International Journal of Innovative Computing, Information and Control
Volume **17**, Number **4**, August **2021**
ICIC International ©2021 ISSN 1349-4198
pp. 1119–1135

GENERATION OF LOGICAL EQUIVALENCES BELONGING TO THE C2LE CLASS APPLIED TO PROGRAM SYNTHESIS BASED ON EQUIVALENT TRANSFORMATION

Katsunori Miura[1] and Kiyoshi Akama[2]

[1]Department of Information and Management Science
Otaru University of Commerce
3-5-21, Midori, Otaru, Hokkaido 047-8501, Japan
k-miura@res.otaru-uc.ac.jp

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

Received January 2021; revised May 2021

Abstract. *This paper proposes a framework that automatically generates multiple logical equivalences belonging to the C2LE class from a goal clause. A logical equivalence is a mathematical formula that describes the equivalence of the declarative meaning between two sets of atomic formulas, and is used to generate programs based on equivalent transformation. In the C2LE class, all atomic formulas are mutually connected via variables. The proposed framework consists of two functions, a generator and a filter, newly defined in this paper. The generator first acquires multiple pairs of sets consisting of atomic formulas from a goal clause, and then generates multiple logical equivalences from the obtained pairs by simple formula transformation. The filter uses an instantiation check to remove incorrect logical equivalences that do not specify equivalence between two sets of atomic formulas. An instantiation check is an original method to determine if two definite clauses are equivalent by specializing the targeted clauses. Experimental results indicate that the proposed framework works efficiently to generate logical equivalences that are useful to generate correct and efficient programs. Furthermore, we also show that the proposed framework can generate logical equivalences that cannot be generated using extant frameworks.*
Keywords: Equivalent transformation, Logical equivalence, Goal clause, Generator, Filter

1. Introduction.

Various practical problems in numerous fields, including business and research, use a *satisfiability problem* (SAT) solver [1, 2, 3] to determine answers that completely satisfy multiple constraints. Efficient SAT solvers are generated by the user changing or improving the solver program. This is because the efficiency of an SAT solver is dependent on the problem being solved by the user. However, it is difficult to generate a universal SAT solver that can solve all problems efficiently. Many SAT solvers use the program provided by the developer, so users cannot freely change or improve the program. To solve a variety of practical problems, an SAT solver requires flexibility to adapt to changing problems.

For example, SAT solvers have been applied in frameworks [4, 5, 6, 7] proposed for finding the optimal allocation configuration of cloud resources. These proposed frameworks use quantitative constraints such as computational performance values and budgets as

DOI: 10.24507/ijicic.17.04.1119

user requirements given to the SAT solver. However, when solving practical problems, we also need to deal with conceptual constraints such as dependency graphs, tree diagrams, and flowcharts. That is, there is no guarantee that all constraints required by the user will be available in the existing SAT solver. If user requirements are simple constraints that are easy to formulate, the existing SAT solvers can work sufficiently. However, there are complex constraints such as service level agreements and legal/policy requirements that are not easily determined by less skilled users. Not all constraints needed to determine the answer are always completely given by the user. Specifically, there are situations where an SAT solver is given an incomplete set of constraints. Therefore, we need a framework that can generate an appropriate SAT solver by changing or improving the solver program.

An *equivalent transformation* (ET) [8, 9, 10, 11] provides a framework that allows users to generate efficient solver programs. In the framework, a problem is specified using definite clauses. A program to solve the problem is a set of rewriting rules called *ET rules*, each of which replaces one of the definite clauses, called a *goal clause*, with one or more clauses while preserving the declarative meaning of the union of the original clause and problem clauses. Users can dynamically improve their solver program by adding new ET rules that are useful for finding answers efficiently during calculations. For example, the framework proposed in [12, 13] generates ET rules based on the *multi-objective genetic algorithm* (MOGA) [14, 15] evaluation results to efficiently find the optimal allocation configuration of cloud resources. Thus, ET rules are important for generating SAT solvers that can be flexibly improved in response to problems. As another framework for generating efficient SAT solvers, *constraint handling rules* (CHR) was proposed by Frühwirth [16, 17]. In CHR, solver programs are generated by describing procedural knowledge using logical formulas that describe equivalence between constraints.

This paper proposes a framework that automatically generates multiple *logical equivalences* (LEs) belonging to the C2LE class from a goal clause. The proposed framework can be used to generate ET rules that apply to a goal clause. An LE [18] is a mathematical formula that describes the equivalence of the declarative meaning between two sets of *atomic formulas* (atoms). The C2LE class is a new LE class defined in this paper. In the C2LE class, all atoms are mutually connected via variables. LEs belonging to the C2LE class help generate many ET rules needed to solve the practical problem. Moreover, there are many ET rules that can only be generated from LEs belonging to the C2LE class. Therefore, the C2LE class is very helpful in improving the solver program to generate efficient SAT solvers.

Miura et al. [18] proposed a method for generating ET rules via LEs, and showed that the LE is useful for generating various classes of ET rules. The method generates ET rules according to the following process: (i) generate a set of LEs from a goal clause, regardless of the correctness of the LEs; (ii) determine a set of correct LEs by removing incorrect LEs based on proof; (iii) generate an ET rule from a correct LE guaranteed in step (ii). A correct LE specifies equivalence between two sets of atoms; it is defined in Section 2.3. The objective of this paper is to propose a new framework for step (i) that automatically generates LEs belonging to the C2LE class.

Previous studies [18, 19] have proposed frameworks for steps (ii) and (iii) to prove the correctness of LEs belonging to the ES class and generate ET rules from them. [20, 21] have also proposed frameworks for step (i) that generate LEs belonging to the Speq and False classes. However, LEs belonging to the C2LE class differ from those of both the Speq and False classes. Thus, this paper proposes a new LE generation framework that is different from that proposed in [20, 21]. LEs belonging to the C2LE, Speq, and False classes are special formulas in the ES class. The syntax of LEs has a set of atoms on both sides of the double-headed arrow. The left-hand side is determined from the body part

of a goal clause, and the right-hand side is generated based on the constraint conditions of the LE class. The right-hand side of the Speq and False classes is generated by simply adding the specified atom. In contrast, the right-hand side of the C2LE class needs to generate a new set of atoms based on various constraint conditions. Thus, this paper provides a more advanced framework than those reported in previous studies.

The proposed framework consists of two functions, a *generator* and a *filter*, newly defined in this paper. When generating LEs from a goal clause, the generator runs first, and then the filter is applied. The generator is used to generate multiple LEs belonging to the C2LE class from a goal clause, regardless of the correctness of the LEs. The filter is used to remove multiple incorrect LEs that do not specify any equivalence between two sets of atoms from the generated LEs.

The generator first selects a set $\mathcal{E}_1$ of atoms from the body part of a goal clause, and then generates a different set $\mathcal{E}_2$ of atoms based on $\mathcal{E}_1$ and the constraint conditions of the C2LE class. An LE is determined from a pair of $\mathcal{E}_1$ and $\mathcal{E}_2$ by simple formula transformation. This approach can efficiently generate LEs used to generate ET rules applicable to a goal clause, whereas, the filter first performs an *instantiation check* for an LE, and then removes the LE if the result is false. An instantiation check is an original method to determine if two definite clauses are equivalent by specializing the targeted clauses. The approach can efficiently remove multiple incorrect LEs only from the generated LEs.

An instantiation check is a method for finding counter-examples of equivalence by specializing the target; it is not a mathematical proof for a logical formula as in [22, 23, 24]. Thus, the proposed framework does not guarantee the correctness of the generated LE. Their correctness is proven in step (ii) of ET rule generation using LE. Reducing the number of incorrect LEs can shorten the execution time of step (ii). The proposed framework can improve the time efficiency of ET rule generation by finding the incorrect LEs in less time than using a mathematical proof.

This paper experimentally evaluates whether the proposed framework can generate multiple LEs belonging to the C2LE class from a goal clause within a practical amount of time. The experiment uses goal clauses that appear in the actual computation of problem solving. By using the actual goal clauses, we show that the proposed framework can generate LEs that help generate ET rules used in the actual computations.

The remainder of this paper is organized as follows. Section 2 defines the LE class treated in this paper and provides examples of LEs that belong to that class. Section 3 describes the generator and filter functions, which are the two components of the proposed framework. We also explain the process of using the generator to generate LEs from a goal clause and applying the filter to removing incorrect LEs from the generated LEs. Section 4 illustrates LE generation from a concrete goal clause according to the steps of the generator. This section also illustrates incorrect LE removal using a concrete LE according to the steps of the filter. Section 5 discusses the effectiveness of LE generation using the proposed framework based on experimental evaluation. Section 6 concludes this paper.

2. **Logical Equivalences.**

2.1. **Closely connected atom set.**

A *closely connected atom set* is an atom set consisting of atoms that are mutually connected via variables. In ET computation, it is important to find a closely connected atom set from the body part of a goal clause. A closely connected atom set is given by Definition 2.3. Let $\mathbb{A}$ be a set of atoms.

Definition 2.1. *Direct link*

e_1 has a direct link to e_2 in $\mathbb{A}$ if there is a variable v such that

1) $e_1 \in \mathbb{A}$*,*
2) $e_2 \in \mathbb{A}$*,*
3) v *appears in* e_1*,*
4) v *appears in* e_2*.*

In this paper, the variable v shown in Definition 2.1 is called a *link variable.*

Definition 2.2. *Connection path*
e_1 *has a connection path to* e_n *in* $\mathbb{A}$ *if there is a sequence of atoms* $[e_1, e_2, \ldots, e_n]$ *such that for all* $k \in \{1, 2, \ldots, n-1\}$, e_k *has a direct link to* e_{k+1} *in* $\mathbb{A}$.

Definition 2.3. *Closely connected atom set*
$\mathbb{A}$ *is a closely connected atom set if for all* e_1 *and* e_2 *in* $\mathbb{A}$, e_1 *has a connection path to* e_2 *in* $\mathbb{A}$.

2.2. **Definition and syntax of LEs.** An LE [18] describes the equivalence of the declarative meaning between two sets of atoms according to first-order predicate logic [25, 26]. Various LE classes can be determined by changing the constraints of an atom set. The LEs treated in this paper are generated using the closely connected atom set, and are said to belong to the *closely connected LE* (C2LE) class. The syntax for LEs that belong to the C2LE class is as follows:

$$\forall_{\overline{v}}(\exists_{\overline{v}_1}\mathcal{E}_1 \leftrightarrow \exists_{\overline{v}_2}\mathcal{E}_2), \tag{1}$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are closely connected atom sets, $\overline{v}_1$ is a set of variables that only appear in $\mathcal{E}_1$, $\overline{v}_2$ is a set of variables that only appear in $\mathcal{E}_2$, and $\overline{v}$ is a set of variables that appear in both $\mathcal{E}_1$ and $\mathcal{E}_2$.

2.3. **Correctness of LEs.** The correctness of LEs depends on the declarative meaning of predicates. Predicates are defined by definite clauses. Given a set D of definite clauses, $\mathcal{M}(D)$ is the declarative meaning of D and is defined as a minimal model of D [8]. An LE is correct with respect to D iff $\mathcal{M}(D)$ is a model of the LE.

2.4. **LE examples.** This section gives LE examples that are written using the *app* and *rev* predicates. The *app* predicate is a user-defined predicate that specifies the concatenation of two lists. Let us explain the *app* predicate using $app([1, 2], [3], A)$. This atom means that variable A is a concatenated list of $[1, 2]$ and $[3]$, that is, $[1, 2, 3]$. The *rev* predicate is a user-defined predicate that specifies the reverse order of elements in a list. Let us explain the *rev* predicate using $rev([1, 2], A)$. This atom means that variable A is a reverse list of the elements in $[1, 2]$.

For example, the following LEs belong to the C2LE class.

$$le_1 : \forall_{\{A,B,D,E\}} \left(\exists_{\{C\}}\{app(A, [B], C), app(C, [D], E)\} \leftrightarrow \{app(A, [B, D], E)\}\right)$$

$$le_2 : \forall_{\{A,B,D,E\}} \left(\exists_{\{C\}}\{app(A, B, C), app(C, [D], E)\} \leftrightarrow \exists_{\{V\}}\{app(B, [D], V), app(A, V, E)\}\right)$$

$$le_3 : \forall_{\{A,B,D\}} \big(\exists_{\{C\}}\{app(A, B, C), rev(C, D)\} \leftrightarrow \exists_{\{V,W\}}\{rev(A, V), rev(B, W), app(W, V, D)\}\big)$$

In le_1, $\mathcal{E}_1$ is $\{app(A, [B], C), app(C, [D], E)\}$ and $\mathcal{E}_2$ is $\{app(A, [B, D], E)\}$. This LE specifies that $\exists_{\{C\}}\{app(A, [B], C), app(C, [D], E)\}$ and $\{app(A, [B, D], E)\}$ are equivalent for all ground substitutions for variables A, B, D, and E.

In le_2, $\mathcal{E}_1$ is $\{app(A, B, C), app(C, [D], E)\}$ and $\mathcal{E}_2$ is $\{app(B, [D], V), app(A, V, E)\}$. This LE specifies that $\exists_{\{C\}}\{app(A, B, C), app(C, [D], E)\}$ and $\exists_{\{V\}}\{app(B, [D], V), app(A, V, E)\}$ are equivalent for all ground substitutions for variables A, B, D, and E.

In le_3, $\mathcal{E}_1$ is $\{app(A,B,C), rev(C,D)\}$ and $\mathcal{E}_2$ is $\{rev(A,V), rev(B,W), app(W,V,D)\}$. This LE specifies that $\exists_{\{C\}}\{app(A,B,C), rev(C,D)\}$ and $\exists_{\{V,W\}}\{rev(A,V), rev(B,W), app(W,V,D)\}$ are equivalent for all ground substitutions for variables A, B, and D.

3. Logical Equivalence Generation Framework.

3.1. Program synthesis and LE generation.

A specification is denoted $\langle\mathbb{D},\mathbb{Q}\rangle$, where $\mathbb{D}$ is a set of definite clauses, called *background knowledge*, and $\mathbb{Q}$ is a set of queries, where a query is a set of definite clauses. We assume that for any definite clause q appearing in $\mathbb{Q}$, any predicate appearing in the head part of q exists neither in $\mathbb{D}$ nor in the body part of q. A program is a set of ET rules. Program synthesis is the process of generating a correct and sufficiently efficient program with respect to $\langle\mathbb{D},\mathbb{Q}\rangle$.

Using ET rules, we have a goal sequence $[G_1, G_2, \ldots, G_n]$, where $G_1 = Q$, and for any i in $\{1, 2, \ldots, n-1\}$, G_{i+1} is obtained from G_i using an ET rule. Any definite clause appearing in a goal sequence is called a *goal clause*. For any query Q in $\mathbb{Q}$, answers with respect to Q and $\mathbb{D}$ are obtained from G_n. An ET rule applies to a definite clause g in G_n and replaces g with one or more clauses while preserving $\mathcal{M}(\mathbb{D}\cup G_n)$. Thus, to generate a program, we generate ET rules that can be applied to a goal clause.

As one of the methods to generate ET rules, the *LE-based method* has been proposed [18]. The LE-based method generates ET rules according to the following process: (i) generate a set $\mathbb{S}$ of LEs from a goal clause g in G_n; (ii) remove incorrect LEs with respect to $\mathbb{D}$ from $\mathbb{S}$ based on proof; (iii) generate an ET rule from a correct LE in $\mathbb{S}$.

For example, let us generate an ET rule that applies to goal clause g_1 using the LE-based method. An *ans* atom is a special atom that represents the answer.

$$g_1\colon ans([D|F], G, E) \leftarrow \underline{app(A,B,C), app(C,[D],E)}, rev(F,B), rev(G,A) \tag{2}$$

The underlined part of g_1 is a closely connected atom set. The following LE is one of the LEs that can be generated from g_1 and belongs to the C2LE class.

$$\forall_{\{A,B,D,E\}}\left(\exists_{\{C\}}\{app(A,B,C), app(C,[D],E)\} \leftrightarrow \exists_{\{V\}}\{app(B,[D],V), app(A,V,E)\}\right)$$

Subsequently, in step (ii), the correctness of this LE is proven, but the explanation of proof is omitted. Finally, given this LE, the following ET rule is generated.

$$app(A,B,C), app(C,[D],E), \{isolated(C)\} \Rightarrow app(B,[D],V), app(A,V,E)$$

$isolated(C)$ is an applicable condition, which means that the variable C does not appear in atoms other than the underlined part of g_1. This ET rule can be applied to g_1.

In the LE-based method, it is important to increase the LE classes that can be generated from a goal clause. Previous studies [20, 21] have provided frameworks for generating LEs that belong to the Speq and False classes. However, the frameworks proposed in [20, 21] cannot generate LEs that belong to the C2LE class, which is newly defined in this paper. The C2LE class is needed to generate several ET rules contained in a program that proves the correctness of the specified LE [19, 21]. This paper proposes a new framework for generating LEs that belong to the C2LE class.

3.2. Outline of LE generation framework for C2LE class.

To efficiently generate LEs belonging to the C2LE class, this paper proposes two functions, a *generator* and a *filter*, as components of the proposed framework.

The generator is used to generate multiple useful LEs belonging to the C2LE class from a goal clause. The C2LE class contains many LEs that help generate ET rules needed to solve problems. However, the C2LE class also contains useless LEs, so without guidelines to narrow the search range of LEs, the cost of generating useful LEs can be high. To generate useful LEs at low cost, the generator focuses on the repositioning of

the arguments that appear in a set of atoms. The generator acquires LEs, regardless of their correctness with respect to $\mathbb{D}$.

The filter is used to remove multiple incorrect LEs from a set of the generated LEs. The correctness of LEs is proven in step (ii) of the LE-based method. As a result of the proof, incorrect LEs are removed. However, the cost of building a computational environment for proof is high, and the calculation to prove their correctness is time-consuming. Therefore, it is better to reduce the number of incorrect LEs before performing step (ii). To remove incorrect LEs at low cost, the filter uses an *instantiation check*. An instantiation check is an original method to determine if two definite clauses are equivalent by specializing the targeted clauses.

As shown in Figure 1, the generator first generates a set $\mathbb{S}$ of LEs from a goal clause g in G_n; subsequently, the filter removes incorrect LEs from $\mathbb{S}$. The initial state of $\mathbb{S}$ is an empty set. The proposed framework allows incorrect LEs to occur in $\mathbb{S}$. Incorrect LEs that remain in $\mathbb{S}$ are removed based on proof in step (ii) of the LE-based method.

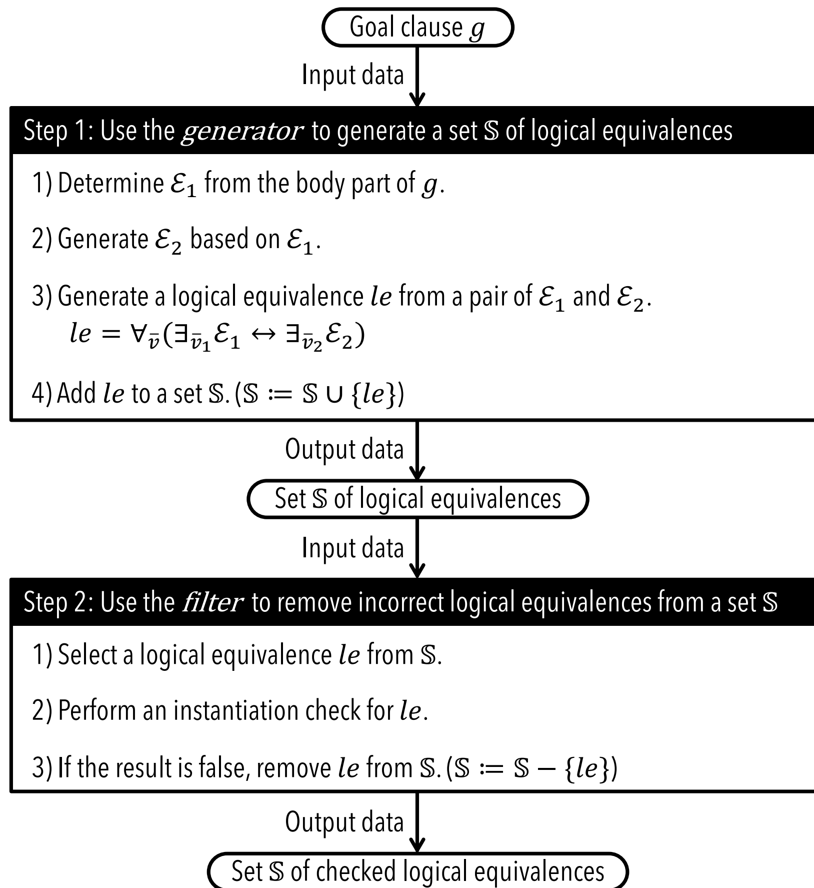

FIGURE 1. Process to generate a set $\mathbb{S}$ of checked LEs from a goal clause g

The generator creates a set $\mathbb{S}$ according to the following process: 1) determine $\mathcal{E}_1$ of an LE from the body part of g; 2) generate $\mathcal{E}_2$ of the LE based on $\mathcal{E}_1$; 3) generate the le defined in Formula (1) from the pair of $\mathcal{E}_1$ and $\mathcal{E}_2$; 4) add le to $\mathbb{S}$.

The filter removes incorrect LEs according to the following process: 1) select le from a set $\mathbb{S}$ of LEs; 2) perform an instantiation check for le; 3) if the result is false, remove le from $\mathbb{S}$.

3.3. **Generation of a set of LEs using the generator.**

3.3.1. *Determination of $\mathcal{E}_1$ in LE.* The generator selects a closely connected atom set from the body part of a goal clause g. In step (iii) of the LE-based method, the head part of an ET rule is generated using $\mathcal{E}_1$ of an LE. Thus, to generate an ET rule that applies to a goal clause g, $\mathcal{E}_1$ should be determined based on the body part of g.

Let $\mathbb{K}$ be a set of built-in predicates. Given a definite clause c, $body(c)$ is a set of body atoms that appear in c. Given a set A of atoms, $pow(A)$ is the power set of A and $prd(A)$ is the set of all predicates that appear in A.

Given a goal clause g ($\in G_n$), $\mathcal{E}_1$ is determined according to the following process: 1) generate a set W from g, where W is $pow(body(g))$; 2) acquire a set B_1 of $\mathcal{E}_1$ candidates from W, where $B_1 \subseteq W$, and for all e in B_1, e is a closely connected atom set, and $prd(e) \cap \mathbb{K} = \emptyset$; 3) select $\mathcal{E}_1$ from B_1.

3.3.2. *Generation of $\mathcal{E}_2$ in LE.* A huge variety of $\mathcal{E}_2$ can be generated based on a given $\mathcal{E}_1$. Thus, guidelines are required to preferentially generate useful $\mathcal{E}_2$ candidates at low cost. One method to generate useful $\mathcal{E}_2$ candidates is to reposition the arguments that appear in $\mathcal{E}_1$. When generating some programs that prove the correctness of the LE, we need an ET rule that changes the position of the argument. In the following ET rule, the positions of the arguments A, $[B]$, and $[D]$ in the head part are changed in the body part.

$$app(A,[B],C), app(C,[D],E), \{isolated(C)\} \Rightarrow app([B],[D],V), app(A,V,E)$$

In step (iii) of the LE-based method, the body part of an ET rule is generated using $\mathcal{E}_2$ of an LE.

The generator determines $\mathcal{E}_2$ candidates by changing the position of the arguments appearing in $\mathcal{E}_1$. When generating $\mathcal{E}_2$, the predicate appearing in $\mathcal{E}_1$ is used. The argument candidates of $\mathcal{E}_2$ are as follows: (a1) arguments other than the link variable that appear in $\mathcal{E}_1$; (a2) concatenated lists of finite lists that appear in $\mathcal{E}_1$; (a3) link variables used in $\mathcal{E}_2$, where any link variable does not appear in $\mathcal{E}_1$. The number of link variables used in $\mathcal{E}_2$ depends on the number of atoms that appear in $\mathcal{E}_2$. If $\mathcal{E}_2$ consists of n atoms, then at least $n-1$ link variables are needed. For example, given $\{app(A,[B],C), app(C,[D],E)\}$ as $\mathcal{E}_1$, the set (a1) is $\{A,[B],[D],E\}$ and the set (a2) is $\{[B,D],[D,B]\}$. Moreover, if $\mathcal{E}_2$ consisting of three atoms is generated, the set (a3) is $\{V,W\}$.

Given a set A of atoms, $arg(A)$ is the set of all arguments that appear in A, $var(A)$ is the set of all variables that appear in A, $lnv(A)$ is the set of variables that do not appear in $var(A)$, $cmb(A)$ is the set of concatenated lists of finite lists that appear in $arg(A)$, and $dlv(A)$ is the set of all link variables that appear in A. Given a set P of predicates and a set T of arguments, $atoms(P,T)$ is the set of atoms that are determined using a predicate in P and arguments in T.

Given $\mathcal{E}_1$ of an LE, $\mathcal{E}_2$ is generated according to the following process: 1) determine a set P of predicates, where P is $prd(\mathcal{E}_1)$; 2) determine a set T of arguments, where T is $(arg(\mathcal{E}_1) - dlv(\mathcal{E}_1)) \cup cmb(\mathcal{E}_1) \cup lnv(\mathcal{E}_1)$; 3) generate a set B_2 of $\mathcal{E}_2$ candidates, where for all e in B_2, e is a closely connected atom set, $e \subseteq atoms(P,T)$, $(var(e) - dlv(e)) = (var(\mathcal{E}_1) - dlv(\mathcal{E}_1))$, $prd(e) = prd(\mathcal{E}_1)$, any v_1 in $(var(e) - dlv(e))$ appears once in e, and any v_2 in $dlv(e)$ appears twice in e; 4) select $\mathcal{E}_2$ from B_2.

3.3.3. *Generation of LEs using $\mathcal{E}_1$ and $\mathcal{E}_2$.* The generator determines an LE from a pair of $\mathcal{E}_1$ and $\mathcal{E}_2$ by simple formula transformation. All pairs of $\mathcal{E}_1$ and $\mathcal{E}_2$ are used, regardless of the correctness of the LE with respect to $\mathbb{D}$. As shown in Formula (1), $\mathcal{E}_1$ is written as the left-hand side in the LE and $\mathcal{E}_2$ is written as the right-hand side in the LE. $var(\mathcal{E}_1) \cap var(\mathcal{E}_2)$ is a set of universally quantified variables of $\mathcal{E}_1$ and $\mathcal{E}_2$ and is written to $\overline{v}$.

$var(\mathcal{E}_1)-\overline{v}$ is a set of existentially quantified variables of $\mathcal{E}_1$ and is written to $\overline{v}_1$. $var(\mathcal{E}_2)-\overline{v}$ is a set of existentially quantified variables of $\mathcal{E}_2$ and is written to $\overline{v}_2$.

3.4. **Removal of incorrect LEs using the filter.**

3.4.1. *Generation of two definite clauses.* When removing incorrect LEs from a set $\mathbb{S}$ using an instantiation check, the filter first generates two definite clauses c_1 and c_2 from an LE, and then performs an instantiation check for c_1 and c_2. c_1 is defined as $(H \leftarrow X_1, \ldots, X_n)$ and c_2 is defined as $(H \leftarrow Y_1, \ldots, Y_m)$, where H is an *ans* atom, X_n and Y_m are atoms. c_1 and c_2 can be generated from an LE by simple formula transformation.

Given le $(= \forall_{\overline{v}}(\exists_{\overline{v}_1}\mathcal{E}_1 \leftrightarrow \exists_{\overline{v}_2}\mathcal{E}_2))$ in $\mathbb{S}$, $H = ans(\overline{v})$, $X_1, \ldots, X_n = \mathcal{E}_1$, and $Y_1, \ldots, Y_m = \mathcal{E}_2$. If an LE is correct with respect to $\mathbb{D}$, for all ground substitutions for $\overline{v}$, the answers obtained from c_1 are completely equal to the answers obtained from c_2. In the filter, a tag u is attached to a variable appearing in $\overline{v}$. If a variable X appears in $\overline{v}$, X is written as $X^{\sim u}$.

For example, let us generate c_1 and c_2 from the following LE.

$$\forall_{\{A,B,D\}}\left(\exists_{\{C\}}\{app(A,B,C), rev(C,D)\} \leftrightarrow \exists_{\{V,W\}}\{rev(A,V), rev(B,W), app(W,V,D)\}\right)$$

Variables A, B, and D appear in $\overline{v}$, so they are written as $A^{\sim u}$, $B^{\sim u}$, and $D^{\sim u}$, respectively. Therefore, c_1 and c_2 are written as follows:

$$c_1 = ans(A^{\sim u}, B^{\sim u}, D^{\sim u}) \leftarrow app(A^{\sim u}, B^{\sim u}, C), rev(C, D^{\sim u}),$$
$$c_2 = ans(A^{\sim u}, B^{\sim u}, D^{\sim u}) \leftarrow rev(A^{\sim u}, V), rev(B^{\sim u}, W), app(W, V, D^{\sim u})$$

An ET rule is generated from a correct LE with respect to $\mathbb{D}$; thus it is required that the correctness of the LE be guaranteed. le is correct with respect to $\mathbb{D}$ iff $\mathcal{M}(\mathbb{D} \cup \{c_1\}) = \mathcal{M}(\mathbb{D} \cup \{c_2\})$. However, the computational cost required to prove this equivalence relationship is high. An instantiation check is a low-cost method for checking non-equivalence of a relationship of c_1 and c_2.

3.4.2. *Instantiation check.* The filter removes several incorrect LEs from a set $\mathbb{S}$ based on the equivalence of the relationship between two definite clauses. When checking for non-equivalence at low cost, it is helpful to find counter-examples of equivalence by specializing the definite clauses. An instantiation check determines if a relationship between two definite clauses c_1 and c_2 is equivalent by specializing c_1 and c_2. An instantiation check is a more time-efficient method than proof to find many incorrect LEs, and it only removes the incorrect LEs, not the correct LEs.

The relationship between c_1 and c_2 is specified as a *relation* atom that is defined as $relation([H|(X_1, \ldots, X_n)], [H|(Y_1, \ldots, Y_m)])$. Definite clause d represents that the relationship between c_1 and c_2 is equivalent. *yes* is an atom with no arguments.

$$d\colon yes \leftarrow relation([H|(X_1, \ldots, X_n)], [H|(Y_1, \ldots, Y_m)])$$

In the instantiation check, a clause set $\mathcal{R}$ $(= \{d\})$ consisting of d is generated, and the state of $\{d\}$ is replaced using ET rules. A ground substitution δ_1 is applied to a variable in $var(\{H\})$ before replacing the clause set $\mathcal{R}$ using ET rules. δ_1 is determined based on the definition of predicates. δ_1 specializes in one variable, not multiple variables. The following state is obtained by applying δ_1 to $\mathcal{R}$.

$$\mathcal{R}\delta_1 = \{yes \leftarrow relation([H\delta_1|(X_1\delta_1, \ldots, X_n\delta_1)], [H\delta_1|(Y_1\delta_1, \ldots, Y_m\delta_1)])\}$$

The initial state of clause replacement using ET rules is $\mathcal{R}\delta_1$, and clause replacement is repeated until there are no applicable ET rules. Clause replacement uses ET rules that apply to a *relation* atom.

As a result of clause replacement, if a relationship between c_1 and c_2 is equivalent on some ground instances, then the same state is obtained from both c_1 and c_2. $\mathcal{R}_1$ and $\mathcal{R}_2$ shown in Formula (3) represent that c_1 and c_2 are in the same state. Conversely, if the relationship between c_1 and c_2 is not equivalent on some ground instances, then different states are obtained. $\mathcal{R}_3$ and $\mathcal{R}_4$ represent that c_1 and c_2 are in different states. σ is a substitution obtained by applying ET rules. δ_2 is a ground substitution applied to a variable in $var(\{H\delta_1\sigma\}) \cap var(\{H\})$. δ is $\delta_1 \cup \delta_2$. In Formula (3), an empty list $[\,]$ represents an empty set, and $[H\delta\sigma]$ represents a singleton consisting of a unit clause $(H\delta\sigma \leftarrow)$.

$$\begin{aligned}
\mathcal{R}_1 &= \{yes \leftarrow relation([\,],[\,])\} \\
\mathcal{R}_2 &= \{yes \leftarrow relation([H\delta\sigma],[H\delta\sigma])\} \\
\mathcal{R}_3 &= \{yes \leftarrow relation([H\delta\sigma],[\,])\} \\
\mathcal{R}_4 &= \{yes \leftarrow relation([\,],[H\delta\sigma])\}
\end{aligned} \tag{3}$$

If $\mathcal{R}_1$ or $\mathcal{R}_2$ is obtained, the instantiation check returns *true*, whereas if $\mathcal{R}_3$ or $\mathcal{R}_4$ is obtained, it returns *false*.

If the current state of a clause set does not exist in Formula (3), then clause replacement continues after applying a ground substitution δ_2. δ_2 is determined so that $\mathcal{R}_3$ or $\mathcal{R}_4$ is obtained. This is because it uses an instantiation check to find incorrect LEs from $\mathbb{S}$.

4. **Example of LE Generation Using the Proposed Framework.**

4.1. **Generating a set of LEs from a concrete goal clause.**

4.1.1. *Goal clause used in this example.* This example uses the following goal clause as the input to the generator. Multiple closely connected atom sets appear in the body part of g_1. This goal clause is the same as Formula (2) shown in Section 3.1.

$$g_1\colon ans([D|F], G, E) \leftarrow app(A, B, C), app(C, [D], E), rev(F, B), rev(G, A)$$

4.1.2. *Example of $\mathcal{E}_1$ determined from the specified goal clause.* First, a set W is generated from goal clause g_1. W is a power set of atoms that appear in the body part of g_1. The atoms that appear in the body part of g_1 are $app(A, B, C)$, $app(C, [D], E)$, $rev(F, B)$, and $rev(G, A)$. Therefore, the following union of b_1, b_2, b_3, and b_4 is determined as a set W generated from g_1, where an empty set is removed from W. The element appearing in b_1 is a set of one atom, the element appearing in b_2 is a set of two atoms, the element appearing in b_3 is a set of three atoms, and the element appearing in b_4 is a set of four atoms.

$$W = b_1 \cup b_2 \cup b_3 \cup b_4$$

where b_1, b_2, b_3, and b_4 are as follows:

$$\begin{aligned}
b_1 = \{&\{app(A, B, C)\}, \{app(C, [D], E)\}, \{rev(F, B)\}, \{rev(G, A)\}\} \\
b_2 = \{&\{app(A, B, C), app(C, [D], E)\}, \{app(A, B, C), rev(F, B)\}, \\
&\{app(A, B, C), rev(G, A)\}, \{app(C, [D], E), rev(F, B)\}, \\
&\{app(C, [D], E), rev(G, A)\}, \{rev(F, B), rev(G, A)\}\} \\
b_3 = \{&\{app(A, B, C), app(C, [D], E), rev(F, B)\}, \\
&\{app(A, B, C), app(C, [D], E), rev(G, A)\}, \\
&\{app(A, B, C), rev(F, B), rev(G, A)\}, \{app(C, [D], E), rev(F, B), rev(G, A)\}\} \\
b_4 = \{&\{app(A, B, C), app(C, [D], E), rev(F, B), rev(G, A)\}\}
\end{aligned}$$

Next, the set B_1 of $\mathcal{E}_1$ candidates is determined from the set W. B_1 is the set of all closely connected atom sets that appear in W. Moreover, for all e in B_1, built-in predicates are not used in any atom in e. In this example, no built-in predicate occurs. Therefore, the following set is determined as the set B_1 of $\mathcal{E}_1$ candidates.

$$\begin{aligned} B_1 = \{&\{app(A,B,C), rev(G,A)\}, \{app(A,B,C), rev(F,B)\},\\ &\{app(A,B,C), app(C,[D],E)\}, \{app(A,B,C), rev(F,B), rev(G,A)\},\\ &\{app(A,B,C), app(C,[D],E), rev(G,A)\},\\ &\{app(A,B,C), app(C,[D],E), rev(F,B)\}\} \end{aligned}$$

Finally, $\mathcal{E}_1$ is selected from the set B_1. Any element in B_1 can be selected as $\mathcal{E}_1$. In this example, the following atom set is selected as $\mathcal{E}_1$. The developed generator software will try to generate LEs for all $\mathcal{E}_1$ candidates.

$$\mathcal{E}_1 = \{app(A,B,C), app(C,[D],E)\}$$

4.1.3. *Example of $\mathcal{E}_2$ generated based on the selected $\mathcal{E}_1$.* First, a set P of predicates and a set T of arguments are generated based on $\{app(A,B,C), app(C,[D],E)\}$ selected as $\mathcal{E}_1$. P is the set of all predicates that appear in $\mathcal{E}_1$. Therefore, P is as follows:

$$P = \{app\}$$

T is the set of argument candidates (a1), (a2), and (a3) defined in Section 3.3.2. The argument candidates of (a1) are A, B, $[D]$, and E. There are no argument candidates of (a2) because no concatenated list is generated. The number of argument candidates of (a3) depends on the number of atoms appearing in $\mathcal{E}_2$. This example assumes that $\mathcal{E}_2$ is composed of two atoms. From this assumption, a variable V is determined as an argument candidate of (a3) because it requires at least one link variable that does not appear in $\mathcal{E}_1$. Therefore, T is as follows:

$$T = \{A, B, [D], E, V\}$$

Next, a set B_2 of $\mathcal{E}_2$ candidates is generated using atoms determined by the sets P and T. Moreover, for all e in B_2, e is a closely connected atom set, $e \subseteq atoms(\{app\}, \{A, B, [D], E, V\})$, $(var(e) - dlv(e)) = \{A, B, D, E\}$, $prd(e) = \{app\}$, each of A, B, D, and E appears once in e, and V appears twice in e. From the above assumption about the number of atoms appearing in $\mathcal{E}_2$, the $\mathcal{E}_2$ candidate is composed of two *app* atoms, and is written as follows:

$$\{app(t_1, t_2, t_3), app(t_4, t_5, t_6)\}$$

where $t_1, \dots, t_6$ are assigned arguments in T. By applying link variable V to this set of atoms, we can obtain the following nine patterns.

$$\begin{array}{lll} \{app(V,t_2,t_3), app(V,t_5,t_6)\} & \{app(V,t_2,t_3), app(t_4,V,t_6)\} & \{app(V,t_2,t_3), app(t_4,t_5,V)\}\\ \{app(t_1,V,t_3), app(V,t_5,t_6)\} & \{app(t_1,V,t_3), app(t_4,V,t_6)\} & \{app(t_1,V,t_3), app(t_4,t_5,V)\}\\ \{app(t_1,t_2,V), app(V,t_5,t_6)\} & \underline{\{app(t_1,t_2,V), app(t_4,V,t_6)\}} & \{app(t_1,t_2,V), app(t_4,t_5,V)\} \end{array}$$

The remaining arguments of each pattern are assigned A, B, $[D]$, and E. B_2 consists of all $\mathcal{E}_2$ candidates that are generated by applying A, B, $[D]$, and E to each pattern. When the remaining arguments are assigned A, B, $[D]$, and E, twenty four $\mathcal{E}_2$ candidates are generated from each pattern. Therefore, the set B_2 consists of two hundred and sixteen $\mathcal{E}_2$ candidates. The details of the $\mathcal{E}_2$ candidates generated from each pattern are omitted. Some of the $\mathcal{E}_2$ candidates generated from the underlined pattern are as follows:

$$\begin{array}{lll} \{app(A,B,V), app([D],V,E)\}, & \{app(B,[D],V), app(A,V,E)\}, & \{app(E,B,V), app([D],V,A)\},\\ \{app(A,E,V), app(B,V,[D])\}, & \{app([D],E,V), app(B,V,A)\}, & \{app(E,A,V), app([D],V,B)\} \end{array}$$

Finally, $\mathcal{E}_2$ is selected from the set B_2. Any element in B_2 can be selected as $\mathcal{E}_2$. In this example, the following atom set is selected as $\mathcal{E}_2$. The developed generator software will generate pairs of $\mathcal{E}_1$ and $\mathcal{E}_2$ for all $\mathcal{E}_2$ candidates.

$$\mathcal{E}_2 = \{app(A, B, V), app([D], V, E)\}$$

4.1.4. *Example of LE generated using the selected $\mathcal{E}_1$ and $\mathcal{E}_2$.* Let us generate an LE using $\{app(A, B, C), app(C, [D], E)\}$ as $\mathcal{E}_1$ and $\{app(A, B, V), app([D], V, E)\}$ as $\mathcal{E}_2$. $\mathcal{E}_1$ is written as the left-hand side in the LE and $\mathcal{E}_2$ is written as the right-hand side in the LE. The variables that appear in both $\mathcal{E}_1$ and $\mathcal{E}_2$ are A, B, D, and E, so $\overline{v}$ is $\{A, B, D, E\}$. Variable C only appears in $\mathcal{E}_1$, so $\overline{v}_1$ is $\{C\}$. Variable V only appears in $\mathcal{E}_2$, so $\overline{v}_2$ is $\{V\}$. Therefore, given $\{app(A, B, C), app(C, [D], E)\}$ and $\{app(A, B, V), app([D], V, E)\}$, the following LE is generated. The generated LE is added to a set $\mathbb{S}$ of LEs.

$$le_4 : \forall_{\{A,B,D,E\}} \left(\exists_{\{C\}}\{app(A, B, C), app(C, [D], E)\} \leftrightarrow \exists_{\{V\}}\{app(A, B, V), app([D], V, E)\}\right)$$

4.2. **Removing incorrect LEs using instantiation check.**

4.2.1. *Incorrect LE used in this example.* We will use the LE le_4 shown in the previous section as the input to the filter. When performing an instantiation check for le_4, definite clauses c_1 and c_2 are generated from le_4. This LE is not correct with respect to $\mathbb{D}$. In $\mathcal{E}_1$ of le_4, D is the last element of list E, whereas in $\mathcal{E}_2$, D is the first element of list E. Therefore, an instantiation check returns *false* because the relationship between c_1 and c_2 is not equivalent.

4.2.2. *Examples of two definite clauses generated from the specified LE.* First, the head parts of definite clauses c_1 and c_2 are generated based on le_4. Their head parts are the same *ans* atom. The variables appearing in $\overline{v}$ are used as arguments for an *ans* atom. The variables appearing in $\overline{v}$ of le_4 are A, B, D, and E. Therefore, $ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u})$ is the head part of each c_1 and c_2. A tag u is attached to the variable appearing in $\overline{v}$.

Next, the body parts of c_1 and c_2 are generated based on le_4. The body part of c_1 is $\mathcal{E}_1$ of le_4, whereas the body part of c_2 is $\mathcal{E}_2$ of le_4. Therefore, c_1 and c_2 are written as follows:

$$c_1 = ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u}) \leftarrow app(A^{\sim u}, B^{\sim u}, C), app(C, [D^{\sim u}], E^{\sim u})$$
$$c_2 = ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u}) \leftarrow app(A^{\sim u}, B^{\sim u}, V), app([D^{\sim u}], V, E^{\sim u})$$

Because le_4 is an incorrect LE, $\mathcal{M}(\mathbb{D} \cup \{c_1\}) \neq \mathcal{M}(\mathbb{D} \cup \{c_2\})$. This example shows that the state of $\mathcal{R}_4$ is obtained from $\{d\}$, where d is generated from c_1 and c_2.

4.2.3. *Instantiation check for the specified LE.* First, a definite clause d is generated based on c_1 and c_2. The head part of d is a *yes* atom, and the body part of d is a *relation* atom. A *relation* atom specifies the relationship between c_1 and c_2. Therefore, d is as follows:

$$d : yes \leftarrow relation([ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u}), app(A^{\sim u}, B^{\sim u}, C), app(C, [D^{\sim u}], E^{\sim u})],$$
$$[ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u}), app(A^{\sim u}, B^{\sim u}, V), app([D^{\sim u}], V, E^{\sim u})])$$

Next, a clause set $\mathcal{R}$ is determined. $\mathcal{R}$ is a singleton $\{d\}$, which is the state s_1 shown in Table 1. Subsequently, a ground substitution δ_1 applied to one variable in $var(\{ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u})\})$ is determined. δ_1 is determined based on the definition of the *app* predicate. In $\mathbb{D}$, the *app* predicate is defined as follows:

$$app([\,], X, X) \leftarrow$$
$$app([X|A], B, [X|C]) \leftarrow app(A, B, C)$$

From this definition, we will specialize the first argument of an *app* atom using δ_1. The first argument of $app\,(A^{\sim u}, B^{\sim u}, C)$ and $app\,(A^{\sim u}, B^{\sim u}, V)$ is variable $A^{\sim u}$. We recommend

TABLE 1. Process flow to obtain the state s_{11} $(= \mathcal{R}_4)$ by clause replacement

State of *relation* atom and application of ET rules
s_1: $\{yes \leftarrow relation([ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u}), app(A^{\sim u}, B^{\sim u}, C), app(C, [D^{\sim u}], E^{\sim u})],$ $[ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u}), app(A^{\sim u}, B^{\sim u}, V), app([D^{\sim u}], V, E^{\sim u})])\}$
$\Downarrow$ Apply δ_1 $(= \{A^{\sim u}/[1]\})$ to variable $A^{\sim u}$
s_2: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), \underline{app([1], B^{\sim u}, C)}, app(C, [D^{\sim u}], E^{\sim u})],$ $[ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), app([1], B^{\sim u}, V), app([D^{\sim u}], V, E^{\sim u})])\}$
$\Downarrow$ Replace using r_1
s_3: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), \underline{app([\,], B^{\sim u}, X)}, app([1 \mid X], [D^{\sim u}], E^{\sim u})],$ $[ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), app([1], B^{\sim u}, V), app([D^{\sim u}], V, E^{\sim u})])\}$
$\Downarrow$ Replace using r_2
s_4: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), app([1 \mid B^{\sim u}], [D^{\sim u}], E^{\sim u})],$ $[ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), \underline{app([1], B^{\sim u}, V)}, app([D^{\sim u}], V, E^{\sim u})])\}$
$\Downarrow$ Replace using r_3
s_5: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), app([1 \mid B^{\sim u}], [D^{\sim u}], E^{\sim u})],$ $[ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), \underline{app([\,], B^{\sim u}, Y)}, app([D^{\sim u}], [1 \mid Y], E^{\sim u})])\}$
$\Downarrow$ Replace using r_4
s_6: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), app([1 \mid B^{\sim u}], [D^{\sim u}], E^{\sim u})],$ $[ans([1], B^{\sim u}, D^{\sim u}, E^{\sim u}), \underline{app([D^{\sim u}], [1 \mid B^{\sim u}], E^{\sim u})}])\}$
$\Downarrow$ Replace using r_3
s_7: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, [D^{\sim u} \mid Z]), app([1 \mid B^{\sim u}], [D^{\sim u}], [D^{\sim u} \mid Z])],$ $[ans([1], B^{\sim u}, D^{\sim u}, [D^{\sim u} \mid Z]), \underline{app([\,], [1 \mid B^{\sim u}], Z)}])\}$
$\Downarrow$ Replace using r_4
s_8: $\{yes \leftarrow relation([ans([1], B^{\sim u}, D^{\sim u}, [D^{\sim u}, 1 \mid B^{\sim u}]), \underline{app([1 \mid B^{\sim u}], [D^{\sim u}], [D^{\sim u}, 1 \mid B^{\sim u}])}],$ $[ans([1], B^{\sim u}, D^{\sim u}, [D^{\sim u}, 1 \mid B^{\sim u}])])\}$
$\Downarrow$ Replace using r_1
s_9: $\{yes \leftarrow relation([ans([1], B^{\sim u}, 1, [1, 1 \mid B^{\sim u}]), app(B^{\sim u}, [1], [1 \mid B^{\sim u}])],$ $[ans([1], B^{\sim u}, 1, [1, 1 \mid B^{\sim u}])])\}$
$\Downarrow$ Apply δ_2 $(= \{B^{\sim u}/[2]\})$ to variable $B^{\sim u}$
s_{10}: $\{yes \leftarrow relation([ans([1], [2], 1, [1, 1, 2]), \underline{app([2], [1], [1, 2])}],$ $[ans([1], [2], 1, [1, 1, 2])])\}$
$\Downarrow$ Replace using r_1
s_{11}: $\{yes \leftarrow relation([\,], [ans([1], [2], 1, [1, 1, 2])])\}$
$*$ The state s_{11} is the same as $\mathcal{R}_4$ shown in Formula (3).

applying a finite list such as [1], $[1, 2]$ to $A^{\sim u}$. This example uses $\{A^{\sim u}/[1]\}$ as δ_1. The state s_2 $(= \mathcal{R}\delta_1)$ shown in Table 1 is obtained by applying δ_1 to $\mathcal{R}$.

Subsequently, $\mathcal{R}\delta_1$ is replaced using four ET rules r_1, r_2, r_3, and r_4. In Table 1, the underlined atoms represent atoms that apply to the ET rule. These rules apply to *relation* atoms. r_1 and r_2 are used to transform the first argument representing the state obtained from c_1, whereas r_3 and r_4 are used to transform the second argument representing the state obtained from c_2. In the instantiation check, c_1 and c_2 are replaced according to unfolding operations [27], while preserving the declarative meaning. ET rules r_1, r_2, r_3, and r_4 perform unfolding operations for c_1 and c_2 written as arguments of a *relation* atom. These rules are generated from particular formulas in the general unfolding operations of

the *app* predicate, and are written as follows:

r_1: $relation([H|X], R), \{S = [app([A|B], C, D)], subset(S, X)\}$
$\Rightarrow \{unify(D, [A|E], Flag), replace(Flag, [H|X], S, [app(B, C, E)], L)\}, relation(L, R)$

r_2: $relation([H|X], R), \{S = [app([\,], A, B)], subset(S, X)\}$
$\Rightarrow \{unify(A, B, Flag), replace(Flag, [H|X], S, [\,], L)\}, relation(L, R)$

r_3: $relation(L, [H|Y]), \{S = [app([A|B], C, D)], subset(S, Y)\}$
$\Rightarrow \{unify(D, [A|E], Flag), replace(Flag, [H|Y], S, [app(B, C, E)], R)\}, relation(L, R)$

r_4: $relation(L, [H|Y]), \{S = [app([\,], A, B)], subset(S, Y)\}$
$\Rightarrow \{unify(A, B, Flag), replace(Flag, [H|Y], S, [\,], R)\}, relation(L, R)$

As a result of clause replacement using r_1, r_2, r_3, and r_4, the state s_9 shown in Table 1 is obtained from s_2. This state is not equal to any of $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, or $\mathcal{R}_4$ shown in Formula (3). Therefore, clause replacement continues after applying a ground substitution δ_2. δ_2 applies to one variable in $var(\{ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u})\}) \cup var(\{ans([1], B^{\sim u}, 1, [1, 1|B^{\sim u}])\})$. In s_9, c_2 is a unit clause, so we determine δ_2 so that c_1 is false. We recommend applying a finite list other than [1] to $B^{\sim u}$. This example uses $\{B^{\sim u}/[2]\}$ as δ_2 because $\mathcal{R}_4$ is obtained. The state s_{10} shown in Table 1 is obtained by applying δ_2 to s_9.

The state s_{11} is obtained from s_{10} using r_1. When s_{11} is written in the form of $\mathcal{R}_4$, an atom H is $ans(A^{\sim u}, B^{\sim u}, D^{\sim u}, E^{\sim u})$, and substitutions δ and σ are $\{A^{\sim u}/[1], B^{\sim u}/[2]\}$ and $\{D^{\sim u}/1, E^{\sim u}/[1, 1, 2]\}$, respectively. Therefore, s_{11} is the same as $\mathcal{R}_4$.

$$s_{11}: \{yes \leftarrow relation([\,], [ans([1], [2], 1, [1, 1, 2])])\}$$

$$\mathcal{R}_4: \{yes \leftarrow relation([\,], [H\delta\sigma])\}$$

Consequently, the relationship between c_1 and c_2 is not equivalent, so *false* is returned and le_4 is removed from the set $\mathbb{S}$ of LEs.

5. Experimental Evaluation of the Proposed Framework.

5.1. **Problem settings and parameters.** In this experiment, we generate a set $\mathbb{S}$ of LEs from the following goal clauses g_1 and g_2. The *neq* predicate is a built-in predicate that specifies that two terms are not equal. The meanings of the *app* and *rev* predicates are the same as those shown in Section 2.4.

$$g_1: ans([D|F], G, E) \leftarrow app(A, B, C), app(C, [D], E), rev(F, B), rev(G, A)$$
$$g_2: ans([C|A], D, E) \leftarrow rev(A, B), app(B, [C], D), rev(D, E), neq([D|A], E)$$

Both g_1 and g_2 are obtained in the actual computation of problem solving using ET rules. From these goal clauses, multiple correct and incorrect LEs belonging to the C2LE class can be generated.

In Step 1 of the proposed framework, the generator uses all $\mathcal{E}_1$ candidates determined from a goal clause and all $\mathcal{E}_2$ candidates generated based on $\mathcal{E}_1$. It also generates $\mathcal{E}_2$ candidates consisting of two or three atoms. In Step 2, when performing an instantiation check, the filter uses the following ET rules in addition to the four ET rules shown in Section 4.2.3. These ET rules are generated from particular formulas in the general unfolding operations of the *rev* predicate.

r_5: $relation([H|X], R), \{S = [rev([A|B], C)], subset(S, X)\}$
$\Rightarrow \{replace(Flag, [H|X], S, [rev(B, D), app(D, [A], C)], L)\}, relation(L, R)$

r_6: $relation([H|X], R), \{S = [rev([\,], A)], subset(S, X)\}$
$\Rightarrow \{unify(A, [\,], Flag), replace(Flag, [H|X], S, [\,], L)\}, relation(L, R)$

r_7: $relation(L, [H|Y]), \{S = [rev([A|B], C)], subset(S, Y)\}$
$\Rightarrow \{replace(Flag, [H|Y], S, [rev(B, D), app(D, [A], C)], R)\}, relation(L, R)$
r_8: $relation(L, [H|Y]), \{S = [rev([\ ], A)], subset(S, Y)\}$
$\Rightarrow \{unify(A, [\ ], Flag), replace(Flag, [H|Y], S, [\ ], R)\}, relation(L, R)$

The filter also determines substitutions δ_1 and δ_2 from the set $\{[\ ], [1], [1, 2]\}$. For example, when determining δ_1 that applies to a variable $X^{\sim u}$, $\{X^{\sim u}/[\ ]\}$, $\{X^{\sim u}/[1]\}$, and $\{X^{\sim u}/[1, 2]\}$ are determined as δ_1 candidates. Given multiple δ_1 candidates, the filter uses all δ_1 candidates to generate a clause set $\mathcal{R}\delta_1$ for each δ_1 candidate.

5.2. **Experimental result.** In Table 2, the values in the $\mathbb{S}$ size column are the numbers of LEs appearing in $\mathbb{S}$. The values in parentheses are the numbers of incorrect LEs. The difference between the values in the $\mathbb{S}$ size column represents the number of incorrect LEs removed using the filter. The values in the Time column are the average times when executing a function five times. Plus/minus values are standard variations. The $\mathcal{E}_1$ candidates shown in the $\mathcal{E}_1$ column are as follows:

e_1: $\{app(A, B, C), rev(G, A)\}$
e_2: $\{app(A, B, C), rev(F, B)\}$
e_3: $\{app(A, B, C), rev(F, B), rev(G, A)\}$
e_4: $\{app(A, B, C), app(C, [D], E)\}$
e_5: $\{app(A, B, C), app(C, [D], E), rev(G, A)\}$
e_6: $\{app(A, B, C), app(C, [D], E), rev(F, B)\}$
e_7: $\{app(B, [C], D), rev(D, E)\}$
e_8: $\{rev(A, B), app(B, [C], D)\}$
e_9: $\{rev(A, B), app(B, [C], D), rev(D, E)\}$

Given goal clause g_1, the generator first generates a set $\mathbb{S}$ composed of 10,692 LEs, and then the filter removes 10,594 incorrect LEs from $\mathbb{S}$. As a result, a set $\mathbb{S}$ of ninety eight checked LEs was obtained. Moreover, the filter completely removed all incorrect LEs appearing in $\mathbb{S}$. The following LE appears in a set $\mathbb{S}$ obtained from g_1 using the generator.

$$\forall_{\{A,B,D,E\}}(\exists_{\{C\}}\{app(A, B, C), app(C, [D], E)\} \leftrightarrow \exists_{\{V\}}\{app(B, [D], V), app(A, V, E)\})$$

By selecting e_4 as $\mathcal{E}_1$, this LE will be obtained at the seventy sixth when generating $\mathcal{E}_2$ consisting of two atoms. This LE is used to generate the following ET rule needed to prove the correctness of the specified LE.

$$app(A, B, C), app(C, [D], E), \{isolated(C)\} \Rightarrow app(B, [D], V), app(A, V, E)$$

As shown in Table 2, if e_4, e_5, and e_6 were selected as $\mathcal{E}_1$, no LE was generated depending on the number of atoms in $\mathcal{E}_2$. For e_4, the number of argument candidates was not enough for the required number. For e_5 and e_6, all variables other than link variables were not universally quantified variables.

Given goal clause g_2, the generator first generates a set $\mathbb{S}$ composed of 1,836 LEs, and then the filter removes 1,698 incorrect LEs from $\mathbb{S}$. As a result, a set $\mathbb{S}$ of one hundred and thirty eight checked LEs was obtained. Moreover, as with LE generation from g_1, the filter completely removed all incorrect LEs appearing in $\mathbb{S}$. The following LE appears in a set $\mathbb{S}$ obtained from g_2 using the generator.

$$\forall_{\{B,C,E\}}\left(\exists_{\{D\}}\{app(B, [C], D), rev(D, E)\} \leftrightarrow \exists_{\{V\}}\{app([C], V, E), rev(B, V)\}\right)$$

TABLE 2. Execution result of LE generation framework when giving g_1 and g_2

Input	$\mathcal{E}_1$	$\mathcal{E}_2$	Generator $\mathbb{S}$ size	Generator Time (msec)	Filter $\mathbb{S}$ size	Filter Time (msec)
g_1	e_1	3 atoms	576	94,755±434	16(0)	72,953±1,221
		2 atoms	36	82±1	2(0)	2,105±18
	e_2	3 atoms	576	96,676±426	16(0)	23,341±205
		2 atoms	36	82±1	2(0)	871±10
	e_3	3 atoms	576	97,120±184	16(0)	103,688±2,686
		2 atoms	36	92±2	2(0)	2,303±15
	e_4	3 atoms	0	1,542,614±19,714	No execution	
		2 atoms	216	860±6	4(0)	10,929±250
	e_5	3 atoms	4320	225,832±1,344	16(0)	851,373±4,447
		2 atoms	0	167±2	No execution	
	e_6	3 atoms	4320	225,282±1,537	24(0)	366,218±1,150
		2 atoms	0	166±2	No execution	
	Total $\mathbb{S}$ size		10,692		98(0)	
g_2	e_7	3 atoms	576	102,657±2,235	32(0)	19,241±351
		2 atoms	36	85±1	4(0)	862±9
	e_8	3 atoms	576	100,530±657	32(0)	15,863±211
		2 atoms	36	83±1	4(0)	740±12
	e_9	3 atoms	576	100,154±1,295	64(0)	31,943±629
		2 atoms	36	93±1	2(0)	757±9
	Total $\mathbb{S}$ size		1,836		138(0)	

By selecting e_7 as $\mathcal{E}_1$, this LE will be obtained at the twenty eighth when generating $\mathcal{E}_2$ consisting of two atoms. The following LE will be obtained at the three hundred and eighty seventh when generating $\mathcal{E}_2$ consisting of three atoms.

$$\begin{aligned}&\forall_{\{B,C,E\}} \big(\exists_{\{D\}}\{app(B,[C],D), rev(D,E)\}\\ &\leftrightarrow \exists_{\{V,W\}}\{app(W,V,E), rev(B,V), rev([C],W)\}\big)\end{aligned} \tag{4}$$

As with the above ET rule, the ET rules generated from these LEs are also needed to prove the correctness of the LEs.

In this experiment, the execution time of each of the generator and filter was within working time. However, when e_4 was selected as $\mathcal{E}_1$, the generator took about twenty five minutes to generate a set B_2 of $\mathcal{E}_2$ candidates consisting of three atoms. This is because there are numerous combinations that assign argument candidates to atoms.

5.3. **Effectiveness of using the proposed framework.** Previously, given a goal clause, ET rules that can be generated from LEs belonging to the C2LE class were manually written based on programming experience. The generator enables us to automatically generate many LEs that belong to the C2LE class. Given an LE by the generator, an ET rule can be generated by simple formula transformation. Therefore, the proposed framework increases the number of ET rules that can be automatically generated from a goal clause. For example, ET rules r_a and r_b are needed to prove the correctness of the LE in Formula (4) in addition to general ET rules. The previous framework [20] generates LEs belonging to the Speq class, so r_a can be generated, but r_b cannot. By using the

proposed framework, r_b can be generated.

$$r_a\colon app(A, [\,], B) \Rightarrow eq(A, B), app(A, [\,], B)$$
$$r_b\colon app(A, B, C), app(C, [D], E), \{isolated(C)\} \Rightarrow app(B, [D], V), app(A, V, E)$$

As shown in Section 3.1, the correctness of LEs is proven in step (ii) of the LE-based method. Reducing the number of incorrect LEs from a set $\mathbb{S}$ shortens the execution time of step (ii). The filter removes multiple incorrect LEs appearing in $\mathbb{S}$. In the experiment, all incorrect LEs were completely removed from $\mathbb{S}$. Therefore, the proposed framework can improve the time efficiency of ET rule generation by the LE-based method. For example, given goal clause g_1, the filter removed 10,594 incorrect LEs. In the current framework for proof, many steps are performed manually, so it takes about ten minutes to prove an LE. Thus, it would take more than seventy days to prove 10,594 incorrect LEs. In contrast, the filter removes 10,594 incorrect LEs in about twenty five minutes.

6. **Conclusions.** This paper proposed an LE generation framework consisting of two functions, a generator and a filter. The generator is used to generate a set $\mathbb{S}$ of multiple LEs belonging to the C2LE class, regardless of the correctness of LEs with respect to $\mathbb{D}$. The filter is used to remove multiple incorrect LEs from $\mathbb{S}$. Moreover, we proposed an instantiation check to efficiently find incorrect LEs. The instantiation check determines if two definite clauses are equivalent by specializing the targeted clauses.

In experimental evaluation, we used goal clauses that appear in the actual computation of problem solving. From the result of the experiment, we discussed the effectiveness of using the proposed framework for ET rule generation.

As future work, we will propose another class of LEs to help generate ET rules in order to increase the number of ET rules generated by the LE-based method. We will also develop application software that automatically proves the correctness of the LE.

REFERENCES

[1] M. Järvisalo, D. L. Berre, O. Roussel and L. Simon, The international SAT solver competitions, *AI Magazine*, vol.33, no.1, pp.89-92, 2012.

[2] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos and K. Leyton-Brown, The configurable SAT solver challenge (CSSC), *Artificial Intelligence*, vol.243, pp.1-25, 2017.

[3] C. Barrett and C. Tinelli, Satisfiability modulo theories, *Handbook of Model Checking*, pp.305-343, 2018.

[4] C. Chen, S. Yan, G. Zhao, B. S. Lee and S. Singhal, A systematic framework enabling automatic conflict detection and explanation in cloud service selection for enterprises, *The 5th International Conference on Cloud Computing*, pp.883-890, 2012.

[5] S. Al-Haj, E. Al-Shaer and H. V. Ramasamy, Security-aware resource allocation in clouds, *The 10th International Conference on Services Computing*, pp.400-407, 2013.

[6] F. Micota, M. Erascu and D. Zaharie, Constraint satisfaction approaches in cloud resource selection for component based applications, *2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing*, 2018.

[7] S. Bayless, N. Kodirov, S. M. Iqbal, I. Beschastnikh, H. H. Hoos and A. J. Hu, Scalable constraint-based virtual data center allocation, *Artificial Intelligence*, vol.278, 2020.

[8] K. Akama, H. Koike and E. Miyamoto, A theoretical foundation for generation of equivalent transformation rules (program transformation, symbolic computation and algebraic manipulation), *Research Institute for Mathematical Sciences Kyoto University Koukyuroku*, no.1125, pp.44-58, 2000.

[9] K. Akama, E. Nantajeewarawat and H. Koike, Function-variable elimination and its limitations, *The 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, vol.2, pp.212-222, 2015.

[10] K. Akama, E. Nantajeewarawat and H. Koike, Model-intersection problems with existentially quantified function variables, *The 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, vol.2, pp.52-63, 2016.

[11] K. Akama and E. Nantajeewarawat, Skolemization that preserves logical meanings, *International Journal of Innovative Computing, Information and Control*, vol.17, no.1, pp.1-13, 2021.

[12] K. Miura, C. Powell and M. Munetomo, Optimal and feasible cloud resource configurations generation method for genomic analytics applications, *The 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp.137-144, 2018.

[13] C. Powell, K. Miura and M. Munetomo, Constrained multi-objective optimization method for practical scientific workflow resource selection, *The 10th International Conference on Evolutionary Multi-Criterion Optimization*, pp.683-694, 2019.

[14] A. Konak, D. W. Coit and A. E. Smith, Multi-objective optimization using genetic algorithms: A tutorial, *Reliability Engineering & System Safety*, vol.91, no.9, pp.992-1007, 2006.

[15] K. Deb, Multi-objective evolutionary algorithms, in *Springer Handbook of Computational Intelligence*, J. Kacprzyk and W. Pedrycz (eds.), Berlin, Springer, 2015.

[16] T. Frühwirth, A devil's advocate against termination of direct recursion, *The 17th International Symposium on Principles and Practice of Declarative Programming*, 2015.

[17] T. Frühwirth, Constraint handling rules – What else?, in *Rule Technologies: Foundations, Tools, and Applications. RuleML 2015. Lecture Notes in Computer Science*, N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke and D. Roman (eds.), Cham, Springer, 2017.

[18] K. Miura, K. Akama, H. Mabuchi and H. Koike, Theoretical basis for making equivalent transformation rules from logical equivalences for program synthesis, *International Journal of Innovative Computing, Information and Control*, vol.9, no.6, pp.2635-2650, 2013.

[19] K. Miura and K. Akama, ET-based bidirectional search for proving formulas in the class ES, *International Journal of Innovative Computing, Information and Control*, vol.10, no.6, pp.1999-2009, 2014.

[20] K. Miura, K. Akama and H. Mabuchi, Generating functionality-based rules for program construction, *International Journal of Innovative Computing, Information and Control*, vol.5, no.9, pp.2463-2479, 2009.

[21] K. Miura, K. Akama, H. Koike and H. Mabuchi, Proof of unsatisfiability of atom sets based on computation by equivalent transformation rules, *International Journal of Innovative Computing, Information and Control*, vol.9, no.11, pp.4419-4430, 2013.

[22] A. Sakurai and H. Motoda, Proving definite clauses without explicit use of inductions, in *Logic Programming '88. LP 1988. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence)*, K. Furukawa, H. Tanaka and T. Fujisaki (eds.), , Berlin, Heidelberg, Springer, 1989.

[23] M. A. Nabulsi and N. A. Hamad, Proof of implications involving quantifiers distribution over logical operators, *Journal of Theoretical and Applied Information Technology*, vol.95, no.12, pp.2824-2829, 2017.

[24] F. Quiam, M. Nabulsi and S. Alqatawneh, Verifying the validity of implications that involve quantifiers using the simplification and logical inference methods, *ICIC Express Letters, Part B: Applications*, vol.10, no.7, pp.571-578, 2019.

[25] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1993.

[26] L. Sterling and E. Shapiro, *The Art of Prolog*, 2nd Edition, The MIT Press, 1994.

[27] K. Akama and E. Nantajeewarawat, Unfolding existentially quantified sets of extended clauses, *The 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, vol.2, pp.96-103, 2016.