A Set and Mapping-based Detection and Solution Method for Structure Clash between Program Input and Output Data

Masaaki HASHIMOTO Katsumi OKAMOTO
ATR Communication Systems Research Laboratories
Sanpeidani, Inuidani, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

Abstract

This paper proposes a set and mapping-based detection and solution method for structure clash between program input and output data. Structure clash is one of the main concerns in JSP (Jackson's Structured Programming). Structure clash is a program implementation issue rather than a program specification issue. Furthermore, structure clash must be accurately detected and solved in order to produce an efficient program. Therefore, there is a very important nonprocedural language class in which a programmer need not think about the clashes, but in which a compiler detects and solves the clashes. In this class, an array-based detection and solution method was previously studied in the nonprocedural language MODEL. However, a set and mapping-based detection and solution method has not been studied, although sets and mappings appear in several very high level nonprocedural languages. An experimental compiler based on the method has been implemented for the entity-relationship model-based nonprocedural language PSDL. The experiment has proved that usable programs will be generated.

Introduction

Structure clash [1] between program input and output data can be defined as the asynchronisation of input and output timing of data which are simultaneously processed by the program. When a programmer of a procedural language, such as C, describes a program with clash, the programmer must detect this, and remove the clash by coding the program so that program intermediate data can be stored in tables, etc., in order to wait for the arrival of other data to be processed simultaneously.

Structure clashes are program implementation issues rather than program specification issues. Furthermore, structure clashes must be accurately detected and solved in order to produce an efficient program. Therefore, there is a very important nonprocedural language class in which a programmer need not think about the clashes, but in which a compiler detects and solves the clashes. For example, in MODEL [2], array variables are specified for all input, output and intermediate data. The MODEL compiler detects the clashes on the arrays by analyzing the array indices, and solves them by assigning tables to the arrays with the clashes. Scalar variables are assigned to the arrays without clashes.

In the above-mentioned language class, the authors have been studying the entity-relationship model [3]-based nonprocedural language PSDL (Program Specification Description Language) [4]. The main features of PSDL are sets and mappings quite different from the above-mentioned arrays, since an entity or relationship type is a set of entities or relationships respectively, and since a relationship type is also a mapping between entities. Although sets and mappings appear in several very high level nonprocedural languages, an automated clash detection and solution method based on them has not been used.

This paper proposes a clash detection and solution method based on sets and mappings. The authors have implemented the experimental PSDL compiler to which the method has been applied, and have obtained the result of the experiment. In the paper, the subproblems of the structure clashes which are called multi-threading clashes [1] are studied. Section 2 explains the structure clashes, especially a nonprocedural language classification from the viewpoint of clashes, and an integrated view of clashes. Section 3 outlines PSDL. Sections 4 and 5 describe the clash detection and solution method. Section 6 outlines the experiment. Section 7 discusses the method and the experiment result.

Structure Clash

Structure clashes are explained in the following:

(1) Structure clash example
An example is presented using a very simple accounting program. The program contains sale and product input files. The sales records contain the sale number, and the name and quantity of product sold. The product records include the product name and price. After inputting the records, the program computes each sale amount by multiplying the product price by the sale quantity, and outputs the account records which contain the sale number and sale amount to the account file. Let us assume that the input records are unsorted.

To compute each sale amount, the product price and sale quantity must be referred to in the product record and sale record both of which contain the same product name. However, the product and sale records which contain the same product name cannot be synchronously input, since the product and sale records are unsorted. Thus, a structure clash is detected.
If the number of product records is small enough to allow all the records to be included in a table defined in the program, the program can be coded by a procedural language, such as C, as follows: the product table is defined by an array variable, and the product input records are all stored in the table. Then, a sale record is input, and the product record which contains the same product name is extracted from the product table. The product price is referred to in the product record, and the sale amount is computed. After that, the account record which includes the sale amount is output. These are iterated for each sale record. Thus, the clash is solved by tables. It is also solved by random access files, data bases, or input record sorting.

(2) Nonprocedural language classification

From the standpoint of clashes, nonprocedural languages can be divided into the following three classes:

Class 1) A programmer must detect and solve the clashes as in the use of a procedural language. A compiler does nothing about the clashes.

Class 2) A programmer need not think about the clashes. A compiler need not detect the clashes. On the assumption that the data will always include clashes, the compiler solves the clashes by assigning tables, etc., to all data.

Class 3) A programmer need not think about the clashes as in class 2. However, a compiler detects the clashes. Then, the compiler solves the clashes by assigning tables, etc., only to the data for which the clashes are detected. To the data without clash, scalar variables are assigned.

For example, Basic Lucid [7] has only scalar variables. The language can not describe a program with clashes, since it is practically impossible to simultaneously store sufficient data in the scalar variables. Therefore, a programmer describes the language after confirming that the program has no clashes. Even in LISP, a programmer detects and solves the clashes for assigning atom, list and array variables accurately. These languages are included in class 1. End user language LOTUS 1-2-3 is an example in class 2. In class 3, MODEL is included. MODEL is outlined in the next item.

The above-mentioned classes 2 and 3 prove that the clashes are program implementation issues rather than program specification issues. Furthermore, the clashes must be accurately detected and solved for the following reason: the programs made by the languages in class 3 are more efficient than that in class 2, since the use of scalar variables makes more efficient use of program memory space and execution time than the use of tables. Therefore, the authors have designed PSDL in class 3.

(3) Array based-detection and solution method

In MODEL, array variables are specified for all program input, output and intermediate data. The equations, such as "ACCOUNT.AMOUNT(S) = IF SALE.PRODUCT-NAME(S) = PRODUCT.NAME(P) THEN SALE.QUANTITY(S) * PRODUCT.PRICE(P) ", are described on the array variables, where S and P are the array indices. Each array element can get a datum only once. Therefore, any of the equations can refer to any data by indicating the array names and their indices without reference to the data input and output timing.

In the compilation, the execution sequence of the equations is determined according to the data flow among the arrays. Procedure iteration loops for processing the individual array elements are generated one by one for each index described in each equation. If the serially executed equations have the same range of index values, they are placed in the same loop. Then, consider an array which appears only in the loop. Each element datum of the array is stored and referred to only in the same iteration cycle. Therefore, the timing for storing and referring to the same element datum is synchronous. Such an array has no clash, and a scalar variable is assigned to the array. Other arrays have clashes, and tables for storing all element data are assigned to the arrays. Thus, the clashes are detected and solved on the basis of arrays.

(4) Integrated view of structure clashes

In [1], three kinds of clashes: multi-threading, ordering and boundary, are individually described by giving examples. Therefore, the authors have been studying an integrated view of the clashes to construct a unified detection and solution method for the three kinds. The asynchronization in the clashes depends on three factors:

an object, such as a datum in an array element or a scalar variable, which carries a timing,
a combination of objects which are simultaneously processed by a computation, such as an equation based on arrays, and
a sequence of objects flow in the same type, such as the sequence of element data flow in the same array, which is first determined by program inputs. These factors are illustrated in Fig. 1 (1).
The combination point at which the sequence is not identical among the object flows is determined to be asynchronous as shown in Fig. 1 (2). There is a structure clash at that point. However, the combination point at which the sequence is identical is determined to be synchronous as illustrated in Fig. 1 (3). There is no clash at that point. It seems that multi-threading, ordering or boundary clashes concern the combination, sequence or object, respectively.

PSDL

In PSDL, program input and output data properties are described as a program specification which is composed of three layers:

An information layer describes the information structure which frames the information in the universe of discourse represented by the input and output data. Therefore, the layer concerns the semantics of the data.

A data layer specifies the input and output data structure, such as business forms. Therefore, the layer concerns the syntax of the data.

An access layer determines the access method of the input and output files.

The PSDL statements are outlined with reference to Figures 2 and 3. Fig. 2 presents a PSDL program specification.

Figure 2: PSDL Program specification.

In Fig. 2, the statements between the INFORMATION and DATA statements on lines 00 and 26 specify the information structure. It is illustrated at the top of Fig. 3.

(1) Entity type, attribute, primary key and entity number

Entities, such as "Television", "Sale No. 1" and "Mr. Tanaka", are represented in the input and output data as shown in Fig. 3. The entity types PRODUCT, SALE and CUSTOMER which include these entities are illustrated by the thick rectangular boxes. Their primary key and nonprimary key attributes are specified below the boxes.

Fig. 3 illustrates the specification. The program is the same as that used in the previous section, except that each sale amount is output with the total sale amount per customer.

Information layer

In Fig. 2, the statements between the INFORMATION and DATA statements on lines 00 and 26 specify the information structure. It is illustrated at the top of Fig. 3.

(1) Entity type, attribute, primary key and entity number

Entities, such as "Television", "Sale No. 1" and "Mr. Tanaka", are represented in the input and output data as shown in Fig. 3. The entity types PRODUCT, SALE and CUSTOMER which include these entities are illustrated by the thick rectangular boxes. Their primary key and nonprimary key attributes are specified below the boxes.

In Fig. 2, each entity type is described by the E (Entity) statement on line 01. Following this statement, each primary key attribute of the entity type is described by the K (Key) statement on line 03, and each nonprimary key attribute by the A (Attribute) statement on line 04. STR (STRING) on line 03 specifies the attribute value set of character strings, and NUM (NUMBER) on line 04 indicates that of numbers. The number of entities in an entity type is described by the EN (Entity Number) statement on line 02. "-50" specifies
that the number is less than 50.

(2) Relationship type, collection and relationship number

Relationships, such as “Televisions are sold in sale No. 1.” and “Mr. Tanaka buys in sale No. 1.”, are represented in the input and output data as shown by the dotted lines between the entities in Fig. 3. The relationship types SOLD and BUY which include these relationships are illustrated by the heavily outlined diamond-shaped boxes.

Each relationship type is described by the R (Relationship) statement on line 05 in Fig. 2. Following this statement, the entity types related to each other by the relationship type are described by the C (Collection) statements on lines 06 and 08. The entity types and their roles are specified in the form “role.entityType”. If the entity types are different, the roles may be dropped as shown in Fig. 2. Following a C statement indicating an entity type, the number of relationships related to one entity in the entity type is described by the RN (Relationship Number) statement on line 07. “M” (Many) on the line specifies that the number is more than zero. “1” on line 09 indicates that the number is exactly one.

(3) Attribute value Dependency constraint (AD)

This constraint is used to obtain the new values of a non-primary key attribute. As illustrated by the arrows in Fig. 3, the SALE AMOUNT is derived by multiplying the PRODUCT PRICE by the SALE QUANTITY, where the entities SALE and PRODUCT are related to each other by the relationship SOLD. Then, the CUSTOMER TOTAL is gained by summing up the SALE AMOUNTS, where the entities SALE are associated with the entity CUSTOMER by the relationships BUY.

In Fig. 2, following the A statement of nonprimary key attribute, the value of which is obtained by computation, an AD is described by the \( = \) (equal) statement on line 15. The attributes referred to by AD are described in the form “attribute” or “role1.relationshipType.role2.entityType.attribute”. The former indicates the attributes of the same entity whose attribute value is obtained, and the latter refers to the attributes of other entities via relationships. Role 1 is the role of the entity whose attribute value is obtained, and role 2 is the role of other entities. Only one relationship type can be described in an AD.

(4) Relationship existence Dependency constraint (RD)

This constraint is used to obtain the new relationships in a relationship type. An example is shown in the following, as no RD is presented in Fig. 2. If a PERSON SKILL possessed by the PERSON is equal to a SECTION SKILL which is suitable for the SECTION, the relationship BELONG_TO exists between the entities PERSON and SECTION. Otherwise, no relationship exists.

In the succession of R and C statements, the condition expression for specifying the relationship existence is described by the RC (Relationship existence Condition) statement.
The attributes referred to by RD are described in the form "role.entityType.attribute" (8).

Entity existence Dependency constraint (ED)
This constraint is used to obtain the new entities in an entity type. Since no ED is shown in Fig. 2, the following example is presented: If a customer total is not zero, the entity ACCOUNT related to the entity CUSTOMER exists. The relationship DEMAND also exists between the two entities. Otherwise, the entity and relationship do not exist. For the entity obtained, its primary key attribute values must be determined.

Therefore, the expression for obtaining the primary key attribute values is described by the = statement in the same manner as AD in the succession of each primary key attribute K statement of an entity type whose entities are acquired by computation. Whether the customer total is zero or not is described in the ON phrase of the = statement. Only one relationship type can be described in an ED.

Data and access layers

(1) Data layer
The data layer statements are described between the DATA and ACCESS statements on lines 26 and 61 in Fig. 2. This layer is illustrated in the middle of Fig. 3. The input and output data structures are hierarchically determined by the elementary, sequence group, iteration group, and selection group data types. Each of these is described by the %, Q (sequence), I (Iteration) and S (Selection) statements respectively. The % statements also specify the datum forms in the same manner as in C. For example, %12s specifies character strings of length 12. Following the I statements, the IX statements specify the indices. Each of the group data types mentioned above is composed of other data types. If the component data types are also group data types, they are indicated by G (Group) statements. The ON phrases on line 29 and 50 specify the End-Of-File condition and the entity SALE number as iteration termination conditions.

The input and output data express information. The information is framed by the entity and relationship types. Therefore, the input and output data represent the entities, their attribute values and relationships. These are specified by the following association constraints: In the case of the entity representation, the elementary data types are associated with the primary key attributes of the entity type by the = statement on line 32 in Fig. 2. With the non-primary key attribute value expression, an elementary data type is connected to the attribute by the = statement on line 34. For the relationship and entity representation, the elementary data types are associated with the primary key attributes of the entity types related to each other by the relationship type. These associations are described by the = statements on lines 40 and 45. Since there is no relationship without entities, the entities are also represented in the input and output data.

(2) Access layer
Following the ACCESS statement on line 61, each file is described by the D (Dataset) statement on line 62. In the statement, the file name, INPUT or OUTPUT usage and record length are specified. The association constraints between the access and data layers are described by indicating the data types such as PRODUCT_DATA.

As mentioned above, the input and output files relate in various ways to the information structure via the data structure. The computation methods are specified in the information structure. Therefore, the three layer description determines the logical relationship between the input and output data, that is, the program specification.

Clash Detection Method

A structure clash detection and solution method based on sets and mappings is described under the following restrictions: 1) The input and output files are all sequentially accessed, and the input records are assumed to be unsorted.

2) Each of the files contains only one file record data type. The I statements are restricted to the record iteration. No S statement is used.

3) Each record represents, at most, one entity and one relationship in the same type.

A directed graph is first constructed from a PSDL program specification. Then, the graph is locally and globally analyzed.

Directed graph

The construction of the directed graph and the synchronization in the graph are described.

(1) Directed graph construction
The vertices of the graph are collected according to the PSDL statements. As shown in Table 1, there are ten kinds of vertices, such as an entity type vertex, which are classified into structure and constraint vertices. The names of the ten kinds almost correspond to the PSDL statements except the following: an entity type vertex specifies not only the entity type, but also all its primary key attributes. Therefore, the attribute vertices are collected according to the nonprimary key attributes only. By the second and third restrictions mentioned above, the data type vertices are collected according to the iteration group data types only, and the information and data layer association constraints under the same iteration group data type are specified by one vertex.

The arcs are drawn between the constraint and structure vertices as illustrated in Table 1. In the information layer, each arc has the same direction in which entities, attribute values or relationships are referred to, or obtained by the constraints. In the data and access layers, each arc has the same direction in which the data flow from the input files, or flow into the output files. The graph in Fig. 4 is made from the program specification presented in Fig. 2. Hereupon, let us consider the vertices and arcs only in Fig. 4. In this paper, 1-way circuits are not discussed.

(2) Synchronization in directed graph
Entities, attribute values, relationships or data flow along each arc as above-mentioned objects which carry timings. Combinations of objects which are simultaneously processed arise at the vertices. However, the constraints in PSDL are formalized so that the sequence is identical among the object flows at their vertices. By the second and third restrictions mentioned above, the sequence is also identical at the data type vertices. The synchronization can be ignored at the dataset type vertices as each of the vertices has only one arc.

In this paper, the arcs are characterized as either synchronous or asynchronous. The sequence of object flow on an asynchronous arc is considered not to be identical with the sequence on any arcs linked to an entity type, nonprimary key attribute or relationship type vertex which is the same as the asynchronous arc. On the contrary, the sequences of the object flows on the synchronous arcs which are linked to the same vertex, are considered to be identical.

Local analysis

The asynchronous arcs are detected by locally analyzing the graph as follows:

1) Set union

If an entity or relationship type vertex has two or more inflow arcs, those inflow arcs are characterized as asynchronous. The entity type can have nonprimary key attributes. The inflow arcs of those nonprimary key attribute vertices are also characterized as asynchronous. In the case of entity types, the reason is as follows:

Consider such an entity type vertex. Although the entities possessing the same primary key attribute value are provided from different inflow arcs, the entities are the same. Therefore, the set union operation must be applied to the entity sets, each of which is obtained from one of the inflow arcs. For the operation, the primary key attribute values of the entity sets are compared. Since the input data are assumed to be unsorted, the entities having the same primary key attribute value are not necessarily provided synchronously from each inflow arc. Therefore, the primary key attribute value of the entities must be stored until other arrivals. After all the entities arrive, the set union operation is completed, and the attribute values of the entities can be referred to. Accordingly, each inflow arc is asynchronous with other adjacent arcs via the entity type and nonprimary key attribute vertices.

In Fig. 4, the arcs A1, A2 and A3 are determined to be asynchronous, since the entity type PRODUCT vertex has two inflow arcs.

2) Mapping cardinalities

An arc between an entity type or nonprimary key attribute vertex and an AD or ED vertex associates a relationship number described by the RN statement. If the number is more than two, or "M", the arc is characterized as asynchronous. In the case of RD, the relationship existence condition is applied to all the collections of entities taken one by one from each of the entity types, where these entity types are related to each other by the relationship type having the RD. Therefore, the same entity appears in many collections, and the relationship numbers are virtually assumed to be "M". The reason for the above-mentioned asynchronousization decision is described in the following:

1) Consider an arc along which an AD, ED or RD refers to attribute values, where the relationship number of the arc is more than two, or "M". Each of the attribute values can be referred to more than two times. Since the input data are assumed to be unsorted, the same attribute value need not necessarily be referred to in a series. Therefore, the attribute values must be stored for the random reference, and the arc is asynchronous with other adjacent arcs via the entity type or nonprimary key attribute vertex. In Fig. 4, the arc A4 is determined to be asynchronous, since the relationship number of the arc is "M".

2) Consider an arc along which attribute values are ob-
Figure 4: Directed graph.

tained by an AD, where the relationship number of the arc is more than two, or "M". In this case, an aggregation function, such as the sum total, is specified. The function needs the intermediate data, and a program is generated which includes the intermediate data in the variables assigned to individual entities. Each intermediate datum can be modified more than two times. Since the input data are assumed to be unsorted, the same intermediate datum is not necessarily modified in a series. Therefore, the intermediate data must be stored for random reference, and the arc is asynchronous with other adjacent arcs via the nonprimary key attribute vertex. In Fig. 4, the arc A5 is determined to be asynchronous, since the relationship number of the arc is "M".

3) Consider an arc along which entities are acquired by an ED, where the relationship number of the arc is more than two, or "M". In this case, the same entity can be obtained by the ED two or more times. Therefore, the set union operation must be applied to the entity type, and the arc is asynchronous with other adjacent arcs via the entity type vertex.

Global analysis

On the result provided by the local analysis, other asynchronous arcs are detected by globally analyzing the graph.

1) 2-way circuit analysis

Consider a 2-way circuit, such as Fig. 5 (1). If all the asynchronous arcs on the circuit have the same direction as illustrated in Fig. 5 (2), the constraint evaluation sequence on the circuit becomes inconsistent for the reason mentioned below. The evaluation can not proceed through a structure vertex having one or two of the asynchronous arcs until all the objects, such as entities, attribute values or relationships, flow into the vertex. This inconsistency is solved by changing one of the opposite direction synchronous arcs into an asynchronous arc as shown in Fig. 5 (3).

2) Global program optimization

The selection of the arcs to be changed is a matter of global program optimization. There can be several asynchronous arc sets which include all the asynchronous arcs detected by the local analysis, and which are consistent in the 2-way circuit analysis mentioned above. From within the group of the sets, the optimized solution is given as an asynchronous arc set which generates the most efficient program.

However, since the method to obtain the optimized solution has not been solved, the authors have used the following method in the experiment described later. All the candidate arcs to be changed are first detected by analyzing all the circuits. From within the group of arcs, the arc which is a candidate of the greatest number of the circuits is changed to asynchronous. These are repeated until no further candidate arc arises.

In Fig. 4, the arcs A6, A7, A8 and A9 are changed into asynchronous arcs, since the arc A5 is asynchronous.

Clash Solution Method

The structure clashes are solved by determining the data and procedure structure of the program as follows:

Data structure

The data structure is determined in the following order.

1) Vertex synchronization

In the graph, the entity type, nonprimary key attribute and relationship type vertices which have the asynchronous arcs are characterized as asynchronous. The vertices of entity types which have the asynchronous nonprimary key attribute vertices are also characterized as asynchronous. The dataset
type vertices are all determined to be asynchronous. Other vertices are characterized as synchronous.

(2) Array and scalar variable assignment
An array variable is assigned to each asynchronous entity type, nonprimary key attribute or relationship type vertex. A scalar variable is assigned to each synchronous entity type, nonprimary key attribute or relationship type vertex. The maximum number of elements in each array is determined by the EN and RN statements.

In Fig. 4, array variables are assigned to vertices PRODUCT, PRODUCT.PRICE, SALE, SALE.AMOUNT, BUY, CUSTOMER and CUSTOMER.TOTAL. A scalar variable is assigned to vertices SALE.QUANTITY and SOLD. Accordingly, the data structure of the example program is determined as shown in Fig. 6 (1).

Procedure structure
The procedure structure is determined in the following order.

(1) Graph division
The graph is divided into synchronous connected subgraphs as follows: The asynchronous vertices are placed on the boundary of the subgraphs, and the synchronous vertices are placed within each subgraph. The adjacent synchronous arcs via a vertex are placed within the same subgraph, since these arcs synchronize with each other. The asynchronous arcs adjacent to the synchronous arcs via the constraint vertices are also included in the same subgraph. All the object flows synchronize with each other in each subgraph. In Fig. 4, the graph is divided into three subgraphs S1, S2 and S3 which are enclosed with dotted lines.

(2) Procedure blocks
A procedure block is assigned to each subgraph. Clash detection by global analysis confirms the following. For two adjacent subgraphs, all the asynchronous vertices on the boundary between the subgraphs have the inflow arcs only in one subgraph, and the other subgraph has only outflow arcs from the vertices. This implies that there is an execution order between the blocks assigned to the subgraphs. Therefore, there is the partial execution order among all the blocks. The total execution order is obtained from the partial order.

(3) Procedures in a procedure block
A procedure is assigned to each constraint or structure vertex. In each subgraph, there is a partial execution order of the procedures, since the constraint and structure vertices are connected by arcs. The total execution order of the procedures is obtained from the partial order.

(4) Iteration loops in a procedure block
Each procedure block includes one or more procedure iteration loops. The constraint which is executed first in a subgraph determines the iteration loop control factor as follows:

1) If a data and access layer association constraint for inputs is executed first, a loop is generated for an input file. The loop iterates for individual records in the file.

2) If an AD is executed first, there is a relationship type asynchronous vertex which is described in the AD. Then, a loop is generated for the relationship type. The loop iterates for individual relationships in the relationship type.

3) If an RD is executed first, there are two or more entity type asynchronous vertices which are related to each other by the relationship type having the RD. Then, loops are generated one by one for each entity type. Each loop iterates for individual entities in each entity type.

4) If a data and access layer association constraint for outputs is executed first, a loop is generated for the entity type which is described in the ON phrase of the IX statement. The loop iterates for individual entities in the entity type.

Experiment
For proving the detection and solution method mentioned above, the experimental PSDL compiler which generates a C program from a PSDL program specification has been developed on the SUN machines. As there are several facets of the method to be studied in the future as mentioned in the
The compiler itself is described by PSDL for extensibility. The compiler is 16,000 lines in scale. However, a portion of it is described in C language. The skeleton which is used for generating C codes is 500 lines in scale.

Concerning the efficiency evaluation, the static line numbers of the C programs generated by the PSDL compiler have been compared with those of the C programs described by the programmers. For accuracy of evaluation, the static line numbers of the assembly programs into which the C programs have been translated by the C compiler have been compared with each other. The ratios of the static line numbers were less than three as shown in Table 2.

### Discussion

The clash detection and solution method and the experiment result mentioned above are discussed.

#### (1) Set and mapping properties

The set union application points and the mapping cardinalities are analyzed in the clash detection in section 4. Additionally, in section 5, an array variable is assigned to an entity or relationship type which is a set, and a graph is divided into connected subgraphs based on the connection through mappings. Thus, the method depends on the properties of sets and mappings which often appear in very high level nonprocedural languages, as they are suitable for representing concepts in a universe of discourse.

#### (2) Accuracy improvements

The method in this paper detects and solves the structure clashes more accurately than that in [8] for the following reasons:

1) Only the vertices in the directed graphs have the synchronization characteristics in [8]. On the contrary, in this paper, both the vertices and arcs have these characteristics. Therefore, it is now possible for the arcs connected with the asynchronous vertices to be determined as synchronous by the local and global analysis. It has also become possible for
the synchronous arcs connected with the same asynchronous vertex to be placed in the same connected subgraph.

2) The synchronization is globally analyzed using the groups of the directed paths, all of which flow from the same vertex, and into the same other vertex in [8]. The global analysis method using the 2-way circuits in this paper has a wider range of application than the previous one.

3) Additionally, the relationship type vertices and the relationship existence dependency constraint vertices have been introduced into the directed graphs. The vertices of the non-primary key attributes of the entity types whose vertices were asynchronous can be made synchronous.

The synchronization defined between two adjacent arcs will be studied for future improvements in the accuracy.

(3) Global program optimization
A problem of global program optimization has been clearly defined in this paper. However, the complete solution of the problem has not yet been obtained, and will be studied in the future.

(4) Efficiency of generated programs
The efficiency of programs generated by the PSDL compiler is described in the previous section. The skeleton made for the experiment can be refined for improving the efficiency. Therefore, the usable programs will be generated by solving the subjects described in this section.

(5) Formalization
The integrated view of structure clashes mentioned in section 2 will be formalized, and the structure clashes will be classified. Additionally, the clash detection and solution method in this paper, especially the method for the unsolved problems, such as 1-way circuits, ordering clashes, and boundary clashes, will be formalized.

Conclusion
The structure clash detection and solution method based on sets and mappings, such as set union and mapping cardinalities, has been described. The method is quite different from the method based on arrays, such as the value ranges of array indices, in MODEL. Sets and mappings are more abstract than arrays. This increases the complexity of the set and mapping-based method. However, the method is valuable, since sets and mappings often appear in very high level nonprocedural languages.

The structure clash detection and solution method in our previous study [8] was of the elementary kind. The current study has introduced such new devices as asynchronization analysis on arcs, 2-way circuit analysis, relationship type vertices and RD vertices. These improve clash detection accuracy. The experiment of the PSDL compiler to which the method has been applied has proved that usable programs will be generated.

Several restrictions, such as no 1-way circuits in the directed graph, unsorted input records, no random access files, etc., are imposed on the method presented in this paper. The authors will study an extension of the method which removes these restrictions.

Acknowledgements
The authors are grateful to K. Habara and K. Yamashita for their suggestions. The authors also wish to thank T. Takekaka and other colleagues in the ATR Communication Systems Research Laboratories for their helpful discussions, and the members of the Japan Information Processing Service Co., Ltd., for their help in the PSDL compiler implementation.

References