe um trabalho [69] que foi inclusive usado como ponto de partida para esta dissertação. Eles apresentaram uma abordagem na qual nenhum processo retia o problema completo. A principal diferença da abordagem deles com relação à proposta nesta dissertação é a forma como é feita a exploração. As principais diferenças são:

- Enquanto que no nosso algoritmo os processos podem trabalhar de forma independentes nas suas partições, no caso deles os processos tem que estar sempre sincronizados.
- No caso deles a escolha de variáveis de decisão é feita de forma global, enquanto que fazemos isso localmente.
- Na abordagem deles as implicações tem que ser transmitidas de um nodo para o outro enquanto que no nosso caso não é necessário transmitir implicações já que os processos trabalham de forma independente.
- No algoritmo deles todos os processos tem que saber como está disposta a topologia para que a comunicação seja reduzida. No nosso caso não há

essa necessidade, temos apenas um despachador que indica o identificar de qual processo está ocioso.

- O retrocesso na abordagem deles é realizado através da sincronia de todos os processos. No nosso caso o retrocesso é realizado localmente pelo solucionador SAT independente do nosso algoritmo.

# Capítulo 6

# Resultados Experimentais

## 6.1 Introdução

Neste capítulo vamos descrever o processo utilizado para se avaliar o algoritmo distribuído proposto e os resultados obtidos por essa avaliação.

A maneira que optamos para avaliá-lo é a comparação ao método de verificação de modelos com fronteiras baseada em SAT usando um método seqüencial. No decorrer do texto vamos nos referir a ele como *monolítico*. Tínhamos duas opções: implementar os dois métodos ou conseguir uma implementação do método monolítico e aproveitar o código para implementar o distribuído.

O software de código aberto NuSMV [72] se encaixou perfeitamente nesta necessidade. Ele já implementa o método monolítico e é bastante modularizado, permitindo reaproveitamento de componentes. Além dessa vantagens, o NuSMV se integra facilmente como o solucionador SAT ZChaff [37], que é um dos melhores da atualidade [73]. O último componente necessário era uma biblioteca para passagem de mensagens que serviria para distribuir o trabalho entre vários processos. A escolhida foi a LAM/MPI [61, 74], que é uma das melhores implementações do padrão MPI. [61].

Todos os componentes citados funcionavam sem problemas sobre o sistema operacional Linux e este portanto foi o sistema operacional escolhido.

Um vez implementado o algoritmo, tínhamos que selecionar os modelos que seriam verificados e que serviriam de entrada para os verificadores. No nosso caso todos os modelos teriam que ser escritos na linguagem do NuSMV. Este por sua vez já oferece em seu pacote uma grande variedade de modelos já escritos na sua linguagem, de onde foram selecionados modelos para os experimentos.

Os testes foram feitos da seguinte forma: para cada modelo executa-se

o método monolítico até que o tempo de resposta se torne impraticável, em seguida executa-se o mesmo teste para o distribuído até que a mesma profundidade tenha sido alcançada. As métricas colhidas em cada execução foram: tempo de resposta, número de cláusulas SAT geradas e utilização de memória pelo solucionador SAT.

Na próxima subseção vamos detalhar os componentes selecionados para a implementação. Em seguida vamos descrever os modelos selecionados para testes. Por fim, mostramos e explicamos os resultados dos experimentos.

## 6.2 Implementação

Nessa seção descrevemos os componentes usados para a implementação do algoritmo proposto.

### 6.2.1 NuSMV

O NuSMV é um verificador de modelos simbólico originário da reengenharia, da reimplementação e da extensão do SMV [4]. O objetivo do NuSMV é estar no estado-da-arte de técnicas de verificação de modelos simbólicas. O NuSMV é um software bem estruturado, modularizado, bem documentado e de código aberto (licença LGPL).

O NuSMV é mantido por quatro grupos de pesquisa. O italiano *Formal Methods Group in the Automated Reasoning System* do ITC- IRST, o americano *Model Checking Group* da *Carnegie Mellon University*, o italiano *Mechanized Reasoning Group* da *University of Genova* da Itália e o também italiano *Mechanized Reasoning Group* da *University of Trento*.

A última versão do NuSMV implementa basicamente verificação de modelos simbólica baseada em BDDs e em solucionadores SAT, técnicas que para muitos são consideradas complementares.

O desenvolvedores do NuSMV buscaram manter o código o mais modularizado possível fazendo as várias funcionalidades do verificador o mais independentes possível. Esta foi uma das principais características que permitiu a implementação de técnicas como BDDs e SAT com o compartilhamento máximo de código.

O NuSMV processa arquivos escritos na linguagem SMV. Nessa linguagem é possível descrever uma máquina de estados finitos por mecanismos de declaração e criação de módulos e processos, correspondendo a um composição síncrona ou assíncrona. Na descrição de modelos as propriedades a serem verificadas podem ser escritas em LTL ou CTL.

Dentro do NuSMV o arquivo de entrada é processado em várias fases. As primeiras fases consistem na análise deste arquivo para que uma representação interna do mesmo possa ser construída. As fases fundamentais da análise são:

1. *Nivelamento.* Neste passo cada módulo e cada processo é instanciado produzindo um modelo síncrono e nivelado onde cada variável tem um nome absoluto.

2. *Codificação Lógica* Neste passo o modelo nivelado é mapeado em um modelo lógico, onde todas as variáveis escalares são convertidas para variáveis lógicas.

Depois da análise segue a fase em que atua o motor de verificação, que pode ser baseado em SAT ou BDDs. As fases poderiam ser descritas da seguinte forma:

1. *Motor de Verificação baseado em BDDs* Neste caso uma representação da máquina de estados finitos em BDDs é construída. Em seguida variações de verificação baseada em BDDs podem ser usadas.

2. *Motor de Verificação baseado em SAT* Neste caso o NuSMV constroí uma representação interna do modelo no formato de uma versão simplificada de um circuito lógico reduzido (*Reduced Boolean Circuit - RBC*), que é um mecanismo de representação de fórmulas proposicionais. A partir do RBC é possível realizar verificação de modelos com fronteiras baseada em SAT e com propriedades LTL. O RBC é convertido em uma fórmula CNF que serve de entrada para um solucionador SAT. No caso do NuSMV já existem dois solucionadores integrados: o SIM e o ZChaff. Se o usuário preferir usar outro solucionador ele pode ainda gerar um arquivo DIMACS que é um padrão compreendido por qualquer solucionador SAT.

Depois da verificação, temos o último passo que é a geração de contra-exemplos. Se o verificador encontrar uma situação na qual a propriedade não é satisfeita este passo transforma essa situação em um caminho, que é o contra-exemplo.

Por fim, o NuSMV é um software livre recomendado tanto para a aqueles que querem fazer a verificação de modelos profissionalmente quanto para pesquisadores que querem experimentar novas técnicas de verificação simbólica de modelos.

### 6.2.2 ZChaff

O ZChaff é reconhecidamente um dos melhores solucionadores SAT da atualidade. Isso pode ser comprovado por suas premiações na tradicional *SAT Competition* que acontece todos os anos acompanhando um dos mais importantes eventos da área de solucionadores SAT. Na competição de 2002 ele levou o prêmio de solucionador mais completo e em 2004 ele levou o prêmio de melhor solucionador da categoria industrial, que engloba problemas gerados a partir de casos reais apresentados pela indústria.

Além de todo este cartão de visitas ele já é integrado ao NuSMV que o deixa em vantagem com relação a todos os outros solucionadores no que diz respeito a nossa escolha de componentes para implementação.

O ZChaff é mantido pelo *Boolean Satisfiability Research Group* da *Princeton University*. Sua terceira versão foi disponibiliza em maio de 2004, o que mostra que é um projeto ativo.

Maiores detalhes sobre o ZChaff podem ser encontrados nas referências [37, 19, 29].

### 6.2.3 MPI

A Interface de Passagem de Mensagens (*Message Passing Interface (MPI)*) é um protocolo de comunicação entre computadores. A MPI surgiu quando a indústria e os centros de pesquisa sentiram a necessidade de definir uma biblioteca padrão para o paradigma de passagem de mensagens. Surgiu então o Fórum MPI, que é um grupo de oitenta pessoas de quarenta organizações representando vendedores de sistemas paralelos, usuários industriais, laboratórios de pesquisa e universidades. Estas pessoas passaram então a desenvolver o padrão MPI buscando os seguintes objetivos:

- Desenvolver uma API se preocupando principalmente com as necessidades dos programadores.
- Permitir comunicação eficiente.
- Permitir a utilização em ambiente heterogêneos.
- Permitir a utilização com as linguagens Fortran e C.
- Prover uma comunicação confiável.
- Definir uma interface similar com as utilizadas até então.
- Definir uma interface que tenha o mínimo interferência nas plataformas já existentes de hardware e redes de computadores.

- Definir uma interface que suporte a utilização de *threads*.

As maiores vantagens de MPI sobre outras interfaces é que ela é portável, já que MPI já foi implementada para quase todas as arquiteturas de memória distribuída, e rápida, já que cada implementação é otimizada para o hardware na qual ela executa.

As implementações mais populares dessa interface são a MPICH [75, 76] e a LAM/MPI [77, 74].

Na nossa implementação optamos pela LAM/MPI pois ela implementa mais funcionalidades da versão 2 da MPI. Uma dessas funcionalidades é a instanciação de processos por um outro processo. Isto é importante pois nos permite calcular quantos processos serão necessários diretamente de dentro do programa.

### 6.2.4 Linux

Todos os componentes citados até então funcionam perfeitamente bem no Linux. Esta é a principal razão para a escolha deste sistema operacional como plataforma de experimentos.

Além dessa vantagem o Linux oferece várias ferramentas que facilitam a vida de desenvolvedores, desde compiladores e depuradores a até ambientes de desenvolvimentos integrados (*IDE - Integrated Development Environments*).

Entre as distribuições Linux usadas, foi escolhida a distribuição Debian. A razão para essa escolha é o excelente sistema de pacotes fornecido por essa distribuição. Este sistema facilitou muito a instalação de todos os aplicativos necessários em um cluster.

## 6.3 Experimentos

Nesta seção definimos com mais detalhes como foram feitos os experimentos com a implementação do algoritmo proposto por este trabalho.

### 6.3.1 Ambiente de Testes

Os testes foram todos executados no cluster do CAID (Centro de Acesso à Informação Digital) da UFMG. Estes cluster é composto por vinte e dois nodos com mesma configuração de hardware e sistema operacional.

O sistema operacional utilizado é o Debian/Linux e ele está instalado de forma homogênea em todos os nodos.

A configuração de hardware de um nodo cluster é:

- Processador: Intel Pentium 4, 3.00 Ghz, 1024 Kb Cache L2
- Memória: 1 GB
- Disco Rígido: 80.0 GB Serial ATA
- Placa de Rede: 1 Gbps

O switch que atualmente está instalado no CAID permite conexões a até 100 Mbps e por isso não podemos usar todo o potencial dos nodos.

### 6.3.2 Modelos

Os modelos escolhidos para testes foram modelos que acompanham o pacote do NuSMV. A seguir uma tabela que lista os modelos selecionados e o número de variáveis lógicas que eles representam em cada estado:

| **Arquivo** | **Número de Variáveis Lógicas** |
|---|---|
| periodic.smv | 35 |
| p-queue.smv | 43 |
| gigamax_ltl.smv | 49 |
| dme5.smv | 180 |
| dme8.smv | 287 |
| dme10.smv | 360 |
| dme20.smv | 720 |
| dme30.smv | 1080 |

Tabela 6.1: Modelos para Experimentos

Todos os exemplos discutidos a seguir não apresentaram contra-exemplos para a profundidade máxima alcançada pelo método monolítico. Outros modelos, no entanto, apresentaram os mesmos contra-exemplos tanto para o método monolítico quanto para o distribuído. Eles não serão discutidos nesse texto pois são exemplos curtos que cujas métricas de um método e do outro são semelhantes.

Os tempos de resposta que serão mostrados a seguir foram calculados dentro do código do NuSMV. As quantidades de memória se referem à memória máxima utilizada pelo ZChaff durante a execução. As quantidades de cláusulas também são calculadas pelo ZChaff ao término de cada execução do solucionador. Notem que a quantidade de memória do ZChaff depende de quantas iterações ele faz no corpo do seu algoritmo DPLL, isto é, nem sempre a quantidade de memória será proporcional ao de cláusulas.

Em todas as execuções do método distribuído foram usados um solucionador primário e cinco solucionadores secundários. Alterações no número de solucionadores secundários não causou impacto nos resultados, fato que será explicado na seção 6.3.10.

### 6.3.3 Modelo 1: periodic.smv

Um exemplo de implementação de três pipelines.
A propriedade LTL verificada tem a seguinte forma: G (!error).

#### Tempos de Resposta

O gráfico 6.1 mostra os tempos de respostas obtidos pelos dois métodos em uma escala logarítmica. O método distribuído mostrou um desempenho bem superior ao monolítico nesse caso. A presença de poucas variáveis lógicas em uma transição favorece o uso de cláusulas replicadas.

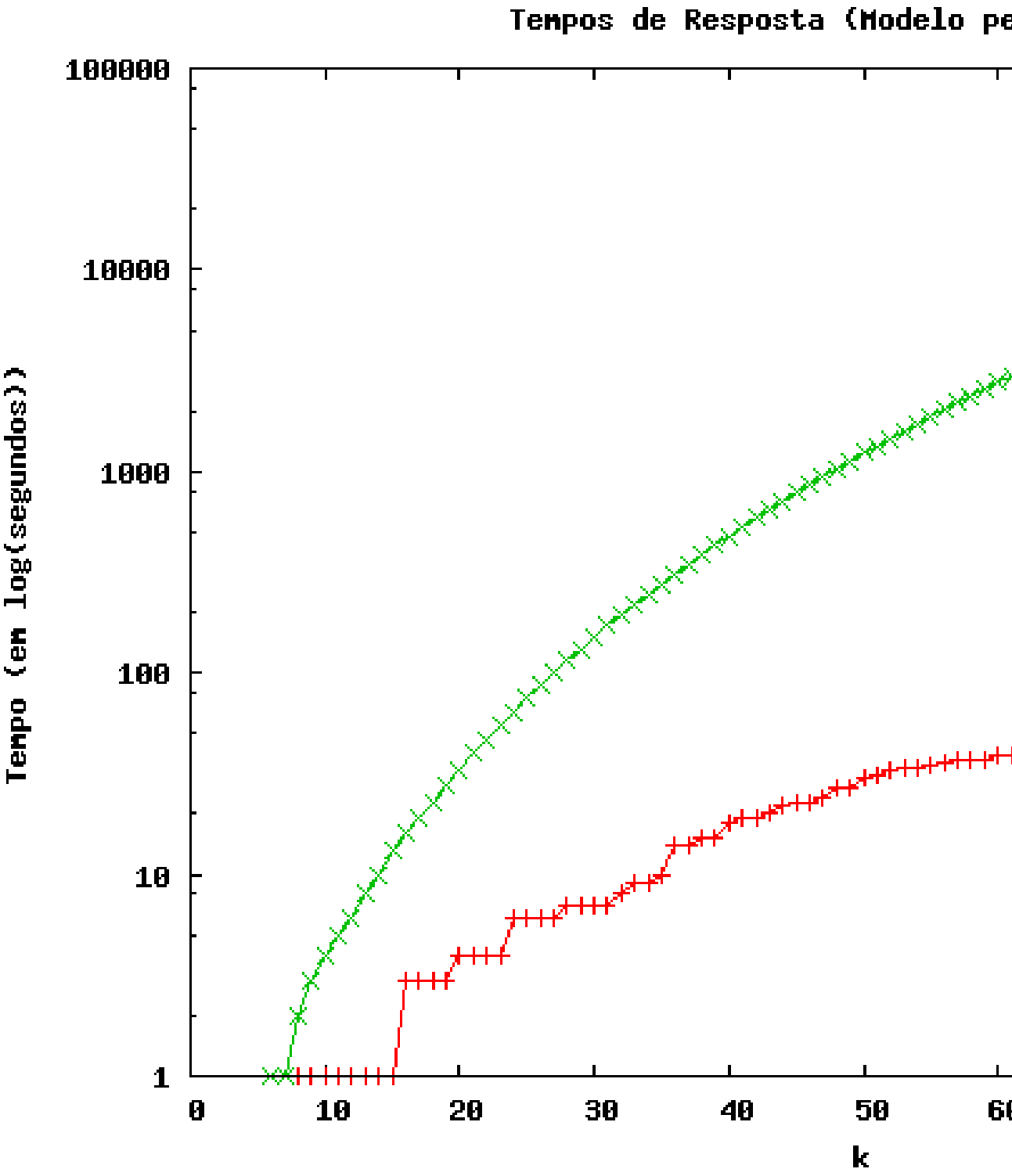


Figura 6.1: Tempos de Resposta do Modelo periodic.smv - Escala Logarítmica

### Utilização de Memória

O gráfico 6.2 mostra a utilização de memória no método monolítico e no método distribuído. No método distribuído foram tomadas medidas tanto do solucionador primário quanto do solucionador secundário. No caso do solucionador secundário estamos lidando com os máximos. Novamente notamos superioridade do método distribuído.

Interessante nesse exemplo é um comportamento do método distribuído que será observado em todos os modelos. Os picos de utilização de memória do solucionador primário sempre coincidem com algum pico do solucionador secundário. Sabemos, através da observação da comunicação, que isso ocorre sempre em que há muita troca de informações entre eles. Outro comportamento notável é que em muitos pontos a utilização de memória pelo solucionador secundário foi nula. Isto acontece porque nesses pontos o solucionador secundário não foi utilizado. O solucionador primário não encontrou soluções para sua partição. Isso se deve a ação das cláusulas replicadas.

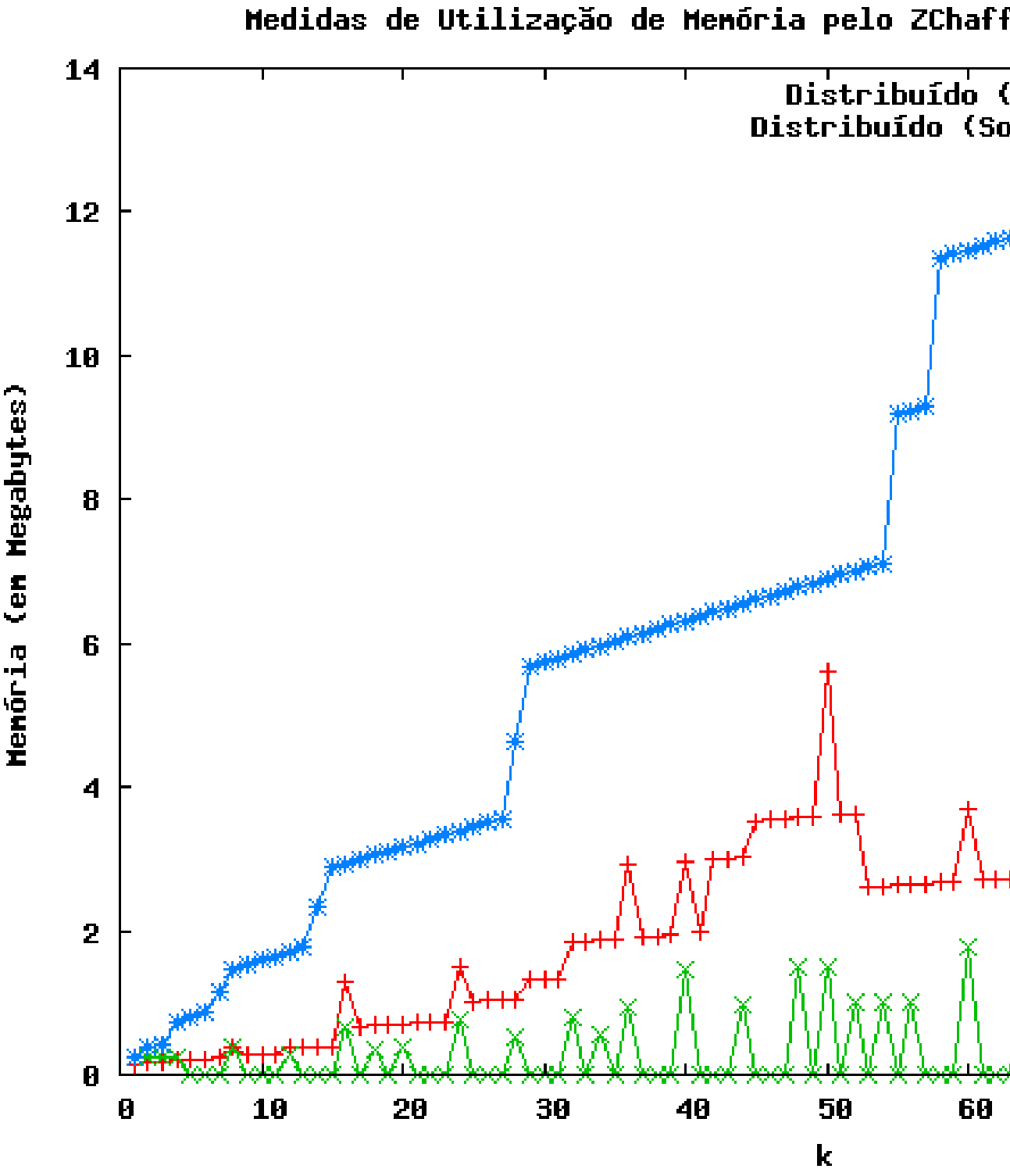


Figura 6.2: Utilização de Memória do Modelo periodic.smv

### Número de Cláusulas

O gráfico 6.3 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído. O método distribuído também mostrou superioridade nesta métrica.

Note o pequeno crescimento no número de cláusulas do solucionador secundário e do solucionador primário, esse acréscimo é devido a cláusulas conflito adicionadas a eles no decorrer da solução.

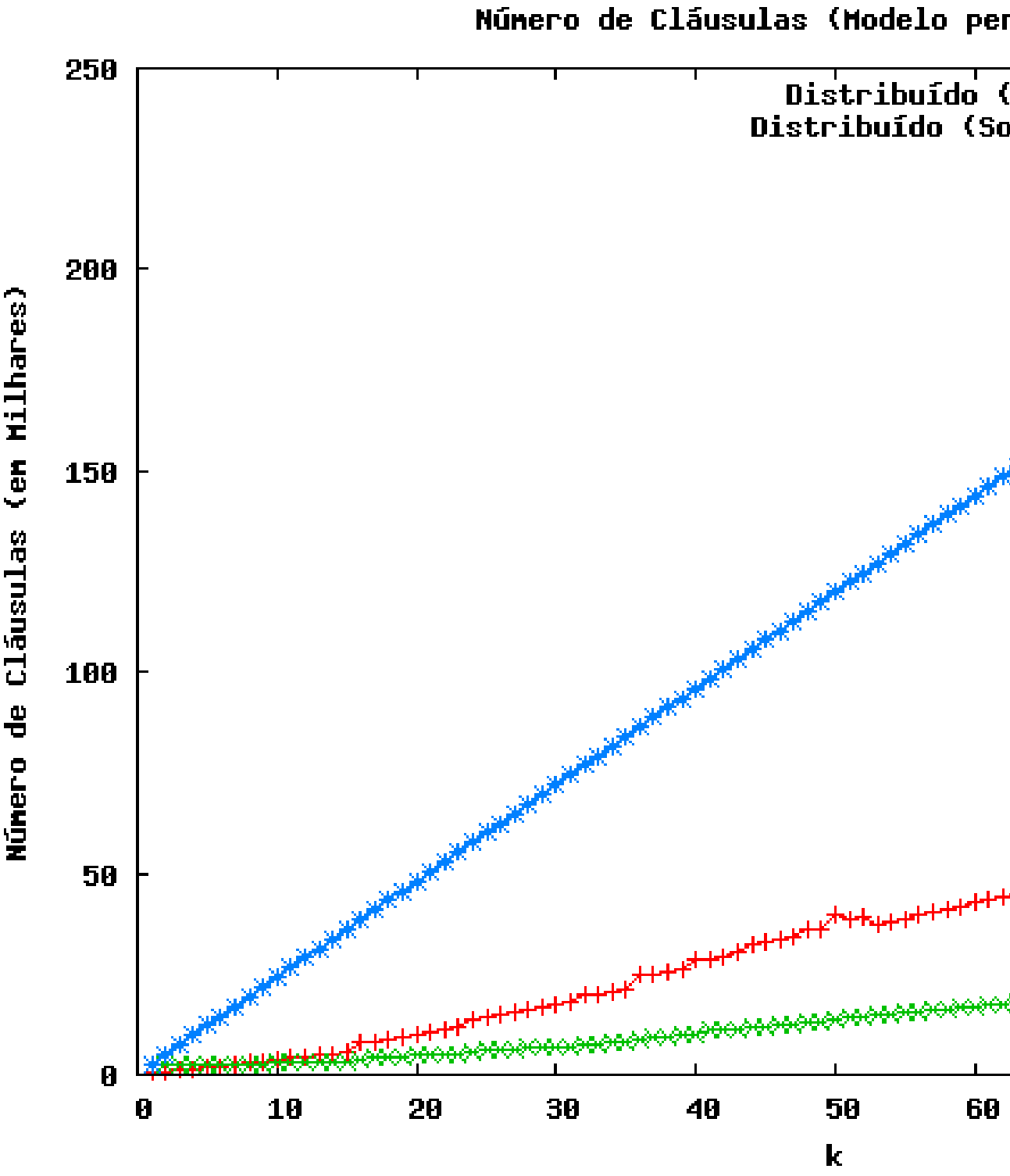


Figura 6.3: Número de Cláusulas do Modelo periodic.smv

### 6.3.4 Modelo 2: p-queue.smv

Um modelo de fila de prioridades.
A propriedade LTL verificada foi: F (out_l[1] = 0).

#### Tempos de Resposta

O gráfico 6.4 mostra os tempos de respostas obtidos pelos dois métodos em uma escala logarítmica. Como no modelo anterior o método distribuído mostrou um desempenho bem superior ao monolítico.

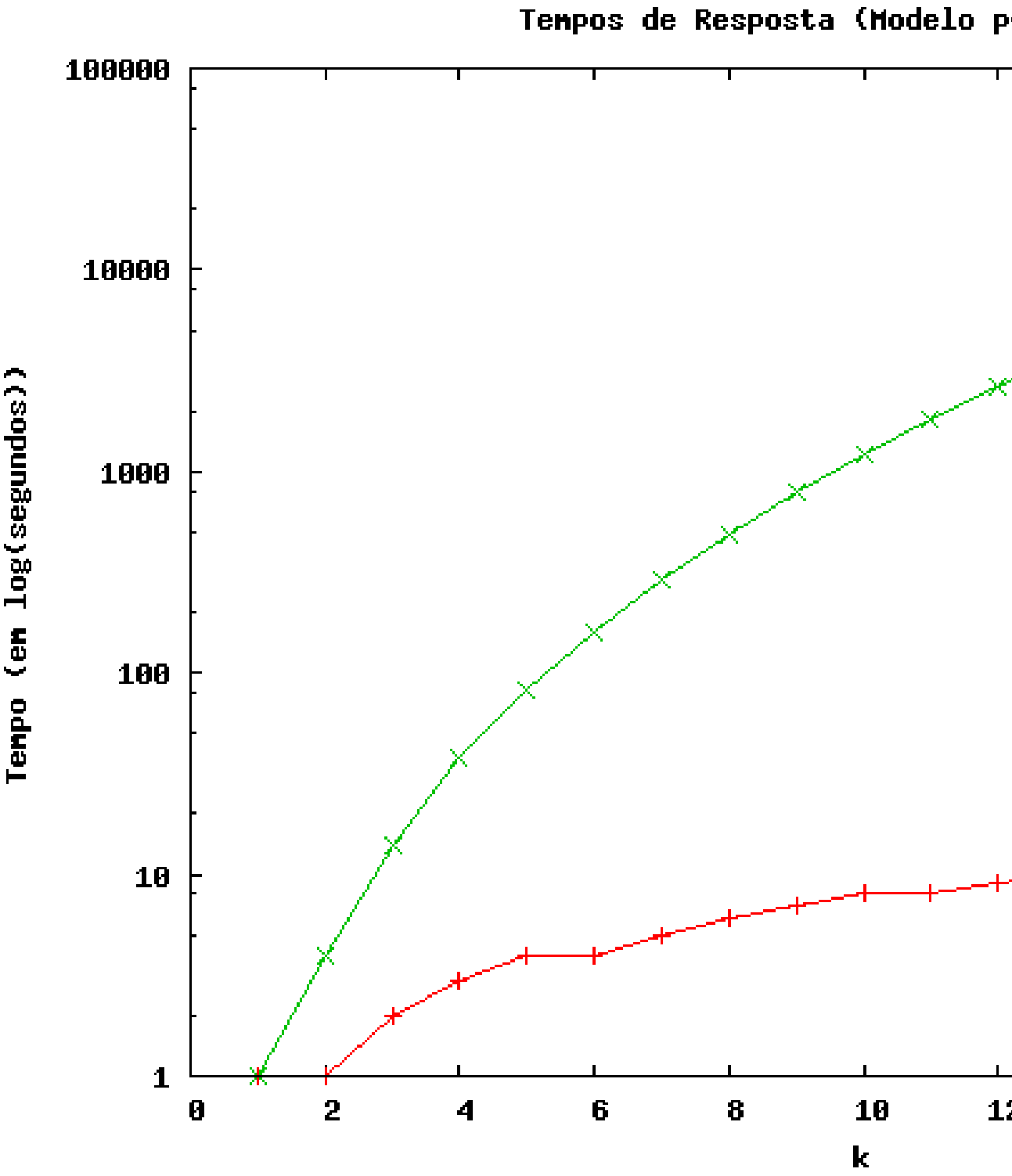


Figura 6.4: Tempos de Resposta do Modelo p-queue.smv - Escala Logarítmica

### Utilização de Memória

O gráfico 6.5 mostra a utilização de memória no método monolítico e no método distribuído em uma escala logarítmica. Novamente notamos superioridade do método distribuído.

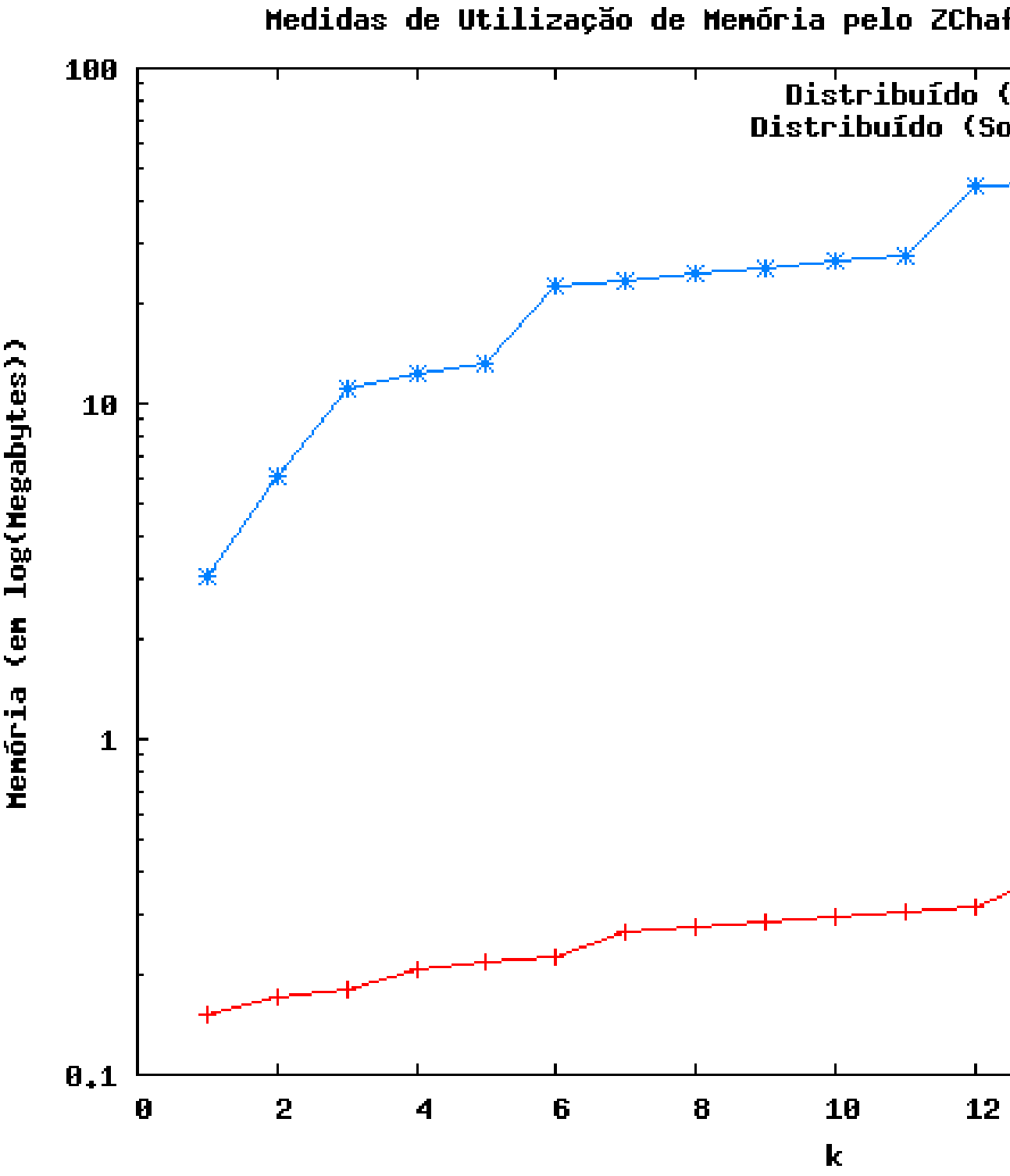


Figura 6.5: Utilização de Memória do Modelo p-queue.smv - Escala Logarítmica

### Número de Cláusulas

O gráfico 6.6 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído em uma escala logarítmica. O método distribuído também mostrou superioridade nesta métrica.

Notamos que neste exemplo o solucionador secundário apresentou um número maior de cláusulas que o solucionador primário. A explicação para este fato é a composição da partição primária que é dependente da fórmula a ser verificada. Neste caso o número de cláusulas geradas para a partição primária foi menor do que as cláusulas geradas para a relação de transição, que compõe a partição secundária.

Note também que o número de cláusulas do solucionador secundário ficou constante durante todo o modelo. Este comportamente é esperado pois este solucionador representa apenas uma transição. As cláusulas conflito adicionadas começam a não fazer diferença significativa.

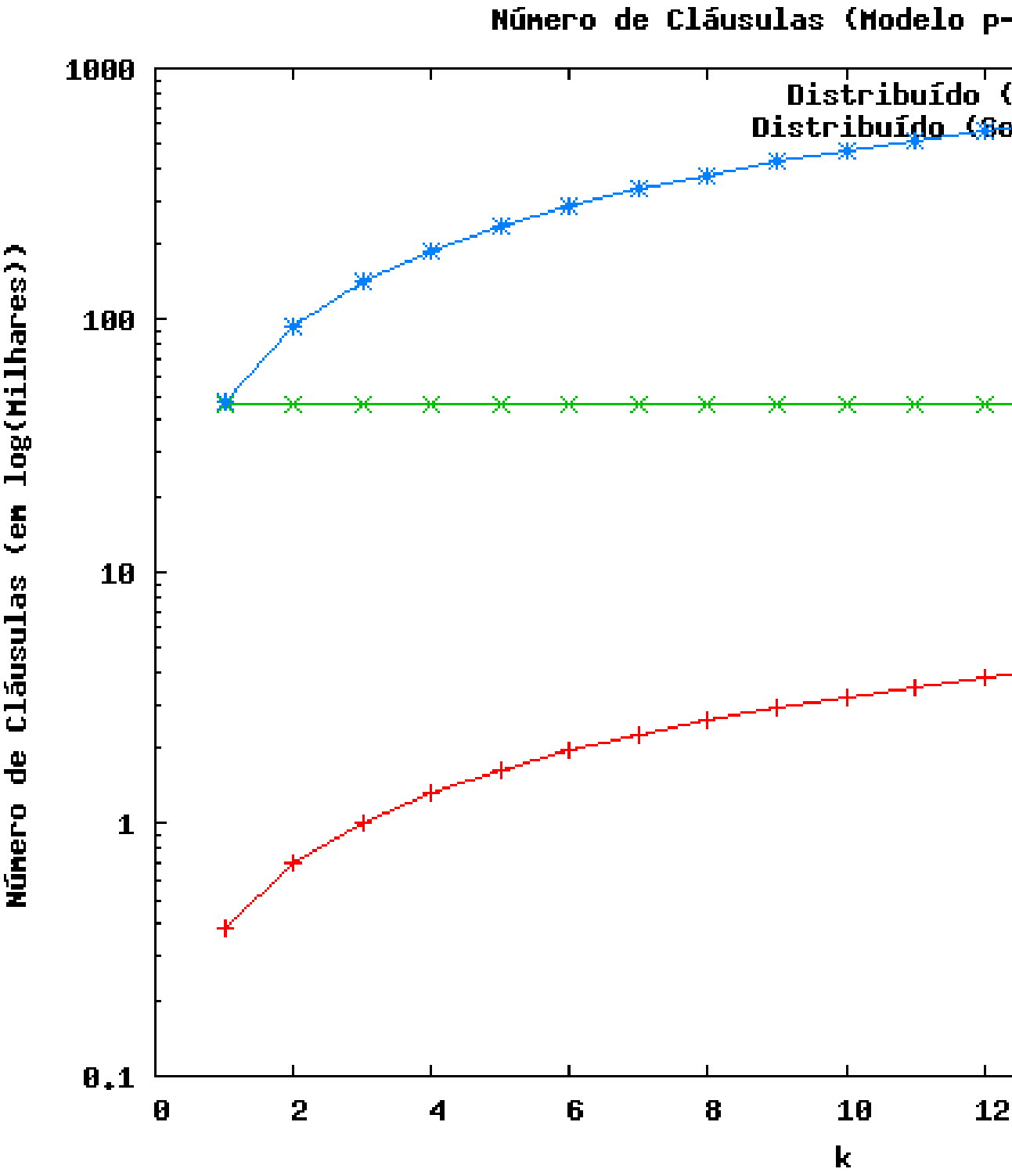


Figura 6.6: Número de Cláusulas do Modelo p-queue.smv - Escala Logarítmica

### 6.3.5 Modelo 3: gigamax_ltl.smv

Um modelo do protocolo de coerência de cache GIGAMAX.
Propriedade LTL verificada: G !(p0.writable & p1.writable).

#### Tempos de Resposta

O gráfico 6.7 mostra os tempos de respostas obtidos pelos dois métodos em uma escala logarítmica. Neste exemplo a diferença de tempos de resposta foi a mais alta. Neste caso o uso de cláusulas conflito apresentou os melhores ganhos. Embora não possamos afirmar supomos que o modelo atende a propriedade verificada para qualquer k.

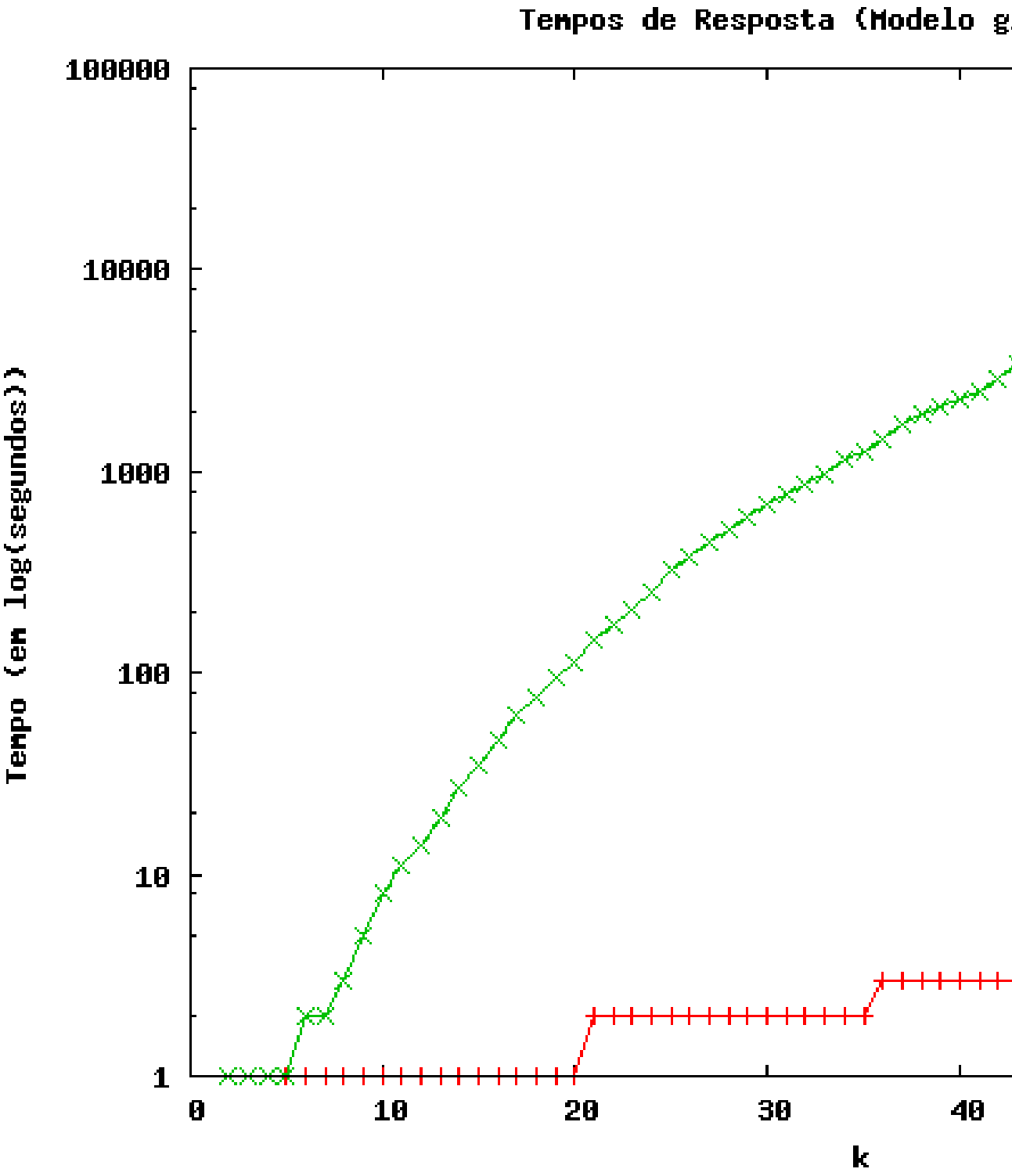


Figura 6.7: Tempos de Resposta do Modelo gigamax_ltl.smv - Escala Logarítmica

### Utilização de Memória

O gráfico 6.8 mostra a utilização de memória no método monolítico e no método distribuído em uma escala logarítmica. Novamente notamos superioridade do método distribuído.

Note que o solucionador secundário nem aparece no gráfico, entendemos que as cláusulas replicadas estão solucionando o problema. Fato que reforça nossa teoria de que o modelo atende a propriedade.

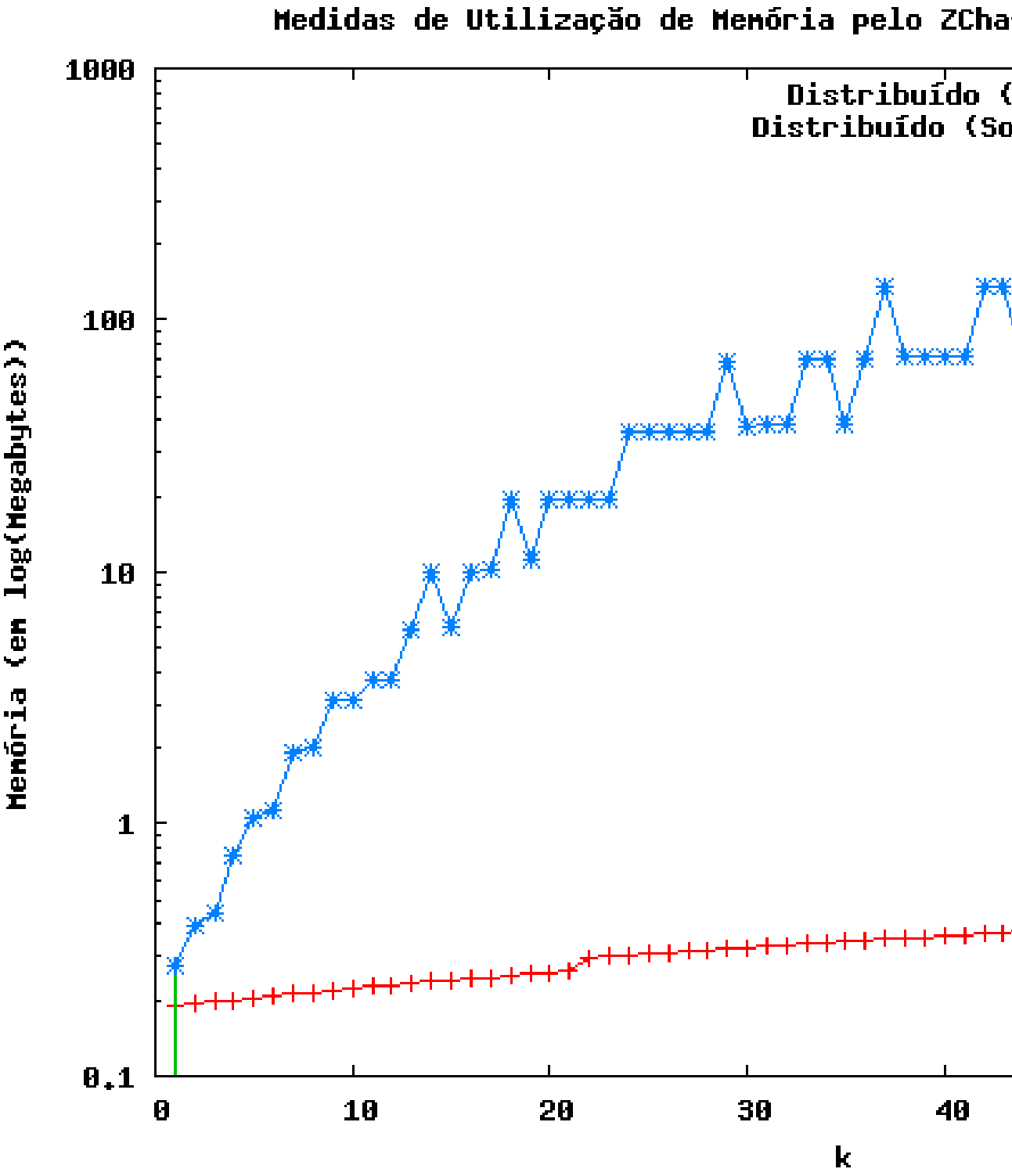


Figura 6.8: Utilização de Memória do Modelo gigamax_ltl.smv - Escala Logarítmica

### Número de Cláusulas

O gráfico 6.9 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído em uma escala logarítmica. O método distribuído também mostrou superioridade nesta métrica.

O comportamento do número de cláusulas é semelhante ao do modelo p-queue.smv.

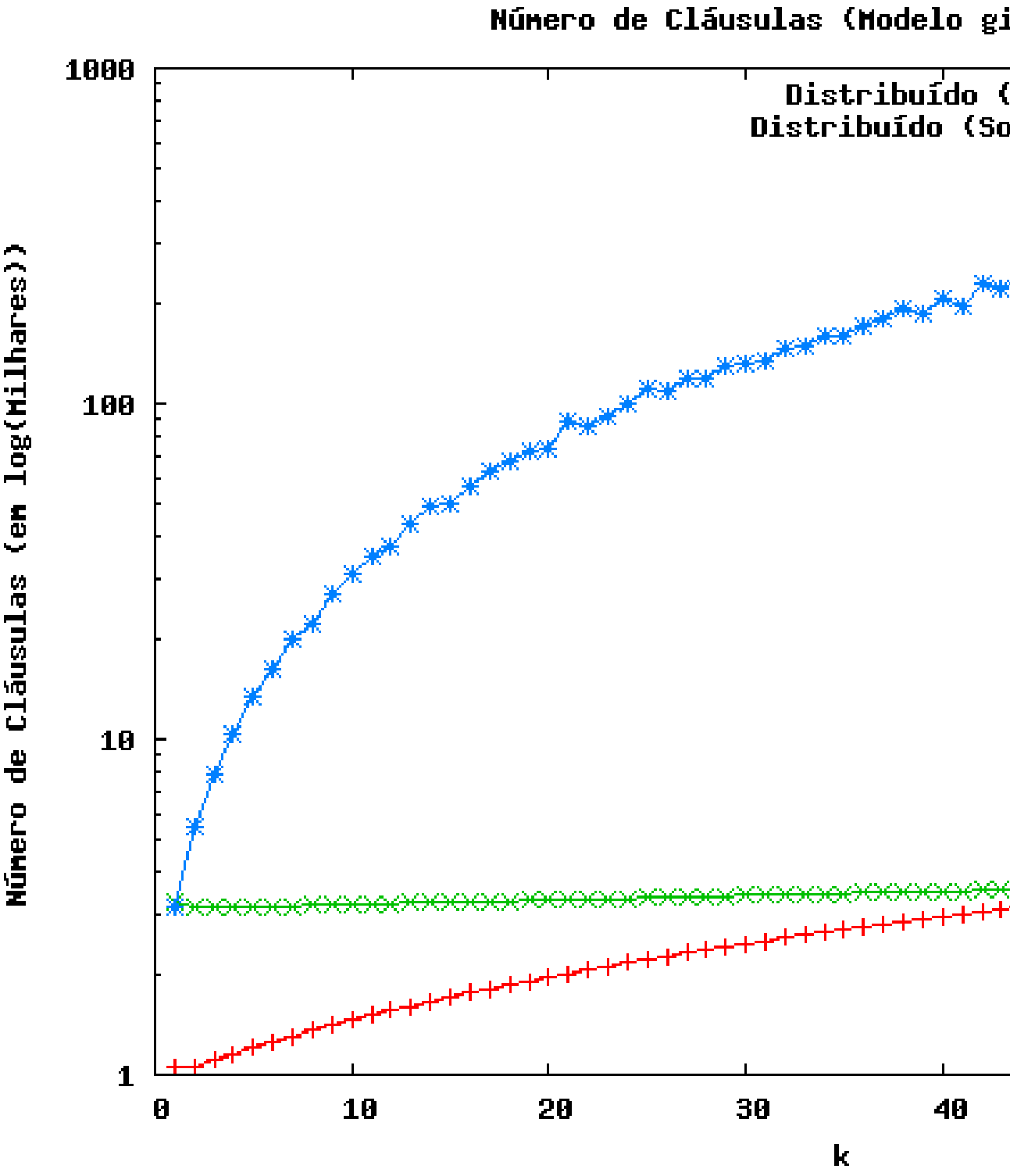


Figura 6.9: Número de Cláusulas do Modelo gigamax_ltl.smv - Escala Logarítmica

### 6.3.6 Modelo 5: dme5.smv

Uma versão assíncrona do algoritmo de exclusão mútua com 5 células. Propriedade LTL verificada: F (cell4_r).

#### Tempos de Resposta

O gráfico 6.10 mostra os tempos de respostas obtidos pelos dois métodos em uma escala logarítmica. Neste exemplo a diferença de tempos de resposta não foi tão grande quanto a dos outros modelos, mas para um k grande conseguimos até uma ordem de magnitude.

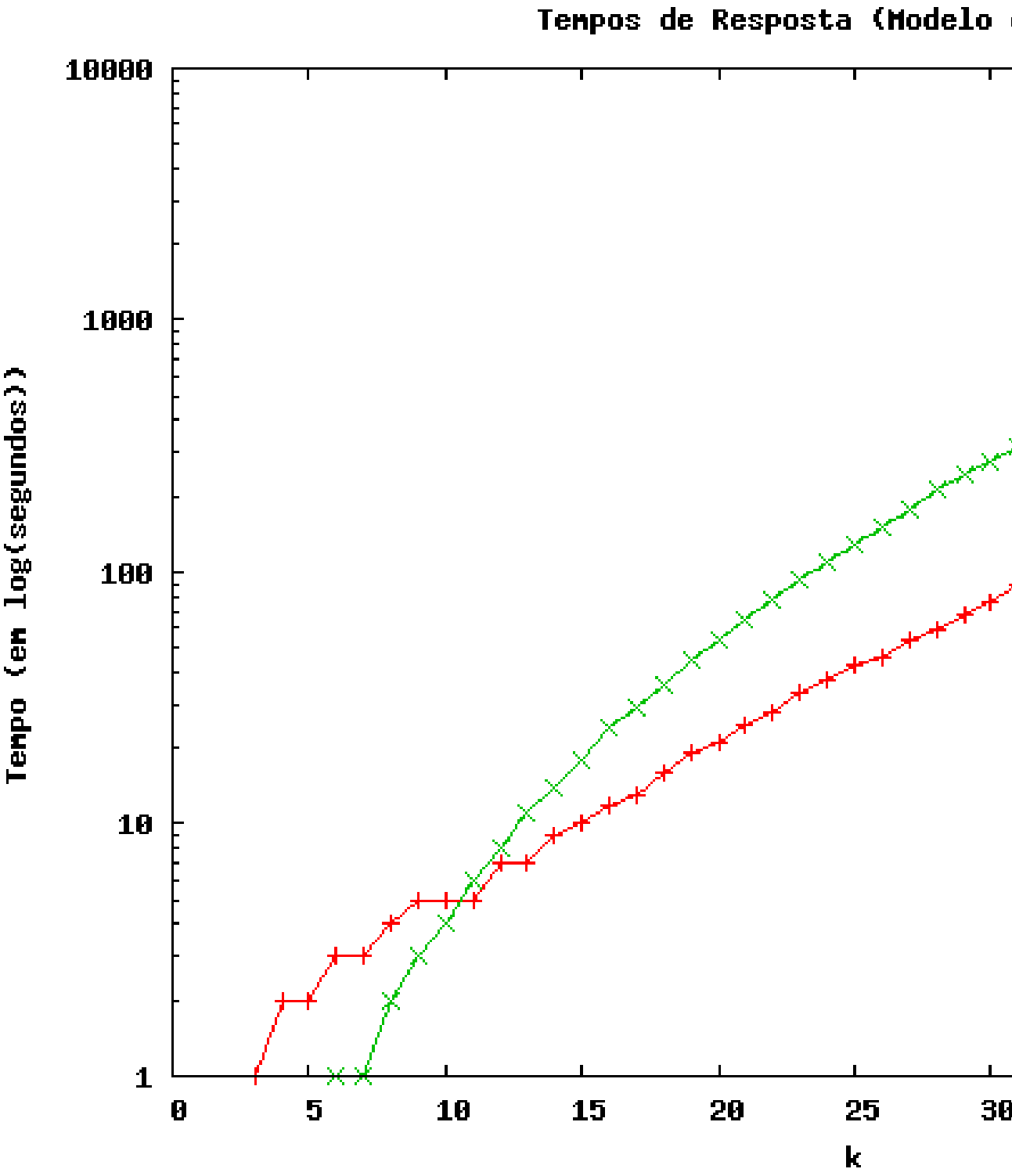


Figura 6.10: Tempos de Resposta do Modelo dme5.smv - Escala Logarítmica

### Utilização de Memória

O gráfico 6.11 mostra a utilização de memória no método monolítico e no método distribuído em uma escala logarítmica.

Neste gráfico podemos verificar que em termos de utilização de memória o solucionador primário e o método monolítico se equipararam. Este é um exemplo em que as cláusulas replicadas não se apresentaram tão eficientes quanto nos outros exemplos. A maior utilização de memória pelos solucionadores secundários indicam que ouve mais chamadas ao ZChaff pelo solucionador primário. Lembre-se que toda vez que um solucionador primário recebe uma cláusula conflito o ZChaff é chamado.

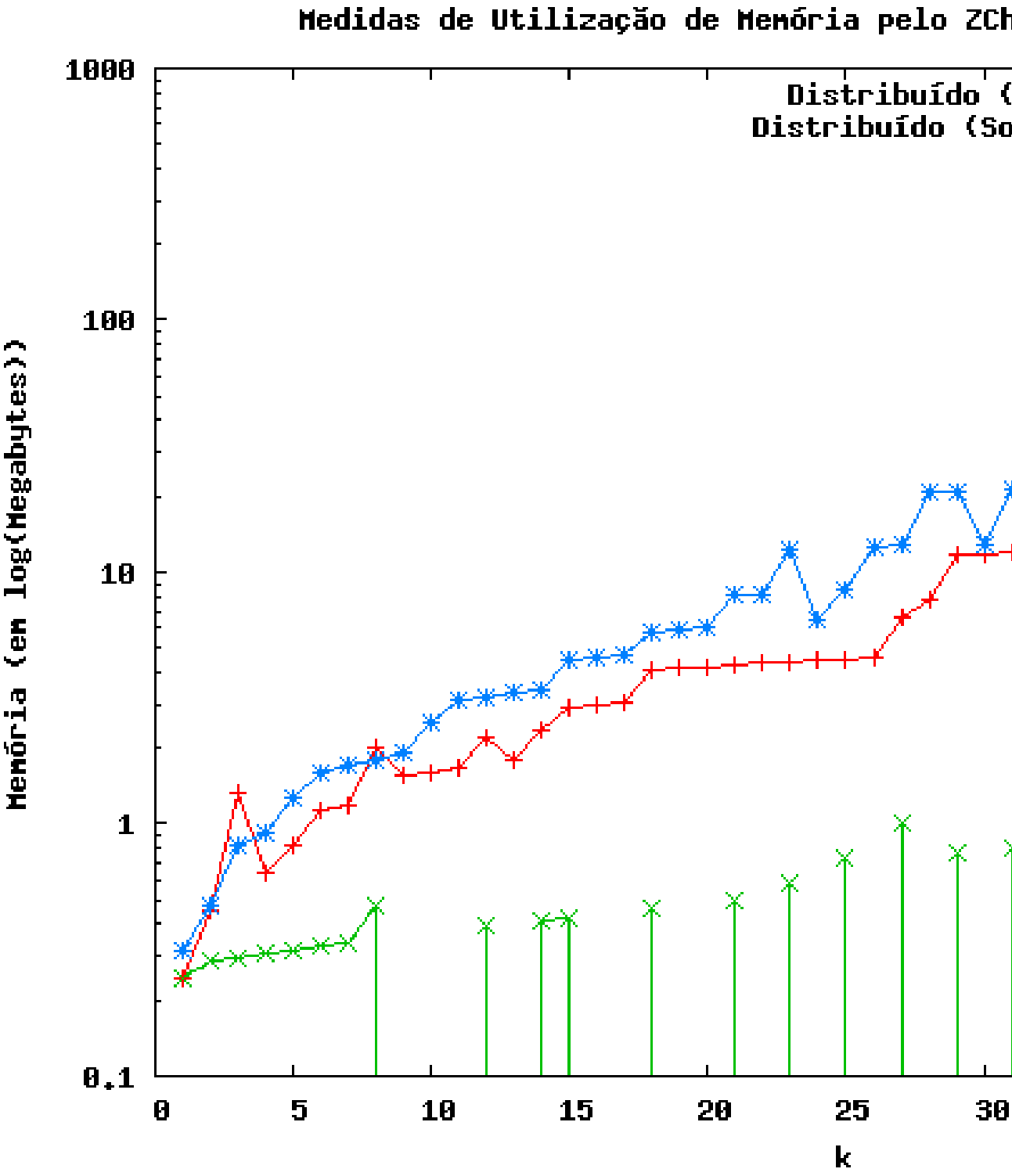


Figura 6.11: Utilização de Memória do Modelo dme5.smv - Escala Logarítmica

### Número de Cláusulas

O gráfico 6.12 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído em uma escala logarítmica. O método distribuído mostrou superioridade nesta métrica. A maior utilização de memória se deu ao maior número de execuções do ZChaff. Comprovando o que alegamos na seção anterior.

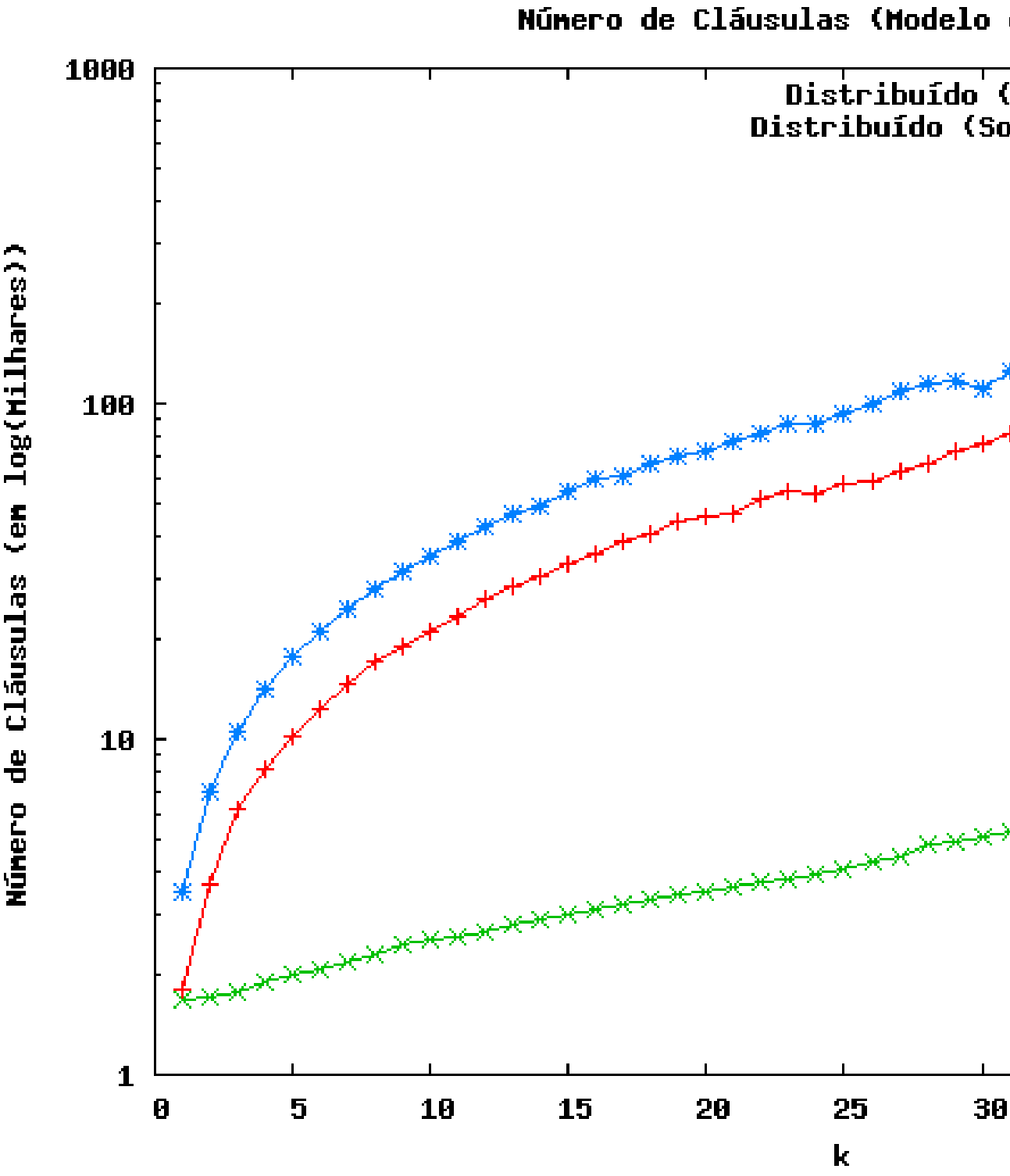


Figura 6.12: Número de Cláusulas do Modelo dme5.smv - Escala Logarítmica

### 6.3.7 Modelo 6: dme8.smv

Uma versão assíncrona do algoritmo de exclusão mútua com 8 células. Propriedade LTL verificada: F (cell7_r).

#### Tempos de Resposta

O gráfico 6.13 mostra os tempos de respostas obtidos pelos dois métodos em uma escala logarítmica. O curioso neste exemplo é que até um k aproximadamente igual a 10 os tempos do monolítico foram superiores. Este comportamento foi notado em todos os modelos com um número de variáveis superior a 200. Existe uma justificativa: inicialmente não existem as cláusulas conflito que possam ser replicadas. Como a transição tem mais variáveis isso significa que mais cláusulas conflito devem ser geradas e portanto mais comunicação deve acontecer.

Na medida que k aumenta o número de cláusulas replicada aumenta e a comunicação passa a ser menos necessária.

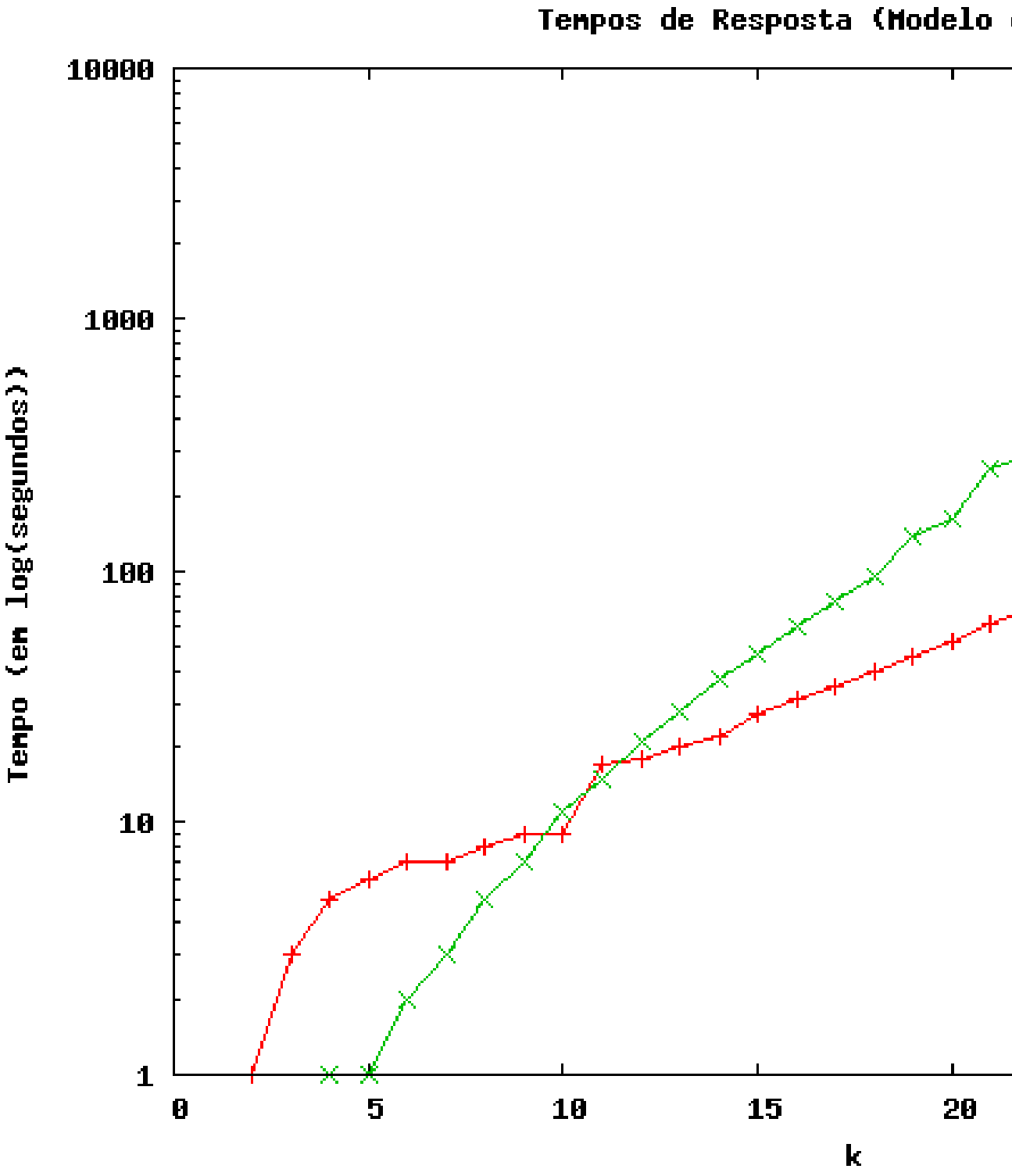


Figura 6.13: Tempos de Resposta do Modelo dme8.smv - Escala Logarítmica

### Utilização de Memória

O gráfico 6.14 mostra a utilização de memória no método monolítico e no método distribuído em uma escala logarítmica.

Note que como foi explicado na seção anterior, nos primeiros valores de k o solucionador secundário utilizou alguma memória, indicando sua atividade e portanto indicando comunicação.

O monolítico para um k grande utiliza uma quantidade de memória mais que duas ordens de magnitude do que o solucionador primário.

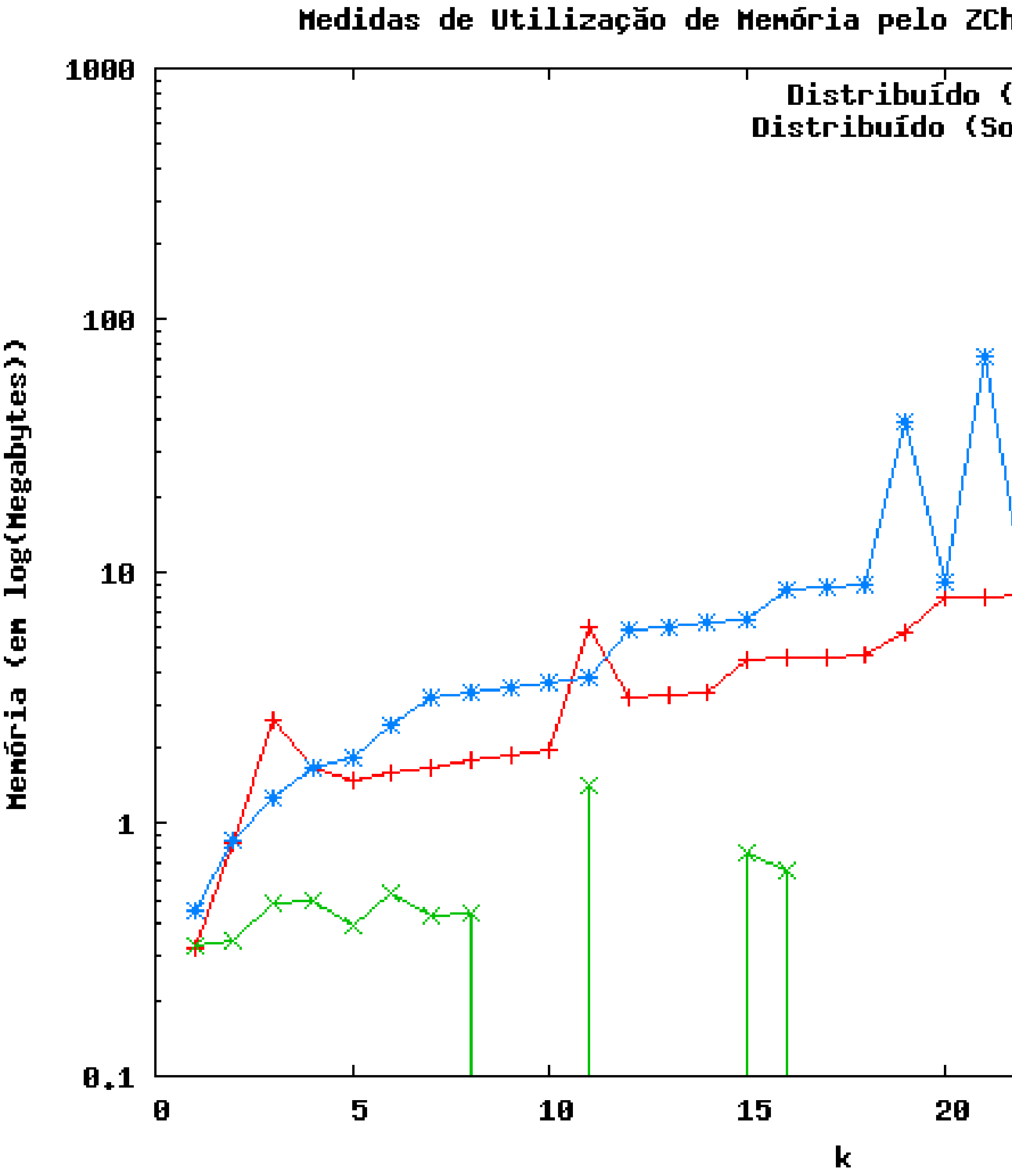


Figura 6.14: Utilização de Memória do Modelo dme8.smv - Escala Logarítmica

### Número de Cláusulas

O gráfico 6.15 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído em uma escala logarítmica. O método distribuído mostrou superioridade nesta métrica. Para um k grande obtivemos um ganho de até uma ordem de magnitude.

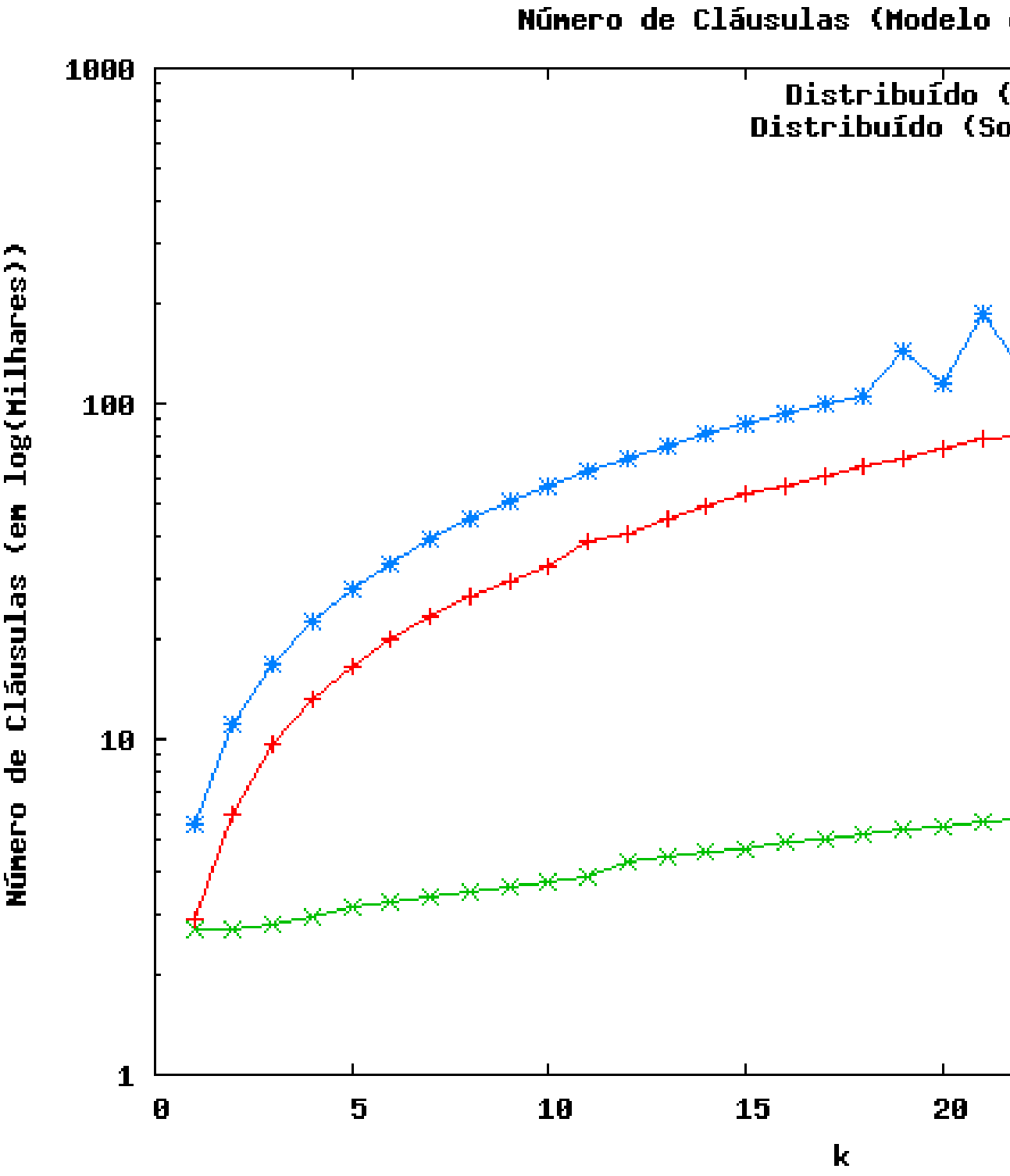


Figura 6.15: Número de Cláusulas do Modelo dme8.smv - Escala Logarítmica

### 6.3.8 Modelo 7: dme10.smv

Uma versão assíncrona do algoritmo de exclusão mútua com 10 células.
Propriedade LTL verificada: F (cell10_r).

#### Tempos de Resposta

O gráfico 6.16 mostra os tempos de respostas obtidos pelos dois métodos em uma escala logarítmica.

Notamos o mesmo comportamento que o modelo dme8.smv. Desta vez o monolítico foi superior até um k igual a 12. O modelo é maior e portanto mais cláusulas conflito tiveram que ser geradas.

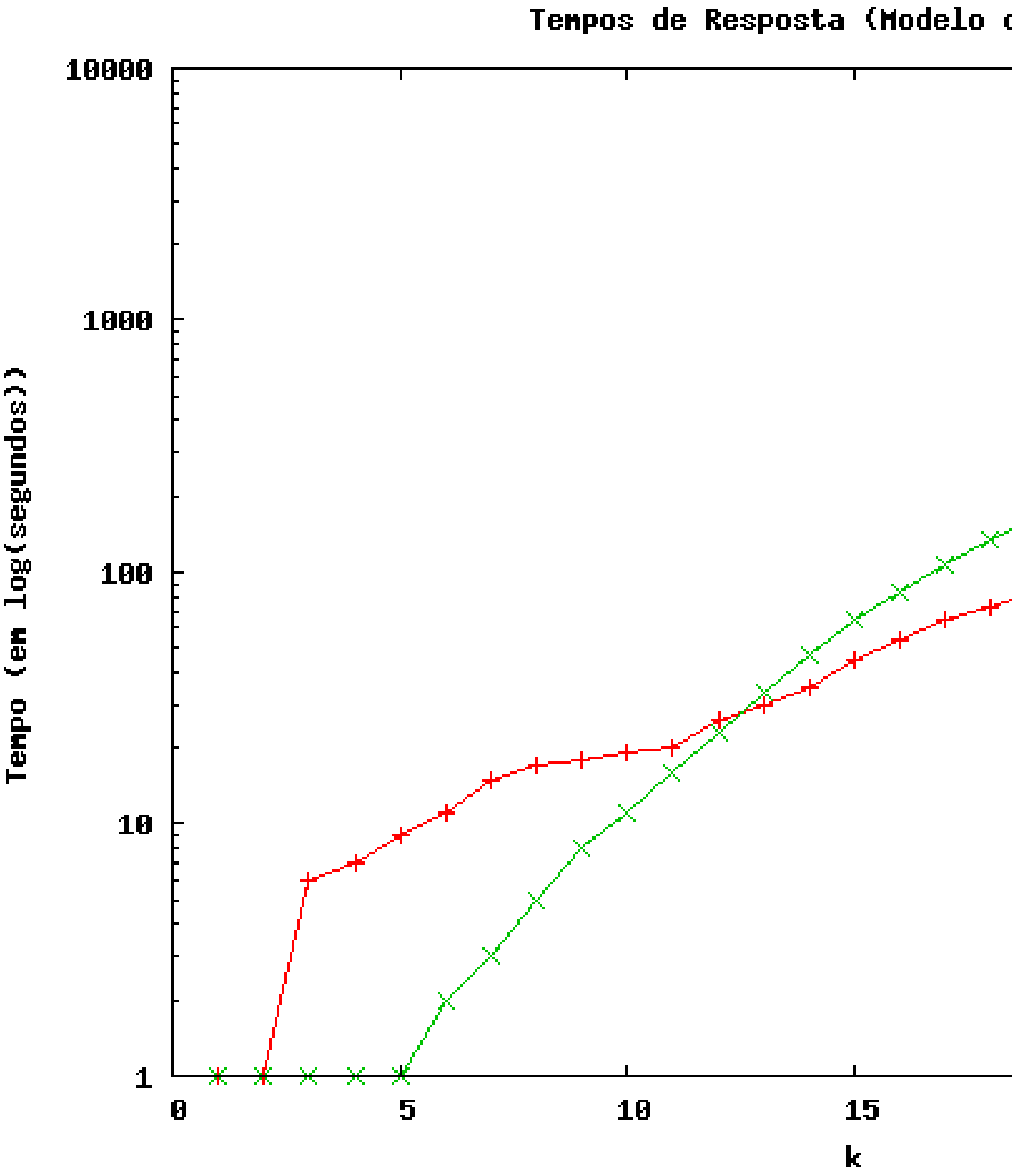


Figura 6.16: Tempos de Resposta do Modelo dme10.smv - Escala Logarítmica

### Utilização de Memória

O gráfico 6.17 mostra a utilização de memória no método monolítico e no método distribuído em uma escala logarítmica. O comportamento foi semelhante ao do modelo dme8.smv.

O monolítico para um k grande utiliza uma quantidade de memória mais que uma ordem de magnitude do que o solucionador primário.

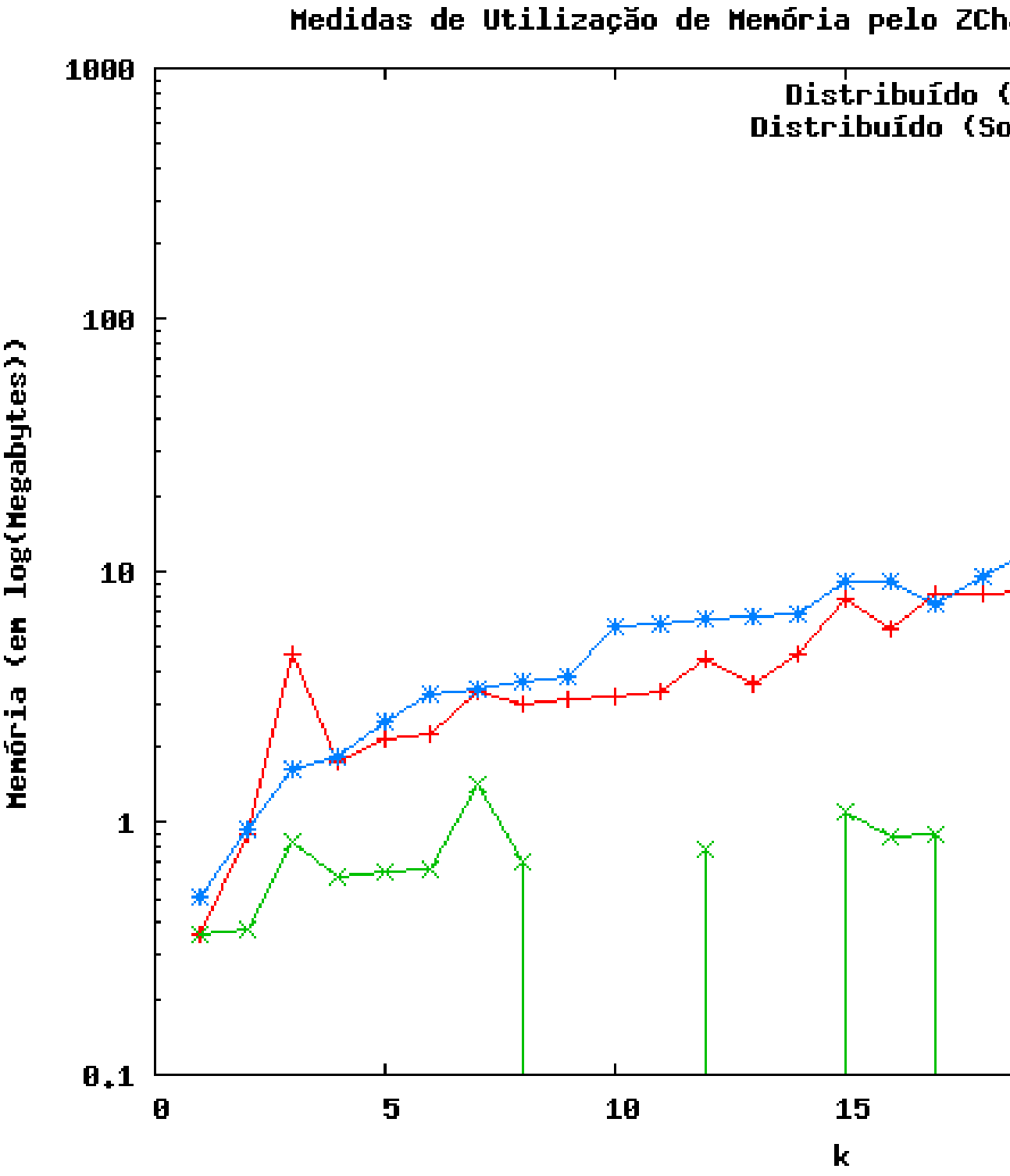


Figura 6.17: Utilização de Memória do Modelo dme10.smv - Escala Logarítmica

### Número de Cláusulas

O gráfico 6.18 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído em uma escala logarítmica. O método distribuído mostrou superioridade nesta métrica. Para um k grande obtivemos um ganho de até uma ordem de magnitude.

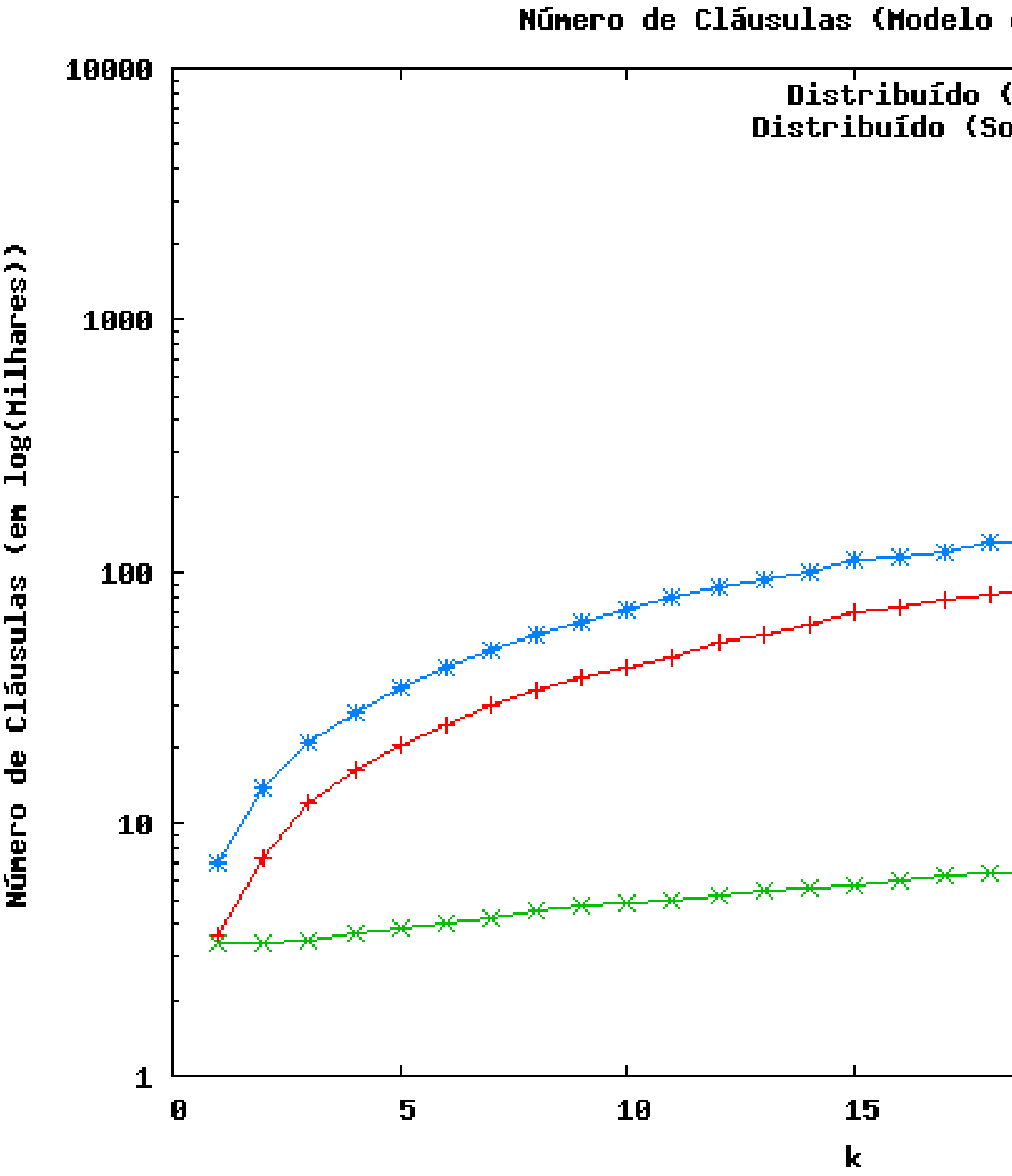


Figura 6.18: Número de Cláusulas do Modelo dme10.smv - Escala Logarítmica

### 6.3.9 Modelo 8: dme20.smv

Uma versão assíncrona do algoritmo de exclusão mútua com 20 células. Propriedade LTL verificada: F (cell19_r).

#### Tempos de Resposta

O gráfico 6.19 mostra os tempos de respostas obtidos pelos dois métodos. Este modelo nos permitiu abandonar a escala logarítmica para que possamos entender com mais precisão o ganho de desempenho.

Para um k igual a 20 gasta-se mais de 7.000 segundos no método monolítico e menos de 1.000 no método distribuído. Um ganho de uma ordem de magnitude.

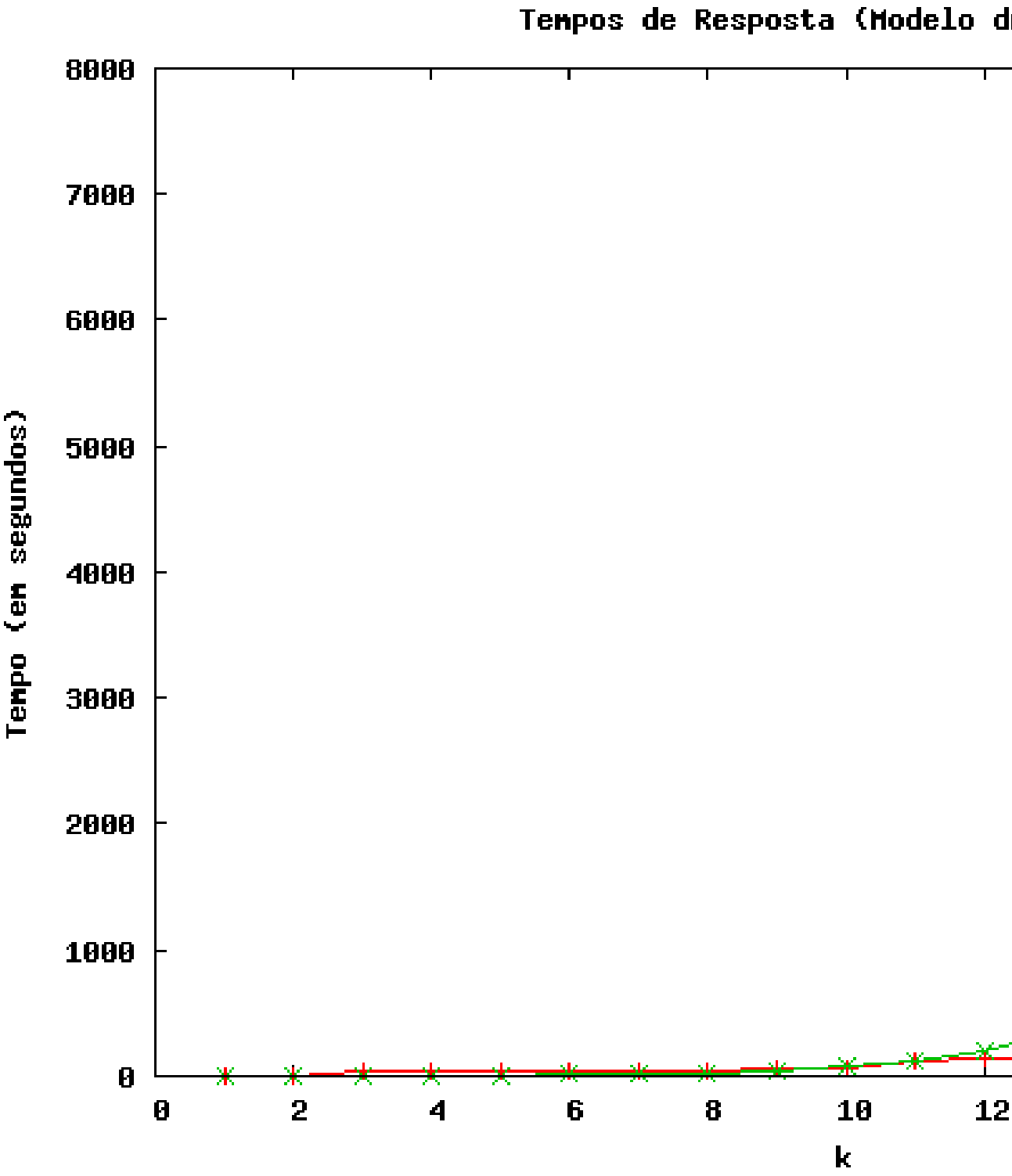


Figura 6.19: Tempos de Resposta do Modelo dme20.smv

### Utilização de Memória

O gráfico 6.20 mostra a utilização de memória no método monolítico e no método distribuído em uma escala logarítmica.

Para um k igual a 20 usamos mais de 400 megabytes de memória para resolver com o método monolítico, para o distribuído usamos menos de 50 megabytes.

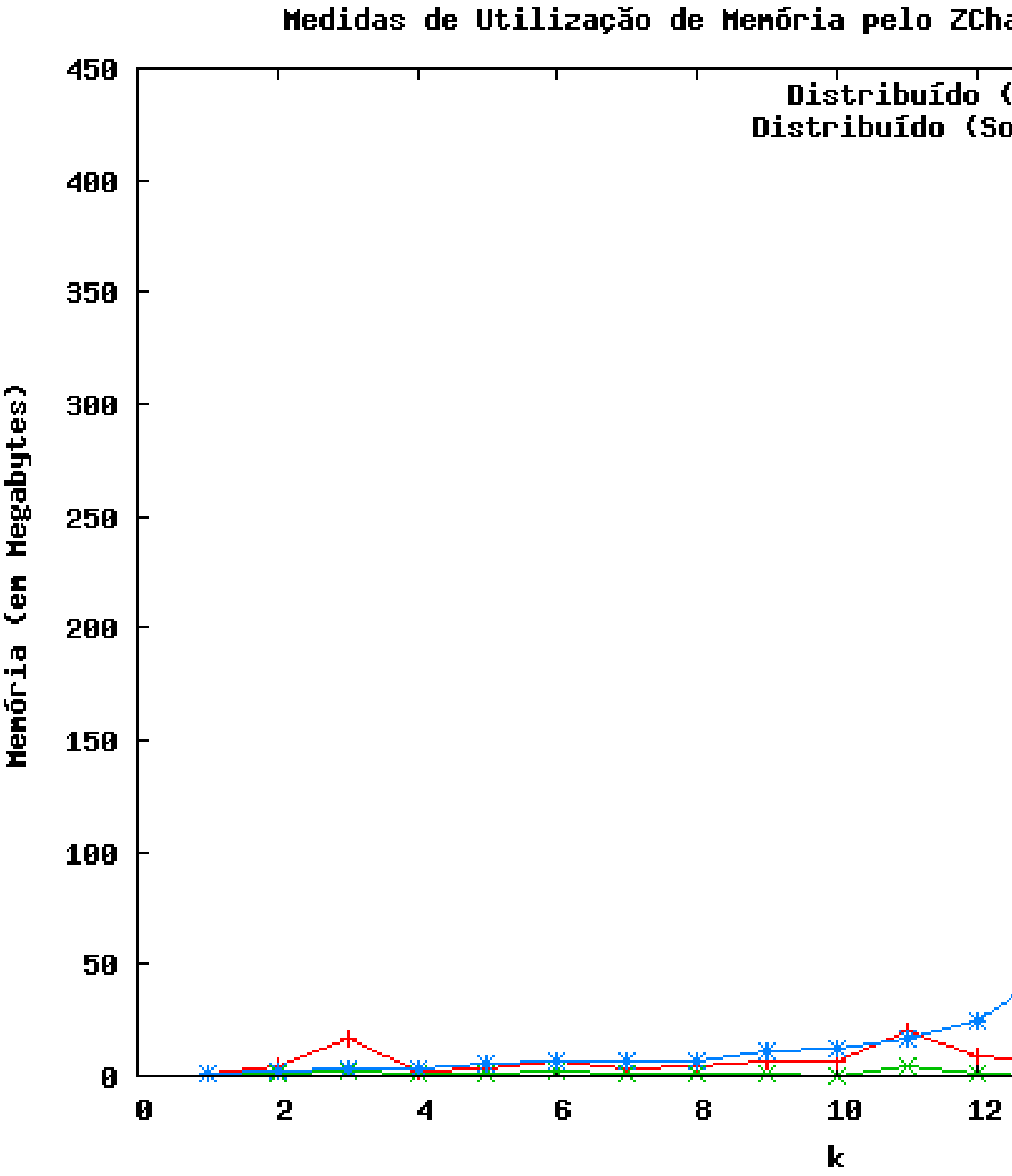


Figura 6.20: Utilização de Memória do Modelo dme20.smv

#### Número de Cláusulas

O gráfico 6.21 mostra o número de cláusulas utilizadas pelo método monolítico e pelo método distribuído em uma escala logarítmica.

Para um k igual a 20 usamos mais de 1.400.000 cláusulas para o monolítico, usamos menos de 200.000 para o distribuído.

### 6.3.10 Escalabilidade

A escalabilidade representou um ponto fraco para o algoritmo proposto. Em todas as medidas tomada o aumento no número de computadores não apresentou aumento de desempenho da aplicação.

Após estudarmos os números podemos notar que o algoritmo distribuído proposto converge para um problema monolítico. Pelo desenho do mesmo é natural que isso ocorra, já que na medida que o tempo passa, os solucionadores secundários repassam informações sobre seus problemas para o solucionador primário. Para os problemas em que atingimos uma maior profundidade tínhamos uma convergência, o solucionador primário passava a representar um problema quase que equivalente ao problema completo.

A vantagem ficou por conta da forma como esse problema equivalente foi gerado. Como as cláusulas conflito replicadas nunca ultrapassavam o tamanho de uma transição trabalhávamos sempre com cláusulas conflito pequenas em relação ao tamanho total do modelo. Cláusulas pequenas favorecem o uso de métodos DPLL como o ZChaff para resolver um problema SAT.

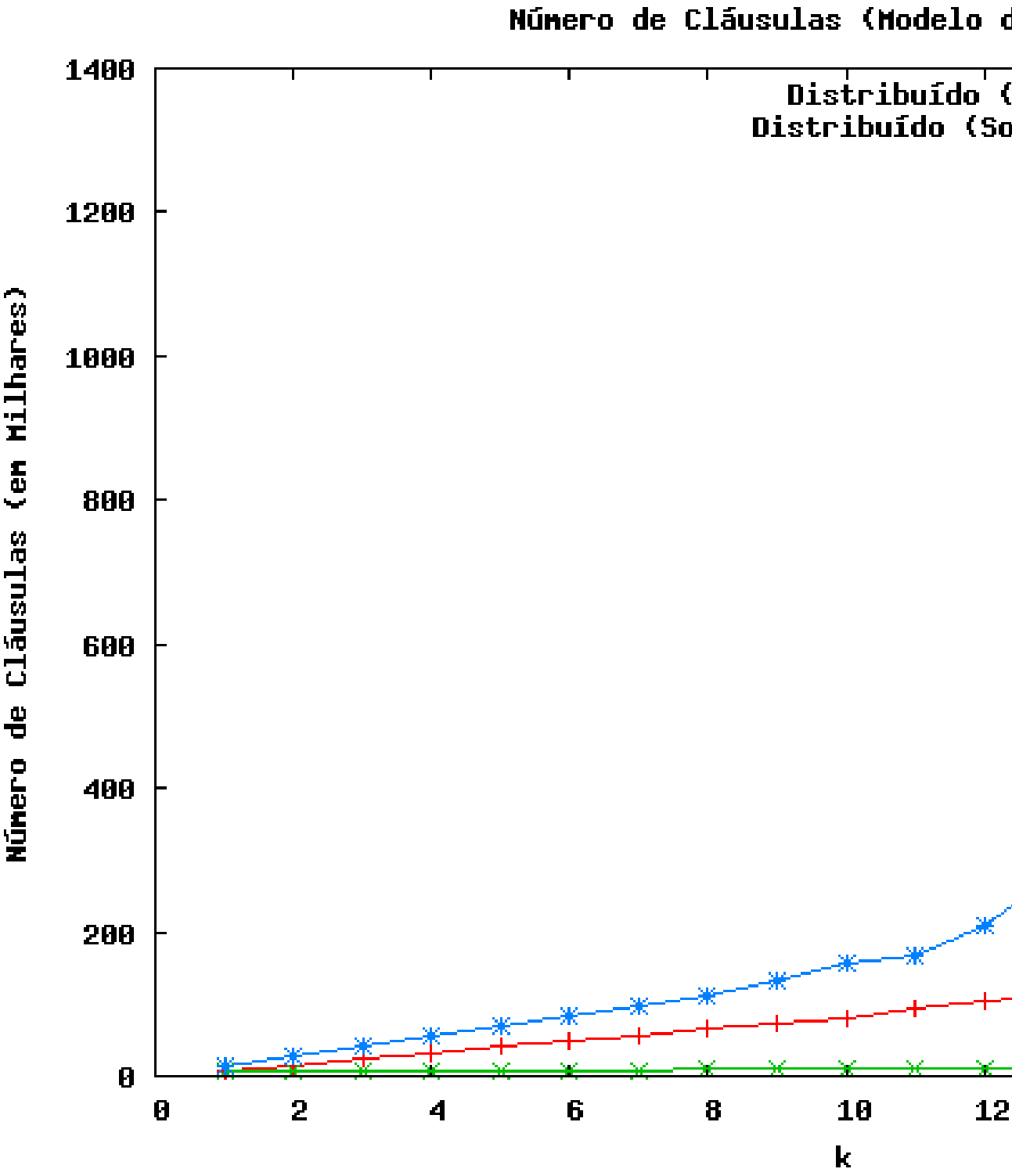


Figura 6.21: Número de Cláusulas do Modelo dme20.smv

# Capítulo 7

# Conclusões e Trabalhos Futuros

## 7.1 Análise dos Resultados

Podemos resumir os resultados da seguinte forma:

- O algoritmo proposto apresenta ganhos por exigir uma menor quantidade de memória do que os algoritmos de verificação originais baseados em métodos SAT monolíticos.
- O algoritmo proposto consegue explorar mais ciclos que os algoritmos de verificação originais baseados em métodos SAT monolíticos.
- O algoritmo proposto gera problemas SAT com o tamanho médio das cláusulas inferior ao das cláusulas do BMC monolítico, o que torna a solução do problema SAT mais rápida.
- O algoritmo proposto não é escalável.

Infelizmente o algoritmo distribuído proposto não escala de maneira significativa. A escalabilidade foi prejudica ao adotarmos técnicas que permitiam uma exploração mais inteligente do espaço de estados. Essa exploração nos ofereceu ganhos significativos em alguns casos de até ordens de magnitude.

À primeira vista, o título do trabalho não está coerente com seu conteúdo. Isto não é verdade pelo seguinte motivo. O particionamento proposto sugere o uso de mais de um processador para resolver o problema, o que nos leva de volta à computação distribuída. A diferença é que esta tecnologia passa a ter um papel de coadjuvante ao invés de ser a protagonista. O papel de protagonista passa a ser ocupado pela replicação de cláusulas conflito geradas à partir de uma fórmula que representa apenas uma transição. Além de serem

replicáveis essa cláusulas podem ser usadas para qualquer ciclo do modelo verificado.

Outro ponto positivo, foi que construímos esse algoritmo utilizando apenas componentes comuns a maioria das redes de computadores existentes, sejam no meio acadêmico ou na iniciativa privada. Nosso algoritmo não exige nenhum componente especial, nem de software ou de hardware. Qualquer interessado que tiver a sua disposição uma rede de computadores pode usufruir dos ganhos apresentados neste trabalho.

## 7.2 Trabalhos Futuros

Existe uma série de trabalhos futuros que podem acrescentar muitas melhorias ao algoritmo proposto. Nesta seção destacaremos dois deles.

O primeiro trabalho seria conseguir particionar o que chamamos de partição primária. A partição primária é composta dos estados iniciais e das propriedades. A sugestão seria particionar com foco nas propriedades. Para explicar melhor vamos voltar ao exemplo usado para explicar o particionamento no Capítulo 5.

$$\begin{aligned}
[\![M,P]\!]_3 &:= I(s_0) \wedge [\![P]\!]_6^0 \wedge \left(\textstyle\bigwedge_{i=0}^{3-1} T(s_i, s_{i+1})\right) \\
[\![M,P]\!]_3 &:= I(s_0) \wedge [\![P]\!]_6^0 \wedge T(s_0,s_1) \wedge T(s_1,s_2) \wedge T(s_2,s_3) \\
[\![M,P]\!]_6 &:= \underbrace{I(s_0) \wedge [\![P]\!]_6^0}_{P1} \wedge \underbrace{T(s_0,s_1) \wedge T(s_1,s_2)}_{P2} \wedge \underbrace{T(s_2,s_3)}_{P3}
\end{aligned}$$

Se quiséssemos verificar a propriedade $F(a \wedge b)$, que diz que em algum estado do futuro teremos $a = 1$ e $b = 1$, teríamos então a seguinte fórmula:

$$\begin{aligned}
[\![M,P]\!]_6 &:= \underbrace{I(s_0) \wedge [(a_0 \wedge b_0) \vee (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)]}_{P1} \\
&\quad \wedge \underbrace{T(s_0,s_1) \wedge T(s_1,s_2)}_{P2} \wedge \underbrace{T(s_2,s_3)}_{P3}
\end{aligned}$$

Note como a fórmula foi dividida em quatro partições $P1$, $P2$ e $P3$. Poderíamos dividir a partição $P1$ se separássemos as propriedades da seguinte forma:

$$\begin{aligned}
[\![M,P]\!]_6 &:= [\underbrace{I(s_0) \wedge (a_0 \wedge b_0)}_{P1} \vee \underbrace{I(s_0) \wedge (a_1 \wedge b_1)}_{P2} \vee \underbrace{I(s_0) \wedge (a_2 \wedge b_2)}_{P3} \\
&\quad \vee \underbrace{I(s_0) \wedge (a_3 \wedge b_3)}_{P4}] \wedge \underbrace{T(s_0,s_1) \wedge T(s_1,s_2)}_{P5} \wedge \underbrace{T(s_2,s_3)}_{P6}
\end{aligned}$$

Teríamos então as partições $P1$, $P2$, $P3$, $P4$, $P5$ e $P6$. Pelo desenho do algoritmo poderíamos ter vários solucionadores primários com as partições de $P1$ a $P4$. As modificações no algoritmo seriam pequenas e os ganhos pelo menos em termos de memória seriam significativos.

O outro trabalho visa tentar suprir uma deficiência do algoritmo que é o balanceamento de carga. O algoritmo não prevê isso e deixa a cargo da MPI realizá-lo. A MPI faz o balanceamento de carga através do instanciamento de processos nos nodos do cluster seguindo uma política *round-robin*. Não é a maneira mais eficiente. A sugestão neste caso é executar o algoritmo com a MPI sobre um cluster Mosix [78, 79, 80, 81].

O Mosix é uma extensão de kernel tipo-Unix para *clustering* com sistema de imagem única (*Single System Image* - SSI). SSI é a propriedade de um sistema que esconde a natureza heterogênea e distribuída dos recursos disponíveis e os apresenta ao usuários e aplicações como um recurso único unificado.

O Mosix consiste de algoritmos adaptativos de compartilhamento de recursos. Esses algoritmos permitem que os nodos do cluster que usam um kernel compilado com as extensões Mosix trabalhem em cooperação. Variações na utilização de recursos do cluster são respondidas dinamicamente pelos algoritmos. O dinamismo é implementado através da migração preemptiva e transparente de processos, que também é responsável pelo balanceamento dinâmico de carga e previne paginação excessiva na memória virtual.

O interessante do Mosix é que basta você executar um programa que o cluster decide onde os processos correspondentes devem ser executados. Os inventores do Mosix chamaram isso de "fork-and-forget". O objetivo do Mosix é melhorar a performance do cluster como um todo e criar um ambiente conveniente para a execução de aplicações paralelas e seqüenciais. O Mosix foi projetado para executar em clusters formados por computadores baseados na arquitetura x86 conectados por uma rede LAN padrão, como a FastEthernet.

# Referências Bibliográficas


[1] Glenford J. Myers. *The Art of Software Testing.* Wiley - Interscience, New York, 1979.

[2] Edmund M. Clarke and Orna Grumberg and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[3] 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. Disponível em: `http://citeseer.nj.nec.com/burch90symbolic.html`.

[4] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach To The State Explosion Problem.* PhD thesis, 1992.

[5] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. Disponível em: `http://citeseer.nj.nec.com/bryant86graphbased.html`.

[6] Armin Biere and Alessandro Cimatti and Edmund Clarke and Yunshan Zhu. Symbolic Model Checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999. Disponível em: `http://citeseer.nj.nec.com/article/biere99symbolic.html`.

[7] Armin Biere and Alessandro Cimatti and Edmund M. Clarke and M. Fujita and Y. Zhu. Symbolic Model Checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999. Disponível em: `http://citeseer.nj.nec.com/biere99symbolic.html`.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[9] I. Gent and T. Walsh. The search for Satisfaction. Technical report, 1999. Disponível em: `http://citeseer.nj.nec.com/gent99search.html`.

[10] M. Davis and H. Putman. A Computation Procedure for Quantification Theory. *Journal of ACM*, 7:201–215, 1960.

[11] S. A. Cook. The Complexity of Theorem Proving Procedures. *ACM Symposium on Theory of Computing*, 1971.

[12] J. Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. Technical report, 1996. Disponível em: `ftp://ftp.eecs.umich.edu/techreports/cse/1996/CSE-TR-292-96.ps.Z`.

[13] Marques-Silva and Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEETC: IEEE Transactions on Computers*, 48, 1999.

[14] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, 1995. Disponível em: `ftp://ftp.cis.upenn.edu/pub/freeman/thesis.ps.gz`.

[15] David McAllester and Bart Selman and Henry Kautz. Evidence for Invariants in Local Search. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 321–326. AAAI Press, 1997.

[16] Martin Davis and George Logemann and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[17] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, 2002. Disponível em: `http://www.ece.cmu.edu/~mvelev/goldberg_novikov_date02.pdf`.

[18] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997. Disponível em: `http://citeseer.nj.nec.com/zhang97sato.html`.

[19] Matthew W. Moskewicz and Conor F. Madigan and Ying Zhao and Lintao Zhang and Sharad Malik. Chaff: Engineering an Efficient

SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001. Disponível em: `http://citeseer.nj.nec.com/moskewicz01chaff.html`.

[20] Jun Gu and Paul W. Purdom and John Franco and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: a Survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997. Disponível em: `http://citeseer.nj.nec.com/56722.html`.

[21] Lintao Zhang and Sharad Malik. The Quest for Efficient Boolean Satisfiability Solvers. *Lecture Notes in Computer Science*, 2404, 2002. Disponível em: `http://link.springer.de/link/service/series/0558/bibs/2404/24040017.htm%`.

[22] David A. Plaisted and Steven Greenbaum. A Structure Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[23] João Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Progress in Artificial Intelligence (EPIA-99)*, volume 1695 of *LNAI*, pages 62–74. Springer, 1999.

[24] J. N. Hooker and V. Vinay. Branching Rules for Satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[25] M. Buro and H. Kleine Buning. Report on a SAT Competition. Technical report, 1992.

[26] Robert G. Jeroslow and Jinchang Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[27] P. Prosser. Hybrid algorihtms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[28] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 203–208. AAAI Press, 1997.

[29] L. Zhang and C. F. Madigan and M. W. Moskewicz and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001. Disponível em: `http://www.ee.princeton.edu/~chaff/iccad2001_final.pdf`.

[30] Laurent Simon and Daniel Le Berre and Edward A. Hirsch. The SAT2002 Competition. Disponível em: `http://citeseer.ist.psu.edu/simon02sat.html`.

[31] Laurent Simon and Philippe Chatalic. SATEx: a Web-based Framework for SAT Experimentation. 2001. Disponível em: `http://citeseer.ist.psu.edu/simon01satex.html`.

[32] E. Goldberg and Y. Novikov. The BerkMin's Web Page. Sítio: `http://eigold.tripod.com/BerkMin.html`.

[33] Niklas Eén and Niklas Sörensson. An Extensible SAT Solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, 2003. Disponível em: `http://www.cs.chalmers.se/~een/Satzoo/`.

[34] Niklas Eén. SatZoo. Sítio: `http://www.cs.chalmers.se/~een/Satzoo/`.

[35] Alexander Nadel. The Jerusat Solver. Sítio: `http://www.geocities.com/alikn78`.

[36] Niklas Sörensson. SatNik. Sítio: `http://www.math.chalmers.se/~nik/`.

[37] SAT Research Group, Princeton University. zChaff. Sítio: `http://ee.princeton.edu/~chaff/zchaff.php`.

[38] E.M. Clarke and O. Grumberg and H. Hiraishi and S. Jha and D.E. Long and K.L. McMillan and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew and L. Claesen and R. Camposano, editor, *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. Disponível em: `http://citeseer.nj.nec.com/article/clarke92verification.html`.

[39] Edmund M. Clarke and Armin Biere and Richard Raimi and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001. Disponível em: `http://citeseer.nj.nec.com/clarke01bounded.html`.

[40] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987. Disponível em: `http://citeseer.nj.nec.com/alpern86recognizing.html`.

[41] G. Stalmarck and Saflund M. Modeling and Verifying Systems and Software in Propositional Logic. *(SAFECOMP'90): Safety Security and Reliability Related Computers for the 1990s*, page 31, 1990.

[42] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen & Co. Ltd, London, 2 edition, 1974.

[43] A. Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[44] Wojciech Penczek and Bozena Wozna and Andrzej Zbrzezny. Bounded Model Checking for the Universal Fragment of CTL. *Fundam. Inf.*, 51(1):135–156, 2002.

[45] E.M. Clarke and O. Grumberg and K. Hamaguchi. Another Look at LTL Model Checking. In David L. Dill, editor, *Proceedings of The Sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 415–427, Standford, California, USA, 1994. Springer-Verlag. Disponível em: `http://citeseer.nj.nec.com/clarke94another.html`.

[46] A. P. Sistla. *Theoretical Issues in the Design of Distributed and Concurrent Systems*. PhD thesis, Harvard University, Cambridge, MA, 1983.

[47] A.P. Sistla and E. Clarke. The Complexity of Propositional Temporal Logic. In *14th ACM Symposium on Theory of Computing*, pages 159–167, 1982.

[48] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specifications. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, 1985.

[49] Valentin Goranko. Temporal Logics of Computations. Disponível em: `http://citeseer.nj.nec.com/goranko00temporal.html`, 2000.

[50] M. Ben-Ari and Z. Manna and Amir Pnueli. The Temporal Logic of Branching Time. In *Eighth Annual ACM Symposium on Principles of Programming Languages*, volume 20, pages 207–226. ACM, 1981.

[51] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[52] E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In J. W. de Bakker and Jan van Leeuwen, editor, *Automata, Languages and Programming, 7th Colloquium*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181, Noordweijkerhout, The Netherland, 1980. Springer-Verlag.

[53] Leslie Lamport. "Sometimes" is sometimes "not never". In *Proceedings of SIGPLAN-80, 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, Nevada, 1980.

[54] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited. *Journal of the Association for Computing Machinery (ACM)*, 33(1):151–178, 1978.

[55] E. M. Clarke and I. A. Oraghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency.*, pages 428–437, Berlin - Heidelberg - New York, 1989. Springer.

[56] E. M. Clarke and E. A. Emerson and A. P. Sistla. Automatic Verification of Finite state Concurrent Systems Using Temporal Logic Specifications. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM, 1983.

[57] Ofer Shtrichman. Tuning SAT Checkers for Bounded Model Checking. In *Computer Aided Verification*, pages 480–494, 2000. Disponível em: `http://citeseer.ist.psu.edu/shtrichman00tuning.html`.

[58] F. Copty and L. Fix and E. Giunchiglia and G. Kamhi and A. Tacchella and M. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. of CAV*, LNCS. Springer Verlag, 2001. Disponível em: `http://www.mrg.dist.unige.it/~sim/simo/Publications/Data/bmcsat.ps.gz`.

[59] Per Bjesse and Tim Leonard and Abdel Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. *Lecture Notes in Computer Science*, 2102, 2001. Disponível em: `http://link.springer-ny.com/link/service/series/0558/bibs/2102/21020454%.htm`.

[60] G. F. Pfister. *In search of clusters*. Prentice Hall, 1998.

[61] R. Hempel. The MPI Standard for Message Passing. *Lecture Notes in Computer Science*, 797:247–252, 1994. Disponível em: `http://www-unix.mcs.anl.gov/mpi/`.

[62] A. Geist and A. Beguelin and J. Dongarra and W. Jiang and R.Manachek and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachussetts, 1994.

[63] Mark Baker and Rajkumar Buyya. Cluster Computing at a Glance. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 3–47. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 1.

[64] Mark Baker. Cluster Computing White Paper. page 119, 2001. Disponível em: `http://arXiv.org/abs/cs/0004014`.

[65] Ian Foster and Carl Kesselman. Computational Grids. In *The Grid: Blueprint for a New Computing Infrastructure*, pages 15–51. Morgan Kaufmann, San Francisco, CA, 1999. Chap. 2.

[66] Ian Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lecture Notes in Computer Science*, 2150, 2001. Disponível em: `http://www.globus.org/research/papers/anatomy.pdf`.

[67] Linus Torvalds. The Linux Edge. *Communications of the Association for Computing Machinery*, 42(4):38–39, 1999. Disponível em `http://www.acm.org:80/pubs/citations/journals/cacm/1999-42-4/p38-torval%ds/`.

[68] Arron Rouse. AMD design will kill competition. *The Inquirer*, 2004. Disponível em: `http://www.theinquirer.net/Default.aspx?article=14038`.

[69] Malay Ganai and Aarti Gupta and Zijiang Yang and Pranav Ashar. Efficient Distributed SAT and SAT-Based Distributed Bounded Model Checking. *Lecture Notes in Computer Science*, 2860:334–347, 2003.

[70] Alessandro Cimatti and Enrico Giunchiglia and Marco Pistore and Marco Roveri and Roberto Sebastiani and Armando Tacchella. Integrating BDD-based and SAT-based Symbolic Model Checking. In *In Proceeding of 4th International Workshop on Frontiers of Combining Systems (FroCoS'2002)*, 2002. Disponível em: `http://nusmv.irst.itc.it/NuSMV/papers/frocos02/ps/frocos02.pdf`.

[71] Ying Zhao and Sharad Malik and Matthew W. Moskewicz and Conor F. Madigan. Accelerating boolean satisfiability through application specific processing. In *ISSS*, pages 244–249, 2001.

[72] Alessandro Cimatti and Edmund Clarke and Enrico Giunchiglia and Fausto Giunchiglia and Marco Pistore and Marco Roveri and Roberto Sebastiani and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *In Proceeding of International Conference on Computer-Aided Verification (CAV 2002)*, 2002. Disponível em: `http://nusmv.irst.itc.it/NuSMV/papers/cav02/ps/cav02.pdf`.

[73] Daniel Le Berre and Laurent Simon. SAT Competition 2004. Sítio: `http://www.satlive.org/SATCompetition/2004/`.

[74] Greg Burns and Raja Daoud and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994. Disponível em: `http://www.lam-mpi.org/download/files/lam-papers.tar.gz`.

[75] ANL/MSU. MPICH - A Portable Implementation of MPI. Sítio: `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[76] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[77] LAM Team at Indiana University. LAM/MPI Parallel Computing. Sítio: `http://www.lam-mpi.org/`.

[78] Amnon Barak and Shai Guday and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*, volume 672. 1993.

[79] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, 1998. Disponível em: `http://citeseer.nj.nec.com/barak98mosix.html`.

[80] Amnon Barak, Oren La'adan and Amnon Shiloh. Scalable Cluster Computing with Mosix for Linux. In *Proceedings of the 5th Annual Linux Expo*, pages 95–100, Raleigh, N.C., 1999. Disponível em: `http://citeseer.ist.psu.edu/barak99scalable.html`.

[81] Steve McClure and Richard Wheeler. MOSIX: How Linux Clusters Solve Real-World Problems. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference (FREENIX-00)*, pages 49–56, Berkeley, CA, 2000. USENIX Ass.