Compiling Domain Consequences

Alexandre Papadopoulos and Barry O'Sullivan
Cork Constraint Computation Centre
University College Cork, Ireland
Email: {a.papadopoulos|b.osullivan}@4c.ucc.ie

***Abstract*—This paper presents a method for computing all the domain consequences of a constraint problem. Domain consequences are a generalisation of prime implicates to multi-valued constraint problems. We define ordered automata to encode a large, potentially exponential, number of domain consequences. We design a range of algorithms that directly operate on this compact representation, with a complexity that depends on its size and not the size of the encoded set. This allows us to generate the domain consequences of a problem even for problems that have an exponential number of domain consequences. Furthermore, a simple empirical study illustrates the effectiveness of the method in compiling a large number of domain consequences, and the compactness of this representation.**

I. Introduction

Many problems in Artificial Intelligence can be seen as instances of consequence finding. Consequence finding corresponds to the problem of deriving specific knowledge that is intensionally contained in a knowledge base, by finding certain consequences of the knowledge base. Techniques for consequence finding and its applications have been extensively surveyed by Marquis [1]. Prime implicates play an important role in consequence finding, and many consequence finding tasks consist in computing the prime implicates of a propositional base, or restricted sets of prime implicates. In their experimental study, Chatalic and Simon [2] show for the first time that consequence finding algorithms can scale up to problems of practical significance.

Existing work in consequence finding has concentrated on propositional logic. In this paper, we decide to tackle multivalued constraint problems. Prime implicates have many uses in AI, such as for diagnosis [3], [2], the ATMS [4], [5], abduction and explanations [2], amongst others. Consequently, a generalisation of consequence finding methods to constraint problems has a potential for many applications too. For example, we suggested in previous work [6] that precise and extensive information about conflicts is key for determining explanations. We proposed to compute in advance all possible conflicts, as a compilation step, where a possible conflict refers to incompatibilities between values of the variables of the problem. This idea, or more precisely the complementary idea, is formalised by the concept of *domain consequence*, which generalise prime implicates to multivalued constraint problems. Of course, the number of consequences a problem entails can be intractably high. Therefore, the actual set of domain consequences has to be, in turn, compactly represented, in order to obtain an efficient *compilation technique*.

Zero-suppressed Binary Decision Diagrams (ZBDDs) have been shown to be an efficient data structure for representing a collection of subsets of a universe, by representing the characteristic function of each subset contained in the collection [7]. This property has been successfully exploited by Chatalic and Simon [8] to represent sets of clauses, with an efficient compression power (see Figure 1). Exploiting this particular semantics, Chatalic and Simon define some additional specific operators that allow them to implement and use the original Davis and Putnam resolution procedure. This approach is the first (and, to our knowledge, single) approach of a consequence finding algorithm that scales up to problems of practical significance.

Example 1. Consider the ZBDD in Figure 1. Each node is labelled by a literal, and has two outgoings arcs: a 1-labelled arc (plain line), meaning the literal is present, a 0-labelled arc (dashed line), meaning the literal is omitted. Paths from the root node to a terminal node, those labelled by **0** or **1**, correspond to sets of literals, only those of which end at node **1** being kept.

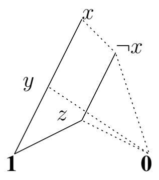

Fig. 1. An example ZBDD encoding a collection of two sets of literals $\{xy, \neg xz\}$.

We propose to adapt this approach for our context. At first glance, one could think that this can be straightforwardly applied to represent sets of domain consequences, since a domain consequence too can be represented as a subset of a universe. However, this approach needs to be extended to take into account the particular semantics of domain consequences, which is different to the semantics of clauses. More specifically, trivial sets (sets including $\{x, \neg x\}$) generalise differently to domain consequences, whether they are tautologies (including $x \vee \neg x$) or contraditions (including $x \wedge \neg x$). This does not affect the representation; however, it affects the interpretation of the representation by the procedures that use it. This is actually a fundamental difference and makes this extension not trivial.

Furthermore, for presentation purposes, we choose to define and use a restricted class of automata instead of ZBDDs (although they are conceptually equivalent [9]), as they allow for a more succinct and legible presentation. However, note that in the rest of the paper, it sometimes helps presentation to first discuss the clausal case. Since existing work on the clausal case relies on ZBDDs, we will, for consistency with the existing literature, present the clausal case using ZBDDs.

The *main contribution* of this paper is a procedure that, based on an automaton representation of domain consequences, computes all the domain consequences of a problem.

II. Prime Implicates and Domain Consequences

It is useful to recall the concept of domain consequence, as defined in [6], which is a generalisation of prime implicates.

Let Φ be a finite set of clauses in Conjunctive Normal Form (CNF) (or any NNF for the purposes of the following definitions), and let C be a disjunction of literals (a *clause*).

Definition 1 (Implicate)**.** Let C and C' be two clauses.

- C is an *implicate* of Φ if $\Phi \models C$.
- C *subsumes* C' iff $C \subseteq C'$ iff $C \models C'$.

Usually we only consider non-trivial implicates, i.e. implicates that are not a tautology. A prime implicate is then defined as follows.

Definition 2 (Prime Implicate)**.** C is a *prime implicate* of Φ iff:

- C is an implicate of Φ and
- $\forall C'$ which is an implicate of Φ, $C' \models C \Rightarrow C \models C'$.
- Equivalently, $\forall C' \subset C$, $\Phi \not\models C'$.

Definition 3 (Prime Implicants)**.** An implicant is a conjunction of literals C such that $C \models \Phi$. Prime implicants are thus defined in a converse way.

We can now generalise those definitions to multivalued constraint problems. Suppose we have a problem defined on a set of variables $X_1, \ldots, X_n$, taking their values from the domains $D(X_1), \ldots, D(X_n)$.

Definition 4 (Domain Conflict)**.** The notion of domain conflict is defined as follows:

- A *domain conflict* for given problem is given by the sequence of domains $C = \langle D_1, \ldots, D_n \rangle$, such that imposing $X_i \in D_i$ for each X_i is inconsistent. Such a conflict can be seen as a conjunction of unary constraints, which is inconsistent.
- Given two domain conflicts $C_1 = \langle D_1, \ldots, D_n \rangle$ and $C_2 = \langle D'_1, \ldots, D'_n \rangle$, we note that $C_1 \subseteq C_2$ if $\forall X_i$, $D_i \subseteq D'_i$.
- A *maximal* domain conflict is a domain conflict C such that no domain conflict $C' \neq C$ exists with $C \subseteq C'$. In other words, every component D_i of C is maximal.

We can observe that the notion of *maximal* domain conflict recovers the classic one of *minimal* conflict, in the sense that for a given i, having $D_i = D(X_i)$ is equivalent to having no constraint at all on X_i. Thus, the more values that are in the D_i sets, the fewer constraints there are on the corresponding X_i variables.

With that definition in mind, we can define the consequence of a given problem, thus generalising the concept of prime implicate.

Definition 5 (Domain Consequence)**.** A domain consequence is defined as follows:

- A *domain consequence* of a problem is given by $P = \langle D_1, \ldots, D_n \rangle$ such that $\langle \overline{D_1}, \ldots, \overline{D_n} \rangle$ is a domain conflict, with $\overline{D_i} = D(X_i) \setminus D_i$.
- Given two domain consequences P and P', whose corresponding domain conflicts are C and C', we have $P \subseteq P'$ if $C' \subseteq C$. To reuse the classic terminology, we may say that P *subsumes* P'.
- A domain consequence P is *minimal* if its corresponding domain conflict is maximal. In other words, no domain consequence $P' \neq P$ exists such that $P' \subseteq P$.

A given consequence of the problem is read as a disjunction. In other words, for any solution of the problem, it must be true that $X_1 \in D_1$ or $X_2 \in D_2$ or... and so on.

Definition 6 (Set of domain consequences)**.** Let Π be a problem. $\mathit{Cons}(\Pi)$ is the set of all the minimal domain consequences of Π, that is if P is a domain consequence of Π, then $\exists P' \in \mathit{Cons}(\Pi)$ such that $P' \subseteq P$, and $\forall P, P' \in \mathit{Cons}(\Pi)$, $P \subseteq P' \Rightarrow P = P'$.

A set of domain consequences must be seen as a conjunction, and so $\mathit{Cons}(\Pi) \equiv \Pi$. The strategy will thus be, given a problem, to compute all of its consequences, as efficiently as possible, and represent them as compactly as possible.

III. Ordered Automata for Collections of Subsets

A. General Definition

We propose to use regular automata to represent a collection of subsets, with an added property that values must be ordered. Concretely, we impose a total order on the alphabet, and this order affects how values can be introduced: a recognised word can be composed only of symbols in strictly increasing order.

Definition 7 (Ordered Automaton)**.** An ordered automaton M is defined as a 6-tuple $\langle Q, \Sigma, \leq, \delta, q_0, F \rangle$, with:

- Q a finite set of states,
- Σ a set of symbols (the alphabet),
- $\leq$ a total order on Σ,
- δ a function $Q \times \Sigma \to Q$ (the ordered transition function), such that for any string s recognised by M, $\forall i < j \leq |s|, s[i] < s[j]$,
- $q_0 \in Q$ the initial state,
- $F \subseteq Q$ the final states.

We additionally impose the usual properties on the automaton, precisely that it should be deterministic, trimmed (remove transitions that cannot reach final states) and minimal. These properties will be implicitly assumed.

We introduce some additional notation:

- $\sigma(q) = \{a \in \Sigma | \delta(q, a) \text{ is defined}\}$, the set of labels of the outgoing transitions of a state;

- $\delta(q) = \{\delta(q, a) | a \in \sigma(q)\}$, the successors of a state;
- $\delta^*(q) = \{q' \in Q | (q' = q) \vee (q' \in \delta(q'') \wedge q'' \in \delta^*(q))\}$, the set of states accessible from a state;
- the digraph $D = (Q, E)$, with $E = \{(q, q') | q \in Q \wedge q' \in \delta(q)\}$, is called the underlying digraph of the automaton.

Let us remark that, as an ordered automaton can only recognise strings of finite length, its underlying digraph must be acyclic.

The semantics that we attach to an ordered automaton M is defined as follows. Given a collection of subsets of a universe U, we build a corresponding ordered automaton M such that:

- Σ is in one-to-one correspondence with U;
- $\leq$ can be the natural total order holding on U, or any arbitrarily chosen total order;
- a subset $S \subseteq U$ is uniquely represented by the string composed of the symbols corresponding to each element in S, in increasing order;
- a subset $S \subseteq U$ is in the collection *iff* the corresponding string is recognised by M.

Let us observe that, with this semantics, an ordered automaton representing a collection of subsets has a single final state if all the subsets of the collections are incomparable. Indeed, if an ordered automaton has more than one final state, then at least one final state must have a successor (two final states with no successor being equivalent). This state recognises a given set which is a subset of any set recognised by any successor of this state.

Some particular cases of this semantics include the empty collection, and the collection containing the empty set only. The empty string represents the empty set. An automaton with only one state that is both initial and final recognises the empty string only, and thus interprets as $\{\emptyset\}$. An automaton that has no final state does not recognise any string, and thus interprets as $\emptyset$ (if it is minimal, the initial state is the unique state).

B. Domain Sequences

We can use this data structure to represent a collection of domain sequences. Indeed, if we assume, without loss of generality, that the domains of each variable do not share common values, we can notice that a domain sequence is merely a subset of the universe defined by the union of all domains. Therefore, we can apply the approach just described.

From a logical point of view, if the considered domain sequences represent domain consequences, a collection of domain consequences interprets as a conjunction (of disjunctions). In particular, $\emptyset$ interprets as *true* and $\{\emptyset\}$ interprets as *false*.

In order to encode a collection of domain sequences with an ordered automaton, we first have to match the values of the initial domains to unique values for each variable, then to impose a total order on the resulting universe (which can result from an existing total order on the original domains). More formally, given a sequence of variables $X_1, \ldots, X_n$, with domains $D(X_i)$, we define an automaton M as follows:

- $\forall a \in D(X_i)$, we create a unique symbol, denoted a_{X_i}, thus $\Sigma = \{a_{X_i} | i \leq n, a \in D(X_i)\}$;
- we set an order $\leq$ such that $a_{X_i} \leq b_{X_j}$ iff $i \leq j$ or $i = j \wedge a \leq b$.

Then, we can encode a specific domain consequence as a string over Σ, and a collection of domain consequences as the automaton recognising the set of the corresponding strings.

Example 2. Let $\mathcal{P} = \{\langle ab, ab, a\rangle, \langle b, bc, a\rangle, \langle b, \emptyset, b\rangle\}$, on the variables X, Y, Z (we assume $a < b < c$). We define $\Sigma = \{a_X, b_X, c_X, a_Y, b_Y, c_Y, a_Z, b_Z, c_Z\}$, declared in increasing order, and the ordered automaton encoding $\mathcal{P}$ is given in Figure 2. Conceptually, we are basically labelling the transitions of the automaton with variable assignments.

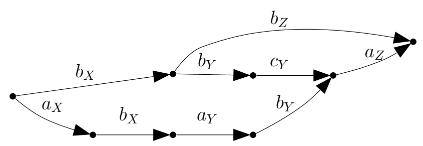

Fig. 2. An automaton representing three domain consequences

IV. Operations on Ordered Automata

With the given semantics, we can realise operations on sets of domain consequences as operations on automata. We will make use of the following notation:

- Given a state q_0 belonging to a certain automaton, as a shortcut notation, we denote by $\mathcal{P}(q_0)$ the collection of domain consequences *encoded by the automaton* whose initial state is q_0.
- Let $q_\emptyset$ be a constant dummy state.
- Let $M_\emptyset = \langle\{q_\emptyset\}, \emptyset, \emptyset, \emptyset, q_\emptyset, \emptyset\rangle$. We have here $\mathcal{P}(q_\emptyset) = \emptyset$; $q_\emptyset$ is not an accepting state.
- Let $M_{\{\emptyset\}} = \langle\{q_\emptyset\}, \emptyset, \emptyset, \emptyset, q_\emptyset, \{q_\emptyset\}\rangle$. We have here $\mathcal{P}(q_\emptyset) = \{\emptyset\}$; $q_\emptyset$ is an accepting state.

In order to simplify presentation, we will assume that, for a given state q and symbol a, $\delta(q, a)$ being undefined is equivalent to having $\delta(q, a) = q_\emptyset$. In particular, we can express $\sigma(q)$ as $\{a \in \Sigma | \delta(q, a) \neq q_\emptyset\}$. This allows us, in the algorithms, not to consider explicitly the case where a transition for a given symbol is undefined.

Finally, we assume that the considered automata are always defined with the same alphabet and the same order.

A. Creating States

Every time a state is built, it has to be registered using the function `register-state`. After a state has been registered, it cannot be further modified (transitions cannot be added or removed, successors cannot be modified). If the newly built state has no outgoing transition, $q_\emptyset$ is returned; this ensures that the automaton is trimmed. If a state already exists that is equivalent to the newly built state, i.e. the same symbols lead to the same states, that state is returned instead. Otherwise, the newly built state itself is returned. This way, the invariant that all states in the register are pairwise inequivalent is maintained, which ensures minimality.

```
Function register-state(q)
1 if $\sigma(q) = \emptyset \wedge \neg isFinal(q)$ then return $q_\emptyset$
2 if $\exists q'$ in the register such that $q$ is equivalent to $q'$ then return $q'$
3 Add $q$ to the register
4 return $q$
```

B. Operators

We can now define a number of operators that form the basic pieces of the compilation procedure. We will not describe the algorithm implementing each operator, as this would unnecessarily load this paper. As an example, we provide the algorithm for one of the operators, and the reader is referred to [10] for further details.

a) Subsumed Removal.: Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two collections of domain consequences. We define $\mathcal{P}_1 \setminus_\models \mathcal{P}_2$ as the set of all the domain consequences of $\mathcal{P}_1$ that are not subsumed by any domain consequence of $\mathcal{P}_2$.

Let M_1 and M_2 be two ordered automata, with respective initial state q_0^1 and q_0^2. We define the operator $q_0^1 \setminus_\models q_0^2$ such that $\mathcal{P}(q_0^1 \setminus_\models q_0^2) = \mathcal{P}(q_0^1) \setminus_\models \mathcal{P}(q_0^2)$.

b) Union.: Let $\mathcal{P}_1, \mathcal{P}_2$ be two collections of domain consequences free from subsumed elements. We define $\mathcal{P}_1 \cup_\mu \mathcal{P}_2$ as the set $\mu(\mathcal{P}_1 \cup \mathcal{P}_2)$, i.e. the union of $\mathcal{P}_1$ and $\mathcal{P}_2$ where only non-subsumed elements are kept. From a logical point of view, $\mathcal{P}_1 \cup_\mu \mathcal{P}_2$ is the conjunction between $\mathcal{P}_1$ and $\mathcal{P}_2$.

Let M_1 and M_2 be two ordered automata free from subsumed elements, with respective initial states q_0^1 and q_0^2. We define the operator $q_0^1 \cup_\mu q_0^2$ in the following function.

```
Function $q_0^1 \cup_\mu q_0^2$
1 if $q_0^1$ is final then return $q_0^1$
2 if $q_0^2$ is final then return $q_0^2$
3 else
4     $q_0 \leftarrow$ Create new state
5     forall $a \in \sigma(q_0^1) \cup \sigma(q_0^2)$ in decreasing order do
6         $\delta(q_0, a) \leftarrow (\delta(q_0^1, a) \cup_\mu \delta(q_0^2, a)) \setminus_\models q_0$
7     return register-state($q_0$)
```

Proposition 1. $\mathcal{P}(q_0^1 \cup_\mu q_0^2) = \mathcal{P}(q_0^1) \cup_\mu \mathcal{P}(q_0^2)$.

Proof: For a given symbol a, if both states have an outgoing transition labelled with a, then the union of respective consequences starting with a must be performed. If only one state has an outgoing transition with a, then the recursive call will be performed with $q_\emptyset$ as one of the operands, and so the other operand will be returned unmodified. At a given iteration of the forall statement, let $\mathcal{P}_1 = \mathcal{P}(\delta(q_0^1, a) \cup_\mu \delta(q_0^2, a))$ and $\mathcal{P}_2 = \mathcal{P}(q_0)$. By induction hypothesis, both are free from subsumed elements. Any element from $\mathcal{P}_2$ subsumed by an element P of $\mathcal{P}_1$ will not be subsumed by $P \cup \{a\}$. However, if an element from $\mathcal{P}_1$ is subsumed by an element of $\mathcal{P}_2$, it will still be after adding a to it. By removing those subsumed elements, we obtain a subsumption-free automaton (see Figure 3). ■

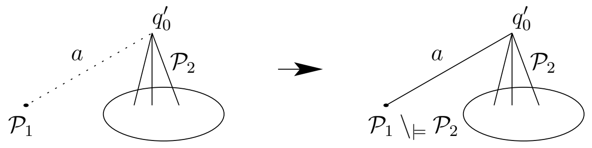

Fig. 3. Adding $\mathcal{P}_1$ without introducing subsumed elements

c) Product.: Let $\mathcal{P}_1, \mathcal{P}_2$ be two collections of domain consequences free from subsumed elements. We define $\mathcal{P}_1 \otimes \mathcal{P}_2 = \mu\{P_1 \cup P_2 | P_1 \in \mathcal{P}_1, P_2 \in \mathcal{P}_2 \wedge P_1 \cup P_2 \text{ is not trivial}\}$. Intuitively, $\mathcal{P}_1 \otimes \mathcal{P}_2$ contains all the domain consequences formed by the union of a domain consequence respectively from $\mathcal{P}_1$ and from $\mathcal{P}_2$ that are not trivial and that are not subsumed by any other such domain consequence. From a logical point of view, $\mathcal{P}_1 \otimes \mathcal{P}_2$ is the disjunction between $\mathcal{P}_1$ and $\mathcal{P}_2$.

Let M_1 and M_2 be two ordered automata free from subsumed elements, with respective initial states q_0^1 and q_0^2. We define the operator $q_0^1 \otimes q_0^2$ such that $\mathcal{P}(q_0^1 \otimes q_0^2) = \mathcal{P}(q_0^1) \otimes \mathcal{P}(q_0^2)$.

V. Compiling the Domain Consequences of a Problem

We now present a procedure that allows us to compute all the domain consequences of a problem, represented by an ordered automaton. This constitutes the main contribution of this paper.

At the core of this compilation procedure is the closure by resolution. Put simply, given an ordered automaton M encoding a collection $\mathcal{P}$ of domain consequences, we need to apply an operator that builds an ordered automaton $Cons(M)$ that encodes $Cons(\mathcal{P})$. We define this operator in this section.

A. The Operator for the Clausal Case[1]

To simplify presentation, we will first consider the clausal case. Let us first define the distribution operator as follows.

Definition 8 (Distribution)**.** Let Φ be a collection of sets of literals. The distribution of Φ, denoted $\boxtimes\Phi$, is the collection of sets of literals obtained by keeping one literal from each set in Φ, and such that non-minimal and trivial sets (i.e. sets containing a literal in non-negated and negated forms) are omitted.

Example 3. Let $\Phi = \{xy, \neg xz\}$, $\boxtimes\Phi = \{xz, \neg xy, yz\}$.

If Φ is a CNF, the operator consists in distributing the conjunction over the disjunction, and thus converting from a CNF to an equivalent DNF. Conversely if Φ is a DNF, the operator distributes the disjunction over the conjunction, thus converting it to an equivalent CNF. In case a DNF is obtained,

[1]We would like to thank Hélène Fargier for her very helpful explanations and for pointing us to a number of relevant papers, which helped us develop this section.

trivial sets correspond to contradictions (including $l \wedge \neg l$), and in the case a CNF is obtained, trivial sets correspond to tautologies (including $l \vee \neg l$). It also follows from this observation that $\boxtimes \boxtimes \Phi \equiv \Phi$.

Example 4. Consider Example 3. If we interpret Φ as the DNF formula $(x \wedge y) \vee (\neg x \wedge z)$, we can interpret $\boxtimes\Phi$ as the CNF formula $(x \vee z) \wedge (\neg x \vee y) \wedge (y \vee z)$, which is equivalent to the DNF Φ.

Let us now make some observations. Let Φ be a CNF, Φ' be the DNF defined as $\Phi' = \bigvee_{C \text{ such that } C \models \Phi} C$, and Φ'' the CNF defined as $\Phi'' = \bigwedge_{C' \text{ such that } \Phi' \models C'} C'$, where both Φ' and Φ'' are free from tautologies, contradictions and subsumed elements. We have:

- $\Phi \equiv \Phi' \equiv \Phi''$;
- $\forall C \in \Phi'$, C is a prime implicant of Φ;
- $\forall C' \in \Phi''$, C' is a prime implicate of Φ.

Theorem 2. $\Phi' = \boxtimes\Phi$ *and* $\Phi'' = \boxtimes\Phi'$.

Proof: Let us simply observe the following:

- A (minimal) hitting set of the sets in Φ that is not a contradiction is a (prime) implicant of Φ: indeed, if every clause in Φ is satisfied, then Φ is satisfied too.
- Conversely a (minimal) hitting set of the sets in Φ' that is not a tautology is a (prime) implicate of Φ', and thus of Φ too.

■

In summary, this discussion tells us that converting a CNF to a DNF and then back to a CNF produces all the prime implicates of the initial CNF. Consequently, applying twice the $\boxtimes$ operator computes all prime implicates of a formula.

Example 5. Consider again Example 3, with $\Phi = \{xy, \neg xz\}$. We have $\boxtimes\Phi = \{xz, \neg xy, yz\}$, and $\boxtimes \boxtimes \Phi = \{xy, \neg xz, yz\}$. The new clause $y \vee z$ is a prime implicate of $(x \wedge y) \vee (\neg x \wedge z)$.

The last and most important step in our approach is to implement this operator on ZBDDs encoding collections of sets of literals. Let Φ be a collection of sets of literals, and suppose we interpret it as a DNF formula. Let l be a literal involved in Φ. By factorisation of l, Φ is equivalent to $(l \wedge f_l) \vee \bar{f}_l$, where f_l is a DNF formula containing all the sets in Φ involving l, from which l has been removed, and $\bar{f}_l$ is a DNF containing all the sets in Φ that do not involve l. Then $\boxtimes\Phi$ is equivalent to $(l \vee \boxtimes\bar{f}_l) \wedge (\boxtimes\bar{f}_l \vee \boxtimes f_l)$. This formula can then be formulated in CNF by considering the results of the recursive calls $\boxtimes\bar{f}_l$ and $\boxtimes f_l$, which are in CNF by induction hypothesis, and by distributing l over $\boxtimes\bar{f}_l$ and $\boxtimes\bar{f}_l$ over $\boxtimes f_l$, thus reformulating both members of the conjunction in CNF. Finally, $\boxtimes\Phi$ is obtained by removing non-minimal and trivial clauses from this result. Obviously, if we interpret Φ as a CNF formula the corresponding reasoning applies too.

This allows us to represent $\boxtimes\Phi$ as a ZBDD as shown in Figure 4. Note that this ZBDD will be free from subsumed elements, but might contain trivial sets (if $\boxtimes\bar{f}_l$ contains sets containing the literal $\neg l$), but we omit this last aspect in the figure in order to keep it simpler.

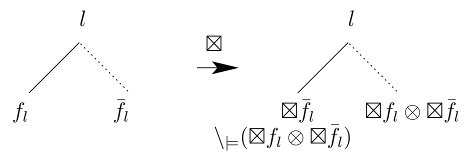
l
f_l
$\bar{f}_l$
$\boxtimes$
l
$\boxtimes\bar{f}_l$
$\backslash_{\models}(\boxtimes f_l \otimes \boxtimes\bar{f}_l)$
$\boxtimes f_l \otimes \boxtimes\bar{f}_l$

Fig. 4. Implementing the $\boxtimes$ operator with clauses

B. Generalisation to Domain Consequences

The operator for the clausal case can be generalised in a straightforward way to domain consequences. Most of the previous discussion holds irrespective of the size of the domains. The only point of attention arises from the definition of trivial sets.

Let $\mathcal{P}$ be a set of domain sequences. So far, we interpreted $\mathcal{P}$ as a set of domain consequences. In that case, we can say that $\mathcal{P}$ is interpreted in conjunctive form, i.e. that $\forall P = \langle D_1, \ldots, D_n \rangle \in \mathcal{P}$, P is interpreted as the disjunction $X_1 \in D_1 \vee \ldots \vee X_n \in D_n$, and $\mathcal{P}$ is interpreted as the conjunction $\bigwedge_{P \in \mathcal{P}} P$ of its domain consequences. Conversely, we say that $\mathcal{P}$ is interpreted in disjunctive form when $\forall P = \langle D_1, \ldots, D_n \rangle \in \mathcal{P}$, P is interpreted as the conjunction $\forall i \leq n, \forall a \in D_i, X_i = a$, and $\mathcal{P}$ is interpreted as the disjunction $\bigvee_{P \in \mathcal{P}} P$. In particular, if $\mathcal{P} = \emptyset$, it interprets as *true* in conjunctive form and *false* in disjunctive form, and if $\mathcal{P} = \{\emptyset\}$, it interprets as *false* in conjunctive form and *true* in disjunctive form.

For a given $P \in \mathcal{P}$, where $\mathcal{P}$ is interpreted in conjunctive form, $P = \langle D_1, \ldots, D_n \rangle$ is a tautology if $\exists i$ such that $D_i = D(X_i)$, i.e. all possible values are allowed for X_i. For a given $P \in \mathcal{P}$, where $\mathcal{P}$ is interpreted in disjunctive form, $P = \langle D_1, \ldots, D_n \rangle$ is a contradiction if $\exists i$ such that $|D_i| > 1$, i.e. X_i must have two distinct values at the same time. Note that the definitions of tautologies and contradictions are equivalent in the case of domains of size 2. This implies that the $\boxtimes$ operator could be defined simply on collections of sets of literals, irrespective of whether they interpret as CNF or DNF. When generalising to domains of arbitrary size, two distinct operators must be defined, taking into account the interpretation of $\mathcal{P}$: one for the conversion from conjunctive to disjunctive form, denoted $\boxtimes_\vee$, and another for the conversion back to conjunctive form, denoted $\boxtimes_\wedge$. At the end of this double conversion, we obtain that $\boxtimes_\wedge \boxtimes_\vee \mathcal{P} = \mathit{Cons}(\mathcal{P})$.

C. The Algorithm

Let M be an ordered automaton free from subsumed elements, with initial state q_0. The two operators $\boxtimes_\vee$ and $\boxtimes_\wedge$ can be described in a general fashion as follows, where either operator is simply denoted by $\boxtimes q_0$.

In order to deal with trivial elements, we need to introduce the following notation.

d) Tautologies.: For each $X_i, 1 \leq i \leq n$, we define the domain sequence $P_{X_i} = \langle D_1^i, \ldots, D_n^i \rangle$, with $D_i^i = D(X_i) \setminus \{\min D(X_i)\}$, and $D_j^i = \emptyset$ otherwise if $j \neq i$. For each $X_i, 1 \leq i \leq n$, we define the state $q^{\wedge}_{X_i}$ as the initial state of the ordered automaton recognising $\{P_{X_i}\}$. For example, suppose $n = 3$, $i = 2$ and $D(X_2) = \{abc\}$, with $a < b < c$. We have $\mathcal{P}(q^{\wedge}_{X_2}) = \{\langle \emptyset, \{bc\}, \emptyset \rangle\}$.

e) Contradictions.: For each $X_i, 1 \leq i \leq n$, we define, for each $a \in D(X_i)$, the domain sequence $P^a_{X_i} = \langle D^i_1, \ldots, D^i_n \rangle$, with $D^i_i = \{a\}$, and $D^i_j = \emptyset$ if $i \neq j$. For each $X_i, 1 \leq i \leq n$, we define the state $q^{\vee}_{X_i}$ as the initial state of the ordered automaton recognising $\{P^a_{X_i} | a > \min D(X_i)\}$. For example, suppose $n = 3$, $i = 2$ and $D(X_2) = \{abc\}$, with $a < b < c$. We have $\mathcal{P}(q^{\vee}_{X_2}) = \{\langle \emptyset, \{b\}, \emptyset \rangle, \langle \emptyset, \{c\}, \emptyset \rangle\}$.

```
Function ⊠$q_0$
if $q_0$ is final then return $q_\emptyset$
else
    $q'_0 \leftarrow$ Create new state
    $\mathit{isFinal}(q'_0) \leftarrow \mathit{true}$
    $q'_0 \leftarrow$ register-state($q'_0$)
    forall $a \in \sigma(q_0)$ in decreasing order do
        $q''_0 \leftarrow$Duplicate $q'_0 \otimes \boxtimes\delta(q_0, a)$
        Let $X_i$ be the variable to which $a$ belongs
        if Operator implemented is $\boxtimes_\vee$ then
            $q'_0 \leftarrow q'_0 \setminus_\models q_{X_i^\vee}$
        if Operator implemented is $\boxtimes_\wedge$ then
            if $a = \min D(X_i)$ then
                $q'_0 \leftarrow q'_0 \setminus_\models q_{X_i^\wedge}$
        $\delta(q''_0, a) \leftarrow (q'_0 \setminus_\models q''_0)$
        $q'_0 \leftarrow$ register-state($q''_0$)
    return $q'_0$
```

Proposition 3. *$\mathcal{P}(\boxtimes_\vee q_0) = \boxtimes_\vee \mathcal{P}(q_0)$ and $\mathcal{P}(\boxtimes_\wedge q_0) = \boxtimes_\wedge \mathcal{P}(q_0)$.*

Proof: Consider first the terminal cases.

If $\mathcal{P}(q_0) = \{\emptyset\}$, q_0 is accepting, and $q_\emptyset$ is returned, and thus $\mathcal{P}(\boxtimes q_0) = \emptyset$.

If $\mathcal{P}(q_0) = \emptyset$, $q_0 = q_\emptyset$. In that case, the **forall** loop is not executed, and q'_0 is returned as such, in which case $\mathcal{P}(\boxtimes q_0) = \{\emptyset\}$.

Consider now the general case, without taking into account trivial element filtering. Figure 5 shows how the operator can be recursively represented by an ordered-automaton, very similarly to the ZBDD representation previously discussed. The invariant that is maintained in the **forall** loop is that $q'_0 = \boxtimes\mathcal{P}_2$ (with $\mathcal{P}_2$ referring to the notation used in Figure 5). It is indeed the case at the first iteration, as $\mathcal{P}_2 = \emptyset$. During an iteration, q''_0 is built according to the method depicted in Figure 5, and at the end of the iteration, the result is assigned to q'_0.

Consider the filtering of trivial elements, which is what differentiates the two actual operators. q'_0 is the state that will be reached by value a. Initially, it is set to $\boxtimes\mathcal{P}_2$. In the case of the distribution to disjunctive form, all elements in $\boxtimes\mathcal{P}_2$ that contain at least one value (but never more than one, by induction hypothesis) belonging to the domain of the same variable X_i must be removed from $\boxtimes\mathcal{P}_2$. This is achieved by applying $q'_0 \setminus_\models q^{\vee}_{X_i}$, as any such element will be subsumed by one of the elements in $\mathcal{P}(q^{\vee}_{X_i})$. On the other hand, in the case of the distribution to conjunctive form, all elements in $\boxtimes\mathcal{P}_2$ that contain all the values but a from of the domain of the same variable X_i must be removed from $\boxtimes\mathcal{P}_2$. This is achieved by applying $q'_0 \setminus_\models q^{\wedge}_{X_i}$, as any such element will be subsumed by the unique element in $\mathcal{P}(q^{\wedge}_{X_i})$. Note that we also have to assume that the automaton returned by $\boxtimes\mathcal{P}_1 \otimes \boxtimes\mathcal{P}_2$, to which q''_0 points in the algorithm, is free from trivial elements. This is the case when the $\boxtimes_\wedge$ operator is considered, as the product operator, as it has been defined, excludes tautologies, but in the case of the $\boxtimes_\vee$ operator, the product operator has to be adapted to exclude contradictions. ■

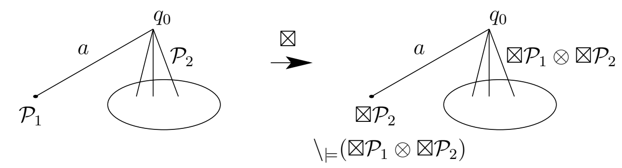
q_0
a
$\mathcal{P}_2$
$\mathcal{P}_1$
⊠
q_0
a
$\boxtimes\mathcal{P}_1 \otimes \boxtimes\mathcal{P}_2$
$\boxtimes\mathcal{P}_2$
$\setminus_\models(\boxtimes\mathcal{P}_1 \otimes \boxtimes\mathcal{P}_2)$

Fig. 5. The general form of the distribution operator on ordered automata

It follows that these operators allow us to compute the closure by resolution of a set of domain consequences encoded by an ordered automaton.

Corollary. $\mathcal{P}(\boxtimes_\wedge \boxtimes_\vee q_0) = \mathit{Cons}(\mathcal{P}(q_0))$.

Concerning the complexity, note that computing the closure by resolution does not have a complexity that is polynomial in the size of the input and output. Indeed, the size of the intermediate automaton in disjunctive form is unrelated to that of the initial and final automaton (and can obviously be exponential in the size of the input). But this is not surprising and cannot be overcome. Indeed, if one could generate all the domain consequences of a problem in a time polynomial in the number of the domain consequences, then this would provide a method to generate all conflicts of a problem in a time polynomial in the number of conflicts. But this contradicts a complexity result in [10], stating that generating all the minimal conflicts or all the maximal relaxations of a problem can only be achieved by incurring the cost related to the number of both (unless *P* = *NP*).

D. Initialising the Compilation

We initialise the procedure by first representing each constraint as an ordered automaton encoding its set of domain consequences, and then combining each of them into a single one. This creates an ordered automaton encoding a set of domain consequences that is equivalent to the problem. Then, the closure by resolution can be applied to infer *all* domain consequences.

More precisely, the set of valid tuples of each constraint can be represented by a classic automaton, regardless of the form of the constraint [11], [10]. Then, we can remark that a classic automaton corresponds to an ordered automaton in disjunctive form, up to a value mapping. For example, the tuple 000 on variables X_1, X_2, X_3, with domain $D(X_i) = \{012\}$ will be mapped to 036. Finally, we apply the $\boxtimes_\wedge$ to the resulting automaton to obtain an ordered automaton representing all the domain consequences of the constraint. Combining the different constraints is simply a matter of using the $\cup_\mu$ operator. The closure by resolution can finally be applied to infer all domain consequences.

VI. An Empirical Study

Although we decided to define ordered automata to represent sets of domain consequences, and to present our algorithms on ordered automata, we chose to implement the procedures using ZBDDs. In fact, ordered automata and ZBDDs are conceptually equivalent, and ordered automata can be implemented in a straightforward way with ZBDDs (both encode collections of sets). This decision allows us to use a mature BDD package, instead of implementing ordered automata manipulation routines on our own. We used JDD [12], a mature and efficient pure Java implementation of BDDs and ZBDDs, based on the well-established BuDDy [13] package.

The objective of the experiments are two-fold. First, to act as a demonstration of our JDD-based implementation in order to show the magnitude of improvement in compilation time over our initial work in this area. Second, to study the behaviour of the compiled structure in terms of the size of the compiled form, the number of domain consequences it encodes, and the compression that is achieved. Therefore, we ran our procedure with the same experimental setup as used in [6], allowing for direct comparison. We generated random uniform constraint networks of binary constraints, with 10 variables, with 10 values, 10 constraints. However, we have to note that, although generated with the same random seed, the random problems were generated on a different machine and architecture. As a result, some instances are different, and are in fact harder for the previous procedure in [6].

We set a cut-off time of one hour, which allowed us to test problems where the tightness of each constraint varied from 1 to 24 allowed tuples, then from 92 to 99. Concerning running times, our new procedure allows us to compute the set of domain consequences of a problem phenomenally faster. For example, the problem with 95 allowed tuples per constraint was the hardest for the procedure in [6], which took 21 hours to complete. Our new prodecure takes merely 166s to compute the same result. With such a substantial difference, we do not see the need to report more detailed times. For further comparison, our previous method only managed to terminate on problems with 1 to 19 allowed tuples, and 95 to 99 allowed tuples, without any time limit. Furthermore, the highest number of domain consequences generated was 251550 for the problem with 95 tuples per constraint with our previous method, against 727195 for the problem with 92 tuples per constraint for the current method.

Some of the generated instances are unsatisfiable (1 to 14 and 22 allowed tuples per constraint). From a compilation point of view, unsatisfiable instances obviously do not present any interest, and quite clearly, our method cannot be as efficient as state-of-the-art methods for proving unsatisfiability. However, we do not necessarily know in advance whether a problem is satisfiable or not, and it is important therefore that our method be robust in this regard. We observe indeed that the performance of our new procedure is mostly affected by the size of the final structure, regardless of the satisfiability of the problem. In contrast, the algorithm in [6] could suffer from the high number of intermediate steps even for unsatisfiable problems, which only have one domain consequence.

Figure 6 and Figure 7 illustrate the compactness of the representation. Figure 6 shows, for each instance, the size of the final ordered automaton (in terms of the number of nodes of the ZBDD encoding it), and the number of domain consequences it encodes. Unsatisfiable instances (1 to 14 and 22 allowed tuples) have only one domain consequence. For most – but not all – instances, the size of the ordered automaton is significantly lower than the number of domain consequences it encodes (note the log scale on the y axis). However, we reach quite a high number of nodes for problems with a high number of domain consequences, which is the real bottleneck for our procedure (as opposed to the problem size).

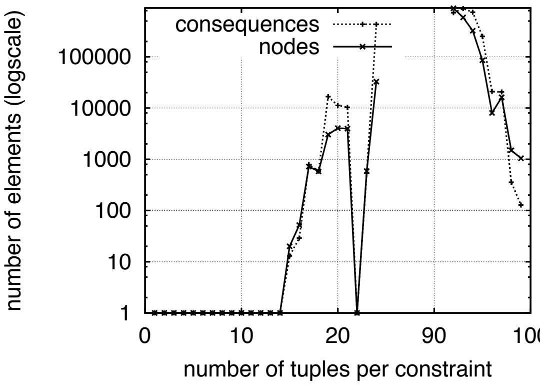

Fig. 6. The final number of consequences and nodes per instance

In order to show in greater detail the compression power of this representation, we plotted in Figure 7 the ratio between the final (i.e. at the end of the procedure) and initial (i.e. at the end of the initialisation) number of consequences, and similarly for the size of the ordered automaton (referred to as the growth on the figure). This number is less than 1 for unsatisfiable instances (as the final number of consequences and nodes is always 1). In order to better illustrate the relationship between those two figures, we also plotted the ratio between those (referred to as compression) in Figure 8. For example, for the problem with 24 tuples per constraint, at the end of the procedure, the number of domain consequences grew almost 50 times more than the size of the ordered automaton that encodes them. Generally, problems with a high number of domain consequences are also the ones with a good compression factor.

It is obvious from those results that the method presented in this paper is still not mature enough to tackle problems of practical significance. However, the phenomenal leap in

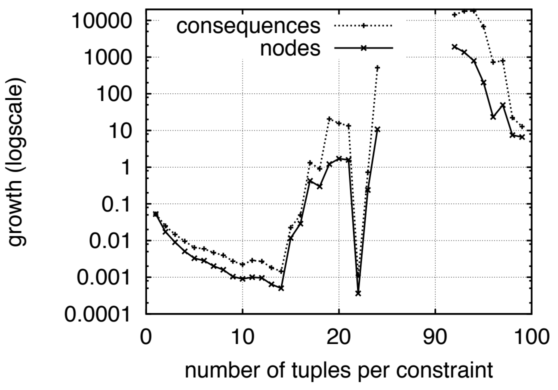

Fig. 7. Growth of number of consequences and nodes per instance

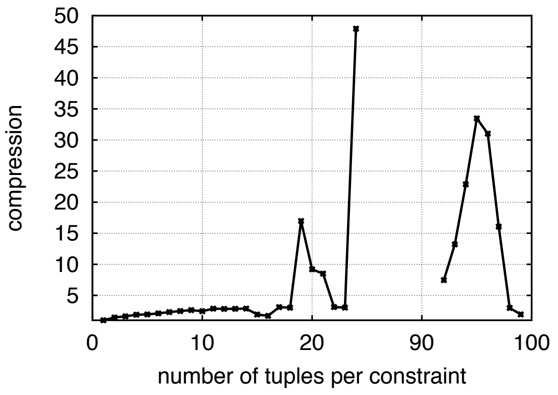

Fig. 8. Ratio between the two growth factors

performance that we achieve over our previous method, which opened a new direction in compilation, strengthens our opinion that we are engaging in a promising research direction.

VII. Conclusions

We presented a data structure and a series of algorithms to represent in a compact way a large set of domain consequences, and that enable to compute the domain consequences of a problem in a more efficient way than with our first method. There exists, to our knowledge, no similar work in the literature, and this is an important step towards a practical way of generating domain consequences. We showed that the main bottleneck lies in the number of domain consequences a problem has. We are currently exploring several directions for fine-tuning our procedures in order to increase the number of domain consequences they can handle. In their experimental study of `zres` [14], Simon and del Val can generate up to 10^{70} clauses [2]. This gives us hope that there is still a lot of room for improvement in our setting too.

In parallel, further study needs to be conducted in order to identify problems that have a low number of domain consequences, and to understand what is their practical interest. In particular, our procedure is very efficient on unsatisfiable problems. Although this is out of the initial motivation of this paper, it could have interesting implications, such as generating explanations for inconsistency. It would also be interesting to structurally characterise problems whose domain consequences are such that they can be more efficiently compressed.

References

[1] P. Marquis, "Consequence finding algorithms," *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, vol. 5, pp. 41–145, 2000.

[2] L. Simon and A. del Val, "Efficient Consequence Finding," in *IJCAI*, B. Nebel, Ed. Morgan Kaufmann, 2001, pp. 359–370.

[3] J. de Kleer, A. K. Mackworth, and R. Reiter, "Characterizing Diagnoses and Systems," *Artif. Intell.*, vol. 56, no. 2-3, pp. 197–222, 1992.

[4] R. Reiter and J. de Kleer, "Foundations of Assumption-based Truth Maintenance Systems: Preliminary Report," in *AAAI*, 1987, pp. 183–189.

[5] B. Selman and H. J. Levesque, "Abductive and Default Reasoning: A Computational Core," in *AAAI*, 1990, pp. 343–348.

[6] A. Papadopoulos and B. O'Sullivan, "Compiling All Possible Conflicts of a CSP," in *CP*, ser. Lecture Notes in Computer Science, I. P. Gent, Ed., vol. 5732. Springer, 2009, pp. 639–653.

[7] S. ichi Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," in *DAC*, 1993, pp. 272–277.

[8] P. Chatalic and L. Simon, "Multi-resolution on compressed sets of clauses," in *ICTAI*. IEEE Computer Society, 2000, pp. 2–10.

[9] T. Hadzic, E. Hansen, and B. O'Sullivan, "On automata, mdds and bdds in constraint satisfaction," in *Proceedings of the ECAI 2008 Workshop on Inference Methods based on Graphical Structures of Knowledge*. Citeseer, 2008.

[10] A. Papadopoulos, "Computing explanations for interactive constraint-based systems," Ph.D. dissertation, University College Cork, December 2011. [Online]. Available: http://hdl.handle.net/10468/510

[11] J. Amilhastre, H. Fargier, and P. Marquis, "Consistency restoration and explanations in dynamic CSPs application to configuration." *Artif. Intell.*, vol. 135, no. 1-2, pp. 199–234, 2002.

[12] A. Vahidi, "Jdd," http://javaddlib.sourceforge.net/jdd/.

[13] J. Lind-Nielsen, "BuDDy: A Binary Decision Diagram library." http://buddy.sourceforge.net.

[14] P. Chatalic and L. Simon, "Zres: The old davis-putman procedure meets zbdd," in *CADE*, ser. Lecture Notes in Computer Science, D. A. McAllester, Ed., vol. 1831. Springer, 2000, pp. 449–454.