Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation with Branching Programs*

Jordan Gergov and Christoph Meinel

FB IV – Informatik
Universität Trier
Postfach 3825
D-5500 Trier

Extended Abstract

Abstract. A central issue in the solution of many computer aided design problems is to find concise representations for circuit designs and their functional specification. Recently, a restricted type of branching programs (OBDDs) proved to be extremely useful for representing Boolean functions for various CAD applications [Bry92]. Unfortunatelly, many circuits of practical interest provably require OBDD–representations of exponential size. In the following we systematically study the question up to what extend more concise BP-representations can be successfully used in symbolic Boolean manipulation, too. We prove, in very general settings,

- The frontier of efficient (deterministic) symbolic Boolean manipulation on the basis of BP–representations are read–once–only branching programs (BP1).
- The frontier of efficient probabilistic manipulation with BP–based data structures are parity read–once–only branching programs ($\oplus$–BP1).

Since BP1s and $\oplus$–BP1s are generally more (sometimes even exponentially more) succinct than OBDD–representations our results make accessible more succinct types of BPs as data structures for practical purposes. (A BP1–package as well as a $\oplus$–BP1–package are in preparation.) On the other side, our results together with the results obtained in [GM92] show that the solution of basic tasks in Boolean manipulation for less restricted BP-types becomes NP–hard.

1 Introduction

Most of the problems in digital system design, combinatorial optimization, mathematical logic, or artificial intelligence can be formulated in terms of Boolean

* Proc. of the *Symposium on Theoretical Aspects of Computer Sciences*, LNCS 665, 576–585, Springer Verlag, 1993

functions. Doing this the problems can be solved by means of algorithms that work with Boolean functions. The efficiency of such algorithms depends essentially on the data structures used for the representation of the involved Boolean functions. Hence, a central issue in the solution of many computer aided design problems is to find "efficient" data structures. Here "efficiency" means, on one hand, that the representation of the Boolean functions should be as succinct as possible, and, on the other hand, that the Boolean manipulation on the basis of this representation can be performed as efficient as possible. Since, usually, these requests are contradictory it is necessary to find good compromises.

Considering data structure based on branching programs, nowadays, the most popular representation that combines these requirements up to a certain degree are OBDDs (ordered binary decision diagrams). OBDDs introduced by Bryant [Bry86] provide a representation well suited for various CAD applications such as verification, synthesis, testing and simulation [e.g. BBR90, BCMD90, FFK88, MWBS88]. Most actual CAD–tools manipulate Boolean functions on the basis of such OBDDs. Unfortunately, the efficiency of OBDD–representations depends crucially on certain ordering constraints. Moreover, there are many functions of practical interest (e.g. integer multiplication, hidden weighted bit function, indirect storage function) which provably require exponential size OBDD–representations [Bry91, Bry92].

Since OBDDs have found such a great resonance in practice it would be most desirable to overcome the difficulties of the OBDD–model. Our approach is the following: On the basis of a systematical investigations of the complexity of the basic tasks of Boolean manipulation in terms of various restricted branching program models [GM92] we try to find out the most succinct BP–representations that allow feasible or at least probabilistically feasible Boolean manipulations.

In the following we prove:

1. One can work with read–once–only branching programs (BP1) similarly efficient as with OBDDs;
2. Working with parity read–once–only branching programs ($\oplus$–BP1) similarly as with OBDDs Boolean manipulations are probabilistically feasible.

Both results are of great practical importance since BP1s as well as $\oplus$–BP1s are generally more (sometimes even exponential more) succinct than OBDDs [BHR91]. Moreover, while it is possible to prove exponential lower bounds on the size of BP1s and to show that $\oplus$–BP1s possess more (sometimes even exponential more) computational power than BP1s (and, hence OBDDs) [A&86,KMW91] up to now nobody was able to prove superpolynomial lower bounds on the size of $\oplus$–BP1s.

The paper is structured as follows. In Section 2 we introduce BP–representations for Boolean functions and mention the basic tasks of Boolean Manipulation. Shortly we cite some facts concerning OBDDs. Then, in Section 3 we show that Boolean Manipulation in terms of read–once–only branching programs is feasible. In order to obtain this result we introduce and investigate in detail consistent BP1s. In the final Section 4 we show that Boolean Manipulation in terms of parity read–once–only branching programs is probabilistically feasible.

The key point in doing this is the development of a probabilistic equivalence test for $\oplus$–BP1s.
Almost all proofs have to be omitted in this extended abstract for the sake of limited space.

2 BP–Representations and Basic Tasks of Boolean Manipulation

2.1 Branching Programs and Ω-Branching Programs

One of the most important and interesting data structures for Boolean functions are branching programs. A *branching program (BP)* [e.g. Mei89] is a directed acyclic connected graph where each node has outdegree 2 or 0. There is a distinguished node, the *source*, which has indegree 0, and at most two nodes of outdegree 0, the *0-sink* and the *1-sink*, which are labelled by the Boolean constants 0 and 1, respectively. The remaining nodes are labelled by Boolean variables taken from a set $X_n = \{x_1, \ldots, x_n\}$. The two edges starting in a non-sink node are labelled by 0 and 1, respectively. The 0–successor and the 1–successor nodes of v are denoted by $v0$ and $v1$, respectively.
A branching program P over $X_n = \{x_1, \ldots, x_n\}$ can be used to *represent* or *compute* a Boolean function $f_P \in \mathbb{B}_n$ in the following way. Each input $w = w_1 \ldots w_n \in \{0,1\}^n$ defines a *computational path* through P that starts at the source. If this path reaches a node v which is labelled by the variable $x_i \in X_n$ then it follows the edge labelled by w_i. Since P is finite and acyclic finally a sink is reached whose label gives the value $f_P(w)$ on w. P is said to be a *BP-representation of* f_P. Two branching programs P, P' are called *computationally equivalent* if they represent the same function
$f_P = f_{P'}$.
In order to increase the computational power of branching programs Ω–branching programs were introduced [Mei88]. An Ω–branching program P over X_n, $\Omega \subseteq \mathbb{B}_2$, is a branching program some of whose non–sink nodes are labelled by binary functions $\omega \in \Omega$ instead of Boolean variables . The Boolean values assigned to the sinks of P extend to Boolean values associated with all nodes of P in the following way: If both successor nodes v_0, v_1 of a node v of P carry the Boolean values δ_0 or δ_1, and if v is labelled by a Boolean variable x_i we associate with v the value δ_0 or δ_1 according to $x_i = 0$ or $x_i = 1$. If v is labelled by a function ω then we associate with v the value $\omega(\delta_0, \delta_1)$. P computes the value 1 (0) on an input $w \in \{0,1\}^n$ if the source of P associates with 1 (0) under w. For $\Omega = \{\vee\}$, $\{\wedge\}$, $\{\oplus\}$, $\{\vee, \wedge\}$ we speak of *disjunctive, conjunctive, parity* or *alternating branching programs,* respectively. An Ω–branching program is said to be a *read–once–only* (Ω–BP1) if each variable appears, on each source–to–sink path, at most once. A BP1 is called an *ordered binary decision diagram* (OBDD) if it tests the variables on all paths according to a predefined order. In [Mei88] it was proved that, within polynomial size, each Ω-BP (Ω-BP1), $\Omega \subseteq \mathbb{B}_2$, is computational equivalent to an Ω'–BP (Ω'-BP1) with $\Omega' \in \{\emptyset,\ \{\vee\},\ \{\wedge\},\ \{\oplus\},\ \{\vee, \wedge\}\}$.

2.2 Basic tasks of Boolean Manipulation

Basic tasks of Boolean manipulation in CAD–systems that have to be performed frequently are
- equivalency test (especially satisfiability test) and
- binary synthesis.

Further manipulation tasks as composition of Boolean Functions, computing a restriction or the complement, evaluation for a given input are either trivial in the framework of the BPs or can be reduced to the above basic tasks. Observe, that working with restricted Ω-BP–representations one has to maintain the restriction in the course of performing the synthesis.

Definition
Let X–Ω–BP denotes a class of restricted Ω–BPs. The *satisfiablility problem for X–Ω–BPs*, $SAT_{X-\Omega-BP}$, consists in the test whether a given X–Ω–BP represents a satisfiable Boolean function. The *equivalence problem for X–Ω–BPs*, $EQU_{X-\Omega-BP}$, consists in the test whether two given X–Ω–BPs represent the same function. If $*$ denotes a binary operator for Boolean functions then the $*$–*synthesis problem for X–Ω–BPs*, $*$–$SYN_{X-\Omega-BP}$, is the problem of constructing an X–Ω–BP–representation for the function $f = f' * f''$ from given X–Ω–BP–representations of f' and f''. Since $\{\wedge, \oplus, \neg\}$ is a complete basis for the set of Boolean operators it suffices to consider $\wedge$–$SYN_{X-\Omega-BP}$ and $\oplus$–$SYN_{X-\Omega-BP}$ (the complement is trivial for BPs).

2.3 OBDDs

Bryant has recognized that OBDD–representations [Bry86] for Boolean functions possess a variety of exceptional properties desirable for practical applications in CAD-systems. Among others, OBDD–representations allow efficient satisfiability and equivalence tests as well as feasible synthesis algorithms. Hence, a number of CAD–tools uses OBDDs as data structure for Boolean functions [e.g. BBR90, BCMD90, FFK88, MWBS88].

Theorem 1.

1. Each function $f \in \mathbb{B}_n$ can be canonical represented by means of a reduced OBDD w. r. t. predefined variable ordering [Bry86] (see the next section for the Definition of reduced BP);
2. EQU_{OBDD} is feasible [FHS78];
3. Let $*$ be a binary operator for Boolean functions. $*$-SYN_{OBDD} is feasible for OBDDs that test variables in the same order[Bry86]. $\Box$

Unfortunately, there exist important Boolean functions such as integer multiplication, hidden weighted bit function, indirect storage function that, for no variable ordering, can be represented by polynomial size OBDDs [e.g. Bry91, BHR91].

3 Deterministic Feasible Boolean Manipulation in Terms of Read-once-only Branching Programs

3.1 Complexity of the Basic Tasks for BP1s

Investigating the complexity of the basic tasks of Boolean manipulation in terms of BP1s one obtains the following results:

Proposition 2.

1. SAT_{BP1} is feasible, $SAT_{BP1} \in \mathbf{P}$;
2. EQU_{BP1} is probabilistic feasible, $EQU_{BP1} \in \mathbf{co}-\mathbf{R}$ [BCW80];
3. EQU_{BP2} is co-NP–complete [GM92] (in a BP2 each variable appears at most two times on a source-to-sink path);
4. $\wedge$–SYN_{BP1} and $\wedge$–SYN_{OBDD} are NP–hard [GM92]. □

3.2 Feasible Boolean Manipulation within BP1–ideals

While the complexity of the satisfiability problem and the equivalence problem for BP1–representations is at least probabilistically feasible binary synthesis for such representations is, due to Proposition 2, **NP**–hard. At the first glance, this fact prohibits the use of BP1s as a data structure for CAD–tools. However, investigating the situation in more detail, we succeeded in realizing that also synthesis can be performed efficiently if one restricts oneself to BP1s of consistent types. For further details we refer to [GM92b].

Definition. (reduced branching programs)

Let v and v' be nodes of a BP P. v and v' are said to be *congruent* if the labels of v and v' are equal, and, if both v, v' are non–sinks, both 0–successor nodes $v0$ and $v'0$ as well as both 1–successor nodes $v1$ and $v'1$ are congruent. v is called *trivial* if it has only one successor in P, i.e. if $v0 = v1$. An *algebraical BP–reduction* applied to a BP P consists in the identification of the subprograms P_v and $P_{v'}$ for two congruent nodes v and v'. A *simple BP–reduction* applied to P consists in the deletion of the label of a trivial node and the identification of this node with its successor node. Undoing an algebraical (simple) BP-reduction is called an *algebraical (simple) BP-extension.* After the application of sufficient many reductions (algebraical, respectively simple reductions) to P we get a BP which can not be further reduced (algebraical, respectively simple reduced), and denote this by $red(P)$ ($red_a(P)$, respectively $red_s(P)$).

Proposition 3. (Efficient reductions of BP1s)

Let P be a BP1. $red(P)$, $red_a(P)$ and $red_s(P)$ are unique determened, and can be constructed in linear time. □

In order to identify certain similarities between different branching programs we have to compare their types.

Definition. (BP1–types, complete BP1–types)

A *BP1–type* is defined similar as a BP1 with the only exception that it possesses merely one sink. The BP–type $typ(P)$ of a BP1 P is derived from P by identifying all sinks of P. A BP1–type over X_n is said to be *complete* if, on each source–to–sink path, each variable of X_n appears. A BP1–type τ is called a *subtype* of

BP1–type τ', $\tau \leq \tau'$, if τ can be constructed from τ' be applying algebraical or simple type–reductions (the BP-reductions can be extended in a obvious way to BP-types).

Definition. (BP1–ideals)
A set $\mathcal{K}$ of BP1s over X_n is called a *BP1–ideal* generated by a BP1–type τ, if

$$\mathcal{K} = \{Q : \ Q \text{ is a BP1 with } red_a(typ(red(Q))) \leq red_a(\tau)\}.$$

Proposition 4.
It can be tested efficiently whether two given BP1s belong to a common BP1-ideal $\mathcal{K}$ generated by a type τ, or whether a BP1 lies in $\mathcal{K}$ if τ is given. □

In terms of BP1–ideals our results can be summarized as follows:

Theorem 5.
Let $\mathcal{K}$ be a BP1–ideal generated by a complete type τ over X_n.

1. Each function $f \in \mathbb{B}_n$ can be canonical represented within $\mathcal{K}$ by means of a reduced program;
2. $EQU_{\mathcal{K}}$ can be decided in linear time;
3. Let $*$ be a binary operator.The synthesis $*$–$SYN_{\mathcal{K}}$ can be performed in time $O(size(P')size(P'')size(\tau))$, and if we drop the requirement to get a result in $\mathcal{K}$ in time $O(size(P')size(P''))$.

Proof.
Let P' and P'' compute f' and f'', and let $* \in \mathbb{B}_2$ be any binary operator. Starting with P' and P'' the procedure *synthesis* given in the appendix of the paper constructs a BP1 P that computes $f' * f''$. After easy modification we can get a BP P in $\mathcal{K}$. □

In [GM92b], combining methods and ideas for decision trees with the methods developed in the preceding section we are able to identify even larger BP1–ideals for which the manipulation remains feasible.

4 Probabilistic Feasible Boolean Manipulation in Terms of Parity Read-once-only Branching Programs

4.1 The Complexity of the Equivalence for Ω-BP1s

Since the equivalence problem for BPs (and, hence, for Ω–BPs) is intractable already for very restricted two–times–only BPs (Proposition 2) we concentrate ourself to Ω–BP1s. Further we have to consider only the following types of Ω–BP1s [Mei88].

Theorem 6.
1. Let $\Omega \in \{\{\wedge\}, \{\vee\}, \{\wedge, \vee\}\}$. Then $EQU_{\Omega-BP1}$ is **co-NP**–complete;
2. $EQU_{\oplus-BP1}$ is probabilistic feasible, $EQU_{\oplus-BP1} \in \mathbf{co-R}$.

Proof.
To prove the second part of the theorem we extend a result of Blum, Chandra and Wegman [BCW80]. The function computed by a $\oplus$-BP1 can be characterized by a polynom over $GF(2^m)$. We can work in $GF(2^m)$ as in $\mathbb{F}_2[x]/p(x)$ where $p(x)$ is an irreducible polynomial of degree m. Such polynomials can be find quickly. For

instance, $x^{2\cdot 3^k} + x^{3^k} + 1$ is irreducible in $\mathbb{F}_2[x]$ for each $k \in \mathbb{N}$ [LN86]. It is easy to see that $k = \lceil log_3(\frac{\lceil log_2(2n) \rceil}{2}) \rceil$ produces a field with sufficient many elements (more than $2n$) and the multiplication in this field can be done efficiently (in time polynomial in n). To complete the proof see the corresponding algorithm given in the appendix. □

Since the proposition continues to hold also for type–restricted Ω–BP1s (see the next section), the only candidates for a manageable extension of BP1s in the settings of Ω–BPs are $\oplus$–BP1s. In the following we are going to show that $\oplus$–BP1s indeed provide a manageable extension at the price of giving up determinism.

4.2 Probabilistic Feasible Analysis and Manipulation of $\oplus$-BP1s

In order to make binary synthesis feasible, similar as in the case of BP1s, we consider type–restricted $\oplus$–BP1s.

Definition. (Type–restricted $\oplus$–BP1s)
A $\oplus$–BP1 P defines a $\oplus$–BP1–type $typ(P)$ in a similar way as BP1s do. P is called *type–restricted* if it is possible to transform $typ(P)$ into an ordinary BP1–type τ_P by means of a sequence of algebraical BP–type–reductions, restricted simple type–reductions applied merely to $\oplus$–nodes, and (simple or algebraical) BP1–type–extensions.

In order to give an example of a type–restricted $\oplus$–BP1 consider $\oplus$–BP1s with the following property. The types of the BP1s rooting in the $\oplus$–nodes are comparable after eliminating $\oplus$–nodes. Indeed, starting in the sinks, all $\oplus$–nodes can be easily eliminated in such $\oplus$–BP1–types. One has only to extend one of the both types by means of certain BP–type–extensions before both types can be identified. Of course, the possibility of such an elimination of the $\oplus$–nodes does nothing say about the redundance of these nodes within the original $\oplus$–BP1s. For example, the $\oplus$–synthesis of BP1s (i.e. the elimination of one $\oplus$–node) can produce a quadratic blow–up.

Definition. (Consistent BP1s and $\oplus$–BP1s)
Two $\oplus$–BP1s (BP1s) P, P' are said to be *consistent* if they are type–restricted and if the corresponding BP1–types τ_P and $\tau_{P'}$ are subtypes of a common type.

Theorem 7.

Let P and P' are two consistent $\oplus$-BP1s.
1.The equivalence of P and P' can be tested probabilistically in polynomial time;
2.The synthesis of P and P', $\wedge$–$SYN_{\oplus-BP1_c}$, can be solved in polynomial time for each $* \in \mathbb{B}_2$.

Proof.
Since the $\oplus$–synthesis can be trivially performed on $\oplus$–BP1s it suffices to consider $\wedge$–$SYN_{\oplus-BP1_c}$. Due to the distributivity of $\wedge$ and $\oplus$ a slight and straightforward extension of the synthesis algorithm for consistent BP1s given in the appendix yields the assertion of Theorem 7. □

5 Concluding Remarks

We have identified consistent BP1s and consistent $\oplus$–BP1s as those classes of branching programs where important CAD–tasks can be performed efficiently. Further there are functions which can be computed with a polynomial size type–restricted $\oplus$-BP1 but require exponential size BP1s (e. g. $\oplus cl_{n,3}$ — odd number of triangles in a graph [A&86]). Since BP1s and hence $\oplus$–BP1s are often much more succinct than OBDDs we essentially contribute to the research of data structures suited for CAD applications. A BP1–package as well as a $\oplus$–BP1–package are in preparation.

References

[A&86] M. Ajtai, L. Babai, P. Hajnal, J. Komlos, P. Pudlak, V. Rödl, E. Szemeredi, G. Turan: Two Lower Bounds for Branching Programs, Proc. 18. ACM STOC, 1986, 30-38.

[BBR90] K. S. Brace, R. E. Bryant, R. L. Rudell: Efficient Implementation of a BDD package. Proc. of 27th Design Automation Conf., 1990, 40–45.

[BCW80] M. Blum, A. K. Chandra, M. N. Wegman: Equivalence of Free Boolean Graphs Can Be Decided Probabilistically in Polynomial Time, IPL 10, 2, 1980, 80–82.

[BHR91] Y. Breitbart, H. B. Hunt III, D. Rosenkrantz: The Size of Binary Decision Diagrams Representing Boolean Functions, submitted to Inf. and Comp. 1991.

[Bry86] R. E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Computers, C-35, 8, 1986, 677–691.

[Bry92] R. E. Bryant: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, to appear in IEEE Trans. Computers, 1992.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, D.L. Dill: Sequential Circuit Verification Using Symbolic Model Checking, Proc. 27th IEEE DAC'90, 1990, 46–51.

[CHS74] R. L. Constable, H. B. Hunt III, S. Sahni: On the Computational Complexity of Scheme Equivalence, Proc. 8th Princeton Conf. on Information Sciences and Systems, 1974.

[FHS78] S. Fortune, J. Hopcroft, E. M. Schmidt: The Complexity of Equivalence and Containment for Free Single Program Schemes. LNCS 62, 1978, 227–240.

[FFK88] M. Fuita, H. Fujisawa, N. Kawoto: Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams, Proc. IEEE ICCAD'88, 1988, 2–5.

[GM92] J. Gergov, Ch. Meinel: Analysis and Manipulation of Boolean Functions in Terms of Decision Graphs, Proc. WG'92, LNCS, 1992.

[GM92b] J. Gergov, Ch. Meinel: Efficient Analysis and Manipulation of OBDDs Can Be Extended to Read–once–only Branching Programs, Forschungsbericht Nr. 92-10, Univ. Trier, 1992.

[LN86] R. Lidl, H. Niederreiter: Introduction to Finite Fields and Their Applications. Cambridge University Press, 1986.

[KMW91] M. Krause, Ch. Meinel, S. Waack: Separating the Eraser Turing Machine Classes $L_e, NL_e, co\text{–}NL_e$ and P_e, TCS 86 (1991), 267–275.

[Mei88] Ch. Meinel: The Power of Polynomial Size Ω–Branching Programs, Proc. STACS'88 (Bourdeaux), LNCS 294, 81–90.

[Mei89] Ch. Meinel: Modified Branching Programs and Their Computational Power, Springer-Verlag, LNCS 370, 1989.

[Mei91] Ch. Meinel: Branching Programs - An Efficient Data Structure for Computer-Aided Circuit Design, Preprint UGH Paderborn, Nr. 93, 1991.

[MKLC89] S. Muroga, Y. Kambayashi, H. C. Lai, J. N. Culliney: The Transduction Method, IEEE Trans. Computers, C-38, 1989, 1404–1424.

[MWBS88] S. Malik, A. Wang, R. Bryant, A. Sangiovanni–Vincentelli: Logical Verification Using Binary Decision Diagrams in a Logical Synthesis Environment. Proc. IEEE Intern. Conf. on CAD, 1988, 6–9.

6 Appendix

6.1 Feasible Algorithm for the Binary Synthesis of BP1s

We start with a procedure that, for each node v of a given BP1 P makes some information available about the variables to be tested in the successor nodes of v.

Definition. (Vectors s_v)
Let P be a BP1 over $X_n = \{x_1, \ldots, x_n\}$, and let v be a node of P. The vector $s_v \in \{0,1\}^n$ is defined as follows:

$$s_v[x] = \begin{cases} 1 \text{ if } x \text{ has to be tested on any } v\text{–to–sink path in } P \\ 0 \text{ otherwise} \end{cases}$$

Lemma.
The set of all vectors s_v, $v \in P$, can be computed in time $O(n * size(P))$. $\square$
Presenting our algorithm we use the following denotations: Let v and v' are nodes of two BP1s P and P', respectively, and let $l(v)$ and $l(v')$ denote the variables tested in v and v'. We set

$$c(v, v') = \begin{cases} 0 \text{ if } l(v) \neq l(v') \text{ and if } s_v[l(v')] = s_{v'}[l(v)] = 1 \\ 1 \text{ otherwise,} \end{cases}$$

and

$$l(v, v') = \begin{cases} l(v) & \text{if } l(v) = l(v') \text{ or } s_{v'}[l(v)] = 0 \\ l(v') & \text{if } s_v[l(v')] = 0 \\ \text{undefined otherwise} \end{cases}$$

By $source(P)$ we denote the source of P. If $x \in X_n$ and x does not appear on any path in P then $P_{|_{x=0}} = P_{|_{x=1}} = P$. In the case $x = l(source(P))$ we denote by $P_{|_{x=\alpha}}$, $\alpha \in \{0,1\}$, the BP1 that starts in the α-successor of $source(P)$.
Algorithm. (Synthesis of consistent BP1s)

Input:
A Boolean operator $* \in \mathbb{B}_2$, and two consistent reduced BP1s P' and P''.
Output:
A BP1 P consistent with P' and P'' that computes $f = f_{P'} * f_{P''}$.
Assumptions:
We assume that vectors s_v are computed for all nodes v of P' and P''.
$T[.,.]$ is a $size(P) \times size(P')$ table. Before the first call of $synthesis(*, P', P'', P)$ all entries of T are equal to zero.
procedure $synthesis(*, P', P'', P)$.

begin
  if
    One of P' and P'' is a α-sink, $\alpha \in \{0,1\}$
  then
    {Generate P by modifying the sinks of the other BP1 according to α and $*$;
    (For instance, if $\alpha = 1$ and $* = \oplus$ a β-sink has to be changed to a $\bar{\beta}$-sink, $\beta \in \{0,1\}$);
    $T[source(P'), source(P'')] :=$ a pointer to the result of the generated P};
  if
    $T[source(P'), source(P'')] \neq 0)$
  then
    $T[source(P'), source(P'')]$ is pointer to the already synthesized BP1 P
  else
    {We introduce a new node v and set $l(v) := l(source(P'), source(P''))$;
    $synthesis(*, P'_{|l(v)=0}, P''_{|l(v)=0}, P_0)$; $synthesis(*, P'_{|l(v)=1}, P''_{|l(v)=1}, P_1)$;
    We construct P by taking v as source and P_α, $\alpha \in \{0,1\}$, as α-successor of v;
    $T[source(P'), source(P'')]$ is set to a pointer to P}; **end**;
begin /** main program **/
  $synthesis(*, P', P'', P)$;
  We compute $red(P)$;
  $return(red(P))$;
end.

6.2 Probabilistical Feasible Equivalence Test for $\oplus$-BP1s

Definition. Let P be a $\oplus$-BP1 and I an integral domain with $char(I) = 2$.Then we assign to each node v of P a polynom p_v in the following way:
1. $p_v = \alpha$ if v is an α-sink;
2. $p_v = (1-x)p_{v_0} + xp_{v_1}$ if v is a inner node where v_α is the α-successor of v, $\alpha \in \{0,1\}$, and $l(v) = x$;
3. $p_v = p_{v_0} + p_{v_1}$ if $l(v) = \oplus$ where v_0, v_1 denote the successors of v.
The polynom of P, $p(P)$, is the polynom of the source v_0 of P, $p(P) = p_{v_0}$.
$p(P)$ can be computed with $size(P)$ many additions and at most $2size(P)$ many multiplications.
Input:
Two $\oplus$-BP1s P' and P''.

Output:
If P' and P'' are equivalent the algorithm answers always with "yes" othercase the algorithm gives "no" with probability greater than $\frac{1}{2}$.
Assumptions:
F is a finite field of characteristic two with more than $2n$ elements.

procedure *equ(P',P'')*;
begin
choose independently and uniformly $x_1, x_2, \ldots, x_n$ *from* F;
compute $p(P')$ *in* F;
compute $p(P'')$ *in* F;
if $(p(P') = p(P''))$
then return("yes"); /*P' and P'' are equivalent */
else return("no"); /* P' and P'' are not equivalent */
end.