Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Projeto Orientado em Computação II

Relatório Técnico

PB-Blasting: Uma Nova Abordagem Para Problemas SMT em Bit-Vectors

Alan Cabral Trindade Prado

Orientador

Prof. Dr. Haniel Barbosa

Belo Horizonte

2024

1 Introdução

1.1 Contexto

O problema da satisfatibilidade booleana (SAT), no qual busca-se determinar se existe uma atribuição de valores a variáveis que torna verdadeira uma dada fórmula em lógica proposicional, é crucial para a ciência da computação, pois foi o primeiro a ser mostrado NP-completo, por Stephen A. Cook e, de forma independente, Leonid A. Levin. Essa conclusão implica que qualquer problema que pertence à classe NP – que inclui uma série de problemas relevantes, como o número cromático de um grafo, o problema do caixeiro viajante e o problema da mochila – pode ser reduzido a ele. Ainda mais, essa importância extrapola a teoria: os diversos avanços em resolvedores SAT nos últimos 25 anos ocasionaram um grande interesse por parte de áreas como teoria dos grafos, projeto de *hardware*, verificação formal e pesquisa operacional em converter, total ou parcialmente, certos problemas em problemas de satisfatibilidade.

A aplicação de lógica proposicional é, contudo, em muitos contextos, insuficiente, pois é incapaz de expressar propriedades de objetos ou relações entre eles. Para isso, pode ser utilizada a lógica de primeira ordem (bem como as de ordem superior), em que são adicionados predicados, funções e quantificadores. Nesse sentido, diversas aplicações requerem a habilidade de raciocinar a respeito de alguma *teoria*, que fixa a interpretação de certos predicados e funções. Alguns exemplos de teorias bem estabelecidas são as de aritmética linear, *strings*, *arrays* e *bit-vectors* – ou até mesmo combinações delas. Em geral, resolvedores que checam a satisfatibilidade de fórmulas com respeito a uma teoria implementam procedimentos de decisão específicos para essa teoria acerca da validade de certos modelos construídos de forma independente a partir da fórmula. Em outras palavras, os módulos referentes a cada teoria são independentes do fluxo principal do resolvedor e podem ser instanciados a depender do problema que será resolvido. A esses resolvedores, damos o nome resolvedores SMT e, ao campo de estudo que abrange essa classe de problemas, satisfatibilidade módulo teorias (SMT).

1.2 Bit-vectors

Sistemas computacionais são propensos a erros e, por isso, procedimentos de decisão para análise desses sistemas são altamente desejáveis. Um sistema computacional utiliza sequências de bits para codificar informações como, por exemplo, números. O domínio finito dessas sequências faz com que a semântica de operações como a adição se desvie daquela que é usualmente aplicado a números - na aritmética usual, não ocorrem *overflows*. Esse comportamento particular exige uma teoria dedicada para tratar essas operações corretamente em resolvedores SMT, a teoria de *bit-vectors*. O principal procedimento de decisão para essa teoria, chamado *bit-blasting*, transforma uma fórmula de bit-vectors

em outra proposicional, equisatisfatível, que pode ser processada por um resolvedor SAT. Entre os motivos para isso, estão a robustez e otimização características a esses resolvedores. Além disso, alguns operadores de bit-vectors têm uma correspondência natural com a lógica proposicional, permitindo traduções diretas e eficientes, como os operadores bit a bit de AND, OR, XOR ou Shift.

1.3 Bit-blasting

O procedimento de bit-blasting começa com a substituição de cada átomo da fórmula por uma nova variável booleana. Essa variável booleana atua como um codificador para aquele átomo específico, criando um esqueleto proposicional da fórmula original. O esqueleto proposicional é essencialmente a estrutura da fórmula, com os átomos substituídos por suas respectivas variáveis booleanas. Em seguida, o algoritmo atribui um conjunto de variáveis booleanas a cada termo de bit-vectors da fórmula, de forma recursiva, representando cada bit do termo com uma variável individual. Essas variáveis booleanas formam um novo vetor, com cada uma correspondendo a uma posição específica do bit-vector, traduzindo assim o termo para o espaço booleano.

Para concluir a transformação, o algoritmo gera uma restrição para cada termo e átomo da fórmula, que são combinadas em uma fórmula proposicional unificada. Essa fórmula final, agora completamente composta por expressões booleanas, é usada como entrada em um resolvedor SAT. Em suma, o bit-blasting converte expressões da teoria de bit-vectors em expressões em lógica proposicional, decompondo a fórmula original em expressões booleanas que preservam a satisfatibilidade do problema original. Para um entendimento mais formal a respeito de átomos, termos e bit-blasting, recomenda-se a leitura de Kroening e Strichman [2].

1.4 PB-blasting

Liew et al. publicaram, em 2020, um artigo que aborda o desafio de problemas em bit-vectors que envolvem multiplicação [3]. Eles propõem o uso de resolvedores pseudo-booleanos para esse problema. Resolvedores pseudo-booleanos são ferramentas que resolvem problemas de otimização e decisão representados por desigualdades lineares com variáveis booleanas (isto é, que podem assumir valores 0 ou 1), combinando elementos de programação inteira e lógica proposicional. De forma concisa, são uma generalização de resolvedores SAT. Uma das formas possíveis na qual um resolvedor pode representar a demonstração de insatisfatibilidade de uma fórmula pseudo-booleana é por meio de planos de corte, que representam passos da demonstração como desigualdades lineares inteiras com variáveis 0 e 1 [1].

O artigo mostra que planos de corte podem oferecer demonstrações de tamanho mínimo (o tamanho de uma demonstração em um sistema arbitrário é definido por seu

número de linhas, que representam passos dessa demonstração) para certas propriedades de circuitos multiplicadores, com uma melhora significativa em relação às abordagens anteriores. Experimentalmente, eles mostram que os resolvedores pseudo-booleanos podem verificar propriedades tanto em nível de palavra, nas quais se considera o valor do bit-vector como a representação binária de um número, quanto em nível de bit, nas quais se fazem afirmações sobre o valor de bits particulares, em fórmulas de bit-vectors com multiplicações, superando os resolvedores SMT tradicionais em várias instâncias. Esses resultados motivaram o desenvolvimento deste projeto.

1.5 Integração do PB-blasting a um resolvedor SMT

O objetivo principal do projeto foi implementar um módulo capaz de resolver problemas da teoria de bit-vectors por meio da conversão do problema para uma fórmula pseudo-booleana no *cvc5*, um dos principais resolvedores SMT da atualidade. Atualmente, o cvc5 suporta 9 átomos e 32 termos na teoria de bit-vectors. No projeto, foi definida e implementada uma representação em lógica pseudo-booleana para um subconjunto relevante desses operadores – uma tarefa que revelou-se desafiadora, visto que não há registros de iniciativas semelhantes. Em particular, alguns operadores, como os de Shift bit a bit, sequer possuem definições previamente estabelecidas na literatura. Com base nessas representações, foi desenvolvido um módulo que converte fórmulas da teoria de bit-vectors em fórmulas pseudo-booleanas equisatisfatíveis, processo que foi denominado *PB-blasting*. Por fim, o módulo foi integrado aos resolvedores pseudo-booleanos RoundingSat e Exact para determinar a satisfatibilidade ou insatisfatibilidade das fórmulas convertidas.

2 Implementação

A implementação de um procedimento de PB-blasting no cvc5 é o principal resultado deste Projeto Orientado em Computação. O código-fonte está disponível em https://github.com/alanctprado/cvc5/tree/bv-pb.

2.1 Ponto de entrada para o PB-Blasting

A classe `BVSolverPseudoBoolean` é o ponto de entrada para a funcionalidade de PB-Blasting no cvc5. Essa classe estende um `BVSolver` e é instanciada como o resolvedor interno para a teoria de bit-vectors quando a opção `--bv-solver=pb-blast` é ativada no cvc5. A classe `BVSolver` é uma classe abstrata que provê a interface que é estendida tanto pelo procedimento de PB-blasting quanto pelo procedimento de bit-blasting. O resolvedor interno da teoria de bit-vectors foi convertido de uma instância da classe que implementa bit-blasting para um `BVSolver`, que é inicializado no construtor. O com-

portamento padrão, quando a opção acima não é declarada explicitamente ao executar o cvc5, é inicializar essa variável com a classe de bit-blasting, mantendo o comportamento original do cvc5.

`BVSolverPseudoBoolean` é responsável por chamar os módulos que convertem fórmulas da teoria de bit-vectors em fórmulas pseudo-booleanas equisatisfatíveis e conectá-las a resolvedores pseudo-booleanos como RoundingSat e Exact. Entre suas operações principais, destacam-se os métodos de *pre-notify* e *post-check*, descritos a seguir.

2.1.1 *Pre-notify* de Fatos

O método `preNotifyFact` é chamado antes que novos fatos sejam notificados ao resolvedor. Ele verifica e armazena os fatos relevantes em um buffer interno (`d_facts`) para processamento posterior. Essa operação é utilizada pelo resolvedor da teoria de bit-vectors para identificar os fatos precisam ser convertidos e checados pelo resolvedor interno.

2.1.2 *Post-check* de Fórmulas

O método `postCheck` é executado após a notificação de todos os fatos ao resolvedor, no nível de esforço `EFFORT_FULL`. Sua principal função é chamar o módulo que faz o PB-Blasting das fórmulas de bit-vectors acumuladas no buffer. O processo ocorre em etapas:

1. Os átomos de bit-vectors armazenados em `d_facts` são convertidos para restrições pseudo-booleanas utilizando a classe `PseudoBooleanBlaster`.
2. As restrições geradas são adicionadas ao resolvedor pseudo-booleano configurado (RoundingSat ou Exact).
3. A satisfatibilidade da fórmula resultante é verificada por meio do método `solve` do resolvedor pseudo-booleano.

Com base no estado retornado pelo resolvedor pseudo-booleano (`PB_SAT` ou `PB_UNSAT`), as seguintes ações são realizadas:

- `PB_SAT`: Os átomos satisfatórios são analisados e depurados, auxiliando no diagnóstico do comportamento do módulo.
- `PB_UNSAT`: Um conflito é construído e comunicado ao gerenciador de inferências (`TheoryInferenceManager`), permitindo que o resolvedor SMT tome as ações apropriadas.

2.2 Gerenciamento estratégias para PB-blasting e cache

A classe `TPseudoBooleanBlaster` desempenha um papel central no gerenciamento e aplicação de estratégias para o blasting de átomos e termos no contexto de restrições

pseudo-booleanas. A seguir, serão suas principais funcionalidades, com ênfase na inicialização de estratégias e no armazenamento de resultados.

2.2.1 Função de armazenamento e recuperação de resultados

A `TPseudoBooleanBlaster` utiliza estruturas de *cache* implementadas por meio de *maps* para armazenar os resultados do PB-blasting de átomos e termos. Isso permite evitar a redundância no processamento e melhora a eficiência do sistema:

- `storeAtom(atom, blastedAtom)`: Armazena o resultado do blasting de um átomo no cache.
- `getAtom(atom)`: Recupera o resultado do blasting de um átomo previamente armazenado.
- `storeTerm(term, blastedTerm)`: Armazena o resultado do blasting de um termo no cache.
- `getTerm(term)`: Recupera o resultado do blasting de um termo previamente armazenado.

Além disso, a classe inclui verificadores auxiliares, como `hasAtom` e `hasTerm`, para determinar se um átomo ou termo já foi processado.

2.2.2 Inicialização de estratégias de PB-blasting

A `TPseudoBooleanBlaster` define uma *function table* de estratégias para o PB-blasting de átomos e termos. Essas estratégias são configuradas nas funções:

- `initAtomStrategies()`: Configura as estratégias padrão para o blasting de átomos. Por exemplo, são definidos métodos padrão para operações como igualdade (`EQUAL`), comparações (`BITVECTOR_ULT`, `BITVECTOR_UGT`), entre outras.
- `initTermStrategies()`: Define estratégias para o blasting de termos, como variáveis (`VARIABLE`), constantes (`CONST_BITVECTOR`), operações lógicas (e.g. `BITVECTOR_AND`) e operações aritméticas (e.g. `BITVECTOR_MULT`).

Essas estratégias são indexadas por um identificador inteiro correspondente ao tipo do átomo ou termo e permitem que cada operação seja tratada de forma especializada.

2.2.3 Interface

Além do gerenciamento de caches e da inicialização de estratégias, a classe fornece métodos virtuais para realizar a o PB-blasting de átomos e termos:

- `blastAtom(atom)`: Responsável pelo blasting de átomos. Deve ser implementada pelas classes derivadas. Usada no `postCheck` da `PseudoBooleanBlaster`.
- `blastTerm(term)`: Realiza a explosão de termos e também é um método virtual a ser implementado. Utilizado pelo método `blastAtom(atom)` acima para realizar os blasting dos operandos do átomo, e de forma recursiva para todos os subátomos destes.
- `newVariable(numBits)`: Cria novas variáveis para uso durante o processo de blasting, de acordo com o número de bits especificado.

2.3 Pseudo-Boolean blaster

A classe `PseudoBooleanBlaster` é uma especialização de `TPseudoBooleanBlaster<Node>` e adiciona funcionalidades específicas para o blasting de átomos e termos da teoria de bit-vectors restrições pseudo-Booleanas. Além de herdar os mecanismos de cache e estratégias configuráveis da classe base, ela define funções para lidar com a manipulação de Nodes (estrutura interna do cvc5 que armazena átomos e termos) e variáveis em um contexto mais prático.

2.3.1 Construtor e integração com o ambiente

O construtor de `PseudoBooleanBlaster` inicializa o ambiente e as estruturas internas necessárias:

- `PseudoBooleanBlaster(Env& env, TheoryState* s)`: Configura o ambiente (`Env`) e o gerenciador de nós (`NodeManager`). Além disso, inicializa o contador de variáveis (`d_varCounter`) para gerar identificadores únicos para as variáveis pseudo-Booleanas.

Essa integração permite que a classe opere em sincronia com outras partes do *framework* do cvc5, em particular valendo-se de Nodes, um tipo de dados interno do cvc5 que fornece recursos como contagem de referências e hash consing.

2.3.2 Blasting de átomos

O método `blastAtom` é responsável por realizar o blasting de átomos, convertendo os literais em suas respectivas restrições:

- Verificação de cache: Caso o átomo já tenha sido processado anteriormente, a função recupera o resultado do cache e evita processamento redundante.
- Reescrita de nós: Antes de aplicar as estratégias de blasting, os nós são reescritos para garantir uma forma normalizada.

- Estratégias específicas: Dependendo do tipo (`Kind`) do nó, são utilizadas estratégias de blasting para átomos normais (`d_atomStrategies`) ou átomos negados (`d_negAtom Strategies`). Outra opção para isso seria criar um codificador proposicional para o átomo (i.e. uma variável pseudo-Booleana que o representa), criar restrições de "se e somente se"entre esse codificador e a restrição, e afirmar para o resolvedor se o literal desse codificador é positivo ou negativo (a depender de o átomo ser negado ou não). Apesar dessa estratégia funcionar bem para o bit-blasting, conjecturou-se que a performance dela seria ruim para resolvedores pseudo-Booleano. Por esse motivo, optou-se por implementar estratégias dedicadas para a negação de átomos.

- Armazenamento do resultado: Após o blasting, o resultado é armazenado no cache para uso posterior.

2.3.3 Blasting de termos

O método `blastTerm` realiza o blasting de termos de bit-vectors, convertendo os operadores em suas respectivas restrições:

- Verificação de cache: Se o termo já foi processado anteriormente, o resultado é recuperado do cache.

- Estratégias específicas: O tipo (`Kind`) do termo determina qual estratégia de blasting será aplicada (`d_termStrategies`).

- Validação e armazenamento: após o blasting, o resultado é armazenado no cache (`storeTerm`). O resultado do blasting de um termo corresponde tanto às restrições geradas para ele, quanto às variáveis correspondentes, que podem ser utilizadas por outros termos que possuem esse termo como subtermo (ou átomos que possuem esse termo como operando).

2.3.4 Geração de Novas Variáveis

A `PseudoBooleanBlaster` fornece um método para gerar as variáveis booleanas correspondentes a um termo, que são usadas durante o processo de blasting:

- `newVariable(unsigned numBits)`: Cria um conjunto de variáveis pseudo-Booleanas, com base em um contador interno (`d_varCounter`) para gerar índices únicos. É gerada uma variável para cada bit do termo (i.e., são geradas w variáveis, em que w é a largura bit-vector resultante do termo). Além disso, uma dada estratégia de conversão pode fazer mais de uma chamada a esse método, com vistas a gerar variáveis extras que podem ser usadas como "don't cares"(variáveis livres). As variáveis retornadas pelo método são armazenadas em um Node de tipo `SEXPR`.

2.4 Estratégias de conversão

As estratégias que são utilizadas pelos métodos `blastAtom` e `blastTerm` da classe `PseudoBooleanBlaster` são implementadas no arquivo `pb_blast_strategies.cpp` e serão descritas na Seção 3.

2.5 Interface com os resolvedores pseudo-Booleanos

Em particular, será descrita a implementação para o resolvedor pseudo-Booleano RoundingSat. Uma implementação análoga é feita para o resolvedor Exact e, naturalmente, pode ser feita também para outros resolvedores.

A classe `RoundingSatSolver` é um *wrapper* para o resolvedor `RoundingSat`. A classe estende a classe abstrata `PseudoBooleanSolver`, que define a interface utilizada pela classe `BVSolverPseudoBoolean` para se comunicar com o wrapper de qualquer resolvedor que eventualmente seja integrado ao sistema. Seus proncipais métodos são:

- `RoundingSatSolver(std::string solverPath, Env& env, StatisticsRegistry& registry, const std::string& name, bool logProofs)`: Construtor da classe. Inicializa o resolvedor com o caminho do binário para o RoundingSat. No caso do RoundingSat, foi necessário alterar o build system do cvc5 de forma a baixar o código fonte do resolvedor e compilar um binário, uma vez que o resolvedor não provê opções para ser compilado como uma biblioteca. O caminho desse binário é injetado no código fonte do cvc5 em tempo de compilação, usando variáveis CMake. No caso do Exact, a integração é feita de forma mais transparente, em que o build system é alterado para baixar e compilar o resolvedor como uma biblioteca. Além disso, o construtor configura também o ambiente do cvc5, um registrador de estatísticas, o nome da instância e a opção de log de provas.
- `void addVariable(const Node variable)`: Adiciona uma nova variável pseudo-Booleana ao resolvedor, garantindo que ela ainda não foi registrada.
- `void addConstraint(const Node constraint)`: Adiciona uma nova restrição pseudo-Booleana ao resolvedor, formatando-a no padrão OPB.
- `PbSolveState solve()`: Cria um arquivo no formato OPB baseado nas variáveis e restrições introduzidas pelos dois métodos anteriores. Resolve a fórmula em questão fazendo uma chamada de sistema ao binário do RoundingSat. Retorna o estado da resolução: `PB_SAT`, `PB_UNSAT` ou `PB_UNKNOWN`.
- `void computeSatisfyingAssignment()`: Computa a atribuição de valores às variáveis, caso o problema seja satisfazível.
- `PbValue modelValue(const VariableId variable)`: Retorna o valor atribuído a uma variável no modelo satisfazível.

2.6 Resumo

As classes acimas descritas podem ser utilizadas para resumir o funcionamento do módulo de resolução de problemas da teoria de bit-vectors por meio da conversão do problema em uma fórmula em lógica pseudo-Booleana. Em auto nível, o resolvedor interno da teoria de bit-vectors é inicializado com um `BVSolverPseudoBoolean`, que

1. Utiliza a classe `PseudoBooleanBlaster` para gerar as restrições pseudo-Booleanas relativas a cada átomo e termo da fórmula;

2. Utiliza o wrapper `RoundingSatSolver` para checar se a conjunção dessas restrições é satisfazível ou não.

A implementação deste módulo exigiu a adição de mais de 3000 linhas de código novas e desenvolvidas pelo aluno ao cvc5, além da criação e/ou alteração de mais de 30 arquivos.

3 Regras de conversão

A seguir, são apresentadas as regras para o PB-Blasting de átomos e termos implementadas neste trabalho. Em especial, destaca-se que a fórmula equisatisfatível de um átomo ou termo é composta pelas restrições descritas no item, acrescidas de todas as restrições geradas de forma recursiva para os respectivos subtermos.

3.1 Átomos

Considere que um átomo opera sobre bit-vectors x e y de largura k que, por sua vez, são termos. Sejam $\mathbf{x}$ e $\mathbf{y}$ os vetores de variáveis pseudo-Booleanas que codificam cada respectivo termo. Note que essas variáveis são as mesmas que foram usadas para criar as restrições para o respectivo termo. Cada átomo pode ser representado pela seguinte restrição:

3.1.1 Igualdade (EQ)

- Padrão: EQ(x, y)

$$\sum_{i=0}^{k-1} 2^i \mathbf{x}_i - \sum_{i=0}^{k-1} 2^i \mathbf{y}_i = 0.$$

- Negado: $\neg$(x, y)

$$\mathbf{z} = \text{XOR}(x,\, y)$$

$$\sum_{i=0}^{k-1} \mathbf{z}_i \geq 0.$$

3.1.2 Menor que sem sinal (ULT)

- Padrão: $\mathrm{ULT}(x,\, y)$
$$\sum_{i=0}^{k-1} 2^i \mathbf{y}_i - \sum_{i=0}^{k-1} 2^i \mathbf{x}_i \geq 1.$$
- Negado: $\neg\mathrm{ULT}(x,\, y) \equiv \mathrm{UGE}(x,\, y)$

3.1.3 Menor ou igual sem sinal (ULE)

- Padrão: $\mathrm{ULE}(x,\, y) \equiv \mathrm{UGE}(y,\, x)$
- Negado: $\neg\mathrm{ULE}(x,\, y) \equiv \mathrm{UGT}(x,\, y)$

3.1.4 Maior que sem sinal (UGT)

- Padrão: $\mathrm{UGT}(x,\, y) \equiv \mathrm{ULT}(y,\, x)$
- Negado: $\neg\mathrm{UGT}(x,\, y) \equiv \mathrm{ULE}(x,\, y)$

3.1.5 Maior ou igual sem sinal (UGE)

- Padrão: $\mathrm{UGE}(x,\, y)$
$$\sum_{i=0}^{k-1} 2^i \mathbf{x}_i - \sum_{i=0}^{k-1} 2^i \mathbf{y}_i \geq 0.$$
- Negado: $\neg\mathrm{UGE}(x,\, y) \equiv \mathrm{ULT}(x,\, y)$

3.1.6 Menor que com sinal (SLT)

- Padrão: $\mathrm{SLT}(x,\, y)$
$$-(2^{k-1})\mathbf{y}_{k-1} + \sum_{i=0}^{k-2} 2^i \mathbf{y}_i + 2^{k-1}\mathbf{x}_{k-1} - \sum_{i=0}^{k-2} 2^i \mathbf{x}_i \geq 1.$$
- Negado: $\neg\mathrm{SLT}(x,\, y) \equiv \mathrm{SGE}(x,\, y)$

3.1.7 Menor ou igual com sinal (SLE)

- Padrão: $\mathrm{SLE}(x,\, y) \equiv \mathrm{SGE}(y,\, x)$
- Negado: $\neg\mathrm{SLE}(x,\, y) \equiv \mathrm{SGT}(x,\, y)$

3.1.8 Maior que com sinal (SGT)

- Padrão: $\mathrm{SGT}(x,\, y) \equiv \mathrm{SLT}(y,\, x)$
- Negado: $\neg\mathrm{SGT}(x,\, y) \equiv \mathrm{SLE}(x,\, y)$

3.1.9 Maior ou igual com sinal

- Padrão: SGE(x, y)

$$-(2^{k-1})\mathbf{x}_{k-1} + \sum_{i=0}^{k-2} 2^i \mathbf{x}_i + 2^{k-1}\mathbf{y}_{k-1} - \sum_{i=0}^{k-2} 2^i \mathbf{y}_i \geq 0.$$

- Negado: $\neg\text{SGE}(x, y) \equiv \text{SLT}(x, y)$

3.2 Termos

Considere que um termo que unário opera sobre um bit-vector x, que um termo binário opera sobre bit-vectors x e y e que um termo n-ário atua sobre bit-vectors x_j. Note que esses bit-vectors sobre os quais os termos operam são, por sua vez, subtermos. Consideremos que todos possuam largura k. Sejam $\mathbf{x}$, $\mathbf{y}$ e $\mathbf{x}_i$ os vetores de variáveis pseudo-Booleanas que codificam cada respectivo subtermo.

Criaremos um vetor resultado de variáveis pseudo-Booleanas $\mathbf{r}$ de tamanho k que codificará o resultado do termo. Note, por exemplo, que o vetor $\mathbf{x}$ corresponde justamente ao vetor resultado criado durante o blasting do subtermo, isto é, os vetores de variáveis dos subtermos são os mesmos que foram usados para restringir os respectivos resultados. Cada termo gera as seguintes restrições:

3.2.1 Variável (VAR)

Nenhuma restrição, apenas cria-se um vetor de variáveis Pseudo-Booleanas de tamanho k, que são livres.

3.2.2 Constante (CONST)

Considere um bit-vector constante $[b_{k-1} \cdots b_1 b_0]$. Geramos as seguintes restrições:

$$\mathbf{r}_i = b_i \;\; \forall\; i,\; b_i \in \{0, 1\}.$$

Por exemplo, para $k = 3$ e o termo constante `b#011`, criam-se as restrições

$$\mathbf{r}_0 = 1, \qquad \mathbf{r}_1 = 1, \qquad \mathbf{r}_2 = 0.$$

3.2.3 Ou exclusivo (XOR)

Operador binário. Para cada $i \in \{0, \ldots, k-1\}$, criam-se as restrições:

$$\begin{aligned}
\mathbf{x}_i + \mathbf{y}_i - \mathbf{r}_i &\geq 0 \\
\mathbf{x}_i - \mathbf{y}_i + \mathbf{r}_i &\geq 0 \\
-\mathbf{x}_i + \mathbf{y}_i + \mathbf{r}_i &\geq 0 \\
-\mathbf{x}_i - \mathbf{x}_i - \mathbf{y}_i &\geq -2
\end{aligned}$$

Note que essas restrições correspondem à tradução de $r_i \iff a_i \oplus b_i$ para a CNF:

$$(x_i \vee y_i \vee \neg r_i) \ \wedge \ (x_i \vee \neg y_i \vee r_i) \ \wedge \ (\neg x_i \vee y_i \vee r_i) \ \wedge \ (\neg x_i \vee \neg y_i \vee \neg r_i).$$

3.2.4 Adição (ADD)

Operador n-ário aplicado em bit-vectors x_j, com $j = 1, \ldots, n$.

$$\sum_{i=0}^{l} 2^i \mathbf{r}_i - \sum_{j=1}^{n} \sum_{i=0}^{k-1} 2^i \mathbf{x}_{\mathbf{j}_i} = 0,$$

em que $l = k + \log_2 n - 1$. Essas $\log_2 n$ variáveis extras são variáveis "don't care", usadas para representar um possível overflow, e não são armazenadas como resultado do termo. As variáveis resultado do termo são apenas as variáveis pseudo-Booleanas $r_0, \ldots, r_{k-1}$.

3.2.5 E bit a bit (AND)

Operador n-ário. Gera as restrições

$$\mathbf{x}_{\mathbf{j}_i} - \mathbf{r}_i \geq 0 \ \ \forall \ i, j$$
$$\mathbf{r}_i - \sum_{j=1}^{n} \mathbf{x}_{\mathbf{j}_i} \geq 1 - n \ \ \forall \ i.$$

3.2.6 Ou bit a bit (OR)

Operador n-ário. Gera as restrições

$$\mathbf{r}_i - \mathbf{x}_{\mathbf{j}_i} \geq 0 \ \ \forall \ i, j$$
$$\sum_{j=1}^{n} \mathbf{x}_{\mathbf{j}_i} - \mathbf{r}_i \geq 0 \ \ \forall \ i.$$

3.2.7 Negação bit a bit (NOT)

Operador unário. Gera as restrições

$$\mathbf{r}_i + \mathbf{x}_i = 1 \ \ \forall \ i$$

3.2.8 Multiplicação (MULT)

Considere um tableau intermediário $\mathbf{t}$, de tamanho k^2, para representar os produtos bit a bit.

- Para cada $i, j \in \{0, \ldots, k-1\}$, as restrições associadas às variáveis no tableau representam representam $t_{i,j} \iff x_i \wedge y_j$ e são:

$$\mathbf{x}_i + \mathbf{y}_j - 2\mathbf{t}_{i,j} \geq 0,$$
$$-\mathbf{x}_i - \mathbf{y}_j + \mathbf{t}_{i,j} \geq -1.$$

- A soma ponderada dos elementos do tableau $\mathbf{t}$, chamada equação do equilíbrio dos pesos[3], é usada para calcular o produto:

$$\sum_{i=0}^{k-1}\sum_{j=0}^{k-1} 2^{i+j}\mathbf{t}_{i,j} - \sum_{i=0}^{2k-1} 2^{i}\mathbf{r}_i = 0,$$

em que, assim como na adição, variáveis auxiliares "don't care", são utilizadas para representar um possível overflow, e não são armazenadas como resultado do termo. No caso da multiplicação, é necessário representar o resultado com o dobro de variáveis em relação à largura do bit-vector. Contudo, as variáveis resultado do termo são apenas as variáveis pseudo-Booleanas $r_0, \ldots, r_{k-1}$.

Essa formulação foi proposta por Liew et al.[3].

3.2.9 Concatenação (CONCAT)

Considera-se a concatenação de n bit-vectors x_j de tamanhos k_j, para $j = 1, \ldots, n$. O vetor do resultado $\mathbf{r}$ possui tamanho total $k = \sum_{j=1}^{n} k_j$. Para cada variável do resultado $\mathbf{r}$, as restrições garantem que ele corresponda ao bit correto do bit-vector $\mathbf{x_j}$, seguindo a ordem de concatenação dos operandos:

$$\mathbf{x_j}_i - \mathbf{r}_{\text{index}} = 0 \quad \forall\ i \in \{0, \ldots, k_j - 1\},\ j \in \{1, \ldots, n\},$$

onde index é o índice correspondente em $\mathbf{r}$, calculado como:

$$\text{index} = \left(\sum_{m=1}^{j-1} k_m\right) + i.$$

3.2.10 Extração (EXTRACT)

Considere a extração de uma subsequência de bits de um bit-vector $\mathbf{x}$ de tamanho k, definida pelos índices *high* e *low*. O resultado da extração é armazenado no bitvector $\mathbf{r}$, de tamanho $n = \mathit{high} - \mathit{low} + 1$.

As restrições garantem que cada variável em $\mathbf{r}$ corresponde ao bit correto do bit-vector original $\mathbf{x}$:

$$\mathbf{x}_i - \mathbf{r}_{i-\text{low}} = 0 \quad \forall\ i \in \{\text{low}, \ldots, \text{high}\}.$$

4 Conclusão

O desenvolvimento deste projeto proporcionou uma série de desafios inéditos ao longo da trajetória acadêmica do aluno, resultando em aprendizados valiosos e transformadores. Um dos aspectos mais marcantes foi a imersão na extensa base de código do *cvc5*. Para isso, foi necessário compreender e modificar o sistema de *build*, explorar as opções de linha de comando e ajustar partes específicas do código. Essa experiência também envolveu

uma profunda familiarização com a estrutura do projeto, os padrões de código adotados e a interface de comunicação entre módulos. Além disso, a configuração do ambiente de desenvolvimento foi um exercício importante para aprimorar habilidades no manejo de projetos de grande escala.

Entretanto, o maior desafio e, ao mesmo tempo, o aspecto mais enriquecedor do projeto foi o desenvolvimento das regras de conversão e a implementação do PB-blasting. Esse processo exigiu não apenas um entendimento técnico aprofundado sobre o problema, mas também criatividade e capacidade analítica para conceber soluções computacionalmente eficientes e corretas. A tarefa demandou uma investigação detalhada do código-fonte do *cvc5*, para compreender e implementar as alterações necessárias de forma coesa e integrada. Dada a escala do projeto (com mais de 3000 linhas de código implementadas) e a natureza detalhista da implementação, que envolveu um grande número de operações aritméticas a fim de computar coeficientes e seus sinais, o processo de depuração e desenvolvimento correto do sistema constituiu um desafio de grande magnitude. Esse processo, contudo, resultou em um significativo amadurecimento técnico e profissional por parte do aluno.

Por fim, é essencial reconhecer o papel da relação entre aluno e orientador ao longo do projeto. O suporte fornecido nas reuniões semanais e nas trocas de mensagens contribuiu para o aprendizado e o progresso contínuos. As orientações recebidas ajudaram a direcionar o trabalho e a superar os desafios encontrados. Este projeto representa apenas o início de um esforço maior, que será ampliado e aprofundado no contexto de um projeto de mestrado.

Referências

[1] W. Cook, C.R. Coullard e Gy. Turán. "On the complexity of cutting-plane proofs". Em: *Discrete Applied Mathematics* 18.1 (1987), pp. 25–38. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(87)90039-4. URL: https://www.sciencedirect.com/science/article/pii/0166218X87900394.

[2] Daniel Kroening e Ofer Strichman. *Decision Procedures*. Springer Berlin Heidelberg, 2016. ISBN: 9783662504970. DOI: 10.1007/978-3-662-50497-0. URL: http://dx.doi.org/10.1007/978-3-662-50497-0.

[3] Vincent Liew et al. *Verifying Properties of Bit-vector Multiplication Using Cutting Planes Reasoning*. en. 2020. DOI: 10.34727/2020/ISBN.978-3-85448-042-6_27. URL: https://repositum.tuwien.at/handle/20.500.12708/15523.