A Curious New Result of Resolution Strategies in Negation-Limited Inverters Problem

Ruo Ando, Yoshiyasu Takefuji

***Abstract*—Generally, the negation-limited inverters problem is a puzzle of constructing an inverter with AND gates and OR gates, and a few inverters. This paper introduces a curious new result about the effectiveness of two powerful ATP (Automated Theorem Proving) strategies in tackling negation-limited inverter problems. Two resolution strategies are UR (Unit Resulting) resolution and Hyper-Resolution. In the experiment, we cope with two kinds of automated circuit construction: three input/output inverters and four input/output BCD Counter Circuit. Both circuits are constructed with a few limited inverters. Curiously, it has been turned out that UR resolution is drastically faster than Hyper-Resolution in the measurement of the size of SoS (Set of Support). Besides, we discuss the syntactic and semantic criteria, which might cause a considerable difference in computation cost between UR resolution and Hyper-Resolution.**

***Index Terms*—Negation-limited inverters problem, automated theorem proving, unit resulting resolution, Hyper-Resolution, set of support**

I. Introduction

A circuit with outputs $\neg x_1, \neg x_2, ..., \neg x_n$ for any Boolean inputs $x_1, x_2, ...x_n$ is called as an inverter. Here, we consider the automated construction of some circuits with AND gates and OR gates and a few NOT gates. This is also called a knotty problem in [5]. Before [5], Larry Wos [4] introduced "two-inverter puzzle" in an article on Automated reasoning. It is easy to construct an inverter by putting n* $Notgates$ in a row. In the two-inverter puzzle, we have constrained that an inverter uses fewer than n NOT gates and constructs an inverter with an arbitrary number of AND gates and OR gates. Sheldon [7] proves that [log(n+1)] NOT gates are necessary and sufficient to construct the inverter

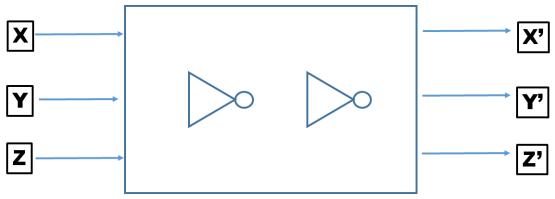

Fig. 1. Two inverters puzzle as negation-limited inverters problem.

Figure 1 depicts the 3-input/3-output version of this problem called a two-inverter puzzle. The black box in the middle of the figure receives binary inputs [0,1] at the terminal of x, y, and z. Each output terminal of x', y', and z' yields the complement of the corresponding input. For example, if x is 0, x' is 1, and so on. Each of the eight possible 3-bit input words yields its complementary expression at the outputs for three terminals. Usually, without any constraint about negation-limited inverters, such a transfer function would be implemented by using three inverters or NOT-gates connected between input and output.

In formal, this puzzle is designed as figuring out a network using any number of AND and OR gates but not more than two (2) NOT's to achieve exactly the same input-output function. The AND's and OR's may have as many inputs as required.

To discuss a curious new result of this paper mainly concerning the two-inverter puzzle, we define the broader problem as follows;

Definition 1. Negation limited inverters problem. As a generalization of the two-inverter puzzle, we define the negation limited inverters problem as the construction of an arbitrary N-bit input / N-bit output circuit with limited inverters of N-1.

In [7], it was proved that the complete set of input variables might be inverted D(n) inverters where D(n) is the small integer y such that $n < 2^y$. According to this proof, the negation limited inverters problem, including two inverter puzzles, also includes the constructing BCD (decade counter circuit) with two inverters [1] discussed in the following section.

Programs for the experiment in this paper are available at [2].

II. OTTER

A. OTTER and its clause sets

In this paper, we use OTTER (Organized Techniques for Theorem-proving and Effective Research), which is the product of Argonne National Laboratory, implemented mainly by W.McCune. Before OTTER was released, E. Lusk, R. Overbeek, et al. [18] proposed the main concepts of the theorem prover. OTTER became popular as the most powerful automated reasoning tool thanks to the works for [12] [13] [14] tackling certain classes of the problem, including negation-limited inverters problem. The fundamental reasoning framework of OTTER is a given-clause algorithm for processing a set of support[10].

The given clause algorithm uses two sets for retaining clauses. One is the set of support (SoS). The reasoning process of OTTER starts with the retaining clause in a support set which has all the chosen input clauses. In the reasoning process, OTTER retains always the initial set of support and the generated clauses. Another set is the usable set. In the first phase, the usable set is separated from the

Manuscript received September 28, 2021; revised June 15, 2022.

Ruo Ando is a project associate professor of Center for Strategic Cyber Resilience Research and Development, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan 101-8430 (email: ruo@nii.ac.jp).

Yoshiyasu Takefuji is a professor of Faculty of Data Science, Musashino University, 3-3-3 Ariake, Koto-Ku, Tokyo, Japan 135-8181 (email: y-take@musashino-u.ac.jp).

set of support. OTTER puts the usable clauses into the set of support for focusing the attention on yielding the additional clauses.

All generated clauses are divided into four classes: SoS, Usable, Passive, and Demodulators. OTTER starts with the retention of a support set, including all of the chosen input clauses. In detail, OTTER handles the four kinds of clauses in the reasoning process as follows:

1) SoS. Clauses are regarded as facts to be kept for participating in the search. They are not picked to make inferences.
2) Usable. Usable is a list of inference rules for keeping clauses available to the reasoning.
3) Passive. This list is for the Unit conflict and forward subsumption. The reasoning program uses the passive list to check if the clauses with opposite sign of the passive list are generated. The passive list does not change and participate in the reasoning.
4) Demodulators. Demodulators are adopted to rewrite newly retained clauses with equational reasoning.

In the experiment, we particularly observe the size of the set of support. For our research, a set of support is the most important indicator for characterizing the reasoning process.

B. Given Clause algorithm

The given clause algorithm enables the program to attempt all combinations from axioms derived from the given clause. From another perspective, the clause combination is yielded from the given clause, which has been focused on. Algorithm 1 shows the procedures of the given clause algorithm. It takes a set of support and a usable list discussed in the previous section.

Algorithm 1 Given clause algorithm

Input: SoS, Usable List
Output: Proof

```
1: while until SoS is empty do
2:    choose a given clause G from SoS;
3:    move the clause g to Usable List;
4:    while  c_1, ..., c_n in Usable List do
5:       while R(c_1, ..c_i, G, c_{i+1}, ..c_n)exists do
6:          A ⇐ R(c_1, ..c_i, G, c_{i+1}, ..c_n);
7:          if A is the goal then
8:             report the proof;
9:             stop
10:         else {A is new odd}
11:            add A to SoS X
12:         end if
13:      end while
14:   end while
15: end while
```

In line 2, the prover picks up G (given clause) from SoS (Set of Support). Two while loops are started in lines 4 and 5 to attempt any and all combinations extracted from the given clause and usable list. Readers are encouraged to check [11] [19] for the basic design of given clause algorithm. In a nutshell, five steps are taken in the given clause algorithm as follows:

1) Choose a clause as the given clause from the clause list of the set of support.
2) Append the given clause to the usable list.
3) Using the inference rule or rules for the inference of all clauses.
4) Test the retention of newly inferred clause
5) Add each yielded new clause to the SoS.

OTTER selects a clause G from the clause set which has been focused on in SoS. In this sense, clause G is called a given clause or focal clause.

III. Resolution strategy

A. Hyper-Resolution

In a basic situation, researchers use the Hyper-Resolution inference rule for formulating a problem. Hyper-Resolution takes a positive clause which is called the nucleus. Simultaneously, satellites that have negative literal are combined with the nucleus. Hyper-Resolution enables the prover to handle large clauses as a sequence of binary resolution for yielding a single positive clause. Robinson has founded the concept of Hyper-Resolution in [15].

Definition 2. Hyper-Resolution. The inference based on hyper-Resolution handles a clause that contains a set of clauses $A[i]$ which contains only positive literals and at least one negative literal simultaneously. When it succeeds, Hyper-Resolution outputs a clause B which has only one positive literal. When successful, clause B is generated by discovering an MGU (most general unifier) denoted as σ.

In Hyper-Resolution, a positive literal is unified in each of the $A[i]$ by MGU (σ) with a negative literal NL. The clauses NL and $A[i]$ should have no variables in common. The general scheme is:

$$\frac{\begin{array}{l} K_{1,l}, ., K1, n \\ ... \\ Km, l, ., Km, n \\ \{\neg L_1, ., \neg L_{m+1}, L_l\}\exists\sigma.\sigma \\ = mgu([|K_1|, ., |K_{m,1}|], [|L_1|, ., |L_m|]) \end{array}}{\{K_{1,2}, ., K_{1,n}, K_{m,2}, .K_{m,n}, L_{m+1}, ., L_l\}\sigma}$$

In the list above, K_i denotes a clause, and L_i denotes a literal. Hyper-Resolution is applied to a set of m unit clauses K1 ... Km and a single nucleus L1, ..., Lm+1 consisting of m + 1 literals.

Roughly corresponding to OTTER's syntax, the "if-then" clause may have conclusion literals with OR operator as follows:

```
1: If P & Q, then R | S
```

deduces

```
1: -P | -Q |-R | S
```

Then, the clash process occurs under the hypothesis literals in the "if-then" with more than one literal. A typical pattern might be as follows:

```
1: -P | -Q |-R | S
2: P | T
```

```
3: Q | W
4: R
5: -------
6: T | W | S.
```

Importantly, Hyper-Resolution requires that all of the negative literals in the "if-then" clause have clashed with corresponding literals in other clauses. For example, from

```
1: -P(x,y) | -Q(x) |-R(x,y)
2: P(z,b)
3: Q(a)
```

Hyper-Resolution deduces

```
1: R(a,b).
```

Hyper-Resolution is the most frequently adopted inference rule in situations in the case that we should not apply equality substitutions. Hyper-Resolution is intuitively natural to human reasoning.

According to [18], "Don't draw any conclusions until all of the hypotheses are satisfied" is the restriction which all negative literals should clash.

In general, for a broad class of reasoning problems, Hyper-Resolution is sufficient. The rule most resembles the inference mechanism used in deduction systems. Also, In OTTER, Hyper-Resolution is the default inference rule.

B. UR-Resolution

In the UR-Resolution (unit-resulting resolution) [16], a clause set is divided into two categories, a non-unit clause, and a unit clause. The non-unit clause is called the nucleus, and the unit clause is called the satellites. The satellites (unit clause) is resolved all except one of its literals with unit clauses. Positive UR-Resolution yields a positive unit clause, while the negative UR-Resolution yields a negative unit clause.

The general scheme is:

$$\frac{\begin{array}{l}K_1\\ \ldots\\ K_m\\ \{L_1,.,L_{m+1}\}\exists\sigma.\sigma = mgu([|K_1|,.,|K_m|],[|L_1|,.,|L_m|])\end{array}}{\{L_{m+1}\}\sigma}$$

K_i denotes a clause in the list above, and L_i denotes a literal.

UR-Resolution inference rules take a set of m unit clauses $K1...Km$ and a single nucleus $L1,...,Lm+1$ consisting of $m+1$ literals. Here, K_i, L_i, then $L_{m+1}\sigma$ is called as the unit resulting resolvent. All pairs of literals K_i, L_i should be complementary in the general scheme. K_i, L_i are assumed to have opposite signs. Because $|K1|$ denotes the atom contained in the literal $K1$, the reasoning process of the simultaneous unifier avoids the signs of the literals.

Definition 3. UR-Resolution. UR-Resolution takes each literal to be removed from the nucleus. Then, taken literals are unified with a unit satellite. In UR-Resolution, both negative and positive resolvents are supported. UR-Resolution is not refutation complete. However, UR-Resolution is refutation complete in coping with horn clause sets.

Hyper-Resolution will reach out to the derivation with only positive literalism. It is sufficient in coping with a large clause of problem. Instead of avoiding all restrictions on all clauses to be derived, UR-Resolution considers the possibility of clauses containing a single literal. Clauses with a single literal are called unit clauses or "units". In UR-Resolution, a unit clause can be described as a statement of fact. On the other hand, multi-literal clauses represent conditional statements in the case that the multi-literal clauses contain both positive and negative literal. Consequently, unit clauses are practical in many situations. UR-Resolution discards the restriction that derived clauses should have only positive literals. In other words, UR-Resolution imposes the restriction that the derived clauses should be units.

```
1: -P | -Q | R
2: P
3: -R
```

UR-Resolution derives -Q. Note that Hyper-Resolution would be unable to derive anything from the contrast. Besides, UR-Resolution focuses on a unit in a way that all but one of the clauses participate in the deduction. Those clauses should be unit clauses, although they can be either positive or negative. Broadly, UR-Resolution focuses on unit clauses, whereas Hyper-Resolution emphasizes positive clauses.

C. Set of Support

In [10], the set of support strategy is illustrated. It guides the reasoning program to select from the clauses characterizing the question under research to be put in the list of the set of support, which is denoted as $list(SoS)$ in OTTER. The corresponding restriction evades adopting a reasoning rule to a set of clauses of which all clauses are derived from the set of support. Consequently, each clause generated and retained is appended to $list(SoS)$. In [10], experimentally, it is pointed out that the best choice in effect for the initial set of support is based on the unique hypothesis and the denial of the theorem under research. The second best choice is the denial of the theorem itself.

Definition 4. Set of Support Strategy. Let S be any nonempty set of clauses. Let U be any inference rule which is the usable list. A nonempty subset of U of S is required under the set of support strategy. Let U_0 be the clause set D. We get C is in D. Or D is a factor of a clause C in T. D is obtained by using T to the clauses $C_1, C_2, ...C_n$ with one of C_j in T_0 at least. C_k are not in T_0. And D is a factor of a clause in T_1.

In the negation limited inverters problem (two-inverter puzzle), SoS list contains the clauses such that the input signals are constructible.

Listing 1. Set of Support list

```
P(00001111, v). input 1
P(00110011, v). input 2
P(01010101, v). input 3
```

We then add a statement in the negative form for the target output state. The clauses as follows represent that one output pattern at least cannot be yielded.

Listing 2. Usable list

```
-P(0000000110, v) |
-P(0001111000, v) |
-P(0110011000, v) |
-P(1010101010, v).
```

The reasoning program is terminated when the unit conflict occurs. Unit conflict is an inference rule that derives a contradiction from unit clauses. For example, unit conflict occurs between P(a,b) and -P(x,b). Theoretically, unit conflict is based on proof by contradiction.

IV. Experiment

In experiment, we use workstation with Intel(R) Xeon(R) CPU E5-2620 v4 (2.10GHz) and 251G RAM.

A. Tracking the size of the set of support

The main loop of the inference and processing clauses operates both on Usable and SoS lists based on the given clause algorithm.

1) Choose an appropriate given clause from SoS.
2) Move selected given clause from $list(SoS)$ to $list(usable)$.
3) Process a newly appended clause ($given\_clause$) with the inference rules set in the usable list.
4) Inspect whether newly retained clauses have the $given\_clause$ or not.
5) Execute the retention test on newly retained clauses and append those clauses to $list(SoS)$.

The main loop is depicted in Algorithm 3 for searching for a refutation.

Algorithm 2 Tracking the size of the set of support

```
1: while given clause is NOT NULL do
2:    index_lits_clash(giv_cl);
3:    append_cl(Usable, giv_cl);
4:    if splitting() then
5:       possible_given_split(giv_cl);
6:    end if
7:    infer_and_process(giv_cl);
8:    giv_cl = extract_given_clause();
9:    track(SoS_size);
10:   track(generated_clauses);
11: end while
```

In line 9, we track the size of the set of support for each iteration step. After line 8, where the clause is selected from SoS (set of support), we track the size of the set of support in each interaction step. In the next section, we show the plot with the iteration step of X-axis and the size of the SoS on Y-axis.

B. Two inverter puzzle

As an example of a logic circuit design problem, we consider the two-inverter puzzle. Let us consider any number of AND, OR gates but no more than two NOT gates. Under this constraint of no more than two inverters, we are to construct a combinational circuit with three inputs {i1, i2, i3} and three outputs {o1, o2, o3}. Thus, the logic circuit to be constructed is represented as: o1=NOT (il), o2=NOT (i2), o3=NOT (i3). Table I shows the input/output diagram of the two-inverter puzzle.

TABLE I
Two inverter puzzle

3-INPUTS			3-OUTPUTS			
x1	0	0	0	1	1	1
x2	0	0	1	1	1	0
x3	0	1	0	1	0	1
x4	0	1	1	1	0	0
x5	1	0	0	0	1	1
x6	1	0	1	0	1	0
x7	1	1	0	0	0	1
x8	1	1	1	0	0	0

The first is that the circuit has three inputs.

$P(x1, x2, x3, x4, x5, x6, x7, x8)$. - (1)

This clause (1) says OTTER can construct a circuit yielding the output P(x1...x8). The complexity of this puzzle is how to keep track of the number of inverters used (only two are allowed in this case). We can use a list for the notation. A variable is used to enable a shortlist to subsume longer lists in the list. For instance, the pattern (0,0,0,0,1,1,1,1) can be generated with no inverters because it is one of the input signals.

$P(0, 0, 0, 0, 1, 1, 1, 1, v)$. - (2)

If an output signal pattern can be yielded using a few inverters, then it does not matter if the same outputs can be generated using one or more inverters. If the output is inverted, one more inverter is appended to the list, which can be described as:

$P(1, 1, 1, 1, 0, 0, 0, 0, L(inv(1, 1, 1, 1, 0, 0, 0, 0), v))$. - (3)

The inverter is denoted by the $inv()$ term. The term $inv()$ represents the pattern of the output of the inverter. In the case that another inverter is used for the resulting output pattern by OTTER, this is represented as:

```
P(0,0,0,0,1,1,1,1,
L(inv(0,0,0,0,1,1,1,1),
L(inv(1,1,1,1,0,0,0,0),v)). - (4)
```

This clause (4) would be subsumed by the preceding two clauses (2) (3) immediately. And then, two clauses (2)(3) are subsumed by (1) because the first clause (1) has the same pattern but has the empty inverter list.

Figure 2 shows the size of the set of support during the reasoning process of Hyper-Resolution. X-axis is the iteration step. Y-axis is the size of the set of support. The size of SoS increases around the iteration step of 800. Then, it increases slowly in the next about 4,000 steps. After plateau from iteration step 1000 to 4700, the SoS size increase speeds up until iteration step 8000.

Figure 3 depicts the size of the set of support during the reasoning process of UR-Resolution. The set size begins to increase rapidly about the iteration step of 600. The speed of increase slows down around the iteration step around 1100.

C. BCD or Decade Counter Circuit

A binary coded decimal (BCD) is a serial digital circuit designed for counting ten digits. BCD resets for every new input from the clock. BCD is also called a “Decade counter” because BCD can go through 10 unique combinations of

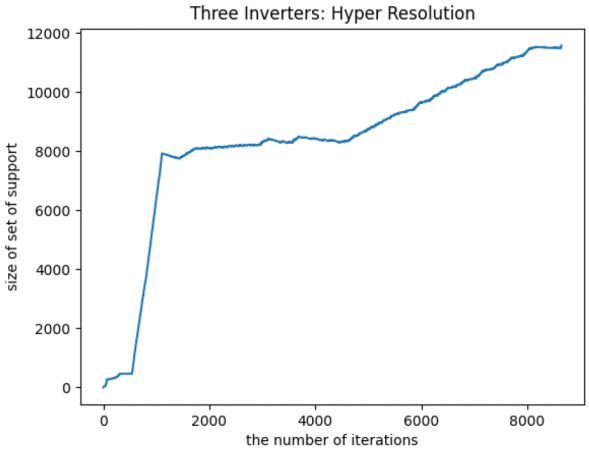

Fig. 2. Hyper-Resolution in constructing 3-input/3-output inverter. X-axis of the number of iteration steps. Y-axis is the size of the set of support.

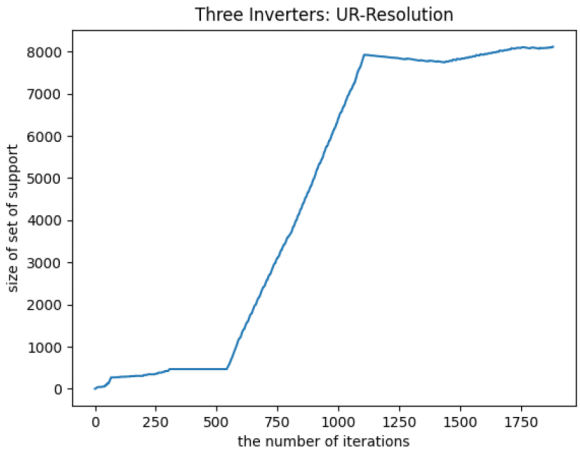

Fig. 4. Hyper-Resolution for constructing BCD. X-axis of the number of iteration steps. Y-axis is the size of the set of support.

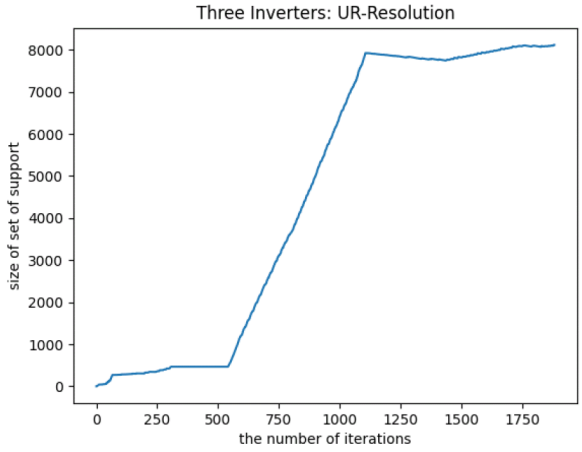

Fig. 3. UR-Resolution in constructing 3-input/3-output inverter. X-axis of the number of iteration steps. Y-axis is the size of the set of support.

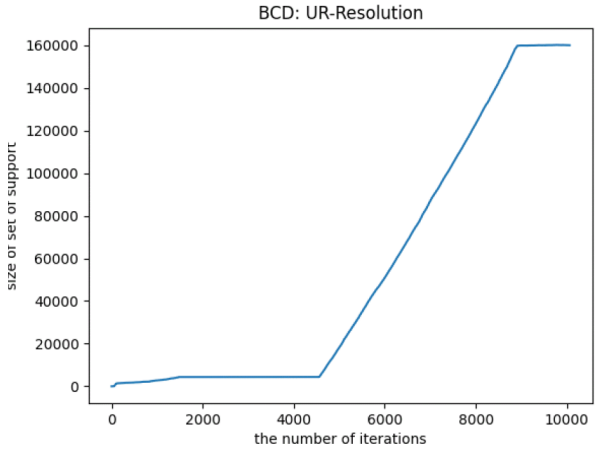

Fig. 5. UR-Resolution for constructing BCD. X-axis of the number of iteration steps. Y-axis is the size of the set of support.

output. With four digits, a BCD counter counts 0000, 0001, 0010, 1000, 1001, 1010, 1011, 1110, 1111, 0000, and 0001 and so on.

TABLE II
BCD (OR DECADE COUNTER CIRCUIT)

current state				next state			
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0

Table I describes the counting operation of the Decade counter. It represents the count of the circuit for the decimal count of input pulses. The NAND gate output is zero when the count reaches 10 (1010).

Figure 2 shows the size of the set of support during the reasoning process of Hyper-Resolution. In detail, X-axis is the iteration step. The size of SoS increases rapidly from around the iteration step of 800. Then, it increases slowly in the next about 4,000 steps. After plateaus from iteration step 1000 to 4700, the SoS size increase speeds up until iteration step 8000.

D. Comparison

In this section, we discuss the complexity of our problems from two points of view: the comparison by resolution strategies and the comparison by puzzles (two-inverters and BCD). Tables III, IV, V, and VI are the combination of rows by the statistics of reasoning processes. In general, hyper resolution takes more cost than UR-Resolution. In table V, in the two-inverter puzzle, UR resolution 53.68x is faster than Hyper-Resolution in the view of generated clauses. In table IV, in BCD, UR resolution is 6.03x faster than Hyper-Resolution in the view of generated clauses. Furthermore, in particular, we have two items of statistics:

- generated clauses: the number deduced clauses for every step of the main loop shown in Algorithm 1.
- set of support: clauses representing the fact and retained clauses tracked in Algorithm 1.

Let us consider the more detailed properties of two resolutions and puzzles by two comparisons.

1) Comparison by resolution strategies: In Table III, the increasing ratio of generated clauses in hyper resolution is about four times larger than the size of SoS (set of support). On the other hand, in Table IV, in UR-Resolution, the increasing ratio of generated clauses (x15.59) is smaller than the size of SoS (x19.7) From this result, we can conclude that hyper resolution is the strategy that tends to increase the size of the set of support. This property of the high increase rate of SoS size in hyper resolution is one of the reasons why hyper resolution takes more cost than UR resolution.

TABLE III
COMPARISON 1: HYPER RESOLUTION IN TWO INVERTERS AND BCD

	Two inverters	BCD	ratio
clauses generated	2069334	287084274	x138.73
hyper res generated	2069334	287084274	x138.73
demod & eval rewrites	2103338	288422196	x137.12
clauses forward subsumed	2049117	286403743	x139.76
(subsumed by SoS)	314773	44200930	x140.42
clauses kept	28858	1016164	x35.21
usable size	8646	335638	x38.82
SoS size	3.99	344900	x29.79
kbytes malloced	126925	311523	x24.53

TABLE IV
COMPARISON 2: UR-RESOLUTION IN TWO INVERTERS AND BCD

	Two inverters	BCD	ratio
clauses generated	3422935	5347949	x15.59
ur res generated	342935	534949	x15.59
demod & eval rewrites	349903	5383579	x15.38
clauses forward subsumed	332937	5177890	x15.55
(subsumed by SoS)	77455	1842716	x23.79
clauses kept	11881	180120	x15.16
usable size	1887	10065	x5.33
SoS size	8118	160002	x19.7
kbytes malloced	7812	79101	x10.12

2) Comparison by circuits: Tables V and VI show that the increasing ratio of generated clauses by switching resolution strategies from hyper to UR in solving BCD is larger than two inverter puzzles. The first column in the fourth row of Table V (x53.68) is bigger than Table VI (x6.03). About the size of the set of support, the increasing rate is relatively small. See the eighth column in the fourth row in Table V (x3.93) and VI (x1.42). This comparison shows that hyper resolution does not scale simply by the length of bits of two puzzles compared with UR resolution.

TABLE V
COMPARISON 3: BCD WITH HYPER RESOLUTION AND UR RESOLUTION

	UR	Hyper	ratio
clauses generated	5347949	2870843274	x53.68
ur res generated	5347949	287084274	x53.68
demod & eval rewrites	53383579	288422196	x53.57
clauses forward subsulmed	5177890	286403743	x55.31
(subsumed by SoS)	1842716	44200930	x23.98
clauses kept	180120	1016164	x5.64
usable size	10065	3356638	x2.15
SoS size	16002	344900	x3.93
kbytes malloced	79101	311523	x148.49

E. Insights

Hyper Resolution takes more computation cost than UR resolution. This is because Hyper Resolution emphasizes syntactic criteria rather than semantics. As shown in Tables V and IV, The number of clauses generated by Hyper Resolution is x53.68 (BCD) and x6.03 (Two-Inverter) times larger than UR Resolution. It is often said that UR Resolution is semantic-oriented and emphasizes the semantic criteria. However, UR Resolution is useless for specific problems where the larger deduction step is necessary. We have observed the curious change point in Figure 4 from x=15000 to x=20000. From this point, the size of the set of support is decreasing. Further inspection is necessary for this phenomenon.

TABLE VI
COMPARISON 4: TWO INVERTER WITH HYPER RESOLUTION AND UR RESOLUTION

	UR	Hyper	ratio
clauses generated	342935	2069334	x6.03
ur res generated	342935	2069334	x6.03
demod & eval rewrites	349903	2103338	x6.01
clauses forward subsulmed	332937	2049117	x6.15
(subsumed by SoS)	77452	314773	x4.06
clauses kept	11881	28858	x2.42
usable size	1887	8646	x4.58
SoS size	8118	11577	x1.42
kbytes malloced	7812	12695	x1.62

TABLE VII
COMPARISON 4: TWO INVERTER WITH HYPER-RESOLUTION AND UR RESOLUTION

	User CPU time	System CPU time
BCD (Hyper)	25801.54	1.94
BCD (UR)	241.66	0.13
Two-inv (Hyper)	25.89	0.01
Two-inv (UR)	3.89	0.01

V. RELATED WORK

Switching theory was first introduced by Shannon [3] with notable success in the practical application of Boolean algebra. The knotty problem known as the two inverter puzzle was first introduced by L.Wos [4]. Sallows [5] discussed the negation-limited inverters problem from the viewpoint of Moore's original problem in circuit design and a seemingly analogous problem in computer programming. Morizumi [6] proposed the linear size of the negation-limited inverter applying only o(n) NOT gates. In [7], it was shown that the complete set of input variables might be inverted D(n) inverters where D(n) is the small integer y such that $n < 2^y$. In [8], Tanaka et al. proposed the design of size O(n log n) inverter with depth O(log n) using [log (n+1)].

Originally, Hyper-Resolution was first illustrated in detail by [9]. The efficiency and completeness of the set of support strategy is discussed by L.Wos et al. [10]. Slaney et al. [11] proposed a model-guided theorem prover which is called SCOTT (Semantically Constrained Otter), with a resolution-based automatic theorem proving.

Another powerful ATP (Automated Theorem Proving) strategy is based on equational reasoning [23]. Many researchers regarded demodulation as the inference rule to remove less obviously redundant information. It is designed to enable reasoning programs to simplify and canonicalize yielded clauses by applying demodulators which are regarded as rewriting rules [24]. Ando and Takefuji [25] propose the application of demodulation for formal methods to analyze

viral software metamorphism. Wos proposes a look-ahead strategy which is called the hot list strategy, to cope with the frequently occurred delay in focusing on a retained conclusion [26]. A paramodulation inference rule is the basis of equational reasoning of OTTER. It consists of two parents and a child. By $fromterm$, the parent contains the equality for replacing literals. The $replacedterm$ is called into the term. Paramodulation is regarded as the generalization of a substitution rule for equational reasoning. Paramodulation serves to build properties of equality along with demodulation. Takefuji applies paramodulation for translating Common Lisp Takefuji. Ando and Takefuji use the paramodulation based strategy called hot list for speeding up graph coloring [21]. Also, Ando applies the hot list strategy for exposing parameter detection of polymorphic viral code [27].

VI. Discussion

A. Hyper-Resolution and UR-Resolution

The hyper-Resolution inference rule has the advantage of coping with larger deduction steps than binary resolution does. On the other hand, Hyper-Resolution has the disadvantage of emphasizing syntactic criteria rather than semantics. UR-Resolution inference rule has the advantage of emphasizing semantic criteria but disadvantages in taking certain types of problems. UR-Resolution requires all inferred clauses (conclusions) drawn into it to be unit clauses. The clauses contain exactly one literal because unit clauses correspond to assertions rather than to a choice of possibilities. Therefore, UR-Resolution is semantically printed. Consequently, it is not hard to see that the Hyper-Resolution inference rule leads to the derivation of clauses with only positive literal in them. Whereas this is sufficient for large clauses of problems, many reasoning tasks require the derivation of clauses containing negative literal.

Hyper-Resolution is good at coping with a large horn clause. UR-Resolution focuses on a unit clause containing a single literal. In the problems like the negation-limited inverters problem of this paper, the clause in SoS list does not contain a large clause as follows:

```
list(sos).
P(00001111, v). % input 1
P(00110011, v). % input 2
P(01010101, v). % input 3
end_of_list.
```

This point should be why UR-Resolution is more effective than Hyper-Resolution in generating the negation-limited inverters.

B. Set of support strategy

The reasoning process of OTTER operates based on the set of support strategy. As we discussed in the previous section, the set of support strategy restricts the reasoning program by selecting a nonempty set of T of S. For a feasible computing time, restriction strategies such as the set of support and weighting are essential for achieving some given assignment. Automated reasoning strategy is divided into four categories: restriction, direction, look-ahead, and redundancy-control. Among the four strategies, the restriction strategy is the most important because the reasoning program generates too many unacceptable conclusions without proper restriction. Currently, the set of support strategy is considered by many to be the most powerful restriction strategy available. In general, its use enables automated reasoning programs to prove theorems in far less computer time and memory than would be required as usual. In [20], a severe test of a set of support strategy is provided in proving theorems relying on the use of Godel's finite axiomatic of set theory.

VII. Conclusion

This paper discusses the novel result of two resolution strategies in the negation-limited inverter problem: UR (Unit Resulting) resolution and Hyper-Resolution. We have observed a significant difference in computing time in applying these two strategies.

Particularly, in the two-inverter puzzle, in the view of generated clauses, UR resolution is 53.68x faster than Hyper-Resolution. Besides, UR-Resolution takes 229.24 sec of user CPU time, which is x113.35 faster than Hyper-Resolution. We have found a significant difference between UR resolution and Hyper-Resolution in the size of SoS (Set of Support) measurements.

We also compare the syntactic and semantic criteria for considering this novel result, which might cause a significant difference in computation cost between Hyper-Resolution and UR resolution. Hyper-Resolution will reach out to the derivation with only positive literalism. It is sufficient in coping with large clauses of the problem. Instead of avoiding all restrictions on all clauses to be derived, UR-Resolution considers the possibility of clauses containing a single literal. For further work, we will inspect the detailed implementation of Hyper-Resolution and UR-Resolution in detail.

References

[1] I. Brcic: "Ideally Fast" Decimal Counters with Bistables. IEEE Trans. Electron. Comput. 14(5): 733-737 (1965)

[2] $https : //github.com/RuoAndo/otter - book/tree/master/neg_inv_problem$

[3] C. E. Shannon, A symbolic analysis of relay and switching circuits,Trans. AIEE 57 (1938), 713 - 723.

[4] L. Wos,A Computer Science Reader, Ed. E. A. Weiss, Springer-Verlag, (1988), pp. 110-137. Orig. pub.Abacus, vol. 2, no. 3 (Spring 1985), pp. 6-21.

[5] Lee C. F. Sallows, "A curious new result in switching theory", The Mathematical Intelligencer volume 12, pages21 - 32(1990)

[6] Hiroki Morizumi, Genki Suzuki: Negation-Limited Inverters of Linear Size. IEICE Trans. Inf. Syst. 93-D(2): 257-262 (2010)

[7] Sheldon B. Akers Jr.: On Maximum Inversion with Minimum Inverters. IEEE Trans. Computers 17(2): 134-135 (1968)

[8] Keisuke Tanaka, Tetsuro Nishino, Robert Beals: Negation-Limited Circuit Complexity of Symmetric Functions. Inf. Process. Lett. 59(5): 273-279 (1996)

[9] Ross A.Overbeek, An implementation of Hyper-Resolution, Computers & Mathematics with Applications Volume 1, Issue 2, June 1975, Pages 201-214

[10] Larry Wos, George A. Robinson, Daniel F. Carson: Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. J. ACM 12(4): 536-541 (1965)

[11] John K. Slaney, Ewing L. Lusk, William McCune: SCOTT: Semantically Constrained Otter System Description. CADE 1994: 764-768

[12] William McCune: Skolem Functions and Equality in Automated Deduction. AAAI 1990: 246-251

[13] William McCune: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. J. Autom. Reason. 9(2): 147-167 (1992)

[14] Ewing L. Lusk, William McCune: Tutorial on High-Performance Automated Theorem Proving. CADE 1990: 681

[15] Wos, L. Review: J. A. Robinson, Automatic Deduction with Hyper-Resolution. J. Symbolic Logic 39 (1974), no. 1, 189–190.
[16] McCharen, J., Overbeek, R., Wos, L., 1967. Complexity and related enhancements for automated theorem proving programs. Computers and Mathematics with Applications, 2, pp 1-16.
[17] Larry Wos: The Problem of Choosing the Type of Subsumption to Use. J. Autom. Reason. 7(3): 435-438 (1991)
[18] Ewing L. Lusk, William McCune, Ross A. Overbeek: ITP at Argonne National Laboratory. CADE 1986: 697-698
[19] Peter Graf, Term Indexing, Springer-VerlagBerlin, Heidelberg ISBN:978-3-540-61040-3, March 1996
[20] Larry Wos, "Automated reasoning: 33 BASIC research problem", Prentice-Hall, Inc., 1988
[21] Ruo Ando,Yoshiyasu Takefuji, "Hot List Strategy for Faster Paramodulation based Graph Coloring", WSEAS TRANSACTIONS ON COMPUTERS, Issue 7, Volume 5, pp1596-1599, July 2006
[22] Y. Takefuji and M. Dowell, "A Novel Approach to a Rule-Based General Purpose Program Translator Using Paramodulation: Case Study of A Franz-To-Common Lisp Translator," Knowledge-Based Systems, 1, 2, 90-93, March 1988.
[23] Larry Wos: The Problem of Reasoning from Inequalities. J. Autom. Reason. 8(3): 421-426 (1992)
[24] Larry Wos: The Problem of Demodulation During Inference Rule Application. J. Autom. Reason. 9(1): 141-143 (1992)
[25] Ruo Ando,Yoshiyasu Takefuji, "Formal method using demodulation against viral software metamorphism", in Proceedings of the 5th WSEAS International Conference on INFORMATION SCIENCE, COMMUNICATIONS and APPLICATIONS, Cancun, Mexico, May 11-14 2005
[26] Larry Wos, Gail W. Pieper: The Hot List Strategy. J. Autom. Reason. 22(1): 1-44 (1999)
[27] Ruo Ando, "Faster parameter detection of polymorphic viral code using hot list strategy", in Proceedings of 15th International Conference on Neural Information Processing of the Asia-Pacific Neural Network Assembly (ICONIP 2008) ,Auckland, New Zealand, November 2008