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

Makoto HARAGUCHI¹, Atsushi HIRATA¹, Ken SADOHARA²
¹Division 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
²Electrotechnical 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 designed 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-knowledge.

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 existing 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, Inte-
The IntelligentPad 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 Int value, and sends an update request to any primitive pad connected 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 outputs 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 update request with the value of addition. For INT 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 connected 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:

\[ a[...; s_j[a^s@s_{j1}, ..., s_{j_n} \Rightarrow s_{j1}, ..., s_{j_m}]; ...], \]

where a is a pad ID, s_j is a slot, a^s is a procedure, called a slot method associated with s_j in a. Its host object is a slot value of s_j. s_{j1}, ..., s_{j_n} are the list of input parameter slots for a^s, and s_{j1}, ..., s_{j_m} 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 specification 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[sl; s2[a^s@s1 \Rightarrow s3]; s8], s1 has no special method, s2 has a method receiving s2 value and changing the value of s3 under the parameter value of s1.

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)∪{a}, Con(c)∪{(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

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 procedure 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 composition, to obtain a composed block representing the behavior of CIP.

When a PIP $a$ receives an update 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 $a^{\ast}@s_{1}, ..., s_{n} \Rightarrow s_{1}^{\ast}, ..., s_{n}^{\ast}$ is invoked and produces the values $V_{1}^{\ast}, ..., V_{m}^{\ast}$ for $s_{1}^{\ast}, ..., s_{n}^{\ast}$, where the values $V_{1}^{\ast}, ..., V_{n}^{\ast}$ for $s_{1}, ..., s_{n}$ 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 $V, V_{1}^{\ast}, ..., V_{n}^{\ast}$ for $s, s_{1}, ..., s_{n}$, and $1 + m$ outputs $V_{1}^{\ast}, ..., V_{m}^{\ast}$ for $s, s_{1}^{\ast}, ..., s_{n}^{\ast}$ in a master pad of $a$. We denote the procedure call as an atomic block $B_{a}^{s} = (p_{a}^{s}(s; s_{1}, ..., s_{n}; s_{1}^{\ast}, ..., s_{n}^{\ast}), (V; V_{1}^{\ast}, ..., V_{n}^{\ast}; V_{1}^{\ast}, ..., V_{m}^{\ast}), \{s, s_{1}, ..., s_{n}, \{s, s_{1}^{\ast}, ..., s_{n}^{\ast}\})$, where $B_{a}^{s}$ obeys the following syntax of block in general:

Procedure update($s, a, V$)
Inputs: a CIP $c = (PIP(c), Com(c))$; 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 $\neq \emptyset$ do

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 $s^\prime \in a'$ such that $(s, s^\prime) \in Con(c)$ do

add $(s^\prime, a', V)$ to Update;

//if there exists no such a slot $s'$, do nothing.
if $a^{\ast}@s_{1}, ..., s_{n} \Rightarrow s_{1}^{\ast}, ..., s_{n}^{\ast}$ is defined for $s$ then

apply $a^{\ast}$ to the present value $V$ with the parameter values $s_{1}, ..., s_{n}$ has obtained;

set the value $V_{j}$ to the slot $s_{j}^{\ast}$ for each $j$;

for each $j$, add $(s^\prime, a', V_{j})$ to Update;

end

end

Each procedure call has the form of:

$p_{a}^{s}(s; s_{1}, ..., s_{n}; V_{1}, ..., V_{m})$, 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 $a^{\ast}$ is not defined for $s$, $B_{a}^{s}$ is simply denoted as $(p_{a}^{s}(s; s_{1}, ..., s_{n}; V_{1}, ..., V_{m}), \{s, s_{1}, ..., s_{n}\})$.

The above figure shows an atomic block $B_{a}^{s} = (p_{a}^{s}(s; s_{1}, ..., s_{n}; s_{1}^{\ast}, ..., s_{n}^{\ast}), (V; V_{1}^{\ast}, ..., V_{n}^{\ast}; V_{1}^{\ast}, ..., V_{m}^{\ast}), \{s, s_{1}, ..., s_{n}, \{s, s_{1}^{\ast}, ..., s_{n}^{\ast}\})$ for a slot $s$. It represents both the invocation of slot method $a^{\ast}$ and a data passing shown by an empty procedure $\phi$ 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 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

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

where $\theta$ 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 $a_1$.

![Figure 5: A composite pad calculating INT1+INT2+INT3=INT4](image)

Thus the act of testing composite IPs starts from a single input event and is accomplished 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

**Testing Composite Pads**

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

![Figure 6: A possible data flow to obtain INT4, given INT1, INT2, INT3](image)
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 subtraction 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 instance, 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 $a_4$ triggers off the flow from $a_4$ to $a_5$. This will be called a basic flow covering the output on $a_5$. The slot method associated with the u-moded slot of $a_4$ is using a parameter input. The another flow caused by an i-moded input on $a_3$ reaches $a_4$ via $a_2$ and $a_1$, and provides a necessary value to the parameter slot. The latter flow will be called a supplementary flow covering parameters.

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 $DF(s_j, a, c) = (B_j, I_j, O_j)$. 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 $DF(s_j, a, c)$ 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, $c(s_u : u, s_1 :$
so that initial data flow of expansion procedure extend the composition structure and the flow of the basic flow. 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 non-deterministically 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 \( s \).

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

Now, given a mode declaration \( c(\ldots, m_j, \ldots) \), we start with an initial composite IP \( c_{top} \) in which u-moded slot \( s'_{top} \) 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 structure 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, \( S1ot3 \) of ADD in Figure 1, with a slot \( S1ot1 \) 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 structure 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.

**Addding new pad at input side:**
Let \( B_u^1 \) and \( DF \) be a block representing a PIP and a present supplementary flow, respectively. Then replace \( DF \) with \( con(B_u^1, DF, s_{out}, s_{in}) \), where \( s_{out} \) and \( s_{in} \) is an output slot of \( B_u^1 \) 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.

**Termination 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 \( c(s_u : u, \ldots, s_j : m_j, \ldots) \) 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 declaration \( mode = c(s_u : u, s_1 : m_1, \ldots, s_n : m_n) \) 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 operators.

**Under a set of several mode declarations**

Suppose we have a family of \( k \) mode declarations \( mode_i = c(m_{i1}, \ldots, m_{in}) \) for \( i = 1, \ldots, k \) and an initial CIP \( c_{top} \) in which a slot \( s_i \) is specified as a
u-moded slot for each mode $j$. $s^u_j$ 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 $s^u_1$ should provide a supplementary flow of another flow caused by $s^u_2$. So we firstly check if $s^u_j$ is an i-moded or an o-moded slot in another data flow caused by $s^u_j$. 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 expansion operators, and to determine a slot $s^j_\ell$ for each $\ell$ such that no $m_{\ell, k}$ ($k = 1, \ldots, k$) 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 connection to new PIP to extend one data flow caused by $s^j_\ell$ does not affect 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 structure 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 declarations.

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

A data flow $DF_1$ under the declaration $m_1(\text{us1} : u, \text{us2} : o, \text{us3} : o)$

A data flow $DF_2$ under the declaration $m_2(\text{us1} : i, \text{us2} : i, \text{us3} : u)$

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

<table>
<thead>
<tr>
<th>in $DF_1$</th>
<th>update</th>
<th>output</th>
<th>output</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>in $DF_2$</th>
<th>input</th>
<th>output</th>
<th>parameter</th>
<th>input</th>
<th>update</th>
</tr>
</thead>
</table>

Suppose further that we try to determine the user slot $\text{us2}$. For $\text{us2}$ has the mode pattern $o$ in $DF_1$ and $i$ in $DF_2$, there exists just one slot, $\text{s3}$, that meets this pattern. The another slot, $\text{s2}$, does not satisfy it, so the possibility 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 \( m_1 = c(i,o,u,o,i) \), \( m_2 = c(o,o,o,i,u) \) and \( m_3 = c(i,u,i,i,o) \). 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 \( N \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>none</th>
<th>{( m_1 )}</th>
<th>{( m_1, m_2 )}</th>
<th>{( m_1, m_2, m_3 )}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1765</td>
<td>138</td>
<td>28</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>16201</td>
<td>4427</td>
<td>521</td>
<td>257</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>167834</td>
<td>15268</td>
<td>6402</td>
</tr>
</tbody>
</table>

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


