

Formalizando Bit-blasting com Pseudo-booleans e verificando no projeto Carcara

Projeto Orientado em Computação II - Pesquisa Científica

Bernardo Borges

Departamento de Ciência da Computação

Universidade Federal de Minas Gerais

Belo Horizonte, Brasil

bernardoborges@dcc.ufmg.br

Abstract—Esta pesquisa envolve a aplicação da teoria de Pseudo-booleans para realizar o procedimento de bit-blasting, utilizado para se raciocinar sobre vetores de bits e operações que os envolvem.

Index Terms—formal methods, pseudo boolean reasoning, bit blasting, proof checking

I. INTRODUÇÃO

A verificação formal é uma área de pesquisa que visa garantir a confiabilidade de sistemas críticos, como hardware e software, onde erros podem ter consequências graves. Entre as ferramentas utilizadas nesse contexto, os solucionadores SMT como o *cvc5* [1] desempenham um papel central ao permitir o raciocínio automático sobre diversas estruturas e operações, como as realizadas sobre vetores de bits, amplamente empregadas em circuitos digitais. Sendo este um processo complexo, técnicas como o bit-blasting são necessárias para dividir o problema de vetores de bits em termos menores, o que tradicionalmente é feito com valores proposicionais, agora experimentamos com valores pseudo-booleans. Este trabalho explora a integração dessas técnicas, propondo uma extensão ao formato de prova *Alethe* [2] e do verificador *Carcara* [3], buscando maior confiança e precisão na validação de provas e seus resultados.

II. REFERENCIAL TEÓRICO

A. Aritmética de Bit-Vectors

A aritmética de Bit-Vectors é amplamente utilizada em áreas como verificação formal, circuitos digitais e design de hardware, pois permite modelar o comportamento de sistemas que operam diretamente em representações binárias. Um vetor de bits é uma sequência de bits (0s e 1s) de comprimento fixo, onde cada bit representa um dígito binário. Operações como soma, subtração, multiplicação e deslocamento são realizadas diretamente sobre esses vetores, respeitando restrições como largura fixa e ocorrência de overflow.

B. Bit Blasting

Uma forma que solucionadores SMT utilizam para raciocinar sobre tais Bit-Vectors é o bit-blasting, uma representação

que utiliza-se de uma variável proposicional para cada bit, o que então pode ser repassado para outras regras de raciocínio.

O bit-blasting converte operações aritméticas e lógicas em vetores de bits para fórmulas proposicionais que podem ser resolvidas por solucionadores SAT (Satisfiability). Nesse processo, cada bit de um vetor é tratado como uma variável proposicional independente, e as operações sobre os vetores são traduzidas em expressões lógicas que representam o comportamento bit a bit. Por exemplo, uma operação de soma em vetores de bits pode ser reduzida a uma série de expressões que modelam o comportamento de somadores completos e meio-somadores, incluindo a propagação de carry (ou transporte) entre os bits. Assim, uma operação que ocorre em um nível mais abstrato, como $A + B$, é traduzida em uma série de restrições lógicas que descrevem a interação entre os bits individuais de A e B .

C. Pseudo-Booleans

Um formato comumente utilizado para representar expressões booleanas é a Forma Normal Conjuntiva (CNF), que consiste da conjunção de cláusulas, em que cada cláusula é a disjunção de variáveis ou negação de variáveis [4]. Neste trabalho, usamos uma outra representação para expressões, chamados Pseudo-Booleans, funções estudadas desde os anos 1960 na área de pesquisa operacional, em programação inteira. Este formato consiste de um somatório do produto de um coeficiente por um literal, que é maior ou igual a uma constante natural. Este formato é exponencialmente mais compacto que o CNF, o que motiva seu uso [5].

D. Bit Blasting Pseudo Booleano

Aproveitando sua estrutura, os Pseudo-Booleans oferecem uma abordagem alternativa para o processo de Bit Blasting, permitindo uma representação mais direta da semântica associada a vetores de bits. Essa semântica está relacionada ao significado numérico ou lógico dos vetores, que pode ser expressa utilizando coeficientes que ponderam a contribuição de cada bit em cálculos aritméticos ou restrições lógicas

E. Formato de Prova Alethe

A corretude é uma preocupação central em solucionadores SMT, dado que eles são amplamente utilizados em verificação formal, onde a precisão é essencial para garantir que sistemas críticos funcionem conforme especificado. No entanto, provar a corretude desses solucionadores é um desafio devido à vasta base de código e às constantes inovações que introduzem alterações frequentes. Quando um solucionador SMT retorna um resultado ‘SAT’, é relativamente simples verificar de forma independente se o modelo gerado satisfaz todas as condições impostas. Contudo, no caso de um resultado ‘UNSAT’, a verificação da corretude requer um *certificado*, ou seja, um registro detalhado dos passos de raciocínio que levaram à conclusão de insatisfabilidade. Nesse sentido, foi desenvolvido o formato Alethe [2], inspirado na linguagem SMT-LIB, que representa de forma flexível e padronizada as provas geradas por solucionadores SMT, que podem ser verificados de forma independente.

F. Verificador de prova Carcara

Carcara [3] é um verificador de prova desenvolvido em Rust pelo laboratório SMITE da UFMG, sob a liderança do professor Haniel Barbosa, projetado para verificar certificados de prova no formato Alethe. Este projeto permite a identificação de erros lógicos nos sistemas que geram esses certificados, aumentando a confiança nos resultados apresentados.

III. CONTRIBUIÇÕES

A contribuição deste trabalho está na definição dos passos tomados para realizar bit-blasting pseudo booleano para cada operação suportada entre bit-vectors, e então a implementação do *checker* no projeto Carcara que permite verificar a correta aplicação de tais regras no formato Alethe.

IV. DEFINIÇÃO DAS REGRAS EM ALETHE

A. Formato das Inequações Pseudo-Booleanas

Uma inequação pseudo-booleana é uma desigualdade da seguinte forma:

$$\sum_i a_i \cdot l_i \geq A$$

onde A é chamado de **constante**, a_i são **coeficientes**, e l_i são **literais**, que são:

- **literal simples**, um termo x ;
- **literal negado**, um termo da forma $(- 1 \ x)$

em que o valor x é uma variável pseudo-booleana, ou seja, ele resolverá para valores 0 ou 1. Todos esses valores são do tipo **Int**.

Para formar um somatório, usamos uma lista de termos somados da forma, $(+ <T1> <T2> \dots 0)$ sempre terminando com um **0**, e cada termo é $(* \ a_i \ <L1>)$, com um coeficiente e um literal.

B. BitBlasting PseudoBooleano em Alethe

Similarmente ao bitblasting regular, o cálculo Alethe usa várias famílias de funções auxiliares para expressar bitblasting pseudo-booleano. As funções **bvsize** e **bv_nⁱ** funcionam da mesma forma que no bitblasting regular, ao passo que **pbbT** e **intOf_m** introduzem e eliminam a representação em pseudo-booleans de um BitVector, que são representados como valores de **Int**.

A família **pbbT** consiste em uma função para cada bitvector de *sort* (**BitVec** n):

$$\mathbf{pbbT} : \underbrace{\mathbf{Int} \dots \mathbf{Int}}_n (\mathbf{BitVec} \ n).$$

o que toma uma lista de argumentos pseudo-booleans e os agrega em um bitvector.

As funções **intOf_m** são o inverso de **pbbT**. Elas extraem um bit de um bitvector como um pseudo-booleano. Assim como o símbolo **extract**, **intOf_m** é usado como um símbolo indexado. Portanto, para $m \leq n$, escrevemos $(_ \ @\mathbf{intOf} \ m)$, para denotar funções

$$\mathbf{intOf}_m : (\mathbf{BitVec} \ n) \rightarrow \mathbf{Int}.$$

e são definidas como

$$\mathbf{intOf}_m \langle u_1, \dots, u_n \rangle := u_m.$$

Todos os outros conceitos não relacionados a essas regras usarão as mesmas definições de bitblasting proposicional.

C. Regras de Predicados

1) **pblast_bveq**:

Considere os bitvectors \mathbf{x} e \mathbf{y} de comprimento n . O bitblasting pseudo-booleano de sua igualdade é expresso por:

$$i. \triangleright (= \ x \ y) \approx A \quad (\mathbf{pblast_bveq})$$

Em que o termo “ A ” é a restrição pseudo-booleana:

$$\sum_{i=0}^{n-1} 2^i x_{n-i-1} - \sum_{i=0}^{n-1} 2^i y_{n-i-1} = 0$$

2) **pblast_bvult**:

A operação ‘unsigned-less-than’, menor ou igual, sem sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright (\mathbf{bvult} \ x \ y) \approx A \quad (\mathbf{pblast_bvult})$$

Em que o termo “ A ” é a restrição pseudo-booleana:

$$\sum_{i=0}^{n-1} 2^i y_{n-i-1} - \sum_{i=0}^{n-1} 2^i x_{n-i-1} \geq 1.$$

3) *pbblast_bvugt*:

A operação ‘unsigned-greater-than’, maior sem sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvugt} \ x \ y) \approx A \quad (\text{pbblast_bvugt})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$\sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} \geq 1.$$

Alternativamente, em termos de **pbblast_bvult**, temos:

$$i. \triangleright \quad (\mathbf{bvugt} \ x \ y) \approx (\mathbf{bvult} \ y \ x) \quad (\text{pbblast_bvugt})$$

4) *pbblast_bvuge*:

A operação ‘unsigned-greater-or-equal’, maior ou igual sem sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvuge} \ x \ y) \approx A \quad (\text{pbblast_bvuge})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$\sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} \geq 0.$$

5) *pbblast_bvule*:

A operação ‘unsigned-less-or-equal’, menor ou igual sem sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvule} \ x \ y) \approx A \quad (\text{pbblast_bvule})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$\sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} \geq 0.$$

Alternativamente, em termos de **pbblast_bvuge**, temos:

$$i. \triangleright \quad (\mathbf{bvule} \ x \ y) \approx (\mathbf{bvuge} \ y \ x) \quad (\text{pbblast_bvule})$$

6) *pbblast_bvslt*:

A operação ‘signed-less-than’, menor com sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvslt} \ x \ y) \approx A \quad (\text{pbblast_bvslt})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{y}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} + 2^{n-1} \mathbf{x}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} \geq 1$$

7) *pbblast_bvsge*:

A operação ‘signed-greater-than’, maior com sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvsge} \ x \ y) \approx A \quad (\text{pbblast_bvsge})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{x}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} + 2^{n-1} \mathbf{y}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} \geq 1$$

Alternativamente, em termos de **pbblast_bvslt**, temos:

$$i. \triangleright \quad (\mathbf{bvsge} \ x \ y) \approx (\mathbf{bvslt} \ y \ x) \quad (\text{pbblast_bvsge})$$

8) *pbblast_bvsge*:

A operação ‘signed-greater-or-equal’, maior ou igual com sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvsge} \ x \ y) \approx A \quad (\text{pbblast_bvsge})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{x}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} + 2^{n-1} \mathbf{y}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} \geq 0$$

9) *pbblast_bvsle*:

A operação ‘signed-less-or-equal’, menor ou igual com sinal, sobre BitVectors com n bits é expressa por:

$$i. \triangleright \quad (\mathbf{bvsle} \ x \ y) \approx A \quad (\text{pbblast_bvsle})$$

Em que o termo “ A ” é verdadeiro se, e somente se:

$$-(2^{n-1})\mathbf{y}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} + 2^{n-1} \mathbf{x}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} \geq 0$$

Alternativamente, em termos de **pbblast_bvsge**, temos:

$$i. \triangleright \quad (\mathbf{bvsle} \ x \ y) \approx (\mathbf{bvsge} \ y \ x) \quad (\text{pbblast_bvsle})$$

D. Regras Aritméticas

1) *pbblast_pbbvar*:

Conversão de um BitVector de n bits para n variáveis pseudo-booleanas introduzidas com **pbbT**:

$$i. \triangleright \quad x \approx \mathbf{pbbT} \ x_1 \dots x_{n+1} \quad (\text{pbblast_pbbvar})$$

2) *pbblast_pbbconst*:

Restrições para cada bit do bitvector constante b :

$$i. \triangleright (b \approx \mathbf{pbbTr}) \wedge \bigwedge_{i=0}^{n-1} (r_i = \mathbf{VAL}(b_{n-i-1})) \quad (\text{pbblast_bvsle})$$

Em que expandimos **VAL**(b_i) em:

- ($b_i = 0$) se b_i é 0
- ($b_i = 1$) se b_i é 1

3) *pbblast_bvxor*:

A operação ‘bxor’ sobre BitVectors com n bits é expressa usando desigualdades PseudoBooleanas por:

$$i. \triangleright \quad (\mathbf{bxor} \ x \ y) \approx [r_0, \dots, r_1] \wedge A \quad (\text{pbblast_bxor})$$

O termo “ A ” é a conjunção dessas desigualdades pseudo-booleanas e o termo \mathbf{r} representa o resultado da operação ‘bxor’ entre \mathbf{x} e \mathbf{y} , para $0 \leq i < n$:

$$-\mathbf{r}_i + \mathbf{x}_i + \mathbf{y}_i \geq 0$$

$$-\mathbf{r}_i - \mathbf{x}_i - \mathbf{y}_i \geq -2$$

$$\mathbf{r}_i + \mathbf{x}_i - \mathbf{y}_i \geq 0$$

$$\mathbf{r}_i - \mathbf{x}_i + \mathbf{y}_i \geq 0$$

4) *pbblast_bvand*:

A operação ‘bvand’ sobre BitVectors com n bits é expressa usando desigualdades PseudoBooleanas por:

$$i. \triangleright (\text{bvand } x \ y) \approx [r_0, \dots, r_1] \wedge A \quad (\text{pbblast_bvand})$$

O termo “ A ” é a conjunção dessas desigualdades pseudo-booleanas e o termo \mathbf{r} representa o resultado da operação ‘bvand’ entre \mathbf{x} e \mathbf{y} , para $0 \leq i < n$:

$$\mathbf{x}_i - \mathbf{r}_i \geq 0$$

$$\mathbf{y}_i - \mathbf{r}_i \geq 0$$

$$\mathbf{r}_i - \mathbf{x}_i - \mathbf{y}_i \geq -1$$

V. VERIFICAÇÃO DAS REGRAS NO CARCARA

Com a definição das regras, partimos para a implementação do checker dentro do Carcara. O primeiro passo é alterações na ‘ast’, a árvore de sintaxe abstrata que faz a leitura de um arquivo de prova no formato alethe e obtém uma representação da estrutura dos termos envolvidos naquela regra. Como um parser para esse formato já existe no projeto, temos apenas que estendê-lo para aceitar os novos símbolos `pbbT` e `int_of`, para introdução e eliminação de bitvectors por meio de variáveis pseudo-booleanas, nesse caso, **Int**.

1) Extensão do Parser:

No arquivo `carcara/src/ast/term.rs` criamos novos construtores:

```
pub enum Operator {
    // ...
    BvPbBTerm
}
pub enum ParamOperator {
    // ...
    BvIntOf
}
```

Com essa extensão podemos usar ‘int_of’ e ‘pbbterm’ nos nossos testes e arquivos contendo estes símbolos serão corretamente lidos pelo sistema.

2) Verificação de Regras:

Podemos dessa forma implementar a verificação de aplicações corretas de cada regra, o que se dará no arquivo ‘`carcara/src/checker/rules/pb_blasting.rs`’,

Vejamos com mais detalhes a verificação da regra **pbblast_bveq**:

Entrada: Argumentos da regra, incluindo um `pool` de termos, um conjunto de `args`, e um termo de conclusão.

Saída: Verifica se a regra foi aplicada corretamente ou gera um erro caso alguma condição não seja satisfeita.

1: **procedure** PBBLAST_BVEQ(*pool*, *args*, *conclusion*)

2: **Passo 1: Verificar a estrutura da conclusão**

3: Extraia os termos x , y , sum_x e sum_y da conclusão, garantindo que ela esteja no formato:

$$(= (= x \ y) (= (- (+ \dots) (+ \dots)) 0))$$

Caso a estrutura não corresponda, gere um erro.

4:

5: **Passo 2: Verificar a largura do bitvector**

6: Recupere a largura do bitvector *size* do `pool` para a variável x .

7: **Passo 3: Verificar os termos de sum_x quanto à sua correção**

8: **for** (i, el) **em** sum_x **do**

9: Extraia o coeficiente c , o índice idx e o bitvector bv do i -ésimo elemento de sum_x , garantindo que o termo esteja no formato:

$$(* \ c \ ((_ \ \text{int_of} \ idx) \ bv))$$

Se $i = 0$, permita que c seja omitido.

10: Converta c e idx para tipos inteiros.

11: Verifique as seguintes condições:

- c é igual a 2^i
- idx é igual a $size - i - 1$
- bv é igual a x

12: **if** qualquer uma dessas condições falhar **then**

13: Gere um erro

14: **end if**

15: **end for**

16: **Passo 4: Verificar os termos de sum_y quanto à sua correção**

17: Realize as mesmas verificações do **Passo 3**, mas com sum_y e y .

18: **Passo 5: Retornar sucesso**

19: **end procedure**

O código completo dessa verificação pode ser encontrado no apêndice, mas o que podemos analisar aqui é que várias propriedades sintáticas são testadas pelo código:

- O número de argumentos aplicados à regra é o esperado?
- O tipo dos termos aplicados é apropriado?
- A estrutura da conclusão corresponde à forma lógica esperada, garantindo que seja uma equivalência entre uma igualdade de bitvector e a igualdade de duas somas subtraídas a zero?
- A largura do bitvector para os termos x e y está definida corretamente e disponível no conjunto de termos?
- Os coeficientes na soma estão representados corretamente como potências de dois, garantindo o alinhamento com a natureza binária dos bitvectors?
- Os índices nos termos da soma são consistentes com a ordenação esperada, especificamente $size - i - 1$, onde i representa a posição atual do bit?
- Os bitvectors sendo indexados dentro das somas estão corretamente correspondidos às variáveis originais x e y ?
- A correspondência geral entre os termos da soma e a igualdade original do bitvector é preservada, garantindo que não haja incompatibilidade na representação da equivalência?

Essas verificações garantem que a regra esteja de acordo com os requisitos sintáticos e semânticos da sintaxe Alethe, garantindo que a transformação representada pela regra seja logicamente válida. Qualquer incompatibilidade na estrutura,

tipo ou conteúdo dos termos resultará em um erro, sinalizando uma violação dos requisitos da regra.

Para atender todos estes aspectos é que o Carcara implementa para cada nova construção um conjunto de testes de unidade, que são executados rapidamente e além de evitar possíveis erros, servem como documentação viva do comportamento esperado por cada regra. Continuando o exemplo da regra **pblast_bveq**, vejamos um destes testes:

Caso de Teste: Igualdade em Bits Simples

1: **procedure** TEST-PBBLAST_BVEQ

2: **Definições:**

- Declarar uma constante bitvector $x1$ de largura 1:
(declare-const $x1$ (_BitVec 1))
- Declarar uma constante bitvector $y1$ de largura 1:
(declare-const $y1$ (_BitVec 1))

3: **Entrada:** Uma cláusula de conclusão para a regra **pblast_bveq**:

- Seja lhs a igualdade entre os bitvectors:

$$lhs = (= x1 y1)$$

- Seja sum_x a representação da soma para $x1$:

$$sum_x = (+ (* 1 ((_int_of 0) x1)) 0)$$

- Seja sum_y a representação da soma para $y1$:

$$sum_y = (+ (* 1 ((_int_of 0) y1)) 0)$$

- Combine no conjunto completo de conclusão:

$$conclusion = (= lhs (= (- sum_x sum_y) 0))$$

4: **Resultado Esperado:** `true` (a regra foi aplicada corretamente e verificada com sucesso).

5: **end procedure**

Neste teste, declaramos duas constantes bitvector de um único bit ($x1$ e $y1$), decompomos os termos de soma em sum_x e sum_y , e fornecemos a cláusula de conclusão completa de forma simplificada. O teste verifica se a regra **pblast_bveq** valida corretamente essa transformação de igualdade, garantindo que todos os requisitos estruturais e semânticos sejam atendidos. O resultado esperado do teste é `true`, indicando a validação bem-sucedida da regra.

CONCLUSÕES

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, “cvc5: A Versatile and Industrial-Strength SMT Solver”, Abril de 2022, Acesso em https://link.springer.com/chapter/10.1007/978-3-030-99524-9_24.
- [2] H. Barbosa, M. Fleury, P. Fontaine, H. Schurr, “The Alethe Proof Format - An Evolving Specification and Reference”, Dezembro de 2024, Acesso em <https://verit.gitlabpages.uliege.be/alethe/specification.pdf>.
- [3] B. Andreotti, H. Lachnitt, H. Barbosa “Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format”, Abril de 2023, Acesso em https://link.springer.com/chapter/10.1007/978-3-031-30823-9_19.
- [4] “Conjunctive normal form”, Encyclopedia of Mathematics, EMS Press, 2001.
- [5] J. Nordström, “Pseudo-Boolean Solving and Optimization”, Fevereiro de 2021.

APÊNDICE

A. Código para a regra *pblast_bveq*

```
pub fn pblast_bveq(RuleArgs { pool, conclusion, .. }: RuleArgs) -> RuleResult {
  // Check that conclusion is an equivalence between bitvector equality
  // and difference of summations equal to zero
  let ((x, y), ((sum_x, sum_y), _)) =
    match_term_err!((= (x y) (= (- (+ ...) (+ ...)) 0)) = &conclusion[0])?;

  // Drops the last element, that is zero
  let sum_x = &sum_x[..sum_x.len() - 1];
  let sum_y = &sum_y[..sum_y.len() - 1];

  // Check that `x`'s bitvector width exists in the `pool`
  let Sort::BitVec(x_width) = pool.sort(x).as_sort().cloned().unwrap() else {
    unreachable!();
  };

  // Transforms the width of x to a usize type
  let x_width = x_width.to_usize().unwrap();

  // Check that `sum_x` has the same length as `x`
  rassert!(x_width == sum_x.len(), CheckerError::Unspecified);

  // Check that `y`'s bitvector width exists in the `pool`
  let Sort::BitVec(y_width) = pool.sort(y).as_sort().cloned().unwrap() else {
    unreachable!();
  };

  // Transforms the width of y to a usize type
  let y_width = y_width.to_usize().unwrap();

  // Check that `sum_y` has the same length as `y`
  rassert!(y_width == sum_y.len(), CheckerError::Unspecified);

  // Check that both bitvectors x and y have the same length
  rassert!(x_width == y_width, CheckerError::Unspecified);

  // Define a closure to check the terms for a bitvector and its summation
  let check_bitvector_sum =
    |sum: &[Rc<Term>], width: usize, bitvector: &Rc<Term>| -> RuleResult {
      for (i, element) in sum.iter().enumerate() {
        // Match `element` with a coefficient times an `int_of` application on bitvector `bv`
        let (c, (idx, bv)) = match_term_err!((* c ((_ int_of idx) bv)) = element)?;

        // Convert `c` and `idx` to integers
        let c: Integer = c.as_integer_err()?;
        let idx: Integer = idx.as_integer_err()?;

        // Check that the coefficient is actually 2^i
        rassert!(c == (1 << i), CheckerError::Unspecified);
        // Check that the index is actually `width - i - 1`
        rassert!(idx == width - i - 1, CheckerError::Unspecified);
        // Check that the bitvector being indexed is actually `bitvector`
        rassert!(*bv == *bitvector, CheckerError::Unspecified);
      }
      Ok(())
    };

  // Use the closure to check the `sum_x` terms
  check_bitvector_sum(sum_x, x_width, x)?;

  // Use the closure to check the `sum_y` terms
  check_bitvector_sum(sum_y, y_width, y)?;

  Ok(())
}
```

B. Teste para a regra pbbblast_bveq

```
mod tests {
  #[test]
  fn pbbblast_bveq() {
    test_cases! {
      definitions = "
        (declare-const x1 (_ BitVec 1))
        (declare-const y1 (_ BitVec 1))
        (declare-const x2 (_ BitVec 2))
        (declare-const y2 (_ BitVec 2))
      ",
      "Equality on single bits" {
        r#" (step t1 (c1 (= (= x1 y1) (= (- (+ (* 1 ((_ int_of 0) x1)) 0)
          (+ (* 1 ((_ int_of 0) y1)) 0)) 0))) :rule pbbblast_bveq) "#: true,
      }
      "Omit multiplication by 1" {
        r#" (step t1 (c1 (= (= x1 y1) (= (- (+ ((_ int_of 0) x1) 0)
          (+ ((_ int_of 0) y1) 0)) 0))) :rule pbbblast_bveq) "#: true,
      }
      "Not a subtraction of sums" {
        r#" (step t1 (c1 (= (= x1 y1) (= (+ (* 1 ((_ int_of 0) x1)) 0) 0)))
          :rule pbbblast_bveq) "#: false,
      }
      "Malformed products" {
        r#" (step t1 (c1 (= (= x1 y1) (= (- (+ (* 0 ((_ int_of 0) x1)) 0)
          (+ (* 1 ((_ int_of 0) y1)) 0)) 0))) :rule pbbblast_bveq) "#: false,
        r#" (step t1 (c1 (= (= x1 y1) (= (- (+ (* 1 ((_ int_of 0) x1)) 0)
          (+ (* 0 ((_ int_of 0) y1)) 0)) 0))) :rule pbbblast_bveq) "#: false,
      }
      "Equality on two bits" {
        r#" (step t1 (c1 (= (= x2 y2) (= (- (+ (* 1 ((_ int_of 1) x2))
          (* 2 ((_ int_of 0) x2)) 0) (+ (* 1 ((_ int_of 1) y2)) (* 2
          ((_ int_of 0) y2)) 0)) 0))) :rule pbbblast_bveq) "#: true,
      }
    }
  }
}
```