Modular Schemes for Constructing Equivalent Boolean Encodings of Cardinality Constraints and Application to Error Diagnosis in Formal Verification of Pipelined Microprocessors

Miroslav N. Velev         Ping Gao
Aries Design Automation, Chicago, IL, U.S.A.
miroslav.velev@aries-da.com

Abstract

We present a novel method for generating a wide range of equivalent Boolean encodings of cardinality, while in contrast all previous Boolean encodings of cardinality have only one form. Experiments for applying this method to automated error diagnosis in formal verification of buggy variants of a complex reconfigurable VLIW processor indicate speedup of up to two orders of magnitude, relative to previous encodings of cardinality. Besides automated debugging of hardware and software, the presented Boolean encodings of cardinality have applications to many other problems.

Introduction

Competition requires companies to develop new microprocessors under short time-to-market periods. It is well known that in the embedded market, a delay of several months can mean the loss of a significant market share, if not the entire market for a new generation of electronic products. Also, design debugging consumes up to 60% of the verification time (Foster 2008), which itself is known to be between 70% and 90% of the time to design a new microprocessor (Anderson and Bhagat 2000).

Correspondence Checking (Velev and Bryant 1999, 2000) is a highly automatic method for formal verification of pipelined processors, based on an inductive correctness criterion using a non-pipelined specification processor to define the correct behavior. By imposing simple modeling restrictions when designing high-level models of pipelined processors, and exploiting the resulting structure of the correctness formulas through the property of Positive Equality (Bryant et al. 2001), Correspondence Checking can scale to formal verification of very complex microprocessors (Velev 2002, 2004; Velev and Gao 2010; Velev and Gao 2011a). The speedup from using Positive Equality is at least 5 orders of magnitude for dual-issue superscalar processors, and is increasing with the complexity of the processor under formal verification (Velev and Bryant 2005). These techniques have been applied at Motorola to formally verify a version of the M.Core pipelined embedded processor and detected 3 bugs (Lahiri et al. 2001). However, previous methods for debugging of incorrect pipelined processors in formal verification with Correspondence Checking (Alizadeh et al. 2010; Velev and Gao 2010) do not guarantee the identification of a minimal set of problems to fix in order to correct a counterexample, and leave it up to the designers to do that.

A major contribution of this paper is a novel method to generate a wide range of equivalent Conjunctive Normal Form (CNF) encodings of cardinality constraints that select at least 1 and at most $k$ out of a set of $n$ domain values. This is done by introducing $k$ single-valued CNF encodings of domains of $n$ values, where each encoding is based on a different set of fresh Boolean variables and selects exactly 1 from the set of $n$ values. Then, a value from the domain is selected to be part of the subset that satisfies the cardinality constraint if and only if the value is selected by at least one of the $k$ encodings. Because of the many ways to define single-valued CNF encodings, e.g., (de Kleer 1989; Iwama and Miyazaki 1994)—see (Velev 2007) for a survey—this method allows us to generate many CNF encodings of cardinality. This is unlike all previous methods for CNF encoding of cardinality that each have only a single form (Bailleux and Boufkhad 2003; Eén and Sörensson 2006; Sinz 2005; Smith et al. 2005; Sulflow et al. 2009).

The contributions of this paper are: 1) a novel method for generating a wide range of equivalent CNF encodings of cardinality constraints; and 2) experimental results indicating speedup of up to two orders of magnitude relative to previous CNF encodings of cardinality, when performing automated error diagnosis in formal verification of buggy variants of a complex reconfigurable VLIW processor.

Background

Correspondence Checking

We use the formal verification method Correspondence Checking—comparing a pipelined implementation processor against a non-pipelined specification (Burch and Dill 1994), using controlled flushing (Burch 1996) to automatically compute an abstraction function, Abs, that maps an implementation state to an equivalent specification state. The safety property (see Fig. 1) is expressed as a formula in the logic of Equality with Uninterpreted Functions and Memories (EUFM), and checks that one step of the implementation corresponds to between 0 and $k$ steps of the specification, where $k$ is the issue width of the implementation. $F_{Impl}$ is the transition function of the implementation, and $F_{Spec}$ is the transition function of the specification. We will refer to the sequence of first
applying \( \text{Abs} \) and then \( \text{F} \text{Spec} \) as the \textit{specification side} of the commutative diagram in Fig. 1, and to the sequence of first applying \( \text{F} \text{Impl} \) and then \( \text{Abs} \) as the \textit{implementation side}. The safety property is the inductive step of a proof by induction, since the initial implementation state, \( \text{Q} \text{impl} \), is completely arbitrary, but possibly restricted by invariant constraints to exclude unreachable states.

![Diagram of a commutative diagram](image)

\textbf{Safety property:}

\[ \text{equality}_0 \lor \text{equality}_1 \lor \ldots \lor \text{equality}_k = \text{true} \]

\textbf{Fig. 1.} The safety correctness property for an implementation processor with issue width \( k \): one step of the implementation should correspond to between 0 and \( k \) steps of the specification, when the implementation starts from an arbitrary initial state \( \text{Q} \text{impl} \) that is possibly restricted by a set of invariant constraints.

In (Velev and Bryant 1999), the style for modeling high-level processors was restricted in order to significantly increase the number of terms (abstracted word-level values in EUFM) that appear only in positive equations. The property of \textit{Positive Equality} (Bryant et al. 2001) allows us to treat such terms as distinct constants, while still performing formal verification. The result is a dramatic pruning of the solution space, and orders of magnitude speedup.

\section*{CNF Encodings of Domains}

Previous work on solving of Constraint Satisfaction Problems (CSPs) by translation to SAT has used many CNF encodings for selecting a domain value from a given domain of \( n \) values. Three such encodings that we use in this paper are:

\textbf{Direct encoding (de Kleer 1989).} A fresh Boolean variable is used to encode the selection of each domain value—see Table 1. An at-least-one constraint is introduced as a CNF clause that is the disjunction of these Boolean variables to ensure that at least one domain value will be selected. For each pair of these Boolean variables, an at-most-one constraint is introduced as a CNF clause that is the disjunction of the negations of these two Boolean variables to ensure that at most one domain value will be selected.

\textbf{Log encoding (Iwama and Miyazaki 1994).} Uses log number of fresh Boolean variables in the size of the domain. Each domain value is selected by a conjunction of literals (i.e., Boolean variables or their negations) corresponding to the binary representation of the number of that domain value—see Table 1. When the domain size is not an exact power of 2, the combinations of values of the Boolean variables that correspond to unused binary numbers are excluded from occurrence by CNF clauses that are each the negation of one unused combination of values. Adding such clauses can be avoided in an optimized version that combines each unused binary number with a used one that differs in the polarity of just one Boolean variable, so that this variable can be eliminated from the conjunction of literals selecting a domain value (Velev 2007).

\textbf{ITE-linear encoding (Velev 2007).} Uses \( n - 1 \) fresh Boolean variables, where each controls one ITE (for “if-then-else”) operator in a chain of \( n - 1 \) ITE operators (i.e., multiplexors) that index the domain of \( n \) values. The first domain value is selected when the first Boolean variable is true—see Table 1. Domain value \( j \) for \( 2 \leq j \leq n - 1 \) is selected when Boolean variables \( 1 \) through \( j - 1 \) are all false and Boolean variable \( j \) is true. Domain value \( n \) is selected when all of the introduced Boolean variables are false.

We will refer to the above as \textit{simple encodings}. For a survey of other simple CNF encodings of domains—see (Velev 2007). A \textit{single-valued CNF encoding} of a domain of \( n \) values selects exactly one domain value in a solution returned by a SAT solver. We will refer to the Boolean variables used in a particular CNF encoding of a domain as \textit{indexing Boolean variables}, and will call the Boolean formula that selects a given domain value \( j \) in an encoding an \textit{indexing Boolean expression} for domain value \( j \), and will denote it \textit{index}(\( j \))—Table 1 illustrates the above encodings.

![Diagram of a hierarchical, recursive, and hybrid encoding](image)

\textbf{Fig. 2.} Example of a hierarchical, recursive, and hybrid encoding. A domain of 12 values, \{\texttt{v9}, \texttt{v1}, ..., \texttt{v11}\}, is divided into three subdomains by Simple Encoding 1 at level 1. Each subdomain is further divided into four parts by Simple Encoding 2 at level 2, where each part is a different domain value.
Table 1. Indexing a domain of four values \(\{0, 1, 2, 3\}\) with the simple encodings: direct, log, and ITE-linear. The digit after each encoding indicates the number of indexing Boolean variables required by that encoding for the given domain. The indexing Boolean variables are denoted by \(i\) with a corresponding subscript.

<table>
<thead>
<tr>
<th>Simple Encoding</th>
<th>Clauses</th>
<th>Indexing Boolean Expressions for Domain ({0, 1, 2, 3})</th>
</tr>
</thead>
</table>
| direct-4        | \(i_0 \lor i_1 \lor i_2 \lor i_3\) | \begin{align*}
\text{index}(0) & := i_0 \\
\text{index}(1) & := i_1 \\
\text{index}(2) & := i_2 \\
\text{index}(3) & := i_3 
\end{align*} |
| log-2           | \(\text{index}(0) := \neg i_0 \land \neg i_1\) | |
| ITE-linear-3    | \(\text{index}(0) := i_0\) | |

Table 2. Indexing a domain of nine values \(\{0, 1, 2, 3, 4, 5, 6, 7, 8\}\) with the 2-level hierarchical encodings direct-3+direct-3, and ITE-linear-2+direct-3, where the + separates the simple encodings for the two levels, which are listed starting with the one for level 1. The first subscript of an indexing Boolean variable indicates the level of the encoding that the variable is part of.

<table>
<thead>
<tr>
<th>Hierarchical Encoding</th>
<th>Clauses</th>
<th>Indexing Boolean Expressions for Domain ({0, 1, 2, 3, 4, 5, 6, 7, 8})</th>
</tr>
</thead>
</table>
| direct-3+direct-3     | \begin{align*}
\text{Level 1:} & \quad \neg i_{1,0} \land \neg i_{1,1} \\
\text{Level 2:} & \quad \neg i_{1,2} \land \neg i_{1,3} \\
\end{align*} | \begin{align*}
\text{index}(0) & := i_{1,0} \land i_{1,2} \\
\text{index}(1) & := i_{1,0} \land i_{1,3} \\
\text{index}(2) & := i_{1,1} \land i_{1,2} \\
\text{index}(3) & := i_{1,1} \land i_{1,3} \\
\text{index}(4) & := i_{1,2} \land i_{1,3} \\
\text{index}(5) & := i_{2,0} \land i_{2,1} \\
\text{index}(6) & := i_{2,0} \land i_{2,2} \\
\text{index}(7) & := i_{2,0} \land i_{2,3} \\
\text{index}(8) & := i_{2,1} \land i_{2,2} \\
\text{index}(9) & := i_{2,1} \land i_{2,3} \\
\text{index}(10) & := i_{2,2} \land i_{2,3} \\
\end{align*} |
| ITE-linear-2+direct-3 | \begin{align*}
\text{Level 2:} & \quad \neg i_{2,0} \land \neg i_{2,1} \land \neg i_{2,2} \\
\end{align*} | \begin{align*}
\text{index}(0) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(1) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(2) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(3) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(4) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(5) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(6) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(7) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\text{index}(8) & := i_{1,0} \land i_{1,1} \land i_{1,2} \land i_{1,3} \\
\end{align*} |

A hierarchical encoding (Velev 2007) for a given domain of \(n\) values can be constructed by using a hierarchy of simple encodings, recursively dividing the given domain into smaller subdomains until a single domain value remains in each subdomain at the lowest level—see Fig. 2. Thus, the indexing Boolean expression for a domain value in a hierarchical encoding is the conjunction of the indexing Boolean expressions for the branches of the simple encodings at each level of the hierarchy that lead to that domain value—see Table 2 for example hierarchical encodings. If more domain values can be selected with a specific hierarchical encoding than the size the given domain, constraints are added to exclude from the solution space the indexing Boolean expressions that do not have a domain value associated. Note that for a given domain size, we can construct many different hierarchical encodings by varying: the number of levels in the hierarchy; the number of branches at each level; and the choice of a simple encoding at each level.

In our previous work, hierarchical encodings resulted in 3 orders of magnitude speedup when solving graph-coloring problems (Velev 2007), 4 orders of magnitude speedup when solving FPGA detailed routing problems (Velev and Gao 2008), 4 orders of magnitude speedup when solving Hamiltonian cycle problems (Velev and Gao 2009), and 8 orders of magnitude speedup when solving routing problems for optical networks (Velev and Gao 2011b). Thus, the motivation to use them for efficient encoding of cardinality constraints in this paper.
Previous CNF Encodings of Cardinality

Cardinality constraints express the condition that between 1 and \( k \) (alternatively exactly \( k \), or in other cases between 0 and \( k \)) domain values from a domain of size \( n \) be selected. We next summarize previous CNF encodings of cardinality by introducing a cardinality Boolean variable \( s_j \), \( 1 \leq j \leq n \), that if true indicates that domain value \( j \) is selected.

**Naïve encoding.** It is defined as a conjunction of all clauses of \( k + 1 \) negated variables \( s_j \) to represent that from each combination of \( k + 1 \) variables at least one is false.

**Sequential counter (Sinz 2005).** A chain of counter cells is defined, where each takes as input one variable \( s_j \) and a unary representation (where an integer \( i \) is represented with 1s in the first \( i \) bits and 0s after that) for the number of variables \( s_j \) that are true in any of the previous counter cells—see Fig. 3.a. Each cell increments its input unary value if the cell’s input variable \( s_j \) is true, and also sets an overflow output \( v_i \) to true if the cell’s partial sum is greater than \( k \). Each output \( v_i \) is constrained to be false for \( i > k \).

**Adder encoding (Smith et al. 2005).** A tree of adders is used to recursively sum the variables \( s_j \)—see Fig. 3.b. At each level of the tree, the adders have bit-width equal to that level in the tree, i.e., there are \( n \) adders of bit-width 1 at level 1, \( n/4 \) adders of bit-width 2 at level 2, and so on. A comparator for less-than-or-equal-to \( k \) with its output set to 1 in the last level enforces the cardinality constraint. Variants of this encoding are used in (Bailleux and Boufkhad 2003; Sinz 2005).

**ITE encoding (Eén and Sörensson 2006).** This encoding is derived from a BDD (Bryant 1986) of a cardinality constraint. ITE operators that are each controlled by a variable \( s_j \) are used to count the number of variables \( s_j \) that are 1—see Fig. 3.c. This is done by building an ITE graph that exploits maximal sharing of common subexpressions and detects conflicts—the output of this graph is connected to 1, and the then-input of an ITE is connected to 0 when \( k + 1 \) variables \( s_j \) have been 1 so far, thus creating a conflict that will be detected by the SAT solver, which will then analyze the reason for the conflict and will backtrack. This encoding can be viewed as a decomposed representation of the naïve encoding.

**Shifter encoding (Sulflow et al. 2009).** A set of \( k \) shifters of bit-width \( n \) are introduced, where each shifts a constant 1 within an \( n \)-bit bit-vector with a shift offset determined by another \( n \)-bit bit-vector consisting of fresh Boolean variables—see Fig. 3.d. A tree of \( n \)-bit OR operators combines the outputs of all shifters and produces a bit-vector of size \( n \), where at least 1 and at most \( k \) bits are 1, and the value of \( j \) is assigned to variable \( s_j \).

A CNF cardinality encoding for the specific case that at most 1 value is selected from a domain of \( n \) values is discussed in (Marques-Silva and Lynce 2007).

**Our CNF Encoding of Cardinality**

We present a novel method for generating a wide range of equivalent CNF encodings of cardinality constraints, each defined by combining a set of simple, or hierarchical, or a mixture of simple and hierarchical CNF encodings that index the same domain of \( n \) values.

**Definition 1**—CNF encoding of cardinality constraints to select at least \( k \) and at most \( k \) out of a set of \( n \) domain values. Let \( k \) single-valued CNF encodings of domains of \( n \) values be introduced, where each encodings is based on a different set of fresh Boolean variables. Let \( \text{index}(i, j) \) be the indexing Boolean expression used in encoding \( i \) to select domain value \( j \), for \( 1 \leq i \leq k \) and \( 1 \leq j \leq n \). Then, let \( s_j \) be a Boolean variable that is introduced in the given CNF cardinality encoding to be true iff domain value \( j \), \( 1 \leq j \leq n \),
is among the subset of domain values selected by the encoding, and is defined:

\[ s_j := \prod_{i=1}^{k} \text{index}(i, j) \]

**Lemma 1.** Defining \( s_j \), \( 1 \leq j \leq n \), as in Definition 1 results in a correct CNF encoding of cardinality of at least 1 and at most \( k \) for a set of \( n \) domain values.

**Proof.** Follows by the construction of the CNF encoding of cardinality in Definition 1. Namely, \( s_j \) will be true iff domain value \( j \) is selected by at least one of the \( k \) encodings. Since the encodings are single-valued, each encoding selects just one domain value for a given assignment of values to all Boolean variables in that encoding. Since there are no constraints to ensure that each of the \( k \) encodings selects a different domain value, then in the case when all \( k \) encodings select the same domain value there will be just one selected domain value, and when each encoding selects a different domain value then there will be \( k \) selected domain values. Thus, at least 1 and at most \( k \) different domain values will be selected out of a domain of \( n \) values.

We similarly define the CNF encodings of cardinality constraints for the condition that exactly \( k \) out of \( n \), or that between 0 and \( k \) out of \( n \) domain values be selected:

**Definition 2—CNF encoding of cardinality constraints to select exactly \( k \) out of a set of \( n \) domain values.** This cardinality encoding is defined by extending Definition 1 with constraints that no two of the introduced single-valued encodings select the same domain value, i.e., \(-\text{index}(i, j) \lor -\text{index}(l, j), \) for \( i \neq l, 1 \leq i, l \leq k, \) and \( 1 \leq j \leq n \) (thus ensuring that exactly \( k \) different domain values will be selected).

**Definition 3—CNF encoding of cardinality constraints to select between 0 and at most \( k \) out of a set of \( n \) domain values.** This cardinality encoding is defined by extending Definition 1 such that each of the \( k \) encodings indexes a domain of \( n + 1 \) values, where the extra domain value represents that none of the original \( n \) domain values is selected (so that when all of the \( k \) encodings select this extra domain value then there will be 0 of the original domain values selected, and otherwise at most \( k \) of the original domain values will be selected when each encoding selects a different domain value from the original domain).

The proofs of correctness for the encodings from Definitions 2 and 3 are similar and are omitted. Other types of cardinality constraints can be defined similarly.

The CNF encoding of cardinality from Definition 1 introduces \( nkl + 2nk + n + c \) clauses, where \( l \) is the average number of indexing Boolean variables in an indexing Boolean expression for the \( k \) CNF encodings of domains that are used, and \( c \) is the number of clauses that represent additional constraints required for the construction of those encodings. The number of introduced CNF variables is \( nk + kp \), where \( p \) is the average number of indexing Boolean variables in the \( k \) CNF encodings of domains that are used. However, while previous complexity analysis for CNF cardinality encodings has focused only on the number of CNF clauses and variables used (Bailleux and Boufkhad 2003; Sinz 2005), note that the ultimate measure of efficiency is the execution time.

Note that Definitions 1 – 3 allow us to define many equivalent Boolean representations of cardinality constraints for given values of \( k \) and \( n \). First, the introduced \( k \) encodings \( (k + 1 \) in Definition 3) can be all of the same type—either simple or hierarchical, but with the same structure—or of different types—again either simple or hierarchical, and some or all of them having different structure. Second, when some or all of the encodings are different, those that are hierarchical can each have a different number of levels, and/or a different number of branches at each level, and/or a different simple encoding used at each level. Thus, our method allows us to exploit many alternative reformulations of a problem requiring cardinality constraints. In contrast, all of the previous methods (Bailleux and Boufkhad 2003; Eén and Sörensson 2006; Marques-Silva and Lynce 2007; Sinz 2005; Smith et al. 2005; Sulflow et al. 2009) produce only one CNF representation of cardinality constraints for given values of \( k \) and \( n \).

**Results**

We conducted experiments on a workstation with a six-core 3.33-GHz Intel Xeon processor, 24 GB of 1,333-MHz memory, and Red Hat Enterprise Linux v5.5. (Only one core was used for each experiment.) We applied our industrial tool flow, combined with a proprietary SAT solver that is faster than the best publicly available SAT solvers by at least a factor of 2. We used 25 buggy variants of a 16-stage, 9-wide VLIW processor that imitates the Intel Itanium (Intel 1999) in features such as predicated execution, register remapping, advanced and speculative loads, branch prediction, multicycle functional units, exceptions, and an 8-entry instruction queue. We also modeled reconfigurable functional units (Velev and Gao 2011a). (A simpler version of this processor with fewer pipeline stages and no reconfigurable functional units was formally verified in (Velev and Bryant 2005).)

When proving safety of the correct implementation processor, the CNF formula for the correctness condition has approximately 2.8M variables, 48M literals, and 14M clauses, and the proof of correctness takes 6,234 s. The controlled flushing (Burch 1996) for this processor takes 23 clock cycles. We adapted the existing Boolean Satisfiability (SAT) based error-diagnosis methods (Mizraei et al. 2008; Smith et al. 2005; Sulflow et al. 2009) in order to identify a minimal set of control signals in a buggy processor that have to be inverted in order to correct a counterexample. This is done by automatically inserting a repair multiplexer (MUX) for each control signal, such that this repair MUX is controlled by a different fresh Boolean variable in the one clock cycle of regular symbolic simulation and the 23 clock cycles of controlled flushing along the implementation side of the commutative correctness diagram in Fig. 1, for a total of 12,984 such Boolean variables. When the Boolean variable controlling a repair MUX is true, that MUX selects the
inverted value of the original signal, and otherwise the actual value of the original signal. Cardinality constraints are introduced for the Boolean variables controlling repair MUXes in order to enforce that only \( k \) of these Boolean variables are true, such that \( k \) is gradually increased, until finding the minimal \( k \) that repairs the counterexample. Table 3 compares the different cardinality encodings on repairing two of the buggy processors. The statistics for the other 23 buggy variants are similar.

Table 3. Results from repairing 2 buggy processors (the statistics for the other 23 are similar).

<table>
<thead>
<tr>
<th>Buggy Processor</th>
<th>Cardinality Encoding</th>
<th>Extra Variables &amp; Clauses from Cardinality Added to Simplified Repair CNF Formula from Step 2</th>
<th>Time to Find Solution for ( k ) [s]</th>
<th>Time to Prove No Solution for ( k - 1 ) [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Indexing Variables</td>
<td>Total Variables</td>
<td>Clauses</td>
</tr>
<tr>
<td>Bug 1 ( k = 6 ) ( n = 12,984 )</td>
<td>Sequential Counter</td>
<td>———</td>
<td>90,888</td>
<td>350,562</td>
</tr>
<tr>
<td></td>
<td>Adder</td>
<td>———</td>
<td>39,005</td>
<td>182,063</td>
</tr>
<tr>
<td></td>
<td>ITE</td>
<td>———</td>
<td>90,845</td>
<td>363,384</td>
</tr>
<tr>
<td></td>
<td>Shifter</td>
<td>———</td>
<td>13,068</td>
<td>807,768</td>
</tr>
<tr>
<td></td>
<td>(direct-3)*, ( o_3 )</td>
<td>162</td>
<td>13,146</td>
<td>464,876</td>
</tr>
<tr>
<td></td>
<td>ITE-linear-5+(direct-3)*, ( o_3 )</td>
<td>156</td>
<td>13,140</td>
<td>326,838</td>
</tr>
<tr>
<td>Bug 2 ( k = 22 ) ( n = 12,984 )</td>
<td>Sequential Counter</td>
<td>———</td>
<td>298,632</td>
<td>1,181,522</td>
</tr>
<tr>
<td></td>
<td>Adder</td>
<td>———</td>
<td>39,053</td>
<td>182,239</td>
</tr>
<tr>
<td></td>
<td>ITE</td>
<td>———</td>
<td>298,125</td>
<td>1,192,504</td>
</tr>
<tr>
<td></td>
<td>Shifter</td>
<td>———</td>
<td>13,292</td>
<td>2,927,192</td>
</tr>
<tr>
<td></td>
<td>(direct-3)*, ( o_2 )</td>
<td>594</td>
<td>13,578</td>
<td>1,669,188</td>
</tr>
<tr>
<td></td>
<td>ITE-linear-5+(direct-3)*, ( o_3 )</td>
<td>572</td>
<td>13,556</td>
<td>1,163,782</td>
</tr>
</tbody>
</table>

In Table 3, (direct-3)* indicates a hierarchical encoding that is recursively composed of applications of the simple encoding direct-3 at each level of the hierarchy. We found two of our cardinality encodings to have the best performance—those based on Definition 1, where each encoding of domains is of the same type, such that ITE-linear-5+(direct-3)* was the best, closely followed by (direct-3)*. Table 3 shows the extra CNF variables and clauses added to the simplified repair CNF formula that already has approximately 0.9M variables and 5.4M clauses. Note the very dramatic reduction of the solution space, when the search is restricted to only the indexing CNF variables in our encodings, compared to the total number of variables of approximately 0.9M plus the extra ones from the encodings. As a result, we achieved speedup of 2 orders of magnitude for satisfiable CNF formulas—see Bug 2 and the column with times to find solution for \( k \)—and at least an order of magnitude for unsatisfiable formulas (when proving that there is no solution for \( k - 1 \)). Also, based on additional experiments that are not shown, the speedup from our encodings is increasing with the size of the domain of the cardinality constraints.

**Conclusion**

We proposed a novel method for generating a wide range of equivalent CNF encodings of cardinality constraints. In contrast, all previous Boolean encodings of cardinality have only one form. Our method is general and applicable to solving any problem that requires cardinality constraints and that can be formulated as an equivalent SAT problem. We used these cardinality encodings for error diagnosis in formal verification of buggy variants of a complex reconfigurable VLIW processor, and achieved speedup of up to two orders of magnitude relative to previous CNF encodings of cardinality.

**References**


