International Journal of Innovative Computing, Information and Control
Volume **8**, Number **10(A)**, October **2012**

ICIC International ©2012 ISSN 1349-4198
pp. **6613–6628**

GENERATION OF SPECIFIC SOLVERS FOR QUERY-ANSWERING PROBLEMS WITH SKOLEM FUNCTIONS

Songhao He[1], Kiyoshi Akama[2] and Bin Li[1]

[1]Graduate School of Information Science and Technology
Hokkaido University
Kita 14, Nishi 9, Kita-ku, Sapporo, 060-0814, Japan
hesonghao@ist.hokudai.ac.jp; zjulb@hotmail.com

[2]Information Initiative Center
Hokkaido University
Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan
akama@iic.hokudai.ac.jp

Received June 2011; revised October 2011

Abstract. *In this research, we propose a novel technology for solving the Query-Answering Problems (QA Problems) including Skolem functions by generating a specific solver corresponding to each QA Problem in the Semantic Web. A specific solver is constructed by using the knowledge (conditions) analyzed from a given QA Problem and is only used to deal with this given problem, since we think by this way the computation efficiency can be much higher than generating a general solver for all QA Problems based on the common knowledge. We first get the clause set including the Skolem functions from a given QA Problem based on the theory of meaning-preserving Skolemization. Then, a specific solver corresponding to this given QA Problem will be generated by using the clause set based on the Bottom-up solution. Moreover, we have also developed the technology to suppress the size of the generated specific solver for reducing the usage of memory in order to deal with larger scale QA Problems. In the final part of this paper, experimental results will be used to prove the efficiency of solving the QA Problems by generating specific solvers.*

Keywords: Query-answering problems, Skolem function, Clause, Bottom-up solution, Specific solver, Meaning-preserving Skolemization

1. Introduction.

We know that the Semantic Web, which enables machines to understand and use web contents to answer the complex human requests, has already been known as the next stage in the evolution of the web [1]. A powerful reasoning system is essential in the developing of Semantic Web for solving the logical problems. There are mainly two kinds of logical problems. The *proof problem* is a pair of $(K1, K2)$, where $K1$ and $K2$ are logical formulae. It is a yes-no question, i.e., the answer to a proof problem $(K1, K2)$ is "yes" if $K1$ logically entails $K2$, and the answer is "no" otherwise. The *query-answering problem* (*QA problem*) [2] is a pair of (K, q), where K is a logical formula and q is an atomic formula (atom). The answer to a QA Problem (K, q) is the set of all ground instances of q that are logically entailed by K. In this research, we mainly work on the solution to the QA Problems.

The conventional approaches for solving the QA Problems are mainly based on the Description Logic (DL) and Rules [3,4]. However, because of the limited ability of expression, QA Problems composed of logical expression such as FOL, Horn Clauses cannot be dealt with soundly, and the correctness of the obtained solutions maybe regarded as a question, which is a major reason why conventional Skolemization is insufficient. Therefore, in order to construct an excellent solution corresponding to even wider range

QA Problems, a theory of the meaning-preserving Skolemization based on the Equivalent Transformation (ET) has been proposed by our research group [5,6]. In the previous research [7], the Bottom-up solution used to deal with the QA Problems with Skolem functions based on the meaning-preserving Skolemization was proposed. The following computation procedures were proposed by [7] to solve the QA Problems.

1) convert K (knowledge) into a set of clauses using meaning-preserving Skolemization
2) construct all models of the clause set based on the Bottom-up solution
3) find the intersection of the models and the set of all ground instances of q (query)

However, the proposed method [7] for deciding the value set used to assign to the Skolem functions in step 1) is not entirely perfect. What is more important is that when reinitializing the Skolem functions to find even more models used to acquire the final correct answer of the QA Problem, the proposed Bottom-up computation in step 2) will be applied. Actually, the computational complexity is considerably large.

The reason for the large computational complexity is that in the previous approach [7], also in conventional approaches [3,4,8-10], the generation of an efficient general solver is thought to be the main target. The general solver takes the most common knowledge (conditions) included in the whole QA Problems into consideration, to solve all the QA Problems. Although the range of the correspondent QA Problems that could be dealt with by the general solver is considerably large, the specific knowledge (conditions) analyzed from each given QA Problem is not been made good use.

In this research, we think from a different angle. For each given QA Problem, we generate a specific solver by using the knowledge (K, clause set) acquired from the given QA Problem based on the meaning-preserving Skolemization. In this way, although the generation of each specific solver by analyzing the QA Problem will cost little time, the execution time (based on Bottom-up computation) of the specific solver is thought to be absolutely fast, especially when it will be repeatedly done only by changing the parameters (whenever reinitializing the Skolem functions). We call this the QA Problem specific approach, which is an absolutely novel technology for solving the QA Problems.

The purposes of this research are concluded as follows.

- The generation method of a *specific Bottom-up solver* by Low level Procedure (LPD, such as C program) for each given QA problem is proposed.
- The method for initializing Skolem functions is proposed.

The rest of the paper is structured as follows. In Section 2, we will introduce the QA Problems defined in our research and the Bottom-up solution, which is the basic idea used to generate a specific solver. In Section 3, the framework for generating a specific solver will be introduced, and the details of each step will be demonstrated with examples. In Section 4, experimental results and analysis will be introduced. In Section 5, conclusions and future work will be talked about.

2. **Query-Answering Problems and Bottom-up Solution.**

2.1. **Query-answering (QA) problems.**

Recently, QA Problems have gained wide attention, owing partly to emerging applications in systems involving integration between formal ontological background knowledge [11] and instance-level rule-oriented components, e.g., interaction between Description Logics and Horn rules [12] in the Semantic Web's ontology-based rule layer. The QA Problem is a problem containing knowledge (K, logical formula) and the query atom (q, atomic formula). And all ground answers have to be obtained. We define a set of ground atom (G) obtained by automatically reasoning the information clauses used to described the QA Problems. The answer A is composed of a set $q\theta$ of ground atoms of the query included in G, where θ is a substitution on q.

The answer to the QA Problem can be formulated by the following formulae.

$$A(K, q) = \{g|K \models g \in G, g = q\theta, \theta \in S\}$$

The answer requests all ground instances g, and g must satisfy the following constraints.

- g must be a logical consequence of K
- g must be a logical instance of q

Logical consequence is a fundamental concept in logic. It is the relationship between the premises and the conclusion of a valid argument.

Here, we take the *Oedipus* [13] problem as an example.

"OE is the child of IO. PO is the child of IO. PO is the child of OE. TH is the child of PO. OE is a patricide. TH is not a patricide. A person's child is a patricide, but his/her grandchild is not a patricide. Who is the person?"

This problem is composed of knowledge (K, "OE is the child of IO. . . . , but his/her grandchild is not a patricide.") and the query (q, "Who is the person?"). The answer (A) is "IO". This QA Problem can be expressed by the following clause set based on the meaning-preserving Skolemization, but there are no Skolem functions in this QA Problem. (There are two or more atoms existing in the left part of the clause P which also includes the query atom ***prob (∗x)***, indicating that the clause set processed in our research includes not only definite clauses, but also indefinite clauses, which represent an even wider range QA problems).

Knowledge (K):
isChild (OE, IO) ←. isChild (PO, IO) ←. pat (OE) ←.
isChild (TH, PO) ←. isChild (PO, OE) ←. ← pat (TH).
prob (∗x), pat (∗b) ← isChild (∗a, ∗x) , pat (∗a), isChild (∗b, ∗a). ----- Clause P
Query Atom (q): ***prob (∗x)***
Answer (A): $\{(K, q)\}$➔{IO}.

2.2. **Bottom-up solution.** The bottom-up solution starts by generating all models from knowledge (K, clause set) in the QA Problem. Then we take the intersection of all models ($K \models g \in G$). Finally, the result of the intersection will be used to compare with the query atom (q) to get the answer of the QA problem. In the process of the bottom-up solution, because all models cannot be requested at a dash, the knowledge set before becoming a model is called as the pre-model in our research. The pre-model becomes a model until it cannot be updated by any clause. In this research, we call the final generated model the representative model, and these models will be used to solve the QA Problem.

2.2.1. *Process of the bottom-up solution.* Bottom-up solution starts with known facts and extends the set of known facts using rules generated from clause set. It derives new facts from exist facts and clause set. The process of the bottom-up solution can be separated into the following steps (Figure 1).

A. Generation of the pre-model (empty set)
B. Select the pre-model
C. Update the pre-model by applying clauses in the clause set (K)
D. Making the representative model
E. Generating product set from all representative models
F. Extracting the instance (answer) of the question atom from the product set

In step C, the flow of processing is different depending on the kind of the applied clause.

- Negative clause (Delete operation)
  Clause: ←P1, P2, . . . , Pn. (n≥1), where P1, P2, . . . , Pn are atoms.

- ∘ The pre-model is deleted when atoms (P1, P2, . . . , Pn) exist.
  - ➢ pre-model: ~~{P1, P2, . . . , Pn}~~
- • Single head clause (Add operation)
  Clause: Q1 ← P1, P2, . . . , Pn. (k = 1, n ≥ 1), where Q1, P1, P2, . . . , Pn are atoms.
  - ∘ The Q1 atom is added to the pre-model after the P1, P2, . . . , Pn atoms were judged to be exist and Q1 atom did not exist in the model.
    - ➢ pre-model: {P1, P2, . . . , Pn}➔{P1, P2, . . . , Pn, Q1}
- • Multi-head clause (Fork and Add operation)
  Clause: Q1, . . . , Qk ← P1, . . . , Pn. (k, n≥1), where Q1, . . . , Qk and P1, . . . , Pn are atoms.
  - ∘ First of all, copy the pre-model k − 1 times. Then, if P1, . . . , Pn atoms were judged to be exist and at least one of the corresponding left part atoms (Q1, . . . , Qk) did not exist in the pre-model, the left one will be added to the corresponding pre-model.
    - ➢ pre-model_1: {P1, . . . , Pn, Q1} ----- original pre-model
      . . .
      pre-model_k: {P1, . . . , Pn, Qk} ----- new generated pre-model

3. **Framework of the Specific Solver Generation.** Figure 2 shows the solver generation framework proposed in our research. Firstly, the clause set is obtained from the given QA Problem based on the meaning-preserving Skolemization. Actually, not all QA Problems include Skolem functions. Here, we first introduce how to generate the specific solver for a QA problem without Skolem functions. And in 3.4, we will specially introduce

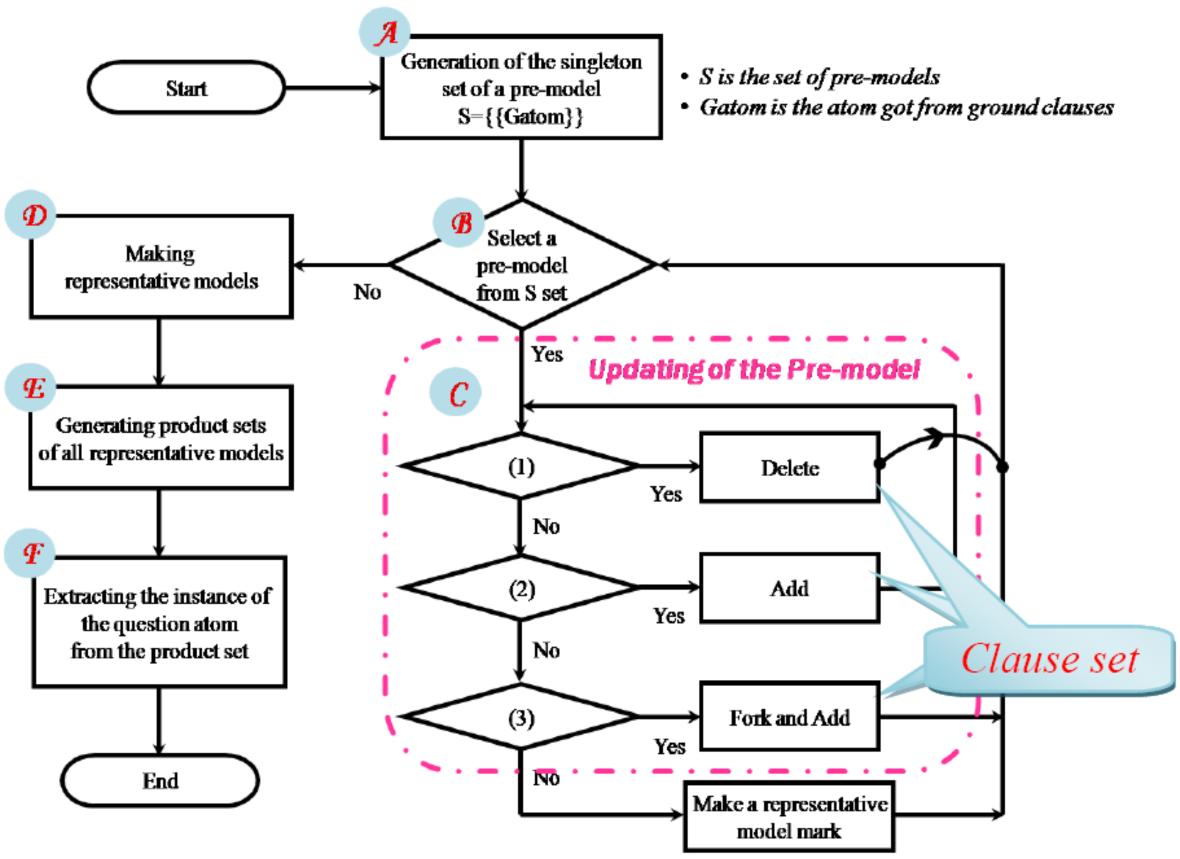

Start
A
Generation of the singleton set of a pre-model S={{Gatom}}
• S is the set of pre-models
• Gatom is the atom got from ground clauses
B
Select a pre-model from S set
No
Yes
D
Making representative models
E
Generating product sets of all representative models
F
Extracting the instance of the question atom from the product set
End
C
Updating of the Pre-model
(1)
Yes
Delete
No
(2)
Yes
Add
No
(3)
Yes
Fork and Add
No
Make a representative model mark
Clause set

FIGURE 1. Process of the bottom-up solution

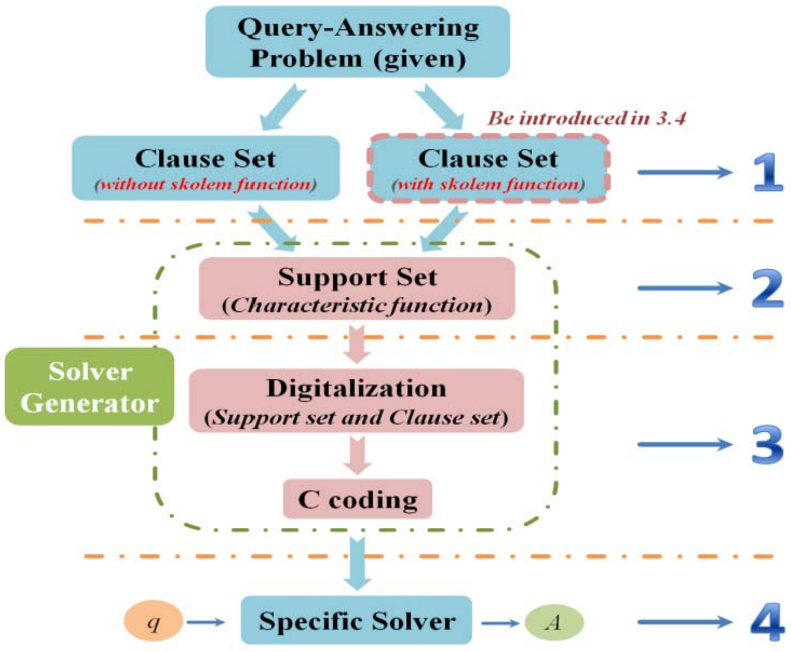

Query-Answering Problem (given)
Be introduced in 3.4
Clause Set (without skolem function)
Clause Set (with skolem function)
1
Support Set (Characteristic function)
2
Solver Generator
Digitalization (Support set and Clause set)
3
C coding
q
Specific Solver
A
4

FIGURE 2. Solver generation framework

the method for processing the Skolem functions and adding the Skolem function clauses to the original clause set to generate the corresponding specific solver. Secondly, to access the knowledge efficiently, basically in low level procedure, the knowledge range, which we call as the *support set*, including all possible ground atoms obtained from the clause set need to be set in a minimized and limited size. Thirdly, because the final solver is generated written by C program to pursue the efficient of the computation, the algorithm about how to convert all atoms and the clause set into C code (*digitalization*) are needed. Finally, we execute the specific solver by inputting the query atom (q), and the answer (A) of the given QA Problem will be the output.

3.1. **Characteristic function and support set.** In pre-model's updating, which is treated as the most important part in Bottom-up solution, shown in Figure 1, firstly, it is necessary to generate an S set (S set: {pre-model1, pre-model2, . . . }, pre-model: {atom1, atom2, . . . }) in which the element is the pre-model; Secondly, each pre-model in the S set will be updated by the clause set, until been changed into the representative model; Finally, the answer is obtained by using representative models. The data structure of the model (pre-model or representative model) is enumeration type. In the operation like the retrieval, the addition, and the deletion, it needs to scan from the first atom of the pre-model every time. The computation is not efficient. We use the characteristic function as a way to overcome the efficiency problem.

3.1.1. *Characteristic function.*

- Definition: a characteristic function is a function defined on a set X (set X is called the support set in this research) that indicates membership of an element in a subset A of X, having the value 1 for all elements of A and the value 0 for all elements of X not in A ($A(x) : X \to \{1, 0\}; x \mapsto A(x)$) [14].
- Expression: the characteristic function is assumed to be f, and $f^{-1}(1)$ is a set, and it has been treated as the pre-model.
- Updating of the pre-model: by applying the characteristic function, when the element is retrieved among the decided range (X set), the existence information can be shown

by bit 1/0. Moreover, when the atom is added to the pre-model, it is possible to do the operation quickly only by converting 0 to 1 by using the characteristic function (Figure 3).

- Bottom-up solution: based on the definition, in order to achieve the characteristic function, it is necessary to make the limited range (support set, like X set) firstly, in which subset A is included (Figure 3). Then, generate one pre-model (P_{start}), and update the pre-model by applying clauses. Finally, the set of representative models is generated (Figure 4), and the intersection result of the set will be taken to obtain the final answer.

3.1.2. *Support set.* The support set is a set G which has the "$\supseteq$" relation with all possibly basic atoms (set g) included in the model derived from the clause set (K) of the QA Problem. In this research, we define the set X introduced in 3.1.1 (*Characteristic function*) as a sample of the support set. By deciding this support set, which contains all possible atoms, the answer searching range of the QA Problem can be decided limitedly. Therefore, we can make use of the characteristic function to obtain the representative models

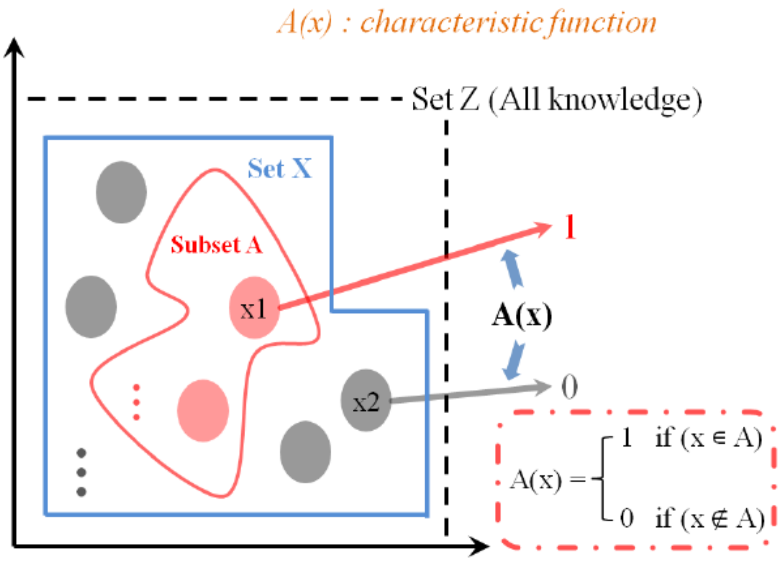

A(x) : characteristic function
Set Z (All knowledge)
Set X
Subset A
x1
x2
1
A(x)
0
A(x) = 1 if (x ∈ A), 0 if (x ∉ A)

FIGURE 3. Characteristic function

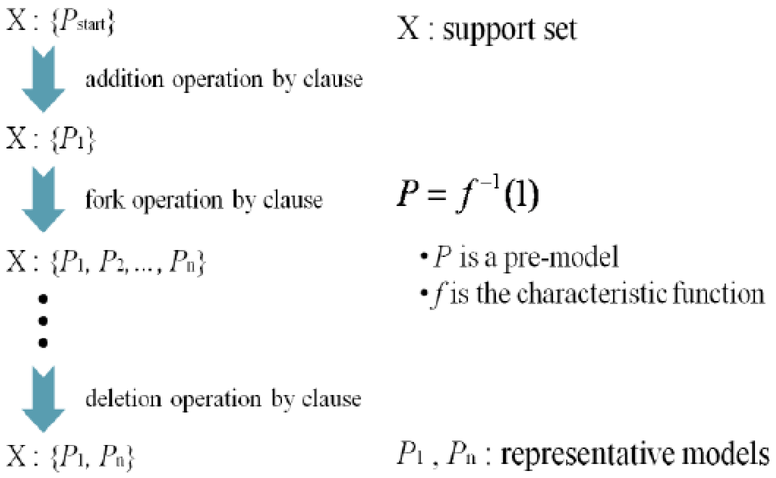


FIGURE 4. Generation of representative models by characteristic function

efficiently. Moreover, the more the basic atom set (g) becomes like the support set (G), the more we can narrow the answer searching range and obtain the answer efficiently. Figure 5 shows the representative model sets included in the support set (representative model $\subseteq$ support set).

- Generation of the support set: The support set generating process has been divided into three steps (A, B, C) shown in Figure 6. The Clause set requested from the given QA Problem (*Oedipus*) is the input, and the correspondent support set generation program is the output. In the approximate processing of A, the possible atoms of the problem are roughly requested. The smaller (more accurate) the support set is requested, the higher calculation cost for generating the support set is needed. Based on the approximate idea, we do not achieve the most accurate support set, but within an approximate range, search a little wider range support set efficiently, and finally generate the program [15] used to get the support set. The support set is requested by executing the program.

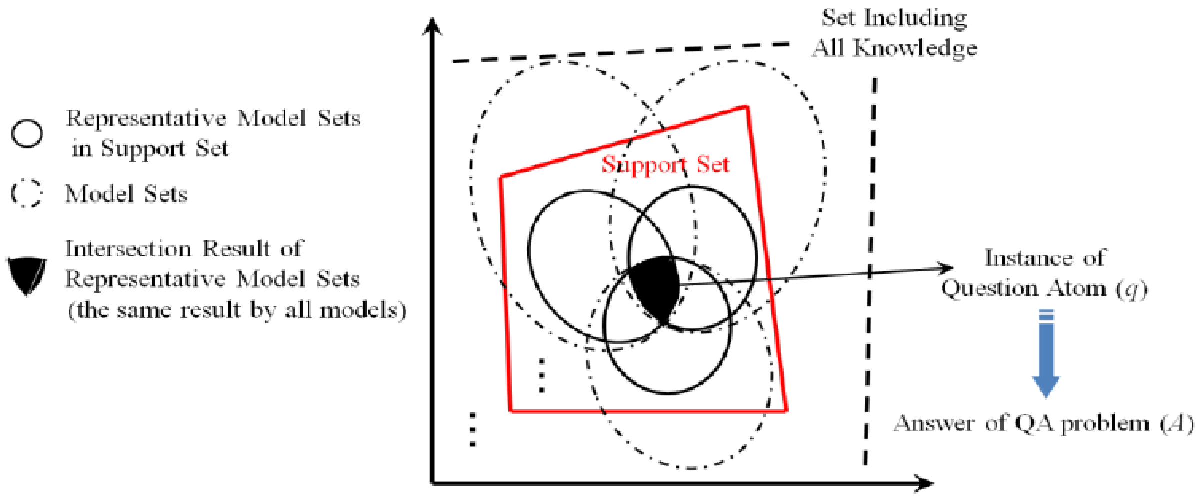

Representative Model Sets in Support Set
Model Sets
Intersection Result of Representative Model Sets (the same result by all models)
Set Including All Knowledge
Support Set
Instance of Question Atom (q)
Answer of QA problem (A)

FIGURE 5. Location of support set in the knowledge set

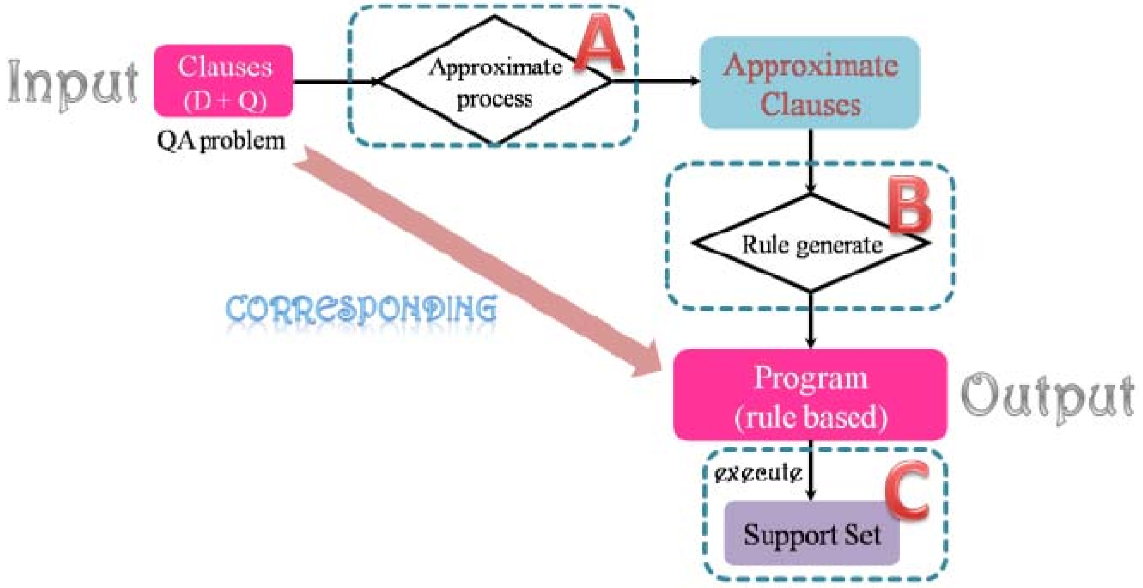


FIGURE 6. Process of generating the support set

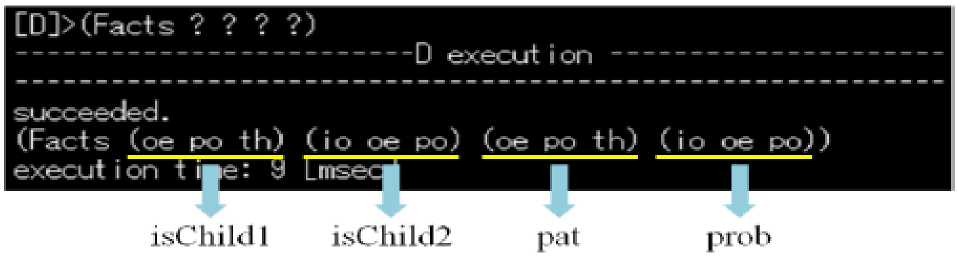
A) Approximate process
Clause set in the Oedipus problem (*Input*):

(isChild oe io) ←. (isChild po io) ←. (isChild po oe) ←.
(isChild th po) ←. (pat oe) ←. ← (pat th).
(prob *x), (pat *b) ← (isChild *a *x), (pat *a), (isChild *b *a).

New generated clauses (*Output*):

(isChild1 oe) ←. (isChild2 io) ←. (isChild1 po) ←. (isChild2 oe) ←.
(isChild1 th) ←. (isChild2 po) ←. (pat oe) ←. ← (pat th).
(prob *x) ← (isChild1 *a), (isChild2 *x), (pat *a), (isChild1 *b), (isChild2 *a).
(pat *b) ← (isChild1 *a), (isChild2 *x), (pat *a), (isChild1 *b), (isChild2 *a).

B) Generation of the approximate clauses: Input of step *B)* is the output of step *A)*.
Output (rule based program):
(whole *isChild1 *isChild2 *pat *prob *x1 *x2 *x3 *x4), {(addelem *isChild1 (oe) on *isChild1new)}→ (whole *isChild1new *isChild2 *pat *prob *x1 *x2 *x3 *x4).
. . .
(whole *isChild1 *isChild2 *pat *prob *x1 *x2 *x3 *x4), {(inter (*pat *isChild2 *isChild1) *mid1), (inter (*isChild2) *mid2), (inter (*isChild1) *mid3), (cons *mid1), (cons *mid2), (cons *mid3), (addelem *pat *mid3 on *patnew)}→ (whole *isChild1 *isChild2 *patnew *prob *x1 *x2 *x3 *x4).
C) Generation of the support set by executing the rule based program got in step B): By applying rules, each predicate set (isChild1, isChild2, pat, prob) that constituted the support set has been expanded (Figure 7).



```
[D]>(Facts ? ? ? ?)
-------------------------D execution --------------------
---------------------------------------------------------
succeeded.
(Facts (oe po th) (io oe po) (oe po th) (io oe po))
execution time: 9 [msec]
```

FIGURE 7. Generating of each predicate set

The support set is composed by combining these predicate sets, and the support set of the *Oedipus* problem is shown as follows.
{(isChild oe io), (isChild oe oe), (isChild oe po), (isChild po io), (isChild po oe), (isChild po po), (isChild th io), (isChild th oe), (isChild th po), (pat oe), (pat po), (pat th), (prob io), (prob oe), (prob po)}

3.2. **Digitalization of support set and the clause set.** As introduced in the research purpose, we want to generate the specific solver (C program) for each QA Problem, it is necessary to make the mechanism about how to convert clauses into the corresponding C program. In order to transfer clauses to C program, the algorithm about how to convert the atom, the basic element consisting of the clause, into the index number of the bit array (idea of the characteristic function) used in C program is very essential.

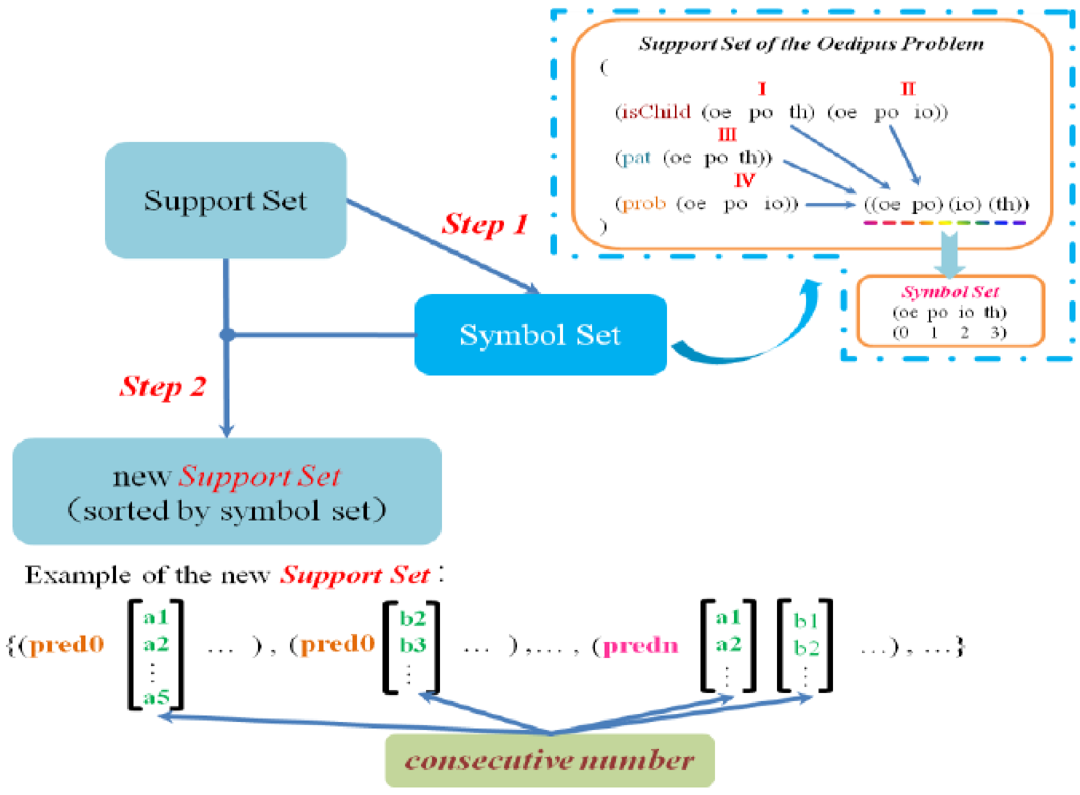


FIGURE 8. Digitalization of the support set

3.2.1. *Digitalization of support set.* First of all, for all atoms included in the support set, the *symbol set* that includes all the symbol values which substitute the argument are requested. Then, the order of symbols in the symbol set will be decided, and each symbol is converted into the natural number. Finally, all atoms in the support set are sorted in alphabetical order of the argument by the order of the symbol set (Figure 8).

Symbol Set:

(oe po io th) → (0 1 2 3)

Input (support set):

{(isChild oe io), (isChild oe oe), (isChild oe po), (isChild po io), (isChild po oe), (isChild po po), (isChild th io), (isChild th oe), (isChild th po), (pat oe), (pat po), (pat th), (prob io), (prob oe), (prob po)}

Output (digitalized support set):

{(isChild 0 0), (isChild 0 1), (isChild 0 2), (isChild 1 0), (isChild 1 1), (isChild 1 2), (isChild 3 0), (isChild 3 1), (isChild 3 2), (pat 0), (pat 1), (pat 3), (prob 0), (prob 1), (prob 2)}

3.2.2. *Digitalization of clause set.* Because we will finally convert each clause into the corresponding C program (if statement/for loops), it is necessary to make the algorithm about how to convert the atom into the address of the bit array. Here, the atom-address calculating function, used to make all basic atoms in the support set correspond to the address of the bit array, is made. Consequently, it is possible to access the address which corresponds to the atom quickly in the pre-model updating process.

In the digitalized support set, the argument of atoms with a consecutive value is brought together. Then, the address function corresponding to these atoms is generated. It

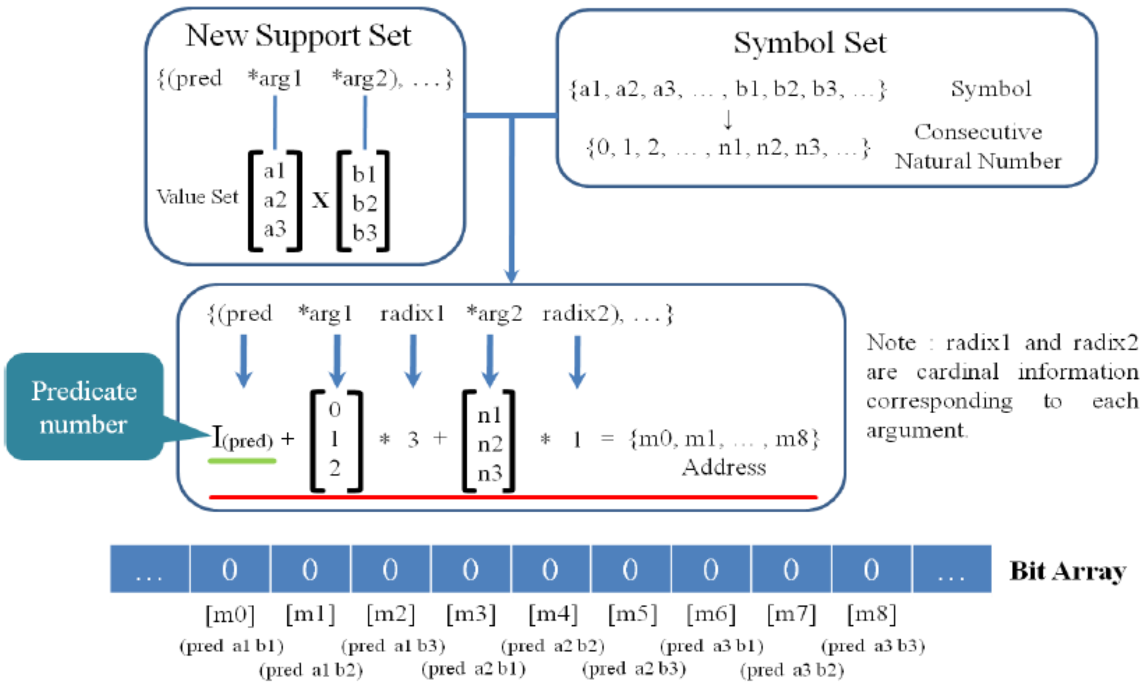

New Support Set
{(pred *arg1 *arg2), ...}
Value Set
a1 a2 a3
X
b1 b2 b3
Symbol Set
{a1, a2, a3, ..., b1, b2, b3, ...}
{0, 1, 2, ..., n1, n2, n3, ...}
Symbol
Consecutive Natural Number
{(pred *arg1 radix1 *arg2 radix2), ...}
Predicate number
I(pred) + [0 1 2] * 3 + [n1 n2 n3] * 1 = {m0, m1, ..., m8}
Address
Note : radix1 and radix2 are cardinal information corresponding to each argument.
Bit Array
[m0] [m1] [m2] [m3] [m4] [m5] [m6] [m7] [m8]
(pred a1 b1) (pred a1 b2) (pred a1 b3) (pred a2 b1) (pred a2 b2) (pred a2 b3) (pred a3 b1) (pred a3 b2) (pred a3 b3)

FIGURE 9. Digitalization of clause set

generates completely different address function for discontinuous argument value in spite of having the same predicate (Figure 9).

The address “PAdr(s1,...,sn)” of all basic atoms “Pred(s1,...,sn)” (There are n arguments) can be decided by the introduced algorithm. It is requested from the predicate number “I(pred)”, and the relative address “Rel(s1,...,sn)” of the predicate “Pred”, based on the following formula.

$$\text{PAdr(s1,...,sn)} = \text{I(pred)} + \text{Rel(s1,...,sn)} \tag{1}$$

Relative address “Rel(s1,...,sn)” is requested from the 1st argument value “Sym(s)” and its cardinal “R(pred,s)” of the last argument by the following formula.

$$\text{Rel(s1,...,sn)} = \text{Sym(s1)}*\text{R(pred,s1)} + \cdots + \text{Sym(sn)}*\text{R(pred,sn)} \tag{2}$$

Cardinal “$R(p, i)$” is requested by using symbol number “$S_{sym}(p, k)$”. For instance, based on the support set how many symbols can substitute the back arguments.

$$R(p, i) = \prod_{k=i+1}^{n} S_{sym}(p, k)$$

By applying the generated address function, atoms in the support set can be converted into address of the bit-array (Figure 9).

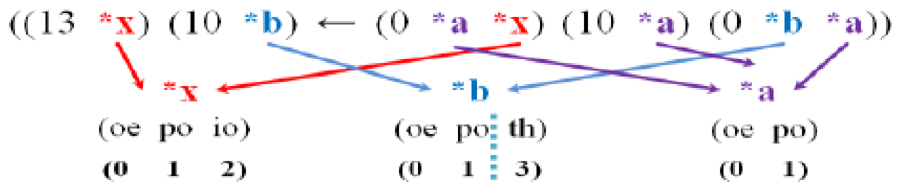


FIGURE 10. Example of limitation of clauses

Here, as an example, the intersection calculation of the value set of each atom's argument in a clause is done. The result is shown in Figure 10.

As shown in Figure 10, because the value set corresponding to the variable (∗b) is not a consecutive value, the clause with condition is generated based on the value set corresponding to the argument (limitation of clauses). The condition part of ground clauses is empty. The following part is a digitalized clause result of the *Oedipus* problem.

- Original clause:

(prob ∗x), (pat ∗b) ← (isChild ∗a ∗x), (pat ∗a), (isChild ∗b ∗a).

- Digitalize clause:

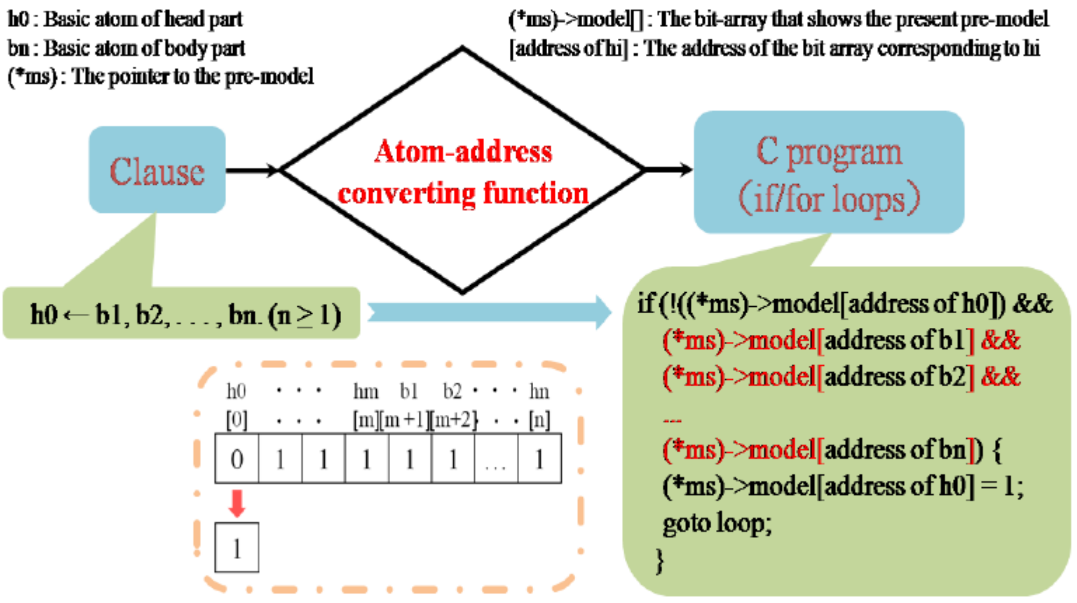
{(∗x 0 2) (∗b **0 1**) (∗a 0 1)} ((13 ∗x) (10 ∗b) ← (0 ∗a ∗x) (10 ∗a) (0 ∗b ∗a))
{(∗x 0 2) (∗b **3**) (∗a 0 1)} ((13 ∗x) (10 ∗b) ← (0 ∗a ∗x) (10 ∗a) (0 ∗b ∗a))

3.3. **C code generation.** The generated solver in this research is composed by three parts, which are main function definition, bit-array declaration and if statement/for loops. We input the query atom (q), and the solver will output the corresponding answer (A). The generation of if statement/for loops is requested by using the digitalize clause generated in 3.2.2. In this research, the expression of the updating clause is composed by the *applying conditions* part and the *applying processes* part. The expression of the clause is basically shown as the following structure.

if (applying conditions) {applying process; . . . }

If atoms at the right side of the clause are contained in the pre-model, while atoms at the left side of the clause are not contained in the pre-model, the *applying conditions* become available. In the following parts, we will introduce the method about how to convert various kinds of clause into if statement or for loops of C program.

3.3.1. *Generating "if statement" from ground clauses.* The transmission from a ground clause to an "*if loops*" in C program is shown as follows (Figure 11).


h0 : Basic atom of head part
bn : Basic atom of body part
(*ms) : The pointer to the pre-model
(*ms)->model[] : The bit-array that shows the present pre-model
[address of hi] : The address of the bit array corresponding to hi
Clause
Atom-address converting function
C program (if/for loops)
h0 ← b1, b2, . . . , bn. (n ≥ 1)
h0 · · · hm b1 b2 · · · hn
[0] · · · [m][m+1][m+2} · · [n]
0 1 1 1 1 1 … 1
1

```
if (!((*ms)->model[address of h0]) &&
  (*ms)->model[address of b1] &&
  (*ms)->model[address of b2] &&
  ---
  (*ms)->model[address of bn]) {
  (*ms)->model[address of h0] = 1;
  goto loop;
 }
```

FIGURE 11. From ground clauses to "*if loops*"

3.3.2. *Generating "for loops" from not ground clause.* A clause that contains variables will first substitute all variables for possible symbols, and then generate new ground clauses. The patterns of the symbol those can substitute the variable increase while the size of the QA problem grows. Therefore, the size of the generated C program will grow, and sometimes it will be impossible to compile. In this research, because the value set corresponding to the variable of each atom in the clause is obtained based on the support set, and been changed into natural numbers (starting from 0) based on the symbol set, the consecutive value will be expressed by one "*for loops*" (Figure 12).

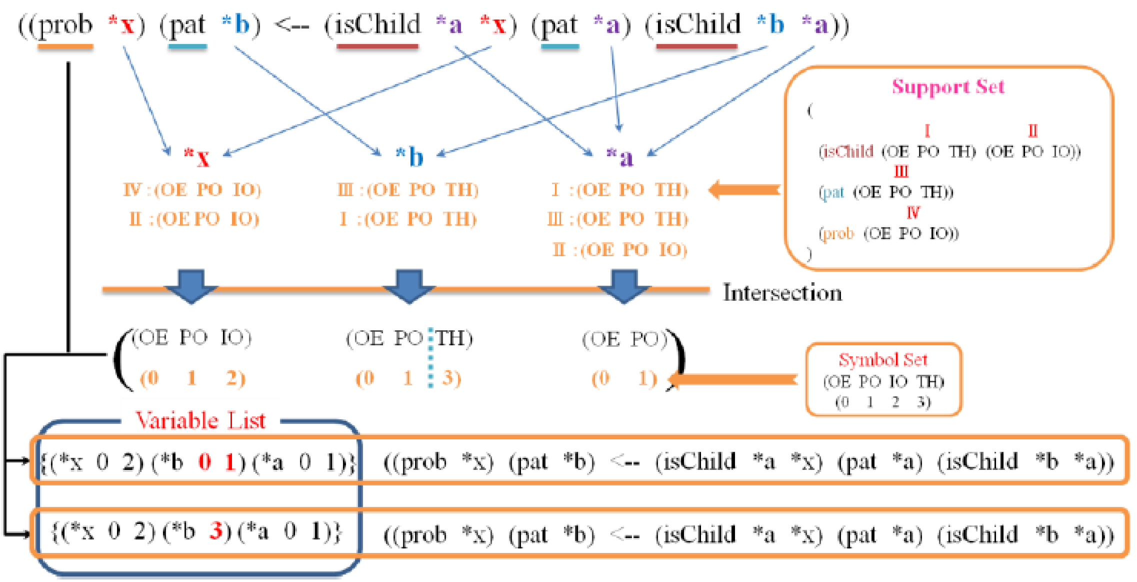

((prob *x) (pat *b) <-- (isChild *a *x) (pat *a) (isChild *b *a))
Support Set
(
I II
(isChild (OE PO TH) (OE PO IO))
III
(pat (OE PO TH))
IV
(prob (OE PO IO))
)
*x
IV :(OE PO IO)
II :(OE PO IO)
*b
III :(OE PO TH)
I :(OE PO TH)
*a
I :(OE PO TH)
III :(OE PO TH)
II :(OE PO IO)
Intersection
(OE PO IO)
(0 1 2)
(OE PO TH)
(0 1 3)
(OE PO)
(0 1)
Symbol Set
(OE PO IO TH)
(0 1 2 3)
Variable List
{(*x 0 2) (*b 0 1) (*a 0 1)} ((prob *x) (pat *b) <-- (isChild *a *x) (pat *a) (isChild *b *a))
{(*x 0 2) (*b 3) (*a 0 1)} ((prob *x) (pat *b) <-- (isChild *a *x) (pat *a) (isChild *b *a))

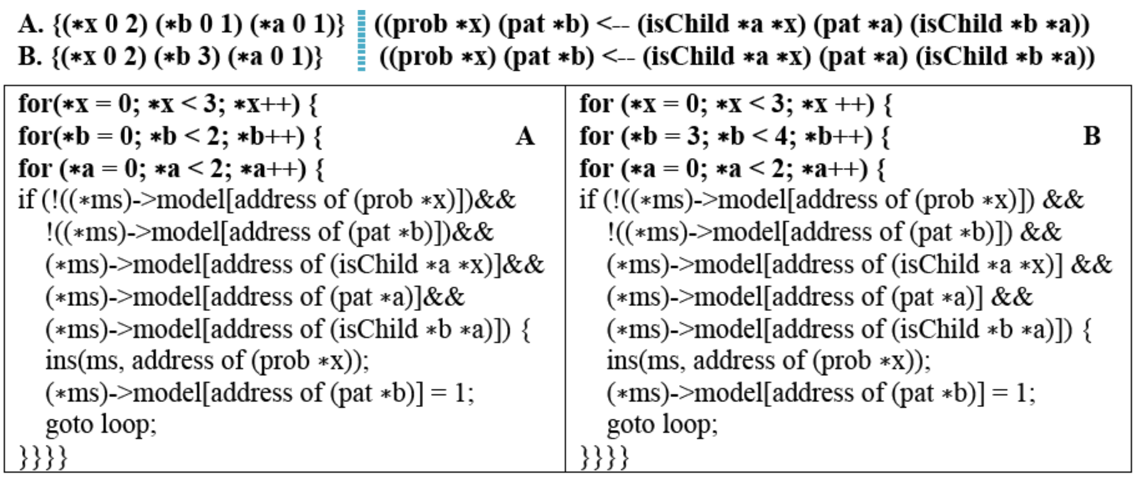
FIGURE 12. From not ground clauses to "*for loops*"

As shown in Figure 12, each clause will be finally transformed into one or more clauses with variable list. First of all, the value set corresponding to the variable of each atom in the clause is obtained based on the support set. Secondly, intersection results of these value sets are requested, and based on the symbol set obtained from the support set the intersection result will be changed into a natural number (starting from 0). Finally, clauses with variable list are generated. These clauses will be finally converted to for loops. Figure 13 shows C programs corresponding to the updating clauses obtained in Figure 12.

3.4. **Process for the QA problem with Skolem functions.** In this clause, we will introduce the method about how to decide the value set used to assign to the Skolem functions included in the clause set, which is acquired from the knowledge (K) based on the meaning-preserving Skolemization. Then, add these Skolem function clauses to the original clause set. Finally, the way for generating the specific solver of the given QA problem is the same as the one without Skolem functions.

3.4.1. *Deciding value set for the Skolem functions.* In Figure 14, the clause set (C_s, *Killer* problem [16]) including Skolem functions ($*f1$, $*f2$) is obtained based on the theory of meaning-preserving Skolemization. Here, we propose an approach by providing the value set for the Skolem functions which will be used in the specific solver generation.

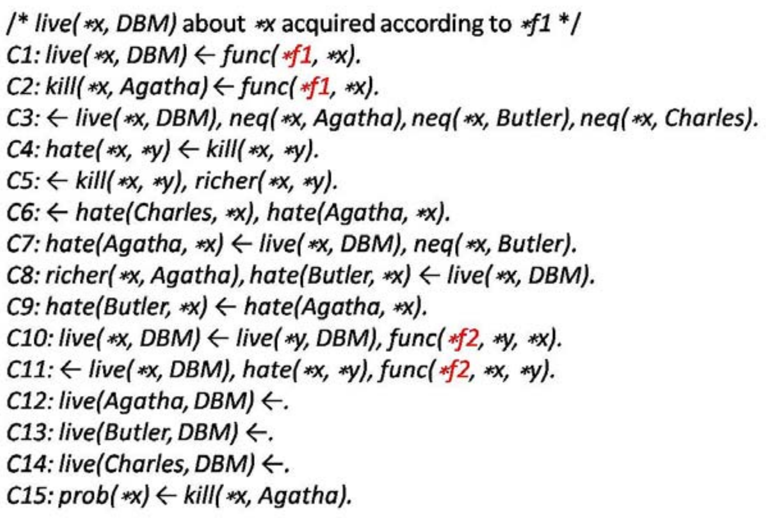
For the Skolem functions, we do not only need to consider constants that have been already included in the given knowledge [5], but also have to assume some constants information which is not included in the given knowledge to set the value for Skolem



```
A. {(*x 0 2) (*b 0 1) (*a 0 1)}  ((prob *x) (pat *b) <-- (isChild *a *x) (pat *a) (isChild *b *a))
B. {(*x 0 2) (*b 3) (*a 0 1)}    ((prob *x) (pat *b) <-- (isChild *a *x) (pat *a) (isChild *b *a))
```

A	B
`for(*x = 0; *x < 3; *x++) {`	`for (*x = 0; *x < 3; *x ++) {`
`for(*b = 0; *b < 2; *b++) {`	`for (*b = 3; *b < 4; *b++) {`
`for (*a = 0; *a < 2; *a++) {`	`for (*a = 0; *a < 2; *a++) {`
`if (!((*ms)->model[address of (prob *x)])&&`	`if (!((*ms)->model[address of (prob *x)]) &&`
`!((*ms)->model[address of (pat *b)])&&`	`!((*ms)->model[address of (pat *b)]) &&`
`(*ms)->model[address of (isChild *a *x)]&&`	`(*ms)->model[address of (isChild *a *x)] &&`
`(*ms)->model[address of (pat *a)]&&`	`(*ms)->model[address of (pat *a)] &&`
`(*ms)->model[address of (isChild *b *a)]) {`	`(*ms)->model[address of (isChild *b *a)]) {`
`ins(ms, address of (prob *x));`	`ins(ms, address of (prob *x));`
`(*ms)->model[address of (pat *b)] = 1;`	`(*ms)->model[address of (pat *b)] = 1;`
`goto loop;`	`goto loop;`
`}}}}`	`}}}}`

FIGURE 13. Consolidating expression of update rules by "*for loops*"



```
/* live(*x, DBM) about *x acquired according to *f1 */
C1: live(*x, DBM) ← func(*f1, *x).
C2: kill(*x, Agatha) ← func(*f1, *x).
C3: ← live(*x, DBM), neq(*x, Agatha), neq(*x, Butler), neq(*x, Charles).
C4: hate(*x, *y) ← kill(*x, *y).
C5: ← kill(*x, *y), richer(*x, *y).
C6: ← hate(Charles, *x), hate(Agatha, *x).
C7: hate(Agatha, *x) ← live(*x, DBM), neq(*x, Butler).
C8: richer(*x, Agatha), hate(Butler, *x) ← live(*x, DBM).
C9: hate(Butler, *x) ← hate(Agatha, *x).
C10: live(*x, DBM) ← live(*y, DBM), func(*f2, *y, *x).
C11: ← live(*x, DBM), hate(*x, *y), func(*f2, *x, *y).
C12: live(Agatha, DBM) ←.
C13: live(Butler, DBM) ←.
C14: live(Charles, DBM) ←.
C15: prob(*x) ← kill(*x, Agatha).
```

FIGURE 14. Clause set of *Killer* problem

functions. Because we do not know if there is any knowledge existing outside the given QA Problem but related to the problem. Therefore, as an example, in *Killer* problem, we do not only gather the constants {*Agatha, Butler, Charles, DBM*} that already exist in given knowledge, but also assume some other constants (in a limited range), like {*Peter, Paul*}, which we cannot judge them right or wrong before computation. Therefore, the Skolem function *func(∗f1, ∗x)* and *func(∗f2, ∗x, ∗y)* will be set as the following clauses.
C_{f1}: *func(∗f1, Agatha), func(∗f1, Butler), func(∗f1, Charles), func(∗f1, DBM), func(∗f1, Peter), func(∗f1, Paul)* $\leftarrow$.
C_{f21}: *func(∗f2 (Agatha, Butler, Charles, DBM, Peter, Paul) (Agatha, Butler, Charles, DBM, Peter, Paul))* $\leftarrow$.

These clauses show the range of {*Agatha, Butler, Charles, DBM, Peter, Paul*} for the Skolem function $*f1$, and {*(Agatha (Agatha, Butler, Charles, DBM, Peter, Paul)), (Butler (Agatha, Butler, Charles, DBM, Peter, Paul)), (Charles (Agatha, Butler, Charles, DBM, Peter, Paul)), (DBM(Agatha, Butler, Charles, DBM, Peter, Paul)), (Peter(Agatha, Butler, Charles, DBM, Peter, Paul)), (Paul (Agatha, Butler, Charles, DBM, Peter, Paul))*} for the Skolem function $*f2$.

3.4.2. *Making efficient use of the Skolem function setting clauses.* Because the clause set of the Skolem functions ($*f1$, $*f2$) is a disjunction relationship, the sum of the possible union for generating the model in the Bottom-up computation (about 6^7) and the consumption of memory will be extremely large. Therefore, in this research, we propose a "*Probability Way*", by which we do not compute all the possible function values to generate the model, but randomly choose one element from each Skolem function setting clauses in one execution (basically 1000 times). In this way, we think the final answer will be converged stably. Figure 15 shows the randomly setting clauses for Skolem function.

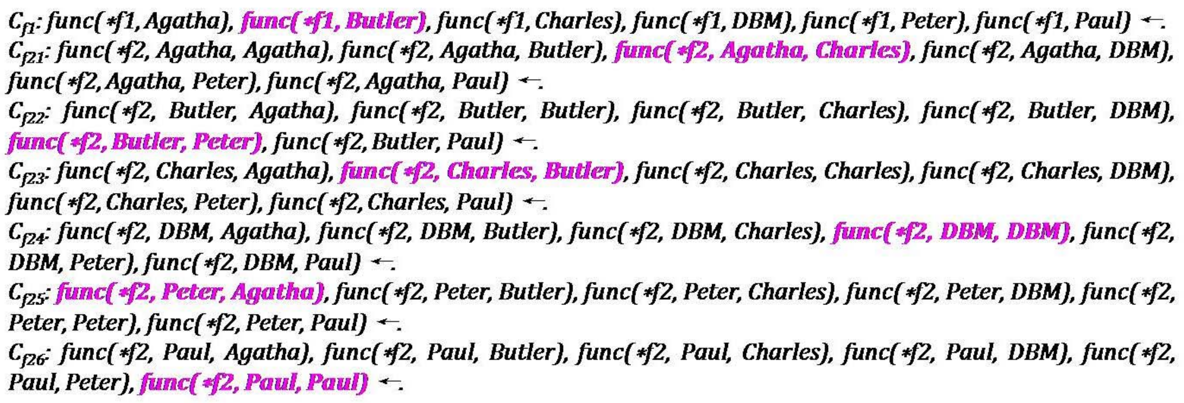

C_{f1}: func(*f1, Agatha), func(*f1, Butler), func(*f1, Charles), func(*f1, DBM), func(*f1, Peter), func(*f1, Paul) ←.
C_{f21}: func(*f2, Agatha, Agatha), func(*f2, Agatha, Butler), func(*f2, Agatha, Charles), func(*f2, Agatha, DBM), func(*f2, Agatha, Peter), func(*f2, Agatha, Paul) ←.
C_{f22}: func(*f2, Butler, Agatha), func(*f2, Butler, Butler), func(*f2, Butler, Charles), func(*f2, Butler, DBM), func(*f2, Butler, Peter), func(*f2, Butler, Paul) ←.
C_{f23}: func(*f2, Charles, Agatha), func(*f2, Charles, Butler), func(*f2, Charles, Charles), func(*f2, Charles, DBM), func(*f2, Charles, Peter), func(*f2, Charles, Paul) ←.
C_{f24}: func(*f2, DBM, Agatha), func(*f2, DBM, Butler), func(*f2, DBM, Charles), func(*f2, DBM, DBM), func(*f2, DBM, Peter), func(*f2, DBM, Paul) ←.
C_{f25}: func(*f2, Peter, Agatha), func(*f2, Peter, Butler), func(*f2, Peter, Charles), func(*f2, Peter, DBM), func(*f2, Peter, Peter), func(*f2, Peter, Paul) ←.
C_{f26}: func(*f2, Paul, Agatha), func(*f2, Paul, Butler), func(*f2, Paul, Charles), func(*f2, Paul, DBM), func(*f2, Paul, Peter), func(*f2, Paul, Paul) ←.

FIGURE 15. Randomly setting clauses for Skolem functions

The Skolem function setting clauses will be the following ones (C_{sf}) this time.
*func(*f1, Butler)*←. *func(*f2, Agatha, Charles)*←. *func(*f2, Butler, Peter)*←. *func(*f2, Charles, Butler)*←. *func(*f2, DBM, DBM)*←. *func(*f2, Peter, Agatha)*←. *func(*f2, Paul, Paul)*←.

Therefore, the whole clause set of *Killer* problem this time will be: $C_s + C_{sf}$.

4. Experiment.

In this section, we do experiments using four problems (e.g., *Killer* [16], *mayDoThesis* [3], *TaxCut* [16], *SteamRoller* [17]). In Table 1, the experiment results have been shown. The "Number of clauses" means the number of original clauses without Skolem function setting clauses. The "Number of Skolem Function" shows the number of Skolem functions required by using the knowledge (K) based on the meaning-preserving Skolemization. We have also taken the generating time of support set and specific solver, and the execution time of the solver as the criteria. The "Number of *RM*" demonstrates the number of representative models generated in 1000 times' solver execution (randomly setting Skolem function clauses each time).

From the experiment results, we can find out that the total time of the specific solver proposed in this research is absolutely shorter than the way proposed in the previous research (about 1/1000). Also, as the size of the given QA Problem becomes bigger (TaxCut < Killer < mayDoThesis < SteamRoller), the generating time of support set and specific solver, and also the execution time become longer. Furthermore, in *SteamRoller* problem, the number of Skolem Function is eight, even more than the others, and the related consuming time is extremely longer than the others. Here, the "Execution Time" of the generated specific solver as shown in Table 1 is the total time of 1000 times' solver execution. By 1000 times of execution for each QA Problem, the "Number of *RM*" has also been shown in Table 1. Figure 16 shows the converging conditions of each QA Problem by making the intersection of all generated *RM* (representative model). We can find out the convergence curve of all *RMs* almost changing into the straight line in the final, which means the target answer range (the black range shown in Figure 5) is obtained.

TABLE 1. Experiment results

QA Problems	Number of clauses	Number of *Skolem Function*	Generating time of *Support Set (ms)*	Generating time of *Solver (ms)*	Execution Time *(ms)*	Number of *RM*	Answer
Killer							
This Research	15	2	2556	207	410	4	prob (Agatha)
Previous Research [7]	15	2	0	0	3630320	1	prob (Agatha)
mayDoThesis							
This Research	19	1	2884	237	513	1000	mayDoThesis (paul) (john mary)
Previous Research [7]	19	1	0	0	5485840	1	mayDoThesis (paul) (john mary)
TaxCut							
This Research	7	2	814	86	335	746	TaxCut (Peter)
Previous Research [7]	7	2	0	0	2730356	1	TaxCut (Peter)
SteamRoller							
This Research	26	8	11533	588	893	417	prob (Fox)
Previous Research [7]	26	8	0	0	11315350	1	prob (Fox)

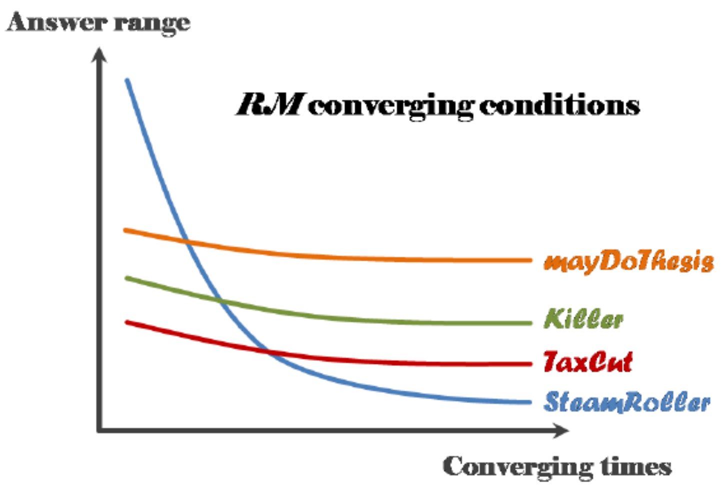


FIGURE 16. Converging conditions of each QA problem

5. **Conclusions.** In this paper, we proposed a new technology about how to generate an efficient and specific solver (C program) corresponding to a given QA Problem. There were three steps in the whole generation process shown as follows.

- *Step 1*: We generated the support set which could be considered as the limited answer searching range. It is the most essential part in the solver generation.
- *Step 2*: We digitalized the generated support set into the index number of the corresponding bit-array, and then digitalized the clause set.
- *Step 3*: A specific solver was generated by using the results of Steps 1 and 2.

Moreover, we proposed the way for initializing the Skolem functions and the efficient way for processing them.

In future work, we will make efforts to simplify the clause set obtained from each given QA Problem in the very beginning by unfold transformation. This may contribute to the reduction of the usage of memory in generating the support set and also the size of the generated specific solver (C program).

REFERENCES

[1] T. Berners-Lee, J. Hendler and O. Lassila, The semantic web, *Scientific American Magazine*, 2008.

[2] S. Tessaris, *Questions and Answers: Reasoning and Querying in Description Logic*, Ph.D. Thesis, The University of Manchester, 2001.

[3] F. M. Donini et al., Al-log: Integrating datalog and description logics, *Journal of Intelligent Information Systems*, vol.16, no.2, pp.227-252, 1998.

[4] A. Y. Levy and M. C. Rousset, Combining horn rules and description logic in Carin, *Artificial Intelligence*, vol.104, no.1-2, pp.165-209, 1998.

[5] K. Akama and E. Nantajeewarawat, Meaning-preserving Skolemization on logical structure, *Proc. of the 9th International Conference on Intelligent Technologies*, pp.123-132, 2008.

[6] K. Akama and E. Nantajeewarawat, Meaning-preserving Skolemization, *International Conference on Knowledge Engineering and Ontology Development*, Paris, France, 2011.

[7] Z. Cheng, K. Akama and T. Tsuchida, Solving "all-solution" problems by ET-based generation of programs, *International Journal of Innovative Computing, Information and Control*, vol.5, no.12(A), pp.4583-4595, 2009.

[8] E. M. Voorhees and D. M. Tice, Building a question answering test collection, *Proc. of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Athens, Greece, pp.200-207, 2000.

[9] T. Mori, M. Nozawa and Y. Asada, Multi-document summarization using a question-answering engine, *The 4th NTCIR Workshop*, Tokyo, Japan, 2004.

[10] Y. Yang, P. Jiang, S. Tsuchiya and F. Ren, Effect of using pragmatics information on question answering system of analects of Confucius, *International Journal of Innovative Computing, Information and Control*, vol.5, no.5, pp.1201-1212, 2009.

[11] S. Staab and R. Studer, *Handbook on Ontologies*, Springer, 2004.

[12] B. Motik, U. Sattler and R. Studer, Query answering for OWL-DL with rules, *Journal of Web Semantics*, vol.3, pp.41-60, 2005.

[13] F. Baader, D. Calvanese et al., *The Description Logic Handbook*, Cambridge University Press, 2001.

[14] *http://en.wikipedia.org/wiki/Characteristic_function.*

[15] K. Akama, E. Nantajeewarawat and H. Koike, Program generation in the equivalent transformation computation model using the squeeze method, *Proc. of PSI2006, LNCS4378*, Springer-Verlag, Heidelberg, Berlin, pp.41-54, 2007.

[16] K. Akama, Hypothesis searching hypose, *ICS, IPSJ*, vol.1986, no.73, pp.89-101, 1986.

[17] M. E. Stickel, Schubert's steamroller problem: Formulation and solution, *Journal of Automated Reasoning*, vol.2, no.1, pp.89-101, 1986.