On Moded Functional IP Syntheses by Reusing Composition Structures

Atsushi HIRATA[1], Makoto HARAGUCHI[1], Ken SADOHARA[2]
[1]Division of Electronics and Information Engineering, Hokkaido University
N13-W8, Kita-ku, Sapporo, 060-0813 Japan
E-mail:atsushi@db-ei.eng.hokudai.ac.jp
E-mail:makoto@db-ei.eng.hokudai.ac.jp
[2]Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, 305-0045 Japan
E-mail:sadohara@etl.go.jp

Abstract

This paper presents an algorithm for synthesizing intelligent pad (IP), given a set of positive examples of a target IP and a source IP. The source IP is intended to be reused in building the target and is in fact embedded into the target IP. More exactly speaking, the target IP has an extended composition structure of the source IP, where the primitive IPs used in the source can be replaced with another IPs under some type constraints on slots. To realize such a reuse of IP in a framework of logic program syntheses, we introduce an input/output mode declaration for each IP that specifies a flow of data passing between several IPs. As a consequence, we can represent each composition structure of moded IP as a linearly covering program clause. Using this class of clauses, we define a family of hypothesis spaces and design a new refinement operator for this family based on which our synthesis algorithm searches for a composition structure achieving the intended behavior exemplified by data presentation.

1 Introduction

This paper examines a possibility of applying theoretical results on inductive inference to solve a synthesis problem of a componentware system, called the IntelligentPad, and shows an algorithm for synthesizing components in that system.

The IntelligentPad system

The IntelligentPad system [1] has been developed to realize and operate various kinds of knowledge media on computers. For the knowledge media such as texts, images, sounds, movies, database objects and programs themselves have various data forms, so it is indeed necessary to have a computer architecture that can handle and distribute them on a uniform platform. For this purpose, the IntelligentPad system provides an environment in which we can produce a collection of software components, called intelligent pads (IPs for short) together with functional linkages between them.

Each IP represents each knowledge medium, where the actual content is wrapped within the IP so that it is distributed to and reusable in other media or in application programs without depending on each data representation form. Moreover, to achieve functional linkages between IPs, every IP is designed to have several slots, through which data should pass from and to another IP, and a family of internal methods reacting on the slot values.

When a slot in an IP is connected with a slot of another IP, the data values of the former are allowed to be passed from and to the latter. Once a slot value is updated by such a data passing, an internal method associated with the slot is then invoked and alters another slot value. The process of sending, receiving and updating slot values is completely determined by the slot connection structure. So only thing user must do is to introduce the slot connection, while the system performs the data handling according to the connection. A composite IP is thus obtained by connecting several slots over several IPs. From the definition, synthesizing a composite IP is equivalent to defining a connection structure between more primitive IPs so that it achieves intended behavior.

Learnability

From a viewpoint of synthesizing complex composite IPs, it is desirable to have a method to guide the connection structure so that the composite IP obtained by the connection achieves the intended behavior. For this purpose, this paper tries to present a first theoretical framework of reusing connection structures of existing IPs to synthesize a new target IP, given an example specification of the target. The techniques for guiding the connection and for synthesizing composite IPs will become very important, since the number of IPs is expected to be increasing rapidly.

In the actual IntelligentPad system, every IP has a corresponding GUI that can support understanding what function it has and what kind of functional linkage is possible with another IPs. However in this paper we consider only a logical structure of IP, the slot connection structure, and present an algorithm to learn it from example specification. The usage of GUI in solving synthesis problem is left as a future work.

As we have already pointed out, the problem of synthesizing composite IPs is solved by giving a method for proposing slot connection structures between more primitive IPs. So if we are able to represent the connections by logical formula, namely by definite clauses, then we can directly apply the results of studies on inductive learning of logic programs. To this end, we make an assumption that every slot has an input/output mode declaration, and show that the slot connection structure under this assumption can define a linear data flow and is therefore represented by a linearly covering program clause [3]. It is

therefore possible to apply the techniques developed in the research area of inductive learning and inductive logic programming.

Analogical transformation of slot connection structures

Although the learnability of slot connection structure would come from the learning theories, we investigate another approach originated from the research area of componentware. One possible approach is to use a notion of "patterns" [2]. A pattern in the case of IPs means a common composition structure that works as a prototype IP in synthesizing a more complex IP. Although the usages of patterns is a promising approach, it is not yet sufficient to present a framework in which we can answer who provides patterns and in what ways they can be constructed. The latter would be a kind of learning meta-knolwedge.

For these reasons, we investigate in this paper the third approach based on Analogical Logic Program Synthesis (ALPS for short) [4, 5]. In ALPS, given a source program and an example specification of a target program in inquiry, the task is to find some program such that it is similar to the source and is consistent with the examples. The source program is expected to provide a structure of target at least partially. To this aim, ALPS is required to search for a partial mapping, called a substitution preserving mapping, so that the transformed structure can be extended to some program consistent with the examples. Moreover, negative examples are used to reject inappropriate mapping, where a mapping is called inappropriate if the program space defined by the mapping has no solution consistent with both positive and negative examples.

The situation is exactly similar to the case of IPs. That is, given a source IP, and an example specification of a target composite IP, the task is to transform the connection structure so that the transformed connection can be extended to some composite IP that achieves the behavior exemplified by the specification. We can say that the source IPs are themselves interpreted as patterns. We thus need not prepare a knowledge base of patterns.

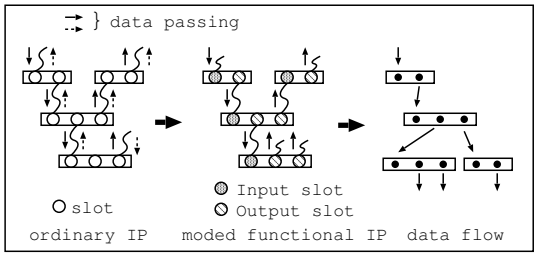
Since a slot connection structure of IPs is represented by a linearly covering program clause, and since ALPS is originally developed for that class of linearly covering programs, it is possible to directly apply ALPS for our purpose. However the task is to find just one valid clause defining the composition structure, while ALPS must search for a logic program, a set of clauses. Furthermore, it is needed to specify input/output behavior of primitive IPs that is not generally described in terms of linearly covering clauses. For these reasons, we present in this paper a new "ALPS for IP", that is specific to our problem of synthesizing and learning composition structure from example specifications.

This paper is organized as follows. In Section 2, we informally describe composition structures of IPs and define them in terms of logic programs. In Section 3, we introduce a notion of similarities between IPs, that is a structure-embedding mapping between clauses. For every notion concerning the synthesis problem of IPs is thus defined in terms of logic programs, we can design a synthesis algorithm apart from the viewpoint of IPs. In fact we present it in Section 4.

2 Moded IP and CIP-clauses

In this section, we show that the problem of synthesizing IPs can be considered as one of inferring logic programs. To realize it, we must describe composite IPs in terms of logic programs. As we have already mentioned in the introduction, a composite IP consists of several primitive IPs with a slot connection structure. Each IP has internal methods reacting on the slot values. Although they can realize relations on slot values, we restrict them to those defining functions, not relations. For this purpose, we designate an input/output

mode to each slot in primitive IPs. Then we can transform the slot connection structure of composite IP into a *linearly covering program clause* (LCP-clause for short). Moreover, we describe in this section our hypothesis space of the transformed LCP-clauses. The inference algorithm, presented in the next section, is designed to search for a valid clause in it.



2.1 Moded functional IP

First we describe the notion of moded functional IPs briefly, and give its definition in terms of logic programs. A moded composite IP is a collection of primitive IPs interlinked each other by slot connection. Each primitive IP has several slots to which we assign input/output mode declaration. Once we designate an input/output mode to each slot, then a data flow in the IP is determined uniquely. The introduction of mode is natural because we often assume such a specific flow when we synthesize a composite IP. In the IntelligentPad system, any connection structure in a composite IP is required to be a tree (connected and acyclic undirected graph of primitive IPs). So the moded IP is also an acyclic directed graph, whenever we represent it as a graph.

Moreover, we assume that every slot is not connected to more than two slots. This means that any output slot is connected to another input slot at most once. So the slot value of the former is never duplicated and passed to more than two IPs. The assumption becomes a key to represent IPs as LCP-clauses, as we see later.

We first define primitive IPs. A primitive IP has several slots which have their own predefined data types.

Definition 1. A primitive IP p has input and output slots $\{si_1, ..., si_m\}$ and $\{so_1, ..., so_n\}$, respectively. For each $si_j (1 \leq j \leq m)$ and $so_k (1 \leq k \leq n)$, τi_j and τo_k are their predefined data types, respectively. A primitive IP p is then encoded as an atomic formula $a_p(V_{si_1}, ..., V_{si_m}; V_{so_1}, ..., V_{so_n})$, where V_{si_j} is an input variable corresponding to the input slot si_j with τi_j, and V_{so_k} is an output variable corresponding to the output slot so_k with τo_k.

The input/output behavior of primitive IP is assumed to be presented by a set of ground instances. Moreover they are required to be "functional" in the sense that, for any constant symbols v_{si_1},...,v_{si_m}, there exists exactly one tuple $(v_{so_1}$,...,$v_{so_n})$ such that $a_p(v_{si_1}$,...,$v_{si_m}; v_{so_1}$,...,$v_{so_n})$ holds in the set of instances.

We secondly define a composite IP C_p consisting of primitive IPs $\{P_j\}$ that are linked together by slot connections. The connection of two primitive IPs P_i and P_j is realized by associating slots of P_i with slots P_j under some data type constraints. For we suppose input/output mode for each slot, input slot of P_i is required to be connected to output slot of P_j. Similarly output slots are associated with input slots.

Once such a slot connection is established, we can classify every slot in the composite IP C_p into the following two kinds:

External slot: Every slot in a P_j , which is not yet connected, is regarded external. They are also used to achieve a connection with a new another IP.

Internal slot: Any slot already connected in C_p is understood as an internal one.

The basic strategy to represent the connections in terms of logic programs is very sim-

ple. Suppose we have two primitive IPs P_i, P_j, input slot s_i of P_i and output slot s_j of P_j. Suppose furthermore these two slots are not yet connected. Once we put them together by connection, then the slot value of the former is passed to the latter. As a result, they share the same data value. The mechanism is easily realized by unifying the corresponding variables, as we represent each slot in a P_i by a logical variable (Definition 1). Moreover, we already assume that any slot is connected to another at most once, the unification for the variable is preformed at most once. Consequently just two occurrences of variables exactly accord with the slot connection. The other variable with a single occurrence is concluded as external. In the definition below, the internal slots are encoded as local variables in a definite clause.

Definition 2. (CIP-clause) A composite IP C_p consisting of primitive IPs $p_1, ..., p_l$ is represented as a definite clause

$$C_p(I_1, ..., I_m; O_1, ..., O_n) :- \\ a_{p_1}(I_1^{(1)}, ..., I_{m^{(1)}}^{(1)}; O_1^{(1)}, ..., O_{n^{(1)}}^{(1)}), \\ ..., \\ a_{p_k}(I_1^{(k)}, ..., I_{m^{(k)}}^{(k)}; O_1^{(k)}, ..., O_{n^{(k)}}^{(k)}).$$

where

1. **Variable Condition:** The variables $I_j^{(i)}$ and $O_j^{(i)}$ occur either once or twice in the body part.
2. **External Variables:** The variables $I_1, ..., I_m; O_1, ..., O_n$ are exactly those that appear in the body just once.
3. **Cycle-Freeness:** There exists no cycle $p_0, p_1, ..., p_n, p_0$ of primitive IPs, where p_k and p_j are defined to be adjacent if the corresponding atoms $a_{p_m}(\mathbf{I^{(m)}}; \mathbf{O^{(m)}})$ $(m = k, j)$ share a variable.
4. **Connectedness:** Any primitive IP p_k in the body has an adjacent IP p_m.

For instance, a CIP-clause $C_p(X;Y)$:- $a_{p_1}(X;Z)$, $a_{p_2}(Z;Y)$ represents a composite IP consisting of atomic IPs p_1 and p_2, where the shared variable Z denotes the connected slot.

Now we briefly explain that the CIP-clauses is a LCP-clause. For this purpose, a simpler form of LCP-clause is presented here. The original definition [3] is more complicated, for the class of programs considered in that paper is wider than ours. First we introduce a notion of blocks. A block is triple $(Body, Input, Output)$, where $Body$ is a sequence of moded atoms, and $Input$ and $Output$ are sets of input and output variables, respectively. Intuitively speaking, the $Body$ is an acyclic flowchart of procedure calls, where the input and output resources are designated by the $Input$ and $Ouput$ of variables, respectively. Each procedure call is specified by an atomic predicate $p(X_1, ..., X_n; Y_1, ..., Y_k)$ with the local input and output resources $\{X_1, ..., X_n\}$ $\{Y_1, ..., Y_k\}$, respectively. These local resources are added by a set union when we compose two procedure calls into a body in parallel. Serial construction of procedure calls is also possible. In some cases we need to pass variables without performing any operations. The empty block meaning "no procedure call" serves for this purpose. The body of procedure calls thus organized becomes a block. Formally we have the following definition:

Definition 3. A block (G, s, t) is inductively defined as follows:

atomic block: $(p(X_1, ..., X_n; Y_1, ..., Y_k),$ $\{X_1, ..., X_n\}$, $\{Y_1, ..., Y_k\})$ is a block, where all the variables are distinct.

parallel block: $((G_1, G_2), s_1 \cup s_2, t_1 \cup t_2)$ is a block, if (G_j, s_j, t_j) is a block for $j = 1, 2$, where we assume without loss of generality that the variable sets for two component blocks are disjoint.

serial blcok: $((G_1, G_2), s_1, t_2)$ is a block, if (G_j, s_j, t_j) is a block and $t_1 = s_2$.

empty block: (ϕ, s, t) is a block, if $t \subseteq s$.

For instance suppose we execute two procedure calls $p_1(X_1; Z, Y_1)$, $p_2(X_2, Z; Y_2)$ in this

order. The overall input is $\{X_1, X_2\}$. In the first procedure call represented by the atomic block $(p_1(X_1; Z, Y_1), \{X_1\}, \{Z, Y_1\})$, X_2 is simply passed to the second procedure call. So use the empty block $(\phi, \{X_2\}, \{X_2\})$. As a result, we apply the parallel block construction to obtain $(p_1(X_1; Z, Y_1), \{X_2, X_1\}, \{X_2, Z, Y_1\})$. Repeating similar arguments, $(p_2(X_2, Z; Y_2)$, $\{X_2, Z, Y_1\}$, $\{Y_2, Y_1\})$ is a block. Finally do the serial construction for these two blocks to get the overall block.

By the definition, we can easily verify that any resource (variable) is used only once as inputs of procedure calls. In this sense we use the term "linearly coveringness" in this paper.

Definition 4. (1) A moded clause $p(I_1, ..., I_n; O_1, ..., O_m) : -Body$ is called a LCP-clause if the $(Body, \{I_1, ..., I_n\}, \{O_1, ..., O_m\})$ is a block. (2) A moded functional composite IP is a logic program consisting of a CIP-clause in Definition 2 and unit clauses prescribed in Definition 1

Theorem 5. *(1) A CIP-clause is a LCP-clause. (2) A moded functional IP defines a function on the Herbrand universe of constant symbols.*

Proof and Remark: The proof is straightforward, since we can easily construct the corresponding block from a CIP-clause whose body describes an acyclic flowchart of primitive IPs defined as atomic blocks. On the other hand, the unit clauses denoting atomic IPs are not LCP-clauses in the sense of original definition in [3]. We apply the notion of linearly covering clauses only to the slot connection structure represented by a CIP-clause.

It should be noted here that the space of possible composite IPs is represented by the set $\mathcal{LI}$ of CIP-clauses: $\mathcal{LI}$ = {a CIP clause $C : (H : -B_s)$}, which will be our hypothesis space. This is because we assume that every ground instance of primitive IPs is provided as a fact to the learner and because our moded functional IP consists of just one CIP clause and a set of ground instances of atomic IPs.

Finally we list the notation used in the following sections. First $v : \tau, L$ and Ls represent a variable v of type τ, a literal and a set of literals, respectively.

$V_{in}(L) = \{v : \tau |\ v$ is an input variable in $L\ \}$,
$V_{out}(L) = \{v : \tau | v$ is an output variable in $L\}$,
$V_i(Ls) = \{v : \tau | v$ is an input variable which appears only once in a set $Ls\}$,
$V_o(Ls) = \{v : \tau | v$ is an output variable which appears only once in a set $Ls\}$,
$V(Ls)$: the set of variables of a set Ls.

3 Analogical reuse of composite IPs

In the previous section, we have shown that connection structures of moded composite IPs are represented by CIP-clauses. It is therefore possible to apply the techniques of Inductive Inference or of Inductive Logic Programing to obtain a method for synthesizing the CIP-clauses (or equivalently the slot connection structure). However in this paper, we examine a possibility to reuse an existing moded functional IP to guide an appropriate connection structure of a target moded IP.

The basic strategy is to use *Analogical Logic Program Synthesis from examples*(ALPS for short) [4, 5]. This is because ALPS involves a notion of similarities between program clauses based on which analogical transformation of source program is carried out to construct a target program. Since the slot connection structure of composite moded IP is represented by a CIP-clause, we define in this section a notion of similarities between CIP-clauses, restricting the original notion presented in ALPS from the viewpoint of reusing slot connection structures.

Although we can consider various kinds of similarity concepts for composite IPs, we focus

our consideration on embedding a slot connection structure of a source IP into the target IP. In other words, the source IP is required to provide a connection structure only, which is captured logically as we have seen in Section 2. The embedded structure will be extended to have overall connection structure of target IP that achieves an intended behavior.

Another aspect of source IP, such as semantic combination of primitive IPs, is neglected in this paper, and is left as a future work. In the actual IntelligentPad system, each primitive IP is an instance pf a class of IPs. Since all the instances inherit every property and method the class has, the semantic information on classes becomes very important in combining IPs from a viewpoint of object-oriented programming.

Now we illustrate how a connection structure is embedded into another IP. Suppose we have two moded composite IPs whose connection structure are represented by the following blocks:

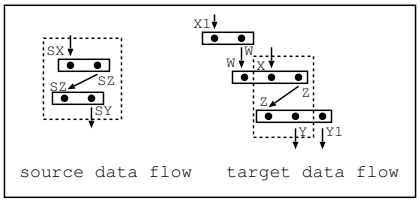
Source B_S: $((sp_1(SX;SZ), sp_2(SZ;SY)),$
$\{SX\}, \{SY\})$

Target B_T:
$((p_1(X_1;W), p_2(W,X;Z), p_3(Z;Y,Y_1)),$
$\{X_1, X\}, \{Y, Y_1\})$

Although B_S is simpler than B_T, data flow in the source starting with the input SX, sending the internal SZ from sp_1 to sp_2 and finally producing SY as a whole output is preserved in the target. In fact we can have the following two possible corresponding flows in B_T:

Flow1 in B_T: p_2 receives X, sends internal Z to p_3. p_3 produces Y (or Y_1) based on Z.
Flow2 in B_T: p_1 receives X_1, sends internal W to p_2. p_2 produces Z via W.

In the case of Flow1, sp_j should be paired with p_{j+1}. Moreover, SX, SY and SZ correspond to X, Y (or $Y1$) and Z, respectively. The embedding of data flow illustrated in the above is easily realized by θ-subsumption between the source block B_S and an "abstracted" target



block B_T, where the abstraction here means both an act of forgetting some arguments in primitive IPs in B_T and an injective correspondence of primitive IPs over B_T and B_S. A class of mappings called Substitution Preserving Mappings (SPM), originally introduced in the study of ALPS, is enough to describe the abstraction. In the above example, the embedding of flow in B_S into Flow1, for instance, is realized by the mapping $\varphi(p_2(W,X;Z)) = sp1(X;Z)$, $\varphi(p_3(Z;Y,Y_1)) = sp2(Z;Y)$ and a θ-subsumption given by a substitution $\theta = \{SX/X, SZ/Z, SY/Y\}$. The predicate p_1 in B_T needs not be considered in the case of this embedding. So the mapping for p_1 may not be defined and is denoted by $\varphi(p_1(X_1;W)) = \perp$.

Formally we have the following definitions. First $Atom_S$ and $Atom_T$ denote a set of atoms representing primitive IPs that can be used in forming source IP and target IP, respectively.

Definition 6. A mapping $\varphi : Atom_T \rightarrow Atom_S$ is called a SPM mapping if

$\varphi(q(X_1,...,X_m;Y_1,...,Y_n))$
$= p(X_{k_1},...,X_{k_o};Y_{l_1},...,Y_{l_p})$ or
$\varphi(q(X_1,...,X_m;Y_1,...Y_n)) = \perp$,
where $\{k_1,...,k_o\} \subseteq \{1,...,m\}$,
$\{l_1,...,l_p\} \subseteq \{1,...,n\}$, and
$X_1,...,X_m$, $Y_1,...,Y_n$ are typed variables.
Moreover If
$\varphi(q_1(X_1,...,X_g;Y_1,...,Y_i))$
$= \varphi(q_2(Z_1,...,Z_h;W_1,...,W_j))$,
then $q_1 = q_2$, $g = h$, $i = j$.

$\varphi(e) = \perp$ means that φ is not defined for e. We extend φ for any set S of atoms as follows.

$\varphi(S) = \{\varphi(A) | A \in S \text{ and } \varphi(A) \neq \perp\}$

For instance, suppose we have a mapping φ:

$$\varphi(q_1(s_1;t_1,t_2)) = p_1(s_1;t_1)$$
$$\varphi(q_2(s_1;t_1)) = p_2(s_1;t_1)$$
$$\varphi(q_3(s_1;t_1)) = \bot$$

Then
we have $\varphi(\{q_1(A;C,E), q_2(C;B), q_3(E,F)\})$ $= \{p_1(A;C), p_2(C;B)\}$.

An embedding of flows is now formally defined in terms of SPM and θ-subsumption:

Definition 7. Let $P : (p_S(\mathbf{I_S};\mathbf{O_S}) : -Body_S)$ and $Q : (p_T(\mathbf{I_T};\mathbf{O_T}) : -Body_T)$ be CIP-clauses with the blocks $(Body_S, \mathbf{I_S};\mathbf{O_S})$ and $(Body_T, \mathbf{I_T};\mathbf{O_T})$, respectively. Suppose further that φ be a SPM. Then we say that Q is similar to P w.r.t. φ if there exists a substitution θ such that $Body_S\theta \subseteq \varphi(Body_T)$.

Note that for the input and output variables of blocks must appear in their bodies in the case of CIP-clauses, the definition also implies that φ can be extended to a mapping ψ such that $P\theta \subseteq \psi(Q)$.

Using the notion of similarities as the embedding, the our inference problem is described as follows:
Given: (1) a CIP-clause P (2) a data presentation of positive facts of target CIP-clause Q and primitive IPs in $Atom_T$ as well. (3) a class Φ of SPM mappings from $Atom_T$ to $Atom_S$.
Remark: For we suppose that CIP-clause defines a function, all the negative examples are automatically derived from the presentation of positive facts.
Find: CIP-clause $Q \in \mathcal{LI}$ and $\varphi \in \Phi$ such that Q is correct w.r.t. the data presentation and that Q is similar to P w.r.t. φ.

For a SPM φ, $\mathcal{LI}[P,\varphi]$ denotes the set of CIP-clauses that is similar to P w.r.t. φ and is in $\mathcal{LI}$. Given a source CIP-clause P, the system generates all the possible mappings φ_j. Therefore, the hypothesis space $\mathcal{LI}$ is divided into subspaces $\mathcal{LI}[P,\varphi_1]$, $\mathcal{LI}[P,\varphi_2]$, ...

Now we are ready to present our inference algorithm in the next section.

4 Synthesis Algorithm

We describe in this section our synthesis algorithm searching for a CIP clause in each subspace $\mathcal{LI}[P,\varphi_j]$, based on an enumeration of SPM mappings $\Phi = \{\varphi_j\}$. If the subspace does not contain a solution clause, then the space should be rejected and skipped. So we firstly consider the problem of "refuting" such an inappropriate subspace.

4.1 Refuting Inappropriate Similarities

If a subspace $\mathcal{LI}[P,\varphi]$ contains no clause which is correct w.r.t. a given data presentation, φ is said to be an *inappropriate similarity*. It is useless to examine the corresponding subspace $\mathcal{LI}[P,\varphi]$. A theoretical idea, known as refutability [6], helps us to prune off such meaningless hypothesis spaces.

Definition 8. A synthesis algorithm $\mathcal{A}$ for $\mathcal{C}$ of programs is said to *refutably infer* $\mathcal{C}$ if $\mathcal{A}$ satisfies the following condition: For any presentation of any intended model M, (1) if there exists a program in $\mathcal{C}$ correct w.r.t. M, $\mathcal{A}$ identifies a correct program in the limit, (2) otherwise $\mathcal{A}$ outputs 'refuted' and terminates. $\mathcal{C}$ is said to be *refutably inferable* if there exists a synthesis algorithm that refutably infers $\mathcal{C}$.

We need an algorithm that refutes $\mathcal{LI}[P,\varphi_i]$, whenever no solution is involved in that space. For this purpose, we restrict the body length of a target CIP-clause Q to ℓ, and present an algorithm that refutably infers $\mathcal{LI}_\ell[P,\varphi_i]$ defined as

$$\mathcal{LI}_\ell[P,\varphi_i] = \{\text{CIP-clause } Q = (H :- Body) \mid$$
$$Q \text{ is similar to } P \text{ w.r.t. } \varphi_i,$$
$$Body \text{ consists of at most } \ell \text{ literals. }\}$$

4.2 Searching and refuting in each subspace

The refutation mechanism is deeply concerned with a method by which we generate possible CIP-clauses in each subspace $\mathcal{LI}_\ell[P, \varphi]$. Roughly speaking, given a SPM mapping φ and a source CIP-clause $P = (p(\mathbf{In_P}; \mathbf{Out_P}) : -Body_P)$, we firstly compute a set of "minimal" block $(Body_Q, \mathbf{In_Q}, \mathbf{Out_Q})$ such that $\varphi(Body_Q) = Body_P\theta$. There may exist several such minimal blocks, since ways of forgetting some slots and of putting flows in correspondences are non-deterministic. The process of finding such blocks is realized by the function procedure $inv(e, V_f, \varphi)$ as shown below.

The next thing to do is to extend the minimal block so that it "grows" into a target block $(Body, \mathbf{In}, \mathbf{Out})$, where the data for checking the validness of blocks are given as ground instances of $q(\mathbf{In}; \mathbf{Out})$. q is the predicate denoting a target composite IP. The growing process needs various operations such as regarding a variable in $\mathbf{In_Q}$ as an overall input variable in $\mathbf{In}$, unifying a variable in $\mathbf{In_Q}$ with another output variable of another new primitive IP to be added to the body, and so on. All the growing operations are carried out by performing a refinement, originally introduced in the study of MIS [7]. It should be noted here that the refinement starts with the maximally general clause $(q(\mathbf{In}; \mathbf{Out}) : -Body_Q)$ corresponding to the minimal block $(Body_Q, \mathbf{In_Q}, \mathbf{Out_Q})$. The variable sets $\mathbf{In_Q}$ and $\mathbf{Out_Q}$ are renamed to $\mathbf{In}$ and $\mathbf{Out}$ of the target predicate q so that we can perform the refinement. Our hypothesis space is thus extended to cover non CIP-clauses and to obtain a valid CIP-clause at the end of refinement processes. The following definition and the procedure will make the point stated as in the above clear.

Definition 9. Suppose $q(\mathbf{In}; \mathbf{Out})$ is the target predicate denoting a desired composite IP. Then a moded clause $p(\mathbf{Input}; \mathbf{Output}) : -Body$ is called a CDF-clause if the $(Body, \mathbf{X}; \mathbf{Y})$ is a block. Moreover $\mathcal{LD}$ denotes the set of all CDF-clauses.

The next definition prescribes the minimal block denoted by $\varphi^{-1}(P)$.

Definition 10.
$\varphi^{-1}(P) = (q(\mathbf{I}; \mathbf{O}) : -Bodys)$, where $Bodys$ is given as $inv(body(P), V_f, \varphi)$.

Function $inv(e, V_f, \varphi)$
Input:
a set of atoms e ; a set of variables V_f
a SPM φ
Output: a set of atoms
begin
if $e = B_1, ..., B_m.$ **then**
forj=1 **to** m **do** $B'_j = inv(B_j, V_f, \varphi)$
return$(B'_1, ..., B'_m)$
else
begin
let $t = p(t_1, ..., t_n)$ be an atom
such that $\varphi(t) = e$
let $Y_1, ..., Y_n$ be distinct variables
in V_f,$V_f := V_f \setminus \{Y_1, ..., Y_n\}$
for $i = 1$ **to** n **do**
if $t_i \in V(e)$ **then** $s_i = t_i$ **else** $s_i = Y_i$
return$(p(s_1, ..., s_n))$
end
end.

We now define a refinement operator from $\mathcal{LD}$ to $2^{\mathcal{LD}}$ as follows.

Definition 11. A mapping $\rho : \mathcal{LD} \rightarrow 2^{\mathcal{LD}}$ is a refinement operator such that, for any $C, D \in \mathcal{LD}$, $D \in \rho(C)$ iff one of the following conditions hold:

- $D \simeq C\{X/Y\}$, where $X : \tau \in V_i(body(C))$, $Y : \tau \in V_i(head(C))$ or $X : \tau \in V_o(body(C))$, $Y : \tau \in V_o(head(C))$.
- $D = C \cup \{\neg B\}$, where $\exists v : \tau \in V_{in}(B) \cap V_o(body(C))$, $(V(B) \setminus \{v\} \cap V(body(C)) = \emptyset$ or $\exists v : \tau \in V_{out}(B) \cap V_i(body(C))$, $(V(B) \setminus \{v\} \cap V(body(C)) = \emptyset$.

For any definite clauses C and D, $C \preceq_\rho D$ means that D is generated from C using the refinement operator ρ repeatedly. $\rho^*[P,\varphi]$ denotes the set $\{C | \varphi^{-1}(P) \preceq_\rho C\}$. We show our refinement operator is sound and complete.

Theorem 12. $\mathcal{LI}[P,\varphi] = \rho^*[P,\varphi] \cap \mathcal{LI}$

Proof. Suppose that $D \in \rho^*[P,\varphi] \cap \mathcal{LI}$. From the definition of ρ, there exists $C = \varphi^{-1}(P)$ such that $C \preceq_\rho D$. Therefore $P = \varphi(C)$ holds. Moreover, from the definition of refinement operator, there exists a substitution μ such that $C\mu \subseteq D$ and $dom(\mu) \cap V(P) = \emptyset$, where $dom(\mu)$ is the domain of μ. Since $C_1 \subseteq C_2$ implies $\varphi(C_1) \subseteq \varphi(C_2)$, $\varphi(C\mu) = \varphi(C) \subseteq \varphi(D)$ holds. Therefore we have $P \subseteq \varphi(D)$ and $D \in \mathcal{LI}[P,\varphi]$.

Suppose that $D \in \mathcal{LI}[P,\varphi]$. Since $P \subseteq \varphi(D)$, $\varphi^{-1}(P)\mu \subseteq D$. From the definition of ρ, $D \in \rho^*[P,\varphi] \cap \mathcal{LI}$ holds.

The refinement operator has a property that we cannot refine any CIP-clause to obtain more special clauses. This means that CIP-clauses appear in the last point of specialization in the refinement processes.

Our algorithm searches each subspace repeatedly, and is presented by the following procedure.

Synthesis Algorithm
Input: a source clause P
  a recursively enumerable set $\{\varphi_j\}$ of SPMs
  a data presentation of a target IP
Output: a clause C
begin
  $T := \emptyset$; $F := \emptyset$
  repeat
    readfact(T, F)
  until $T \neq \emptyset$
  for $j = 1$ **to** N **do begin**
    $S := \{\varphi_j^{-1}(P)\}$
    repeat
      choose $C \in S$
      if $C \notin \mathcal{LI}_\ell$ **then**
        if $\exists A \in T$, $C \not\vdash A$ **then**
          $S := S \setminus \{C\}$ **else**
          $S := S \setminus \{C\} \cup \rho(C)$
        else begin
          while $\forall A \in T$, $C \vdash A$ and
          $\forall A \in F$, $C \not\vdash A$ **do**
          begin
            output(C) ; readfact(T, F)
          end
        $S := S \setminus \{C\}$ **end**
      until $S \neq \emptyset$; $j := j + 1$
  end
end.

Procedure readfact(T,F)
begin
  read the next fact $\langle \alpha, V \rangle$
  if $V =$true **then** $T := T \cup \{\alpha\}$
  else $F := F \cup \{\alpha\}$
end.

Our algorithm for $\mathcal{LI}_\ell[P,\varphi_i]$ can refute inappropriate similarities before reaching every CIP-clause. This is simply because $\mathcal{LI}_\ell$ is finite. A more important thing relating to the refutability comes from the fact that checking the validness of hypothetical clauses is possible not only for CIP-clauses but also for CDF-clauses. This means that, when we succeed in finding the incorrectness of a CDF-clause w.r.t. a given data, we can prune off all the CIP-clauses more specific than the CDF-clause.

5 Concluding Remarks

In this paper, we showed a possibility of applying theoretical results on an inductive inference research to a synthesis problem of IntelligentPad.

In the IntelligentPad system, a new composite IP is obtained utilizing several IPs that are provided beforehand or already constructed. Their slots are adequately connected to obtain a desirable behavior of the composite IP. Thus, reusability of IPs is a remarkable property in the IntelligentPad system.

Such a reusability of components helps to synthesize a logic program as theoretically and empirically investigated in the literature[4, 5]. An analogical logic program synthesis system, ALPS, synthesizes a new logic program (target program) that is similar to a given source program. That is, it is considered that ALPS synthesizes the target program by reusing the source program.

From these similar aspects, we expected that a synthesis problem of IPs could be formalized as a problem with which ALPS deals. For this purpose, we tried to transform the slot connection structure of an IP into a CIP-clause. Since the class of CIP-clauses is a restricted class of ones that ALPS can potentially manipulate, the hypothesis space for the target IP is drastically limited. Although our actual whole search space largely covers the hypothesis space, we can expect a drastic reduction of the whole search caused by the refutability of (sub)search spaces. As the result, the target IP would be found efficiently.

The study of this paper is the first step in applying inductive inference methods to solve a real synthesis problem in the IntelligentPad system. Therefore several interesting works still remain to be done. It is most important to show empirical results. First we should show an effectiveness of providing a source IP in order to synthesize a target IP. It corresponds to showing an importance of reusability of components. In practice, we compare our system reusing source IP with one without source IP in that how long CPU time it takes to synthesize a target IP. Second, we verify an effectiveness of our search reduction by the refutability of the subsearch spaces. The numbers of the pruned CIP-clauses and all CIP-clauses in the whole search space will be compared.

It would also be worth observing the difference of search methods. Our system adopts a MIS-like top-down search method. On the other hand, some ILP systems with bottom-up methods have been proposed. For example, GOLEM is well known as one of such systems. We compare our system with GOLEM for synthesis problems of IPs. As presented in this paper, our system has been tailored to synthesize IPs. To make a fair comparison, therefore, it would be desired to tailor the original GOLEM to effectively synthesize IPs. Such a system is currently under investigation and implementation.

Moreover, although this paper dealt with a slot connection structure that is linear, more complicated slot connections are allowed in the IntelligentPad system. It would be worth extending the current framework to deal with such complicated ones. It would also be interesting to characterize a good/bad source IP for synthesizing a target IP.

References

1. Y. Tanaka. A meme media architecture for fine-gain component software. 2nd International Symposium on Object Technologies for Advanced Software, Kanazawa, 1996.
2. R. Hirano. A method of supposing software development for a synthetic media architecture. Ms thesis, Hokkaido University, 1995 (In Japanese).
3. H. Arimura and T. Shinohara. Inductive inference of prolog programs with linear data dependency from positive data, Proc. Information Modelling and Knowledge Bases V, pp. 365-375, IOS press.
4. K. Sadohara and M. Haraguchi. Using abstraction schemata in inductive logic programming. 7th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence Vol. 1297, pages 256-263. Springer-Verlag, 1997.
5. K. Sadohara. A sturdy on analogical logic program synthesis from examples. PhD thesis, Tokyo Institute of Technology, 1996.
6. Y. Mukouchi and S. Arikawa. Towards a mathematical theory of machine discovery from facts. Theoretical Computer Science, 137:53-84, 1995.
7. E.Y. Shapiro. Inductive inference of theories from fact. Technical Report 192, Yale University Computer Science Dept., 1981.