A Rule-Based Tool for Reverse Engineering from Source Code to Graphical Models

Hai Huang Kazuo Sugihara Isao Miyamoto

Dept. of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, HI 96822, U. S. A.

Abstract

This paper presents a rule-based tool MG (Model Generator) *for reverse engineering from source code to graphical models. For a given graphical model formalism, generation rules describe how to translate each primitive element (e.g., an if statement) of a programming language into a piece of a diagram in the graphical model and how to assemble the produced pieces of diagrams and generate diagrams from them. MG can produce software specification in different models by changing the generation rules. This feature enables maintainers to obtain the information they need by specifying rules for generating it. MG is also independent of programming languages. It is currently used to generate four models and links between them from COBOL source code, where each model represents a coherent aspect of the source code such as control flow and functional structure.*

1 Introduction

Software maintenance plays a very important role in the software life cycle, since software systems are continuously changing, from the first version released to customer, until the software systems are discarded. It was reported [13] that companies spend 60 to 80 % of their software budgets on maintaining existing software and 90 % of software resources is consumed on fixing and reengineering existing applications. One of the major reasons for the high cost of software maintenance is lack of necessary documentation about the software systems to be maintained. For most of software systems, the only reliable information for maintenance activities is source code of the system. Original requirements and design specifications may no longer exist or may be out of date. Changes may not be documented and no one knows how the changes are applied. The only way to know properties and behavior of the system is to study the source code. Without support of a computer system, this is time-consuming and very expensive work.

In recent years, some reverse engineering tools have been developed [1, 2, 3, 11]. However, they are oriented to software written in one specific programming language and extract only one or a few particular aspects of the software from source code. This limitation brings difficulties if the maintainers want to extract information from source code other than the information the tools provide. Some AI based reverse engineering systems such as PUDSY [12], Proust [9, 10], Harandi and Ning's system [5, 6] also deal with programs written in one particular programming language and abstract specific information from the programs. Little flexibility is provided by these tools in order to allow maintainers to specify the information that they need to extract or abstract from source code.

The *Software Maintenance Assistant* (*SMA*) is under development at the Software Engineering Research Laboratory in the Department of Information and Computer Sciences, the University of Hawaii. It aims at providing maintainers with an integrated environment for supporting their maintenance activities. Reverse engineering tools in SMA have been developed which extracts requirements and design specifications from source code. They are currently used in SMA to generate eight graphical models and links between them from COBOL source code: IOPM (Internal Operational Profile Model) representing control flow, EOPM (External Operational Profile Model) representing external behavior of a system, FSM (Functional Structure Model), UIM (User Interface Model), FM (Function Model), CPM (Conceptual Process Model) representing data flow, DM (Data Model), and DBM (Database Model) [8].

This paper presents *Model Generator* (*MG*) which is one of the reverse engineering tools and is used to

0-8186-2830-8/92 $03.00 © 1992 IEEE

produce the first four of the eight models listed above. MG is a rule-based tool in which, for a given programming language and a given model formalism, rules describe how to translate each primitive element (e.g., an if statement) of the programming language into a piece of a diagram in the model and how to assemble the produced pieces of diagrams and generate diagrams from them. It is implemented in C on SUN SPARC stations.

An important feature of MG is that it is flexible to deal with different programming languages and extract various types of information from source code. A limitation of MG is that it is difficult to generate a certain type of models such as CPM by using only the rule-based approach. The other reverse engineering tools in SMA generate such models by *ad hoc* algorithms [8], although a rule-based approach is also used to some extent in order to make the tools independent of programming languages.

Another important feature of MG is that it uses graphical models to represent the information extracted by reverse engineering. Furthermore, all the graphical models used in MG are represented in a uniform language *MERA* (*Meta-Entity-Relation-Attribute*) [4]. MERA is an extension of the well-known Entity-Relationship model. It is enriched with type hierarchies, abstraction of objects (either entities and relations), and semantic constraints.

The most unique feature of MERA is that it can be used to define a variety of graphical models by creating a *meta-graph* in MERA that describes the formalism of a specific model. The MERA editor reads the meta-graph and configures itself to edit diagrams in the graphical model defined by the formalism described in the meta-graph. It is implemented in C using the X window system and currently used in SMA to deal with over a dozen different graphical models for software specification. The models are based on diverse paradigms such as Petri nets, finite automata, queueing networks, data flow, and control flow.

This paper is organized as follows. Section 2 presents the configuration of the rule-based tool MG for reverse engineering. Section 3 discusses representation of rules and an algorithm for model generation. Section 4 presents examples of reverse engineering produced by MG. Finally, Section 5 summarizes results of this paper.

2 Rule-Based Reverse Engineering

Graphical models are very useful for requirement and design specifications of a software system as well as documentation. In software engineering practice, various graphical models such as data flow diagram, structure chart, and control flow diagram are used in different stages of system life cycle. Thus, these graphical models are targets of our reverse engineering tools.

In general, there are three types of knowledge required to construct a graphical model from source code of a program:

- knowledge about a programming language in which the program is written,
- knowledge about a formalism of the graphical model to be constructed, and
- knowledge about relationships between graphical components of the model and primitive elements of the programming language.

Tools for constructing the graphical model from source code should have these types of knowledge. However, ways of the knowledge representation make differences among the tools.

Although tools which embed the knowledge in implementation may be efficient, they lose flexibility. The hard-wired knowledge limits usage of the tools. They can only deal with programs written in a particular programming language and extract certain types of information from source code. Changes on the knowledge included in a tool may require the tool to be re-implemented or even re-designed.

In contrast, a rule-based approach can bring flexibility by explicitly representing the knowledge as rules of model generation. Any change of the knowledge only affects the rules. Few modifications of the tool are needed. Rules are much easier to modify than the tools themselves. Furthermore, the rule-based approach provides users with a means of customizing the tools so that the tools can deal with different programming languages and produce the models that the users want.

The rule-based approach has its own limitation due to complexity of rules. Since a load of the knowledge is shifted from a tool to rules, the capability of the tool strongly depends on the expressive power of rules. The more expressive power the rules have, the more variety of models can be generated and at the same time the more complicated the rules are.

There are two problems associated with the complexity of rules. One is difficulty to create rules. Another is difficulty to verify the rules. Due to these problems, the rules should not be complicated and hence the expressive power of rules should be limited to some extent.

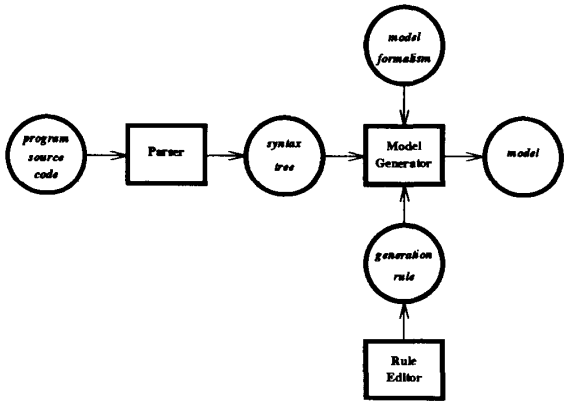

Figure 1: Model generator for reverse engineering.

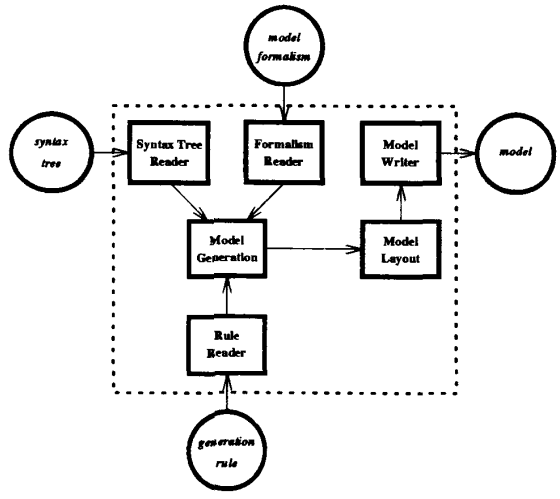

Figure 2: Configuration of Model Generator.

Graphical models whose components have syntactically close relationships with primitive elements of programming languages are suitable for the rule-based approach, since it is easy to describe transformations in rules. However, it is somehow difficult to create model generation rules which include processing of the information extracted from source code. If rules could describe complicated processing, the rules would be just like a program and hence it would be difficult to write them and check their consistency. Therefore, the rule-based approach is effective to generate those models whose graphical components are syntactically related to elements of a programming language, but not all models.

Model Generator (*MG*) is a rule-based tool that is one of reverse engineering tools in SMA. We intend to use MG to generate graphical models for which rules are easy to write and verify. Among those models used in SMA are *IOPM* (*Internal Operational Profile Model*) representing control flow, *EOPM* (*External Operational Profile Model*) representing external behavior, *FSM* (*Functional Structure Model*) representing relationships between modules and relationships between modules and external objects, and *UIM* (*User Interface Model*) representing a user interface.

MG is used for reverse engineering in SMA as shown in Figure 1. *Parser* analyzes source code of a system and produces a syntax tree of the source code. For programs written in different programming languages, different parsers are needed. Due to differences between programming languages, a detail structure of the syntax tree is dependent on the language in which the source code is written. *Model Formalism* provides the information about the syntax of a model. This information includes entity types defined in the model, relation types defined between entities, and attributes for the entities and relations. *Generation Rule* captures knowledge about how to generate models of a particular type from the syntax tree. It is a key component of MG to make MG independent of programming languages and target models. The syntax and semantics of generation rules will be discussed in Section 3. *Rule Editor* provides a means of creating or modifying generation rules. It is a syntax-driven editor which ensures syntactically correct editing.

The configuration of MG is shown in Figure 2. MG has six components. Among them, the model generation component is an inference engine of MG. It finds an applicable rule, interprets the rule, and performs the actions specified in the rule. Details on implementation of MG are presented in [8].

3 Model Generator

3.1 Generation Rules

A generation rule consists of two parts: *Triggers* and *actions*. A trigger indicates when the rule can be applied. Each rule may have one or more triggers. When any of the triggers in the rule has triggered, actions in the rule are executed sequentially. Each trigger has a set of *conditions* to be satisfied. The first one is a required condition called node type. It indicates to which type of a node in the syntax tree the rule is applied. The other conditions are optional and indicate the additional requirements to apply the rule.

The conditions in each trigger are logically connected with AND. Each trigger also has a set of return values, each of which is the identification number of an entity generated by an action of the rule. All triggers in a rule are logically connected with OR.

Rules and triggers are evaluated in the order of the rules stored in a knowledge-base and in the order of the triggers in each rule. In general, the model produced depends on the order of rules. As mentioned above, however, we intend to use MG to generate a model for which only one rule is usually applicable at a time and there is little interference between rules when two or more rules are applicable at a time. For those models, the model produced does not depend on the order of rules and it is easy to check the rules manually.

Each rule has a set of actions which are executed sequentially. There are six types of actions: *Graph*, *entity*, *relation*, *connect*, *process*, and *group*. A *graph* action creates a graph. It takes four arguments – the name of the graph, formalism used in the graph, entry and exit entities of the graph. The scope of the *graph* action includes all the subsequent actions contained in the rule and all the subsequent actions executed before the next *graph* action is executed.

An *entity* action creates an entity in a graph. It takes six arguments – id of the entity, type of the entity, name of the entity, an optional reserved flag, an optional unique flag, and attributes of the entity. The reserved flag of the entity indicates whether the entity is reserved and will be kept in the model if it is merged with another entity. The unique flag indicates whether the name of the entity is unique in the model.

A *relation* action creates a relation or link between two entities. It takes seven arguments – id of the relation, type of the relation, name of the relation, subject entity, object entity, an optional reserved flag, and attributes of the relation. The subject entity and the object entity of a relation can be in different graphs. In this case, a link between the graphs is created instead of a relation. For the entity which is not created by the actions in the current rule, the graph name and the entity name are used to locate the entity. The reserved flag indicates whether the relation is reserved and will be kept in the model if it is merged with another relation.

A *process* action tells MG to find a new rule to process the node indicated by the action. It takes two arguments – the node to be processed and a set of entities to be returned. The entities in the action correspond to entities included in the trigger condition of the rule which processes the node. If the trigger condition includes more entities than the action, extra entities are discarded. If the trigger condition includes less entities than the action, the remaining entities in the action are set to be empty.

A *group* action groups a set of actions and applies them repeatedly to a set of nodes which share the same parent. The process ends if the stop node is processed or there does not exist a node which is a sibling of the currently processed node. The group action takes three arguments – start node, stop node which may be missing, and a set of actions. It can take any type of an action as its sub-action, except a *group* type action.

A *connect* action acts almost same as a *relation* action, except it can be used only as a sub-action of a *group* action. It has one more argument than the *relation* action – an optional reverse flag. It creates a relation or link between two entities which are created in two subsequent loops of a *group* action. The subject entity is created by the last loop of the *group* action and the object entity is created by the current loop of the *group* action, or vice versa if the reverse flag indicates so.

The following is a rule for generating control flow in IOPM (Internal Operational Profile Model) from an IF statement in ANSI COBOL 85 [7]. In a syntax tree, the IF statement corresponds to a node whose type is "IF-ST." The node has three children: *Condition* node, *then* node and *else* node. Children of the *then* node and *else* node are syntax trees for statements in the *then* part and the *else* part of the IF statement, respectively.

```
RULE(CONDITION("IF\_ST" RETURN(1 9))
 ENTITY(1 "place" NAME("")
  ATTRIBUTE("initial\_no\_tokens" NAME("0")))
 ENTITY(2 "branch" NAME(PATH(1): NODE-ELEMENT))
 ENTITY(3 "place" NAME("")
  ATTRIBUTE("initial\_no\_tokens" NAME("0")))
 ENTITY(4 "place" NAME("")
  ATTRIBUTE("initial\_no\_tokens" NAME("0")))
 ENTITY(9 "place" NAME("")
  ATTRIBUTE("initial\_no\_tokens" NAME("0")))
 PROCESS(PATH("THEN\_NODE") RETURN(5 6))
 PROCESS(PATH("ELSE\_NODE") RETURN(7 8))
 RELATION(1 "consuming\_arc" NAME("") 1 2)
 RELATION(2 "condition" NAME("") 2 3 RESERVED
  ATTRIBUTE("condition" NAME("true"))
  ATTRIBUTE("probability" NAME("0.5")))
 RELATION(3 "condition" NAME("") 2 4 RESERVED
  ATTRIBUTE("condition" NAME("false"))
  ATTRIBUTE("probability" NAME("0.5")))
 RELATION(5 "dummy" NAME("") 3 5)
 RELATION(6 "dummy" NAME("") 6 9)
 RELATION(7 "dummy" NAME("") 4 7)
 RELATION(8 "dummy" NAME("") 8 9))
RULE(CONDITION("THEN\_NODE" RETURN(1 2))
 CONDITION("ELSE\_NODE" RETURN(1 2))
  GROUP(PATH(1)
   PROCESS(PATH(0) RETURN(1 2))
   CONNECT(1 "dummy" NAME("") 2 1)))
```

In general, each rule deals with one element in a

programming language. For a composite element such as a branch statement (an IF statement in COBOL), one rule deals with only the main part of the element and leaves the other parts for other rules. In the above example, the rule for IF_ST node generates only the skeleton of the branch control flow and leaves the *then* and *else* parts to the other rules. Similarly, the rule for the *then* and *else* parts provide only the frame for each part and let other rules to handle individual statements in the *then* and *else* parts. In this way, it is easy for users to check the correctness of the rule manually, since they only have to concentrate on one rule at a time and need not worry about interferences between rules most of time.

3.2 An Algorithm for Model Generation

There are two inputs for model generation: A syntax tree to be processed and a set of generation rules to be used. Each node of the syntax tree contains the following information: Id of the node, name of the node, type of the node, value of data contained by the node, and some other relevant information. The rules contain the information mentioned in Section 3.1. The output of model generation is a set of diagrams in MERA which represent various aspects of software.

A recursive algorithm for model generation is presented below, where it is initially applied to the root of the syntax tree.

1. Find a rule applicable to the current node.
2. If there is no rule applicable to the node, create a dummy entity and fill the return values with the dummy entity. Then, if another rule has invoked the last rule, return to it. Otherwise, terminate the process.
3. If an applicable rule is found, execute actions of the rule sequentially as follows.
   (a) If an action is a *graph* action, create a new graph and set the current graph to it.
   (b) If an action is an *entity* action, create an entity and put it into the entity array which stores return values.
   (c) If an action is a *relation* action, create a relation which connects the subject and object entities when both of them are in the same graph or create a link between the subject and object entities when they are in different graphs.
   (d) If an action is a *process* action, find the node to be processed and apply the model generation algorithm to the node recursively.
   (e) If an action is a *connect* action, create a relation or link which connects the subject and object entities.
   (f) If an action is a *group* action, save the current graph and find the start node and the stop node. Then, execute the above substeps according to the type of sub-actions of the *group* action from the start node. When one loop of execution of sub-actions has finished, reset the current graph to the saved graph.
4. If there is no more action to be executed, set the return values from the entity array. Then, if another rule has invoked the last rule, return to it. Otherwise, terminate the process.

4 Examples

An example of source code is given below, which is a section of an airline reservation program written in COBOL programming language.

```
UPDATE-RESERVATION SECTION.
UPDATE-RESERVATION-ENTRY.
  IF TR-TRANSACTION-CODE = ZERO
    PERFORM MOD-IFY
  IF RR-PASSENGER-NAME1 = SPACES AND
     RR-PASSENGER-NAME2 = SPACES
    MOVE SPACES TO PRINT-RECORD
    MOVE 'RESERVATION RECORD DELETED' TO PRT-MESSAGE
    MOVE RESERVATIONS-RECORD TO PRT-REC
    WRITE PRINT-RECORD
  ELSE
    PERFORM WRITE-NEW-RESER
  ELSE IF RR-PASSENGER-NAME1 = SPACES OR
          RR-PASSENGER-NAME2 = SPACES
    PERFORM MOD-IFY
    MOVE SPACES TO PRINT-RECORD
    MOVE 'RESERVATION ADDED' TO PRT-MESSAGE
    MOVE TRANSACTION-RECORD TO PRT-REC
    WRITE PRINT-RECORD
  ELSE
    MOVE SPACES TO PRINT-RECORD
    MOVE 'NO ROOM ON FLIGHT' TO PRT-MESSAGE
    MOVE TRANSACTION-RECORD TO PRT-REC
    WRITE PRINT-RECORD.
UPDATE-RESERVATION-EXIT.
  EXIT.
```

The current reverse engineering tools in SMA generates eight graphical models and links between them from COBOL source code. Four of the models are generated by MG. As examples, three models for the source code given above are presented in Figures 3, 4 and 5. The first model is IOPM (Internal Operational Profile Model) which represents control flow by using a Petri net. The second model is EOPM (External Operational Profile Model) which represents

external behavior by using a state transition diagram. The third model is FSM (Functional Structure Model) which represents a functional structure of the source code such as relationships among code blocks and relationships between code blocks and external objects (e.g., files and terminals).

There are five types of entities of IOPM in Figure 3: Entry, exit, process, place, and branch. The entry and exit represent the start and end of control flow, respectively. A process, a place, and a branch represent an action, a state, and a branch in control flow, respectively.

There are three types of entities of FSM in Figure 5: Actor, object, and action. An actor represents a code block which performs some specific function. An object represents a file or terminal which is manipulated by a code block. An action represents an operation that is applied to an object by an actor.

Another example of source code in ANSI COBOL 85 is given below. Note that it contains two loops. IOPM produced from this source code is shown in Figure 6

```
UP-DATE SECTION.
UP-DATE-ENTRY.
  IF RP-END-OF-TRANS = '1' AND
     RP-END-OF-RESERV = ZERO
    PERFORM FINISH-RESERVATION
      UNTIL RP-END-OF-RESERV = '1'
  ELSE IF RP-END-OF-TRANS = ZERO AND
          RP-END-OF-RESERV = '1'
    PERFORM FINISH-TRANSACTION
      UNTIL RP-END-OF-TRANS = '1'
  ELSE IF RP-END-OF-TRANS = ZERO AND
          RP-END-OF-RESERV = ZERO
    PERFORM MATCH-FLIGHT
  ELSE
    NEXT SENTENCE.
UP-DATE-EXIT.
  EXIT.
```

5 Conclusion

This paper have presented a rule-based tool MG (Model Generator) for reverse engineering from source code to graphical models defined in MERA (Meta-Entity-Relation-Attribute) that is a uniform language for a variety of graphical models. MG enables maintainers to obtain the information they need by specifying rules for generating it. It is also independent of programming languages in which source code is written. MG is currently used in SMA (Software Maintenance Assistant) as a tool for generating four models from COBOL source code, where each model represents a coherent aspect of the source code such as control flow and functional structure.

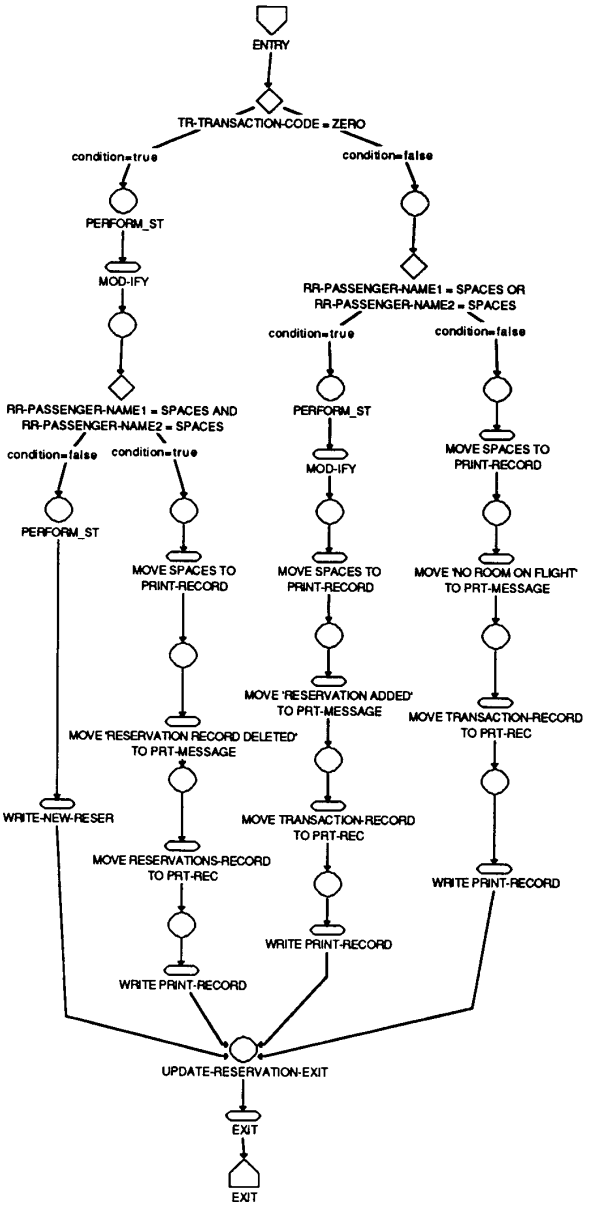

Figure 3: Internal Operational Profile Model.

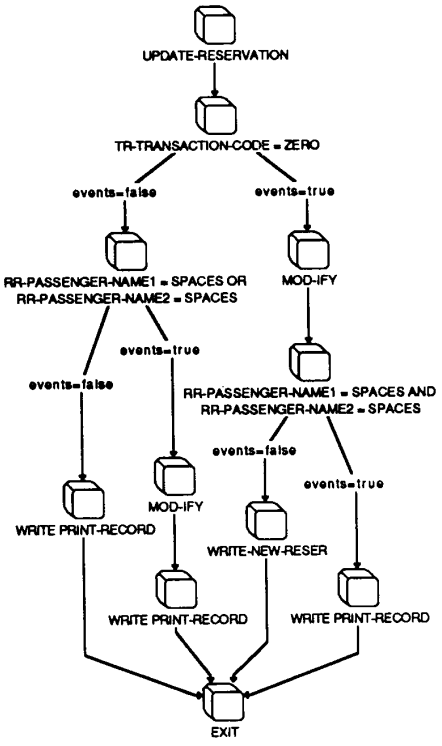

Figure 4: External Operational Profile Model.

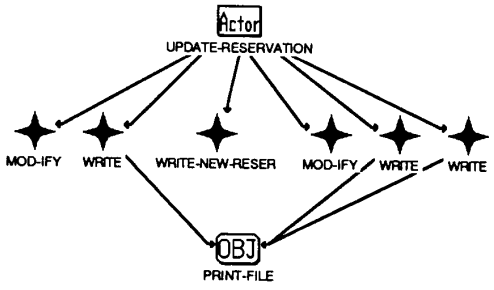

Figure 5: Functional Structure Model.

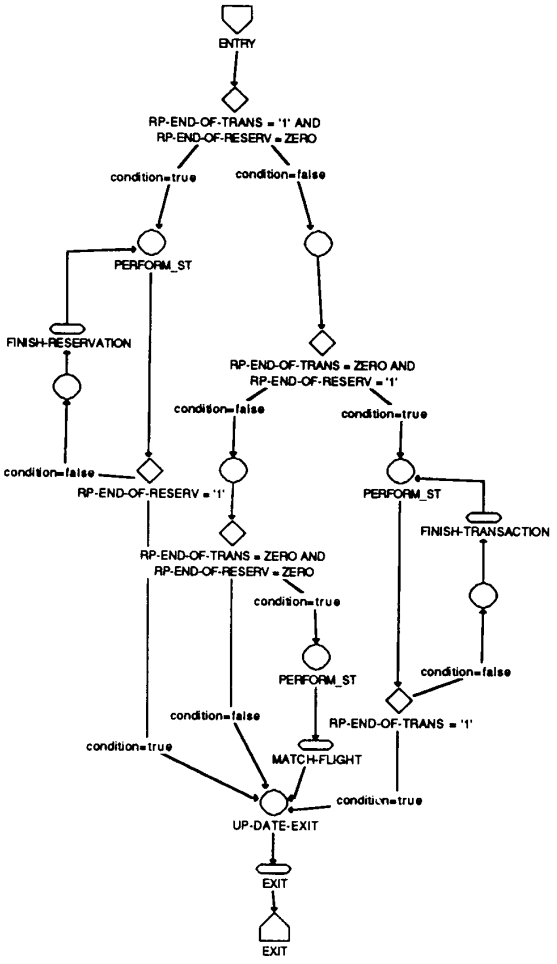

Figure 6: IOPM of another example.

Acknowledgements

The authors would like to thank Narsimha Polykampally and Chengdi Sheng for their help on implementation of Model Generator and anonymous referees for their helpful comments on an earlier manuscript of this paper.

References

[1] P. Benedusi, A. Cimitile and U. De Carlini, "A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance," *Proc. IEEE 1989 Conference on Software Maintenance*, Oct. 24-27, 1989, pp. 180-189.

[2] Y. F. Chen and C. V. Ramamoorthy, "The C information abstractor," *Proc. 10th International Computer Software and Applications Conference (COMPSAC)*, Oct. 8-10, 1986, pp. 291-298.

[3] Y. F. Chen, M. Y. Nishimoto and C. V. Ramamoorthy, "The C information abstraction system," *IEEE Trans. on Software Engineering*, Vol. 16, No. 3, pp. 325-334, Mar. 1990.

[4] D. Chin, K. Takeda and I. Miyamoto, "MERA: A flexible graphical software description metalanguage," Software Engineering Research Laboratory, Dept. of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, Tech. Rep. 91-47, Sep. 1991.

[5] M. T. Harandi and J. Q. Ning, "PAT: A knowledge-based program analysis tool," *Proc. 1988 IEEE Conference on Software Engineering*, Oct. 24-27, 1988, pp. 312-318.

[6] M. T. Harandi and J. Q. Ning, "Knowledge-based program analysis," *IEEE Software*, Vol. 7, No. 1, pp. 74-81, Jan. 1990.

[7] Jim Inglis, *COBOL 85 for Programmers*, John Wiley & Sons, New York, NY, 1989.

[8] H. Huang, N. Polkampally and C. Sheng, "PU subsystem," Software Engineering Research Laboratory, Dept. of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, Tech. Rep. 91-66, Dec. 1991.

[9] W. L. Johnson and E. Soloway, "PROUST: Knowledge-based program understanding," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 3, pp. 267-275, Mar. 1985.

[10] W. L. Johnson, *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, Palo Alto, CA, 1986.

[11] D. R. Kuhu, "A source code analyzer for maintenance," *Proc. 1987 IEEE Conference on Software Maintenance*, Sep. 21-24, 1987, pp. 176-180.

[12] F. J. Lukey, "Understanding and debugging program," *International Journal of Man-Machine Studies*, Vol. 12, No. 2, pp. 189-202, Feb. 1980.

[13] J. Moad, "Maintaining the competitive edge," *Datamation*, Vol. 36, No. 4, pp. 61-66, Feb. 15, 1990.