Data Flow Analysis of IntelligentPad – A Basis of Reusing and Synthesizing Component Pads

Makoto HARAGUCHI[1], Atsushi HIRATA[1], Ken SADOHARA[2]
[1]Division of Electronics and Information Engineering, Hokkaido University
N-13, W-8, Kita-ku, Sapporo 060-0813, Japan
E-mail:{makoto, atsushi}@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 introduces a notion of data flows and mode declarations for IntelligentPad, a kind of component softwares. The mode specifies how component pads are used for testing the behavior of composite pads. Under the constraints obtained from a family of mode declarations, we present an expansion procedure to generate hypothetical composite pads. An experimental result shows that the usages of mode declarations can reduce the size of search space of composite pads admissible with respect to the mode declarations.

Introduction

The IntelligentPad system (Tanaka 1996) is a kind of component software, and has been desinged so that user can produce a complex software, called a composite pad, by combining primitive components, called primitive pads. Each primitive pad can be regarded as a collection of data items and procedures operating on them. To access the data stored in a pad from another pad, every pad is designed to have several slots, through which data items should pass from and to another pads.

For IntelligentPad is a kind of component softwares, it is usual or natural to consider a framework for reusing existing pads to build new pads based on the composition structures of the former pads. One possible approach is to use a notion of "patterns" (Hirano 1995). A pattern in the case of pads means a common composition structure that works as a prototype pad in synthesizing a more complex pad. 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 extracted from object bases of pads. The latter would be a kind of learning meta-knolwedge.

Another relevant studies can be found in (Sadohara 1997; A.Hirata 1998) in which a theoretical framework for analogical reuse of logic programs (representing pads) in synthesizing target programs is presented. Given a source composite pad with a fixed data flow, represented by a moded logic program, they use a top-down search method of Inductive Logic Programming (F.Bergadano 1995) to find a hypothetical composite pad consistent with a test data presentation. However the problem is that the search space of possible hypothetical pads is too huge to find an appropriate one in a realistic time and space. A key to solve the problem seems to regard composite pads as data flow generators and to make a constraint that various data flows should be realized in a single composition structure.

A composite pad can generates various data flows depending on input events user makes. Conversely an composite pad is designed and to be synthesized so that a single composition structure of pad can realize several data flows occurred in it. From this viewpoint, we introduce in this paper a notion of admissible flows of pads with respect to a mode declaration specifying how users make input events and observe output events on pads. Furthermore an algorithm that generates pads whose flows are admissible is presented, given an exsiting pad whose composition structure is reused to build the structure of expanded pads. Finally we present an experimental result based on which we discuss a research plan under developing. The result shows that using several flows can reduce the number of possible pads.

IntelligentPad

Before presenting a formal description of pad, we show a simple example. The example pad is supposed to calculate addition of three integers, and will be composed from a primitive pad, `ADD`, that executes binary addition. `ADD` has three slots. The first two slots, `Slot1` and `Slot2`, are used to store input integers, and are supposed to have internal methods performing the operation `Slot1+Slot2`. When a user makes an input event on `Slot1` or on `Slot2`, the method associated with the slot is then invoked, do the addition, and put the result in `Slot3`. To make an input/output events, Intel-

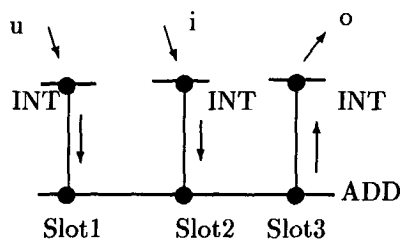


Figure 1: A composite pad for binary addition

ligentPad system provides us another primitive pad, **INT**, with just one slot **Int**. So the overall composition structure is shown by Figure 1, where the horizontal lines and the dotted circle denote pads and the slots in them, respectively.

The vertical lines in the figure show slot connections according to which data items in the slots are passed from and to another slots. For instance, suppose user puts his/her mouse on the leftmost **INT** and types an integer x. **INT** receives x as its **Int1** value, and sends an update request to any primitive pad connceted to **Int1**. Then **ADD** receives the update request with the value x to be stored in **Slot1**. When a pad receives an update request for its slot s, it stores the value in s, and invokes a method m^s, associated with the slot, for the value. In the case of **ADD**, the slot method m^{Slot1} takes x as its host object, uses **Slot2** as its parameter input, and output the result **Slot1+Slot2** in **Slot3**.

Now the content of **Slot3** is changed, so **ADD** sends another update request for any pad whose slot is connected to **Slot3**. Consequently the rightmost pad **INT** receives the request with the value of addition. For **Int1** slot has no method associated with it, and for the pad is not connected to other pad any more, the update process terminates at this pad.

The fundamental behavior of primitive pads, the update process, can be summarized as follows:

1. The initial update request is given by an input event users make.

2. A primitive pad, when receives an update request with a value, it stores the value in the corresponding slot s_c, and apply its slot method m if it is defined. s_c is called the update slot for the request.

3. The pad sends update requests to all the pads whose slots are connected to the output slots of m. If the update slot s_c is connceted to another pad p except the original pad sending the update request, another request is also sent to p to share the value of s_c. Consequently the update process does not "go back" to the original pad.

4. The update process terminates if there exists no connection through which update request is furthermore sent. Any composite pad should be organized as a tree of pads so that the update processes always terminate.

Now we formally introduce IntelligentPad (IP) system. A primitive pad (PIP for short) a can be represented as a frame structure of the form:

$$a[...; s_j[a^{s_j}@s^i_{j1}, .., s^i_{jn_j} \Rightarrow s^o_{j1}, .., s^o_{jm_j}]; ...],$$

where a is a pad ID, s_j is a slot, a^{s_j} is a procedure, called a slot method associated with s_j in a. Its host object is a slot value of s_j. $s^i_{j1}, .., s^i_{jn_j}$ are the list of input parameter slots for a^{s_j}, and $s^o_{j1}, .., s^o_{jm_j}$ is the list of output slots. We assume here that the set of input parameter slots, the set output slots and $\{s_j\}$ are disjoint. The number of slot methods is at most one, for each slot. A description of slot ID not followed by method specfication indicates that no method is defined. Among those slots, just one slot is designated as a primary slot, and is used to connect with another slot of another pad called a master pad. A slot s in a except primary slot is also connected with another slot of another pad. In this case, a works as a master pad of the latter (See the definition of composite IP below.). $ps(a)$ denotes the primary slot. In addition, we often use the notation $s \in a$ meaning that s is a slot of a. For instance, in a PIP $a[s_1; s_2[a^{s_2}@s_1 \Rightarrow s_3]; s_3]$, s_1 has no special method, s_2 has a method receiving s_2 value and changing the value of s_3 under the parameter value of s_1.

A composite pad (CIP) is a collection of PIPs interlinked each other by slot connections. A composite pad c is inductively defined as an undirected graph $(PIP(c), Con(c))$, where $PIP(c)$ and $Con(c)$ are sets of nodes (PIPs) and slot connections, respectively.

1.Let a be a PIP, then $(\{a\}, \phi)$ is a CIP.
2.Suppose c is a CIP $(PIP(c), Con(c))$, a new PIP $a \notin PIP(c)$, and $s \in b \in PIP(c)$. then $(PIP(c) \cup \{a\}, Con(c) \cup \{(ps(a), s)\})$ is a CIP, where $ps(a)$ is the primary slot of a. a and b are called a sub and a master pad, respectively. This sub/master relationship between PIPs forms CIP as an undirected tree (Figure 2).

Now we show an interpreter (Figure 3) to theoretically describe how a CIP behaves. It runs on the assumptions we have just made in defining PIPs and CIPs. First the interpreter is invoked by an input event that updates the value of slot in a PIP. By using slot methods and slot connections, the change of slot values

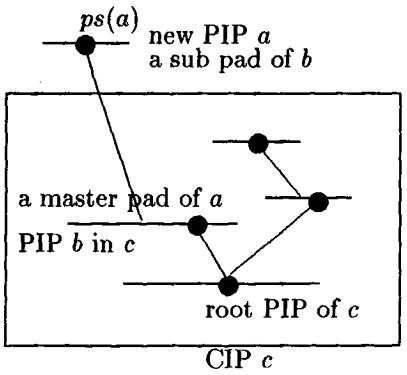


Figure 2: Composite pad in the form of tree

will update another slots of another pads. Such a process of updating slot values spreads over CIP through the slot connections.

Data Flow and its Block Representation

As we have discussed in Introduction, IntelligentPad is regarded as a data flow generator. In fact, given an input event that yields an update request (the variable **Update** in the procedure **update**), the process of updating slot values and visiting another pad via the connections can be viewed as a dataflow caused by the input event. This section introduces a notion of blocks that represent data flows formally.

In the definition below, each data flow is a series of data passing and procudre calls that can be organized serially. First we define an atomic block representing the behavior of PIP, and then introduce how we combine atomic blocks, by serial compostition, to obtain a composed block representing the behavior of CIP.

When a PIP a receives an updata request specified by (s, a, V), a performs the following three operations:

(1) a receives V as an input to the slot s,

(2) a passes V to another slot whenever it is connected to s.

(3) A slot method $a^s@s_1^i, ..., s_n^i \Rightarrow s_1^o, ..., s_m^o$ is invoked and produces the values $V_1^o, ..., V_m^o$ as outputs for $s_1^o, ..., s_m^o$, where the values $V_1^i, ..., V_n^i$ for $s_1^i, ..., s_n^i$ are supposed to be provided as parameter inputs.

As a result, PIP a, when updating $s \in a$, can be a procedure call with $1 + n$ inputs $V, V_1^1, ..., V_n^i$ for $s, s_1^i, ..., s_n^i$ and $1 + m$ outputs $V, V_1^o, ..., V_m^o$ for $s, s_1^o, ..., s_m^o$. We denote the procedure call as an atomic block $B_a^s = (p_a^s($ $s; s_1^i, ..., s_n^i; s_1^o, ..., s_m^i)$
$(V; V_1, ..., V_n; V_1, ..., V_m)$,
$\{s, s_1^i, ..., s_n^i\}, \{s, s_1^0, ..., s_m^0\})$,
where B_a^s obeys the following syntax of block in general:

```
procedure update(s,a,v)
Inputs: a CIP c = (PIP(c), Con(c)) ;
        an input event specified by (s, a, v)
begin
 Update := {(s, a, v)} ;
   /meaning that the content of s in a should be
     updated to the new value v./
 while Update ≠ φ do
  begin
   choose (s, a, v), mark s, and remove it from Update ;
    /the marking is used to prevent
     the update process from going back/
   set the value v in the slot s in a ;
   for each non-marked s' ∈ a'
     such that (s, s') ∈ Con(c) do
    add (s', a', v) to Update ;
     / if there exists no such a slot s', do nothing /
   if a^s@s_1^i, ..., s_n^i ⇒ s_1^o, ..., s_m^o is defined for s then
    begin
     apply a^s to the present value v
      with the parameter values that s_1^i, ..., s_n^i has
      to obtain new values v_1, ..., v_m for s_1^o, ..., s_m^o;
     set the value v_j to the slot s_j^o for each j ;
     for each j, add (s', a', v_j) to Update
       whenever (s', s_j^o) ∈ Con(c), where s' ∈ a'.
    end
  end
end
```

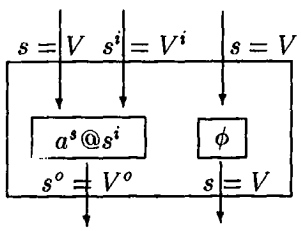
Figure 3: A theoretical interpreter of pads

block ::= (list of procedure calls,
input slot list, output slot list).

Each procedure call has the form of:

$p_a^s($ update slot s ; parameter slot list;
output slot list),
corresponding variable list).

Note that the value of slot in each procedure call is obtained by accessing the corresponding variable. Furthermore, if the slot method a^s is not defined for s, B_a^s is simply denoted as $(p_a^s(s;;)(V;;), \{s\}, \{s\})$.



The above figure shows an atomic block $B_a^s = (p_a^s(s; s^i; s^o)(V; V^i; V^O), \{s, s^i\}, \{s, s^o\})$ for a slot s. It represents both the invocation of slot method a^s and a data passing shown by an empty procedure ϕ that simply passes the data value.

Now we describe how we combine atomic blocks into a composed block. Suppose we have two blocks B and B' that have an output slot s and an input slot s',

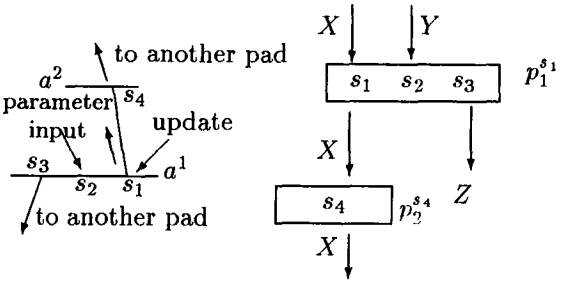
respectively. If If the slots s and s' are connected, the value of s, represented by a variable V, is forwarded to s'. As a result, the variable V' of s' and V share the same value. The act of sharing values is simply realized by unifying two variables.

Definition (Block Composition): Suppose we have two blocks $B = (B_1, I, O))$ and $B' = (B_2, I', O'))$ and $s \in O$ and $s' \in I'$ are given so as to be connected. Then the composed block is defined as

$$con(B, B', s, s') = \\ ((B_1, B_2\theta), I \cup I' - \{s'\}, O' \cup O),$$

where θ is a substitution unifying two variables corresponding to s and s'.

The following figure shows a data flow occurred in a composite pad and the corresponding block representation. We suppose a flow caused by updating the slot s_1 in the PIP a_1.



$$((p_1^{s_1}(s_1; s_2; s_3)(X; Y; Z), p_2^{s_4}(s_4; ;)(X; ;)), \\ \{s_1, s_2\}, \{s_1, s_3, s_4\})$$

It should be noted here that the slot s_1 is remained in the output slot list even after the composition. This is because we still have a possibility to make some PIP to be a subpad of a^1 by connecting to s_1. In this case, s_1 works as an output slot.

Now finally in this section, we procedurally define a dataflow caused by update request. In the procedure in Figure 4, the variable `Visited` denotes a set of primitive pads whose slot is marked in the procedure in Figure 3

From the definition of $DF(s, a, c)$, the block representing data flow caused by an update for s in a, its input slot list always contains s, the initial update slot, as one of its input slot. Moreover, parameter slots used in each slot method encountered in the data flow $DF(s, a, c)$ are added to the input slot list I of $DF(s, a, c) = (B, I, O)$. As a result, $I - \{s\}$ shows the set of all slots that are used as parameters.

Testing Composite Pads

User makes an input event by keyboard or by mouse operation on one of primitive pads in a composite pad.

```
begin
 let block variable B := B_a^s ;
 Visited := {a} ;
 while there exists a PIP a' ∉ Visited whose slot s' is
     connected to some output slot s_o ∈ O
        in B = (Body, I, O)
     do
      begin
        B := con(B, B_a'^s', s_o, s') ;
        Visited := {a'} ∪ Visited ;
      end
 return B
end
```

Figure 4: Definition of data flow $DF(s, a, c)$

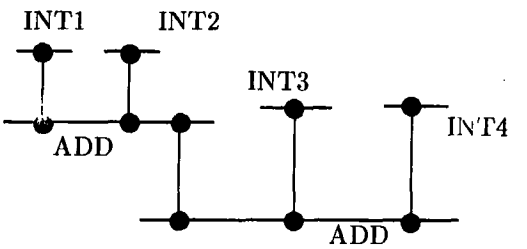


Figure 5: A composite pad calculating INT1+INT2+INT3=INT4

Then the data flow by the input event occurs, and the contents of slots are updated through the flow. When the update process terminates, he/she can check by his/her eyes to confirm that the desired result is shown on some particular pads he/she likes to check.

For instance, for a composite pad in Figure 5, using four `INT` pads and calculating `INT1+INT2+INT3 =INT4`, there exists a flow shown by Figure 6.

Thus the act of testing composite IPs starts from a single input event and is accomplised by a data flow caused by it.

The selection of slots on which user makes some input event can be changed according to each particular

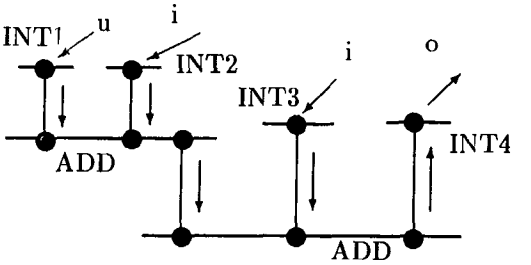


Figure 6: A possible data flow to obtain INT4, given INT1, INT2, INT3

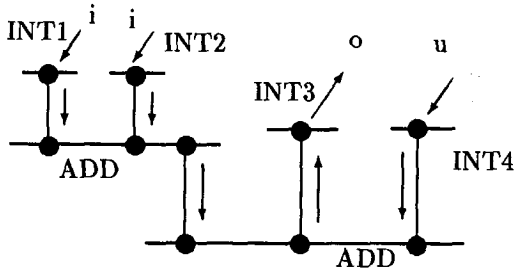


Figure 7: A Possible Data Flow to obtain INT3, given INT1, INT2, INT4

test. For instance, suppose we are checking if the composite pad computes substraction of integers. For this purpose, firstly update INT4, and invokes the method associated with the third slot `Int3` that calculates `Int2 = Int3 - Int1` among the slots in `INT`. The corresponding data flow is shown in Figure 7.

Although the composition structures are the same one, the usages of slots in testing composite pad may be changed. For instace, the primary slot of `INT1` is used for both "update" and "parameter input" slots. The slot in `INT3` is used as parameter input in the flow in Figure 6, while it works as an output in the flow in Figure 7

Here the notion of "mode" is introduced to specify how each slot is used in a possible flow.

Definition Mode Declaration (1) A composite pad c with input/output slots $\{s_1, ..., s_n\}$ is simply defined as a CIP c such that the slots s_j are declared to be used by user. c is now denoted as $c(s_1, ..., s_n)$.

(2) A mode declaration for a composite pad $c(s_1, ..., s_n)$ is a list of moded slots $s_j : m_j$, where m_j is one of following symbols:

"u": a slot updated by a user event.
"i": a slot served as a parameter input.
"o": a slot to be checked as an output.
"-": a slot of no concern,

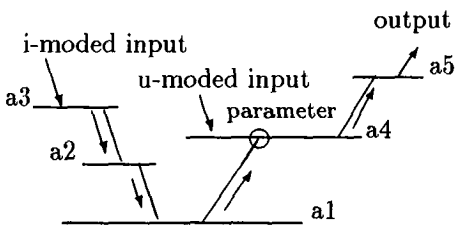
and "u"-moded slot is supposed to appear just once. A composite IP with the mode declaration is denoted as $c(s_1 : m_1, ..., s_n : m_n)$, and is simply called a moded CIP.

While updating each slot in a composite IP, several slot methods may use some parameter inputs. One way to handle parameter inputs encountered in the data flow is to assume their default values that are provided when the pads are newly created. Other way is to ask the user to specify their exact values as long as the parameter values are parts of his/her whole input.

In this paper we require that the user should be responsible even for parameters. This motivates the notion of coverability meaning that each parameter input in a data flow caused by an input event should be covered by another data flow triggered by another input event. We restrict our consideration to data flows that meet this notion. Similarly we assume that any output slot for which user shows his/her interests should be updated via data flow caused by input event.

For instance, the declaration $c(s_1 : u, s_2 : i, s_3 : o, s_4 : -)$ indicates the following procedure for our test. Before testing the behavior of c by updating the slot s_1 of "u"-mode, it is supposed to make an input event on $s_2 : i$ and to provide adequate values for parameter slots that are needed to execute slot methods in the data flow given by updating $s_1 : u$. In addition, the output slot s_3 is also expected to appear in the data flow from the input event on $s_1 : u$.

In the following figure, the u-moded input on $a4$ triggers off the flow from $a4$ to $a5$. This will be called a basic flow covering the output on $a5$. The slot method associated with the u-moded slot of $a4$ is using a parameter input. The another flow caused by an i-moded input on $a3$ reaches $a4$ via $a2$ and $a1$, and provides a necessary value to the parameter slot. The latter flow will be called a supplementary flow covering parameters.



A basic flow and a supplementary flow.

Definition (Coverability of parameter slots). Suppose $c(s_u : u, s_1 : m_1, ..., s_n : m_n)$ is a moded CIP, and let $DF(s_u, a, c) = (B, I, O)$ be the data flow caused by updating s_u in $a \in c$. Then c is said to satisfy Coverability Condition w.r.t. the mode declaration, if, for any slot $s \in I - \{s_u\}$, there exists an i-moded slot s_j such that $s \in O_j$, the output slot list defined by $DF(s_j, a_j, c) = (B_j, I_j, O_j)$. That is, the value of s, working as a parameter in the data flow caused by the u-moded slot s_u, is defined in the data flow caused by some i-moded slot s_j. In this context, $DF(s_u, a, c)$ and $DF(s_j, a_j, c)$ are called a basic flow and a supplementary flow, respectively. It could be a case that there still remains some parameter slots that are used in the supplementary flow. This happens when $I_j - \{s_j\} \neq \phi$. In such a case, apply the coverability condition recursively to the supplementary flows themselves.

Definition (Confirmation of output slots). Under the same precondition of Coverability, $c(s_u : u, s_1 :$

$m_1, ..., s_n : m_n)$ is said to satisfy Confirmation Condition, if any output slot $s_j : o$ is an element of O of $DF(s_u, a, c) = (B, I, O)$. That is, any output slots should be reached by the basic flow.

Definition (Admissibility):. A moded composite IP $c = c(s_u : u, s_1 : m_1, ..., s_n : m_n)$ is called admissible, if c satisfies both Coverability and Confirmation Conditions. In what follows, we consider only admissible CIPs w.r.t a given set of mode declarations.

Building Admissible Composite Pads

Based on the definition of admissibility, we now go on to the topic of generating possible admissible pads. This will be a basis of reusing and synthesizing CIPs. First we show such a method, given a single mode declaration, and then extends it to satisfy a set of several declarations.

Under just one mode declaration

The generation method presented here is desinged not to make unnecessary PIPs as well as linkages between them. Thus we make some additional conditions that meet our method.

Condition1: For any parameter slot $s \in I - \{su\}$ of $DF(s_u, a, c) = (B, I, O)$, there exists exactly one i-moded slot $s' \in a'$ that covers s. This is equivalent to asking user to select exactly one slot as an i-moded slot. If we have some parameter slots in $DF(s', a', c)$, then apply the assumption recursively.

Condition2: Any parameter slot $s^I \in a'$ of the basic flow or a supplementary flow, s^I appears as an update slot in the flow caused by some i-moded slot $s : i$. Intuitively speaking, the value of s^I is supplied by some slot connection, not by a method of another slot in the same primitive pad of s^I. This definition prohibits circular references between parameter inputs.

Now, given a mode declaration $c(...., m_j, ...)$, we start with an initial composite IP c_{top} in which u-moded slot s^u_{top} is already specified to determine the basic flow at least partially. The c_{top}, in the case of flow in Figure 6, can be the CIP in Figure 1. Observe that both the composition strucutre and the flows of the target CIP in Figure 6 can be extended from the structure and the flow of the c_{top}. In fact, by connecting the output slot, `Slot3` of `ADD` in Figure 1, with a slot `Slot1` of `ADD1`, newly created `ADD` pad, and by determining the slot in the middle `INT` in Figure 1 as an i-moded slot, we realize that both the flow and the composition structure are partially obtained. Thus the expansion procedure extend the composition strucutre so that initial data flow of c_{top} is extended.

The processes of connecting new PIPs to the present CIP and of determining which slots are i-moded or o-moded are described by the following expansion operators.

Determination of i/o-moded slots:
First we suppose the present basic flow $DF(s_u, a, c) = (B, I, O)$.

1. **Determining o-moded slot:** Simply choose a slot $s \in O - \{s_u\}$ whose pad appears as a leaf node of the present basic flow.

2. **Determining i-moded slot:** For a parameter input slot $s \in I - \{s_u\}$ of basic flow, when any supplementary flow is not yet formed, we can nondeterministically choose s itself as the i-moded slot. If some supplementary flow $DF(s', a', c')$ covering s is already formed, then choose s' as the i-moded slot to cover s.

Additing new pad at input side:
Let B^s_a and DF be a block representing a PIP and a present supplementary flow, respectively. Then replace DF with $con(B^s_a, DF, s_{out}, s_{in})$, where s_{out} and s_{in} is an output slot of B^s_a and an input slot of DF, respectively. If the s_{out} is not a primary slot, the pad a' containing s_{in} should be a root pad. If the s_{out} is a primary slot, there exists two ways to put a into DF: put it as a master pad under some conditions, or put it as a subpad. In both cases logical behavior of the composed pad are the same. So we assume that a is connected to DF as its subpad, without loss of generality.

Similarly, we define the introduction of new pads at output side so that we futhermore extend the basic flow.

Temination Condition:
We repeat applying the above two operators until all i-moded or o-moded slots are decided. An admissible CIP c w.r.t. a mode declaration $c(s_u : u, ..., s_j : m_j, ...)$ is synthesized after every i-mode or o-mode slot is decided. No more application of expansion operators for the admissible CIP is made. Conversely, it is proved that, for any mode declaraton $mode = c(s_u : u, s_1 : m_1, ...s_n : m_n)$ and a CIP c such that c is admissible w.r.t. $mode$ and that c is an extension of the initial CIP c_{top}, c can be obtained by applying two expansion opeators.

Under a set of several mode declarations

Suppose we have a family of k mode declarations $mode_i = c(m_{i,1}, ...m_{i,n})$ for $i = 1, ..., k$ $(k \leq n)$ and an initial CIP c_{top} in which a slot s^u_i is specified as a

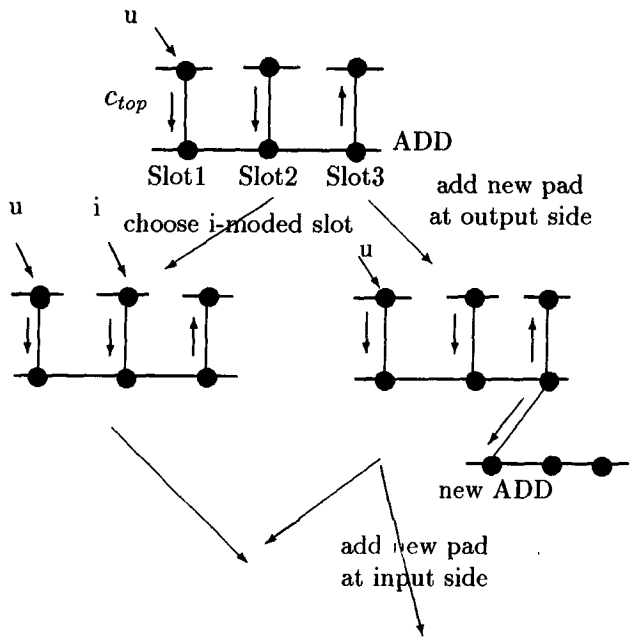


Figure 8: Expansion process

u-moded slot for each $mode_i$. s_i^u can work as i-moded or o-moded slot in another mode $mode_j$ $(i \neq j)$. For instance, for the two declaration: $mode_1 = c(u, i, -)$ and $mode_2 = c(i, o, u)$, the slot s_1^u should provide a supplementary flow of another flow caused by s_2^u. So we firstly check if s_i^u is an i-moded or an o-moded slot in another data flow caused by s_j^u. If the test fails, the c_{top} is rejected by the reason that there exists no extension that meets the mode declarations. If the test succeeds, then we begin to examine an extension of c_{top} by the exapnsion operators, and to determine a slot s_ℓ for each ℓ such that no $m_{i,\ell}$ (i= 1,...k) is "u".

The expansion operators are modified to take the relationship between several declarations into account.

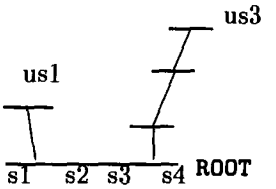
1. Apply the rule of determining o-moded or i-moded slot in one mode declaration $mode_i$ such that the determination satisfies the condition for every other mode declaration $mode_j$ $(j \neq i)$.

2. Apply the rule of adding new pad for some mode declaration $mode_j$. Note that the creating connce-tion to new PIP to extend one data flow caused by s_ℓ does not effect on another data flow under another $mode_i$, so the operation is safe in this sense.

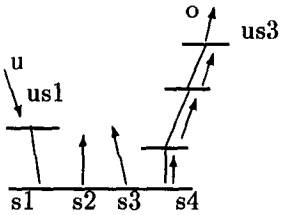
3. Termination Condition is the same as in the case of single declaration.

The number of possible composition strucutre is not reduced by the operation 2, adding new pads and connections. On the other hand, the determinations of user slots are constrained by the family of current data flows corresponding several declrations.

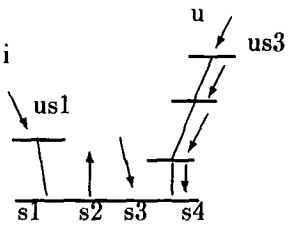
For instance, suppose we have two flows for the same composition structure in which some user slots are not yet specified.



The user slot **us2** is not yet determined.



A data flow $DF1$ under the declaration $m_1(us1 : u, us2 : o, us3 : o)$



A data flow $DF2$ under the declaration $m_2(us1 : i, us2 : i, us3 : u)$

For the root pad **ROOT**, we suppose two blocks treating slots as in the following table:

	s1	s2	s3	s4
in DF1	update parameter	output	output parameter	output
in DF2	input	output	input	update

Suppose further that we try to determine the user slot **us2**. For **us2** has the mode pattern $\begin{pmatrix} o & \text{in DF1} \\ i & \text{in DF2} \end{pmatrix}$, there exists just one slot, s3, that meets this pattern. The another slot, s2, does not satisfy it, so the possiblility is cut off.

An Experimental Result and Future Research

We have implemented a test algorithm for expanding CIPs to see how the mode declarations can reduce the number of possible CIPs that meet the declarations. Here we show a table, where we suppose three mode

declarations $m1 = c(i,o,u,o,i)$, $m2 = c(o,o,o,i,u)$ and $m3 = c(i,u,i,i,o)$. The table entry shows the number of expanded CIPs under the mode declarations, where the number of new pads added in the expansion process is limited up to N.

N	none	$\{m1\}$	$\{m1, m2\}$	$\{m1, m2, m3\}$
1	21	5	2	1
2	1765	138	28	14
3	162021	4427	521	257
4	-	167834	15268	6402

For the CIPs have various data flows according to mode declarations, we can do a partial test to check if the present CIP in the expansion process satisfies the test data or not. If the test fails, any admissible CIP that can be expanded from that CIP also fails the test. Thus we have a pruning method, given a test data presentation. Thus the number of possible CIPs is expected to be furthermore reduced under the test data.

From the viewpoint of reusing existing CIPs, it is unrealistic to suppose a very large N. This is because synthesis using a large number of pad creations might be faster than reusing inadequate existing CIPs. For this reason, we are now investing a system consisting of two components: One is a pad retrieval system that provides an initial CIP. The other one is a pad generation system, as described in this paper, for expanding the retrieved pad by adding relatively smaller number of primitive pads.

References

A.Hirata. 1998. On moded functional ip syntheses by reusing composition structures. In *Proc. Workshop on Applied Learning Theory, DOI Technical Report*, 59–63. kyushu Univ.

F.Bergadano. 1995. *Inductive logic programming, from Machin Learning to Software Engineering.* The MIT Press.

Hirano, R. 1995. A methodology of supporting software development for a synthetic media architecture. Master thesis (in japanese), Hokkaido University.

Sadohara, K. 1997. Using abstraction schemata in inductive logic programming. In *7th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence Vol. 1297*, 256–263. Springer-Verlag.

Tanaka, Y. 1996. A meme media architecture for fine-gain component software. In *2nd International Symposium on Object Technologies for Advanced Software.*