Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

WHAT-IF: a function to estimate the impacts of potential changes in a software

S. Ajila,[a] H. Basson[a,b] & J.-C. Derniame[a]

[a] *CRIN-CNRS, Bâtiment LORIA, Campus scientifique, BP 239, 54506 Vandoevre-lès-nancy, France*

[b] *Université du Littoral, IUT de Calais, 62100 Calais, France*

Abstract

Software maintenance is generally known to be time consuming and expensive. One of the main reasons for this is that it is difficult for software engineers and analysts to understand fully a software system that they did not design or implement. Even when they develop their own system, it is easy to forget the details of the design or code developed at earlier stages.

Some of the questions a software engineer may want to ask when making a change are : *what other parts of the software must I change if I carry out this modification? who in the developing team must be informed of this modification? what other versions or configurations of the software system must I change?* The effect of changing a part of a software system depends on the item of change, type of change and environment of change. Therefore, when making a change it is necessary to analyze its impacts and keep track of the details of implementing the change. In this paper we present a model that will allow *a priori*, the impact analysis and propagation of objects change in a software system. Our work is primarily aimed at large software systems.

WHAT-IF is a knowledge-based system. It uses an attributed multigraph to represent the structure of a software system from different points of view. The inference engine for impact analysis and change propagation is written in Prolog. Thus, the inference engine can be fired each time there is a need to evaluate the impact of a particular change.

Key words : software maintenance, software components, impact analysis, change propagation, relationships, knowledge base.

1 Introduction

During software development, objects are produced which describe the system from different points of view. For instance, the requirement definition specifies the problems which have to be solved, the design describes the system's structure in terms of subsystems and their interconnections, the specification gives a formal description of the software and the code gives an "operational" dependencies between the various modules.

As needs evolve, software must be amended to adapt to the new environment. Often such changes introduce inconsistency in the system because of the ripple effect on the rest of the software. Our capacity to make changes to software is limited if we must rely on manual methods such as reading the design document, looking at the code, etc.

In general, software evolution may be triggered by changes to the requirements upon which the software system was specified; to the functional specification; to the design; to the system application environment (communication with other software, i/o devices, etc.); to environment technology of the system (programming languages, etc.); and changes to the source code itself [Davies 88].

Whatever the type of change that triggers the evolution of a software, we believe that, modifying a software system involve the following :

1. Understanding the *meaning* and the *relationship* between the item of change and the structure of the software system,
2. Analyzing the *impact* of a change on the system,
3. *Specifying* the change and *validating* the changed system,
4. *Evaluating* the *cost* and *deciding* whether to carry out the change or not, and
5. *Recording* the history of "change related information" and *evaluating* the quality of change (i.e. *examining* the feedback information from users of the changed product).

In this paper we will limit ourselves to the aspect of impact of a change.

The effect of changing an object depends largely on the item of change, type of change and the environment of change. For example, changing a software requirement can affect different project teams working on different aspects of the software under development, whereas changing a function call parameter may not have the same impact on these project teams.

Therefore, when a developer decides on a modification, he needs to analyze its impact and keep track of the details of implementing the change. Even a simple change, such as adding an argument to a function's formal parameters, has many facets. The programmer must not only change the function definition, documentation, and tests, but must change all the calling functions as well. This may require figuring out who the callers

are, finding the code, and even modifying export lists. It may also involve changing an interface in the architecture of the system and changing a formal parameter definition.

Depending on each of the software *views*, different impact analysis and change propagation issues may arise. These may also lead to different kinds of *domain specific knowledge* [Kaiser 88, Harandi 90] and heuristics.

Our work is primarily aimed at software systems of tens or more thousands of lines of code. Dealing with such system is known as *programming-in-the-large*[Ramamoorth 86]. Generally, these systems involve large, distinct groups of people in specification, in design, in implementation and, in testing and maintenance.

The rest of this paper is organized as follows : in section 2 we give a detail description of WHAT-IF model, while we present in section 3 a prototype of the model, and a conclusion is given in section 4.

2 What-If model

2.1 Definitions

Let S be a software system represented by a set of objects $O_S = \{o_1, o_2, ..., o_n\}$. Let $T_s = \{t_1, t_2, ..., t_n\}$ be the set of change types that can be carried out on S such that for a given change $\{t_i, o_j\}$ we can define

$$f_{impact}\{t_i, o_j\} \longrightarrow \{o_1, ..., o_i, o_k, ...\}$$

in such a way that some system properties in S remain invariant if the change is finally carried out and the affected objects $\{o_1, ..., o_i, o_k, ...\}$ are also modified accordingly, and where :

- the objects in O_S are linked by explicit dependencies,
- T_s depend on the software system *view*,
- f_{impact} is the impact analysis function, and
- $\{o_1, ..., o_i, o_k, ...\}$ are the "direct victims" of the change i.e. those objects that have direct relationships with o_j.

The ability to estimate the impact of a change (in terms of objects that may be affected by a change) and perform change propagation depends on the following factors:

- software system views,
- change types,
- set of objects to be manipulated,
- dependency relationships between objects of S, and
- f_{impact} and its properties.

Therefore, there is the need for us to define explicitly these factors. The definition of which is presented in the subsections that follow.

2.2 Software system views

A software system view S_{view} is a set of abstractions that gives possible characteristics of the software system. When making a change, it is necessary to define a sort of "objects change life cycle" and this definition must be based on the characteristics of the software system.

Software system views have been categorized [Avellis 91] as : programming language and structural view, architectural view, software life cycle view (in this case one can have water-fall life cycle view, spiral life cycle view, etc.), and domain view.

Each of these views has advantages and disadvantages. For example, the programming view has the advantage that it is easy to implement and in most cases, it is based on the structure of the programming language. Its disadvantage is that it focuses only on the programming language dependency relationships and requires that the maintainer creates a sort of "mental knowledge" of the relationships between the design or the specification and the programming level.

In this work, we are interested in the impact of change across software life cycle phases. Therefore, our system of views is based on what we choose to call "mixed views approach" : *Water-fall and domain views.*

In this approach, we have four life cycle phases. Each phase represents a view. Therefore, we have "requirement view", "specification view", "design view", and "programming view" [Davis 88]. For each view, there is a "domain view" [Harandi 90] which makes explicit the fine-grained dependencies between system parts. This is illustrated in figure 1. The advantage of this approach is that we can explain the impact of a change in terms of domain specific view and life cycle view. It also allows us to specify impact of change in terms of the following combination of views : requirement-specification view (r1), specification-design view (r2), design-programming view (r3), requirement-programming view (r4), requirement-design view (r5), and specification-programming view (r6).

2.3 Change types and set of objects in What-If

For each view, we need to define change types that will operate on the objects of the view. In some cases, such changes may be trivial such as type change (variable, function, etc.) in programming view, merging and deletion in design-programming view, or such as modification and elimination in requirement-specification view. It may be complex in some cases and this

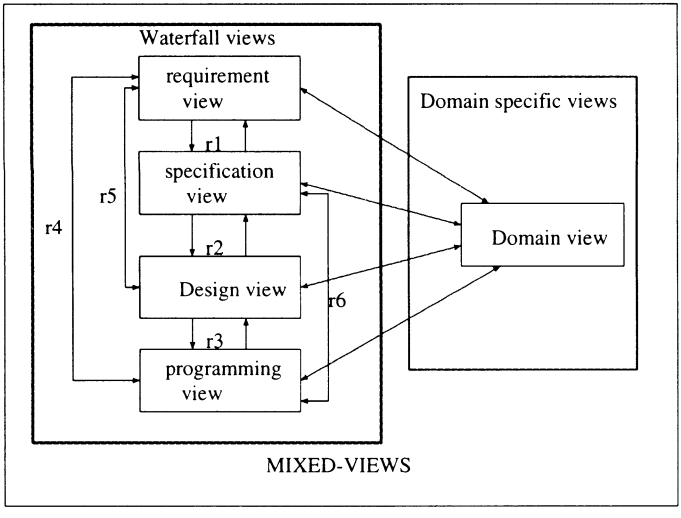


Figure 1: WHAT-IF model system of views

may need the intervention of specialists and users of the system before the change type can be specified.

The objects manipulated by WHAT-IF are described by the following hierarchy:

- requirements objects, e.g chapters, sections, subsections, etc. in the requirement document;
- specification objects, e.g. specification functions and operations;
- design objects, e.g. objects that encode design decisions such as active and passive objects and operations in HOOD (Hierarchical Object Oriented Design);
- implementation objects, e.g modules, procedures, functions, blocks, packages, tasks, and variables.

2.4 Dependency relationships between objects

Software objects are related to each other by complex dependencies and constraints [Wilde 89] [Harandi 90][Ajila 92]. Therefore, an understanding of *system dependencies* is fundamental for efficient software change.

In this work, we identify the following kinds of dependencies:

- **inter-phases dependency relationships :** these are relationships between objects of different views, and
- **intra-phase dependency relationships :** these dependency rela-

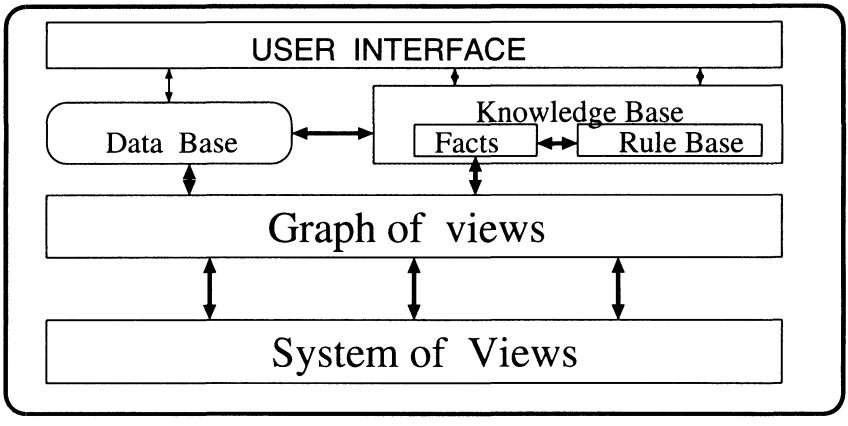


Figure 2: WHAT-IF global architecture

tionships depend on a particular system of view. Example : In the programming view we may have the following :
S_i -**calls**- S_j, if both S_i and S_j are subprograms;
S_i -**creates**- S_j, e.g. tasks in Ada;
S_i -**is-declared-in**- S_j, if S_i is declared inside S_j;
S_i -**is-a-parameter-of**- S_j, if S_i is a parameter and S_j is a function;
etc.

2.5 f_{impact} and its properties

Defining $f_{impact}\{t_i, o_j\} \longrightarrow \{o_1, ..., o_i, o_k, ...\}$ is not enough for impact analysis and change propagation. What we actually needed is to define:

$$f_{impact}\{t_i, o_j\} \longrightarrow \{o_1(o_{11}, o_{12}, ..., o_{1n}), ..., o_i(o_{i1}, o_{i2}, ..., o_{in}), o_k(o_{k1}, o_{k2}, ..., o_{kn}), ...\}$$

where

$\{o_1, ..., o_i, o_k, ...\}$ are the direct victims of change and
$\{(o_{11}, o_{12}, ..., o_{1n}), ..., (o_{k1}, o_{k2}, ..., o_{kn}), ...\}$ are the objects that are indirectly concerned by the change.

In this case we can compute both the *direct* and the *indirect* "victims" of change. In order to do this we have a choice between two methods of representation [Rich 92] : "shallow method" and "deep method". Given the disadvantages of the shallow method as enumerated by Rich et al. [Rich 92], we adopted the second method for our model.

Therefore to represent f_{impact}, do impact analysis and change propagation, we adopt the following :

- A graph is used to represent the structure of each view. The nodes of the graph correspond to the objects of the view while the arcs correspond to the dependencies between the objects. Therefore, for the entire system, we have a "multi-graph" [Yau 81][Basson 92] consisting of *graph of requirements view, graph of specifications view, graph of design view,* and *graph of programming view.* Graph representation is chosen here because it can be used as a powerful software engineering technique [Westfechtel 92].

- The multi-graph is then represented by the set of its simple graph components and by their dependencies,

- Composite dependency relations, constraints, and all other knowledge such as pre and post conditions for the application of a particular type of change are organized as rules into the knowledge base of the model (figure 2).

- The graph components and their dependencies are also organized as a set of facts (figure 2).

- Information specific to each software component is stored in a data base (figure 2).

In what follows, we present a prototype of this model. The prototype is based on the design-programming view (r3 in figure 1). We use the HOOD design method to capture the design view and Ada programming language for the programming view. Our choice is based on the fact that HOOD and Ada allow us to see a sort of object oriented concepts in Software development [Lai 91][Heitz 92]; and that some software engineering environments provide an "automatic" translation from HOOD object description skeleton(ODS) to Ada component specifications. This is the case of CONCERTO (a software development environment).

3 Overview of the WHAT-IF model prototype

The overall architecture of the prototype is shown in figure 3. The prototype illustrates four major sub-systems : the CONCERTO environment, the extraction sub-system, the knowledge base and the abstraction sub-system.

- We use the HOOD tool in the CONCERTO environment to generate ODS and the specification part of Ada. The Ada specification is then completed by using an editor.

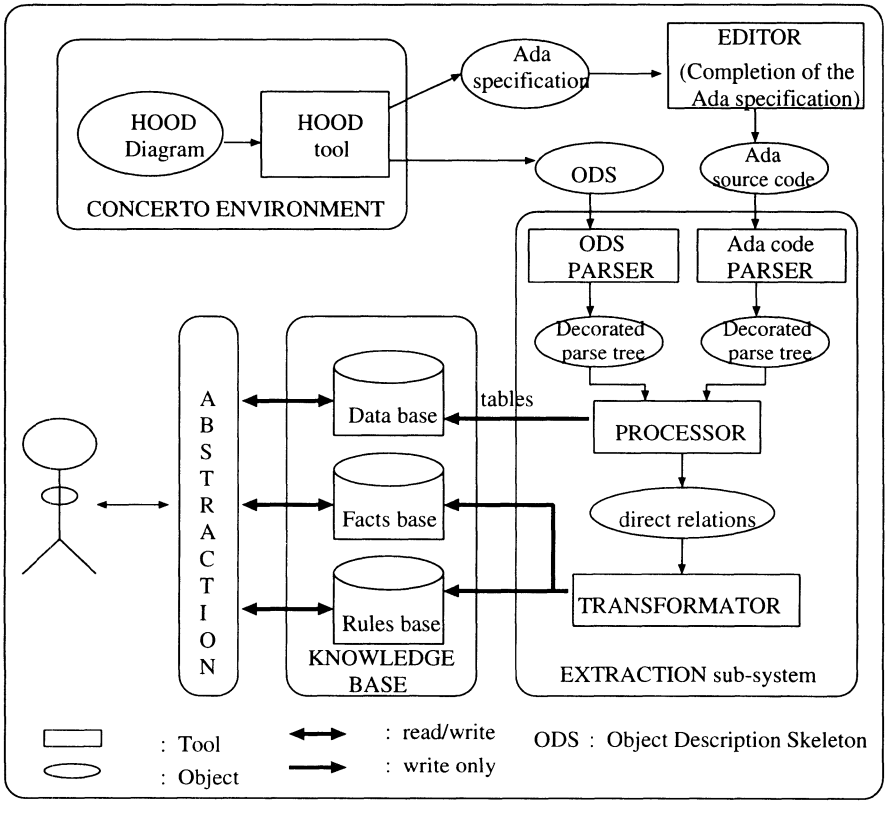


Figure 3: WHAT-IF model prototype

- The EXTRACTION sub-system produces the necessary data such as tables, direct relations which are later translated into Prolog-like language.
- The knowledge base consists of a data base for storing tables, a fact base, and a rule base.
- The ABSTRACTION sub-system allows the user to interact with the knowledge base and to creates (or builds) a set of personal relations by adding new rules and facts to the system and by relating the existing ones in a different manner.

In the following, we will analyze the extraction and the abstraction sub-systems.

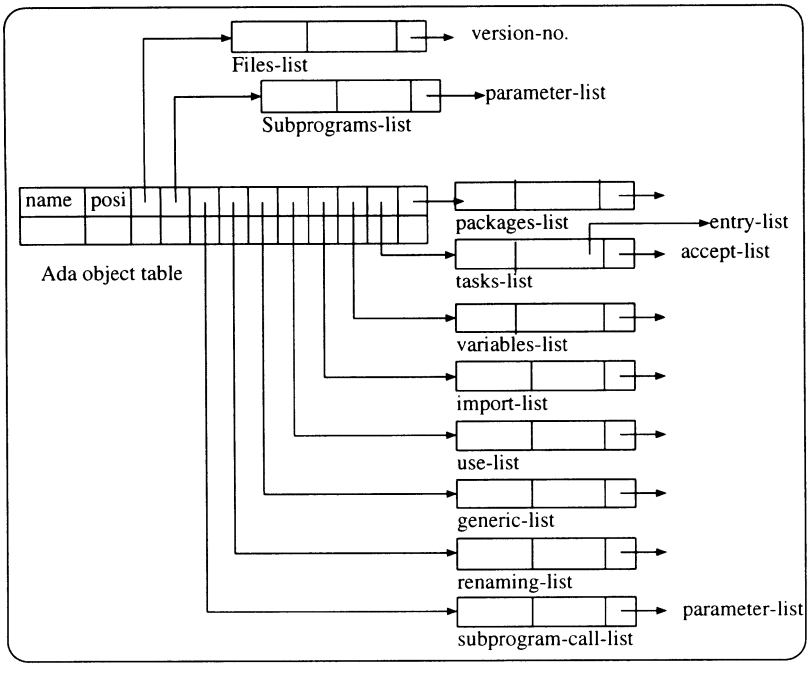


Figure 4: Ada object table

3.1 Extraction sub-system

The EXTRACTION sub-system consists of four main parts : the ODS parser, the Ada parser, a processor, and a transformator. The ODS parser accepts ODS file and produces a decorated parse tree; likewise the Ada parser accepts as input Ada source code and produces a decorated parse tree. The processor processes the parse trees and produces data structures representing the direct relations on which the transformation is based. Finally, the transformator extrapolates the direct relations into a set of Prolog rules. Figures 4 and 5 show two simplified versions of data structures produced by the processor.

The Ada object table (figure 4) contains the names of objects in Ada program and for each object, the following lists : files-list, subprograms-list, subprogram-call-list, renaming-list, generic-list, used-list, import-list, variables-list, tasks-list, and packages-list.

This table (Ada object table) gives also a summary of several direct relations. Let OB-Ada denotes an object of type package, task, function, or procedure in Ada program, and Var-Ada denotes a set of variables, and let N stand for a set of integers, then, the Ada object table contains the following relations :

(R1) *is-parameter-of* : {OB-Ada, Var-Ada, N}, defined as (obj, var, i) ∈ *is-parameter-of* iff object obj declares the variable var as a parameter in the i^{th} position.

(R2) *calls* : {OB-Ada, OB-Ada}, defined as (obj_1, obj_2) ∈ *calls* iff obj_1 directly calls (or active) obj_2.

(R3) *type-return-value* : {OB-Ada, Var-Ada}, defined as (obj, var) ∈ *type-return-value* iff object obj is a function and the returned value after invocation of obj is of type var.

(R4) *is-declared-in* : {OB-Ada, Var-Ada}, defined as (obj, var) ∈ *is-declared-in* iff object obj declares variable var.

(R5) *var-contains-var* : {Var-Ada,Var-Ada}, defined as (var_1, var_2) ∈ *var-contains-var* iff var_2 is contained within var_1 and var_1 is either a record or a type definition.

(R6) *obj-contains-obj* : {OB-Ada, OB-Ada}, defined as (obj_1, obj_2) ∈ *obj-contains-obj* iff obj_2 is contained within obj_1.

(R7) *obj-rename* : {OB-Ada, OB-Ada}, defined as (obj_1, obj_2) ∈ *obj-rename* iff obj_2 is the new name of obj_1 in the Ada renaming concept.

(R8) *var-rename* : {Var-Ada, Var-Ada}, defined as (var_1, var_2) ∈ *var-rename* iff var_2 is the new name of var_1 in the Ada renaming concept.

(R9) *activates* : {OB-Ada, OB-Ada}, defined as (obj_1, obj_2) ∈ *activates* iff obj_1 directly activates obj_2 and obj_2 is a task body.

(R10) *is-the-body-of* : {OB-Ada, OB-Ada}, defined as (obj_1, obj_2) ∈ *is-the-body-of* iff obj_2 is the specification part of obj_1.

(R11) *instantiates* : {OB-Ada, OB-Ada}, defined as (obj_1, obj_2) ∈ *instantiates* iff obj_2 is generic Ada object and obj_1 instantiates obj_2 by using the Ada keyword **new**.

(R12) *matches* : {OB-Ada, OB-Ada, Var-Ada, N}, defined as (obj_1,obj_2, var, i) ∈ iff (obj_1,obj_2) ∈ *calls* and obj_1 defines var as parameter at position i when calling obj_2.

The HOOD object table (figure 5) contains the names of each object defined, their mode (active or passive), and for each object the following list : operation-list, used-objects-list, is-implemented-by-list, and internal-objects-list.

Let OB-hood denotes HOOD object and OP-hood denotes HOOD operation, then the HOOD object table summarizes the following relations :

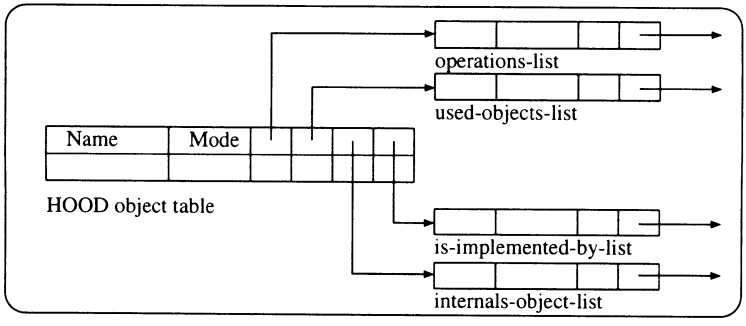


Figure 5: HOOD object table

HOOD objects	Corresponding Ada objects
objects (active or passive)	packages
operations	procedure or function
constraint operations	tasks
class objects	generic packages
object control structure (OBCS)	tasks in the body of packages
operation control structure (OPCS)	separate units in the body of procedures

Figure 6: Mapping of HOOD objects to Ada objects

(R13) *defines* : {OB-hood, OP-hood}, defined as (obj, op) $\in$ *defines* iff obj is an HOOD object (active or passive) and op is an operation defined by obj.

(R14) *includes* : {OB-hood, OB-hood}, defined as $(obj_1, obj_2) \in$ *includes* iff obj_1 is the parent object of obj_2.

(R15) *implemented-by* : {OP-hood, OP-hood}, defined as $(op_1, op_2) \in$ *implemented-by* iff op_1 is a parent object operation, op_2 is a child object operation, and $(obj_1, op_1) \in$ *defines* and $(obj_2, op_2) \in$ *defines* such that $(obj_1, obj_2) \in$ *includes*.

(R16) *use* : {OB-hood, OB-hood, OP-hood, OP-hood}, defined as $(obj_1, obj_2, op_1, op_2) \in$ *use* iff $(obj_1, obj_2) \in$ *includes*, $(obj_1, op_1) \in$ *defines* and $(obj_2, op_2) \in$ *defines* such that $(op_1, op_2) \in$ *implemented-by*.

From the table of mapping between HOOD and Ada objects (figure 6), we define the following relation :

(R17) *mapped-to* : {OB-Ada, OB-hood}, defined as (obj_1, obj_2) $\in$ *mapped-to* iff obj_1 is defined as HOOD object (object, operation, opcs, obcs) and obj_2 is defined as Ada object (package, task, procedure, etc.), and obj_1 is mapped to obj_2 according to the mapping rule in figure 7.

The transformator extrapolates these relations from the tables (Ada object table, HOOD object table, and the mapping table) and represents each of them with Hilog rule.

Hilog [Chen 89] is a logic programming language (developed at the Computer Science Department of SUNY Stony Brook). It is an extension of Prolog and possesses a higher-order syntax, so that arbitrary terms may occupy the function position of terms. Thus, p(a), q(b)(c), X(d), and Y(e)(f) are all valid terms in Hilog. The semantics are first order like Prolog.

Thus, for objects in Ada program and in HOOD, the transformator produces Hilog rules (figure 7) such as :

- is-parameter-of(obj, var, i). (R1)
- calls(obj1, obj2). (R2)
- type-return-value(obj, var). (R3)
- is-declared-in(obj, var). (R4)
- var-contains-var(var1, var2). (R5)
- obj-rename(obj1, obj2). (R7)
- defines(obj1, obj2). (R13)
- includes(obj1, obj2). (R14)
- etc.

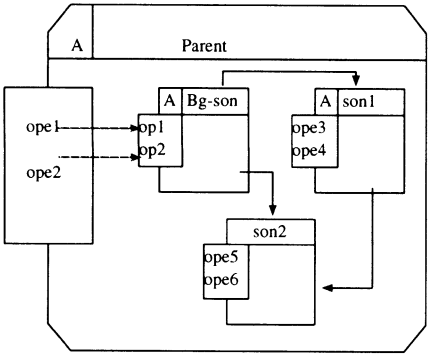
We refer to these as "facts", and they are stored in the facts base.

3.2 Abstraction sub-system

The ABSTRACTION sub-system is not a finished product. It is designed in such a way as to allow the users to formulate relations by adding rules to the system and by relating the existing ones in such a way as to prove some assertions or to creates personal queries. At present, it is implemented to support real time answers for the following queries :

1. WHAT-IS(object-name, level) : Every object in the system must be identified. This query allows such identification and gives insight into the visibility of the object. "object-name" refers to the object and "level" indicates the object view (HOOD or Ada). This query interacts directly with the data base.

   example : WHAT-IS(Parent, hood)? (figure 7)
   Then, the reply is as follows :



HOOD diagram of object Parent

```
...

package Bg-son-ada is
  procedure op1-ada(opr1);
  procedure op2-ada(opr2);
end Bg-son-ada;
package son1-ada is
  procedure ope3-ada(opr1 : in INTEGER);
  procedure ope4-ada(opr2 : in BOOLEAN);
end son1-ada;
package son2-ada is
  procedure ope5-ada;
  procedure ope6-ada;
end son2-ada;

with Bg-son-ada, son1-ada, son2-ada;
package body Parent-ada is

...

begin
 if False then
   son1-ada.ope3-ada(opr3);
   son1-ada.ope4-ada(opr4);
 else
   son2-ada.ope5-ada;
 end if;
end Parent-ada;
```

An example of Ada program of Parent

```
object-hood(Parent).
object-hood(Bg-son).
object-hood(son1).

...

defines(Parent,ope1).
defines(Parent,ope2).
includes(Parent,Bg-son).
includes(Parent,son1).
includes(Parent,son2).

pack-spec(Bg-son-ada).

...

is-declared-in(ope3-ada,opr1).
is-declared-in(ope4-ada,opr2).

obj-contains-obj(Bg-son,op1).
obj-contains-obj(Bg-son,op2).

...

calls(Parent-ada, ope3-ada).
calls(Parent-ada,ope4-ada).
calls(Parent-ada,ope5-ada).

...

imports(Parent-ada,Bg-son-ada).
imports(Parent-ada,son1-ada).
imports(Parent-ada,son2-ada).
```

Example of Hilog rules generated for Parent

Figure 7: HOOD, Ada, and Hilog rules example

```
Object           : Parent
Mode             : active
Internal objects : Bg-son, son1, son2
Operations       : ope1, ope2
```

2. WHAT-LIST(level) : This query gives the list of all the objects at a given level. It interacts directly with the data base.

3. WHAT-IF(object-name, level, change-type) : This is the impact analysis query and the most important of all the queries. The parameter "level" can be empty. In such a case, it is a global impact analysis. If "level" is hood, then the impact analysis is limited to the HOOD view and, if ada, the analysis is limited to Ada view.

The "change-type" is defined as follows :
- when "level" is hood, change-type can be
i) object mode change-type (OMT),
ii) operation constraint change-type (OCT), and
iii) operation/object suppression change-type (OOT).

-when "level" is ada, change-type can be
i) parameter (deletion/addition) change-type (PAT),
ii) type definition change-type (TDT),
iii) assignment statements change-type (AST).

-when "level" is empty, then, the change-type can be any of the above. In such a case the system evaluates the object-name type in order to know where to start the analysis. For example, if object is define at the level of HOOD, the analysis starts from the HOOD view and its propagated to the Ada view, similarly for Ada object. This query invokes internally the WHAT-IS query.

Rules and constraints are defined in such a way as to takes into consideration the different queries and change-types. Therefore, for each query and change-type, rules are activated to satisfy the requirement of the query.

Example : WHAT-IF(ope1, OCT)? (figure 7).
In this case, "level" is empty. Therefore, the analysis will be global. Since "ope1" is an HOOD operation, the analysis will start from HOOD view and propagate to Ada view. The reply to this query is as follows :

```
Visibility :
              Operation      : ope1
              Constraint     : non-constraint
              Implemented by : op1.Bg-son

Impact < 1> :
              if the <OCT> of <ope1> is changed,
                 then the <OCT> of <op1> must also be changed
                 because <ope1> <is-implemented-by> <ope.Bg-son>.

Propagation :
              <ope1>        <is-mapped-to>     <ope1-ada>
              <op1>         <is-mapped-to>     <op1-ada>
              <ope1-ada>    <is-defined-in>    <Parent-ada>
              <op1-ada>     <is-defined-in>    <Bg-son-ada>

Impact < 2>
```

```
if the <OCT> of <ope1> is changed,
   then <Parent-ada> and <Bg-son-ada> must be
   modified in order to take into consideration
   this change.
```

4 Conclusion

Maintenance of a software requires a tool for the impact analysis and propagation of a change. This paper has presented an approach for the design and implementation of such a tool. WHAT-IF model captures the entire life cycle view of a software.

Even though WHAT-IF model prototype is based on HOOD and Ada views, the global architecture itself (figure 2) is general and could be used for other type of views. The prototype also allows the participation of the users by building their own relations, and queries are sufficiently simple enough to use.

This experimentation has shown that a logic based programming language can be a "time saving" prototyping language. Although, we do not recommend Hilog as interface language for practicing maintenance engineers, but a Prolog like language can be a good start for an experiment of this kind. Finally, we are aware of the problem of performance with the rule-based systems. But with the recent scientific development and research for the integration of relational data base and logic base system, we believe that the performance can be improved.

References

[**Ajila 92**] S. Ajila, H. Basson, and N. Boudjlida, *Software Process Assistance: a case study of the impact of object modification during software development,* in 1st African Conference on Research in Computer Science, Yaoundé, Cameroun, pp 73-84, vol. 1,Oct. 14-20, 1992.

[**Avellis 91**] G. Avellis, A. Iacobbe, D. Palmisano, G. Semeraro and C. Tinelli, *An Analysis of Incremental Assistant Capabilities of a Software Evolution Expert System,* in Proc. of the Conference on Software Maintenance '91, CSM91, Sorrento, Italy, Oct. 1991.

[**Basson 92**] Henry Basson, M.C. Haton, and J.C. Derniame, *Characteristics graph of software quality,* In International symposium on Computer and Information Sciences VII, pp.455-461, Nov. 1992.

[**Chen 89**] W. Chen, M. Kifer, and D.S. Warren, em Hilog : A first-Order Semantics of Higher-Order Logic Programming, in Proceeding of North American Conf. on Logic Programming, pp 1090-1114, 1989.

[**Davis 88**] C.G. Davis, P.J. Layzell, *Rules to Govern Change in JSD-Based Systems,* in Proc. of IEEE Conference on Software Maintenance, CSM88, Phoenix AZ, pp 34-40, 1988.

[**Harandi 90**] M. T. Harandi and J. Q. Ning, *Knowledge-Based Program Analysis,* IEEE Software, January 1990.

[**Heitz 92**] Maurice Heitz, *Towards more formal developments through integration of behavior expression notations and methods within HOOD developments,* 5^{th} International Conference on Software Engineering & Applications, Toulouse, Dec. 7-11, 1992.

[**Kaiser 88**] G. E. Kaiser and S. S. Popvitch, *Intelligent Assistance for Software Development and Maintenance,* IEEE Software 5, no 3: pp 40-49. 1988.

[**Lai 91**] M. Lai, *Conception orientée objet: Pratique de la méthode HOOD,* DUNOD Informatique, Paris, 1991.

[**Lieberherr 89**] K.J. Lieberherr and I.M. Holland, *Tools for Preventive Software Maintenance,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.2-13, Miami FL, 1989.

[**Ramamoorthy 86**] C.V. Ramamoorthy, V. Garg and A. Prakash, *Programming in the Large,* IEEE Trans. on Software Engineering, Vol.SE-12, No. 7, pp 769-783, July 1986.

[**Rich 92**] Charles Rich and Richard C. Waters, *Knowledge Intensive Software Engineering Tools,* IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 5, October 1992.

[**Westfechtel 92**] B. Westfechtel, *A Graph-Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents,* 5th Int. Workshop on CASE, Montréal, Québec, Canada, July 6-10 1992.

[**Wilde 89**] N. Wilde, R. Huitt, and S. Huitt, *Dependency Analysis Tools: Reusable Components for Software Maintenance,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.126-131, Miami FL, 1989.

[**Yau 81**] S. S. Yau and P. C. Grabow, *A model for representing programs using hierarchical graphs,* IEEE Trans. Software Eng., vol SE 7, pp. 556 - 574, Nov 1981.