Joint NCC & IIS Bull., Comp. Science, 18 (2002), 81–101
© 2002 NCC Publisher

Cross-program data flow visualization

N. N. Filatkina

Maintenance and transformation of large program systems require means for comprehension of their data dependences. We consider facilities for data flow visualization implemented in the RescueWare® system[1] — a workbench for modernization of legacy systems. One of the key points in the process is the user-guided identification and extraction of business rules, which could be performed successfully only if a user can efficiently trace the data flow. A typical legacy system consists of a large number of comparatively small modules that extensively exchange data either directly or through files, screens, and databases, and hence the analysis cannot be restricted to a single module. On the other hand, it is practically impossible and useless to show the complete data flow graph of the system. We present some techniques of taming the data flow complexity.

1. Introduction

The paper presents the experience gained during the development and use of the system for modernization and reengineering of legacy software called RescueWare [19]. In this section we briefly describe the architecture and objectives of the system, as well as the basic source code navigation means it provides.

1.1. RescueWare

Among the main problems that were to be solved by the developers of RescueWare, the following should be noted:

- huge size of real-life systems: hundreds of megabytes and millions of lines of code are typical characteristics;
- incompleteness of the source code: some components may be unavailable for detailed analysis, e.g., when they exist only as binaries;
- a variety of different and often inexactly formalized dialects of the source languages;
- heterogeneity of legacy systems: it is insufficient to restrict consideration to program code without analysis of inter-program dependences emerging from read/write access to files, databases and screen forms. Therefore, not only the programs but also the definitions of screen

[1]RescueWare® is the registered trademark of Relativity Technologies, Inc.

forms and database schemes are subjects of analysis and transformation;

- multi-lingual systems: although initially RescueWare was designed for the COBOL programming language, gradually the spectrum of the source languages was extended to PL/I, Natural, Ideal, etc.

RescueWare provides the user with a graphical environment running on a workstation and consisting of a set of tools called *roadmaps* which implement separate steps of the underlying methodology. With respect to this methodology, the reengineering process is divided into three main phases:

1. Legacy Understanding: at this phase the inventory of the source code is followed by the automatic analysis, which reveals both the internal syntactic and semantic structure of programs and other components, and the relationships between them. The collected information is used for verification of the system completeness and irredundancy, as well as for various statistical reports based on program metrics. The visual analysis is supported by various graphical representations of the program system and by both intra- and inter-program navigation.
2. Knowledge Mining: one of the main actions performed at this phase is the so-called *Business Rule Extraction* (BRE). It is aimed at identification and separation of the business logic of an application. The extracted part of the code together with its complement is called a *slice*. Depending on the nature of the business logic and the subsequent purposes, RescueWare supports several categories of slices. Although the extraction of business rules is fully automatic, the identification of slices and specification of their parameters may require a thorough knowledge of the system and hence needs a proper support. Because of that the Knowledge Mining phase is essentially integrated with the Legacy Understanding phase.
3. Generation: this last phase ports the source code and/or generated slices to different platforms and/or different programming languages. The list of the RescueWare target languages includes Java, C/C++, Visual Basic, etc.

1.2. HyperCode

Although, as we have stated above, the analysis of a legacy system should take into account all types of its components, the main emphasis is on the analysis of the program code. On the other hand, since the programs could be written in different languages, we need a program model with the abstraction level sufficient for a uniform description of the internal program structure and relationships. Moreover, representation of a program for the purpose of visualization and navigation could be less detailed as compared to the complete annotated parse tree: for this sort of activity the user may not need to inquire into all the details of the expression structure or the semantics of the basic operations. Therefore, the representation should be derived in some way from the complete parse tree and not be overcrowded with irrelevant details. However, in certain circumstances the user may be interested in informational and control structure of the program detected by context dependent analysis rather than in pure hierarchy of constructs. The typical examples of such relationships are "declare-instance" relationship linking the declaration of some data item with all its instances in the program code, or "def-use" relationship connecting all program points, where some variable is assigned to the points where the value of the variable is used.

We consider a program as an aggregate of related constructs characterized by appropriate attributes. This is, in fact, very similar to the entity-relationship (ER) approach firstly proposed by Chen [4]. The model was adapted to visualization purposes in the following way:

1. A special relationship was introduced to realize nesting of constructs. Unlike all other relationships, this one is untyped and is treated in a special way. This relationship emulates the syntactical hierarchy of constructs.
2. Each construct is bound to the source text, i.e., its source file name and the coordinates of its beginning and end are specified.
3. The model is extended by introduction of a hierarchy of construct types similar to the concept of inheritance. It allows for defining abstract construct types (e.g., "statement") and deriving concrete construct types (e.g., "conditional" or "assignment").

The resulting superposition of the syntax tree and relationships over the textual representation is similar to hypertext, provided that appropriate navigation facilities are available. We have extended the usual hypertext concept with the following properties:

- Only the program constructs may serve as reference carriers.
- The references are typed, the only exception is the reference to the parent in the syntax tree.
- A construct may have several associated references and, in particular, several references of the same type.
- A reference carrier (as a text fragment) may have nested reference carriers.

Obviously, this model is abstract enough to match practically any programming language; in fact, any structured text-based document with cross-references may be represented in this way. Moreover, the model allows for natural description of heterogeneous multi-lingual systems. The generality is achieved by passing the model as an additional parameter specifying the meaning of visualized information. The model reflects the diversity of the analyses relevant to the current purposes; thus a new model and the corresponding data can assign new properties and navigation paths to the old program.

The subsystem that provides the model-based visualization of the source code in RescueWare is called HyperCode [1]. This system is organized as an extensible set of components, each representing a specific "view" of the program: from a usual textual representation and an abstract syntax tree to control and data flow graphs. All these "views" are synchronized in the sense that whenever the user selects a program construct in one "view", all the rest are automatically positioned to the same construct. It is not necessarily the same construct; for example, when a variable is selected in the procedural division of a program, the "view" reflecting the program data may be positioned to the variable definition.

2. Visualization of memory layout

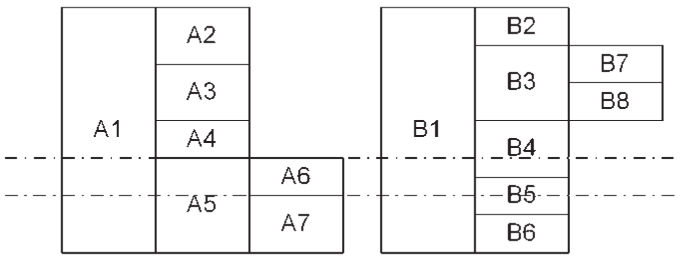


Figure 1. Memory overlay

The choice of data representation for the data flow analysis was influenced primarily by the fact that the source languages allow for addressing the same memory location via different structured data items. This is a well-known alias problem. In the languages like COBOL the problem is solved rather simply, since the aliases are explicitly indicated by REDEFINES clause and the sizes and offsets of data items can easily be calculated.

If two structures *A* and *B* share common memory (further on we will call this sort of dependence a *redefinition*), then modification of one structure implicitly modifies the state of another. Moreover, since both the size and the hierarchy of A and B may differ, the overlay of the structures may be the cause of the situation when modification of a field of A affects several substructures of B. Thus the dependences between the data items of A and B are defined according to the intersection of memory locations of these data items. For example, Figure 1 shows two structures *A1* and *B1* sharing the same memory, and modification of the field *A6* of *A1* leads to modification of the fields *B4* and *B5* of *B1*.

Taking into account the arbitrariness of structure nesting and possibility of partial data redefinition, we decided that it is not convenient to describe data flow in terms of syntactic constructs. Instead we came to a finer granularity: we consider all elementary memory locations emerging from intersection of all mutually redefining data items.

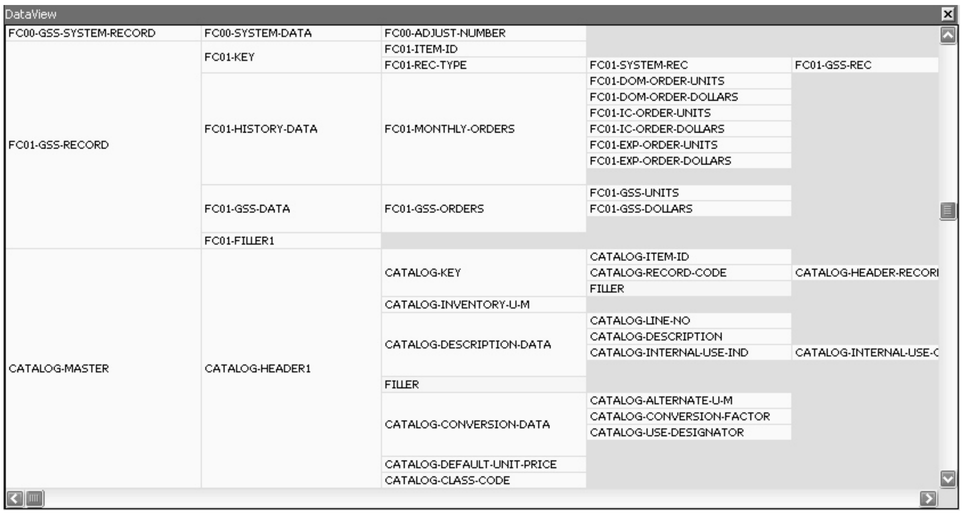


Figure 2. DataView

This approach allows for a more detailed graph of data dependences, but on the other hand it requires expressive visual methods for explication of the origin of a particular memory location, as well as of the corresponding data declarations and binding to the source code. The DataView component realizes a HyperCode "view" that displays the data layout diagram of the current program. The size of the data elements is not strictly proportional to their actual size, but reflects how various data elements overlap in the memory. As it is usual for HyperCode, selection of an item in the diagram automatically positions all other "views", and the source text in particular, to declaration of the corresponding data item. The reverse also holds:

selecting a declaration in the source text highlights its memory location in the DataView. A snapshot of DataView is shown in Figure 2.

3. Basic data flow dependences

The main sources for data flow are statements that are classified into two separate categories: *move* and *compute* statements. They differ in the nature of the dependence between the result and arguments of a statement. When complex structures appear on both sides of a statement,

- the *compute* relationship means that the value of the whole resulting data item, as well as the value of any of its parts, depends on the whole argument-structure.
- the *move* relationship means byte-by-byte copying of an argument to the memory result. Hence the rules for computation of dependences between the parts of the result and the argument are similar to the *redefines* relationship but, in contrast to it, *move* is directed.

A special class of relationships reflects assignment of constants, including initialization of data items. These assignments do not affect the overall structure of the data flow graph but are very helpful for better understanding, especially in the cases when constants are treated as the starting points of navigation.

4. Cross-program information

All relationships listed above are of intra-program nature. A special HyperCode construct type called *port* represents inter-program relationships, such as calls to different programs, sending and receiving data to and from screen forms, read/write operations for the database tables, etc. Each port is specified by its type and the references to data items through which the port communicates. We perform the global analysis that matches ports of different programs and thus collects inter-program dependences. This activity is very similar to usual linking. All port-to-port relationships have the same character as the *move* relationship.

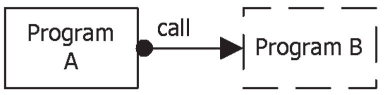
We distinguish the following types of ports with respect to data flow analysis:

1. *Call*: a point in the source code of a dynamic or static call to another program. The data are transferred via either the indicated parameters or a commonly used global memory area.

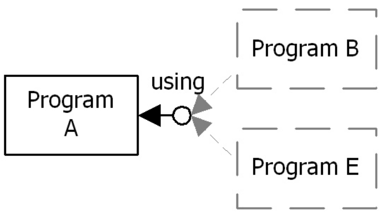
2. *Using*: an entry point to the program. It allows for specification of input external data, which are to be used in the program.
3. *Screen*: a point in the program where data are send to or received from a screen form.
4. *File*: a program point, which performs a read/write operation from/to a file specified as a logical file name.
5. *Database*: a program point which accesses a table in the project database.

These ports realize four different kinds of external communication between programs:

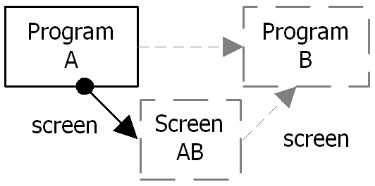
1. *Direct Connection*: the name of the destination program can be obtained from the source program code. A static call-port is an example of this connection.



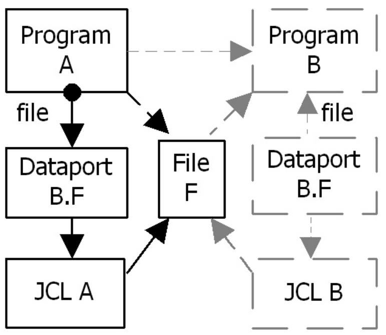
2. *Possible Connection*: the relationship with the program is indicated in the calling program on the opposite side of the relationship. For example, in the case of *Using* port, the relationship cannot be found in the program itself, but from the source code of programs that call the given one.



3. *Transit Connection* (through an external object): a program communicates via placing data onto an external object. The typical example is the usage of the screen forms: one program evaluates parameters and displays them on a screen form, while another program fetches the user input from the same form and processes it. This relationship similarly reflects *Database* ports.



4. *Semi-Dynamic Connection*: occurs in a more complicated case, when the relationship cannot be established on the basis of the source and destination program analysis; it emerges from the information kept in a separate object. For example, when a program writes to a file, it uses a logical file name, and even if two programs use the same logical file name, this does not necessarily imply that they write to the same file. The physical file name is associated with a logical file name for a given program in a separate job control file (JCL) which runs the program. Therefore, the data flow relationship can be established only when both programs and both JCL files are available for analysis. Note that the relationship may be changed without modification of the programs themselves.



The above classification of kinds of program communication deals only with static relationships. In practice, the relationships may become determined at run-time only and depend on calculation and ultimately on the user input. As a trivial example, a user might input the name of a screen to be displayed. When such a dynamic relationship is discovered during the source code parsing, the HyperCode information is supplemented with the so-called *decision points.*

There exist two ways to resolve decisions:

1. Automatic: is applied when the source code analysis may result in a finite set of values of an expression that denotes external objects, and all induced relationships are added.
2. Manual: allows a user to arbitrary indicate the realizable relationships. The solution is based on the user's expert knowledge. In particular, the user may indicate only those relationships that are relevant to the current tasks.

In any case, the problem of decision resolution is obviously algorithmically undecidable — any automatic method can guarantee only approximate results. But in practice, in the majority of cases, even a simple domain analysis leads to fairly good results.

The information obtained from decision resolution can be used to detect inter-program communication in addition to the static information. That applies to any of the four kinds of external communication described above. Note that accuracy may increase when missing relationships are found and decrease when a redundant or erroneous resolution is made.

5. Port matching

Even in the simplest case, the explication of inter-program relationships requires global knowledge on the current project state, availability of necessary programs, screens and control files linking logical and physical file names. Moreover, it is important to keep in mind that the global information may vary in time. For example, missing programs or control files may be registered in the project. Similarly, modification of a single legacy file does not yield complete recollection of the project-wide information. So, from the point of view of a visualizing tool, the inter-program information is an integral static piece of the project-wide graph of data flow dependences. The situation with external relationships is different, since they might emerge and disappear without direct modification of the source program code.

The global indexed information about inter-program relationships is kept in the so-called *repository.* Based on the repository information, we can find what objects are referenced from the program being considered. Knowing the names of referenced objects and the attachment of these objects to HyperCode ports, we can construct inter-program relationships for the purpose of data flow visualization. For example, suppose we know the name of a physical file which is written in a given program point. Then we can fetch from the repository the names of all programs reading from the same file. Further on, for each program we can find the corresponding logical name in the HyperCode database and, finally, the program points where the information from the original program point comes.

The procedure of detection of dependences between variables depends on the port type:

1. For a *Call* port, we know the name of the called program and so it is sufficient to check if the repository contains that program and, if so, find paired *Using* port in its HyperCode database.
2. For a *Using* port, we need to find all programs that have ports of *Call* type addressing the given program name.

3. In order to find the receiving programs in the case when information is passed through the *Screen* port, we need to collect complete information about screen forms that are registered in the project. Then for every screen form we find all programs that read from this screen and program points inside these programs with the corresponding screen form name.
4. The procedure for *Database* ports is similar to that for *Screen* ports, with the only difference that objects of DMSRecord type are used instead of screen forms.
5. For a *File* port, we have to look at the information associated with a relevant control file and to map the logical file name to the physical file name. Then from the same information we can find all other programs and respective logical file names that are used to access the same file, and finally use the HyperCode databases to find all paired *File* ports.

The design of the algorithm for reconstruction of inter-program relationships should take care of the following principle issues:

- The graph of cross-program dependences cannot be constructed statically once and for all, because its fragments can be partially modified in the process of project analysis.
- Dynamic reconstruction of dependences could involve a rather resource-consuming search over the repository and HyperCode databases for paired program points. Moreover, we might need to find the dependences of a particular data item many times and it is important to avoid repetitive reconstruction of the same relationship.

The solution that satisfies both of these issues is to incrementally update some internal table which keeps (partial) matching of ports. Whenever an object in the repository is modified, we can update only that part of the table which corresponds to the modified object.

Yet another optimization consists in construction of the matching table on demand: the analysis of a program is postponed until the moment when it is actually needed for users explorations. The program can be reached either directly, when the user selects a starting point for data flow analysis inside this program, or indirectly, when the data flow reaches a paired port inside the program.

So, with the help of information about heterogeneous system recorded in the repository, we reconstruct the relationships between parameters of ports in different programs and use these relationships in cross-program navigation.

6. Data dependences processing

Thus, the following information is collected by the source code analysis for the purpose of data flow visualization:

- Allocation of all data items: for each data item x the pair [offset$_x$:size$_x$] specifying the memory location is calculated.
- Table of relationships between data items: for each relationship its type (*compute* or *move*) and the reference to a HyperCode construct that caused the relationship is determined.
- Table of relationships for constant assignments and initializations.
- Table of ports: contains the name, data transfer direction (in, out, or bidirectional), and the references to communication data items for each port.

The tracing of cross-program data flows becomes possible because the visualization system includes the means for port matching as described in the previous section.

The data flow graph is constructed for a data item that was selected by a user as a starting point. The graph shows dependences of the memory location of the data item and is constructed by an iterative algorithm. The nodes of the graph fall into three categories:

1. a data item node represents a memory location which corresponds to a non-empty set of (parts of) data items;
2. a port node represents the place where the data flow crosses the limits of a program;
3. a constant node represents a constant.

For a given memory location [offset$_0$:size$_0$], the following algorithm constructs the data flow graph:

1. Initialize the pool of active memory locations with a one–element set [offset$_0$:size$_0$]; add the new data item node [offset$_0$:size$_0$] to the data flow graph.
2. Fetch and remove a memory location [offset:size] from the pool.
3. Find all pairs of data items (a, b) related by some relationship R and having a non–empty memory intersection with [offset:size]:

$$[\text{offset:size}] \cap [\text{offset}_a\text{:size}_a] \neq 0$$

or

$$[\text{offset:size}] \cap [\text{offset}_b\text{:size}_b] \neq 0.$$

4. For each pair found at Step 3, calculate the initial memory locations, i.e.,
$$[\text{offset}_1\text{:size}_1] = [\text{offset:size}] \cap [\text{offset}_a\text{:size}_a]$$
and
$$[\text{offset}_2\text{:size}_2] = [\text{offset:size}] \cap [\text{offset}_b\text{:size}_b].$$

5. For each of the non-empty initial memory locations, calculate the induced memory location [offset$'$:size$'$] on the opposite side of the relationship R. For example, if $\text{size}_1 \neq 0$ then, depending on the type of R, we have the following cases:

   (a) *a move b* : Find the memory location to which the [offset$_1$:size$_1$] is transferred when a is assigned to b, i.e., [offset$'$:size$'$] belongs to [offset$_b$:size$_b$]:
   $$\begin{gathered}\text{offset}' = \text{offset}_b + (\text{offset}_1 \text{ - } \text{offset}_a)\\ \text{size}' = \min(\text{size}_1,\text{size}_b \text{ - } (\text{offset}_b \text{ - } \text{offset}');\end{gathered}$$

   (b) *a compute b*: Since in this type of relationship any part of the data item a depends on the whole data item b, then
   $$\begin{gathered}\text{offset}' = \text{offset}_b\\ \text{size}' = \text{size}_b.\end{gathered}$$

   The case $\text{size}_2 \neq 0$ is treated symmetrically.

6. If the size of the induced memory location $\text{size}' \neq 0$, then a new data item node [offset$'$:size$'$] is added to the data flow graph (only if this node does not already exist). The node [offset$'$:size$'$] is connected to the data item node [offset:size]. The arc leads from [offset$'$:size$'$] to [offset:size] if [offset$'$:size$'$] was induced from a, and backward otherwise. The arc is marked by the relationship R. The memory location [offset$'$:size$'$] is added to the pool if a fresh node was created.

7. Find all ports which have a communication data item *com* that overlaps with [offset:size]:
$$[\text{offset:size}] \cap [\text{offset}_{com}\text{:size}_{com}] \neq 0.$$

8. Further processing is similar to the *move* relationship.

9. Find all constants related to some data item a; add a constant node and the arc from this node to [offset:size].

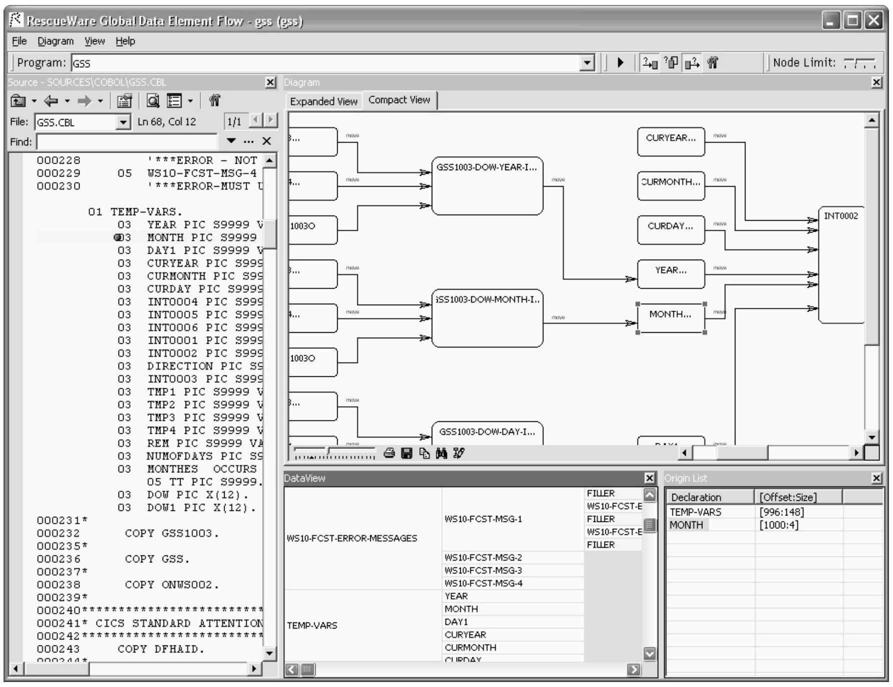
10. If the pool of active memory locations is empty, then terminate, otherwise resume from Step 2.

The data flow graph constructed by the algorithm may be used to show both forward and backward dependences inside the program to which the initial memory location belongs and the points where the data is transferred to other programs.

7. Graph visualization

Apparently, graph is one of the most adequate visual representations of a data flow. A typical scenario of data flow investigation looks like the following:

1. A user selects a starting point for the analysis.
2. Some part of the data flow graph for this point is constructed and displayed.
3. By selecting a node in the graph, a user obtains a list of data items that overlap with the corresponding memory location sorted in the decreasing order of “relevance”.



```
000228          '***ERROR - NOT
000229      05  WS10-FCST-MSG-4
000230          '***ERROR-MUST U

         01 TEMP-VARS.
             03  YEAR PIC S9999 V
             03  MONTH PIC S9999
             03  DAY1 PIC S9999 V
             03  CURYEAR PIC S999
             03  CURMONTH PIC S99
             03  CURDAY PIC S9999
             03  INT0004 PIC S999
             03  INT0005 PIC S999
             03  INT0006 PIC S999
             03  INT0001 PIC S999
             03  INT0002 PIC S999
             03  DIRECTION PIC S9
             03  INT0003 PIC S999
             03  TMP1 PIC S9999 V
             03  TMP2 PIC S9999 V
             03  TMP3 PIC S9999 V
             03  TMP4 PIC S9999 V
             03  REM PIC S9999 VA
             03  NUMOFDAYS PIC S9
             03  MONTHES  OCCURS
                 05 TT PIC S9999.
             03  DOW PIC X(12).
             03  DOW1 PIC X(12).
000231*
000232       COPY GSS1003.
000235*
000236       COPY GSS.
000237*
000238       COPY ONWS002.
000239*
000240*************************
000241* CICS STANDARD ATTENTION
000242*************************
000243      COPY DFHAID.
```

WS10-FCST-ERROR-MESSAGES	WS10-FCST-MSG-1	FILLER
		WS10-FCST-E
		FILLER
		WS10-FCST-E
		FILLER
	WS10-FCST-MSG-2	
	WS10-FCST-MSG-3	
	WS10-FCST-MSG-4	
TEMP-VARS	YEAR	
	MONTH	
	DAY1	
	CURYEAR	
	CURMONTH	
	CURDAY	

Declaration	[Offset:Size]
TEMP-VARS	[996:148]
MONTH	[1000:4]

Figure 3. Data flow visualization

4. When a user selects an item in the list, HyperCode positions both the source code and memory layout on this data item. Then a user may resume the process by taking the selected data item as a new starting point.
5. If the selected node is a port node, a user may jump to another program by fetching the initial memory location attached to the port node at Step 8 of the algorithm from the previous section.

In practice it has turned out that the main problem of such a visualization is huge and intricate data flow graphs. For example, as soon as the data flow reaches some "system" variable, the whole program data flow graph is trapped. Partially this is due to the fact that we consider context-independent data flow analysis. Some approaches to the solution of this problem are described below.

7.1. Constraints

The practice of data flow exploration in the general context of business rule extraction showed that, depending on how the starting point was obtained, a user is interested in either the forward data flow analysis, or backward, or both. For example, if the starting point is an in-port, then it makes sense to trace the forward dependences. On the contrary, for a constant-based condition, a user most probably will need to navigate the data flow backward. For this purpose we provide options allowing for cutting off the part of the data flow graph that is not interesting to the user.

The construction and placement of the whole data flow graph is not only resource consuming, but also, generally speaking, undesirable because a user will hardly be able to comprehend the huge picture at once. It is more important to present some neighborhood of the given data item and provide a mechanism to control the size of the sub-graph depending on the user preferences and the complexity of the graph. We allow for imposing a restriction on the total number of nodes in the neighborhood and implicitly on its radius, presuming the breadth-first search method of graph construction.

7.2. Graph unfolding

Suppose that we have two dependences between a couple of data items: the first one being direct while the second one being formed by a chain of assignments and redefinitions by other data items. Then the clearness of the first dependence is of higher priority. The significance of local information is reflected by the use of *graph unfolding* — a tree that is obtained from the graph by systematic copying of nodes with more than one arc. The procedure is also known as *cloning*.

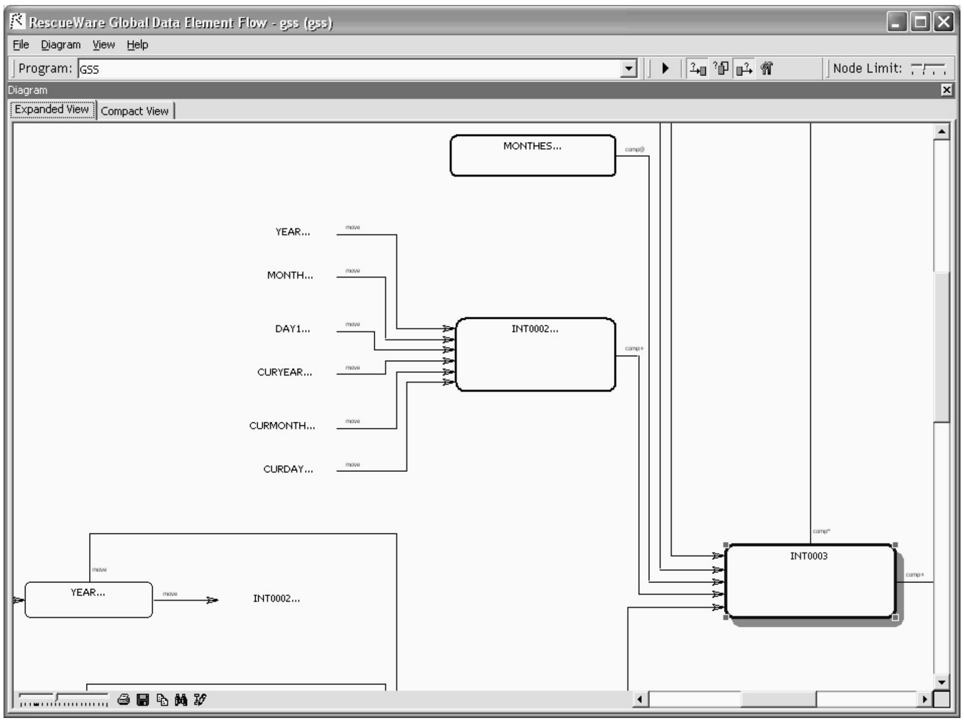


Figure 4. Data flow graph unfolding

The cloning is performed in such a way that each node of the original graph has a counterpart, called *representative*, in the resulting tree with the same number of adjacent arcs and a number of clones with exactly one incoming arc. The methods of visualization of and manipulation with unfolded graphs are much simpler than the general ones. In particular, we can guarantee that for any node all related nodes are placed nearby. The unfolding may also be placed in such a way that all arcs go in one direction, e.g. from left to right, even if the data flow graph presents both forward and backward dependences. Of course, the tree structure can be placed without arc intersections.

Figure 4 shows an example of unfolding layout. All clones are displayed borderless. We allow for interactive restructuring of unfolding: if a user wants to see the relationships of some clone node, he can swap it with a corresponding representative node. The resulting tree is obviously a different unfolding of the same data flow graph.

8. Path explorer

The possibility to restrict the size of the neighborhood of the initial data item is often not enough to get rid of the redundant information in the data flow graph. In order to achieve the precise specification of the neighborhood, we allow a user to construct it incrementally, selecting only the significant information paths. This process resembles a game of maze exploration: the user purposefully moves along some path gradually getting more general vision of the maze. The system prevents the user from falling into infinite loops and highlights the places that were already visited but not yet investigated. Moreover, when the user returns to some previously visited place he finds it in the same degree of openness at which he has left it. Unlike the game of maze, we do not construct and place the whole "map" in advance. The preliminary (and for the most part unnecessary) work of graph placement would take too much time. On the other hand, if the graph is constructed incrementally, the addition of a new node may completely change the layout and disorient the user. We have decided to use the standard tree view representation by extending its behavior in such a way that at most one of the nodes corresponding to the same memory location can be expanded.

A typical scenario of data flow exploration is the following:

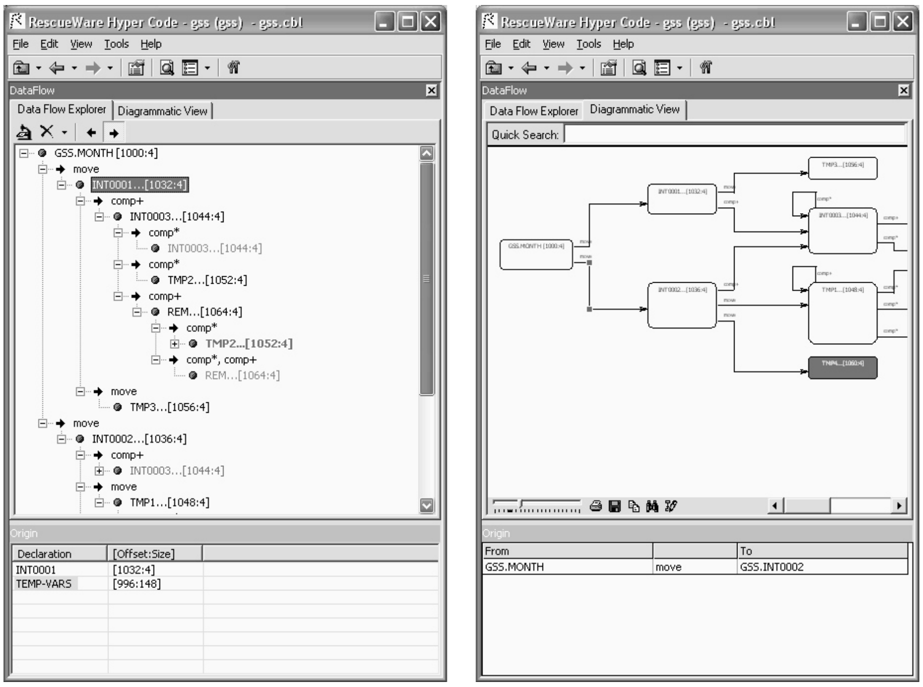
1. A user specifies the starting point of the analysis.
2. Expansion of the node that corresponds to the current program point leads to the list of all its immediate dependences and corresponding data items.
3. A data item from the list that was obtained on the previous step or any item that was not expanded yet is selected as the next current point.
4. Steps 2, 3 are iterated as long as needed.

Experimentally we have found out that the readability of the tree improves significantly if browsing is restricted either to forward or backward direction. The direction of exploration is fixed for each starting point. A user may change the direction but then the process of exploration resumes from the beginning.

A user may also switch to the diagrammatic representation at any time in order to have the general view of the explored area.

9. Impact report

If the starting point is a data item that is used for inter-program communication, for example, that is passed as a parameter via a common global area



```
⊟ GSS.MONTH [1000:4]
  ⊟ move
    ⊟ INT0001...[1032:4]
      ⊟ comp+
        ⊟ INT0003...[1044:4]
          ⊟ comp*
              INT0003...[1044:4]
          ⊟ comp*
              TMP2...[1052:4]
          ⊟ comp+
            ⊟ REM...[1064:4]
              ⊟ comp*
                ⊞ TMP2...[1052:4]
              ⊟ comp*, comp+
                  REM...[1064:4]
      ⊟ move
          TMP3...[1056:4]
  ⊟ move
    ⊟ INT0002...[1036:4]
      ⊟ comp+
        ⊞ INT0003...[1044:4]
      ⊟ move
        ⊟ TMP1...[1048:4]
```

Declaration	[Offset:Size]
INT0001	[1032:4]
TEMP-VARS	[996:148]

From		To
GSS.MONTH	move	GSS.INT0002

Figure 5. Path explorer

to the program being called, it might happen that the graph of dependent data items touches upon a considerable part of programs in the project and passes through a large number of external objects, such as screen forms, tables, or files. Huge sizes and heterogeneity of the resulting graph requires a systematic study and processing of acquired knowledge. The impact report presents the resulting graph as a hierarchy over data items and corresponding memory locations. On the top level of the hierarchy, the information is grouped by repository objects. Here external objects, which carry inter-program relationships, are considered along with programs. Such a grouping allows for catching the general picture straight away: how many and what programs depend on the value of the data item that was chosen as the starting point. The next level of detail contains the list of data items that caused a particular object to appear in the report. For instance, for a screen form, those data items that are used to communicate with the screen are listed. Finally, it is possible to see the so-called *confirmation path* for each impacted occurrence of a data item. In contrast to the graphical representation of data flows, this approach is not aimed at showing all existing dependence paths from the starting point to a particular program point that appears in the data flow graph. We show only the list of all affected program points and are ready to explain why they were affected by producing an arbitrary (hopefully the shortest) path in the data flow graph that leads from the starting point to the selected item. The linear presentation of a confirmation path makes it much easier to trace it as compared to graphical representation. That is especially important because of a large size of data flow graphs for real programs.

A typical scenario of using impact report consists in the following:

1. A user specifies a number of starting points for the analysis and initiates the construction of the report.
2. When the report is done, a user selects an object of interest on the top level for detailed investigation.
3. Moving down the hierarchy, a user selects an impacted data item for an object selected on the previous step.
4. Then, for each affected occurrence of the data item, a user may see a confirmation path.
5. If it is necessary to view another object or data item, a user should return to a higher level of the hierarchy and resume exploration in a different direction.

Thus we present a general project-wide view in a compact form at the expense of ignoring local details.

10. Related works

Today the problems of reengineering are very popular. An extensive survey of both research and commercial projects in the field can be found in [17, 20]. The tools for program code browsing similar to HyperCode are widely used both in software development tasks [8, 7, 12, 14, 15] and in the problems of software comprehension and maintenance [5, 11, 13, 16].

The most related system for program comprehension, which we know of, is Gupro [3]. It allows for working with a project that include programs in various languages, including Cobol and JCL. Gupro supports incremental analysis and inter-program relationships.

The diagrammatic representation of information collected by various types of program analyzers is extensively used in a number of systems. One of the most advanced among them is Rigi and its extensions [10, 18]. The importance of reducing the visual complexity of a graph is addressed in [9], where hierarchical displays are considered: selection operations alternate with hiding or grouping operations. Since we aim at the development of an interactive tool for data flow exploration, we should provide a fast and good-quality method of automatic graph drawing. An overview of graph drawing methods can be found in [21].

A different technique for information visualization is offered by SeeSoft system. It represents the text lines as thin colored rows within columns. The color of each row is determined by a statistic associated with each line [6]. SeeSlice promotes this technique for program slice visualization and, in particular, cross-program data flow impact analysis [2].

11. Conclusion

The study of a data flow is an important subtask in the process of program understanding. We have described some means for visualization of a data flow in heterogeneous legacy systems. The presented techniques allow for processing of a large amount of information gathered by the data flow analysis. The system combines various representations of dependences between data items and allows a user to focus on the view that is the most appropriate to her/his current task.

The low-level type systems of legacy languages, such as COBOL, lead to the necessity of dealing with memory locations, rather than the declared data items. This provides simple rules for evaluation of dependences based on the overlay of memory locations, but at the same time requires special efforts to establish natural correspondence between memory locations and source code constructs.

The main problem we tried to attack is the huge size and entanglement of data flow graphs. Several approaches aimed at reducing the complexity by incremental construction of the graph and limitation of the viewpoint neighborhood have been presented.

References

[1] Baburin D.E., Bulyonkov M.A., Emelianov P.G., Filatkina N.N. Visualization facilities in program reengineering // Programming and Computer Software. — 2001. — Vol. 27, N 2. — P. 69–77.

[2] Ball Th., Eick S. G. Visualizing program slices // Proc. of IEEE Symp. on Visual Languages, St. Louis, Missouri, 1994 (VL1994). — P. 288–295.

[3] Kullbach P. D. B., Winter A., Ebert J. Program Comprehension in Multi-Language Systems. — Koblenz, 1998. — (Tech. Rep. / Universität Koblenz-Landau, Institut für Informatik; N 4–98).

[4] Chen P. P. The entity relationship model — toward an unified view of data // ACM Trans. on Database Syst. — 1976. — Vol. 1, N 1. — P. 9–36.

[5] Cleveland L. A program understanding support environment // IBM Systems J. — 1989. — Vol. 28, N 2. — P. 324–344.

[6] Eick S. Graphically displaying text // J. of Computational and Graphical Statistics. — 1994. — Vol. 3, N 2. — P. 127–142.

[7] Morrison R.R. et al. Current directions in hyper-programming // Lect. Notes Comput. Sci. — 2000. — Vol. 1755. — P. 316–340.

[8] Jones T. S., Welsh J. Software visualisation in a language based editor // Proc. of the 1997 Software Visualisation Workshop, Adelaide, Australia, 11–12 Dec. 1997. — P. 9–18.

[9] Kimelman D., Leban B., Roth T., Zernik D. Reduction of visual complexity in dynamic graphs // Lect. Notes Comput. Sci. — 1994. — Vol. 894. — P. 218–225.

[10] Müller H. A. Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications: PhD thes. — Rice University, Houston, 1986.

[11] Müller H. A., Tilley S. R., Orgun M. A., Corrie B. D., Madhavji N. H. A reverse engineering environment based on spatial and visual Proc. of SIGSOFT'92. — SIGSOFT Software Engineering Notes. — 1992. — Vol. 17, N 5. — P. 88–98.

[12] Normark K. A hyperstructure programming environment for CLOS // Technology of Object-Oriented Languages and Systems — TOOLS4. — Prentice Hall, 1991. — P. 127–140,.

[13] Oman P. et al. Maintenance tools // IEEE Software. — 1990. — Vol. 7, N 3. — P. 59–65.

[14] Østerbye K. Literate Smalltalk programming using hypertext. — Aalborg, 1993. — (Rep. / Univ. of Aalborg. Institute for Electronic Systems; N R 93-2025).

[15] Pokrovskii S.B., Stepanov G.G. Hypertext-based environment for software development // Aids and Tools for Programming Environment. — Novosibirsk, 1995. — P. 100–110 (in Russian).

[16] Sim S. E., Clarke Ch. L. A., Holt R. C., Cox A. M. Browsing and searching software architectures // Proc. of the IEEE Intern. Conf. on Software Maintenance. — IEEE Computer Society Press, 1999. — P. 381–390.

[17] Sim S. E., Storey M.-A. D. A structured demonstration of program comprehension tools // Proc. of WCRE 2000, Brisbane, Australia, 23–25 Nov. 2000. — P. 184–193.

[18] Best C., Storey M.-A. D., Michaud J. SHriMP views: An interactive and customizable environment for software exploration // Proc. of Intern. Workshop on Program Comprehension (IWPC '2001), Toronto, Ontario, Canada, May 12–13, 2001. — P. 111–112.

[19] Automated Software Re-engineering / Ed. by A. N. Terekhov, A. A. Terekhov. — St.-Petersburg, 2000 (in Russian).

[20] Tilley S. R. Domain-Retargetable Reverse Engineering: PhD thes. — Department of Comput. Sci., Univ. of Victoria, January 1995 (available as technical report DCS-234-IR).

[21] Tollis I. G., Di Battista G., Eades P., Tamassia R. Graph Drawing: Algorithms for the Visualization of Graphs. — Prentice Hall, 1999.