International Journal of Innovative Computing, Information and Control
Volume **20**, Number **1**, February **2024**

ICIC International ©2024 ISSN 1349-4198
pp. **197–211**

THE SQUEEZE METHOD: A METHOD FOR PROGRAM CONSTRUCTION BY ACCUMULATING EQUIVALENT TRANSFORMATION RULES

Kiyoshi Akama[1] and Ekawit Nantajeewarawat[2,*]

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

[2]School of Information, Computer and Communication Technology
Sirindhorn International Institute of Technology
Thammasat University
99 Moo 18, Km. 41 on Paholyothin Highway, Khlong Luang, Pathum Thani 12120, Thailand
*Corresponding author: ekawit@siit.tu.ac.th

Received April 2023; revised August 2023

Abstract. *In the equivalent transformation (ET) computation model, a specification is a set of problems, a program is a set of prioritized rewriting rules, and computation consists in successive reduction of problems by rule application. As long as answer-preserving rewriting rules, called ET rules, are used, correct computation results are guaranteed. In this paper, a framework for program synthesis, called the squeeze method, in the ET model is discussed. The method receives a problem as an input, and generates a set of prioritized rewriting rules as an output program. It constructs a program by accumulation of rules one by one on demand, with the goal of producing a correct and efficient program. By using the conventional resolution-based computation theory, such an effective method of program construction is not achievable due to the lack of a large variety of ET rules.*
Keywords: Program synthesis, Incremental program construction, Equivalent transformation rule, Rule generation, Prioritized ET rules

1. **Introduction.** Equivalent transformation (ET) is one of the most fundamental principles of computation, and it provides a simple and general basis for verification of computation correctness. Computation by ET was initially implemented in experimental natural language understanding systems during 1990-1992 [1], and the idea was further developed into a new computation model, called the *ET model* [2, 3, 4]. A program in this model is a set of prioritized rewriting rules for answer-preserving transformation of problems, and a problem solving process consists in successive rule application. In contrast with declarative computation paradigms such as logic programming [5] and functional programming [6], programs are clearly separated from a declarative description in a specification in the ET model. A specification specifies a set of problems of interest. From a specification, answer-preserving rewriting rules, called *ET rules*, are generated.

In the ET model, program synthesis is defined as generation of a set of prioritized ET rules from a specification. A demand-driven method for program synthesis, called the *squeeze method*, is employed. Based on some heuristics, which are given as its parameters, the method suggests patterns of atoms to be transformed in a problem solving process,

DOI: 10.24507/ijicic.20.01.197

and uses meta-computation techniques [7] for automatic generation of ET rules for the suggested atom patterns.

Program synthesis using the squeeze method is superior to the one in logic programming. Given an input logical problem such as a proof problem and a QA problem, the squeeze method tries to generate a program to solve the input problem. The conventional concept of computation in logical problem solving is based on procedural reading of logical formulas. A set of clauses is identified as a program and is called a logic program. The concept of a logic program is different from that of a program in the ET approach. Typically, SLD resolution is applied to a set of clauses to solve proof problems and QA problems. However, SLD resolution cannot solve all logical problems. One typical example is the pal-pal QA problem (Section 2), which is represented by a set of definite clauses. We revealed that the pal-pal problem can never be solved in finite time by using SLD resolution [8, 9]. Simple resolution-based computation without the possibility of using many kinds of ET rules is the major reason for the computational limitations of conventional logical reasoning and answer finding. Program transformation has been used for improving the practical solvability of conventional logic programs [10, 11, 12]. However, program transformation cannot generate a program for solving the pal-pal QA problem. The squeeze method in the ET paradigm overcomes the difficulty of conventional logic programming. All logical problems that can be solved in conventional logic programming can also be solved by the squeeze method straightforwardly. The squeeze method can solve strictly a larger class of logical problems compared to that of logical problems that can be solved by the techniques in conventional logic programming.

The paper progresses from here as follows. Section 2 recalls model-intersection (MI) problems based on clauses and if-and-only-if formulas (*iff*-formulas), and provides an example, called the pal-pal problem, which is used as the running example in subsequent sections. Section 3 introduces the squeeze method, which generates a program from a given problem formulated on clauses and *iff*-formulas. Section 4 illustrates the construction of a program for solving the pal-pal problem. Section 5 explains a method of rule generation by using meta-computation and induction, where a non-splitting ET rule is obtained for the pal-pal problem. Section 6 devises a more efficient solution for the pal-pal problem by generation of non-splitting rules by meta-computation. Section 7 compares the ET framework with the conventional resolution-based computation framework from the viewpoint of the squeeze method. Section 8 concludes the paper.

2. Formalization as MI Problems with *iff*-Formulas.

We review MI problems based on descriptions consisting of clauses and *iff*-formulas, and introduce a running example, which will be used in subsequent sections.

2.1. The pal-pal QA problem.

Assume as background knowledge the set consisting of the three *iff*-formulas below, where pal, rv, eq, and ap stand for "palindrome", "reverse", "equal", and "append", respectively.

(1) $\forall x : pal(x) \leftrightarrow rv(x, x)$.

(2) $\forall x, \forall y : rv(x, y) \leftrightarrow$
$((eq(x, [\,]) \wedge eq(y, [\,])) \vee \exists a, \exists w, \exists u : (eq(x, [a|w]) \wedge rv(w, u) \wedge ap(u, [a], y)))$.

(3) $\forall x, \forall y, \forall z : ap(x, y, z) \leftrightarrow$
$((eq(x, [\,]) \wedge eq(y, z)) \vee \exists a, \exists w, \exists u : (eq(x, [a|w]) \wedge eq(z, [a|u]) \wedge ap(w, y, u)))$.

These formulas are referred to as $iff(pal(x))$, $iff(rv(x, y))$, and $iff(ap(x, y, z))$, respectively. Let E_0 be the set consisting of these three formulas.

Consider the problem "find every ground list t of ground terms such that the two lists $[1|t]$ and $[2|t]$ are palindromes", which is called the pal-pal query-answering (QA)

problem and is referred to as prb_1 in later sections. Let E_1 be the first-order formula $\forall x : (pal([1|x]) \wedge pal([2|x]) \rightarrow ans(x))$.

2.2. **Existence-finding formalization.** Given a logical formula E, let $Models(E)$ denote the set of all models of E. Referring to E_0 and E_1 in Section 2.1, we formulate the pal-pal QA problem by $\bigcap Models(E_0 \wedge E_1)$. Then the answer to this pal-pal QA problem is formalized as

$$answer_{pal\text{-}pal} = \varphi_1 \left(\bigcap Models(E_0 \wedge E_1) \right),$$

where, letting $\mathcal{G}_{\rm u}$ be the set of all ground user-defined atoms and $\mathcal{G}_{\rm t}$ the set of all ground terms, φ_1 is defined as a mapping from the power set of $\mathcal{G}_{\rm u}$ to $\mathcal{G}_{\rm t}$ such that for each $G \subseteq \mathcal{G}_{\rm u}$, $\varphi_1(G) = \{t | ans(t) \in G\}$.

2.3. **MI problems in triple forms on *Pair*.** We use clauses and *iff*-formulas for representation of problems. Formally, $Pair$ is defined as the set of all pairs $\langle Cs, E \rangle$ such that Cs is a set of clauses and E a set of *iff*-formulas. Let CLS be the set of all clauses and IFF the set of all *iff*-formulas. We define a model-intersection (MI) problem on the pair space $Pair$ as a triple $\langle Cs, E, \varphi \rangle$, where (i) $Cs \subseteq$ CLS, (ii) $E \subseteq$ IFF, and (iii) φ is a partial mapping from the power set of $\mathcal{G}_{\rm u}$ to some set W, which is called an *extraction mapping*. The answer to the MI problem $\langle Cs, E, \varphi \rangle$, denoted by $ans_{\rm MI}(Cs, E, \varphi)$, is defined by

$$ans_{\rm MI}(Cs, E, \varphi) = \varphi \left(\bigcap Models(\textsc{Fol}(Cs) \wedge E) \right),$$

where $\textsc{Fol}(Cs)$ is the first-order formula that is equivalent to Cs.

2.4. **Separation of the query part in the clause part.** The notion of an independent extraction mapping is next recalled [13]. Given a set P of predicates, let $GAtoms(P)$ denote the set of all ground atoms in $\mathcal{G}_{\rm u}$ the predicates of which belong to P. An extraction mapping φ is said to be *independent* of a set P of predicates iff for any $G_1, G_2 \subseteq \mathcal{G}_{\rm u}$, if $(G_1 - G_2) \cup (G_2 - G_1) \subseteq GAtoms(P)$, then $\varphi(G_1) = \varphi(G_2)$.

Given an MI problem $\langle Cs, E, \varphi \rangle$, we assume that we have a query part Q of the clause set Cs, i.e., $Cs = Q \cup Cs'$, where Cs' is some clause set, such that φ is independent of the set of all predicates occurring in Cs' and E. For example, the pal-pal QA problem in Section 2.1 is formalized as $\langle Cs_1, E_0, \varphi_1 \rangle$, where

1) $Cs_1 = Q_1 \cup Cs'_1$,
2) Q_1 is the singleton clause set $\{(ans(x) \leftarrow pal([1|x]), pal([2|x]))\}$,
3) $Cs'_1 = \emptyset$,
4) E_0 is the set of *iff*-formulas $\{iff(pal(x)), iff(rv(x, y)), iff(ap(x, y, z))\}$, and
5) φ_1 is a mapping that associates with any $G \subseteq \mathcal{G}_{\rm u}$ the set of all ground terms t such that $ans(t) \in G$.

Obviously, φ_1 is independent of the set of all predicates that occur in Cs'_1 and E_0.

3. **Program Construction Using the Squeeze Method.** We introduce the squeeze method for producing a program from a given problem that is formulated on a pair of a set of clauses and a set of *iff*-formulas.

3.1. **ET rules and a solution.** A *program synthesis problem* is described as follows: Given a specification, construct a program that can solve correctly as many problems as possible in the specification. A specification is regarded here as a set of problems.

Based on the set E_0 of *iff*-formulas (see Section 2.1), the following ET rules may be used to solve the pal-pal proof problem [8]:

r_{pal}: $pal(*x) \Rightarrow rv(*x, *x)$.

r_{rv_0}: $rv(*x, *y)$
$\Rightarrow \{=(*x, [\,]), =(*y, [\,])\}$;
$\Rightarrow \{=(*x, [*a|*v])\}, rv(*v, *w), ap(*w, [*a], *y)$.

r_{ap_0}: $ap(*x, *y, *z)$
$\Rightarrow \{=(*x, [\,]), =(*y, *z)\}$;
$\Rightarrow \{=(*x, [*a|*v]), =(*z, [*a|*w])\}, ap(*v, *y, *w)$.

However, these ET rules cannot solve the pal-pal QA problem successfully [9]. When these ET rules are applied to the pal-pal QA problem, we have infinite computation that never reaches the correct answer. The program $\{r_{pal}, r_{rv_0}, r_{ap_0}\}$ cannot solve the pal-pal QA problem. Generation of useful ET rules is the core of this problem solving. We will propose a method for generating useful ET rules for solving a given problem.

3.2. **The squeeze method.** In the ET model, a specification is a set of problems, a program is a set of prioritized ET rules, and computation consists in successive reduction of problems by rule application. Program synthesis can be viewed as a search for a sufficiently efficient program that is correct with respect to a given specification. We propose a method of program synthesis, called the squeeze method, in the ET model. The method receives a problem as an input, accumulates ET rules and specifies their priorities, applies the obtained ET rules for transforming the input problem under the rule priorities, and finally produces a set of prioritized ET rules as an output program.

Consider MI problems on the pair space *Pair* that were formulated in Sections 2.3 and 2.4, where an MI problem is represented as a triple $\langle Q \cup Cs, E, \varphi\rangle$. To ensure correct computation results, only ET rules will be generated. In this paper, we will generate ET rules that do not change $E \subseteq$ IFF and φ. Such ET rules should satisfy the following condition: For any clause sets Q_1, Q_2, Cs_1 and Cs_2, if a rule transforms Q_1 and Cs_1 into Q_2 and Cs_2, then

$$\varphi\left(\bigcap Models\left(\textsc{Fol}(Q_1) \wedge \textsc{Fol}(Cs_1) \wedge E\right)\right) = \varphi\left(\bigcap Models\left(\textsc{Fol}(Q_2) \wedge \textsc{Fol}(Cs_2) \wedge E\right)\right).$$

Meta-computation is an algorithm for generating this class of ET rules [7, 14]. Rule generation based on meta-computation is used by the squeeze method. Given a set E of *iff*-formulas and an atom pattern as input, the meta-computation algorithm automatically generates an ET rule for transforming a meta-description that is determined by the input atom pattern. In order to obtain an efficient program, rules with fewer bodies are preferable as they typically lead to shorter transformation sequences. Rules are prioritized accordingly. By the "demand-driven" characteristic of the method, redundancy in a resulting program tends to be minimized.

The squeeze method constructs a program by accumulation of rules one by one on demand, with the goal of producing a correct and efficient program. It is outlined as follows:

repeat
1. run the current program under control specified by rule priority
2. **if** some obtained final clause is not a unit clause
   begin
   - 2.1 select one or more atoms in the body of a non-unit final clause
   - 2.2 determine a general pattern of the selected atoms
   - 2.3 generate a rewriting rule for transforming atoms that conform to the obtained pattern

2.4 assign priority to the obtained rule
2.5 add the obtained rule to the current program
end
until the obtained final clauses are all unit clauses

Heuristics are used for suggesting a suitable rule to be generated and added in each iteration. They are given through the following three parameters:

[RUN] Control of execution at Step 1.
[TAR] Guidelines on selection of target atoms at Step 2.1.
[PAT] Guidelines on determination of a general atom pattern at Step 2.2.

4. **Program Construction for Solving the Problem prb_1.** For solving the pal-pal QA problem prb_1 (see Section 2.1), construction of a program, named P_A, which consists of the rewriting rules in Figure 1, will now be illustrated. The rewriting rules in Figure 1 are guaranteed to be correct ET rules, which also guarantee the correctness of the solution shown in Table 1. Construction of this solution for prb_1 using the squeeze method is summarized as

$$\text{Problem } prb_1 \Rightarrow \text{Meta-meta-rules} \Rightarrow \text{Rewriting rules in Figure 1}$$
$$\Rightarrow \text{Computation in Table 1,}$$

where the meta-meta-rules are shown in Figure 2 and Figure 3.

r_{pal}: $pal(*x) \Rightarrow rv(*x, *x)$.
r_{rv_1}: $rv([*a|*x], *y) \Rightarrow rv(*x, *v), ap(*v, [*a], *y)$.
r_{rv_2}: $rv(*x, *y), rv(*x, *z) \Rightarrow \{=(*y, *z)\}, rv(*x, *y)$.
r_{ap_1}: $ap(*x, *y, [*a|*z])$
$\Rightarrow \{=(*x, [\,]), =(*y, [*a|*z])\}$;
$\Rightarrow \{=(*x, [*a|*v])\}, ap(*v, *y, *z)$.
r_{rv_3}: $rv(*x, [*a|*y]) \Rightarrow \{=(*x, [*u|*v])\}, rv(*v, *w), ap(*w, [*u], [*a|*y])$.
r_{ap_2}: $ap(*x, [*a], [*b, *c|*y]) \Rightarrow \{=(*x, [*b|*v])\}, ap(*v, [*a], [*c|*y])$.
r_{ap_3}: $ap(*x, [*a], [*b]) \Rightarrow \{=(*x, [\,]), =(*a, *b)\}$.
r_{rv_4}: $rv([\,], *x) \Rightarrow \{=(*x, [\,])\}$.

FIGURE 1. P_A: Rewriting rules for solving prb_1

4.1. **Parameters for constructing the program P_A.** The following parameters are used:

[RUN] Usual rule selection based on rule priority is employed under one constraint: employment of low-priority rules should be minimized.
[TAR] One or more atoms are selected each time, using the following guidelines:
- Select an atom that has a specific structure (e.g., $ap([1|x], y, z)$ is preferable to $ap(x, y, z)$ since $[1|x]$ is more specific than x).
- Select atoms that have common variables (e.g., $ap(x, y, z)$ and $ap(x, v, w)$ with x being a common variable).
- A smaller number of selected atoms is preferable for efficient rule application.

[PAT] A more general pattern is preferable as long as it does not lead to a rule with a larger number of bodies.

TABLE 1. Computation of P_A for solving prb_1

#	State	Rule
1	$\Big\{ans(x) \leftarrow \underline{pal([1\|x])}, pal([2\|x])\Big\}$	r_{pal}
2	$\Big\{ans(x) \leftarrow rv([1\|x],[1\|x]), \underline{pal([2\|x])}\Big\}$	r_{pal}
3	$\Big\{ans(x) \leftarrow \underline{rv([1\|x],[1\|x])}, rv([2\|x],[2\|x])\Big\}$	r_{rv_1}
4	$\Big\{ans(x) \leftarrow rv(x,m), ap(m,[1],[1\|x]),$ $\underline{rv([2\|x],[2\|x])}\Big\}$	r_{rv_1}
5	$\Big\{ans(x) \leftarrow \underline{rv(x,y1)}, ap(y1,[1],[1\|x]),$ $\underline{rv(x,y2)}, ap(y2,[2],[2\|x])\Big\}$	r_{rv_2}
6	$\Big\{ans(x) \leftarrow rv(x,m), \underline{ap(m,[1],[1\|x])},$ $ap(m,[2],[2\|x])\Big\}$	r_{ap_1}
7	$\Big\{ans([\,]) \leftarrow rv([\,],[\,]), ap([\,],[2],[2]),$ $ans(x) \leftarrow \underline{rv(x,[1\|y])}, ap(y,[1],x),$ $ap([1\|y],[2],[2\|x])\Big\}$	r_{rv_3}
8	$\Big\{ans([\,]) \leftarrow rv([\,],[\,]), ap([\,],[2],[2]),$ $ans([v\|x]) \leftarrow rv(x,y),$ $ap(y,[v],[1\|z]),$ $ap(z,[1],[v\|x]),$ $\underline{ap([1\|z],[2],[2,v\|x])}\Big\}$	r_{ap_2}
9	$\Big\{ans([\,]) \leftarrow rv([\,],[\,]), \underline{ap([\,],[2],[2])}\Big\}$	r_{ap_3}
10	$\Big\{ans([\,]) \leftarrow \underline{rv([\,],[\,])}\Big\}$	r_{rv_4}
11	$\{ans([\,]) \leftarrow\}$	–

$pal(*x) \Rightarrow rv(*x, *x).$
$rv([\,], *x) \Rightarrow eq(*x, [\,]).$
$rv([*a|*x], *y) \Rightarrow rv(*x, *m), ap(*m, [*a], *y).$
$ap([\,], *y, *z) \Rightarrow \{=(*y * z)\}.$
$ap([*a|*x], *y, *z) \Rightarrow \{=(*z, [*a|*z1])\}, ap(*x, *y, *z1).$
$ap\left(*p^{k1}, *q, *r\right) \Rightarrow eq(*p, [*a|*x]), eq(*r, [*a|*z]), ap(*x, *q, *z).$
$rv\left(*p^{k1}, *q\right) \Rightarrow eq(*p, [*a|*x]), rv(*x, *r), ap(*r, [*a], *q).$

FIGURE 2. Meta-meta-rules for user-defined predicates

$eq(*x^1, *y^2) \Rightarrow \{(false)\}$.
$eq(*x^2, *y^1) \Rightarrow \{(false)\}$.
$eq\,(*x^{k1}, [\,]) \Rightarrow \{(false)\}$.
$eq\,([\,], *x^{k1}) \Rightarrow \{(false)\}$.
$eq([*a|*b], [\,]) \Rightarrow \{(false)\}$.
$eq([\,], [*a|*b]) \Rightarrow \{(false)\}$.
$eq(*x, [\,]), eq(*x, [*a|*y]) \Rightarrow \{(false)\}$.
$eq(*x^a, [\,]) \Rightarrow \{rmInfo(*x), =(*x, [\,])\}$.
$eq([\,], *x^a) \Rightarrow \{rmInfo(*x), =(*x, [\,])\}$.
$eq\,(*x^a, [*a|*b]) \Rightarrow \{rmInfo(*x), =(*x, [*a|*b])\}$.
$eq\,([*a|*b], *x^a) \Rightarrow \{rmInfo(*x), =(*x, [*a|*b])\}$.
$eq\,(*x^a, *y^a) \Rightarrow \{rmInfo(*x), rmInfo(*y), =(*x, *y)\}$.
$eq\,(*x^k, *y^k) \Rightarrow \{rmInfo(*x), rmInfo(*y), =(*x, *y)\}$.
$eq\,(*x^{k1}, [*a|*b])\,, \{pvar(*a), pvar(*b)\}$
$\Rightarrow \{putInfo(*a, a), putInfo(*b, k), rmInfo(*x), =(*x, [*a|*b])\}$.
$eq([*a|*x], [*b|*y]) \Rightarrow eq(*a, *b), eq(*x, *y)$.
$eq(*v, *a), \{pvar(*v)\} \Rightarrow \{=(*v, *a)\}$.
$eq(*a, *v), \{pvar(*v)\} \Rightarrow \{=(*v, *a)\}$.
$eq(*a, *a) \Rightarrow$.

FIGURE 3. Meta-meta-rules for the eq predicate

TABLE 2. Constructing P_A (with reference to the states in Table 1)

Iteration	Last state	Atom pattern	Rule obtained	Priority assigned
1st	1	$pal(*x)$	r_{pal}	PR-1
2nd	3	$rv([*a|*x], *y)$	r_{rv_1}	PR-1
3rd	5	$rv(*x, *y), rv(*x, *z)$	r_{rv_2}	PR-1
4th	6	$ap(*x, *y, [*a|*z])$	r_{ap_1}	PR-2
5th	7	$rv(*x, [*a|*y])$	r_{rv_3}	PR-1
6th	8	$ap(*x, [*a], [*b, *c|*y])$	r_{ap_2}	PR-1
7th	9	$ap(*x, [*a], [*b])$	r_{ap_3}	PR-1
8th	10	$rv([\,], *x)$	r_{rv_4}	PR-1
9th	11	–	–	–

4.2. **Iterations generating the program P_A.** With the above parameters, the squeeze method produces the program P_A consisting of the rules in Figure 1 within nine iterations as shown in Table 1 and Table 2. All rules in Figure 1 can be automatically generated by using meta-computation and the correctness of the generated rules is guaranteed. Generation of these ET rules by meta-computation is explained in Section 5 and Section 6.

The 1st iteration: The initial program contains no rule. Running the program makes no change to the initial problem prb_1. Either $pal([1|x])$ or $pal([2|x])$ may be selected, and the atom pattern $pal(*x)$ is determined. An ET rule for this pattern, named r_{pal}, is generated. Since r_{pal} has a single body, assign a high priority level, say PR-1, to it.

The 2nd iteration: The current program contains only r_{pal}. Following the first two transformation steps in Table 1, running this program yields the third state in the table as the last state. The two body atoms in this state have the same pattern, and one of them is selected as the target atom. Set $rv([*a|*x], *y)$ as the target atom pattern. Generate an ET rule for this pattern, and r_{rv_1} is obtained. Since r_{rv_1} has a single body, assign the priority level PR-1 to it.

The 3rd iteration: Following the first four transformation steps in Table 1, running the current program results in the fifth state in the table. Select the two rv-atoms in this state as the target atoms, and set the pair of $rv(*x, *y)$ and $rv(*x, *z)$ as the target pattern. Devise the multi-head ET rule r_{rv_2} for it. Again assign the priority level PR-1 to this rule.

The 4th iteration: Following the first five transformation steps in Table 1, the current program now yields the sixth state as the last state. Select the first ap-atom in this state as the target atom, and set $ap(*x, *y, [*a|*z])$ as the target atom pattern. Generate an ET rule for this pattern, and the rule r_{ap_1} is obtained. Since this rule has more than one body, assign a lower priority level, say PR-2, to it.

The 5th iteration: By the first six transformation steps in Table 1, the current program transforms prb_1 into the seventh state in the table. By the constraint imposed upon by the parameter [RUN], although this state can be transformed further using r_{ap_1}, this transformation step is not made. Instead, a new rule is constructed. The first rv-atom in the second clause of this state is selected as the target atom, and the atom pattern $rv(*x, [*a|*y])$ is determined. The ET rule r_{rv_3} is then generated, and the priority level PR-1 is assigned to it.

The 6th iteration onwards: By following the squeeze method three more iterations, the ET rules r_{ap_2}, r_{ap_3}, and r_{rv_4} are generated and added to the program in succession. The priority level PR-1 is given to each of them. When running the resulting program with the input problem prb_1, a problem consisting only of unit clauses is obtained, and the construction ends.

5. **Inductive Generation of Multi-head Rules.** We explain a method of generating multi-head ET rules by using meta-computation and induction, which is one of the key techniques for solving logical problems. By induction, results of infinitely many meta-computations are combined into a process of making one rewriting rule.

5.1. **Information-attached variables in meta-clauses.** A meta-clause is a clause that represents a set of clauses by instantiation. For representing a meta-clause, we can use information-attached variables. For instance, $C = \left(h\left(E^{k1}, F^a\right) \leftarrow ap\left(G^k, F^a, D\right)\right)$ is a meta-clause, where E^{k1}, F^a and G^k are information-attached variables. E^{k1} unifies with any list of length $k+1$, F^a any first-order term, and G^k any list of length k. By instantiation, C represents a set of clauses. A meta-clause is transformed by meta-meta-rules. For instance,

$$eq\left(*x^{k1}, [*a|*b]\right), \{pvar(*a), pvar(*b)\} \\ \Rightarrow \{putInfo(*a, a), putInfo(*b, k), rmInfo(*x), =(*x, [*a|*b])\}$$

is a meta-meta-rule that is applicable to an eq-meta-atom $eq(t_1, t_2)$ if there are a pure variable V and a substitution θ such that t_1 is an information-attached variable V^{k1},

$V = *x\theta$, $t_2 = [*a|*b]\theta$, $*a\theta$ is a pure variable, and $*b\theta$ is a pure variable. After the application of this meta-meta-rule, the *eq*-meta-atom is removed with the following side effects: (i) the pure variable V is unified with the list $[*a|*b]\theta$, (ii) a is attached to the variable $*a\theta$, and (iii) k is attached to the variable $*b\theta$. Reduction of a list of length $k+1$ to a list of length k can be realized by this meta-meta-rule.

5.2. **Meta-computation seeking for the *rv* function rule.** Construction of the multi-head rule r_{rv_2} by meta-computation is summarized as

$$\text{Problem } prb_1 \Rightarrow \text{Meta-meta-rules } R_{mm} \Rightarrow \text{Rewriting rule } r_{rv_2}$$

where R_{mm} includes, not only basic meta-meta-rules in Figure 2 and Figure 3, the *ap* function meta-meta-rule that is obtained in Section 5.3 by lift-up of correct rewriting rules.

When we reach the fifth state in Table 1, assume that we take the two *rv*-atoms in the clause body as target atoms and we try to generate a non-splitting ET rule called the *rv* function rule:

$$rv(*x, *y), rv(*x, *z) \Rightarrow \{=(*y, *z)\}, rv(*x, *y).$$

Let E^n be an information-attached variable that matches a list of length n, where $n = 0, 1, 2, \ldots$. Note that $E^0 = [\,]$ and if $n > 0$, then E^n is a non-empty list. Let $rule_n$ be the rewriting rule

$$rv(E^n, F), rv(E^n, H) \Rightarrow rv(E^n, F), eq(F, H).$$

Then the *rv* function rule is an ET rule if $rule_n$ is an ET rule for all $n = 0, 1, 2, \ldots$. We will prove that these rules are ET rules by induction.

5.2.1. *Base case.* Letting $n = 0$, we have the base case:

$$rv(E^0, F), rv(E^0, H) \Rightarrow rv(E^0, F), eq(F, H).$$

The correctness of this rule is obvious since $rv(E^0, X)$ implies $X = [\,]$.

5.2.2. *Inductive case.* Let $rule_k$ be the rewriting rule

$$rv\left(E^k, F\right), rv\left(E^k, H\right) \Rightarrow rv\left(E^k, F\right), eq(F, H).$$

Assume that $rule_k$ is an ET rule, which is an inductive assumption. We start meta-computation from the following initial meta-clause:

$$h(C^a, D^a) \leftarrow rv\left(E^{k1}, C^a\right), rv\left(E^{k1}, D^a\right).$$

Note that the meta-atom $h(C^a, D^a)$ in the left-hand side of this meta-clause can be used for monitoring how the variables C^a and D^a are changed. When C^a and D^a are changed into the same meta-term by meta-computation, we can obtain the meta-atom $eq(C^a, D^a)$. For meta-clause transformation, we use the correct meta-meta-rules in Figure 2 and Figure 3. By the last meta-meta-rule in Figure 2 and the meta-meta-rules in Figure 3, the meta-clause with a list of length $k+1$ above is transformed into the following meta-clause with a list of length k:

$$h(C^a, D^a) \leftarrow \underline{rv\left(E^k, F\right)}, ap(F, [G^a], C^a), \underline{rv\left(E^k, H\right)}, ap(H, [G^a], D^a).$$

By applying the rule $rule_k$ in the inductive assumption, we obtain the meta-clause

$$h(C^a, D^a) \leftarrow rv\left(E^k, F\right), ap(F, [G^a], C^a), eq(F, H), ap(H, [G^a], D^a).$$

By further transformation, we reach the meta-clause

$$h(C^a, D^a) \leftarrow rv\left(E^k, F\right), \underline{ap(F, [G^a], C^a)}, \underline{ap(F, [G^a], D^a)}.$$

Assume that we have the *ap* function meta-meta-rule

$$ap(*x, *y, *z1), ap(*x, *y, *z2) \Rightarrow ap(*x, *y, *z1), eq(*z1, *z2).$$

By applying this *ap* function meta-meta-rule, we obtain the meta-clause

$h(C^a, D^a) \leftarrow rv\left(E^k, F\right), ap(F, [G^a], C^a), eq(C^a, D^a).$

The meta-atom $eq(C^a, D^a)$ in the right-hand side can be removed by a meta-meta-rule with the unification of the variables C and D. Finally, we reach the meta-clause

$h(C^a, C^a) \leftarrow rv\left(E^k, F\right), ap(F, [G^a], C^a).$

Meta-computation now terminates since no existing meta-meta-rule is applicable to this meta-clause. The two meta-atoms $rv\left(E^k, F\right)$, $ap(F, [G^a], C^a)$ in the right-hand side of the meta-clause are not taken to make a new rewriting rule. Instead of these meta-atoms, special attention is given to the first and the second arguments of the h-meta-atom in the left-hand side. Since they are the same, we reconsider the initial meta-clause and then derive a new rewriting rule by imposing an equality constraint using the meta-atom $eq(F, H)$, i.e.,

$rv\left(E^{k1}, F\right), rv\left(E^{k1}, H\right) \Rightarrow rv\left(E^{k1}, F\right), rv\left(E^{k1}, H\right), eq(F, H).$

By factoring the two rv-meta-atoms in the right-hand side of this rule in the presence of $eq(F, H)$, we obtain $rule_{k+1}$, i.e.,

$rv\left(E^{k1}, F\right), rv\left(E^{k1}, H\right) \Rightarrow rv\left(E^{k1}, F\right), eq(F, H),$

which is an ET rule since it is obtained by meta-computation using only correct meta-meta-rules.

5.2.3. *Making an ET rule by induction.* From the results of the base case and the inductive case above, $rule_n$ is an ET rule for $n = 0, 1, 2, \ldots$. Hence, we have the rewriting rule

$rv(*x, *y), rv(*x, *z) \Rightarrow rv(*x, *y), eq(*y, *z).$

By moving the eq-meta-atom to an execution part, we derive the rv function rule r_{rv_2} in Figure 1, i.e.,

$rv(*x, *y), rv(*x, *z) \Rightarrow \{=(*y, *z)\}, rv(*x, *y).$

5.3. **Meta-computation seeking for the *ap* function meta-meta-rule.** In the meta-computation in Section 5.2, we assumed the availability of the ap function meta-meta-rule

$ap(*x, *y, *z1), ap(*x, *y, *z2) \Rightarrow ap(*x, *y, *z1), eq(*z1, *z2).$

Here we try to generate this meta-meta-rule.

5.3.1. *Base case.* The rewriting rule

$ap(X^0, Y^a, A^a), ap(X^0, Y^a, B^a) \Rightarrow ap(X^0, Y^a, A^a), eq(A^a, B^a)$

is an ET rule since $X^0 = [\,]$ and $Y^a = A^a = B^a$.

5.3.2. *Inductive case.* We start meta-computation from the initial meta-clause

$h(A^a, B^a) \leftarrow ap\left(X^{k1}, Y^a, A^a\right), ap\left(X^{k1}, Y^a, B^a\right).$

By the meta-atom $h(A^a, B^a)$ in the head of the above meta-clause, we monitor how C^a and D^a are changed. We apply the meta-meta-rules in Figure 2 and Figure 3, and obtain the meta-clause

$h([K^a|G], [K^a|J]) \leftarrow ap\left(L^k, D^a, G\right), ap\left(L^k, D^a, J\right).$

We use the inductive assumption rule, and obtain the meta-clause

$h([K^a|G], [K^a|J]) \leftarrow ap\left(L^k, D^a, G\right), eq(G, J).$

Finally, we reach the meta-clause

$h([K^a|J], [K^a|J]) \leftarrow ap\left(L^k, D^a, J\right).$

The first and the second arguments of the h-meta-atom in this meta-clause are the same. By reconsidering the initial meta-clause, we derive the eq-meta-atom $eq(A^a, B^a)$ and obtain the rewriting rule

$ap\left(X^{k1}, Y^a, A^a\right), ap\left(X^{k1}, Y^a, B^a\right) \Rightarrow ap\left(X^{k1}, Y^a, A^a\right), ap\left(X^{k1}, Y^a, B^a\right), eq(A^a, B^a).$

By factoring the two *ap*-meta-atoms in the right-hand side of the rule with the existence of the *eq*-meta-atom $eq(A^a, B^a)$, we have the *ap* function rewriting rule

$$ap\left(X^{k1}, Y^a, A^a\right), ap\left(X^{k1}, Y^a, B^a\right) \Rightarrow ap\left(X^{k1}, Y^a, A^a\right), eq\left(A^a, B^a\right).$$

5.3.3. *Deriving the ap function meta-meta-rule.* From the results of the base case and the inductive case above, we have the *ap* function rewriting rule

$$ap(*x, *y, *z1), ap(*x, *y, *z2) \Rightarrow ap(*x, *y, *z1), eq(*z1, *z2).$$

By lifting up this rewriting rule, we derive the *ap* function meta-meta-rule

$$ap(*x, *y, *z1), ap(*x, *y, *z2) \Rightarrow ap(*x, *y, *z1), eq(*z1, *z2).$$

Note that the rewriting rule above and the meta-meta-rule that is obtained by lifting it up are syntactically the same in this particular case.

6. **Non-Splitting Solution for the Problem prb_1.** Seeking for efficient computation, the squeeze method may produce a non-splitting program by generation of only non-splitting rules by meta-computation. We explain a non-splitting solution for the pal-pal problem, which is obtained by making another multi-head rule by the squeeze method.

6.1. **Towards generation of non-splitting rules.** A non-splitting solution for the pal-pal QA problem can also be constructed by the squeeze method. Construction of the non-splitting solution for prb_1 is summarized as

$$\begin{aligned}&\text{Problem } prb_1 \Rightarrow \text{Meta-meta-rules} \Rightarrow \text{Rewriting rules in Figure 4}\\ &\Rightarrow \text{Computation in Table 3,}\end{aligned}$$

where the meta-meta-rules are shown in Figure 2 and Figure 3. We can obtain the program P_B consisting of the rules in Figure 4, which solves prb_1 correctly. All rules in Figure 4 can be automatically generated by using meta-computation and the correctness of the generated rules is strictly guaranteed.

$$\begin{aligned}
r_{pal}\colon&\quad pal(*x) \Rightarrow rv(*x, *x).\\
r_{rv_1}\colon&\quad rv([*a|*x], *y) \Rightarrow rv(*x, *v), ap(*v, [*a], *y).\\
r_{rv_2}\colon&\quad rv(*x, *y), rv(*x, *z) \Rightarrow \{=(*y, *z)\}, rv(*x, *y).\\
r_{ap_4}\colon&\quad ap(*x, [1], [1|*z]), ap(*x, [2], [2|*z]) \Rightarrow \{=(*x, [\,]), =(*z, [\,])\}.\\
r_{rv_4}\colon&\quad rv([\,], *x) \Rightarrow \{=(*x, [\,])\}.
\end{aligned}$$

FIGURE 4. P_B: Non-splitting rewriting rules for solving prb_1

This section focuses on generation of the rule r_{ap_4}, which is a non-splitting rule and is used in P_B in place of the splitting rule r_{ap_1} in the program P_A. Construction of the rewriting rule r_{ap_4} by meta-computation is summarized as

$$\text{Problem } prb_1 \Rightarrow \text{Meta-meta-rules } \Rightarrow \text{Rewriting rule } r_{ap_4},$$

where the meta-meta-rules include basic meta-meta-rules in Figure 2 and Figure 3.

We compare the non-splitting solution with the splitting solution in Table 1. The number of computation steps in Table 3 is smaller than that in Table 1. The number of rules in the obtained program P_B is smaller than that in the program P_A.

When we reach the sixth clause in Table 1, there are three atoms in the right-hand side of the clause. We may select the second atom $ap(y, [1], [1|x])$. Assume that we determine the pattern $ap(*x, *y, [*a|*z])$ for this selected atom. By meta-computation, we can generate the rule r_{ap_1} in Figure 1. This rule splits the sixth clause into two clauses, which are

TABLE 3. Computation of P_B for solving prb_1

#	State	Rule
1	$\Big\{ans(x) \leftarrow \underline{pal([1\|x])}, pal([2\|x])\Big\}$	r_{pal}
2	$\Big\{ans(x) \leftarrow rv([1\|x],[1\|x]), \underline{pal([2\|x])}\Big\}$	r_{pal}
3	$\Big\{ans(x) \leftarrow \underline{rv([1\|x],[1\|x])}, rv([2\|x],[2\|x])\Big\}$	r_{rv_1}
4	$\Big\{ans(x) \leftarrow rv(x,y), ap(y,[1],[1\|x]), \underline{rv([2\|x],[2\|x])}\Big\}$	r_{rv_1}
5	$\Big\{ans(x) \leftarrow \underline{rv(x,y1)}, ap(y1,[1],[1\|x]), \underline{rv(x,y2)}, ap(y2,[2],[2\|x])\Big\}$	r_{rv_2}
6	$\Big\{ans(x) \leftarrow rv(x,y), \underline{ap(y,[1],[1\|x])}, \underline{ap(y,[2],[2\|x])}\Big\}$	r_{ap_4}
7	$\{ans([\,]) \leftarrow rv([\,],[\,])\}$	r_{rv_4}
8	$\{ans([\,]) \leftarrow\}$	–

shown in the seventh state in Table 1. To obtain the non-splitting program P_B, we have to select body atoms that can produce non-splitting rules. Since a non-splitting rule is never obtained by selecting an atom pattern consisting of a single atom at the sixth clause, we proceed to select a pattern consisting of more than one atom. We explain the generation of a non-splitting rewriting rule in the program P_B in the next subsection.

6.2. **Meta-computation seeking for the *ap-ap* rule.** When we reach the sixth clause in Table 3, i.e.,

$ans(x) \leftarrow rv(x,y), ap(y,[1],[1|x]), ap(y,[2],[2|x]),$

assume that we want to generate a non-splitting ET rule that can be applied to this clause. After the failure by the selection of a single body atom, we select the set of two atoms $\{ap(y,[1],[1|x]), ap(y,[2],[2|x])\}$ in the clause body. We start meta-computation from the following meta-clause:

$h\left(B^{k1}, A^a\right) \leftarrow ap\left(B^{k1},[V^1],[V^1|A^a]\right), ap\left(B^{k1},[W^2],[W^2|A^a]\right).$

After 12 applications of the meta-meta-rules in Figure 2 and Figure 3, we reach the meta-clause

$h([\,],[\,]) \leftarrow.$

From the above two meta-clauses, we have the rewriting rule

$ap(*y,[1],[1|*x]), ap(*y,[2],[2|*x]) \Rightarrow eq(*y,[\,]), eq(*x,[\,]).$

By moving the eq-meta-atoms in the right-hand side of the rule to its execution part, we have the rewriting rule (r_{ap_4})

$ap(*y,[1],[1|*x]), ap(*y,[2],[2|*x]) \Rightarrow \{=(*y,[\,]), =(*x,[\,])\}.$

7. **Comparison to the Conventional Approach.** The ET framework is next compared with the conventional resolution-based computation framework from the viewpoint of the squeeze method.

7.1. **Lack of rules in the conventional logic.** The conventional logical computation theory has both representational limitations and computational limitations. It was shown in [8] that the pal-pal proof problem cannot be solved by SLD resolution. It was also shown in [9] that the pal-pal QA problem cannot be solved by SLD resolution. The primary reason for the unsolvability of the pal-pal problems by SLD resolution is the lack of multi-head ET rules. Definite clauses can only support single-head rules in computation based on SLD resolution.

Conventional logic programming has a serious limitation that it cannot solve all logical problems on first-order logic. The squeeze method in the ET paradigm overcomes the difficulty of conventional logic programming. All logical problems that can be solved in logic programming can be solved by the squeeze method straightforwardly. Moreover, the squeeze method can solve many logical problems that cannot be solved by conventional logic programming. Multi-head rules are indispensable for solving some class of logical problems. The squeeze method can generate multi-head rules and can apply them for transforming logical formulas.

7.2. **ET rules.** The conventional concept of computation in logical problem solving is based on procedural reading of logical formulas. Only one inference rule, i.e., the resolution rule, is used in conventional computation by SLD resolution, and control for such computation specifies how to select an occurrence of an atom in a clause at each computation step. Simple resolution-based computation without the possibility of using many kinds of ET rules is the major reason for the computational limitations of conventional logical reasoning and answer finding.

7.3. **Program synthesis and problem solving.** Program synthesis can be viewed as a search for a sufficiently efficient program in a certain space of correct programs with respect to a given specification. In the ET computation model, a program consists of prioritized ET rules, and a program can be constructed by rule generation together with assignment of priority to each rule one by one. We can select an optimal rule application sequence efficiently by taking less-splitting rules. This provides a very powerful method of program generation.

SLD-resolution-based logic regards a set of clauses as a program. Reduction to generation of clauses does not work well since it is difficult to decide how many clauses should be generated. Computation by SLD resolution for solving the pal-pal QA problem is similar to the computation by using the program $\{r_{pal}, r_{rv_0}, r_{ap_0}\}$ in Section 3.1, which leads to infinite computation without reaching the correct answer.

7.4. **Limitations of the concept of logic program.** Program transformation has been used for improving the efficiency of logic programs [10, 11, 12]. However, program transformation in the resolution-based logic can never reach a correct program that can solve the pal-pal QA problem since there is no logic program that can solve this problem completely [8]. Logic programming could not achieve a general and practical method of generating correct and efficient programs. This paper explained that the squeeze method on the ET-based computation theory can generate a correct and efficient program for solving the pal-pal problem. The concept of conventional logic program prevents us from establishing a general and practical method of generating correct and efficient programs.

8. **Concluding Remarks.** Conventional logic adopts computation with a very limited number of inference rules, e.g., the resolution rule. This is a major hindrance to development of correct and efficient programs. In the ET model, a specification is a set of problems, a program is a set of prioritized ET rules, and computation consists in successive

problem simplification by rule application. As long as ET rules are used, correct computation results are guaranteed. ET rules can be generated by meta-computation with guarantee of correctness. Program synthesis can be viewed as a search for a sufficiently efficient program in a certain space of correct programs with respect to a given specification. The squeeze method provides a basis for a framework for program synthesis in the ET model, which supports the employment of the full power of various ET rules. The method combines a solution to a given problem with synthesis of a program. It constructs a program by accumulation of ET rules one by one on demand, with the goal of producing a correct and efficient program. The method receives a problem as an input, accumulates ET rules and specifies their priorities, applies the resulting ET rules for transforming the input problem under the specified rule priorities, and finally produces a set of prioritized ET rules as an output program. With the squeeze method, we can accumulate various efficient ET rules and overcome the limitations of the conventional theory of logical computation. The invention of the squeeze method justifies the superiority of ET-based logic over the conventional inference-based logic.

Acknowledgment. This work was partially supported by (i) JSPS KAKENHI Grant Numbers 25280078 and 26540110, (ii) the Center of Excellence in Intelligent Informatics, Speech and Language Technology and Service Innovation (CILS), Thammasat University, and (iii) the Intelligent Informatics and Service Innovation (IISI) Center, Sirindhorn International Institute of Technology (SIIT), Thammasat University. The authors also gratefully acknowledge the helpful comments and suggestions from the reviewers.

REFERENCES

[1] K. Akama, Y. Nomura and E. Miyamoto, Semantic interpretation of natural language descriptions by program transformation, *Computer Software*, vol.12, no.5, pp.45-62, 1995.

[2] K. Akama, Y. Shigeta and E. Miyamoto, Solving logical problems by equivalent transformation: A theoretical foundation, *Journal of the Japanese Society for Artificial Intelligence*, vol.13, pp.928-935, 1998.

[3] K. Akama and E. Nantajeewarawat, Formalization of the equivalent transformation computation models, *Proc. of the 5th International Conference on Intelligent Technologies*, Houston, TX, USA, pp.190-199, 2004.

[4] K. Akama and E. Nantajeewarawat, Formalization of the equivalent transformation computation models, *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol.10, no.3, pp.245-259, 2006.

[5] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1987.

[6] P. Hudak, Conception, evolution and application of functional programming languages, *ACM Computing Surveys*, vol.21, no.3, pp.359-411, 1989.

[7] K. Akama, E. Nantajeewarawat and H. Koike, Program synthesis based on the equivalent transformation computation model, *Proc. of the 12th International Workshop on Logic Based Program Synthesis and Transformation*, Madrid, Spain, pp.285-304, 2002.

[8] K. Akama and E. Nantajeewarawat, Solving proof problems with equivalent transformation rules, *International Journal of Innovative Computing, Information and Control*, vol.18, no.1, pp.331-344, 2022.

[9] K. Akama and E. Nantajeewarawat, Solving query-answering problems based on equivalent transformation, *International Journal of Innovative Computing, Information and Control*, vol.18, no.5, pp.1547-1558, 2022.

[10] A. Pettorossi and M. Proietti, Transformation of logic programs: Foundations and techniques, *Journal of Logic Programming*, vols.19&20, pp.261-320, 1994.

[11] A. Pettorossi and M. Proietti, Rules and strategies for transforming functional and logic programs, *ACM Computing Surveys*, vol.28, no.2, pp.360-414, 1996.

[12] A. Pettorossi and M. Proietti, Transformation of logic programs, *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol.5, no.94, pp.697-787, 1998.

[13] K. Akama, E. Nantajeewarawat and T. Akama, Side-change transformation, *Proc. of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, vol.2, Seville, Spain, pp.237-246, 2018.
[14] 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*, no.1125, pp.44-58, 2000.

Author Biography

Kiyoshi Akama received the B.Eng. and M.Eng. degrees in Control Engineering from Tokyo Institute Technology, Japan, in 1973 and 1975, respectively; and the D.Eng. degree in Control Engineering from Tokyo Institute Technology, Japan, in 1989. He was an assistant professor at Faculty of Engineering, Tokyo Institute Technology, Japan, 1979-1981; a lecturer at Faculty of Letters, Hokkaido University, Japan, 1981-1989; an associate professor at Faculty of Engineering, Hokkaido University, Japan, 1989-1999; a professor at Center for multimedia Studies, Hokkaido University, Japan, 1999-2003; a professor at Information Initiative Center, Hokkaido University, Japan, 2003-2013; a specially-appointed professor at Information Initiative Center, Hokkaido University, 2013-2015; a professor at Graduate School of Information Science and Technology, Hokkaido University, Japan, 1999-2015.

Dr. Akama is currently an emeritus professor of Hokkaido University, Japan. His research interests include artificial intelligence, computer science, logic and computation, program generation and computation based on the equivalent transformation model, programming paradigms, and knowledge representation.

Ekawit Nantajeewarawat received the B.Eng. degree in Computer Engineering from Chulalongkorn University, Thailand, in 1987; and the M.Eng. and D.Eng. degrees in Computer Science from the Asian Institute of Technology, Thailand, in 1991 and 1997, respectively.

Dr. Nantajeewarawat is currently an associate professor of computer science at Sirindhorn International Institute of Technology, Thammasat University, Thailand. His research interests include knowledge representation, automated reasoning, rule-based equivalent transformation, program synthesis, formal ontologies, and object-oriented modelling.