IESEM: Integrated Environment for Software Evolution Management

Gerardo Canfora*, Filippo Lanubile°, Giuseppe Visaggio°

* Dipartimento di Ingegneria dell'Informazione ed Ingegneria Elettrica,

University of Benevento, Italy

° Dipartimento di Informatica,

University of Bari, Italy

Abstract

Software evolution has not a common paradigm which practitioners can adhere to. On the contrary, there is a wide range of models, methods, techniques, and tools which are selected according to the specific task, the application domain, the professional experience and the organizational culture. We argue that different approaches and technologies may be combined into a unique platform to satisfy the needs of software systems which evolve over long periods of time.

This paper presents the Integrated Environment for Software Evolution Management (IESEM) which includes software repositories, reverse engineering tools, rationale capture tools, software measurement tools, and a user-friendly interface. It can manage heterogeneous systems characterized by various design methods and programming languages. IESEM is based on a central repository which stores software engineering artifacts, program code, design and implementation decisions in the form of a traceability graph. The repository stores also software

measures computed both from programs and external CASE repositories. Measures are used to control software degradation during its evolution and to support decisions based on quality factors.

The key concepts of IESEM, its design, and implementation are presented. The use of IESEM during development and maintenance is discussed. A case study shows IESEM's effectiveness in performing maintenance tasks.

1. Introduction

Software maintenance lacks a strong theoretical foundation. Although important key concepts have been proposed in the past, there is no general agreement on the definition, the classification of maintenance activities, the modeling of the maintenance process, and even on name itself. The term *evolution* is often used in place of maintenance to express the continuous improvement of software over time [26]. Software systems undergo continuous and gradual change like living beings. Some species survive but others become extinct. To increase the fitness of old systems and thus lengthen their life, reverse engineering and reengineering have become very popular in the last few years. While reverse engineering is an examination process which produces abstractions useful for understanding the existing software, reengineering also involves the alteration of the subject system to renovate it [7]. But abundant documentation and models are not themselves a solution to the maintenance problem. If maintainers do not find quick answers to their questions and do not see a clear correspondence between abstract models and programming objects, they lose their trust in documents and make changes starting from the code only. This is not an acceptable situation because understanding programs requires more information than code alone can offer.

At present, there is no a correct way, agreed on by everybody, to maintain software systems [23]. Despite this lack of theoretical basis, we believe it is possible to build useful technologies and integrate them in platforms for making experiments.

The Integrated Environment for Software Evolution Management (IESEM) is a general framework which combines different approaches to the software maintenance problem. Our

project originated from a number of insights about software maintenance gained from our experience and previous research work.

- The focus on source code is inadequate [3], [4]. Modification should also involve the other life cycle work products, for example requirements and design.
- Software systems are not homogeneous single-version programs [15], [33]. The wide variety of technologies, CASE tools, programming languages, development methods and standards is often incorporated in multi-version systems (systems existing in many experimental versions) or in heterogeneous systems (systems characterized by mixed approaches). Design for change can be achieved by specifying the core of a solution (essential design) before the introduction of constraints in the form of specific programming features (language-oriented design).
- Software is not made up of isolated objects [20], [34]. Software objects are closely related each other. Relationships exist among objects from the same life cycle phase (internal traceability) and among objects which belong to work products of different life cycle phases (external traceability). Understanding an unknown program and locating with precision what parts in are involved in a modification require the explicit representation of the links among programming concepts in the implementation domain, solution concepts in the design domain, and problem concepts in the analysis domain.
- Critical design knowledge is lost during development [22], [36]. Design rationales are usually contained in the agents rather than in the deliverables of the software life cycle. So what we can see is only the effect of decisions on the final product. When and why decisions were made and what consequences they had, are questions maintainers need to ask to achieve a clear understanding of a software system.

- Software degradation cannot be controlled without measurement [21], [27]. Measurement captures unambiguous information about the properties of things. It requires clear goals and provides evidence that goals have been achieved. For example, a goal of preventive maintenance may be the localization of the areas within a system which have high risks of degradation.

IESEM embodies the requirements made by the lessons above by combining different technologies such as software repositories, project databases, reverse engineering tools, bridging tools, design rationale capture tools, measurement tools, and data analysis tools. Section 2 describes IESEM's architecture and illustrates the current status. Section 3 shows how IESEM fits into the development part of the software life cycle, while Section 4 discusses its use to perform maintenance tasks. Related work is compared with IESEM in Section 5. Finally, Section 6 concludes the paper.

2. IESEM architecture

IESEM consists of a repository and four major components: Translator, Extractor, Traceability Support System and System Quality Monitor, as shown in Figure A.

The IESEM repository retains a record of the design process, including both software objects and rationales. The Translator converts design information from commercial CASE tools to the IESEM conceptual model and back. The Extractor uses a set of language-oriented design models to transform source code files into objects and relationships. Both the Translator and Extractor measure intramodular and intermodular attributes of software components. The Traceability Support System provides functions for capturing decision rationales, linking decisions to objects,

and managing both internal and external traceability. The System Quality Monitor enables the collected measures to be analyzed with respect to some quality criteria.

2.1. IESEM conceptual model

All information on software systems is part of the IESEM repository. We use enhanced entity-relationship (EER) diagrams to describe the repository at a conceptual level, as viewed by IESEM tools and users. The graphical notation is the one described by Elmasri and Navathe [11] except for the representation of the generalization/specialization relationship, which has been simplified by drawing a short cross-line instead of a circle with constraint notes.

The design of the conceptual model will be presented in various steps, following a top-down method from the main level to the detailed ones. Entity types may be duplicated in some EER diagrams as an effect of the decomposition.

In the top-level EER diagram, shown in Figure B, a system is viewed as a collection of subsystems and software objects. The design of the repository handles more than one system, methodology and programming language. The S*oftware Object* entity is specialized into three disjoint subclasses according to the life-cycle phase: *Requirements Specification*, *Essential Design*, and *Language-Oriented Design*. Requirements specification describes a problem specifying the properties that solutions must satisfy, essential design describes a software solution as a highly portable architecture, and language-oriented acts as an implementation blueprint which makes provision for all the limitations or housekeeping constructs of the programming environment. The next specialization level of the diagram is in accordance with the specific analysis methods, design paradigms and programming languages used in the software environment. Further specializations lead to typed software objects which characterize the

software artifacts. These objects are linked by relationship types expressing the dependencies between objects of the same life cycle phase (internal traceability). For the sake of brevity we only partially show how the entity types are related to each other. Figure C and Figure D, respectively show the EER diagrams which model the essential design via functional decomposition and the language-oriented design with C and embedded SQL. Other EER diagrams may be seen in a previous paper [24].

The IESEM repository also handles links between objects of different life-cycle phases (external traceability). The links are enriched with the decisions which have a role in the transformation from a higher to a lower abstraction level, as shown in the EER diagram of Figure E. There are two kinds of decisions: design and implementation decisions. Design decisions describe the mapping of software objects from the requirements specification to the essential design, while implementation decisions describe the next transformation from the essential design to the language-oriented design. If no alternative solutions have been investigated then software objects may also be directly linked by the *derives* relationship type. To describe the structure of the space of all design alternatives and to provide a historical record of design information, four recursive relationship types are defined on the *decision* entity. The g*eneralizes* relationship type describes a generic kind of decision where the problem, the alternatives and the solution are specified without linking the decision to software objects. The related decisions are defined as instances of the generic decision, inheriting its attributes and adding the specific links with the software objects. The *causes* relationship type links two decisions when the problem to be solved by the latter decisions (effect) is provoked by the choice in the former decision (cause). The *justifies* relationship type is a weaker tie between two decisions because the justification of the latter decision is provided by the former decision. The *succeeds* relationship type links two decisions

which have been taken in different times to the same problem. The latter decision (successor) is a new version of the former decision (predecessor) and differs as regards the adopted solution.

Figure F shows a simple decision with its attributes. The first two attributes, *id* and *project* identify uniquely a decision in the IESEM repository. The attribute *author* gives the name of the person responsible for the decision. The attributes c*reation date* and l*ast update* contain timestamps which record historical information on decisions. The attribute *status* determines whether the decision is waiting, active, or old. The attribute *prob desc* contains a textual description of the problem and the attribute *prob type* classifies the problem according to a taxonomy of design decisions [36]. *Alt* is a compound iterative attribute which describes the set of available alternatives. The field *alt.id* is a relative identifier of the alternative, while *alt.desc* informally describes the potential solution. The attribute *sol* shows the alternative which has been selected as solution. The attribute *just desc* explains the choice using natural language, and the attribute *just type* catalogues the justification with respect to quality factors, design constraints, or previous decisions.

Similarly, there is a different set of attributes for each software object type. The relationship types may have attributes, too. Apart from descriptive attributes, code and design metrics are also represented as attributes, once they have been calculated. For example, in a C language-oriented design, the measures of size and control flow complexity describe properties of the entity types *function*, *source code file*, and *executable program*, while the measures of coupling describe the properties of both entity type *executable program* and relationship types which aggregate functions in source code files and those in executable programs.

2.2. The Translator

The Translator is a bridge technology tool which integrates in IESEM the commercial CASE tools used for forward engineering. The Translator has three main functions: Export, Import, and Metric Computation

The first function, Export, is used to generate graphical information which shows the internal structure of programs. The user can view different useful abstractions of programs, according to the diagrammers provided by the CASE tool. For example, we can display the structure chart of an entire program or the action diagrams of a single module. The Export function converts a language-oriented view of the IESEM repository to the format required by the CASE repository.

The second function, Import, populates the IESEM repository after forward engineering tools have been used by developers. In our example, the models produced by an analysis CASE tool may be converted to a specification schema of information engineering or another kind, depending from the analysis method used by the CASE tool. Artifacts produced by the design CASE tool may be converted to an essential design schema of functional decomposition or object-oriented kind, depending on the design paradigm supported by the design tool. If the design CASE tool is used at a lower abstraction level, the software structure can be directly mapped to a language-oriented schema of C kind, COBOL or other, depending on the target language supported by the CASE environment.

The third function, Metric Computation, works when the Import function is activated. Pseudo-code and design metrics are computed and stored in the IESEM repository. The pseudo-code metrics are computed by parsing the procedural detailed design stored in the CASE repository. The computation of design metrics, on the contrary, has the IESEM repository itself as input, because it captures all the necessary information about the relationships between modules.

2.3. The Extractor

The Extractor is a reverse engineering tool which produces structural information, according to a language-oriented schema, and computes software metrics. The function of the Extractor is similar to that of a compiler: it analyzes the source code of some high level programming language, with the difference that code generation is replaced by database generation.

The Extractor works in five steps: the lexical phase, the syntax phase, the semantic phase, the storage phase, and the design metrics computation phase. During the lexical phase, data related to the lexical analysis are collected, i.e. operators specifying actions or operands representing data. In the syntax phase, objects which are recognizable by a syntax analysis are produced, including instances of entity types, calling relationships, and component declarations. The semantic phase builds data structures to produce data flow information such as the *uses* and *modifies* relationships between functions and variables. In the storage phase, all entity and relationship types which characterize one language-oriented schema, are stored in the IESEM repository. Finally, the design metrics computation phase uses the structural information, which has previously been stored in the IESEM repository, to compute intermodular attributes.

The Extractor was built with the help of Lex and Yacc [30]. For each programming language, its BNF is converted to a Yacc description, while the tokens are defined using regular expression in Lex format. The resulting specification is a parser for the target programming language. The specification is augmented by the ability to compute metrics and the function calls for database manipulation.

2.4. The Traceability Support System

The Traceability Support System (TSS) is a tool for capturing rationales. The TSS also enables captured decisions to be linked with software objects and the resulting traceability graph, called *model dependency descriptor* [8], to be navigated. The TSS has a layered architecture consisting of: (1) TSS core, (2) services layer, and (3) interaction layer.

The TSS core provides database schemes for recording decisions and linking them to the software objects in the IESEM repository. It is also responsible for search mechanisms, providing a query language for the retrieval of decisions, software objects, and their mutual relationships. The TSS core is the only layer which depends on the DBMS used to manage the IESEM repository.

The services layer provides operations for organizing information about decisions and software objects, and facilities for tracing objects inside the model dependency descriptor. Manipulation services, summarized in Table 1, are used to build and modify the decision nodes of the model dependency descriptor. Linking services, in Table 2, are used to create or remove edges among software objects and decisions. Finally, the navigation services, in Table 3, are used to trace related decisions from software objects and back.

A decision may be in waiting, active, or old status. A *waiting* decision exists but is isolated or partially linked. The waiting status represents an incomplete specification during constructive design or modification. After all links have been added, the decision becomes *active* by an explicit command (Connect) and the links are consolidated. In the active status, no modifications can be made to the model dependency descriptor. The active status represents a stable situation where the decision is used to support program comprehension and impact analysis. A decision

becomes *old* when it is replaced by a new one. The history of software evolution is preserved by storing old decisions and linking them to their successors.

The interaction layer provides a graphical user interface to work interactively with TSS. A user may invoke the TSS services through a set of push buttons in the bottom area of the screen. Decisions may be interactively retrieved by means of various search options. The *identification* option enables the choice to be made from a list of existing identifiers. The *date* option selects decisions from or up to a given date. The *keyword* option looks for decisions using their structured part. The *pattern* option searches for strings inside the descriptive fields. The *role* options select decisions according to the role that a participating decision plays in each relationship instance, for example generic, instance, cause, effect, and so on. Finally, the *isolated* option finds any unconnected decision in the model dependency descriptor.

2.5. System Quality Monitor

The System Quality Monitor (SQM) is a tool which incorporates the three main quality functions [38]: assessment, control, and prediction. Quality assessment provides a relative comparison of the quality of software components. Quality control identifies software components whose quality values exceed standard quality thresholds or degrade over time. Quality prediction forecasts the quality of software components using measures which are available early in the software life cycle.

In each case, a model of quality is used to define the association between the external and internal attributes [14] of software products in the IESEM repository. Although external attributes, such as maintainability, reliability or usability are the ones that most software people need to know, they cannot be measured directly. Indirect measures of internal attributes, like size,

modularity, or coupling are needed to measure external attributes and the most general attribute, quality.

The System Quality Monitor uses a lattice model to decompose the external attributes into measurable terms. Figure G shows the quality model for a modular component, having maintainability as the main attribute of interest. The model has been adapted from the attribute hierarchy of Esteva and Reynolds [12]. The second level is made up of internal attributes (coupling, control flow structure, data structure, size, and documentation) which are believed to have an impact on maintainability. A further level of decomposition associates the internal attributes with indirect or direct measurable attributes which are stored in the IESEM repository. In the McCall *et al.* quality model [32], the first level attribute is called *quality factor*, the second level attributes *quality criteria*, and the leaf nodes *quality metrics*.

The quality model can be fixed, if a user agrees on the decomposition, or defined by the user himself if he wants to identify specific lower metrics and relationships between these. The leaf metrics must be stored in the IESEM repository as attributes of some entity or relationship type.

The SQM enables a user to select an entity of interest and an attribute of the entity. The result is a metric vector made up of the quality metrics which decompose the selected attribute in the quality model. For example, if the attribute *maintainability* of a C source code file is selected, then the metric vector will be composed of the following metrics: information flow [19] (*IF*), McCabe's cyclomatic complexity [31] (*vG*), number of distinct operands [18] (η_2), total occurrences of operands [18] (N_2), volume [18] (*V*), length [18] (*N*), total lines of code [13] (*TLOC*), lines of code [9] (*LOC*), and comment density [13] (*CD*).

The IESEM user can visualize the metric vector and build different analysis models for interpreting data. SQM has no built-in analysis capabilities but it interfaces with external tools with different underlying analysis techniques.

2.6. Status

IESEM runs under Microsoft Windows on an IBM compatible PC. The IESEM repository has been implemented as a multidatabase and it is processed by the Gupta's SQLBase relational database system. The Translator and Extractor have been written in C language and use the SQLBase C Application Programming Interface (API) to communicate with the database server. To date, we are able to translate the ADW Encyclopedia, a CASE repository of Knowledgeware, and to extract programs written in C, COBOL, Pascal, TurboPascal, dBase, and Clipper. The Traceability Support System and the System Quality Monitor have been developed using Gupta's SQLWindows, a graphical database application development tool. The System Quality Monitor exports data towards SAS System (statistical analysis), C4.5 (decision trees), BrainMaker (neural networks), and HNeT Discovery Package (holographic networks).

3. Using IESEM during development

IESEM does not alter the classic software life cycle which views development as made up of the analysis, design and implementation phases. IESEM augments each of these phases with new activities to accumulate knowledge in the form of design rationales, traceability links, and software metrics. The assumption is that the added development effort for the new activities will be repaid during the software evolution. A prerequisite for the development environment is the use of CASE tools for analysis and design because the manual data entry of software objects in the IESEM repository is an intolerable burden for developers.

Table 4 summarizes the high level tasks during each phase of development. The IESEM repository is filled with successive increments. After the analysis has been completed, the specification schema is imported from the analysis CASE tool using the Translator. Then the first decisions on design issues are created and linked to the specification objects using the Traceability Support System.

After the design has been completed, the Translator is again used, this time to import the essential design from the design CASE tool and compute design metrics. Then, using the Traceability Support System, design decisions are linked to the deliverables of the essential design, creating instances of generic decisions when necessary. At this stage, the design decisions in the model dependency descriptor can become active.

In the implementation phase, primary implementation decisions are edited but left hanging. After the program has been codified, the language-oriented design is imported and code metrics are computed by means of the Extractor. Then, using the Traceability Support System again, the remaining links and decisions are created and the implementation decisions connected.

The System Quality Monitor is run in the design and implementation phases to support software verification. Having chosen a quality attribute of interest, critical software components are identified by looking at software measures whose values are large compared with the mean for the system as a whole. Measures can also be exported to be analyzed by external classifiers and predictors which incorporate more sophisticated evaluation techniques for quality control and quality prediction. As a result, decisions can be revised by exploring the effects of a tentative solution, discovering new alternatives, or choosing alternatives which were previously discarded. This has a major implication for the maintenance process because degradation may be prevented by reviewing a small subset of the system with a consequent saving of resources.

4. Using IESEM during maintenance

With respect to development, maintenance is complicated by the additional constraint due to the existing implementation of the system. If information about the system is ambiguous, lost or false, owing to degradation of the system, the maintainer cannot understand the software system and the change may be unreliable or, worse, unrealizable. We first show how IESEM fits into the software maintenance process and then discuss a case study where IESEM has been used to support software maintenance.

4.1. Software maintenance model

A model of the maintenance process starts from a request for change and finishes with the modification made. Table 5 summarizes the activities which occur during the maintenance process, together with the IESEM tools which support the maintainer task. Two important features characterize this model. The first is that change localization is made before its implementation. The availability of the Traceability Support System encourages this pre-change impact analysis. The second feature is the role of metrics as a controlling mechanism both to manage the requests for change and to monitor system quality after each change. Each task is discussed in the following.

- *Assess a request for change*

The goal of change is classified as corrective, adaptive, or perfective [28], to verify its coherence with the global maintenance policy. For example, in a maintenance environment with a huge backlog of maintenance requests, corrective changes could have priority over adaptive, and adaptive changes over perfective. Software measures may help to prioritize the change requests for each category by highlighting the critical areas in the software system where

modifications are more difficult to perform or preventive intervention may be recommended [37]. The System Quality Monitor assists this task by enabling software measures to be investigated with a specific quality factor in mind.

- *Make a hypothesis of change*

This task involves the understanding of the system being changed. One reason why program understanding is so difficult is that usually the only information the maintainer possesses comes from the source code. Even if design documentation is present it is often informal and not coupled to the source code. If, on the contrary, the design rationale has been recorded, the maintainer may reconstruct the design history, understand what critical decisions were made, what alternatives were analyzed and the reason why one alternative was preferred to the others. The original designer may not have considered the best solution, or have discarded it for some reason, or the context could have changed so that the decision should be re-analyzed. The Traceability Support System has a number of search mechanisms for investigating this accumulated knowledge and allows a maintainer to find the relevant information, even in a large system.

- *Analyze the impact of change*

This task determines what parts of the software system are affected by the proposed change. This involves not only source code but also work products of the design and analysis phase. Traceability supported by explicit links among software artifacts and decisions helps to find what should be updated. The navigation services of the Traceability Support System enable impact analysis to be performed before the change. A maintainer may evaluate the extension of the impact and assess the effort and the activities to implement the change.

- *Implement the change*

It is important to update all the supporting documentation when implementing a change. The IESEM repository, too, must be updated by running the Extractor for the new source code, the Translator if the specification or design have been modified, and the Traceability Support System to update the decisions and traceability links. Software measures should be newly computed and recorded in the IESEM repository.

- *Revalidate the program*

  Before delivering the modified software, the changed parts are tested against new requirements and the overall software system is subjected to regression testing. Besides testing, software measures are checked to see if the new version is less maintainable as a result of the change, in which case some contrasting action must be taken. The System Quality Monitor enables the successive versions of a software system to be compared and warns the maintainer if a degradation is occurring.

4.2. An example of use

We now show the use of IESEM in an actual scenario and explain how a modification task was performed with help of traceability links, decisions, and software measures.

After having developed the first version of the Extractor, we tested it with Pascal programs of about 4000 LOC. Unfortunately, the computation of the structure metrics required fourteen hours compared with the forty-five minutes spent on the first four phases. This took so long that users avoided this option when starting the Extractor. A perfective change was required with the goal of improving the performance of the structure metrics computation.

First, we used the System Quality Monitor to look at software measures of the Extractor subsystem responsible of the structure metric computation. There were no outliers to discourage

the acceptance of the request. To diagnose the change we questioned the Traceability Support System for any decisions neglecting the efficiency factor. We found an implementation decision, shown in Figure H, about retrieval of module indirect flows needed to compute the information flow metric. The solution consisted of using a nested SQL query with ease of implementation as motivation. An alternative was to break the query down into two parts and to process the intermediate results using an iteration structure. We ran the two alternative implementations interactively to verify their performances. The former alternative took 157 seconds whereas the latter only 7 seconds. Since this query was executed twice for each software module, its effect on program performance was onerous, so we decided to choose the second alternative, which had previously been discarded. Impact analysis was performed invoking the TSS services *Propagate Backward* and *Propagate Forward*, starting from the decision. The decision was found to be connected in input to one module and in output to a C function and to one query, as shown in Figure I. The scope of the change was very limited and so its implementation presented no problem. Implementing the change required running the Extractor to incorporate the new version and the TSS to update the decision and link it with the new C objects. As a result of the modification, the response time with the sample Pascal program dropped to one hour and thirty-five minutes. Comparing the software measures with SQM, before and after the modification, we recognized a loss in maintainability of the module because the modification had complicated the module implementation with new data and control structures, but the overall maintainability remained stable as only one module had been changed. Table 6 summarizes the software metrics which decompose the attribute maintainability before and after the change.

5. Related work

Comparing IESEM with other work, there is considerable intersection with many existing approaches to supporting the evolution of software: program abstractors, project databases, design rationale capture tools and measurement tools.

Program abstractors, for example CIA [6], DATA_tool [5], or CARE [29], parse the program text and store the program structure information in a database according to a conceptual data model for the target programming language. Abstract views are built to perform dependence analysis. The cost of creating information is low because the task is fully automated but these tools support only internal traceability at the code level. Since changes usually need to be done as quickly as possible, there is the risk that modifications are applied as bad-documented patches which ignore the other software artifacts for timeliness reasons. As a consequence, further changes of the system will be difficult and error-prone.

Project databases, including PACT [39], ESE [35], or DIF [16] store information about work products representing documents which have been produced in various phases of the life cycle. External traceability is supported so that impact analysis may be performed on the whole documentation base. But in many cases, maintainers spend their efforts trying solutions which had been already investigated and discarded for same reason. Thus, maintenance requires not only to know what a program is intended to do but also why a program is the way it is.

Design rationale capture tools keep track of the reasoning behind the current status of a software system, thus providing a communication mechanism among designers and maintainers. Many proposals have been appeared in the last years, for example OSC [2], D-HyperCase [41], SYBIL [25], MACS [10, 17], or DRCS [22]. Although there are differences as regards the reasoning model, the computational mechanisms and the relationships between decisions and software artifacts, all provide information that, if kept up to date, is precious for software comprehension,

modification and reuse. What distinguishes our work with IESEM is viewing decisions as part of the link which enable the transformations of life cycle work products to be traced.

Moreover, IESEM provides a basis for a quality-driven decision making process, where decisions are justified by measurable quality factors. The QDV tool [40] already suggested a similar relationship between design rationales and quality characteristics but it does not provide automatic extraction, recording, and analysis of software measures. Few commercial tools extract measures from programs and they usually ignore design metrics. CASE tools do not yet provide automatic metric collection of the designs in their repositories. To fill this lack, IESEM measures code and design metrics both from programs and CASE repositories.

Like many integrated packages, IESEM is not superior to each tool mentioned above in its specific field. What distinguishes IESEM is the *holistic* view of software evolution, which implies the integration of tools, methods, and standards in a unique environment where all the components work together to support the maintainer tasks.

6. Conclusions and future work

IESEM has been developed to assist the maintenance of software systems. A set of tools, working upon a common database, accomplishes this mission.

IESEM incorporates different design methods and programming languages. Decision rationales are part of the IESEM repository and are used together with software artifacts to build a traceability graph. Software metrics are computed from programs and CASE representations and stored in the repository to be further analyzed according to a quality model. They are used to evaluate software degradation during its evolution and to support decisions based on quality factors.

IESEM has been used successfully to track the development process of middle size systems developed by our students and to support modifications. Preliminary results show that the use of IESEM allows the maintainer to make correct change proposals rapidly and to isolate precisely the system portion affected by modification. The recorded decisions sustain the understanding effort by providing the rationale behind the design. Software metrics were used to select components urgently needing restructuring or to send warnings if the initial structure becomes degraded. Overhead on the design process was significant because of capturing decisions and specifying links. While the decision-capture task cannot be minimized, we intend to automate the derivation links between software objects.

Controlled experiments are in progress to investigate the effects of IESEM on maintainers' performances. At the moment, experimentation involves university students [1] but we intend to move towards professional programmers.

IESEM is currently a stand-alone prototype on a personal computer platform. To have a significant impact in an industrial environment, the technology is migrating to a UNIX workstation platform with the HP object-oriented database system OpenODB and X-Windows user interface. Future releases of IESEM will support cooperative work.

Acknowledgments

This work was partially supported by Progetto Finalizzato Sistemi Informatici and Calcolo Parallelo of the CNR (Italian National Research Council) under grants 89-00 052.69, 90.00 705.69, 91.00 930.69.

Our thanks to Victor Basili (University of Maryland) who provided many useful ideas during discussions about IESEM. We also thank Aurora Lonigro, Antonio Ciullo and many others from the University of Bari who helped to develop and test IESEM.

References

[1] F. Abbattista, F. Lanubile, G. Mastelloni, G. Visaggio, "An experiment on the effect of design recording on impact analysis", *Proc. of the International Conference on Software Maintenance*, September 1994.

[2] G. Arango, L. Bruneau, J. Cloarec and A. Feroldi, "A tool shell for tracking design decisions", *IEEE Software* (March 1991) 75-83.

[3] V. R. Basili, "Viewing maintenance as reuse-oriented software development", *IEEE Software* (January 1990) 19-25.

[4] I. D. Baxter "Design maintenance systems", *Communications of the ACM*, **35**, 4 (April 1992) 73-89.

[5] G. Canfora, A. Cimitile, and U. de Carlini, "A logic-based approach to reverse engineering tools production", *IEEE Transactions on Software Engineering* **18**, 12 (December 1992) 1053-1064.

[6] Y. Chen, M. Nishimoto and C. V. Ramamoorthy, "The C Information Abstraction System", *IEEE Transactions on Software Engineering* **16**, 3 (March 1990) 325-334.

[7] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software* (January 1990) 13-17.

[8] A. Cimitile, F. Lanubile and G. Visaggio, "Traceability based on design decisions", *Proc. of the Conference on Software Maintenance*, November 1992, pp.309-317.

[9] S. D. Conte, H. E. Dunsmore and V. Y. Shen, *Software Engineering Metrics and Models*, Menlo Park, CA: Benjamin/Cummings, 1986.

[10] C. Desclaux, "Capturing design and maintenance decisions with MACS", *Software Maintenance: Research and Practice* **4**, 4 (December 1992) 215-231.

[11] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Menlo Park, CA: Benjamin/Cummings, 1989.

[12] J. C. Esteva and R. G. Reynolds, "Learning to recognize reusable software by induction", *International Journal of Software Engineering and Knowledge Engineering* **1**,3 (September 1991) 271-292.

[13] N. Fenton, *Software Metrics: A Rigorous Approach*, London: Chapman & Hall, 1991.

[14] N. Fenton, "Software measurement: a necessary scientific basis", *IEEE Transactions on Software Engineering* **20**, 3 (March 1994) 199-206.

[15] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke, "Viewpoints: a framework for integrating multiple perspectives in system development", *International Journal of Software Engineering and Knowledge Engineering*, **2**, 1 (1992) 31-57.

[16] P.K. Garg and W. Scacchi, "A hypertext system to manage software life-cycle documents", *IEEE Software* (May 1990) 90-98.

[17] M. Georges, "MACS: maintenance assistance capability for software", *Software Maintenance: Research and Practice* **4**, 4 (December 1992) 199-213.

[18] M. H. Halstead, *Elements of Software Science*, New York: Elsevier North-Holland, 1977.

[19] S. M. Henry and D. Kafura, "Software structure metrics based on information flow", *IEEE Transactions on Software Engineering* **SE-7**, 5 (September 1981) 510-518.

[20] E. Horowitz and R. C. Williamson, "SODOS: a software documentation support environment - its definition", *IEEE Transactions on Software Engineering*, **SE-12**, 8 (August 1986) 849-859.

[21] D. Kafura and G. R. Reddy, "The use of software complexity metrics in software maintenance", *IEEE Transactions on Software Engineering*, **SE-13**, 3 (March 1987] 335-343.

[22] M. Klein, "Capturing design rationale in concurrent engineering teams", *Computer* (January 1993) 39-47.

[23] P. J. Layzell and L. A. Macaulay, "An investigation into software maintenance - perception and practices", *The Journal of Software Maintenance: Research and Practice* **6**, 3 (May-June 1994) 105-120.

[24] F. Lanubile and G. Visaggio, "Maintainability via structure models and software metrics", *Proc. of the 4th International Conference on Software Engineering and Knowledge Engineering*, June 1992, pp.590-599.

[25] J. Lee and K. Y. Lai, "What's in design rationale?", *Human-Computer Interaction* **6**, 3-4 (1991) 251-280.

[26] M. M. Lehman, "Programs, life cycles, and laws of software evolution", *Proc. of the IEEE* **68**, 9 (September 1980) 1060-1076.

[27] J. A. Lewis and S. M. Henry, "On the benefits and difficulties of a maintainability via metrics methodology", *Software Maintenance: Research and Practice*, **2** (1990) 113-131.

[28] B. P. Lientz and E. B. Swanson, "Problems in application software maintenance", *Communications of the ACM* **24**, 11 (November 1981) 763-769.

[29] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune and P. Tulula, "CARE: an environment for understanding and re-engineering C programs", *Proc. of Conference on Software Maintenance*, September 1993, pp.130-139.

[30] T. Mason and D. Brown, *lex & yacc*, Sebastopol, CA: O'Reilly & Associates, 1990.

[31] T. J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering* **SE-2**, 4 (December 1976) 308-320.

[32] J. A. McCall, P. K. Richards and G. F. Walters, *Factors in Software Quality, Vols I, II, III*, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977.

[33] D. L. Parnas, "On the design and development of program families", *IEEE Transactions on Software Engineering*, **SE-2**, 1 (March 1976) 1-9.

[34] S. L. Pfleeger and S. A. Bohner, "A framework for software maintenance metrics", *Proc. of the Conference on Software Maintenance*, November 1990, pp.320-327.

[35] C. V. Ramamoorthy, Y. Usuda, A. Prakash, W. T. Tsai, "The Evolution Support Environment system", *IEEE Transactions on Software Engineering* **16**, 11 (November 1990) 1225-1234.

[36] S. Rugaber, S. Ornburn and R. J. LeBlanc, Jr., "Recognizing design decisions in programs", *IEEE Software* (January 1990) 46-54.

[37] N.F.Schneidewind, "Setting maintenance quality objectives and prioritizing maintenance work by using quality metrics", *Proc. of Conference on Software Maintenance,* October 1991, pp.240-249.

[38] N.F.Schneidewind, "Methodology for validating software metrics", *IEEE Transactions on Software Engineering* **18**, 5 (May 1992) 410-422.

[39] I. Simmonds, "Configuration management in the PACT software engineering environment", *ACM SIGSOFT Software Engineering Notes* **17**, 7, 118-121.

[40] I. Tervonen, "Quality-driven validation: a link between four research traditions", *Proc. of the 4th International Conference on Software Engineering and Knowledge Engineering*, June 1992, pp.370-377.

[41] C. Wild, K. Maly and L. Liu, "Decision-based software development", *Software Maintenance: Research and Practice* **3**, 1 (March 1991) 17-43.

Name	Brief Description
Create	Store a new decision after it has been edited
Delete	Remove a waiting decision from the repository
Update	Modify the content of a waiting decision
Instance	Instantiate a decision from a generic one adding a Gen link
Replace	Substitute an active decision with a new one adding a Succ link
Reuse	Get a decision from another project
Connect	Carry a waiting decision to active status
Disconnect	Revert an active decision to waiting status

Table 1. The manipulation services of the TSS

Name	Brief Description
AddDerLink	Link a higher-level software object to a lower-level software object
AddInpLink	Link a higher-level software object to a decision
AddOutLink	Link a decision to a lower-level software object
AddCauseLink	Link a cause decision to an effect decision
AddJustLink	Link a justifying decision to a justified decision
RemoveLink	Remove a link from the repository

Table 2. The linking services of the TSS

Name	Brief Description
Next	Return the immediate adjacent nodes in the graph
Previous	Return the immediate adjacent nodes in the inverse graph
TraceForward	Starting from a higher-level software object, return the lower-level software objects and the decisions involved
TraceBackward	Starting from a lower-level software object, return the higher-level software objects and the decisions involved
PropagateForward	Return the scope of effect of a decision in the graph
PropagateBackward	Return the scope of effect of a decision in the inverse graph
History	Returns all the historical predecessors of a decision
WhichLink	Return the link type between two nodes in the graph

Table 3. The navigation services of the TSS

Phase	Tasks	IESEM tools
Analysis	1. Write requirements specification	Analysis CASE tools
	2. Import requirements specification	Translator
Design	3. Write design decisions and link them to requirements specification	TSS
	4. Write essential design	Design CASE tools
	5. Import essential design and compute design metrics	Translator
	6. Link design decisions to essential design	TSS
	7. Look for outliers	SQM
	8. Classify and predict	External data analysis tools
Implementation	9. Write implementation decisions and link them to essential design	TSS
	10. Write, compile, and debug code	
	11. Import the language-oriented design and compute code metrics	Extractor
	12. Link implementation decisions to language-oriented design	TSS
	13. Look for outliers	SQM
	14. Classify and predict	External data analysis tools

Table 4. Task outline during development

Tasks	IESEM tools
1. Assess a request for change	SQM
2. Make a hypothesis of change	TSS
3. Analyze the impact of change	TSS
4. Implement the change	CASE tools, Translator, Extractor, TSS
5. Revalidate the program	SQM

Table 5. Task outline during maintenance

Metric	Module before change	Module after change	Subsystem before change	Subsystem after change
IF	100	81	6639856	6639818
v(G)	4	14	641	661
η_2	22	35	773	785
N_2	104	290	12810	13208
V	1621	4992	257353	265352
N	297	829	34725	35819
TLOC	37	108	5350	5516
LOC	29	77	4671	4787
CD	0.027	0.055	0.020	0.021

Table 6. Maintainability measures before and after change

List of figures

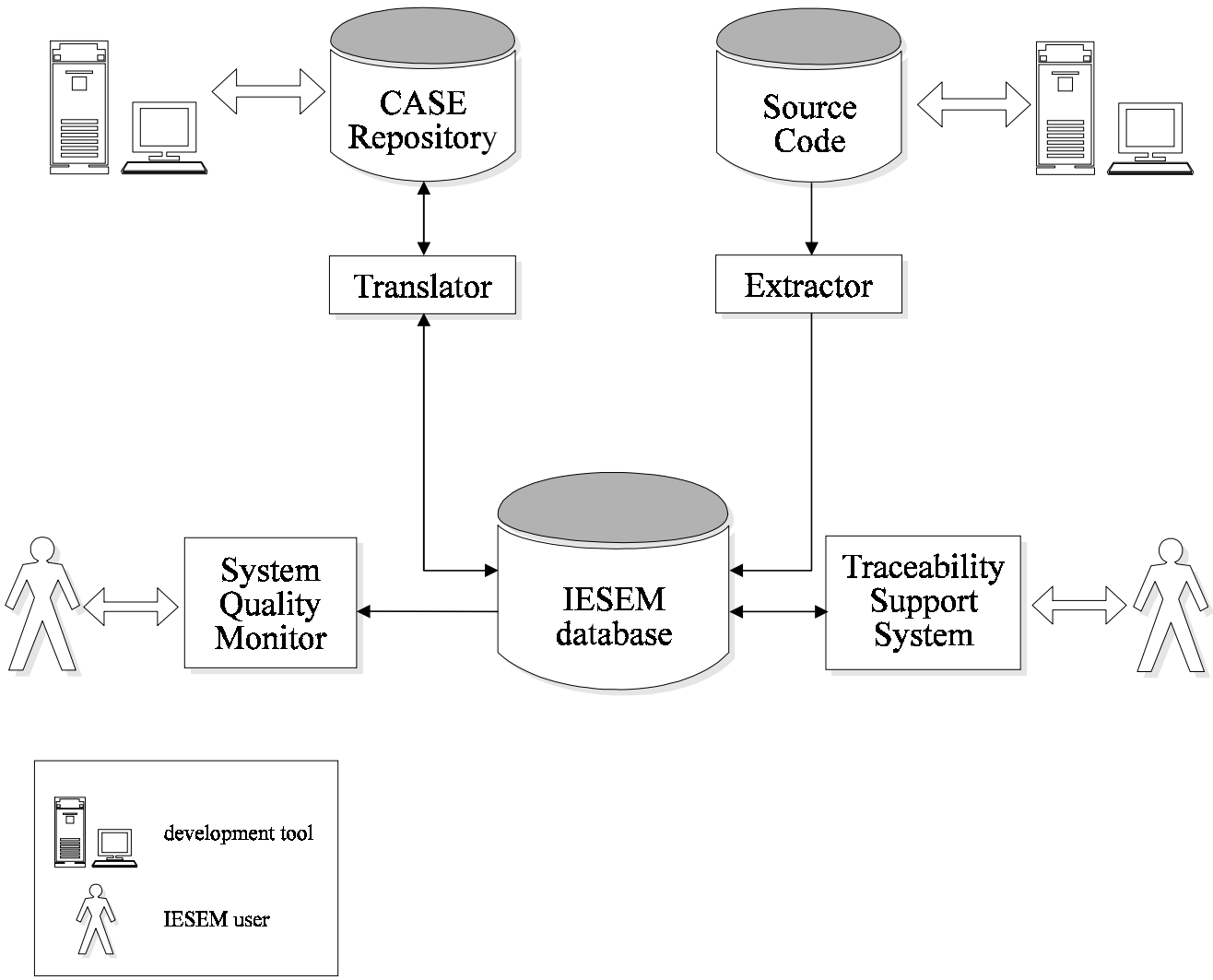

Figure A. IESEM architecture

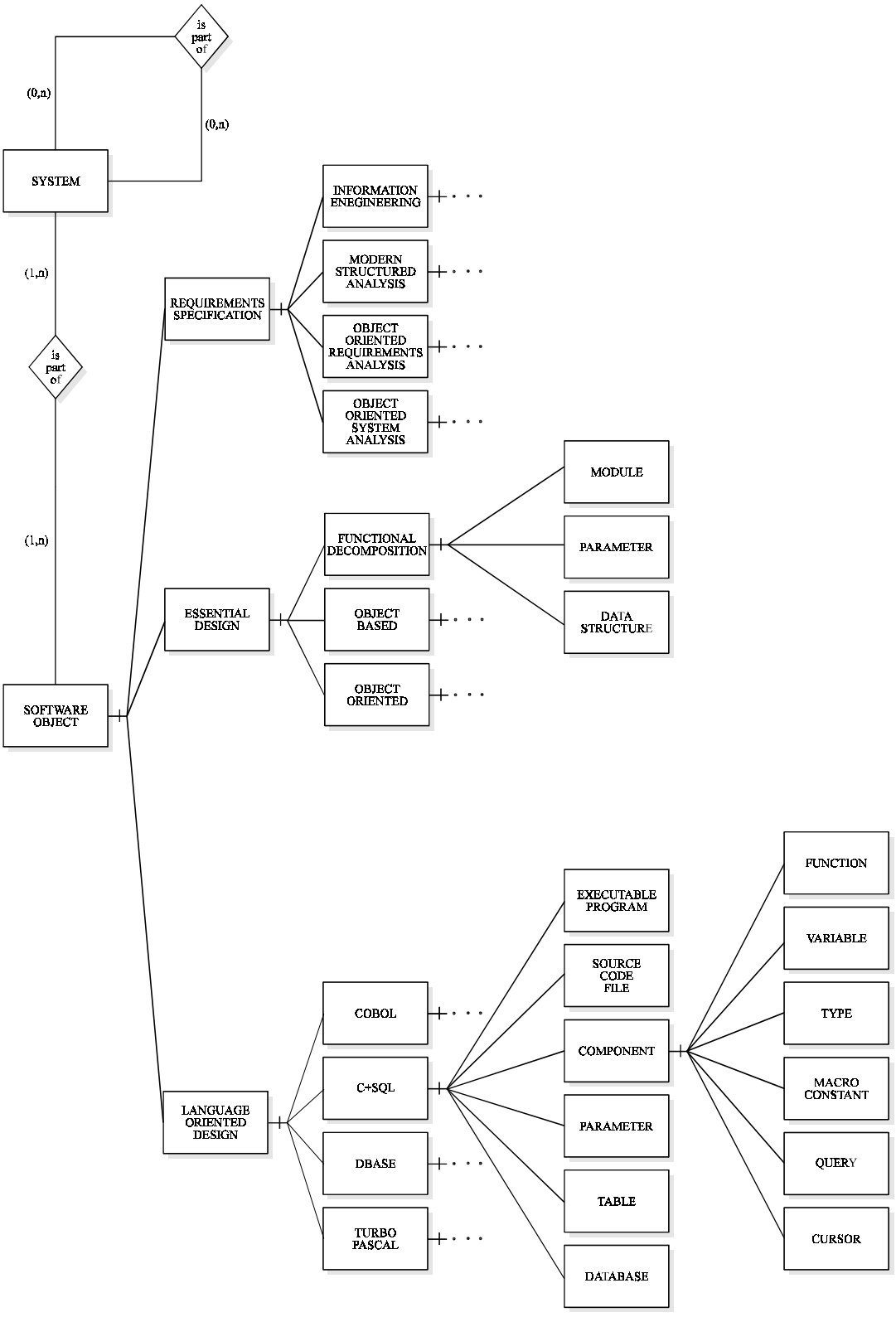

Figure B. Main EER diagram

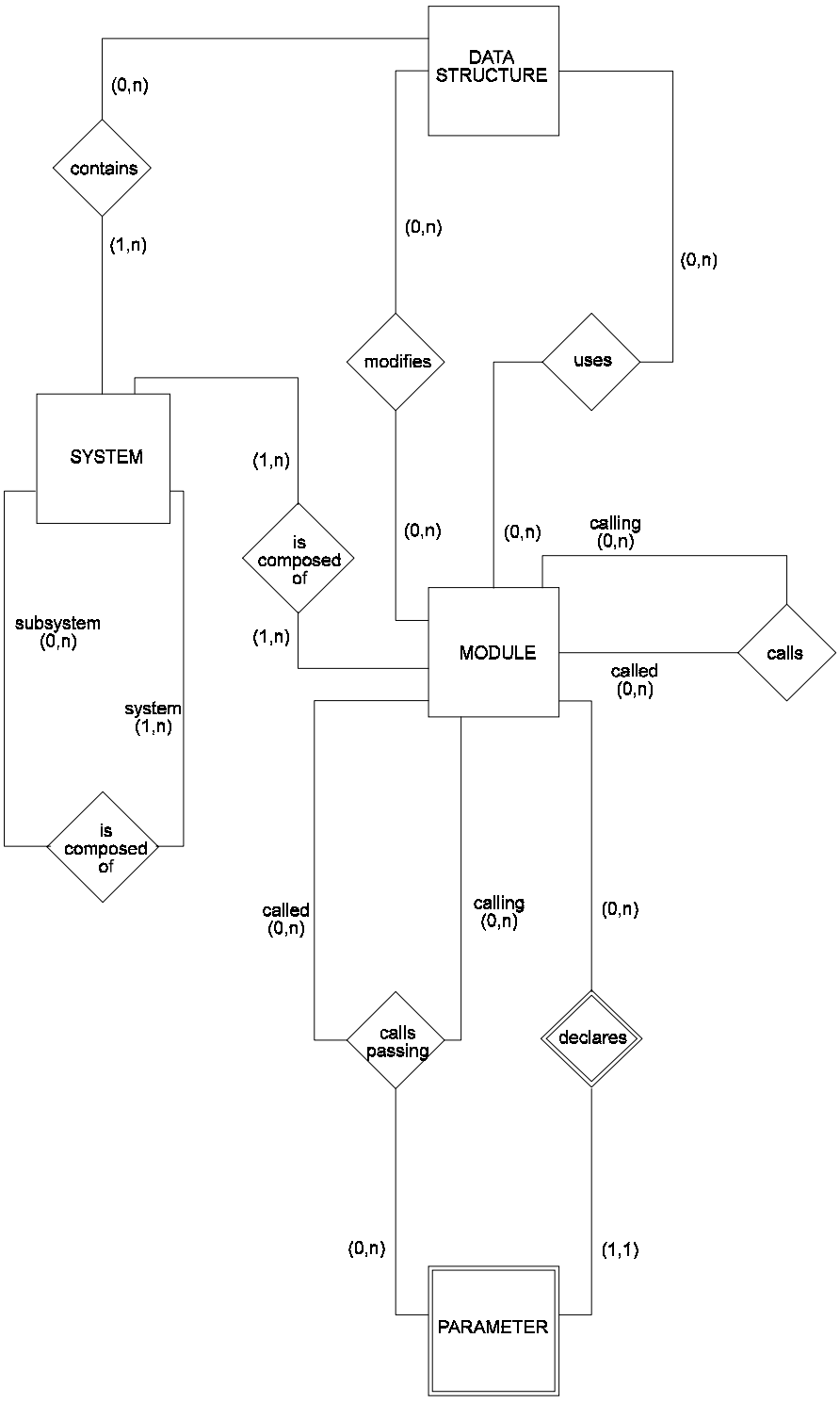

Figure C. EER schema of essential design via functional decomposition

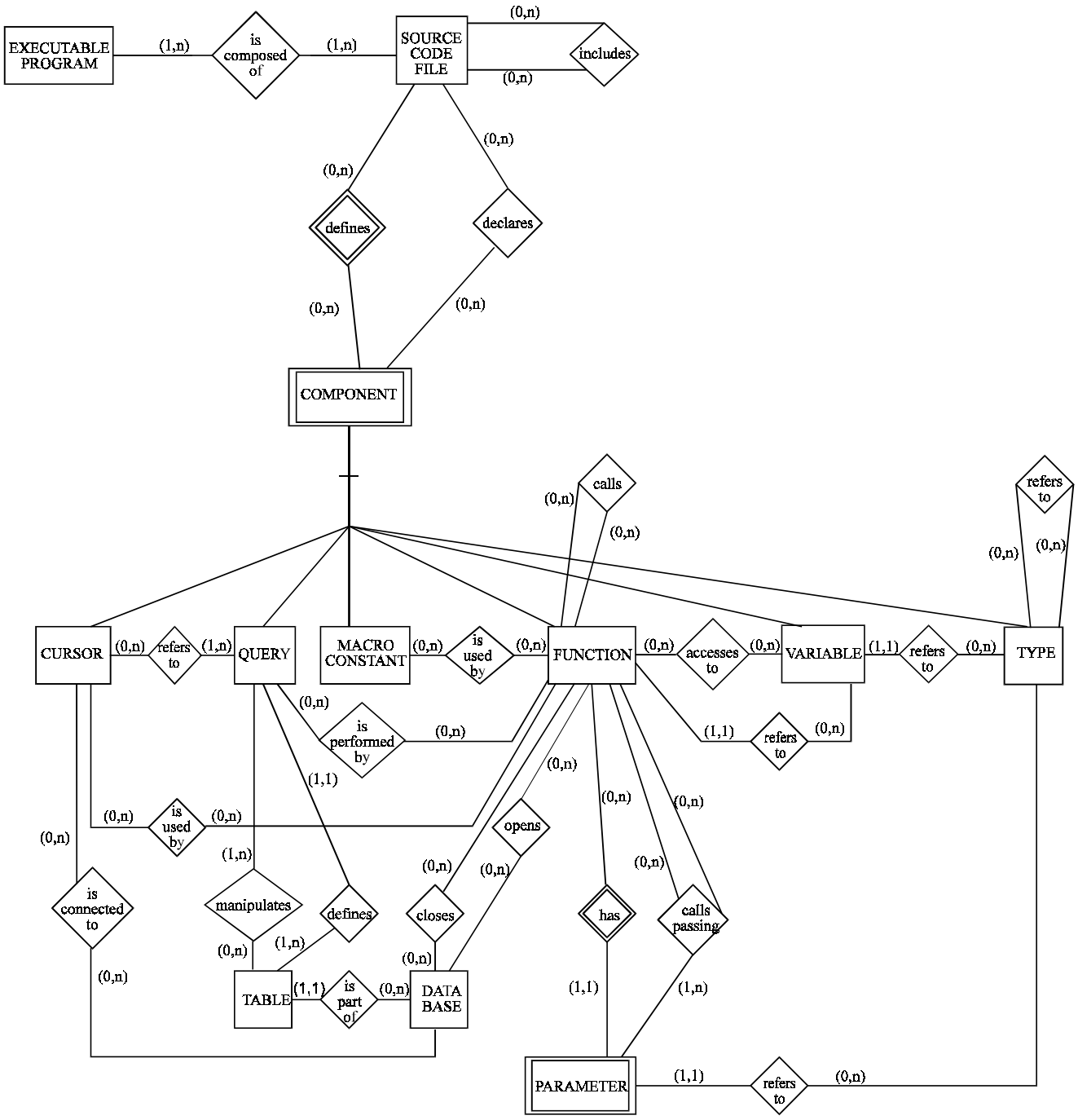

Figure D. EER schema of C+SQL language-oriented design

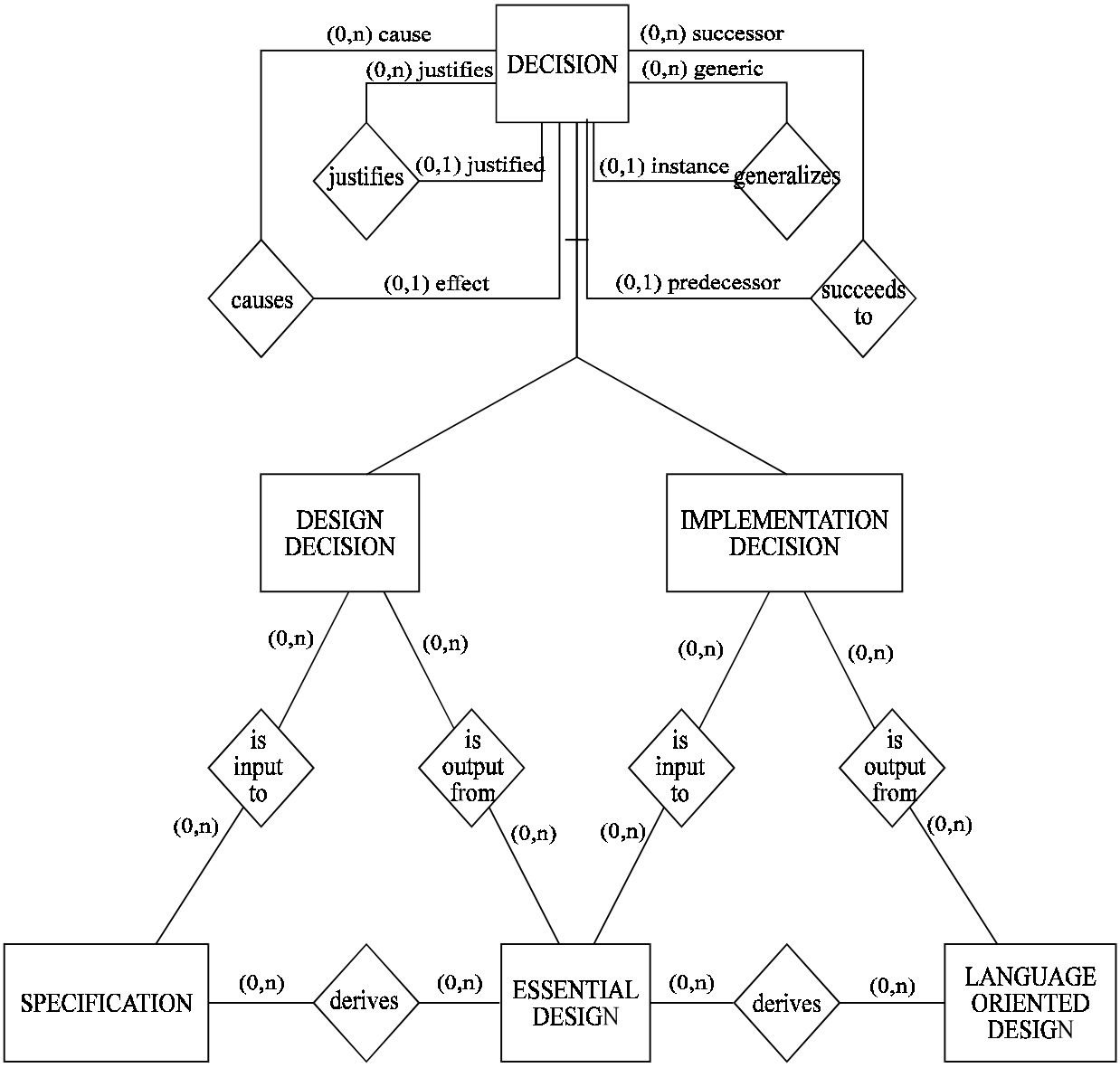

Figure E. EER schema of external traceability

project TSS
author Filippo
creation date 12/1/94 10:05:07
last update 12/1/94 10:05:07
decision id: d34
status active
prob
desc: "How to represent TSS services in the graphical interface"
type: Representation
Alt
id: a1
desc: "Present each service as a button on the screen to be invoked by use of the mouse"
id: a2
desc: "Hide each service in menu which must be pulled down before a service can be chosen"
sol
id: a1
desc: "Present each service as a button on the screen to be invoked by use of the mouse"
just
desc: “Buttons improve the visibility of operations although they take up more screen space than menu pull-down. A prototype of interfaces has shown that the current configuration of operations fits in the screens”
type: Usability

Figure F. A design decision

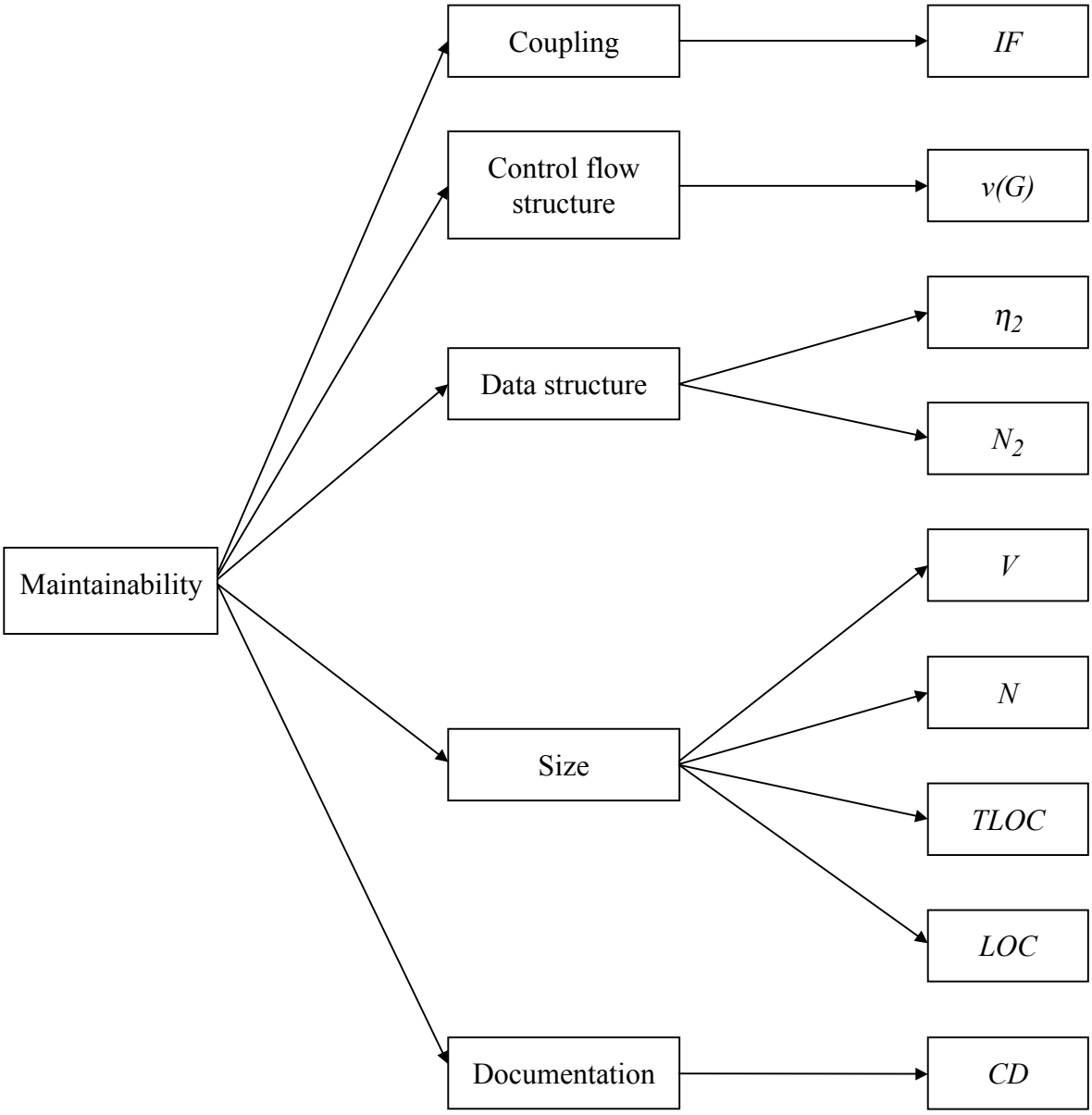

Figure G. The decomposition of the maintainability attribute

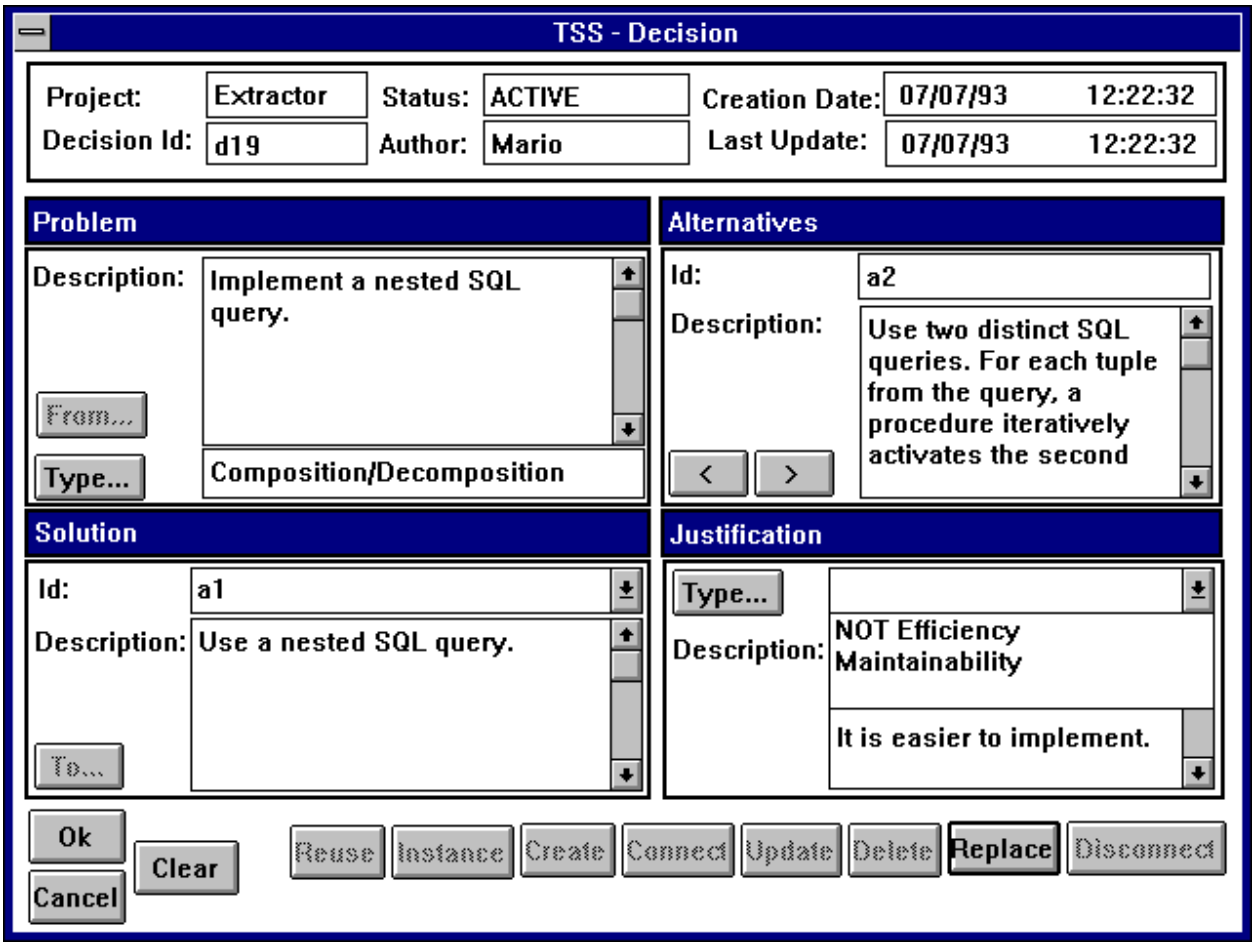

Figure H. The TSS screen with the decision candidate for change

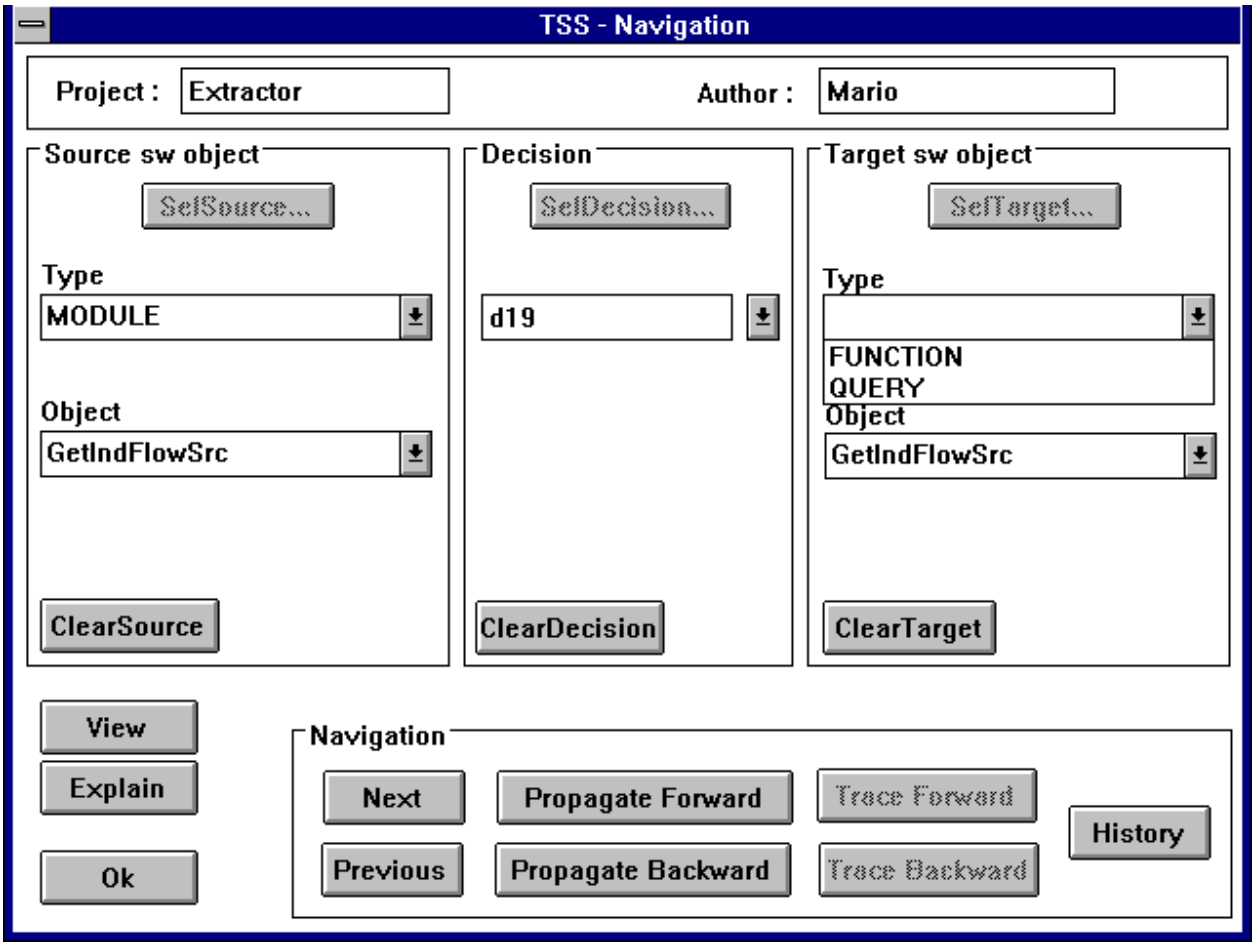

Figure I. The TSS screen with the links of the decision candidate for change