PUC

Monografias em Ciência da Computação
nº 28/91

Design Recovery
A Multi-Paradigm Approach

Julio Cesar S. P. Leite
Antonio Francisco do Prado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 28/91
Editor: Carlos J. P. Lucena

Dezembro, 1991

Design Recovery
A Multi-Paradigm Approach *

Julio Cesar S. P. Leite
Antonio Francisco do Prado

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386 Telex:31078 Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

Resumo:

Reutilização de software é uma disciplina fundamental para se atingir incrementos de produtividade na produção de software. Uma forma de reutilização desejavel é a reutilização de desenhos de software, que por serem mais abstratos permitem uma maior flexibilidade de implementação. Neste artigo propomos uma estratégia para recuperação de desenho de software. Esta estratégia utiliza-se de três paradigmas: Inspeções, Método de Jackson para Sistemas (JSD) e Prototipação. A combinação desses três paradigmas é avaliada e é demonstrado o seu uso na recuperação do desenho da máquina de Draco.

Design Recovery
A Multi-Paradigm Approach

Julio Cesar S. P. Leite
Antonio Francisco do Prado
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

December 1990

Abstract

One of the reutilization aspects that is of major concern is how to get old code and turn it into a reusable asset. Our article proposes and analyzes a multi-paradigm approach to the problem of Design Recovery. Design Recovery is the first step in turning old code into a resuable product. Our aproach is being used in a special kind of old code, the Draco Prototype, a system that implements the concept of reusable components to the point of achieving reusability of Analysis, Design and Code. In recovering the Draco design, we have used Inspections, JSD, and Prototyping. The combination of those different paradigms is analyzed in detail and the results are reported.

I Introduction

Software reusability has started to be established as an important research area in software engineering. Several researchers [Freeman 88] have been pointing out that reuse is one of the effective ways to fight the so ever growing backlog of demanded software. The application of software reusability has, in most of the practical cases, been restricted to code reuse. Work in the direction of high orders of reuse [Matsumoto 87], [Neighbors 89], [Arango 88], [Seppänen 90], [Baxter 90], [Biggerstaff 89] has been performed but there are still many problems. Code reuse, which is being pursued by several software producers, suffers from an excessive knowledge fragmentation and as such is very much dependent on a well organized library [Prieto-Diaz 87]. High orders of reuse will provide less fragmentation and shall diminish the high cost of retrieval of reusable code.

Ideally a software construction process should be conducted within the reusability framework, that is maximize the reusability and produce products that have a high probability of reusability. The old lesson of Parnas that we should plan for change is a characteristic of the software business. What is usually a problem is that in general the code that is around in most software development systems [Freeman 87] was not written with reusability in mind, and is very difficult to reuse. Although this old code is hard to reuse, we can not afford the price of throwing away this knowledge. We must keep this knowledge and improve our capability of reusing this encoded, but fragmented, knowledge. As such we need to dig out knowledge from a source that was not structured for posterior retrieval and reuse.

Design recovery aims to capture, mainly from the existing code, the structure of the original design such that components make themselves more visible, as well as to provide means for a re-engineering of the components. Arango [Arango 88] defines design recovery as:

> Given an implemented software system, the problem of design recovery consists of producing a specification for that system and an explanation of why the existing implementation is in fact an implementation of the specification.

This paper reports on a strategy that is being used to recover the Draco design. Our main objective is to support a cooperative task in which a team identify module and data abstraction groupings and recover design structures. The concept of Inspections [Fagan 76] together with Prototyping is the base for the understanding and interpretation of the available information. The JSD [Jackson 83] is the base for the representation of recovered design structures. Our proposed approach is not as broad as, for instance, the Desire system [Biggerstaff 89], which aims an automation of the recovery process by relying in a domain model as a repository of known plans. Matching available information with pre-existing plans is the job of an automated assistant, which helps the software engineer in the recover task. Our work has basically centered its attention on the manual recover process as performed by a cooperative team.

The paper is organized as follows. The second section gives a brief description of the Draco system and the situation in which the design recovery is taking place. The third section describes our strategy. The fourth describes the process used to recover the Draco design and the results so far. We conclude describing some opportunities for tool support.

II The Problem

Draco is a prototype developed by Neighbors [Neighbors 84] to demonstrate his thesis that, similar to industrial production lines, software could be produced by assembling components. Central to this idea was the argument that using the components strategy, it would be possible to reuse analysis, design and code, thus achieving great improvements on the software production productivity.

On further developments of the Neigbhors' ideas, the Reuse Project and the Advanced Software Engineering Project at the University of California, Irvine, under the direction of Professor Peter Freeman, built the concepts underlining what is now called the Draco Paradigm. This paradigm, if compared to the recent proposals [Agresti 87] to substitute the life cycle paradigm is singular. Instead of relying on a general specification language, it relies on special specification languages. These special specification languages, called Domain languages, should encapsulate the knowledge of a defined problem area, such that systems dealing with this area could be specified in a language with constructs related to the problem area.

Draco as a software system, Figure 1, is composed of 4 subsystems. Besides the Parser, the Transformer, the Prettyprinter and the Refinement, there is a part of this last subsystem that is worth mentioning, the Tactics. The Tactics is the part of Draco where there is an encoding of meta level design knowledge. This meta level knowledge is responsible for the automation of the refinement process. The Parser is central to the Draco system because it defines the internal form, the anchor representation used by the system, in which all the change of representations takes place. In the SADT of Figure 1, the meta aspect of Draco is presented by the parser generator, the transformer generator, the prettyprinter generator and the refinement generator. The use of the codified domains is represented by the input language program, which is parsed, transformed, and refined to an executable language.

Our research group at PUC-Rio is interested in studying the Draco Paradigm, specially the process of knowledge acquisition oriented towards special languages. In order to pursue our research efforts and get more people involved it is important to have the Draco prototype around. Due to several problems related to software infrastructure we could not run the Draco system, which required a special Franz Lisp version. Our decision then was to reimplement Draco. This reimplementation is planned to happen in two steps, one the design recovery step and the other the re-engineering step. The design recovery step would produce, besides the design, a new implementation in Scheme.

Once the design becomes available, a re-engineering effort will transform the existing design in an object-oriented design, targetting the new implementation to a real object oriented language.

The situation in which we are doing the design recovery could be pictured as follows.

- The Draco original code, a 3000 line UCI-Lisp [Meehan 79] program, with no design documentation.
- The absence of a real expert in the Draco implementation.
- The UCI-Lisp manual.
- The Draco manual.
- Neighbors' thesis.
- The absence of a UCI-Lisp environment.

To this situation several notes are worth mentioning to fully understand the problem.

- Although 3000 lines of code is nothing spectacular, this counting is just the Draco kernel, since several of its parts are described in a meta-language which is then transformed to Lisp.
- Although we do not have Neigbhors around, Leite, several years back, performed a partial recovery exercise by documenting the UCI-Lisp version of Draco.
- Although the running of Draco is in Franz Lisp, its code is more difficult to read than the original UCI-Lisp version, since it was produced automatically by Draco [Arango 86].

• The personnel involved in the recovery process were students with little knowledge of Lisp.

Summarizing, the major source of information for the process was the UCI-Lisp code. The efforts for recovering Draco design were supported by an inspection procedure, a prototyping effort using the PC Scheme language, and the JSD tools to register the captured design structure. The next section is oriented to provide details on our proposed approach to design recovery for the situation where the information available is basically old code without a running environment.

III The Multi-Paradigm for Design Recovery

In order to recover the Draco design in the environment as described in Section II, we have used a multi-paradigm process that aims to combine Inspections, Prototyping and the Jackson System Development. The process starts by dividing the object of recovery in **subsystems**. For each subsystem a combination of Inspection and Prototyping is performed and the design is modeled by JSD structure diagrams, and a JSD network model. In the combination of Inspection and Prototyping, inspections are performed and a prototype is produced to validate the conclusions achieved by the inspection process. This process is described by the JSD structured diagram in Figure 2.

Our approach aims to recover designs from artifacts where the basic source of information is the code. Worst, this code can not be executed in the existing environment. Because of this particular characteristic it is necessary that the inspection not only involves the reading of the code but its conversion to a new language, where there is an available translator(**code transformation**). The prototyping process aims to validate the new code as well as the inspection process itself. Using the knowledge acquired in this process a JSD model is constructed in a bottom-up fashion.

Following we present a brief overview of the named paradigms and how they were used in our proposed process.

III - a) Inspections

Inspections was proposed a more than a decade ago as a managerial paradigm for software production. Since Fagan [Fagan 76] first published his paper several projects outside IBM used the paradigm and recently there were reports on the effectiveness of inspections [Ackerman 89]. Briefly, Inspections as proposed by Fagan, are performed by a group of people with well defined roles and activities. The roles are: moderator, designer, coder/implementor and tester. The activities are: Overview and Preparation, Inspection, Rework and Follow-up.

In our design recovery process we have used a variant of Fagan's proposition. The following are the roles present in our inspection process:

Moderator - The person who plans, controls and manages the inspections.

Designer - The person responsible for producing the design recovery.

Coder/Implementor - The programmer responsible for the code transformation.

Tester - The programmer responsible for writing and/or executing test cases or testing the prototype.

The activities we have in our inspection process, see Perform Inspection in Figure 2, are the same as the four originally proposed, but with differences that are worth mentioning. Following we provide a brief description of each activity.

Overview and Preparation - The Moderator first describes the overall area being addressed, the main and the intermediate goals to be achieved. The documentation is distributed to all the inspection participants. The preparation is subdivided in two parts. The first part involves the actors in the coder/implementor roles. Their tasks are to perform the code transformation from the original language to the available language. This task involves a lot of effort in reading the code and the supporting documents to maintain the same semantic in its transformed version. The second part is performed by the actors in the roles of Moderator, Designer and Tester who will read both the old version and the new version of the code to be prepared for the inspection meeting.

Inspection - The coder/implementor describes each translated procedure of the subsystem and the moderator/designer make questions, based on a checklist, to the coder/implementor.

The objective of this step is to find errors and to clarify the doubts in the code. The checklist is a domain oriented list that poses questions at candidate problem areas. The checklist helps the inspector in asking the right questions and not forgetting to ask questions. This checklist with clues on finding errors, may be studied and used during the code examination. All issues raised in the inspection will be addressed in the **rework** and the **follow-up** operations.

Rework - All errors or problems reported in the inspection are corrected by the designer or coder/implementor. The result of this step is a new tranformed code version.

Follow-up - The moderator must verify if all errors and problems discovered in the inspection were resolved in the rework. The result of this step is a valid version of the code transformation.

III - b) Prototype

Prototyping turned out to be a buzz word as researchers and practitioners found out the problems with the waterfall model [Sigsoft 82] [Agresti 87]. Prototypes are used to provide early demonstration of behaviour of a certain artifact in order to shorter the validation cycle. In our approach, prototyping is the test of the transformed code. In this case the prototype in its several versions is the code implemented in the new language.

The process of implementing the prototype is achieved by a manual transformation of the available code into a new language(**code transformation**). This transformation, however, is eased by the mandatory use of the language as close as possible to the original one. In order to test the prototype, outputs produced by the original version, should be compared with outputs of the new code, for the same inputs.

The use of prototyping is essential, in our case, since the inspection uses it to help the test and the understanding of the system being recovered. As shown in Figure 2, if the **implementation** and **test** result in success, the moderator accepts the prototype, otherwise it is rejected and a new **inspection/implement and test** cycle is executed.

III - c) J S D

The Jackson System Development [Jackson 83] [Cameron 86] is a method that emphasizes the modeling activity of the problem addressed. JSD makes a clear distinction between native activities(actions) of a problem being modeled and eventual functions that a software system should perform. Models become more elaborated as the development progresses up to the Implementation stage. The native activities are clustered in entities with their related attributes, which later are transformed in processes. There is no global memory, data is shared between processes by state vectors or data streams. The interactions between these processes will determine the behavior of the model, even before the Implementation stage.

Once the native activities are identified from the bottom-up, JSD is clearly a top-down development method. Our use of JSD is singular. Instead of departuring from a problem analysis, we are departuring from **an implementation**. This characterizes the use of JSD in a bottom-up fashion.

Although we depart from the implementation, there is not enough knowledge to construct the System Implementation Diagram, that is the actual implemented design. The approach we use is to decompose the problem into parts, the decomposition being dependent on the problem, recover each part and then integrate the models produced.

The recovery of each part is the core of our method, the result is expressed in a **SSD** (System Structure Diagram) which is equivalent to the one produced at the Network Stage - Elaboration phase. This diagram is an initial version which should be revised when the models are integrated. The integration should produce the **SID** (System Integration Diagram), which should mirror the closest as possible the real physical design. The final SID and the supporting network model would be the end products of the recovery process.

IV Draco Design Recovery

As discussed in Section II, our objective is to have the Draco machine working in order to study the paradigm. Since available Draco is not supported by our software plataform, we decided to re-engineer the machine. The knowledge embodied in the Draco machine was not completely available in a representation other than code, so we needed to perform knowledge extraction from the code itself.

The multi-paradigm approach just described was an ideal candidate, mainly for its characteristics, which matched the main characteristics present in the Draco case, that is:

a) existence of a team composed of the two authors and two computer science students,

b) need to produce a prototype, and

c) the nonexistence of a design document

In this section we will describe how the suggested approach has been used in the Draco case, stressing the interaction between Inspections and the production of a working prototype. We conclude reporting on the results achieved at this stage of the project.

IV - a) Draco Recovery

The process used on Draco Recovery with the data and resources mentioned in II is best described by the JSD Structured Diagram in Figure 3. This diagram is an instantiation of the diagram in Figure 2 with the actions specialized for the Draco case. Following we will describe each part of the process.

Subsystem Structure

The parts identification were performed by a UCI-Lisp cross-analyzer developed in Scheme. This analyzer shows which procedures are called by one specific procedure and which ones call this same procedure, within a subsystem.

The scope of each subsystem to be recovered in each stage of the recovery development can't become totaly clear initialy, but this first analysis helps the starting of the process. As soon as the work goes ahead we can get more comprehension about the role of each subsystem.

Inspection and Prototyping

To recover each Draco subsystem, first we read the UCI Code, then produced the translated code and next we implemented and tested the prototype until its acceptance.

Perform Inspection

We performed the Draco Inspection in four repeated actions described below and, as a result of this step, we had a Prototype of the subsystem being translated.

- Understand the UCI Code and Codify into Scheme.
- Inspection Scheme Code.
- Correct and Create a new version.
- Validate the version.

These actions are described below, with more details.

Understand UCI Code and Codify into Scheme

The understanding of the statements and controls structures of the concerned languages, through the reading of their manuals, makes it possible to codify each UCI procedure into the new language (Scheme), sometimes even without a clear comprehension about the role of the procedure in the subsystem.

The first step on the recovery development is the **code transformation**, that is, each statement of a procedure in UCI Lisp is converted into Scheme Lisp, keeping the same semantic. The result of this step is documented as showed in Figure 4, which is the **version 0** of the transformations.

This document contains, for each procedure, on one side the original UCI Lisp code and on the other the translated Scheme code. Besides, it contains the explanation of the procedure's role in the subsystem based on the knowledge that we could get at that point in time.

The most important aspect of this transformation is to preserve the correctness of the original procedure. By that we mean that any transformed procedure must have the same property and results of the original one.

In order to make the code transformation possible, some UCI functions, which didn't exist on Scheme, had to be constructed and implemented. In this case we had to implement in Scheme the UCI **DF**, **Catch/Through**, **Some**, **Ttymsg**, **Ldiff**, **Aexplodec** commands. The **Fexpr** facility of UCI Lisp was implemented using a series of Scheme Macros. The same was done for Catch/Through control. These two aspects, one refering to evaluation and scope of variables, and the other to the flow of control were the major problems encountered so far.

Inspection Scheme Code

After we get the transformed version (Figure 4), we can perform the inspection of the transformed code to find errors and to clarify doubts about the translated code.

Four people constituted the inspection team: a Moderator, a Designer and two Coder/Implementor. Three inspections with about four hours each were sufficient to validade the transformed code in the case of the Parser Subsystem.

The time to do inspections and resulting rework was scheduled and managed by attention of the Moderator. At each inspection meeting the Coder/Implementor presented a new code version, documented as shown in Figure 4, which was used to understand the code and to find errors by the Designer and Moderator. An annotated version of this same document is shown in Figure 5. The checklist used to provide suggestions on where and how to detect defects is showed in Figure 6.

Correct and Create a New Version

As the errors and doubts about the code are pointed out by the Designer, they are corrected and explained by the Coder/Implementor who creates a new version of the code.

The translated code of each procedure must be verified and depending on the result that we get in the verification, we can accept the current version or we annotate the errors to be corrected in the new version.

Validate The Version

Once there are no doubts about the code transformations of each procedure in the subsystem, the Moderator accepts the version, otherwise we **Perform Inspection** until the correct version can be obtained.

Implement And Test The Prototype

After all the procedures from a subsystem are verified, they are compiled into the prototype to be tested. Depending on the test results, the new code could be accepted as the final version or it is rejected and will be submitted to new inspections and validations.

Accept The Prototype

The prototype is accepted when it produces the same results of the original code being recovered. The Designer performs the Tester role for the inspection process and he can verify the correctness of the translated code by running the new code with the same input data of the examples we have for execution in the original code, and comparing the results produced by the prototype with those in the samples.

Reject The Prototype

Errors are identified by the Tester and depending on the complexity of the error the prototype may be immediately corrected by the Tester himself or be rejected.

Construct a JSD Network Model

In order to construct the System Specification Diagram, the Network Model, the **lisp procedures** were considered **entities** and the **commands** were considered **actions** . The several processes are linked by **data streams** and **state vectors.** Arguments were considered data streams, and global variables were transformed in an entity named **GLOBALVAR**, from which all global access was described via state vectors. This basic heuristic has allowed us to construct a first version of a model which is to be integrated on a **SID**(System Implementation Diagram) to be elaborated as the final action of the Recovery process (see Figure 3).

The procedures behavior are expressed by **Structures Diagrams** which specifies how their actions are ordered in time.

Integrate the Models

This phase of the Method has been executed until now. As soon as we have the **SSD** of the Draco major subsystems we can start their revision and their integration into a **SID**.

IV - b) Results So Far

The Draco recovery was divided in four stages as we have defined four subsystems, **Parse, Transform, Refine** and **Prettyprint** , shown in Figure 1. These subsystems constitute the four intermediate goals of the Draco work recovery. The Parse subsystem with about 800 UCI lines was recovered. Approximately the same number of Scheme Lisp code lines were generated, compiled and tested. As the result of the errors that was found on the inspections, implementations an test, three versions of the document shown in Figure 4 were created.

Using the Jackson System Development we created **Structure Diagrams**, as shown in Figure 8, for each function that was included in the Parse Subsystem. The composition of all those functions resulted in the **Parse Subsystem Network Model** , shown in Figure 9. The complexity of this Model is due to the inclusion of the parser generator definition together with the core parser code. In Figure 7 there is a detail of Figure 9.

The existence of **Global Variables** in the original design forced us to create a special symbol (marked with an asterisk) in the SSD. This symbol works as an entity from where processes reads data through state vectors.

The Scheme version accepted was finally validated by the the generation of the Draco Parser itself. At the moment, the recovery work is being concentraded on the Transform and Refine subsystems. The experience on Parse subsystem gave us more familiarity with the process as well as with the UCI Lisp, Scheme Lisp and the proper Draco system, such that we expect to have a better performance on the others subsystems.

V Conclusion

We can not afford to throw away our investment in existing code. We need to find out ways to recover the design of existing kwowledge in code form, and re-engineer those designs to make them real reusable assets. That is a hard task. The knowledge is fragmented and the recovering process is expensive. In our research project at PUC-Rio we are facing a situation where it is mandatory to recover and re-design a software artifact, and that was our main motivation in designing a process to recover software design.

Our approach of mixing different paradigms has been effective. Although the manual translation of UCI Lisp to Scheme is not a difficult problem, it is not trivial, specially since Scheme is a lexical scoped Lisp. The adjustment with respect to FEXPR and the CATCH/THROW control, was not so trivial. With that regard, our observations pointed out that the use of Inspections improved the team's understandability of the problem, as well as shortened the conversion cycle.

Collecting productivity data on the environment in which we are operating (graduate and undergraduated students) seems difficult, and we do not claim to have all of the data, but we have observed several positive points on the use of our approach.

First of all, we have managed to structure the work, by using Inspections, as a manageable process. Since using students is very different than using employees or contractors, we list this as an important point. Second, the system we are dealing with is extremely complex and the inspection meeting was decisive to make the full understanding of the recovered subsystem, the Parser, possible. Third, the time spent debuging programs was shortened, since the Inspection meetings took most of the team's time.

With respect to modeling, or documenting the design, our use of JSD has been productive. The construction of the network model of the Parser, demanded as lot of effort, but made it clear for the team how the several parts of this subsystem were put together. Although we did not derive the implementation diagram, of Draco, nor used the model produced for the re-design task, we can observe that the model produced has structured and encapsulated knowledge.

Our future plan is to continue the use of this design recovery process for the Draco code as well as for other cases. As much as we gain confidence on our approach and get feedback from its use, we may improve some aspects of it.

Although we did not produce tool support, besides the trivial cross-analyzer, it is an ongoing project the use of hypertext to support Inspections meetings by making possible that the code be annotated and revised interactively as well as making explicit the linkage with produced JSD documents. One of the authors, Prado, is currently working on a JSD tool that should be integrated with such hypertext support.

VI Acknowledgments

Marcelo Santana and Alexandre Seidl have been developing an excellent job as members of our Draco team. We acknowledge their help and effort in recovering Draco.

VII References

[Ackerman 89] Ackerman, A. Frank, Buchwald, Lynne S., Lewski, Frank H. Software Inspections: An Effective Verification Process. In IEEE Software, May 1989.

[Agresti 87] Agresti, W. In New Paradigms for Software Development, W. Agresti, Ed., IEEE Computer Society, Long Beach, CA 1987.

[Arango 86] Arango, G., Baxter, I., Freeman, P., and Pidgeon C. TMM: Software Maintenance by Transformation. IEE Software, 3(3):17-39, May 1986.

[Arango 88] Arango, Guillermo F. Domain Engineering for Software Reuse, Ph.D. dissertation thesis, University of California Irvine, 1988.

[Baxter 90] Baxter, I. Transformational Maintenance by Reuse of Design Histories, Ph.D. thesis, University of California Irvine. Tech Report 90-36, 1990.

[Biggerstaff 89] Biggerstaff, T. Design Recovery for Maintenance and Reuse, IEEE Computer, July 1989.

[Cameron 86] Cameron, John R. An Overview of JSD. IEEE - Trans. on Software Engineering, SE-12, NO 2, Feb 1986.6

[Fagan 76a] Fagan, M. E. Design and Code Inspections to Reduce Errors in Program Development, IBM SYST J., 1976.

[Freeman 87a] Freeman, P. Software Perspectives Addison Wesley, Reading, MA, 1987.

[Freeman 87b] Freeman, P. Software Reusability. IEEE - Computer Society, March, 1987.

[Jackson 83b] Jackson, M. A. System Development. Prentice Hall International, Inc, 1983.

[Matsumoto 87] Matsumoto, Y. A Software Factory: An Overall Approach to Software Production In Software Reusability, pages 155-156 . IEEE Computer Society Press, 1987.

[Meehan 79] Meehan, J. R. The New UCI LISP Manual, LEA, Hillsdale, New Jersey, 1979.

[Neighbors 84] Neighbors, J. The Draco Approach to Constructing Software from Reusable Components. IEEE Trans. on Software Engineering, SE - 10:564-573, September 1984.

[Neighbors 84] Neighbors, J. A Method for Engineering Reusable Software Systems. In Reusable Software. Addison Wesley, 1988.

[Prieto 87] Prieto, Diaz R. Domain Analysis for Reusability. In Proc. Compsac - 87. Tokyo, Japan, October 1987.

[Seppänen 90] Seppänen, V. Acquisition and Reuse of Knwowledge to design embedded Software, Technical Research Centre of Finland Publication 66 , 1990.

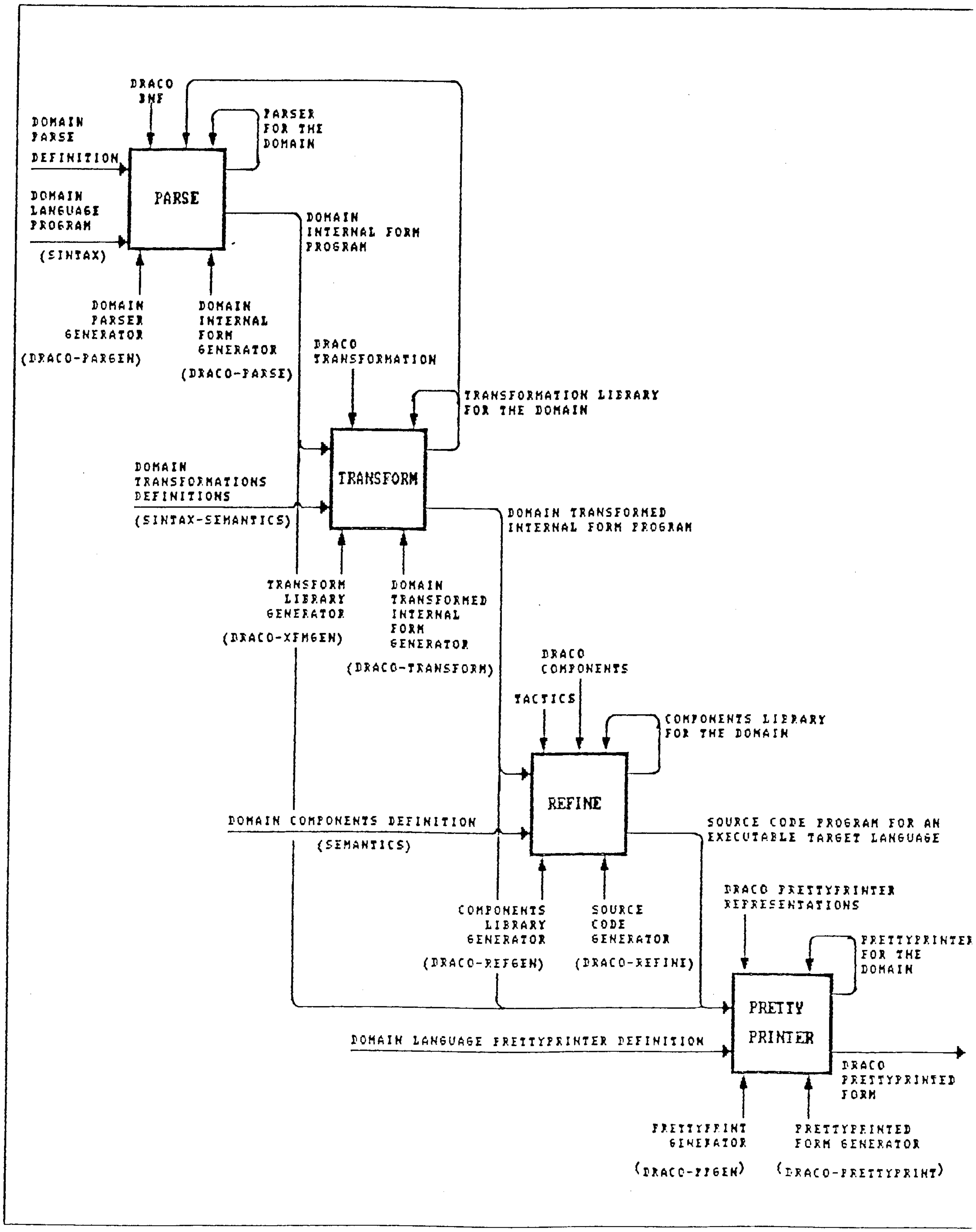

DRACO BNF
DOMAIN PARSE DEFINITION
PARSER FOR THE DOMAIN
DOMAIN LANGUAGE PROGRAM
(SINTAX)
PARSE
DOMAIN INTERNAL FORM PROGRAM
DOMAIN PARSER GENERATOR
(DRACO-PARGEN)
DOMAIN INTERNAL FORM GENERATOR
(DRACO-PARSE)
DRACO TRANSFORMATION
TRANSFORMATION LIBRARY FOR THE DOMAIN
DOMAIN TRANSFORMATIONS DEFINITIONS
(SINTAX-SEMANTICS)
TRANSFORM
DOMAIN TRANSFORMED INTERNAL FORM PROGRAM
TRANSFORM LIBRARY GENERATOR
(DRACO-XFMGEN)
DOMAIN TRANSFORMED INTERNAL FORM GENERATOR
(DRACO-TRANSFORM)
DRACO COMPONENTS
TACTICS
COMPONENTS LIBRARY FOR THE DOMAIN
DOMAIN COMPONENTS DEFINITION
(SEMANTICS)
REFINE
SOURCE CODE PROGRAM FOR AN EXECUTABLE TARGET LANGUAGE
COMPONENTS LIBRARY GENERATOR
(DRACO-REFGEN)
SOURCE CODE GENERATOR
(DRACO-REFINE)
DRACO PRETTYPRINTER REPRESENTATIONS
PRETTYPRINTER FOR THE DOMAIN
DOMAIN LANGUAGE PRETTYPRINTER DEFINITION
PRETTY PRINTER
DRACO PRETTYPRINTED FORM
PRETTYPRINT GENERATOR
(DRACO-PPGEN)
PRETTYPRINTED FORM GENERATOR
(DRACO-PRETTYPRINT)

Figure 1

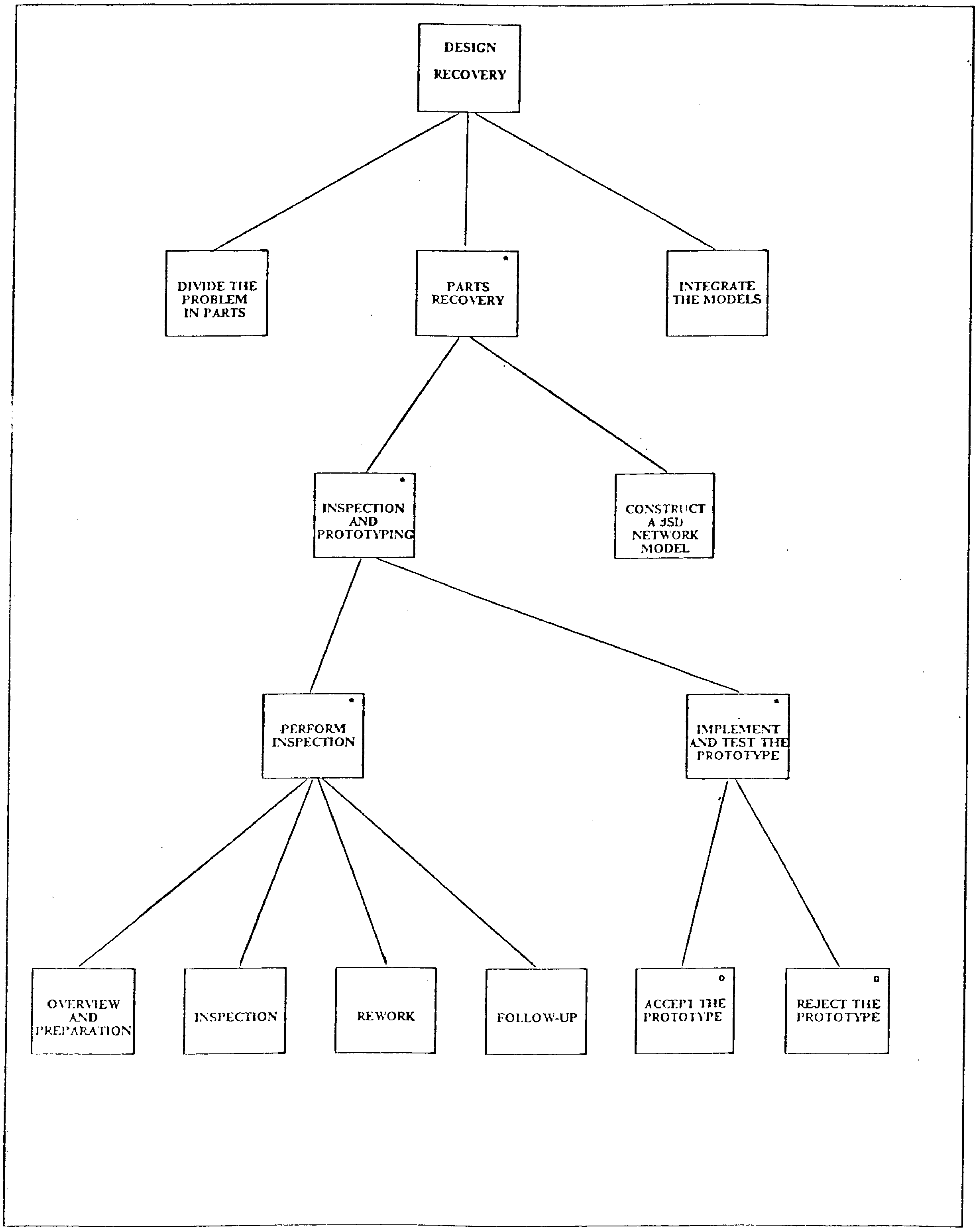

DESIGN RECOVERY
DIVIDE THE PROBLEM IN PARTS
PARTS RECOVERY *
INTEGRATE THE MODELS
INSPECTION AND PROTOTYPING *
CONSTRUCT A JSD NETWORK MODEL
PERFORM INSPECTION *
IMPLEMENT AND TEST THE PROTOTYPE *
OVERVIEW AND PREPARATION
INSPECTION
REWORK
FOLLOW-UP
ACCEPT THE PROTOTYPE o
REJECT THE PROTOTYPE o

Figure 2

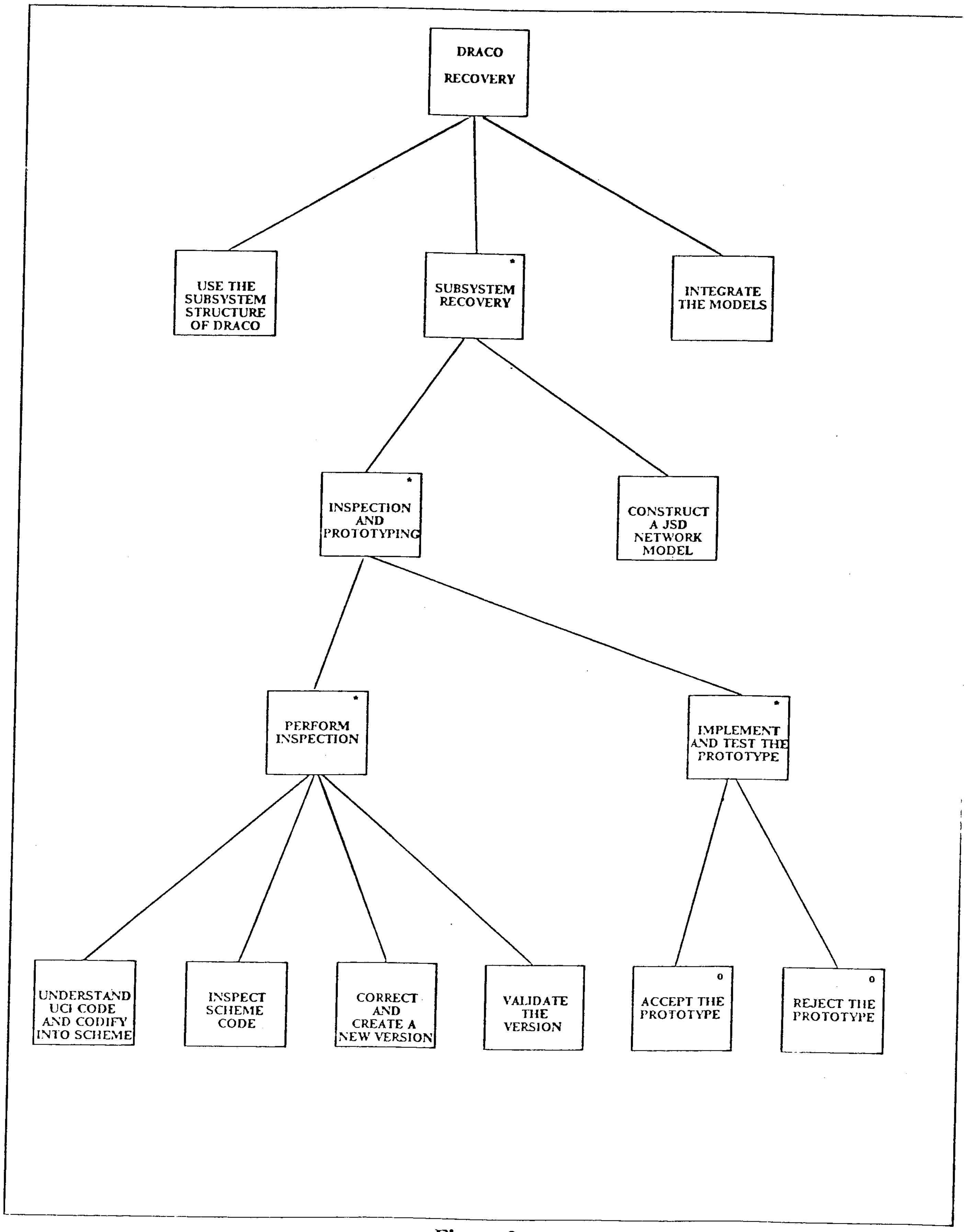

DRACO RECOVERY
USE THE SUBSYSTEM STRUCTURE OF DRACO
SUBSYSTEM RECOVERY *
INTEGRATE THE MODELS
INSPECTION AND PROTOTYPING *
CONSTRUCT A JSD NETWORK MODEL
PERFORM INSPECTION *
IMPLEMENT AND TEST THE PROTOTYPE *
UNDERSTAND UCI CODE AND CODIFY INTO SCHEME
INSPECT SCHEME CODE
CORRECT AND CREATE A NEW VERSION
VALIDATE THE VERSION
ACCEPT THE PROTOTYPE o
REJECT THE PROTOTYPE o

Figure 3

```
(DF Parser-Rule (Exps)
    (Prog (Stk Global-ParseRule)
       (Setq Global-ParseRule (Car Exps))
       (Setq Stk Global-Stack)
       (Cond ((Eval (Cadr Exps))
              (Cond ((Eq Stk (Cdr Global-Stack)) (Return T))
                    (T (TTYmsg T
                              "ERR: rule "
                              Global-ParseRule
                              " succeeded but did not return
                                one PARSER-NODE"
                              T)
                       (Setq Global-ErrorCount
                             (Add1 Global-ErrorCount))
                       (Setq Global-Stack Stk)
                       (Push Global-Stack '(*ERROR*))
                       (Return T))))
             (T (Cond ((Eq Stk Global-Stack)
                       (Return (Metasym-Pvar)))
                      (T (TTYmsg T
                                "ERR: rule "
                                Global-ParseRule
                                " failed but constructed
                                  PARSER-NODEs"
                                T)
                         (Setq Global-ErrorCount
                               (Add1 Global-ErrorCount))
                         (Setq Global-Stack Stk)
                         (Return NIL))))))
```

```
(DF Parser-Rule (Exps)
   (Let ((Stk Global-Stack))
     (Set! Global-Parse-Rule (Car Exps))
     (If (Eval (Cadr Exps))
        (If (Eq? Stk (Cdr Global-Stack))
           #T
           (Begin
             (TTYmsg T
                     "ERR: rule "
                     Global-ParseRule
                     " succeeded but did not return
                       one PARSER-NODE"
                     T)
             (Set! Global-Errorcount (1+ Global-ErrorCount))
             (Set! Global-Stack Stk)
             (Push Global-Stack '(*ERROR*))
             #T))
        (If (Eq? Stk Global-Stack)
           (Metasym-Pvar)
           (Begin
             (TTYmsg T
                     "ERR: rule "
                     Global-ParseRule
                     " failed but constructed
                       PARSER-NODEs"
                     T)
             (Set! Global-ErrorCount
                   (1+ Global-ErrorCount))
             (Set! Global-Stack Stk)
             '()))))
```

PARSER-RULE

This sub-routine takes as argument a list of
a rule name (CAR EXPS) followed by its corresponding code (CADR EXPS).
PARSER-RULE then evaluates the associated code and verifies if it
as expected had generated a one and only one node into the
Global-Stack.If it had been so it just signals true, otherwise, an error
treatment is executed.

Figure 4

```
(DF Parser-Rule (Exps)
    (Prog (Stk Global-ParseRule)
       (Setq Global-ParseRule (Car Exps))
       (Setq Stk Global-Stack)
       (Cond ((Eval (Cadr Exps))
              (Cond ((Eq Stk (Cdr Global-Stack)) (Return T))
                    (T (TTYmsg T
                                "ERR: rule "
                                Global-ParseRule
                                " succeeded but did not return
                                 one PARSER-NODE"
                                T)
                       (Setq Global-ErrorCount
                             (Add1 Global-ErrorCount))
                       (Setq Global-Stack Stk)
                       (Push Global-Stack '(*ERROR*))
                       (Return T))))
             (T (Cond ((Eq Stk Global-Stack)
                       (Return (Metasym-Pvar)))
                      (T (TTYmsg T
                                  "ERR: rule "
                                  Global-ParseRule
                                  " failed but constructed
                                    PARSER-NODEs"
                                  T)
                         (Setq Global-ErrorCount
                               (Add1 Global-ErrorCount))
                         (Setq Global-Stack Stk)
                         (Return Nil))))))
```

```
(DF Parser-Rule (Exps)
   (Let ((Stk Global-Stack))
     (Set! Global-Parse-Rule (Car Exps))
     (If (Eval (Cadr Exps))
         (If (Eq? Stk (Cdr Global-Stack))
             #T
             (Begin
               (TTYmsg T
                        "ERR: rule "
                        Global-ParseRule                  1
                        " succeeded but did not return
                          one PARSER-NODE"
                        T)
               (Set! Global-Errorcount (1+ Global-ErrorCount))
               (Set! Global-Stack Stk)
               (Push Global-Stack '(*ERROR*))
               #T))
         (If (Eq? Stk Global-Stack)
             (Metasym-Pvar)
             (Begin
               (TTYmsg T
                        "ERR: rule "                      2
                        Global-ParseRule
                        " failed but constructed
                          PARSER-NODEs"
                        T)
               (Set! Global-ErrorCount
                     (1+ Global-ErrorCount))
               (Set! Global-Stack Stk)
               '0)))))
```

Observações:

1) Qual a estrutura de EXPS ?

2) Qual a condição para o erro tipo 2 ?

3) Qual a diferença entre Global-Stack
e (CDR Global-Stack) ?

PARSER-RULE

This sub-routine takes as argument a list consisting of
a rule name (CAR EXPS) followed by its corresponding code (CADR EXPS).
Parser-Rule then evaluates the associated code and verifies if it
as expected had generated a one and only one node into the
Global-Stack. if it had been so it just signals true, otherwise, an error
treatment is executed.

Figure 5

CHECKLIST

Completeness

1. Are all sources of input identified?
2. Are all types of outputs identified?
3. Are all procedures that compound the subsystem defined?
4. Are all the parameters defined in each procedure?
5. Are all required parameters passed correctly?
6. Are the data structures of each local and global variables defined?
7. Had all possible side-effects been accounted for?

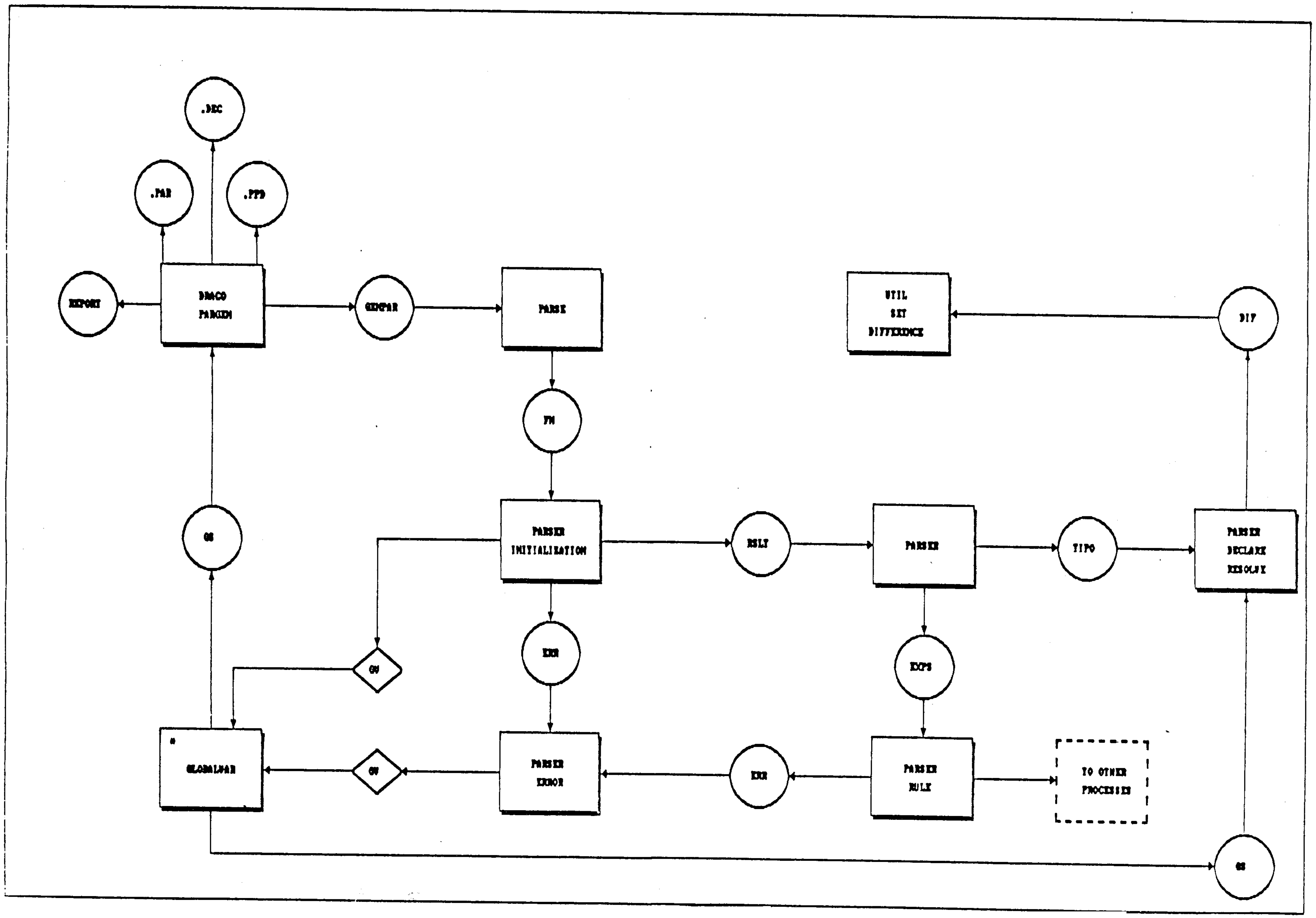
Ambiguity

1. Are all special terms clearly defined?
2. Is the scope of the globals and locals variables kept on?

Consistency

1. In the Selection and Repetition the error condition is correctly tested?
2. Are correct the variables used for test?
3. All the properties of the control structures of the original code are preserved on the new translated code?
4. Are all the functions correctly defined with their properties?
5. How are the structures of the S-Expressions?

Figure 6


REPORT
.PAR
.DEC
.PPD
DRACO PARGEN
GENPAR
PARSE
FN
UTIL SET DIFFERENCE
DIF
GS
PARSER INITIALIZATION
RSLT
PARSER
TIPO
PARSER DECLARE RESOLVE
GV
ERR
EXPS
GLOBALVAR
GV
PARSER ERROR
ERR
PARSER RULE
TO OTHER PROCESSES
GS

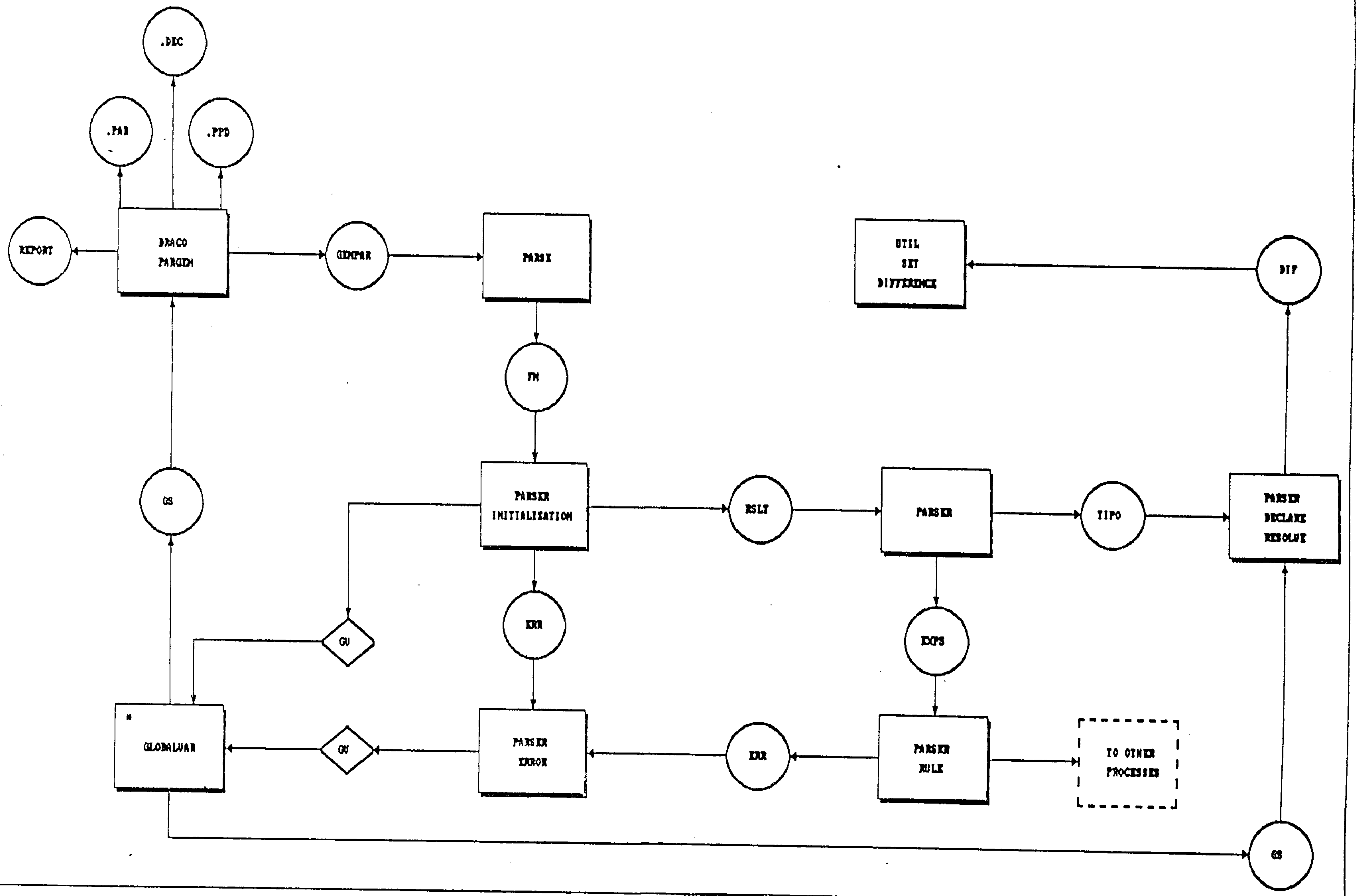

.DEC
.PAR
.PPD
REPORT
DRACO PARGEN
GENPAR
PARSE
FN
UTIL SET DIFFERENCE
DIF
GS
PARSER INITIALIZATION
RSLT
PARSER
TIPO
PARSER DECLARE RESOLVE
GV
ERR
EXPS
*
GLOBALVAR
GV
PARSER ERROR
ERR
PARSER RULE
TO OTHER PROCESSES
GS

Figure 7

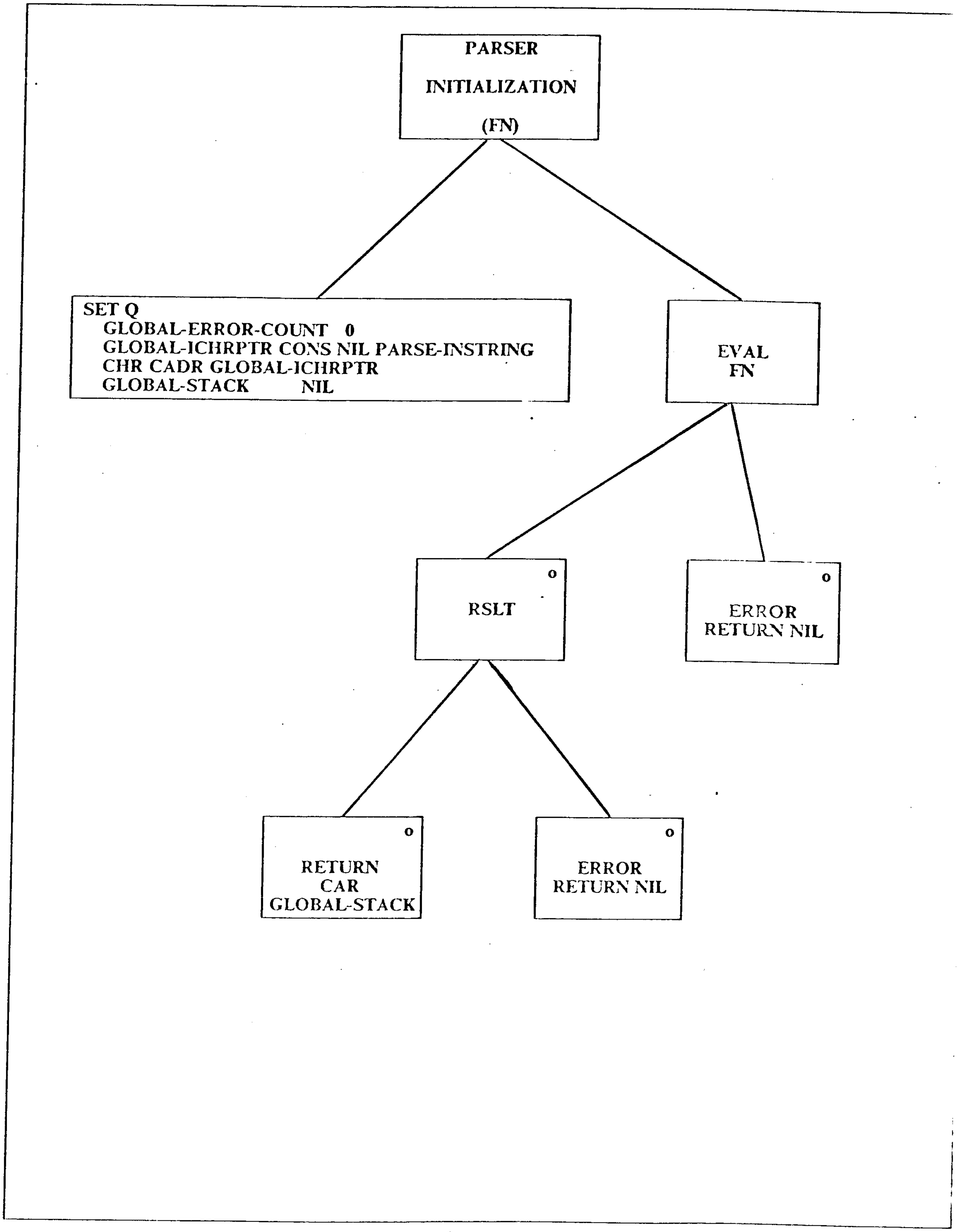

PARSER
INITIALIZATION
(FN)
SET Q
GLOBAL-ERROR-COUNT 0
GLOBAL-ICHRPTR CONS NIL PARSE-INSTRING
CHR CADR GLOBAL-ICHRPTR
GLOBAL-STACK NIL
EVAL
FN
o
RSLT
o
ERROR
RETURN NIL
o
RETURN
CAR
GLOBAL-STACK
o
ERROR
RETURN NIL

Figure 8

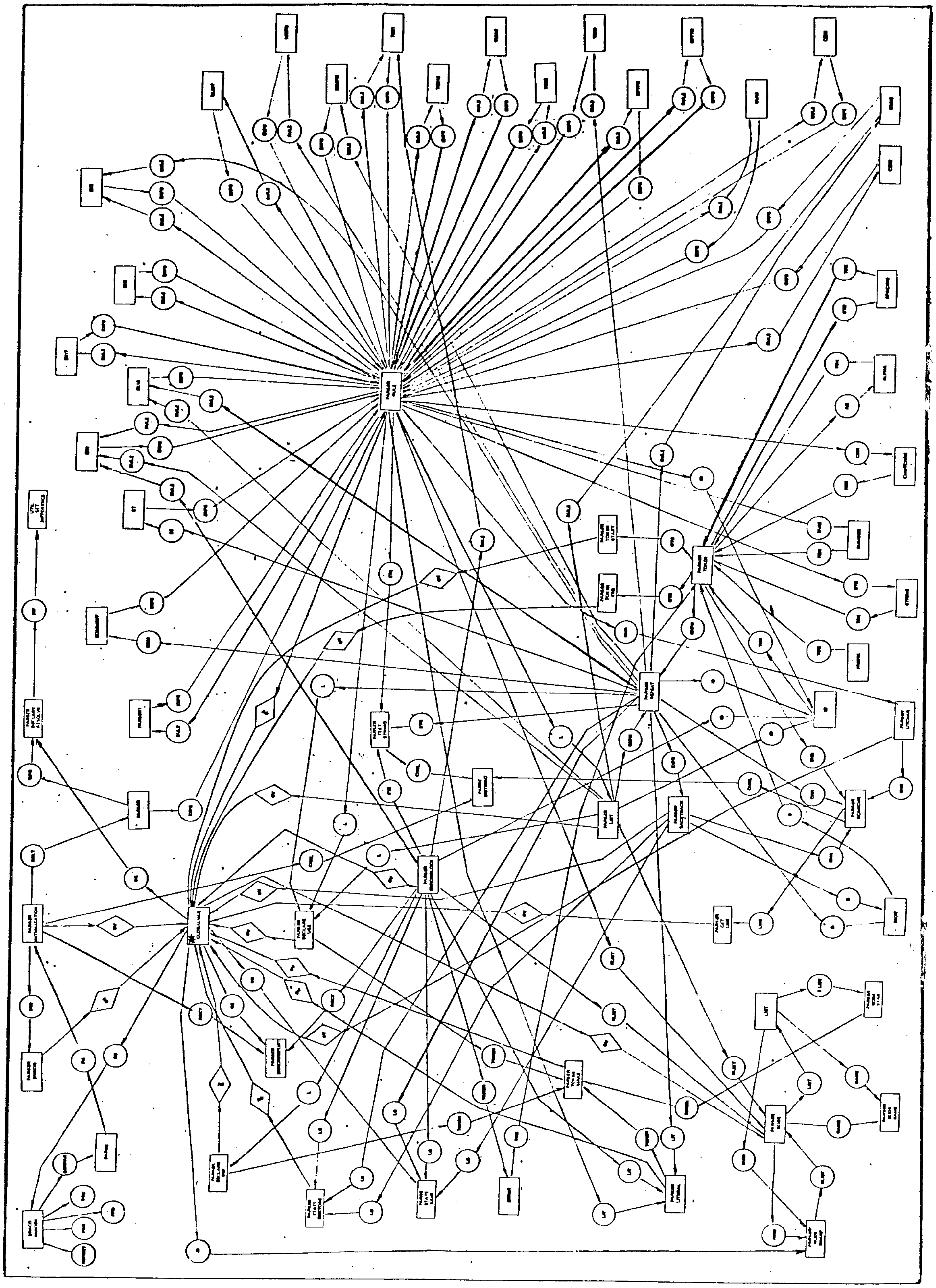


Figure 9