An MSG-Method for Inductive Logic Program Synthesis

Prepared by: Esra ERDEM

Supervisor: Asst. Prof. Pierre FLENER

Final Report

This report is submitted to the Department of Computer Engineering and Information Science of Bilkent University Faculty of Engineering in partial fulfillment of the requirements of the Senior Project course CS492.

Abstract

SYNAPSE2 is an upgraded version of SYNAPSE (developed by Flener [2]), which is an implementation of an inductive synthesis mechanism that semi-automatically synthesizes divide-and-conquer logic algorithms in a schema-guided way, from non-incrementally given examples and properties of an intended relation. SYNAPSE2 goes through six steps, instantiating the place-holders of a chosen divide-and-conquer logic algorithm schema. At Step 5, the MSG-Method is used as a tool. The MSG-Method is used to infer a logic algorithm of an intended relation, if it is possible to get a logic algorithm using only $=/2$ predicate in the body.

In this project, our aim is to simplify and extend the existing MSG-Method (introduced by Flener [2]) independent of the place-holder and its schema, and to re-implement Step 5 of SYNAPSE2, which corresponds to the steps'5 and 6 of SYNAPSE. In this report, the main concepts are introduced as preliminaries in Section 1, the synthesis mechanism is explained with an example (in SYNAPSE2) in Section 2, the MSG-Method is focused on in Section 3, after then Step 5 is explained in Section 4, finally the project is discussed in Section 5.

Contents

1 Introduction

The main concepts are introduced before starting to discuss the project. First, the main concepts are presented generally; then, getting more precise, step by step, we focus on the specific concepts in order to reach the goal of the project. We firstly introduce the concept of synthesis in Section 1.1; next, we will discuss logic algorithms in Section 1.2, and logic algorithm schemata in Section 1.3; finally, we will explain schema-guided logic algorithm synthesis in Section 1.4.

1.1 Synthesis

We are concerned with synthesis in the framework of programming, so let's define algorithm synthesis and program synthesis as follows:

Definition 1.1 *Algorithm Synthesis* is developing correct *algorithms* (how the problem can be solved) from *specifications* (what the problem is, and how to use a program solving it).

Definition 1.2 *Program Synthesis* is developing correct *programs* (how the problem can be solved) from *intentions* (what the problem is).

In order to differentiate algorithm synthesis from program synthesis more precisely, let's explain the difference between an algorithm and a program, and between specifications and intentions. The difference between an algorithm and a program is that an algorithm is non-executable but a program is. The difference between specifications and intentions is that intentions are informal and they are in our minds (in fact, intentions are what we think the problem is); but, specifications can be both formal and informal (in fact, specifications are what we express the problem as and how to use a program solving that problem). As a result we can say that program synthesis is algorithm synthesis followed by implementation and preceded by specification acquisition. We will focus on algorithm synthesis rather than program synthesis.

The synthesis mechanisms can be classified as follows [1]:

- deductive synthesis: transformational synthesis, proofs-as-programs synthesis, and schema-guided synthesis;
- inductive synthesis: trace-based synthesis and model-based synthesis;
- abductive synthesis;
- analogical synthesis;
- knowledge-based synthesis: empirical synthesis, synthesis by inspection, and synthesis from natural language specifications.
- etc.

We will focus on algorithm synthesis, using mostly deductive synthesis (schema-guided synthesis) and inductive synthesis (trace-based synthesis).

1.1.1 Deductive Synthesis

In deductive synthesis, we use *axiomatic specifications*.

Definition 1.3 An *axiomatic specification* is a set of axioms about the predicate for the intended relation and the non-primitive predicates in these axioms.

Example 1.1 The intended relation $plateau/3$ is given by an informal specification as follows:

$plateau(N, E, P)$ iff P is a plateau of N elements equal to E,
where N is a positive integer, E a term, and P a non-empty list.

A sample axiomatic specification in FOL is as follows:

$\forall\ N, E, P\ (plateau(N, E, P) \Leftrightarrow \forall\ X\ (member(X, P) \Rightarrow X = E) \wedge length(P, N))$
where $member/2$ and $length/2$ are primitive predicates. □

Schema-Guided Synthesis is, given a *schema* (a template algorithm with fixed data flow, and constraints) and a *specification*, instantiating the place-holders of the template to the actual computations such that they satisfy the constraints, which yields an algorithm. (See example in Section 2.2)

1.1.2 Inductive Synthesis

In inductive synthesis, we use *specifications-by-examples*.

Definition 1.4 A *specification-by-examples* consists of *positive* and *negative examples*, where *positive examples* "belong" to the intended relation and *negative examples* do not "belong" to the intended relation.

Example 1.2 For $plateau/3$:

$$\begin{aligned}\mathcal{E}^+(plateau) = \{\ & plateau(1, a, [a]),\\ & plateau(2, a, [a, a]),\\ & plateau(3, b, [b, b, b])\ \}\end{aligned}$$

$$\begin{aligned}\mathcal{E}^-(plateau) = \{\ & plateau(1, a, [a, a]),\\ & plateau(2, a, [b, a])\ \}\end{aligned}$$

where $\mathcal{E}^+(plateau)$ designates positive examples and $\mathcal{E}^-(plateau)$ designates negative examples. □

In **Trace-Based Synthesis**, the examples are given at once; after tracing these examples, the traces are generalized into a recursive program.

1.2 Specifications

Specifications express what the problem is, and how a program to solve that problem is used. Specifications can be formal or informal. We will use informal specifications of the form:

$r(X_1, ..., X_n)$ iff $R[X_1, ..., X_n]$
where $\langle X_1, ..., X_n \rangle \in dom(R)$.
where X_i are distinct variables,
r is a predicate used for the relation to be specified,
$dom(R)$ is the domain of R,
that is the Cartesian product of the domains of $X_1, ..., X_n$,
R is a description of the specified relation.

Example 1.3 An informal specification for $sort/2$ is given as follows:

$sort(L, S)$ iff S is a non-decreasing permutation of L,
where L and R are integer lists. □

An informal specification for each problem used in this paper is given in appendix.

1.3 Logic Algorithms

We will focus on algorithm synthesis, in the framework of logic programming. Let's first explain *logic algorithms* and *logic programs*.

Definition 1.5 A *logic algorithm* of a predicate r/n, denoted $LA(r)$, is expressed as a first-order logic well-formed formula (wff) of the following form:

$\forall\ X_1 \ldots \forall\ X_n\ r(X_1, ..., X_n) \Leftrightarrow B[X_1, ..., X_n]$
where X_i are distinct variables,
$r(X_1, ..., X_n)$ is the *head* of $LA(r)$,
$B[X_1, ..., X_n]$ is the *body* of $LA(r)$:
B is a wff represented in prenex DNF and
$X_1, ..., X_n$ are the only free variables of B.

Example 1.4 A logic algorithm for $plateau/3$ is given as follows:

$$
\begin{array}{l}
\forall\ N, E, P\ plateau(N, E, P) \Leftrightarrow \exists\ TN, HN, TP \\
\qquad\qquad N = s(0) \ \wedge P = [E] \\
\qquad\quad \vee\ N > s(0) \ \wedge N = s(TN) \\
\qquad\qquad\qquad\qquad \wedge\ plateau(TN, E, TP) \\
\qquad\qquad\qquad\qquad \wedge\ P = [E|TP]
\end{array}
$$

where $s(N)$ denotes $N + 1$. □

Definition 1.6 A *logic program* is a set of clauses defining relations between objects.

Previously, we have told that we would focus on algorithm synthesis rather than program synthesis. In fact, we will focus on logic algorithm synthesis.

1.4 Algorithm Schemata

Algorithms can be synthesized by different methodologies such as Divide-and-Conquer (D&C), Generate-and-Test (G&T),... We can abstract a group of algorithms synthesized by a methodology in such a way: we create a template (with place-holders) so that those algorithms are instances of this template satisfying some pre-determined constraints. This abstraction is what we call an algorithm schema. As we will focus on logic algorithm synthesis, we will define logic algorithm schemata in Section 1.3.1; and in Section 1.3.2, we will focus on D&C logic algorithm schemata.

1.4.1 Logic Algorithm Schemata

A group of logic algorithms can be abstracted to give a logic algorithm schema, defined in the following way:

Definition 1.7 A *logic algorithm schema* for a predicate with n arguments is expressed as a second order logic wff form, called *template*:

$\forall\ X_1 \ldots \forall\ X_n\ R(X_1, ..., X_n) \Leftrightarrow B[X_1, ..., X_n]$
where X_i are distinct variables,
$R(X_1, ..., X_n)$ is the *head* of the logic algorithm schema,
R being a predicate variable,
$B[X_1, ..., X_n]$ is the *body* of the logic algorithm schema
in prenex DNF having first-order variables among $X_1, ..., X_n$
and all second-order variables as its only free variables,
and with a set of *constraints*, which are second-order logic wff forms
having all second-order variables of the template as the only free variables.

In the above definition, n is a form variable. When it is replaced by a natural number (and all X_i, where $1 \leq i \leq n$, are shown, that is no "..." is used), the schema will be a second-order logic wff. When we instantiate second-order variables with predicate terms (of the λ-calculus) then we'll have a first-order logic wff, which is a logic algorithm.

1.4.2 D&C Logic Algorithm Schemata

Logic algorithm schemata abstract a group of logic algorithms. Logic algorithms can be grouped and abstracted according to a synthesis methodology. Therefore, we can give logic algorithm schemata for D&C logic algorithms. Let's first give a synthesis of a D&C logic algorithm and then give a logic algorithm schema for it.

Example 1.5 The synthesis of a program for $sort/2$ is given step-by-step as follows:

Step 1: We should select the parameter to be decomposed so that the problem can be divided into subproblems, that is the divide step of a D&C algorithm. This parameter is called as induction parameter and it should be of an inductive(ly defined) type, that is it can be decomposed into sub-structures (i.e. integers, sequences are inductively defined types, but a character is not).

For $sort/2$, we can choose L or S as induction parameter; let's choose L as induction parameter.

Step 2: In a D&C algorithm, if we divide the problem into subproblems, we should solve each subproblem recursively. For recursion, we should find the minimal case (stopping case) where the induction parameter is in a minimal form; that is we should find the minimal and non-minimal forms of the induction parameter.

For $sort/2$, the induction parameter L is a list of integers (from the specification), so its minimal form is the empty list ($L = []$); its non-minimal form is a list of at least one integer ($L = [\_|\_]$).

Step 3: We can divide the problem into subproblems, as we identified the minimal and the non-minimal forms of the induction parameter. Decomposition of the induction parameter can be done in at least the following three ways [1]:

Intrinsic Decomposition: The induction parameter is decomposed such that it has at least one head and at least one tail, reflecting the type of the induction parameter.

For $sort/2$, the intrinsic decomposition of L is $L = [HL|TL]$.

Extrinsic Decomposition: The induction parameter is decomposed such that it has at least one tail and may not have any head, reflecting the type of some other parameter than induction parameter, or reflecting the intended relation.

For $sort/2$, an extrinsic decomposition of L is $partition(L, P, S, B)$, which reflects the intended relation.

Logarithmic Decomposition: The induction parameter is decomposed such that it has at least two tails of about equal size and no head.

For $sort/2$, the logarithmic decomposition of L is $split(L, F, S)$.

Step 4: After dividing the problem into subproblems, we recursively solve each subproblem, by recursive calls (one for each tail), that is the conquer step of D&C algorithm.

For $sort/2$, the recursive call is $sort(TL, TS)$.

Step 5: We can combine all solutions to all subproblems to find the solution to the problem, where solutions relate induction parameter to the others.

For $sort/2$; in the minimal case: if $L = []$ then $S = []$, and in the non-minimal case: if $L = [\_|\_]$ then $L = [HL|TL]$ and $sort(TL, TS)$ and $insert(HL, TS, S)$.

Therefore, a logic algorithm for $sort/2$ with intrinsic decomposition of L is as follows:

$$
\begin{array}{lll}
sort(L,S) \Leftrightarrow & & \\
& L = [] & \wedge\ S = [] \\
& \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] \\
& & \wedge\ sort(TL, TS) \\
& & \wedge\ insert(HL, TS, S)
\end{array}
\qquad \square
$$

We can give a D&C logic algorithm schema, which abstracts all D&C logic algorithms, as follows [1]:

$$
\begin{array}{lll}
R(X,Y,Z) \Leftrightarrow & & \\
 & Minimal(X) & \wedge\ Solve(X,Y,Z) \\
\bigvee_{k=1}^{c} & NonMinimal(X) & \wedge\ Decompose(X,\overrightarrow{HX},\overrightarrow{TX}) \\
 & & \wedge\ Discriminate_k(\overrightarrow{HX},\overrightarrow{TX},Z) \\
 & & \wedge\ (\quad SolveNonMinimal_k(\overrightarrow{HX},\overrightarrow{TX},Y,Z) \\
 & & \qquad \mid \\
 & & \qquad \bigwedge_{i=1}^{t} R(TX_i,TY_i,Z) \\
 & & \qquad \wedge\ Process_k(\overrightarrow{HX},\overrightarrow{HY},Z) \\
 & & \qquad \wedge\ Compose_k(\overrightarrow{HY},\overrightarrow{TY},Y,Z) \qquad)
\end{array}
$$

1.5 Stepwise Schema-Guided Logic Algorithm Synthesis

Given a (logic) algorithm schema and a specification of an intended relation, step-by-step instantiation of the place-holders of the template to the actual computations, such that they satisfy the constraints, is stepwise schema-guided synthesis of (logic) algorithms.

Example 1.6 Given the schema in Section 1.5, we can do the synthesis in five steps as follows:

Step 1: Selection of the induction parameter;

Step 2: Synthesis of *Minimal* and *NonMinimal*;

Step 3: Synthesis of *Decompose*;

Step 4: Introduction of recursive atoms;

Step 5: Synthesis of *Solve*, the $SolveNonMinimal_k$, the $Process_k$, $Compose_k$, and $Discriminate_k$.

□

We will focus on SYNAPSE, a stepwise schema-guided D&C logic algorithm synthesizer, implemented by Flener [2].

2 The Synthesis Mechanism

SYNAPSE is an implementation of an inductive synthesis mechanism that semi-automatically synthesizes D&C logic algorithms in a schema-guided way, from non-incrementally given examples and properties of an intended relation. SYNAPSE goes through seven steps, instantiating the place-holders of a chosen D&C logic algorithm schema.

SYNAPSE was developed by Flener [2]. It is implemented in Prolog. SYNAPSE2 will be an upgraded version of SYNAPSE. The latter will be conceptually simplified, its power will be extended, and it will be re-implemented in Gödel to get SYNAPSE2.

In Section 2.1, we focus on the concept of msg. In Section 2.2, the six steps of SYNAPSE2 are shown on an example. In Section 2.3, the objectives of the project are explained.

2.1 The Concept of MSG

The concept of most-specific generalization (msg) is explained in the following definitions and examples considering terms and atoms. First, we explain what is meant by *specific* and *general* in most-specific generalization.

Definition 2.1 Term x is *more general than* term y, if y can be obtained by substitution θ made on x, i.e., if $y = x\theta$. We also say that x is *less specific than* y, that y is *less general than* x, or that y is *more specific than* x.

Example 2.1 $f(X,4,Y)$ is more general than $f(2,4,6)$ with $\theta = \{X/2, Y/6\}$. □

Definition 2.2 The *msg* of terms $x1$ and $x2$ is a term y, that is more general than $x1$ and $x2$, but less general (upto renaming) than any other term z that is more general than $x1$ and $x2$.

Example 2.2 The msg of $f(1,E,s,[],L,[a,b])$ and $f(1,[a,b],X,[d],M,E)$ is $f(1,Q,W,T,Y,R)$. □

Definition 2.3 The *msg* of a non-empty set $\mathcal{A}$ of terms, denoted $msg(\mathcal{A})$, is $msg(msg(\mathcal{T}),t)$, where $msg(\{t\}) = t$, and $\mathcal{A} = \mathcal{T} \cup \{t\}$.

Example 2.3 The msg of $\{f(1,E,s,[],L,[a,b]), f(1,[a,b],X,[d],M,E)\}$ is $f(1,Q,W,T,Y,R)$. □

Definition 2.4 The *msg* of any two atoms $p(x_1,x_2,...,x_n)$ and $p(y_1,y_2,...,y_n)$ is an atom $p(z_1,z_2,...,z_n)$, where the msg of x_i and y_i is z_i, for $i = 1,2,...,n$.

Example 2.4
The msg of $insert(1,[2],[1,2])$ and $insert(3,[5],[3,5])$ is $insert(X,[Y],[X|Y])$, where $< X,[Y],[X|Y] >= msg(< 1,[2],[1,2] >, < 3,[5],[3,5] >)$. □

Definition 2.5 The *msg* of a non-empty set $\mathcal{A}$ of atoms, denoted $msg(\mathcal{A})$, is $msg(msg(\mathcal{T}),t)$, where $msg(\{t\}) = t$, and $\mathcal{A} = \mathcal{T} \cup \{t\}$.

Example 2.5 The msg of $\{insert(1,[2],[1,2]), insert(3,[5],[3,5])\}$ is $insert(X,[Y],[X|Y])$. □

2.2 The Six Steps of SYNAPSE2

The D&C schema that will guide SYNAPSE2 is [1]:

$$
\begin{array}{lll}
R(X,Y,Z) \Leftrightarrow & & \\
& Minimal(X) & \wedge\ Solve(X,Y,Z) \\
\bigvee_{k=1}^{c} & NonMinimal(X) & \wedge\ Decompose(X,\overrightarrow{HX},\overrightarrow{TX}) \\
& & \wedge\ Discriminate_k(\overrightarrow{HX},\overrightarrow{TX},Z) \\
& & \wedge\ (\quad SolveNonMinimal_k(\overrightarrow{HX},\overrightarrow{TX},Y,Z) \\
& & \qquad |
\end{array}
$$

$$\begin{array}{l} \bigwedge_{i=1}^{t} R(TX_i, TY_i, Z) \\ \wedge\ Process_k(\overrightarrow{HX}, \overrightarrow{HY}, Z) \\ \wedge\ Compose_k(\overrightarrow{HY}, \overrightarrow{TY}, Y, Z) \qquad) \end{array}$$

SYNAPSE2 will have the following six steps:

1. instantiating the left hand side of the equivalence in the schema;
2. choosing the induction parameter, and instantiating $Minimal$ and $NonMinimal$;
3. choosing the decomposition style, and instantiating $Decompose$,
4. instantiating the recursive call(s);
5. instantiating $Process_k$, the $Compose_k$, the $Solve$ and $SolveNonMinimal_k$;
6. instantiating $Discriminate_k$.

After these seven steps, we get a logic algorithm that is true at least for the examples and properties of the intended relation, given at the beginning of the synthesis. We say that the logic algorithm *covers* the examples and properties given for synthesis. What is tried to achieve is to produce logic algorithms that are true for any example or property of the intended relation.

We explain these seven steps by synthesizing a logic algorithm for $insert/3$, of which an informal specification is given below:

$insert(E, L, R)$ iff non-descending list R is non-descending list L with E, where E is an integer, and L, R are integer lists.

A specification by examples and properties of $insert/3$ is:

$$\begin{array}{lll} \mathcal{E}\ (insert) = \{ & insert(1, [], [1]), & (E_1) \\ & insert(2, [3], [2, 3]), & (E_2) \\ & insert(4, [2], [2, 4]), & (E_3) \\ & insert(2, [1, 3], [1, 2, 3]), & (E_4) \\ & insert(5, [2, 4], [2, 4, 5]), & (E_5) \\ & insert(3, [5, 6], [3, 5, 6]) & (E_6)\} \end{array}$$

$$\begin{array}{lll} \mathcal{P}\ (insert) = \{ & insert(X, [], [X]), & (P_1) \\ & insert(X, [Y], [X, Y]) \Leftarrow X \leq Y, & (P_2) \\ & insert(X, [Y], [Y, X]) \Leftarrow X > Y, \} & (P_3) \end{array}$$

step 1 instantiates the left hand side of the equivalence in the schema, giving out the first approximation of $LA(insert)$.

$LA_1(insert)$: $insert(E, L, R) \Leftrightarrow true$ $\qquad \{E_1 - E_6\}$

This means that $LA_1(insert)$ covers examples from E_1 to E_6 represented by the annotation $\{E_1 - E_6\}$.

step 2 selects the induction parameter X provided that it is of an inductive type. For $insert/3$, all E, L, and R are of inductive types. SYNAPSE2 will sequentially try all of them, and produces different logic algorithms for $insert/3$. The auxiliary parameter Z is selected by trying all E, L, and R. Assume L is the induction parameter and E is the auxiliary parameter. Hence $\{\ X/L,\ Z/E,\ Y/R\ \}$. In the given examples: L is minimal when it is empty ($Minimal(L) \Leftrightarrow L{=}[]$), and it is non-minimal when it has at least one element ($NonMinimal(L) \Leftrightarrow L = [\_|\_]$).

$$
\begin{array}{lll}
LA_2(insert)\colon\ insert(E,L,R) \Leftrightarrow & & \\
 & L = [] & \{E_1\} \\
 & \vee\ L = [\_|\_] & \{E_2 - E_6\}
\end{array}
$$

step 3 can decompose a non-minimal L in various styles: using extrinsic, intrinsic, or logarithmic decomposition. Assume intrinsic decomposition of L is selected. By the Database Method, the *Decompose* is instantiated by *decomposeInsert*:

$$decomposeInsert(L, HL, TL) \Leftrightarrow L = [HL|TL].$$

Hence:

$$
\begin{array}{lll}
LA_3(insert)\colon\ insert(E,L,R) \Leftrightarrow & & \\
 & L = [] & \{E_1\} \\
 & \vee\ L = [\_|\_] \wedge L = [HL|TL] & \{E_2 - E_6\}
\end{array}
$$

step 4 instantiates the recursive call. If all examples covered by the non-minimal non-recursive case are also covered by the recursive case, step 4 does not introduce a non-minimal non-recursive case; otherwise, step 4 introduces a non-minimal non-recursive case.

In our problem, all examples covered by the non-minimal non-recursive case are also covered by the recursive case, so step 4 instantiates only the recursive case.

$$
\begin{array}{lll}
LA_4(insert)\colon\ insert(E,L,R) \Leftrightarrow & & \\
 & L = [] & \{E_1\} \\
 & \vee\ L = [\_|\_] \wedge L = [HL|TL] & \\
 & \qquad\qquad \wedge\ insert(E, TL, TR) & \{E_2 - E_6\}
\end{array}
$$

Note that a non-minimal non-recursive case is not introduced into the logic algorithm, because it is not detected at this step. In this case, it must be introduced into the logic algorithm at another step. A non-minimal non-recursive case may be detected in some cases, and introduced into the logic algorithm, at this step.

step 5, instantiates $Process_k$, $Compose_k$, $Solve$, and $SolveNonMinimal_k$. Instead of instantiating $Process_k$ and $Compose_k$ separately, instantiates both at the same time. The $Process_k(HL, HR, E)$ and $Compose_k(HR, TR, R, E)$ are combined as $ProcComp_k(HL, TR, R, E)$. Let $ProcComp_k/4$, $Solve/3$, and $SolveNonMinimal_k/4$ are instantiated by $pcInsert_k/4$, $solveInsert/3$, and $snmInsert/4$ respectively.

At the beginning, we instantiate the minimal case. The minimal case has only one covered example, which is E_1. Generalizing this example using P_1, Step 5 instantiates $Solve$ by $solveInsert$:

$$solveInsert(E, L, R) \Leftrightarrow R = [E].$$

Then, we instantiate the non-minimal case(s). As we do not know how many non-minimal subcases will be in our logic algorithm, and we know that there will be at least one non-minimal recursive subcase, we start Step 5 assuming that there is one non-minimal case at present. Hence, $c = 1$ at this state. The value of c may change later.

Step 5 abduces two example sets, one for $pcInsert/4$ and the other for $snmInsert/4$, from the examples given for $insert/3$:

$$\begin{array}{lll} \mathcal{E}(pcInsert) = \{ & pcInsert(3, [2], [2,3], 2) & (E_2') \\ & pcInsert(2, [4], [2,4], 4) & (E_3') \\ & pcInsert(1, [2,3], [1,2,3], 2) & (E_4') \\ & pcInsert(2, [4,5], [2,4,5], 5) & (E_5') \\ & pcInsert(5, [3,6], [3,5,6], 3) & (E_6') \end{array}$$

$$\begin{array}{lll} \mathcal{E}(snmInsert) = \{ & snmInsert(3, [], [2,3], 2) & (E_2'') \\ & snmInsert(2, [], [2,4], 4) & (E_3'') \\ & snmInsert(1, [3], [1,2,3], 2) & (E_4'') \\ & snmInsert(2, [4], [2,4,5], 5) & (E_5'') \\ & snmInsert(5, [6], [3,5,6], 3) & (E_6'') \end{array}$$

At this state $c = 1$, this means that we assume that all examples that correspond to the examples in the given example set for $pcInsert/4$, that is $\{E_2 - E_6\}$, will be covered by one non-minimal recursive case, and there is no non-minimal non-recursive case.

First, we find the msg of $\mathcal{E}(pcInsert)$ as follows:

Examples	HL	TR	E	R
E_2'	3	$[2]$	2	$[2,3]$
E_3'	2	$[4]$	4	$[2,4]$
E_4'	1	$[2,3]$	2	$[1,2,3]$
E_5'	2	$[4,5]$	5	$[2,4,5]$
E_6'	5	$[3,6]$	3	$[3,5,6]$
msg	Y	$[K\|M]$	Z	$[A,B\|M]$

In a D&C algorithm, what the recursive call gives out must be used to compose the result of that algorithm. That is, in $pcInsert_k(HL, TR, R, E)$, the constituents of TR must be used to construct R, and the constituents of HL or E may be used to construct R, that is R is to be constructed from the constituents of TR and any of the constituents of HL, E, and any basic constants such as $\{0, nil, ...\}$. We illustrate this in the tabular representation: the constituents of any of the terms between the first bar and the second bar are supposed to be used for constructing the term to the right of the second bar. The bottom line gives the msg of the examples to the left of the first bar.

The most-specific-generalization of $\mathcal{E}(pcInsert)$ gives $pcInsert/4$ as follows:

$$\begin{aligned}&\forall E, HL, TR, R\ (pcInsert(HL, TR, R, E) \Leftrightarrow \\ &\qquad \exists X, Y, K, M, Z, A, B\ ((E = Z) \wedge (HL = Y) \wedge (TR = [K|M]) \wedge \\ &\qquad\qquad (R = [A, B|M])))\end{aligned}$$

which can be rewritten as follows without quantifiers:

$$pcInsert(HL, TR, R, E) \Leftrightarrow R = [\_, \_|M] \wedge TR = [\_|M]$$

We can not instantiate $ProcComp_k(HL, TR, R, E)$ with $pcInsert(HL, TR, R, E)$, that is when $c = 1$ (one non-minimal recursive case covering $\mathcal{E}(pcInsert)$), the constituents of TR aren't used to construct R.

Next, we partition these examples into two sets (then $c = 2$), $\{E'_3 - E'_5\}$ and $\{E'_2, E'_6\}$. In parallel, we partition $\mathcal{E}(snmInsert)$ into two sets, $\{E''_3 - E''_5\}$ and $\{E''_2, E''_6\}$, because we should check if any of the example sets, $\{E_3 - E_5\}$ and $\{E_2, E_6\}$, is covered by a non-minimal non-recursive case, that is if it is necessary to introduce a non-minimal non-recursive case.

1. We find the msg of the example set $\{E''_3 - E''_5\}$.

Examples	HL	TL	E	R
E'_3	2	[]	4	[2, 4]
E'_4	1	[3]	2	[1, 2, 3]
E'_5	2	[4]	5	[2, 4, 5]
msg	Y	K	X	$[Y, B\|C]$

The most-specific-generalization of E''_3, E''_4, and E''_5 gives $snmInsert_1/4$ as follows:

$$\begin{aligned}&\forall E, HL, TR, R\ (snmInsert_1(HL, TL, R, E) \Leftrightarrow \\ &\qquad \exists X, Y, K, M, B, C\ ((E = X) \wedge (HL = Y) \wedge (TL = [K|M]) \wedge \\ &\qquad\qquad (R = [Y, B|C])))\end{aligned}$$

which can be rewritten as follows without quantifiers:

$$snmInsert_1(HL, TR, R, E) \Leftrightarrow R = [HL_{\,|}\_] \wedge TL = \_$$

In a D&C algorithm, in the non-minimal non-recursive case the constituents of the input terms may be used to compose the result of that algorithm. That is, in $snmInsert_k(HL, TL, R, E)$, the constituents of HL, TL, or E may be used to construct R, that is R may be constructed from the constituents of HL, TL, E, and any basic constants such as $\{0, nil, ...\}$.

In $snmInsert_1/4$, the term R is constructed from HL and any other term. Therefore, $\{E_3 - E_5\}$ is not covered by a non-minimal non-recursive case. Then, we have to check if $\{E_3 - E_5\}$ can be covered by a non-minimal recursive case.

We find the msg of $\{E'_3 - E'_5\}$ as follows:

Examples	HL	TR	E	R
E'_3	2	[4]	4	[2, 4]
E'_4	1	[2, 3]	2	[1, 2, 3]
E'_5	2	[4, 5]	5	[2, 4, 5]
msg	Y	$[K\|M]$	X	$[Y, K\|M]$

The most-specific-generalization of E'_3, E'_4, and E'_5 gives $pcInsert_1/4$ as follows:

$$\begin{aligned}&\forall E, HL, TR, R\ (pcInsert_1(HL, TR, R, E) \Leftrightarrow\\ &\qquad \exists X, Y, K, M\ ((E = X) \wedge (HL = Y) \wedge (TR = [K|M]) \wedge\\ &\qquad\qquad (R = [Y, K|M])))\end{aligned}$$

which can be rewritten as follows without quantifiers:

$$pcInsert_1(HL, TR, R, E) \Leftrightarrow R = [HL|TR] \wedge TR = [\_|\_]$$

The term R is constructed from the constituents of TR and any of the constituents of HL and E.

Therefore, $\{E_3 - E_5\}$ is covered by a non-minimal non-recursive case.

2. We find the msg of the example set $\{E''_2, E''_6\}$ as follows:

Examples	HL	TL	E	R
E'_2	3	[]	2	[2, 3]
E'_6	5	[6]	3	[3, 5, 6]
msg	Y	Z	X	$[X, Y\|Z]$

The most-specific generalization of E''_2 and E''_6 gives $snmInsert_2/4$ as follows:

$$\begin{aligned}&\forall E, HL, TR, R\ (snmInsert(HL, TR, R, E) \Leftrightarrow\\ &\qquad \exists X, Y, Z\ ((E = X) \wedge (HL = Y) \wedge (TL = Z) \wedge\\ &\qquad\qquad (R = [X, Y|Z])))\end{aligned}$$

which can be rewritten as follows without quantifiers:

$$snmInsert_2(HL, TR, R, E) \Leftrightarrow R = [E, HL|TL]$$

In $snmInsert_2/4$, the term R is constructed from the constituents of E, HL, and TL. Therefore, $\{E''_2, E''_6\}$ is covered by a non-minimal non-recursive case.

As a consequence, step 5 instantiates $ProcComp/4$ by $pcInsert_1/4$, and a non-minimal non-recursive case, $snmInsert_2/4$, is introduced into the logic algorithm. At this point, we have two non-minimal cases and $c = 2$. Hence:

$$
\begin{array}{lll}
LA_5(insert)\text{: } insert(E,L,R) \Leftrightarrow & & \\
\quad L = [] \quad \wedge R = [E] & & \{E_1\}\ \{P_1\} \\
\vee\ L = [\_|\_] \wedge L = [HL|TL] & & \\
\qquad\qquad \wedge R = [E, HL|TL] & & \{E_2, E_6\} \\
\vee\ L = [\_|\_] \wedge L = [HL|TL] & & \\
\qquad\qquad \wedge insert(TL, TR, E) & & \\
\qquad\qquad \wedge R = [HL|TR] \wedge TR = [\_|\_]\{E_3 - E_5\} & &
\end{array}
$$

step 6, using the Proofs-As-Programs Method [2] based on $P_1 - P_3$, instantiates the $Discriminate_k$.

$$
\begin{array}{lll}
LA_6(insert)\text{: } insert(E,L,R) \Leftrightarrow & & \\
\quad L = [] \quad \wedge R = [E] & & \{E_1\}\ \{P_1\} \\
\vee\ L = [\_|\_] \wedge L = [HL|TL] & & \\
\qquad\qquad \wedge R = [E, HL|TL] \wedge E < HL & & \\
\qquad\qquad \wedge R = [E, HL|TL] & & \{E_2, E_6\}\ \{P_2\} \\
\vee\ L = [\_|\_] \wedge L = [HL|TL] & & \\
\qquad\qquad \wedge R = [HL, E|\_] \wedge E \geq HL & & \\
\qquad\qquad \wedge insert(TL, TR, E) & & \\
\qquad\qquad \wedge R = [HL|TR] \wedge TR = [\_|\_]\{E_3 - E_5\}\ \{P_3\} & &
\end{array}
$$

2.3 The Objectives of The Project

In SYNAPSE2, Step 5 will have to find instantiations for $ProcComp_k$, the $Solve$, and $SolveNonMinimal_k$ from examples that are abduced from the originally given examples of R. These examples are positive, and should be generalized into a logic algorithm tightly, so that the instantiations to be found at the end of generalization should not cover more than the intended relation behind these examples.

The MSG-Method is used as a method in SYNAPSE, and will be used in SYNAPSE2. What is aimed at by my senior project is to simplify and extend the existing MSG-Method (introduced by Flener [2]) independently of the place-holder and its schema, and to re-implement step 5 of SYNAPSE2, which corresponds to the Steps 5 and 6 of SYNAPSE.

3 The MSG-Method

The MSG-Method is used to find the logic algorithm of an intended relation $\mathcal{Q}$, if it is possible to get a logic algorithm using only $=/2$ in the body. For this purpose, a set of examples of $\mathcal{Q}$ is required. Besides the examples, which argument(s) are used in the construction of the others should also be given, stating they are mandatory or optional in this construction. This is done by *construction modes*.

Example 3.1 For the predicate $pcInsert/4$ the examples are given in the following syntax:

$$pcInsert(HL, TR, R, E).$$

In this case, we expect that R should be syntactically constructed from TR, which means that the constituents of TR (variables and constants that occur in TR) should occur in R. Also we expect that the constituents of E and HL may occur in R. Finally, we expect that R should be constructed from only TR, HL, and E. Therefore, TR is mandatory, and both E and HL are optional for R. Then we express this consequence by the construction mode of the arguments as follows:

$$pcInsert(any_1, in_1, out_1, any_1),$$

where in_1 means the argument in the respective position in $pcInsert(HL, TR, R, E)$ is mandatory in constructing the argument in the respective position of out_1,
any_1 means the argument in the respective position in $pcInsert(HL, TR, R, E)$ is optional for constructing the argument in the respective position of R, and
out_1 means the argument in the respective position in $pcInsert(HL, TR, R, E)$ is being constructed from only the arguments in the respective positions of in_1 and any_1.

For instance, $pcInsert(1, [2], [1, 2], 2)$ and $pcInsert(5, [4], [4, 5], 4)$ satisfy the construction mode above; but, $pcInsert(1, [4], [1, 2, 3], 2)$ does not satisfy the construction mode given above (because the constituents of TR aren't used in constructing R), neither does $pcInsert(1, [4], [1, 2, 4], 1)$ (because R is not constructed from the constituents of TR, HL, and E only). □

Informally, the MSG-Method finds the msg of a set of examples of a predicate q/n. If the msg of these examples satisfies the construction mode of the arguments of q/n, then the msg of these examples is said to be *admissible*. If the msg of a set of examples of a predicate q/n is admissible, then these examples are said to be *compatible* with each other, and their msg gives rise to a logic algorithm for q/n. If the msg of a set of examples of a predicate q/n is not admissible, then the MSG-Method partitions this example set into subsets (called *cliques*), in which the examples are compatible with each other. As each clique gives an msg, the set gives rise to a disjunctive logic algorithm covering the original example set.

We know about the concept of msg introduced in Section 2.1. Let's first focus on the concept of admissibility in Section 3.1, and then focus on the concept of compatibility in Section 3.2 before going into details of the MSG-Method.

3.1 The Concept of Admissibility

Up to now, we informally defined a predicate to be admissible if it satisfies the given construction mode. We should formalize it. Let's do it step-by-step, starting with the atom $r(A, I, O)$, where A, I, and O designate terms, and its construction mode $r(any, in, out)$.

Definition 3.1 The *constituents* of a term X are the variables and the constants that occur in X.

We denote the constituents of term X with $constituents(X)$.

Example 3.2 The constituents of $[1,2,3]$ are 1, 2, 3, and nil, and the constituents of $[HL|TL]$ are HL, TL, and nil. □

In a first approximation, if the constituents of I occur in O then $r(A,I,O)$ is admissible, that is if $constituents(I) \subseteq constituents(O)$.

Example 3.3 $pcLength(a,0,1)$ and $pcLength(b,1,2)$ are admissible w.r.t. the construction mode $pcLength(any,in,out)$, and their msg is $pcLength(E,TN,s(TN))$, which is admissible with the construction mode $pcLength(any,in,out)$, because $constituents(TN) \subseteq constituents(N)$. □

Example 3.4 Let's check if the msg of a set of examples of $pcEfface/3$ is admissible or not, if the examples are given as $pcEfface(< HL,E >,TR,R)$ with the construction mode $pcEfface(any,in,out)$. The msg of the example set $\{pcEfface(< d,e >,[],[d])$,$pcEfface(< i,j >,[k],[i,k])\}$ is $pcEfface(< HL,E >,TR,[HL|TR])$, where $R = [HL|TR]$. As TR syntactically occurs in R, the msg of the given example set of $pcEfface/3$ is admissible. □

Example 3.5 Let's check if the msg of a set of examples of $snmEfface/3$ is admissible or not, if the examples are given as $snmEfface(\langle HL, E, TL\rangle, \langle\rangle, L)$ with the construction mode $snmEfface(any, in, out)$. The msg of the example set $\{snmEfface(\langle b,b,[c]\rangle, \langle\rangle, [c])$,$snmEfface(\langle f,f,[g,h]\rangle, \langle\rangle, [g,h])\}$ is $snmEfface(\langle HL, E, TL\rangle, \langle\rangle, TL)$, where $L = TL$.
The msg of the given example set of $snmEfface/3$ is admissible with the construction mode $snmEfface(any, in, out)$, but what is the use of $\langle\rangle$? In fact, L is constructed from TL.
Therefore, not only the constituents of I should occur in O but also O should be constructed from A and I only, for $r(A,I,O)$ to be admissible with the construction mode $r(any,in,out)$, that is if

$$constituents(I) \subseteq constituents(O) \text{ and}$$
$$constituents(O) \subseteq constituents(\langle I,A\rangle) \ \square$$

Example 3.6 Let's check if the msg of a set of examples of $pcFirstN/3$ is admissible or not, if the examples are given as $pcFirstN(HN,\langle TL,TR\rangle,\langle L,R\rangle)$, with the construction mode $pcFirstN(any,in,out)$. The msg of the example set $\{pcFirstN(1,\langle[b],[]\rangle,\langle[a,b],[a]\rangle)$,$pcFirstN(2,\langle[b,c],[b]\rangle,\langle[a,b,c],[a,b]\rangle)\}$ is $pcFirstN(HN,\langle TL,TR\rangle,\langle[X|TL],[X|TR]\rangle)$, where $L = [X|TL]$ and $R = [X|TR]$.
In the msg of the given example set of $pcFirstN/3$, the term $\langle L,R\rangle$ is constructed not only from $\langle TL,TR\rangle$, but also from a new "invented" variable X shared by L and R.
Therefore, $r(A,I,O)$ is admissible if

$$constituents(I) \subseteq constituents(O) \text{ and}$$
$$(constituents(O) - sharedVars(O)) \subseteq constituents(\langle I, A\rangle)$$

where $sharedVars(O)$ is the set of the variables shared by the components of O, if O is a tuple. □

Example 3.7 The msg of $\{pcCompress(a, [], [\langle a, s(0)\rangle]), pcCompress(c, [\langle d, s(0)\rangle], [\langle c, s(0)\rangle, \langle d, s(0)\rangle])\}$ is $pcCompress(HL, TC, [\langle HL, s(0)\rangle|TC])$. In this case, C is constructed from HL,TC, and 0.
We should allow that the basic constants such as $\{0, nil, ...\}$ may be "invented" to construct O in $r(A, I, O)$. Then $r(A, I, O)$ is admissible with the construction mode $r(any, in, out)$ iff

$$constituents(I) \subset constituents(O) \text{ and}$$
$$(constituents(O) - sharedVars(O)) \subseteq (constituents(\langle I, A\rangle) \bigcup \{0, nil, , .\})$$ □

Example 3.8 $pcFirstN'(HL, \langle TN, TR\rangle, \langle s(TN), [HL|TR]\rangle)$ is admissible, with the given construction mode $pcFirstN(any, in, out)$. Notice that TN occurs in N and TR occurs in R, that is TN does not occur in R, and TR does not occur in N. We should take this into consideration in the definition of admissibility. □

Definition 3.2 Let an atom $r(\overrightarrow{A}, \overrightarrow{I}, \overrightarrow{O})$ be given with the construction mode $r(any_1, any_2, ..., any_a, in_1, in_2, ..., in_b, out_1, out_2, ...out_b)$,
where $\overrightarrow{A}$ is a vector of a terms A_i,
$\overrightarrow{I}$ is a vector of b terms I_i,
$\overrightarrow{O}$ is a vector of b terms O_i,
then $r(\overrightarrow{A}, \overrightarrow{I}, \overrightarrow{O})$ is *admissible* w.r.t. the construction mode $r(any_1, any_2, ..., any_a, in_1, in_2, ..., in_b, out_1, out_2, ...out_b)$ iff the following hold:

$$\forall\, i : 1 \leq i \leq b : \; constituents(I_i) \;\subseteq\; constituents(O_i) \tag{1}$$

$$(constituents(\langle \overrightarrow{O} \rangle) - sharedVars(\overrightarrow{O})) \;\subseteq\; (constituents(\langle \overrightarrow{I}, \overrightarrow{A}\rangle) \bigcup \{0, nil, ...\}) \tag{2}$$

In (1), we want to express that each input term at the respective position of in_i is used to construct only the term at respective position of out_i, not the other result terms, so we use I_i (respectively O_i) but not $\overrightarrow{I}$ (respectively $\overrightarrow{O}$).

In (2), we want to express that the result term is constructed from the other terms, the basic constants $\{0, nil, ..\}$, and the shared variables of components of O, if Q is a tuple, so we use $\overrightarrow{O}$ (respectively $\overrightarrow{I}$ and $\overrightarrow{A}$) but not O_i. In fact, $sharedVars(O_i)$ does not make sense, because $sharedVars$ is defined only for n-tuples, where $n > 1$.

However, sometimes O does "invent" constants other than the basic ones (see $parity/2$); we don't cover such problems; if a logic algorithm for $parity/2$ is wanted, then the specifier or the synthesis has to be changed.

Definition 3.3 Let $\mathcal{A}$ be a set of atoms with the same predicates and arities, then $\mathcal{A}$ is *admissible* w.r.t. a construction mode M iff all atoms in $\mathcal{A}$ are admissible w.r.t. M.

Example 3.9 $\{pcPlateau(<\ 2, b\ >, [b], [b, b]),\ pcPlateau(<\ 3, c\ >, [c, c], [c, c, c])\}$ is admissible w.r.t. the construction mode $M \equiv pcPlateau(any, in, out)$ because both $pcPlateau(< 2, b >, [b], [b, b])$ and $pcPlateau(< 3, c >, [c, c], [c, c, c])$ are admissible w.r.t. M. □

Theorem 3.1 Let $\mathcal{A}$ be a non-empty set of atoms with the same predicates and arities, and M be a construction mode for the atoms in $\mathcal{A}$; then, if the msg of $\mathcal{A}$, denoted $msg(\mathcal{A})$, is admissible w.r.t. M then $\mathcal{A}$ is also admissible w.r.t. M.

Proof: The $msg(\mathcal{A})$ is more general than all atoms in $\mathcal{A}$, that is for each atom a_i in $\mathcal{A}$ there exists a substitution θ_i such that $msg(\mathcal{A})\theta_i = a_i$ (by definition of being *more general*). In other words, atom a_i in $\mathcal{A}$ is an instance of $msg(\mathcal{A})$. If $msg(\mathcal{A})$ is admissible w.r.t. M,then every atom a_i in $\mathcal{A}$ is admissible (by the definition of admissibility, msg, and instance). If every atom a_i in $\mathcal{A}$ is admissible then $\mathcal{A}$ is admissible by the definition of admissibility of a set of atoms. □

Example 3.10 The $msg(\{pcPlateau(\langle 2, b\rangle, [b], [b, b]), pcPlateau(\langle 3, c\rangle, [c, c], [c, c, c])\})$ is $pcPlateau(\langle X, Y\rangle, [Y|Z], [Y, Y|Z])$, which is admissible w.r.t. the construction mode $pcPlateau(any, in, out)$, denoted M. As both $pcPlateau(\langle 2, b\rangle, [b], [b, b])$ and $pcPlateau(\langle 3, c\rangle, [c, c], [c, c, c])$, which are instances of $pcPlateau(\langle X, Y\rangle, [Y|Z], [Y, Y|Z])$, are admissible w.r.t. M, the set $\{pcPlateau(\langle 2, b\rangle, [b], [b, b]), pcPlateau(\langle 3, c\rangle, [c, c], [c, c, c])\}$ is admissible w.r.t. M. □

Example 3.11 The atom $insert(1, [2], [1, 2])$ is admissible w.r.t. the construction mode $insert(any, in, out)$, denoted M, and $insert(2, [1], [1, 2])$ is also admissible w.r.t. M; however, the $msg(\{insert(1, [2], [1, 2]), insert(2, [1], [1, 2])\})$, which is $insert(X, [Y], [K, M])$, is not admissible w.r.t. M.

This example shows that the converse of the theorem is not true. □

3.2 The Concept of Compatibility

After giving the concept of admissibility, we can focus on compatibility.

Definition 3.4 Given two atoms, $a1$ and $a2$, which have the same predicates and arities, $a1$ and $a2$ are *compatible* (with each other) w.r.t. a construction mode M iff the msg of $a1$ and $a2$ is admissible w.r.t. M.

Example 3.12 The atoms $pcFirstPlateau(b, [b], \langle [b, b], [] \rangle)$ and $pcFirstPlateau(j, [j], \langle [j, j, k], [k] \rangle)$ are compatible (with each other) w.r.t. the construction mode $pcFirstPlateau(any, in, out)$, denoted M, because their msg $pcFirstPlateau(X, [X], \langle [X, X|Y], Y \rangle)$ is admissible w.r.t. M. $\square$

Example 3.13 The atoms $insert(1, [1], [1, 1])$ and $insert(2, [3], [2, 3])$ are compatible w.r.t. the construction mode $insert(any, in, out)$, denoted M, because their msg $insert(X, [Y], [X, Y])$ is admissible w.r.t. M.

Also, $insert(1, [1], [1, 1])$ and $insert(5, [4], [4, 5])$ are compatible w.r.t. M, because their msg $insert(X, [Y], [Y, X])$ is admissible w.r.t. M.

However, $insert(2, [3], [2, 3])$ and $insert(5, [4], [4, 5])$ are not compatible w.r.t. M, because their msg $insert(X, [Y], [Z, K])$ is not admissible w.r.t. M.

Therefore, compatibility is not transitive. $\square$

Example 3.14 The atom $snmEfface(d, [e], [d], e)$ is not admissible w.r.t. the construction mode $snmEfface(any, any, out, any)$, denoted M.Therefore, $snmEfface(d, [e], [d], e)$ is not compatible with itself w.r.t. M. Hence, the compatibility is not reflexive. $\square$

The compatibility relation is symmetric by definition of admissibility.

Definition 3.5 Let $\mathcal{A}$ be a non-empty set of atoms with the same predicates and arities, then $\mathcal{A}$ is *compatible* w.r.t. a construction mode M iff any distinct two atoms are compatible w.r.t. M.

Example 3.15
The set $\{pcFirstPlateau(b, [b], \langle [b, b], [] \rangle), pcFirstPlateau(j, [j], \langle [j, j, k], [k] \rangle)\}$ is compatible w.r.t. the construction mode $pcFirstPlateau(any, in, out)$, denoted M, because $pcFirstPlateau(b, [b], \langle [b, b], [] \rangle)$ and $pcFirstPlateau(j, [j], \langle [j, j, k], [k] \rangle)$ are compatible (with each other) w.r.t. M. $\square$

Theorem 3.2 Let $\mathcal{A}$ be a set of at least two atoms, where all of atoms in $\mathcal{A}$ are with the same predicates and arities, and let M be a construction mode for the atoms in $\mathcal{A}$; then:

1. $\mathcal{A}$ is compatible w.r.t. M iff $msg(\mathcal{A})$ is admissible w.r.t. M;
2. $\mathcal{A}$ is compatible w.r.t. M iff $\mathcal{A} \bigcup msg(\mathcal{A})$ is compatible w.r.t. M.

Proof:

1. Let's prove the first part of the theorem:

   $\mathcal{A}$ is compatible w.r.t. a construction mode M
   iff any two atoms a_1 and a_2, where a_1,$a_2 \in \mathcal{A}$, are compatible w.r.t. M
   (by the definition of compatibility of a non-empty set of atoms),
   iff the msg of any two atoms a_1 and a_2, where a_1,$a_2 \in \mathcal{A}$, is admissible w.r.t. M

(by the definition of compatibility of atoms),
iff the msg of $\mathcal{A}$ is admissible w.r.t. M
(by the definition of admissibility of a non-empty set of atoms).

2. Let's prove the second part of the theorem:

$$msg(\mathcal{A} \cup msg(\mathcal{A})) = msg(msg(\mathcal{A}), msg(\mathcal{A}))$$
(by the definition of msg of a non-empty set of atoms), (3)

$$msg(msg(\mathcal{A}), msg(\mathcal{A})) = msg(\mathcal{A})$$
(by the definition of msg of two atoms), (4)

$msg(\mathcal{A} \cup msg(\mathcal{A}))$ is admissible w.r.t. M iff $msg(\mathcal{A})$ is admissible (5)

$(\mathcal{A} \cup msg(\mathcal{A}))$ is compatible w.r.t. M iff $msg(\mathcal{A} \cup msg(\mathcal{A}))$ is admissible w.r.t. M
(by the first part of the theorem), (6)

$msg(\mathcal{A})$ is admissible w.r.t. M iff $\mathcal{A}$ is compatible w.r.t. M
(by the first part of the theorem), (7)

$(\mathcal{A} \cup msg(\mathcal{A}))$ is compatible w.r.t. M iff $\mathcal{A}$ is compatible w.r.t. M (by)

. (8)

□

3.3 The Universal Case

After giving the concepts of admissibility and compatibility, we can go into details of the MSG-Method. Let's first focus on the universal case: stating what the universal case of the MSG-Method does in Section 3.3.1 and stating how it does this in Section 3.3.2.

3.3.1 What does the Universal Case do?

Let's first define what we mean by a universal example.

Definition 3.6 A *universal example* for a predicate q/n is the universal closure of an atom of q/n.

Example 3.16 $length([a, b], 2)$ is a universal example for $length/2$.
$length([A, B], 2)$ also is a universal example for $length/2$, that is $\forall A, B\ (length([A, B], 2))$. □

In fact, what's called so far an example is a universal example; and from now on, when we say example, we mean universal example.

In the universal case, a set of examples of a predicate q/n with a construction mode of M is given to find a logic algorithm for q/n, but using only $=/2$ as a primitive. As a result, our logic algorithm for q/n covers the given example set. However, we don't want logic algorithm for q/n to cover only the given example set but also an "extension" of the example set, where this "extension" of the example set reflects our intended relation Q correctly. Let's noe define the *natural extension* of a non-empty set of atoms with the same predicates and arities.

Definition 3.7 Let $\mathcal{A}$ be a set of a non-empty set of universal examples with the same predicate and arities, and $\mathcal{P}$ be a partition of $\mathcal{A}$ into compatible subsets; then the set of atoms covered by the msg of any element of $\mathcal{P}$ is called the *natural extension* of $\mathcal{A}$ w.r.t. $\mathcal{P}$.

Example 3.17 Let $\mathcal{A}$ be a set of examples for $insert/3$ as follows:

$$
\begin{array}{lll}
\mathcal{A}=\{ & insert(1, [2], [1, 2]) & (E_1) \\
& insert(3, [4, 5], [3, 4, 5]) & (E_2) \\
& insert(2, [1], [1, 2]) & (E_3) \\
& insert(4, [3, 5], [3, 4, 6]) & (E_4)
\end{array}
$$

Let $\mathcal{P}$ partition $\mathcal{A}$ into two subsets, that is $\mathcal{P} = \{\mathcal{A}_1, \mathcal{A}_2\}$, where $\mathcal{A}_1 = \{E_1, E_2\}$ and $\mathcal{A}_2 = \{E_3, E_4\}$. Then $msg(\mathcal{A}_\infty)$ is $insert(X, [Y|Z], [X, Y|Z])$ and $msg(\mathcal{A}_\in)$ is $insert(X, [Y|Z], [Y, X|Z])$.

The natural extension of $\mathcal{A}$ w.r.t. $\mathcal{P}$ contains all atoms covered by $msg(\mathcal{A}_1)$ or $msg(\mathcal{A}_2)$. For instance, $insert(2, [4], [2, 4])$ is covered by $msg(\mathcal{A}_1)$ and $insert(4, [2], [2, 4])$ is covered by $msg(\mathcal{A}_2)$, so both are in the natural extension of $\mathcal{A}$ w.r.t. $\mathcal{P}$. □

We can now state the goal of the universal case of the MSG-Method as follows: given an example set of atoms with the same predicates and arities, and a construction mode M, the MSG-Method finds a logic algorithm, defined in terms of the $=/2$ primitive only, covering at least the given example set, and correct w.r.t. some natural extension of the example set. The problem is correctly solved iff that natural extension of the example set is the unknown intended relation Q.

3.3.2 How is the Universal Case solved?

First, we are given an example set $\mathcal{E}$ for a predicate q/n, and a construction mode M. This example set should be large enough to give a large natural extension, therefore to give a correct logic algorithm for q/n.

Next, we construct the compatibility graph w.r.t. M of $\mathcal{E}$, where vertices represent examples and edges represent the compatibility relation w.r.t. M.

Then, we decompose this graph into its connected components, and partition each component into a least number of maximal compatible subgraphs (*cliques*).

Finally, the disjuncts of the msgs of the cliques give a logic algorithm for q/n.

Let's expalin why we need to partition each connected component into cliques. As compatibility is not transitive, the msg of any connected component may not be admissible w.r.t. M. For a component to be compatible w.r.t. M, it should be strongly connected (i.e. it should be a clique). We partition connected components to get cliques.

Example 3.18 Let $\mathcal{A}$ be a set of examples for $insert/3$ as follows:

$$
\begin{array}{ll}
\mathcal{A}=\{insert(3,[3],[3,3]) & (E_1)\\
\quad insert(1,[2],[1,2]) & (E_2)\\
\quad insert(2,[1],[1,2]) & (E_3)
\end{array}
$$

and $M = insert(in, in, out)$.

$\mathcal{A}$ has a compatibility graph which is already connected w.r.t. M. However, its msg $insert(X, [Y|M], [Z, K|M])$ is not admissible w.r.t. M, therefore $\mathcal{A}$ is not compatible w.r.t. M (by first part of Theorem 3.2). We partition the compatibility graph of $\mathcal{A}$ into a least number of cliques, that is we partition $\mathcal{A}$ into two subsets, e.g. $\mathcal{A}_1 = \{E_1, E_2\}$ and $\mathcal{A}_2 = \{E_3\}$, where both $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible w.r.t. M. We don't partition $\mathcal{A}$ into $\mathcal{A}_1 = \{E_1\}$, $\mathcal{A}_2 = \{E_2\}$, and $\mathcal{A}_3 = \{E_3\}$, because then we don't get a least number of cliques from the compatibility graph of $\mathcal{A}$. □

3.4 The Existential Case

We focused on the universal case in the previous section. In parallel to Section 3.3, let's state what the existential case of the MSG-Method does in Section 3.4.1, and how itdoes this in Section 3.4.2.

3.4.1 What does the Existential Case do?

In the universal case, a set of universal examples for a predicate q/n with a construction mode M is given to find a logic algorithm for q/n, using only the $=/2$ primitive. None of the universal examples of q/n need to be evaluated for their correctness w.r.t. a specification of q/n; for instance, $length([A, B], 2)$ is a universal example for $length/2$, and it doesn't need to be checked if it's correct w.r.t. a specification of $length/2$, whatever the instances of A and B are. However, some examples are not universal; for instance, $firstN(A, [b], B)$ is correct w.r.t. a specification of $first/3$ only for a subset of all possible instances of A and B; that is, $firstN(0, [b], [])$ is correct w.r.t. a specification of $firstN/3$, however $firstN(0, [b], [b])$ is not correct w.r.t. that given specification of $firstN/3$. Therefore, the universal closure of $firstN(A, [b], B)$, that is $\forall A, B \ firstN(A, [b], B)$, is not correct w.r.t. a specification of $firstN/3$; however, the existential closure of $firstN(A, [b], B)$, that is $\exists A, B \ firstN(A, [b], B)$, is correct w.r.t. a specification of $firstN/3$ The existential case of the MSG-Method takes the existential closure of atoms, hereafter called the existential examples, into consideration. Let's define what an existential example is.

Definition 3.8 An *existential example* for a predicate q/n is the existential closure of an atom of q/n.

Example 3.19 $firstN(A, [b], B)$ is an existential example for $firstN/3$, that is $\exists A, B \ firstN(A, [b], B)$. □

In the existential case, a set of examples of a predicate q/n with a construction mode M is given to find a logic algorithm for q/n, using the $=/2$ primitive only. As a result, our logic algorithm for q/n not only covers the given example set but also is correct w.r.t. some natural extension of any "admissible alternative" of examples in this example set. Let's define what we mean by "admissible alternative".

Definition 3.9 If any instance a' of any existential example a is admissible w.r.t. a given construction mode M, then a' is an *admissible alternative* of a w.r.t. M.

Example 3.20 The atom $efface(a, [a, b], [b])$ is an admissible alternative of $efface(a, L, [b])$ w.r.t. a given construction mode M, e.g. $firstN(any, out, in)$. The atom $efface(a, [a, b, b], [b])$ also is an admissible alternative of $efface(a, L, [b])$ w.r.t. M. □

Definition 3.10 Let $\mathcal{A}$ be a non-empty set of existential examples with the same predicates and arities, and M be a construction mode for $\mathcal{A}$. The set containing any admissible alternative of each example in $\mathcal{A}$ w.r.t. M is called an *admissible alternative* of $\mathcal{A}$ w.r.t. M. □

Example 3.21 Let $\mathcal{A} = \{efface(a, L, [b]),\ efface(c, L, [c, d])\}$ and M be a construction mode, e.g. $pcPlateau(any, out, in)$. An admissible alternative of $\mathcal{A}$ w.r.t. M is $\{efface(a, [a, b, b], [b]),\ efface(c, [c, d], [c, d])\}$. □

The existential case is solved correctly iff this natural extension of any admissible alternative of the given example set is the unknown intended relation.

3.4.2 How is the Existential Case solved?

First, we are given an example set $\mathcal{E}$ for a predicate q/n and a construction mode M.

Next, we should get the set of all admissible alternatives of $\mathcal{E}$ w.r.t. M.

Finally,for each admissible alternative of $\mathcal{E}$ w.r.t. M, we should invoke the algorithm given for the universal case of the MSG-Method. However, as the set of all admissible alternatives for a set of existential examples for a predicate q/n may be infinite.

Example 3.22 Let $\mathcal{A} = \{efface(a, L, [b]),\ efface(c, L, [c, d])\}$ and M be a construction mode, e.g. $pcPlateau(any, out, in)$. Each admissible alternative of $\mathcal{A}$ contains an instance of $efface(a, L, [b])$, where L is a non-empty list containing any number of a's and at least one b, and an instance of $efface(c, L, [c, d])$, where L is a non-empty list containing at least one c and at least one d. Therefore the set of all admissible alternatives of $\mathcal{A}$ is an infinite set. $\square$

We should somehow prune some of the admissible alternatives that aren't correct w.r.t. a specification of q/n. For instance, (referring to the previous example) $efface(a, [a, b, b], [b])$ is not correct w.r.t. a specification of $efface/3$ and the admissible alternatives of $\mathcal{A}$ w.r.t. M containing $efface(a, [a, b, b], [b])$ should be pruned from the set of all admissible alternatives of $\mathcal{A}$ w.r.t. M. We need some heuristics for this purpose. This remains as a future work.

4 Step 5 of SYNAPSE2

First, we'll state what Step 5 of SYNAPSE2 does in Section 4.1, and then how it does this in Section 4.2.

4.1 What does Step 5 do?

Step 5 instantiates $Process_k$, $Compose_k$, $Solve$, and $SolveNonMinimal_k$ of the D&C logic algorithm schema given in Sction 2.2.

Remember that Step 4 instantiates the recursive call. If all examples, in the example set $\mathcal{E}(q)$ given for a predicate q/n, except the ones $\mathcal{E}_1(q)$ that are covered by the minimal case, that is $\mathcal{E}(q) - \mathcal{E}_1(q)$, are covered by a non-minimal cases, then Step 4 doesn't introduce a non-minimal non-recursive case. If some of the examples in $\mathcal{E}(q) - \mathcal{E}_1(q)$, that is $\mathcal{E}_2(q)$, is covered by a non-minimal non-recursive case but not by a recursive case, then Step 4 introduces a non-minimal non-recursive case covering $\mathcal{E}_2(q)$, and a recursive case covering the rest of the examples in $\mathcal{E}(q) - \mathcal{E}_1(q)$, that is $\mathcal{E}_3(q)$, where $\mathcal{E}_1(q)$, $\mathcal{E}_2(q)$, and $\mathcal{E}_3(q)$ form a partition of $\mathcal{E}(q)$.

$$
\begin{array}{llll}
LA_4(q)\text{: } q(X,Y,Z) \Leftrightarrow & & & \\
& minimal(X) & & \mathcal{E}_1(q) \\
\vee & nonMinimal(X) & \wedge\ decompose(X, \overrightarrow{HX}, \overrightarrow{TX}) & \mathcal{E}_2(q) \\
\vee & nonMinimal(X) & \wedge\ decompose(X, \overrightarrow{HX}, \overrightarrow{TX}) & \\
& & \wedge\ q(TX, TY, Z) & \mathcal{E}_3(q)
\end{array}
$$

where $\mathcal{E}_1(q)$ and $\mathcal{E}_3(q)$ are non-empty sets of examples, but $\mathcal{E}_2(q)$ may be an empty set.

Step 5 transforms $LA_4(q)$ into $LA_5(q)$:

$$
\begin{array}{llll}
LA_5(q)\text{: } q(X,Y,Z) \Leftrightarrow & & & \\
& minimal(X) & \wedge\ sole(X,Y,Z) & \mathcal{E}_1(q)\ \cup\ \mathcal{P}_1(q) \\
\bigvee_{k=1}^{s} & nonMinimal(X) & \wedge\ decompose(X, \overrightarrow{HX}, \overrightarrow{TX}) & \\
& & \wedge\ solveNonMinimal_k(\overrightarrow{HX}, \overrightarrow{TX}, Y, Z) & \mathcal{E}_2(q) \\
\bigvee_{k=1}^{r} & nonMinimal(X) & \wedge\ decompose(X, HX, TX) &
\end{array}
$$

$$\begin{array}{ll} \bigwedge_{i=1}^{t} q(TX_i, TY_i, Z) & \\ \wedge\ process_k(\overrightarrow{HX}, \overrightarrow{HY}, Z) & \\ \wedge\ compose_k(\overrightarrow{HY}, \overrightarrow{TY}, Y, Z) & \qquad\qquad \mathcal{E}_3(q) \end{array}$$

where $\mathcal{E}_1(q)$, $\mathcal{E}_2(q)$, and $\mathcal{E}_3(q)$ form a partition of $\mathcal{E}(q)$, such that $\mathcal{E}_1(q)$ and $\mathcal{E}_3(q)$ are non-empty sets of examples, but $\mathcal{E}_2(q)$ may be an empty set, and $\mathcal{P}_1(q)$ is a subset of $\mathcal{P}(q)$, the property set for q/n. The sets $\mathcal{E}_2(q)$ and $\mathcal{E}_3(q)$ may not be the same ones as in $LA_4(q)$.Note that $s + t{=}c$, where c is the form variable used in the schema given in Section 2.2. to represent the number of non-minimal cases.

4.2 How does Step 5 solve the Problem?

First, Step 5 instantiates $Solve/3$ by $solve/3$, invoking the MsSG-Method on $\mathcal{E}_1(q)$ and $\mathcal{P}_1(q)$.

Step 5 instantiates $Process_k$ and $Compose_k$ at the same time. The $Process_k(\overrightarrow{HX}, \overrightarrow{HY}, Z)$ and $Compose_k(\overrightarrow{HY}, \overrightarrow{TY}, Y, Z)$ are combined as $ProcComp_k(\overrightarrow{HX}, \overrightarrow{TY}, Y, Z)$. Step 5 instantiates $ProcComp_k/4$ and $SolveNonMinimal_k/4$ by $procComp_k/4$ and $solveNonMinimal_k/4$ respectively.For this purpose, Step 5 abduces two example sets for $procComp_k$ and $solveNonMinimal_k$ from $\mathcal{E}_3(q)$ as $\mathcal{E}(procComp)$ and $\mathcal{E}(solveNonMinimal)$ respectively., such that for any example E_i in $\mathcal{E}_3(q)$, there exist corresponding examples E_i' in $\mathcal{E}(procComp)$ and E_i'' in $\mathcal{E}(solveNonMinimal)$.

Next, Step 5 partitions $\mathcal{E}(procComp)$ into a least number of compatible sets invoking the MSG-Method with the construction mode $procComp(any, in, out, any)$. If the MSG-Method yields "too many" cliques, then call another method. The criterion for "too many" is given in Heuristic 1 as follows:

Heuristic 4.1 If the number of cliques the MSG-Method yields is greater than or equal to the number of properties, then there are "too many" cliques.

Step 5 also divides $\mathcal{E}_3(q)$ and $\mathcal{E}(solveNonMinimal)$ w.r.t. the partition of $\mathcal{E}(procComp)$; that is if $\mathcal{E}(procComp)$ is partitioned into a least number of compatible sets $\mathcal{E}_1(procComp)$,...,$\mathcal{E}_c(procComp)$, where $n{\geq}1$, then $\mathcal{E}_3(q)$ is partitioned into $\mathcal{E}_31(q)$,...,$\mathcal{E}_3c(q)$, and $\mathcal{E}(solveNonMinimal)$ is partitioned into $\mathcal{E}_1(solveNonMinimal)$,..., $\mathcal{E}_c(solveNonMinimal)$, such that if E_n' is in $\mathcal{E}_i(procComp)$ then the corresponding examples E_n'' and E_n are in $\mathcal{E}_i(solveNonMinimal)$ and $\mathcal{E}_3i(q)$ respectively, where $1{\leq}i{\leq}c$ and n is greater than or equal to 1 and is less than or equal to the number of elements in $\mathcal{E}_3(q)$.

Then, for each of $\mathcal{E}_1(solveNonMinimal)$,...,$\mathcal{E}_c(solveNonMinimal)$, Step 5 checks if $msg(\mathcal{E}_i(solveNonMinimal))$ is admissible or not w.r.t. the given construction mode $solveNonMinimal(any, any, out, any)$, denoted M. After finding $msg(\mathcal{E}_i(solveNonMinimal))$, Step 5 should decide if it should introduce a non-minimal non-recursive case or not. This is decided by Heuristic 2.

Heuristic 4.2 If the result parameter exists, that is $constituents(Y){\neq}\emptyset$, then if $\mathcal{E}_i(solveNonMinimal)$, which is a set among the partitioned sets $\mathcal{E}_1(solveNonMinimal)$,...,$\mathcal{E}_c(solveNonMinimal)$, corresponding to $\mathcal{E}_31(R)$,...,$\mathcal{E}_3c(R)$ respectively, has an admissible msg w.r.t. the construction mode

$solveNonMinimal(any, in, out)$, where $|\overrightarrow{TX}|$=$|Y|$=1 for $solveNonMinimal(\langle\overrightarrow{HX}, \overrightarrow{TX}, Z\rangle, \langle\rangle, Y)$, then recursion is useless for $\mathcal{E}_3 i(R)$.

If $msg(\mathcal{E}_i(solveNonMinimal))$ is admissible w.r.t. M, then $\mathcal{E}_3 i(q)$ is united to $\mathcal{E}_2(q)$, and eliminated from $\mathcal{E}_3(q)$; otherwise, $msg(\mathcal{E}_i(procComp))$ is used to obtain a recursive case.

Finally, we invoke the MSG-Method on the abduced examples $\mathcal{E}(solveNonMinimal)$ from $\mathcal{E}_2(q)$ with the construction mode $solveNonMinimal(any, any, out, any)$ to introduce the non-minimal non-recursive case(s), if any.

Example 4.1 Let's illustrate Step 5 for *efface/3*.

$$
\begin{array}{lll}
\mathcal{E}(\mathit{efface}) = \{ & \mathit{efface}(a, [a], []) & (E_1) \\
& \mathit{efface}(b, [b, c], [c]) & (E_2) \\
& \mathit{efface}(e, [d, e], [d]) & (E_3) \\
& \mathit{efface}(f, [f, g, h], [g, h]) & (E_4) \\
& \mathit{efface}(j, [i, j, k], [i, k]) \ \} & (E_5) \\
\mathcal{P}(\mathit{efface}) = \{ & \mathit{efface}(X, [X|T], T) & (P_1) \\
& \mathit{efface}(X, [Y, X|T], [Y|T]) \ \} & (P_2)
\end{array}
$$

Logic algorithm we get after Step 4 is as follows:

$$
\begin{array}{lll}
LA_4(\mathit{efface}): \mathit{efface}(E, L, R) \Leftrightarrow & & \\
\quad L = [\_] & & \{E_1\} \\
\quad \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] & \{E_2, E_4\} \\
\quad \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] & \\
& \wedge\ \mathit{efface}(E, TL, TR) & \{E_3, E_5\}
\end{array}
$$

Note that Step 4 has introduced a non-minimal non-recursive case, because it detected that $\{E_2, E_4\}$ can not be covered by a recursive case.

First, Step 5 instantiates $Solve/3$ by *solveEfface/3*, generalizing E_1 using P_1.

$$solveEfface/3\ Leftrightarrow\ L = [E]\ wedge\ R = [\_]$$

Let's
assume that Step 5 instantiates $ProcComp_k$ by $pcEfface_k$, and $SolveNonMinimal_k$ by $snmEfface_k$ for *efface/3*.

Next, Step 5 abduces $\mathcal{E}(pcEfface)$ and $\mathcal{E}(snmEfface)$, for now c=2 (i.e. two non-minimal cases, note that one of them is introduced at Step 4), as follows:

$$
\begin{array}{lll}
\mathcal{E}(\mathit{pcEfface}) = \{ & \mathit{pcEfface}(d, [], [d], e) & (E_3') \\
& \mathit{pcEfface}(i, [k], [i, k], j) \ \} & (E_5')
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{E}(\mathit{snmEfface}) = \{ & \mathit{snmEfface}(d, [e], [d], e) & (E_3'') \\
& \mathit{snmEfface}(i, [j, k], [i, k], j) \ \} & (E_5'')
\end{array}
$$

Step 5 invokes the MSG-Method on $\mathcal{E}(pcEfface)$ with the construction mode $pcEfface(any, in, out, any)$, denoted M_1. As $msg(\mathcal{E}(pcEfface))$ is $pcEfface(HL, TR, [HL|TR], E)$, which is admissible w.r.t. M_1, the $msg(\mathcal{E}(pcEfface))$ is compatible w.r.t. M_1. Therefore, $\mathcal{E}(pcEfface)$ forms already a clique, and it doesn't need to be partitioned. Hence, neither $\mathcal{E}(snmEfface)$ nor $\{E_3, E_5\}$ is divided into subsets.

Then, Step 5 finds $msg(\mathcal{E}(snmEfface))$ w.r.t. the construction mode $snmEfface(any, any, out, any)$, denoted M_2, as $snmEfface(HL, [E|TTL], [HL|TTL], E)$, which is not admissible w.r.t. M_2. Therefore, Step 5 uses $msg(\mathcal{E}(pcEfface))$ to instantiate a recursive case.

Finally,we invoke the MSG-Method on the abduced examples $\mathcal{E}(snmEfface)$ from $\{E_2, E_4\}$ with the construction mode M_2.

$$
\begin{array}{lll}
\mathcal{E}(snmEfface)=\{ & snmEfface(b, [c], [c], b) & (E_2'') \\
 & snmEfface(f, [g, h], [g, h], f)\ \} & (E_4'')
\end{array}
$$

The $msg(\mathcal{E}(snmEfface))$ is $snmEfface(HL, TL, R, E)$, which is admissible w.r.t. M_2. Therefore, $msg(\mathcal{E}(snmEfface))$ is used to instantiate the non-minimal non-recursive case.

Hence,

$$
\begin{array}{llll}
LA_5(efface): & efface(E, L, R) \Leftrightarrow & & \\
 & \quad L = [\_] & \wedge\ L = [E] \wedge R = [] & \{E_1, P_1\} \\
 & \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] & \\
 & & \wedge\ HL = E \wedge R = TL & \{E_2, E_4\} \\
 & \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] & \\
 & & \wedge\ efface(E, TL, TR) & \\
 & & \wedge\ R = [HL|TR] & \{E_3, E_5\}
\end{array}
$$

Step 6 will yield a logic algorithm for $efface/3$ as follows:

$$
\begin{array}{llll}
LA_6(efface): & efface(E, L, R) \Leftrightarrow & & \\
 & \quad L = [\_] & \wedge\ L = [E] \wedge R = [] & \{E_1\} \\
 & \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] & \\
 & & \wedge\ HL = E \wedge R = TL & \{E_2, E_4, P_1\} \\
 & \vee\ L = [\_|\_] & \wedge\ L = [HL|TL] & \\
 & & \wedge\ HL \neq E & \\
 & & \wedge\ efface(E, TL, TR) & \\
 & & \wedge\ R = [HL|TR] & \{E_3, E_5, P_2\}
\end{array}
$$

□

5 Conclusion

After introducing some preliminaries, we focused on SYNAPSE2, an upgraded version of SYNAPSE, which is an implementation of an inductive synthesis mechanism that semi-automatically synthesizes D&C logic algorithms in a schema-guided way, from non-incrementally given examples nd properties of an intended relation.

The synthesis mechanism of SYNAPSE2 differs from that of SYNAPSE in the following way:

- In SYNAPSE, the given examples of the intended relation are considered as either ground or non-ground examples.
  In SYNAPSE2, the given examples of the intended relation are considered as either universal or existential examples.

- In SYNAPSE, we can not easily present the synthesis, if the predicate is called with the arguments in a different order than in the schema.
  In SYNAPSE2, this is more flexible, in the sense that the instantiation of the actual computations to the place-holders of the given schema is done by lambda-calculus, in order to consider the projection, when there is no auxiliary parameter, and the reordering of the parameters, when the order of the parameters in the predicate of the intended relation is to be reordered to coincide with the order of the parameters in the schema.

- In SYNAPSE, at Step 4, the examples assumed to be covered by the recursive case are checked if they are admissible or not w.r.t. the construction mode M, which is $q(any, in, out)$, that is $|any|$=$|in|$=$|out|$=1 for a predicate q/n. If they are not admissible w.r.t M, then these non-admissible examples w.r.t. M are used to introduce a non-minimal non-recursive case.
  In SYNAPSE2, at Step 4, the admissibility check of the examples covered by the recursive case w.r.t. M is removed. Step 4 introduces a non-minimal non-recursive case only if it can detect that some examples are not covered by a recursive case. In SYNAPSE2, the admissibility is a concept for msg, for modularity purposes.

- In SYNAPSE, the admissibility of an atom w.r.t. a construction mode is defined with the first part of the definition of the admissibility of an atom in SYNAPSE2. In this project, we incrementally developed the admissibility concept starting from its definition in SYNAPSE. Also, the compatibility concept introduced in SYNAPSE is extended, in order not to consider a non-admissible atom compatible with itself w.r.t. a construction mode. Finally, the theorems related to the concepts of admissibility and compatibility, introduced in SYNAPSE, are revised and proved. Therefore, the MSG-Method is conceptually simplified, as the examples do not need to be checked for their admissibility before invoking the MSG-Method on themselves w.r.t. a construction mode.

- In SYNAPSE2, Steps 5 and 6 of SYNAPSE are combined as Step 5. The simplification that we don't check the admissibility of the examples at Step 4 led to this combination; but more than this, how to dcide a non-minimal non-recursive case will be introduced is simplified and is more elegant, as there is no recomputtation, such as invoking the MSG-Method on an example set more than once with different construction modes.

The simplification and extension of the existing MSG-Method is not so pwerful yet, because the concept of admissibility is not defined so powerfully yet. Let's give an example to show what the definition of admissibility lacks.

Example 5.1 Let $\mathcal{E}(parity)$ be an example set for $parity/2$ as follows:

$$\mathcal{E}(arity)=\{\ parity([], even),\\ parity([a], odd),\\ parity([b,b], even)\ \}$$

and the construction mode $parity(in, out)$, denoted M.

The constituents of [], $[a]$, and $[b,b]$ don't occur in *even*, *odd*, and *even* respectively; and *even* and *odd* are neither constructed from the constituents of [], $[a]$, $[b,b]$, and basic constants, nor shared variables. In fact, they are "local constant"s of $parity/2$.

Although, this is the case, intuitively *even* and *odd* are constructed "using" [], $[a]$, and $[b,b]$, and [], $[a]$, and $[b,b]$ are "used" to construct *even*, *odd*, and *even* respectively. That is $\mathcal{E}(arity)$ should be admissible w.r.t. M. □

The admissibility definition may be extended in the following ways:

- The equations in the definition of admissibility may be changed as follows:
  $\forall\ i : 1 \le i \le b :\ usedToConstruct(constituents(I_i), constituents(O_i)$
  and
  $constructedUsing((constituents(<\overrightarrow{O}>) - sharedVars(<\overrightarrow{O}>)),$
  $(constituents(<\overrightarrow{I}, \overrightarrow{A}>) \bigcup \{0, nil, ...\}))$
  where $usedToConstruct(a, b)$ iff a is used to constrcut b,
  and $constructedUsing(a, b)$ iff a is constructed using b.

  We mean that, $usedToConstruct/2$ and $constructedUsing/2$ need not be $\subseteq /2$.

- There may be a function f, where we can replace the equations in the definition of admissibilty as follows:
  $\forall\ i : 1 \le i \le b :\ f(constituents(I_i))\ \subseteq\ constituents(O_i)$
  and
  $f^{-1}((constituents(<\overrightarrow{O}>) - sharedVars(<\overrightarrow{O}>)))\ \subseteq$
  $(constituents(<\overrightarrow{I}, \overrightarrow{A}>) \bigcup \{0, nil, ...\})$

- The local constants may be somehow introduced into the definition of admissibility, the equations in the definition of the admissibility may be refined as follows:
  $\forall\ i : 1 \le i \le b :\ constituents(I_i)\ \subseteq\ constituents(O_i)$
  and
  $(constituents(<\overrightarrow{O}>) - sharedVars(<\overrightarrow{O}>))\ \subseteq$
  $(constituents(<\overrightarrow{I}, \overrightarrow{A}>) \bigcup \{0, nil, ...\} \bigcup\ localConstants(r))$

- Or any other alternatives...

The definition of admissibility needs an extension, that is the extension of the definition of admissibility remains as a future work. For now, SYNAPSE2 does not cover problems, such as $parity/2$, but these problems may be solved by imposing the specifier change the specification accordingly, or changing the synthesis, which again remains as a future work.

Another future work remained is in the existential case of the MSG-Method: the set of all admissible alternatives of the given set of existential examples may be infinite. Invoking the MSG-Method on an infinite set needs some heuristics to prune some elements of the list. This is examined in Section 3.4.2.

If we consider SYNAPSE2, the MSG-Method doesn't fulfill the needs of SYNAPSE2 at Step 5. We need other methods, because the MSG-Method yields a logic algorithm which can be defined with $=/2$ primitive only. As a related work, there is a method, called Synthesis Method, which handles the recursive $ProcComp_k$.

Example 5.2 In insertion-sort, $insert/3$ is used to instantiate $ProcComp_k$. The abduced example set for $pcSort$ is used as an example set to infer a logic algorithm for $pcSort$, which is $insert$, by invoking the synthesis mechanism. □

As a future work, there should be a method, which handles the case that $ProcComp_k$ should be instantiated by a non-recursive predicate, probably a predicate pre-introduced.

Example 5.3 In $minimum/2$, the $pcMinimum$ needs to use $\leq$ and $>$, that is:

$$\begin{aligned} pcMinimum(M1, M2, M) \Leftrightarrow\ & M1 \leq M2 \wedge M = M1 \\ & M1 > M2 \wedge M = M2 \end{aligned}$$

□

As a conclusion, we not only need more extended definition of admissibility, hence compatibility and the MSG-Method, but also need other methods than the MSG-Method.

References

[1] Pierre FLENER. *Logic Program Schemata: Synthesis and Analysis.* Tech. Report, BU-CEIS-9502, 1995.

[2] Pierre FLENER. *Logic Program Synthesis from Incomplete Information.* Kluwer Academic Publishers, 1995.

[3] Pierre FLENER. Tech. Report, BU-CEIS-96xx, 1996.

Appendix

A Informal Specifications for Some Predicates

append(A,B,C) iff C is the concatenation of B to the end of A,
where A, B, C are lists.

compress(L,C) iff the ith element of C is a <vi,ci> couple,
such that the ith plateau of L has ci elements equal to vi,
where L is a list and C a compact list.

delOddElems(L,R) iff R is L without its odd elements,
where L, R are integer-lists.

efface(E,L,R) iff R is L without its first existing occurrence of E,
where E is a term, L a non-empty list, and R a list.

firstPlateau(L,P,S) iff P is the first plateau of L, and S is the
corresponding suffix of L,
where L is a non-empty list, P a plateau, and S a list.

halves(L,F,S) iff F is the first half of L, and S is the second half of L,
such that S has 0 or 1 elements more than F,
where L, F, S are lists.

insert(E,L,R) iff R is L with E added to it,
where E is an integer and L, R are non-decreasing integer-lists.

integer(N) iff N is an integer,
where N is a term.

intList(L) iff L is an integer-list,
where L is a term.

length(L,N) iff N is the number of elements of L,
where L is a list and N a natural number.

member(E,L) iff E is an element of L,
where E is a term and L a non-empty list.

member(A,N,K,I) iff K is the Ith element of A,
where A is a non-decreasing integer-array indexed from 1 to integer N,
and K is an integer.

merge(A,B,C) iff C is the merger of A and B,
where A, B, C are non-decreasing integer-lists.

odd(N) iff N is odd,

where N is an integer.

ordered(S) iff S is non-decreasing,
where S is an integer-list.

partition(L,P,S,B) iff S (respectively B) contains the elements of L that are smaller than (respectively bigger than or equal to) P (which is called the pivot),
where L, S, B are integer-lists and P is an integer.

permutation(L,P) iff P is a permutation of L,
where L, P are lists.

plateau(N,E,P) iff P is a plateau of N elements equal to E,
where N is a positive integer, E a term, and P a non-empty list.

reverse(L,R) iff R is the reverse of L,
where L, R are lists.

sort(L,S) iff S is a non-decreasing permutation of L,
where L, S are integer-lists.

sum(L,S) iff S is the sum of the elements of L,
where S is an integer and L an integer-list.

B Implementation for Finding Cliques of a Graph

```
% powerset(As,Ss) iff Ss denotes all subsets of As except the empty set
% where As is a list of atoms and Ss is a list of lists of atoms.

powerset(As,Ss):-findall(S,(subsetG(S,As), S\==[]),Ss).


% subsetG(A,B) iff A is a subset of B
% where  A and B are lists.
% ... used for generating subsets.

subsetG([],[]).
subsetG(S,[H|T]) :-
        S=[H|V],
        subsetG(V,T).
subsetG(S,[_|T]) :-
        subsetG(S,T).
% subsetG(A,B) iff A is a subset of B
% where  A and B are lists.
% ... used for verifying subsets.

subsetV([],_).
subsetV([H|T],L2):-
        delete(H,L2,NewL2),
        subsetV(T,NewL2).

delete(E,[E|R],R).
delete(E,[HL|TL],[HL|TR]) :-
        delete(E,TL,TR).



%-----------------------------------------
%-----------------------------------------


% equalSets(L1,L2) iff L1 and L2 represent the same set
% where L1 and L2 are lists of elements of same type.

equalSets([],[]).
equalSets(L,S):-length(L,X), length(S,X), haveSameElts(L,S).

% haveSameElts(L1,L2) iff L1 and L2 have the same elements
% where L1 and L2 are lists of elements of same type.
```

```
haveSameElts([],_).
haveSameElts([HL|TL],S):- member(S,HL), haveSameElts(TL,S).


%-----------------------------------------


% complete(As,L,M,Cl) iff the component L is complete, i.e. a clique,
% via compatibility w.r.t. M, and Cl is an atom denoting Pos, the positions of
% atom in L,and their msg MsgL
% where L is a non-empty list of atoms, As is a list of atoms, Cl is an atom,
%       and M ...

connected(b,d,_).
connected(d,b,_).
connected(a,c,_).
connected(c,a,_).

%complete(_,[],_,clique([],_)).
%complete(As,[HL|TL],M,clique([HPos|TPos],_)):-
%                             findall(T, connected(HL,T,M), Ts),
%                              subsetV(TL,Ts),
%                              select([HPos],As,[HL]),
%                              complete(As,TL,M,clique(TPos,_)).
complete(As,L,M,clique(Pos,MsgL)):-
                      msg(L,MsgL),
                      admissible(MsgL,M),
                      select(Pos,As,L).


% completeComponents(As,Ss,M,CCs) iff CCs denotes the complete components
% among the subsets Ss of As, where connectedness is via compatibility
% w.r.t. the given costruction mode M
% where Ss and CCs are non-empty lists of lists of atoms, As is alist of atoms,
%        and M is ...

completeComponents(As,Ss,M,CCs):-completeComponents(As,Ss,M,[],CCs).


% completeComponents(As,Ss,M,CurCCs,CCs) iff CCs denotes the complete
% components among the subsets Ss of As, where connectedness is via
% compatibility w.r.t. the given costruction mode M, and CurCCs denotes
% the currently detected complete components
% where Ss, CurCCs, and CCs are non-empty lists of lists of atoms, As is alist
%       of atoms, and M is ...
```

```
completeComponents(_,[],_,CurCCs,CurCCs):-!.
completeComponents(As,[HSs|TSs],M,CurCCs,CCs):-
               complete(As,HSs,M,Cl),!,
               TempCCs=[Cl|CurCCs],
               completeComponents(As,TSs,M,TempCCs,CCs).
completeComponents(As,[_|TSs],M,CurCCs,CCs):-
               completeComponents(As,TSs,M,CurCCs,CCs).


%-----------------------------------------
%-----------------------------------------


% sumLength(S,L) iff L is the sum of lengths of lists X in "clique(X,_)"s of S
% where S is a list of atoms of "clique/2" and L is an integer.

sumLength(L,S):-sumLength(L,0,S).
sumLength([],S,S).
sumLength([clique(HS,_)|TS],T,L):-length(HS,LHS),
                        T1 is T+LHS,
                        sumLength(TS,T1,L).


%-----------------------------------------



% union(L1,L2,L3) iff L3 is the union of lists L1 and L2
% where L1,L2, and L3 are lists of elements of same type.

union([],L,L):-!.
union(L,[],L):-!.
union([HL1|TL1],L2,L3):-member(L2,HL1),!,
                       union(TL1,L2,L3).
union([HL1|TL1],L2,[HL1|L3]):-union(TL1,L2,L3).


% unionall(S,U) iff U is the union of all lists occuring in "clique/2"s of S
% where S is a list of atoms of "clique/2", and U is a list of atoms.


unionall(S,SU):-unionall(S,[],SU).
unionall([],U,U):-!.
unionall([clique(HS,_)|TS],TU,SU):-union(HS,TU,TU1),
                        unionall(TS,TU1,SU).
```

```
%----------------------------------------


% intersection(L1,L2,L3) iff L3 is the intersection of L1 and L2
% where L1, L2, and L3 are lists of elements of same type.

intersection([],_,[]):-!.
intersection(_,[],[]):-!.
intersection([HL1|TL1],L2,[HL1|TL3]):-member(L2,HL1), !,
                                      intersection(TL1,L2,TL3).
intersection([_|TL1],L2,L3):-intersection(TL1,L2,L3).



% intersectionall(S,I) iff I is the intersection of all lists occuring in
% "clique/2"s ofS
% where S is a list of atoms of "clique/2", and I is a list of atoms.


intersectionall([clique(S,_)],S):-!.
intersectionall([clique(HS1,_),clique(HS2,_)|TS],I):-
                               intersectionall([clique(HS2,_)|TS],NI),
                               intersection(HS1,NI,I).


%----------------------------------------


% partition(As,CCs,S) iff S is a partition of As into cliques, where each
% clique is an element of CCs
% where As is a list of atoms, and CCs and S are lists of atoms of "clique/2"

partition(As,CCs,S):- length(As,LAs),
                      subsetG(S,CCs),
                      sumLength(S,LAs),
                      unionall(S,SU),
                      equalSets(SU,As),
                      intersectionall(S,[]).


%----------------------------------------

% findSmallest(L,S) iff S is one of the smallest lists in L
% where L is a non-empty list of lists of elements of type X, and S is a list
% of elements of type X.
```

```
findSmallest(L,S):-findTheSmallest(L,SC),
                   findEqual(L,SC,S).

% findEqual(L,L1,L2) iff L2 is a list, in L, equal in length with L1,
% a member of L
% where L is a non-empty list of lists of elements of type X, and L1 and L2
% are non-empty lists of elements of type X.

findEqual(L,L1,L2):-length(L1,X),member(L,L2),length(L2,X).


% findTheSmallest(L,S) iff S is the smallest list in L
% where L is a non-empty list of lists of elements of type X, and S is a list
% of elements of type X.

findTheSmallest([L],L):-!.
findTheSmallest([HL1,HL2|TL],SC):-findTheSmallest([HL2|TL],NSC),!,
                                  smaller(HL1,NSC,SC).

% smaller(L1,L2,SL) iff SL is the smallest of L1 and L2
% where L1, L2, and SL are lists of the same type elements.

smaller(X,Y,X):-length(X,LX),
                length(Y,LY),
                LX =< LY,!.
smaller(_,Y,Y).


%-----------------------------------------


% cliques(As,CCs,Cls) iff Cls is a partition  of As into a least number of
% cliques, where each clique is an element of CCs
% where As is a list of elements of type X, and CCs and Cls are lists of lists
% of elements of type X.

cliques(As,CCs,Cls):-findall(S,partition(As,CCs,S),Cs),!,
                     findSmallest(Cs,Cls).


%-----------------------------------------
%-----------------------------------------


% getCliques(As,M,Cls) iff Cls is a partition of the given atoms As into a
```

```
% least number of cliques w.r.t. the given construction mode M
% where Cls and As are lists of atoms, and M is ...

getCliques([],_,[]):-!.
getCliques(As,M,Cls):-powerset(As,Ss),
                      completeComponents(As,Ss,M,CCs),
                      length(As,N),iota(N,Ps),
                      cliques(Ps,CCs,Cls).
```