Software as Learning. Validation and Maintenance Issues.

Jose Hernandez-Orallo
Universitat Politècnica de València, Departament de Sistemes Informàtics i Computació
Camí de Vera 14, Aptat. 22.012 E-46071, València, Spain
E-mail: `jorallo@dsic.upv.es`

Abstract— **We reconsider validation and maintenance characteristics of software systems under the analogy between software science and philosophy of science or, more precisely, between software construction and machine learning (ML). From this outset, many classical techniques from ML can be used. In particular, we adapt a constructive extension of reinforcement learning to address the question in a formal way. We define a measure of software 'predictiveness', which is identified with software validation, to represent the stability of a system. An inversely related measure, the probability of modification, is also obtained for each component and for the whole system. The application in practice of these measurements is discussed. From here, we present some models of maintenance cost based on a detailed combination of predictiveness and modifiability. We study different software arrangement topologies theoretically. Hierarchised topologies, especially downward confluent ones such as trees and lattices involve less maintenance costs. Moreover, some intuitive rationales are confirmed, namely that compressed systems and coherent models (without patches or exceptions) are manifestly more maintainable.**

Index Terms— **Software Validation, Software Maintenance Models, Machine Learning, Induction, Software Redundancy, Software Components Topology.**

I. INTRODUCTION

THE analogy between programs and scientific theories outlined in (Fetzer 1991) and characterised in (Hernández-Orallo 2000b), and the modern view of software as an experimental science (Basili et al. 1986) (Basili 1993) has left behind the previous unsuccessful analogue between software entities such as specification, program and verification, and mathematical entities such as problem, theorem and proof (see a thorough discussion in Hernández & Quintana 2000a) .

Philosophy of science provides a much more enlightening paradigm for software construction, by explicitly recognising that software engineering is an experimental science but also that the development of an actual software system requires more inductive techniques (Partridge 1997) than deductive ones.

More concretely, machine learning (ML) is a more precise and practical framework for this new paradigm for software engineering. A software system is regarded as a learning system. Traditional software systems are viewed as eager learning systems, where the system is an intensional and operative expression of the requirements, which behaves correctly in a certain environment. By using the ML terminology, requirements can be identified with the training data, the software system is just a working hypothesis and correction is more properly viewed as predictive accuracy.

The new wave of software paradigms, under very different banners such as intelligent software (Maes 1995), smart software and software agents (Genesereth & Katchpel 1994)(Nwana 1996), interactive software (Wegner 1996) or adaptive software (Lieberherr 1996) rounds off the analogy with ML, because they can be seen as more reactive or interactive learning systems, and many results and techniques, especially from lazy methods (Aha 1997), can be applied to them from the sub-fields of query learning, case-based reasoning, knowledge acquisition and revision, etc.

In our opinion, it is somehow short-sighted to try to develop intelligent, smart, interactive or adaptive software from scratch, without regarding more than thirty years of theoretical and experimental results from ML. Even in the case of 'traditional' software, it is worth adapting some constructions, techniques, methods and theoretical results to better understand the development and nature of software systems.

In this way, this paper 'reuses' for software development a theory initially devised to account for constructive reinforcement learning. The following theory is adapted from (Hernandez-Orallo 2000). The idea is based on the fact that, whatever the approach to knowledge construction, the validation must come from a *reinforcement* with respect to the evidence. Moreover, the revision or modification of knowledge must come from a partial or total weakness of the theory or, in other words, a loss of reinforcement.

Thus, the use of reinforcement as a tool for the study of the *validation* and *revision* of an inductive theory is translated into its use for the *validation* and *maintainability* issues of software systems.

The paper is organised as follows. Section 2 adapts the ML paradigm for software, highlighting some assumptions and technical preliminaries to work on with. Section 3 applies a theory of reinforcement to software components and Section 4 defines some measures of system predictiveness, stability and probability of modification, by using an upward propagation of reinforcement. Section 5 introduces several models to account for the propagation of modifications, usually top-down. Finally, section 6 merges these two 'propagations' in order to study the maintenance cost of different software topologies. The most technical work is described in the appendix. Section 7 comments on different extensions and refinements of the previous study and Section 8 concludes the paper.

II. ML PARADIGM FOR SOFTWARE

We have just described how the analogy between software and learning can inspire and justify the use of several frameworks and techniques from ML, and it even forces a re-understanding of software quality factors and life-cycles (see Hernández & Quintana 2000a). However, the adaptation for validation and maintenance issues has some technical difficulties.

A. Adapting the ML framework

First, the sample data for constructing a software system is composed of experience from other software systems, software repositories and requirements information. The experience and software repositories can be well formalised under the usual “background knowledge” in ML, which can be expressed in an intensional way and is supposed to be validated. However, the information that is usually gathered up for requirement elicitation is not composed mostly of extensional data such as input-output pairs or positive and negative examples. On the contrary, this information provided for the construction of a software system is composed of base cases, scenarios, interviews and a great amount of intensional knowledge.

Secondly, once the system is in operation or in the implantation stage, the validation cannot come exclusively from its use, it necessarily must be combined with the user’s satisfaction about the product, by extending reinforcement with rewards and penalties.

Thirdly, to study maintainability, we must study two different things: the predictiveness of software, i.e., the expectancy of future modifications, which directly depends on the reinforcement which has been distributed upwards, and the consequences that each change may have in other components, which determines a downward flow. However, the first topology is dynamic while the second one is usually static.

Sample data. Training set

It is essential to discern what will constitute the examples or sample data from which reinforcement originates. In ML, these cases are usually facts, correspondences, pairs of input-outputs, etc. Classically, it is said that the behaviour of any system can be described in terms of input-output pairs, i.e. a function, expressed under a proper codification. However, theoretically, it has been proved that most complex systems cannot be identified by a finite data set of input-outputs (Gold 1967). Intuitively, part of the data must be given in an intensional way. Finally, even if this important fact is ignored, in practice, the effort to convert a software system in terms of binary input-output is not sensible nowadays.

Contrarily, it is more practical to extend the notion of example. Apart from input-output pairs, we can identify many sorts of examples in the training phase (or requirements elicitation). They may be extensional, such as a use case, a scenario, a row in a correspondence table, a query and answer, etc., or an intensional concept. Even each sentence from the specification in natural language can be used as an example.

As we will see, any of these sorts of examples can be used for reckoning reinforcement. The only requirement is the definition of a proper notion of 'accordance', i.e., that a system is in accordance with some example. For instance, in the case of input-output pairs, the idea of accordance is extremely simple; if the system receives the input and returns the output as a result, the system is in accordance with the example. However, it may be more difficult to define 'accordance' for other kind of examples. In any case, it is important not to measure the *different* examples with the same value, because some of them are incomparable. Hence, the idea is to study reinforcement in a separate way for each sort and then try to put all that information together.

Granularity of propagation

One of the objectives of our study is the detection of which parts are being more reinforced than others, in order to know which are more predictive to future situations, or in other words, are less expectable to be modified in the future.

The first thing to do is to recognise the entities or components where we are going to centre the measurement. Although in the following we will focus on software components, our idea of component will always be broader than "component software" (Szyperski 1998) and easily extensible to any other system component, either physical (hardware) or logical (software).

The most minute choices, such as an instruction, show that reinforcement must not be distributed by the execution trace. For instance, some instruction can appear in a loop, being unfairly reinforced. On the other hand, the choice of large components provides wider views of how the software is being used. However, this higher level presents some other problems. For example, a module *A* can make use of another module *B* for just one functionality whereas it can use a module *C* for many functionalities. In some way, *C* should be more reinforced than *B*, but this granularity does not allow this appreciation. Although some of these problems are solved in section 3, once again, the idea is to measure reinforcement for the greater number of granularities as possible and then try to understand all the information jointly.

Validation data. User's accordance

The idea of accordance for the training set is clear. All the examples are usually labelled with positive and negative tags such as "the system should behave as the following scenario describes" or "the following situation should not happen".

However, when the system is in use, most of the situations are not like the training set, so they must be accompanied by the tag "this behaviour has been correct" or "this has been a system error". This feedback can be given by the environment, an external system or, more consciously, the user. In the case the behaviour is 'correct', the system is reinforced, as when a new example is predicted by a theory. On the other hand, when the behaviour is detected as 'incorrect', we have a *prediction error* of the system. A simpler assumption could be that things are going well until some feedback states the contrary. In this way, software is being reinforced as time passes by and no modification has been necessary. However, it has been shown in most ML paradigms that learning from positive data only is much harder.

For semantic-based representational languages, there is usually a notion of proof or positive covering, i.e., a theory covers an example iff the example can be derived from the theory. This results in a Boolean notion of accordance. Examples of these languages are propositional languages, Horn theories, full logical theories, functional languages, some kind of grammars, and even higher-order languages. However, with our generalisation of example and with general software systems, one cannot assign a true or false label to the behaviour of a system wrt. some case. It is more accurate to talk about a degree of correctness, from absolute correctness to full malfunctioning.

Definition 2.1. We denote the accordance of a given example e wrt. a system S by $S \supset^{\alpha} e$.

For convenience, $-1 \leq \alpha \leq 1$. In the following, we will refer to *e* as a positive example of *S* when α is l and a negative example if α is –l.

In the simplest case, when a system can be specified by a function $F \subset I \times O$, a positive example is just any element of $e \in F$. If we define $F^{neg} = \{ (i,o) \in I \times O$ such that $\exists e' = (i,o') \in F$ and $e' \neq e \}$ then we have that a negative example is just any element from F^{neg}. If *F* is complete, we have only three situations: positive hits, not covered cases and errors. Hence, $\alpha \in \{1, 0, -1\}$. Positive and negative samples are just subsets of *F* and F^{neg}. respectively. In concept learning or ILP (see e.g. Muggleton & De Raedt 1994), we have that $O = \{ false, true \}$ and it is said that the theory or system *S* covers the examples iff $F \subset S$. In more complex cases, the user or other client systems are responsible for providing the value of α for each example.

III. SOFTWARE AND REINFORCEMENT

In some areas, like artificial neural networks or decision trees, there has been extensive work on how to distribute (or propagate) the apportionment of credit (or reinforcement), initially inspired in intuitive or neurobiological considerations and more recently based on Bayesian frameworks.

The motivation of (Hernandez-Orallo 2000) is to extend the idea of reinforcement to constructive languages, or languages with the ability of re-description, where the dependencies (connections) among the parts are completely flexible in kind and with a dynamical variable topology.

This flexibility allows the study of reinforcement at almost any granularity. For instance, a *component* can be a rule, a procedure or a function, a class, a method, a variable, a module or any other higher division. A system will be just a set of components.

For a correct apportionment of credit, we need to discern which parts are justifiably responsible for the system to behave correctly for a particular example. In other words, if a component can be removed without affecting the functionality of the system wrt. some example, it is clear that this example should not be reinforced. The following definitions try to formalise and extend this idea:

Definition 3.1. A component r_i is said to be β-necessary wrt. S for an example e iff

$$S \supset^{\alpha} e \quad \text{and} \quad S - \{r_i\} \supset^{\alpha'} e \quad \text{and} \quad \beta = \alpha - \alpha'$$

In general, if $\beta = 0$ we say that r_i is not necessary. On the contrary, if $\beta = 1$ we simply say that r_i is necessary. For instance, if we consider a system S composed of modules, we can have that the system covers an example e and without a module m_i, the system does not cover the example, so $S \supset^1 e$, $S - \{m_i\} \supset^0 e$ and $\beta=1$.

Definition 3.2. A system S is reduced for an example e iff

$$S \supset^{\alpha} e \quad \text{and} \quad \neg\exists\ r_i \in S \ \text{ such that } r_i \text{ is not necessary for } e$$

Definition 3.3. Reduced Set: $RS(e, S) = \{ S_i \subset S, S_i \text{ is reduced for } e \}$,

This excludes subsystems with components which are not useful for increasing the accordance of the system wrt. the example.

However, it is not clear how to assign a credit to each rule, as the following example shows:

Example 3.1

Suppose a system S with four components $\{ r_1, r_2, r_3, r_4 \}$ with $S = \{ r_1, r_2, r_3, r_4 \} \supset^1 e$, $S_1 = \{ r_1, r_2, r_3 \} \supset^1 e$, $S_2 = \{ r_1, r_2 \} \supset^{0.5} e$, $S_3 = \{ r_2, r_3 \} \supset^{0.9} e$ and for any other subset of S we have $\alpha = 0$.

We have that $RS(e, S) = \{ S_1, S_2, S_3 \}$.

We can particularise a different set for each rule.

Definition 3.4. $RS_r(e,S) = \{ S_i : S_i \subset RS(e,S) \text{ and } r \in S_i \}$

And from here we compute the credit of each rule in the following way:

Definition 3.5. $credit\,(r, e) = \{ \Sigma_{S' \in RS_r(e, S)}\, \alpha \mid S' \sqsupset^{\alpha} e \} / card(RS(e, S))$

For the previous example, definition 3.5 gives these reasonable values: $credit(r_1, e) = 0.5$, $credit(r_2, e) = 0.8$, $credit(r_3, e) = 0.63$, and $credit(r_4, e) = 0$.

However, in the case of software, the influence of the different components is not additive. Very important modules, classes or functions do not perform anything valuable on their own, whereas interface components are much more visible to the user. Hence, we will only consider the 'saturated' subsystems:

Definition 3.6. A subsystem S' of S is saturated for an example e iff $\neg\exists\ r_i \in S$ such that

$$S' \sqsupset^{\alpha} e \quad \text{and} \quad S' \cup \{r_i\} \sqsupset^{\alpha'} e \quad \text{and} \quad \alpha' - \alpha > 0$$

Theorem 3.7. A subsystem S' of S is saturated for an example e iff $\neg\exists\ r_i \in S$ such that $S'' = S' \cup \{r_i\}$ and r_i is β-necessary wrt. S'' for an example e with $\beta > 0$.

Proof. If r_i is β-necessary wrt. S'' for an example e, we have by definition 3.1 that $S'' \sqsupset^{\alpha''} e$ and $S'' - \{r_i\} \sqsupset^{\alpha'''} e$ and $\beta = \alpha'' - \alpha'''$. Since $S'' = S' \cup \{r_i\}$ then $\alpha'' = \alpha'$ and $\alpha = \alpha'''$. Since $\beta > 0$, we have that $\alpha'' - \alpha''' > 0$ and $\alpha' - \alpha > 0$. □

Definition 3.8. $SS(e,S) = \{ S_i \subset S,\ S_i$ is both reduced and saturated wrt S for $e \}$,

We will refer to the elements of SS as *alternative* subsystems. For the previous example we have that $SS(e, S) = \{ S_1 \}$. Finally, we can define the set of alternative subsystems which contain r as,

Definition 3.9. $SS_r(e,S) = \{ S_i : S_i \subset SS(e,S) \text{ and } r \in S_i \}$

One of the first results of these definitions is that a subsystem is a set of components. That is to say, it is independent of the trace, of how many times a component is used for a given example. This ultimately allows the following definitions.

Definition 3.10. The pure reinforcement $\rho\rho(r)$ of a component r from a system S wrt. some example e is defined as:

$$\rho\rho(r, e) = \Sigma_{S' \in SS_r(e, S)} \{\alpha : S' \sqsupset^{\alpha} e \}$$

In other words, $\rho\rho(r)$ is computed as the sum of 'accordances' from the alternative subsystems for e where r is used. If there are more than one alternative subsystem for a given *e, all* of them are reckoned, but, as we have said, for the same subsystem, a component is computed only once.

The proportion of examples from a given evidence E where r is used, can be computed as

Definition 3.11. The probability of r being used for a given example from evidence $E= \{e_1, e_2, ..., e_n\}$ can be approximated by:

$$P_{use}(r) = \Sigma_{e \in E} \{\text{if } \rho\rho(r, e) > 0 \text{ then } 1 \text{ else } 0 \} / \text{card}(E)$$

For a set of examples, i.e., an evidence E, we extend definition 3.10 in the obvious way:

Definition 3.12. The pure reinforcement $\rho\rho(r, E)$ of a component r from a system S wrt. some given evidence $E = \{e_1, e_2, ..., e_n\}$ is defined as:

$$\rho\rho(r, E) = \Sigma_{i=1..n}\, \rho\rho(r, e_i)$$

Definition 3.13. The (normalised) reinforcement is defined as:

$$\rho(r, E) = 1 - 2^{-\rho\rho(r, E)}$$

In the following, we will omit E when there is no possible confusion. Def. 3.13 is justified by the convenience of maintaining reinforcement between 0 and 1, while rendering easy the computation of reinforcement because each time a new example is covered by a system, the reinforcement of the components that have been used are incremented by $\rho'(r) = (\rho(r) + 1) /2$.

Definition 3.14. The mean reinforced ratio $m\rho(S)$ of a system S with m components is defined as:

$$m\rho(S) = \Sigma_{r \in S}\, \rho(r)/m$$

Finally, we measure the validation *wrt. the evidence*.

Definition 3.15. The course $\chi_S(\ e\ )$ of a given example e wrt. a system S is defined as:

$$\chi_S(e) = max_{\,S' \subset SS(e,\, S)}\ \{\ \Pi_{r \in S'}\ \rho(r)\ \}$$

More constructively, $\chi_S(e)$ is computed as the product of all the reinforcements $\rho(r)$ of all the components r of S used in an alternative subsystem of e. If a component is used more than once, it is computed once. If f has more than one alternative subsystem, we select the greatest course.

Before using definition 3.15 throughout the paper we must verify that it is robust to the introduction of *fantastic* (unreal) concepts. A fantastic concept can be easily constructed by making it necessary for the rest of components, and the mean reinforcement ratio $m\rho(S)$ is increased unfairly. Fortunately, the following theorem shows that this is not possible with course:

Theorem 3.16. The course of any example cannot be increased with *fantastic* components.

Proof. Since the *fantastic* component r_f is necessary for the rest of components it must now appear in all the alternative components for all the evidence, with $l(E)=n$. Thus, and the reinforcement of r_f is exactly $1 - 2^{-n}$ and the reinforcements of all the r_i remain the same. Hence, the course of all the n examples is modified to $\chi'(e_j) = \chi(e_j) \cdot r_f = \chi(e_j) - \chi(e_j) \cdot 2^{-n}$. Since n is finite, for all $e_j \in E$, $\chi'(e_j)$ can never be greater than $\chi(e_j)$.□

IV. VALIDATION PROPAGATION BY REINFORCEMENT

In the ML and philosophy of Science literature, there is a variety of evaluation criteria to select the most plausible hypothesis, i.e., the one with less prediction errors (Merhav and Feder 1998). From this variety, the MDL (Minimum Description Length) principle (Rissanen 1978, 1996) (Barron et al. 1998) and the MLE (Maximum Likelihood Estimator) method have been thoroughly studied and inter-related (Kearns et al. 1999). Associated with them are some popular validation methods such as cross-validation, which is also connected with different notions of hypothesis stability and reinforcement.

A. Definitions of predictiveness and stability

Intuitively, a theory that has been reinforced by the past evidence is more likely to behave properly for the future evidence. Differently from other evaluation criteria, we have given measures of reinforcement for each component, and not a unique value for the whole system. In order to estimate the predictive accuracy (or predictiveness) of a system, we must give a single value. The most natural idea is the mean of all the courses of all the examples in the evidence:

Definition 4.1. The mean course $m\chi(S, E)$ of a system S wrt. an evidence E, with $n = \text{card}(E)$, is defined as: $m\chi(S, E) = \Sigma_{e \in E}\, \chi_S(e)/n$.

In (Hernandez-Orallo 2000), the maximisation of $m\chi(S, E)$ and the MDL principle have been theoretically related. Logically, the shorter the theory the more probability that reinforcement would be more concentrated. In the same paper there are some examples that show that $m\chi(S, E)$ is a more compensated criterion than the MDL principle. Finally, it is possible to formalise the concept of intensionality by using reinforcement. A system is intensional if there are not examples covered by some component with low reinforcement value. Intensionality is shown to be closely related to cross-validation. In other words, systems with components added to the system to cover some exceptional examples, i.e. patches, have less stability.

These extensional parts are not validated and it is highly unlikely that new examples would be covered by these parts, so the system will probably be revised.

In the same way, we can translate these rationales to software systems. Hypothesis stability in ML is converted into system stability, i.e., the system endurance to requirement changes.

Predictiveness is thus distinguished as an actual software quality factor, inversely related to the number of modifications for evolving requirements in the same context (Hernández & Ramírez 2000b). Reinforcement can be used as a very appropriate measure to estimate the probability of modification. More concretely, the probability of modification of a component can be directly specified from the reinforcement value.

Definition 4.2. The isolated probability of modification is:

$$P_{\text{mod}}(r) = 1 - \rho(r) = 2^{-\rho\rho(r)}.$$

It is obvious that this defines a probability, since $0 \leq P_{\text{mod}}(r) \leq 1$. The term 'isolated' is motivated by the aim that definition 4.2 should only measure the probability of modification that originates from each component r, not that other components could occasion the modification of r.

From here, it is straightforward to obtain the probability of modification of the whole system:

Theorem 4.3. If we consider independent the isolated modification of each rule of a system S, the isolated probability of modification of a system S is: $P_{\text{mod}}(S) = 1 - \Pi_{r \in S}\, \rho(r)$

Proof. Since the modification of each component is an independent fact, and S is defined as the set of rules, the probability of modification of one or more element of this set is obtained in the classical way: $P_{\text{mod}}(S) = 1 - \Pi_{r \in S}\, (\overline{P}_{\text{mod}}(r)) = 1 - \Pi_{r \in S}\, (1 - P_{\text{mod}}(r)) = 1 - \Pi_{r \in S}\, \rho(r)$.□

The absolute stability of a system can be defined as $\sigma(S) = 1 - P_{\text{mod}}(S) = \Pi_{r \in S}\, \rho(r)$, i.e., the probability that a system is not modified at all. This stability *of the whole theory* is a very strict requirement, so we will define another notion of stability later.

We have been given probabilities of modification throughout the whole life cycle of the system. However, it would be interesting to measure the probability of modification just for the following k examples. Given the probability of use of one component for one example $P_{\text{use}}(r)$, given by definition 3.11, we can approximate, by a simple combinatorial analysis, that the probability that one or more of the following k examples would use r is $1 - (\overline{P}_{\text{use}}(r))^k$. Then

Definition 4.4. The isolated probability of modification of component r before example k is:

$$P_{\text{mod}}(r, k) = P_{\text{mod}}(r) \cdot \{\, 1 - (\overline{P}_{\text{use}}(r))^k \,\}$$

Theorem 4.5. Given a component r from system S and an evidence $E = \{e_1, e_2, ..., e_n\}$, such that $\exists e \in E\ \rho\rho(r, e) > 0$ (i.e., it is a useful component), then, as k grows, we have that $P_{\text{mod}}(r, k)$ approximates $P_{\text{mod}}(r)$.

Proof. From definition 4.4, we have that $\lim_{k\to\infty} \{\, P_{\text{mod}}(r, k) \,\} = \lim_{k\to\infty} \{\, P_{\text{mod}}(r) \cdot \{\, 1 - (\overline{P}_{\text{use}}(r))^k \,\}\} = P_{\text{mod}}(r) \cdot \{\, 1 - \lim_{k\to\infty} (\overline{P}_{\text{use}}(r))^k \,\}$. Since there exists an example e such that $\rho\rho(r, e) > 0$, then, by definition 3.11, we have that $P_{\text{use}}(r) > 0$, or consequently $\overline{P}_{\text{use}}(r) < 1$. Hence, $\lim_{k\to\infty} (\overline{P}_{\text{use}}(r))^k = 0$, and this yields: $\lim_{k\to\infty} \{\, P_{\text{mod}}(r, k) \,\} = P_{\text{mod}}(r) \cdot \{\, 1 - 0 \,\} = P_{\text{mod}}(r)$. □

In the following, we will suppose there are no useless components, and we will use definition 4.2 and theorem 4.3 to work with long-term life cycles. However, definition 4.4 would be useful to compute short-term modification probabilities and maintenance costs. It even can be modified to consider the last n' examples instead of the whole evidence. For instance, if a module has not been used in the last 4 months, it is not likely that a modification would affect this module.

B. Measurement in practice

Definition 4.2 and theorem 4.3 provide a means to evaluate the predictiveness of a system or, in some way, how much validated it is. However, as we said, there are still some details to resolve in order to make these measurements applicable for software systems: (1) one cannot measure the different examples with the same value, and (2) reinforcement can be measured for different granularities of components.

Weighting the evidence

In ML, examples are usually regularised to the same significance. However, in software, it is difficult to balance some kinds of examples, such as an input-output pair with a scenario. In addition, some examples are used to describe exceptional behaviours, with low frequency of use whereas other examples are introduced to represent the main part of a system or frequent operation. The following extension is useful if one can assign a significance degree d_e to the examples which conform the evidence E.

Definition 4.6. The *'grounded' course* $\chi'(e)$ of a given example e wrt. a system is computed as the normal course $\chi(e)$ multiplied by the significance degree of e. More formally, $\chi'(e) = \chi(e) \cdot d_e$.

Another approach is the repetition of the examples which are more significant. This is exactly equivalent to the use of the previous definition, by repeating each example e in the evidence d_e. times.

Weighting components

The same approach is not valid with components. We cannot compare the reinforcement of a module with the reinforcement of a function. However, if one uses modules as components for obtaining a mean course $m\chi(S,E)$ and an absolute stability $1-P_{mod}(S)$, it is possible to make the same thing for another granularity, e.g. functions, to obtain a different mean course $m\chi'(S, E)$ and absolute stability $1-P'_{mod}(S)$. If one wants to combine both measurements, a major problem arises. In general, the grosser the granularity the greater the mean course and absolute stability. The reason is quite simple. For the same system, the finer the granularity the greater the number of components and reinforcement must be scattered. In the extreme case, if we consider only a component, the system itself, we have the maximum value for $m\chi(S, E) = \Sigma_{e\in E}\, \chi_S(e)/n = \Sigma_{e\in E}\, \rho(S)/n = \Sigma_{e\in E}\, (1 - 2^{-n})\, /n$, which converges quickly to 1 if $n = card(E)$ increases.

To equilibrate the matter there are two options: (1) the introduction of a factor directly related to the number of components, and (2) the introduction of a factor inversely related to the size of each component. The first one may propitiate the pseudo-repetition of components, i.e., components that are always needed in conjunction. Hence, we will choose this second option.

With size(r) we will denote the size of a component r, with the only restriction for size that for all r, size(r) ≥ 1. We extend the definitions in the following way:

Definition 4.7. The extended pure reinforcement is defined as: $\rho\rho^*(r) = \rho\rho(r)$ / size(r).

Likewise we could define the extended normalised reinforcement $\rho^*(r)$ and the extended course $\chi^*(r)$.

Finally, with this modification, reinforcement can be associated with the idea of reusability. Inside a single system, a module or component is reinforced if it is used for many cases or examples. Moreover, the last modification favours granularity which also eases reusability. At the level of different applications, and by considering the evidence as the set of all the examples for these different applications, a highly reinforced module is reused to cover many groups of examples.

V. Modification propagation

We have talked about predictiveness, as the ability of a system to behave correctly for evolving requirements. This gives an isolated probability of modification $P_{mod}(S)$ whatever the part of the system is considered. However, to estimate maintenance costs it is important to know the consequence of each chan[illegible]i.e., how many components are to be modified and how difficult these modifications are.

The following figure shows the two main factors that affect maintenance: the probability of modification which is inversely related to the validation or predictiveness characteristic, and the modifiability of the components which are more likely to be revised.

However, in the literature of software modifiability, there is usually no detailed relationship between the probability of modification of each component and the modifiability of each component. In general, the relation is between the validation of the whole system and the modifiability of the whole system. Figure 5.1. shows the difference of accuracy between the classical approach and ours.

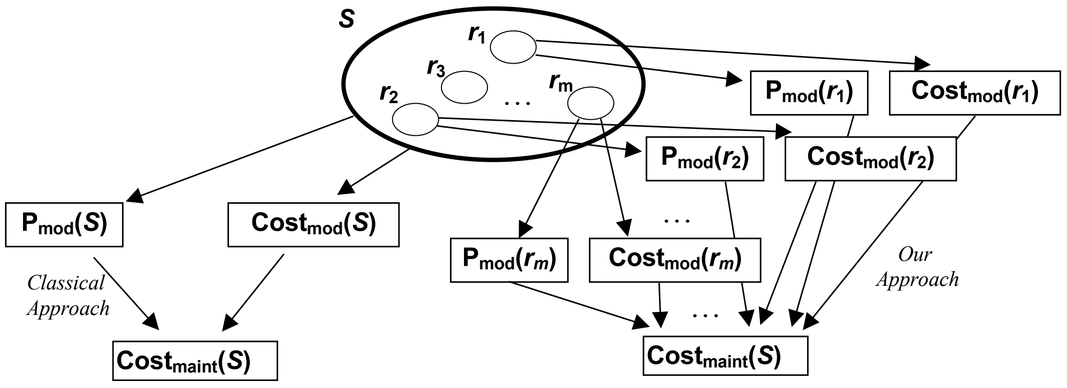

Fig 5.1. Two different ways to estimate the Maintenance Cost

In this section, we present the necessary framework to introduce different particularised models for approximating this maintenance cost, denoted by $\mathrm{Cost}_{\mathrm{maint}}(S)$.

Model 0

The easiest (but less realistic) model for modifiability is the assumption that every component modification is independent to the rest of components. In this case, it is only necessary to know that each component has a modification cost, a real number that we denote by $\mathrm{Cost}_{\mathrm{mod}}(r)$:

Definition 5.1. The *isolated* cost of maintenance is defined as:

$$\mathrm{Cost}^0_{\mathrm{maint}}(S) = \Sigma_{r \in S}\, \mathrm{P}_{\mathrm{mod}}(r) \cdot \mathrm{Cost}_{\mathrm{mod}}(r)$$

Although more detailed than the classical $\mathrm{P}_{\mathrm{mod}}(S) \cdot \mathrm{Cost}_{\mathrm{mod}}(S)$, this last definition has been computed according to the isolated probability of modification.

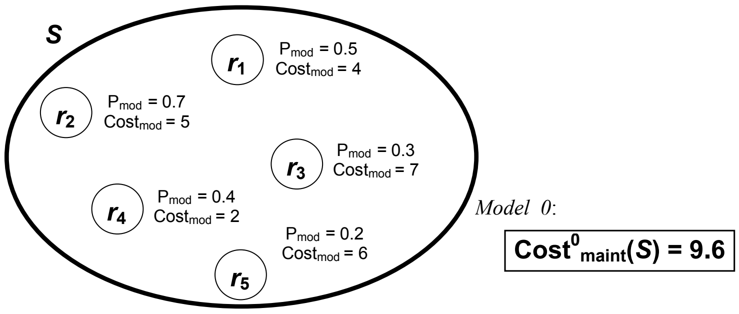

Fig 5.2. Example of estimation using Model 0

For instance, given the system illustrated in figure 5.2, the classical approach takes $P_{mod}(S) = 0.95$ to obtain $Cost_{maint}(S)$. For instance, if it is estimated that each modification would affect 1.5 components on the average, $Cost_{mod}(S)$ can be computed as $Cost_{mod}(S) = 1.5 \cdot \Sigma_{r \in S} Cost_{mod}(r)/card(S) = 7.2$. Finally, $Cost_{maint}(S) = P_{mod}(S) \cdot Cost_{mod}(S) = 6.84$.

In this example, both values, $Cost_{maint}(S)$ and $Cost^0_{maint}(S)$ are not too despair, but, in general, they can differ a great deal. Despite the fact that model 0 is more detailed that the classical one, it is still very simple because it ignores the relationships between components where modification propagation flows.

5.1. Modification dependencies

Given a representation language, there are different notions of dependency. There are functional dependencies, where execution (and semantics) propagates (usually bottom-up) and static dependencies, where modification propagates (usually top-down).

It is difficult to establish exactly which are the modification dependencies of a given system. It depends on the degree of encapsulation of the components, their coupling and other semantic or syntactical considerations. Moreover, these factors are highly reliant on the granularity of components. For instance, if a module or class is modified in its declaration it is easier to detect the modules or classes which are expected to be modified than if a single line of a program is modified.

Once these questions are resolved for a particular system, the modification dependencies can be formalised by the term "r depends on t_i" that we write r ↲ t_i. For all the dependencies of a single component we will also use the following notation r ↲ $t_1, t_2, .. t_n$. This dependency relation does not need to be reflexive or transitive.

Definition 5.2. The direct ascendant set of a component r is defined as: $DAsc(r) = \{ r' / r$ ↲ $r' \}$

Definition 5.3. The direct descendant set of a component r is: $DDes(r) = \{ r' / r'$ ↲ $r \}$

We define the relation ↲* as the transitive and reflexive closure of the dependency relation ↲. Formally,

Definition 5.4. For any two components r_a , r_b, we have that r_a ↲* r_b holds iff $r_a = r_b$ or r_a ↲ r_b or there exists another r_c such that r_a ↲* r_c and r_c ↲* r_b.

Definition 5.5. The ascendant set of a component r is defined as: $Asc(r) = \{ r' / r$ ↲* $r' \}$

Definition 5.6. The descendant set of a component r is: $Des(r) = \{ r' / r'$ ↲* $r \}$

These dependencies are more or less difficult to establish depending on the granularity chosen for the examples. For instance, in a procedural language, suppose that a function f uses functions g and h in its definition, h uses function i in its definition, and function i uses g. The resulting components and dependencies are: f ↲ g, f ↲ h, h ↲ i, and i ↲ g. By the transitivity closure, ↲* extends ↲ to h ↲* g, f ↲* i , f ↲* g and all the reflexive relations.

In the same way, one can extend dependencies to sets of functions, or modules. In this case, the 'uses' or 'includes' directives are a good overestimation to modification dependencies. How much these dependencies overestimate modification dependencies relies on the kind of modification (in behaviour or declaration) and the encapsulation of each module.

Finally, to give a much more present and realistic view, in some modelling stages or languages, dependencies are very heterogeneous, as the following simple object model illustrates:

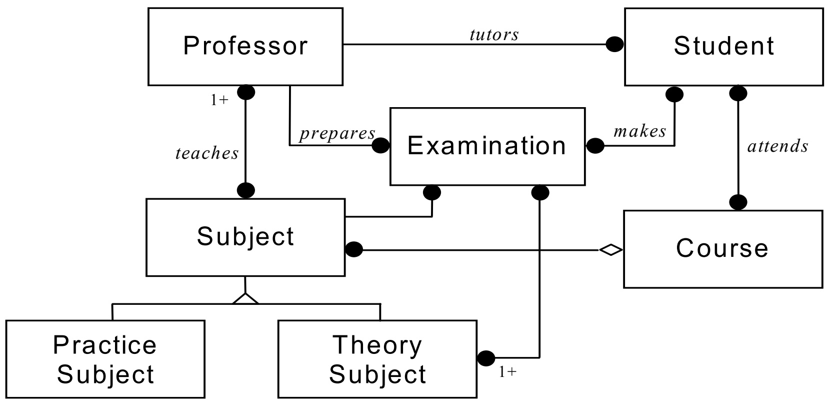

Fig 5.3. Example of heterogeneous dependencies

If we identify classes with components, in many cases we could identify modelling relationships (associations, aggregations and inheritance) in one or both ways, according to external information to the model (or design model information). In any case, the dependencies that can be extracted are barely representative of the modification dependencies between classes. A better approximation can be made by studying the methods and other relationships between classes.

In short, it is possible to define ↲ for any granularity and any representational language, but the accuracy to which ↲ represents modification dependencies is heavily contingent on this granularity and any other experience or information which may be used to refine the estimate.

Besides, not any relation ↵ can be used. There is an important property that this relation must hold, acyclicness, i.e., ↵* must be a partial order relation. This limitation is not very restrictive because, although *static* functional dependencies are frequently cyclic and *static* modelling dependencies are sometimes cyclic (like figure 5.3), the instantiated dependencies of an effective program are acyclic.

This hierarchisation was advocated long ago by (Dijkstra 1968) and (Parnas 1972): "*We have a hierarchical structure if a certain relation may be defined between the modules or programs and that relation is a partial ordering. The relation we are concerned is "uses" or "depends upon*"".

However, if the dependency relation is cyclic, with two components r_1 and r_2 such that r_1↵*r_2 and r_2↵*r_1, then a fictitious component r_f must be inserted to make r_1↵*r_f and r_2↵*r_f and the cycle is removed. Obviously, the costs and probabilities of modification should be readjusted among r_1, r_2, r_f and other components involved.

Model 1

Once relation ↵ is approximated, we can remake the effective probability of modification introduced in the previous section. We can define a new measure which weights the isolated probability of modification and the scope of each modification (its propagation), assuming $\mathrm{P}_{\mathrm{mod}}(r)$ independent.

Definition 5.7. Given the acyclic relation ↵ for modification dependencies, the related probability of modification $\mathrm{P}^*_{\mathrm{mod}}$ of a single component is defined as:

$$\mathrm{P}^*_{\mathrm{mod}}(r) = 1 - \overline{\mathrm{P}}_{\mathrm{mod}}(r) \cdot \Pi_{ai \in Dasc(r)} \left(\overline{\mathrm{P}}^*_{\mathrm{mod}}(a_{\mathrm{i}}) \right)$$

where Π is defined to be 1 if it has no factors.

Finally, model 1 can be introduced accordingly:

Definition 5.8. $\mathrm{Cost}^1_{\mathrm{maint}}(S) = \Sigma_{r \in S}\, \mathrm{P}^*_{\mathrm{mod}}(r) \cdot \mathrm{Cost}_{\mathrm{mod}}(r)$

Let us extend the example of figure 5.2 with some dependencies. The new model applied in fig. 5.4 shows that the cost of maintenance increases, due to the consideration of these modification propagation that were not taken into account in model 0.

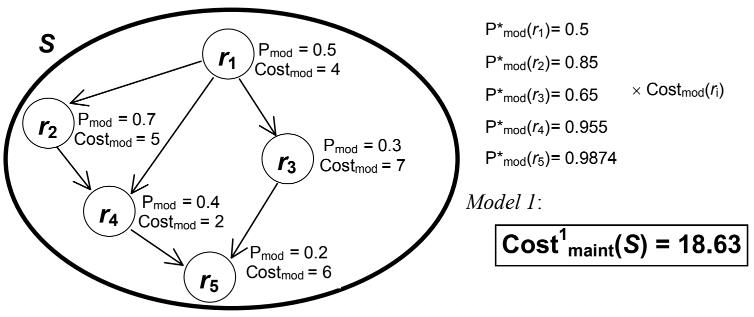

Fig 5.4. Example of Estimation using Model 1

However, this model presents some problems. It over-propagates modification, because it modifies the probabilities through all the paths that dependencies draw. For instance, in Fig. 5.4, we can observe that $\text{P*}_{\text{mod}}(r_4)$ takes into account $\text{P*}_{\text{mod}}(r_1)$ twice: from the path $r_1 \rightarrow r_2 \rightarrow r_4$ and from the path $r_1 \rightarrow r_4$ directly. The same happens with $\text{P*}_{\text{mod}}(r_5)$.

Model 1b

The correction must only consider each dependency once by using the set of ascendants instead of a recursive reckoning of the dependencies.

Definition 5.9. Given the acyclic relation $\lrcorner$ for modification dependencies, the corrected related probability of modification $\text{P*}^{b}_{\text{mod}}$ of a single component is defined:

$$\text{P*}^{b}_{\text{mod}}(r) = 1 - \Pi_{a \in Asc(r)} \left(\overline{\text{P}}_{\text{mod}}(a) \right)$$

and we redefine the cost of maintenance

Definition 5.10. $\text{Cost}^{1b}_{\text{maint}}(S) = \Sigma_{r \in S} \text{P*}^{b}_{\text{mod}}(r) \cdot \text{Cost}_{\text{mod}}(r)$

The previous example is corrected to $P^{*b}(r_1)=0.5$, $P^{*b}(r_2)=0.85$, $P^{*b}(r_3)=0.65$, $P^{*b}(r_4)=0.91$, $P^{*b}(r_5)=0.9496$, which gives $\mathrm{Cost}^{1b}_{\mathrm{maint}}(S) = 18.32$.

Finally, we could use this model to define the detailed stability of a system.

Definition 5.11. The stability of a system S is defined as $\sigma(S) = 1 - \Pi_{r \in S} P^{*b}_{\mathrm{mod}}(r)$

Model 2

Although model 1b is useful to define stability, it does not proceed in an additive way with the modification costs. For instance, it is more intuitive to proceed bottom-up as follows: if we have a modification at component r, we have to *add* the cost of all the components which depend on it, as follows:

Definition 5.12. The *accumulate* cost of a component r is defined as:

$$\mathrm{AcCost}_{\mathrm{mod}}(r) = \Sigma_{a \in Desc(r)} \mathrm{Cost}_{\mathrm{mod}}(a)$$

And once again, the cost of maintenance of a system S could be defined as:

Definition 5.13. $\mathrm{Cost}^{2}_{\mathrm{maint}}(S) = \Sigma_{r \in S} P_{\mathrm{mod}}(r) \cdot \mathrm{AcCost}_{\mathrm{mod}}(r)$

And the results are quite different in this case: $\mathrm{AcCost}_{\mathrm{mod}}(r_1)=24$, $\mathrm{AcCost}_{\mathrm{mod}}(r_2)=13$, $\mathrm{AcCost}_{\mathrm{mod}}(r_3)=13$, $\mathrm{AcCost}_{\mathrm{mod}}(r_4)=8$, $\mathrm{AcCost}_{\mathrm{mod}}(r_5)=6$, that gives $\mathrm{Cost}^{2}_{\mathrm{maint}}(S) = 29.4$.

VI. SYSTEM TOPOLOGIES AND MAINTENANCE COST

Section 4 presented a method to obtain a validation (or predictiveness) measure for a software system, using reinforcement propagation. This measure is inversely related to the modification probability. Section 5 have introduced a dependency relation where modification propagates. As we said, the major problem of this dependency relation is that it is difficult to obtain. In general, when a component r is modified, the set of components which are to be modified depends mostly on the utilisation rate from the other components. This use rate is precisely what determines reinforcement. This insight motivates the following assignment:

Assumption 6.1. The modification dependency graph, determined by relation $\lrcorner$, usually top-down, matches *reversely* with the validation reinforcement graph, usually bottom-up.

Although this assumption is controvertible, it has many advantages as a working approximation,

- modification dependencies can be determined by the course of reinforcement.
- conversely, the course of reinforcement, which is extremely variable and uncertain for static models, can be approximated by the graph of modification dependencies.

On the other hand, this approximation has also some inconveniences. Not all granularities admit this matching. Moreover if one tries to mix up different granularity in both ways, for instance, using a procedural granularity to assign reinforcement and using an object-oriented granularity for modification dependencies, the results may be useless.

The final justification of this assignment is that it allows a theoretical study of the trade-off between validation (or predictiveness) and modifiability. More concretely, in order to obtain a validated (reinforced) system, a component should be used in the greater number of cases (and other modules) as possible. However, this would compromise modifiability, because any simple modification would propagate to an enormous number of other components.

There is a long debate about the convenience of high fan-in and low fan-out and vice-versa. The slogan of reusability is keep fan-out high and keep fan-in low. The slogan of modification in inheritance is to avoid a great number of children. This discussion is somehow paradoxical because for every dependency which goes out from a component, there is another component where it arrives to. In other words, mean fan-in is always equal to mean fan-out. So, it is more sensible to talk about high or low connectivity or, more meaningfully, to talk about topologies.

Intuitively, a hierarchical arrangement of dependencies eases the modification of the leaves situated at the bottom without the modification of the leaves at the top. This was recognised by (Parnas 1972) long ago: "*The partial ordering gives us two additional benefits. First, parts of the system are benefited (simplified) because they use the service of [upper] levels. Second, we are able to cut off the [lower] levels and still have a usable and useful product. [...]. The existence of the hierarchical structure assures us that we can "prune" off the [lower] levels of the tree and start a new [reversed] tree on the old trunk. If we had designed a system in with the "[high] level" modules made some use of the "[low] level" modules, we would not have the hierarchy, we would find it much harder to remove portions of the systems.".* However, one can wonder if the shape of this graph should be tree-like or root-like, the latter implicitly advocated by Parnas. This motivates a more detailed analysis of configurations of a given program *P*.

Definition 6.2. The Bottom or Minimal Set of a program *P*, denoted Bot_P, is composed of every component $b \in P$ such that $\neg\exists c \in P$, $c \neq b$, such that $c \dashv b$. In other words, $Bot_P = \{ b : Des(b) = \{ b \}\}$.

Definition 6.3. The Top or Maximal Set of a program *P*, denoted Top_P, is composed of every component $t \in P$ such that $\neg\exists c \in P$, $c \neq t$, such that $t \dashv c$. In other words, $Top_P = \{ t : Asc(t) = \{ t \}\}$.

According to the previous characteristics, we are going to study five different *topologies*:

- Topology 1: "Horizontal": No dependencies at all. Obviously, $P = Bot_P = Top_P$. We will consider the following extreme cases:

a) Compensated. $\forall i\ \rho\rho(c_i) = n / m$.

b) With exceptions. $\exists j\ \rho\rho(c_j) = n - m + 1$ and the rest are exceptions (or patches) $\rho\rho(c_i) = 1$, $i \neq j$.

- Topology 2: "Vertical": The dependency relation $\lrcorner$ obeys a full order relation $<$, $\forall\ c_1, c_2 \in P$, $c_1 \neq c_2$, then $\neg(c_2 < c_1) \leftrightarrow c_1 < c_2$. There is a unique top component $t \in Top_P$, i.e., card(Top_P) = 1. There is a unique bottom component $b \in Bot_P$, i.e., card(Bot_P) = 1. From here, the following properties hold, $\forall\ c \in P$, $c \neq t$, then $c < t$ and $\forall\ c \in P$, $c \neq b$, then $b < c$.
- Topology 3: Lattice: The dependency relation $\lrcorner$ obeys a partial order relation $<$. There is a unique top component t such that $\forall\ c \in P$, $c \neq t$, then $c < t$ and a unique bottom component b such that $\forall\ c \in P$, $c \neq b$, then $b < c$. We will consider three extreme cases:

a) A unary lattice which corresponds to topology 2.

b) Wide lattice with depth = 3, where the middle level has $m-2$ components.

c) Binary lattice. We assume $m = 2^k - 1 + 2^{k-1} - 1 = 3 \cdot (2^{k-1}) - 2$, with k being a natural number.

- Topology 4: Tree: The dependency relation obeys a partial order relation $<$ with no unique top element ($card$(Top$_P$) ≥ 1). There is a unique bottom component b such that $\forall\ c \in P$, $c \neq b$, then $b < c$. We will consider three prototypical cases:

a) One vertical branch (i.e. $card$(Top$_P$) = 1). Equivalent to topology 2.

b) Wide tree with depth = 2, where the top level has $m-1$ components.

c) Binary tree. We assume $m = 2^k - 1$, with k being a natural number.

- Topology 5: Root (inverse tree): The dependency relation obeys a partial order relation $<$ with no unique bottom component ($card(Bot_P) \geq 1$). There is a top component t such that $\forall\ c \in P$, $c \neq t$, then $c < t$. We will consider three prototypical cases:

a) One vertical branch (i.e. $card(Bot_P) = 1$). Equivalent to topology 2.

b) Wide root with depth = 2, where the bottom level has $m-1$ components..

c) Inverse binary tree. We assume $m = 2^k - 1$, with k being a natural number.

Cases b) and c) will be studied in two ways: compensated and with exceptions. We will assume that all components have the same size and that all examples have the same significance. From here,

Theorem 6.5. Given n examples $e_1, e_2, ..., e_n$, a program of m components arranged under topologies 2, 3 or 4 has the following properties:

- For every component c from P, the pure reinforcement $\rho\rho(c)$ is n, and the normalised reinforcement $\rho(c) = 1 - 2^{-n}$.
- For every example e_i the course $\chi(e_i) = (1-2^{-n})^m$. Hence, the mean course is $m\chi(E) = (1-2^{-n})^m$.
- For every component c, the isolated probability of modification $P_{\text{mod}}(c)$ is 2^{-n} and $P_{\text{mod}}(P) = 1 - (1-2^{-n})^m = 1 - m\chi(E)$.

Proof. Topologies 2, 3 and 4 have a unique bottom b, and obviously, all the examples are covered by this bottom component b. Hence, $\rho\rho(b) = n$, and $\rho(b) = 1$. Since we consider static dependencies, and all the components are required by b, because $\forall c \in P$, $c \neq b$, then $b \lrcorner\ c$, they all have the same pure reinforcement $\rho\rho(c) = n$. The rest of properties follow from here by applying previous definitions. □

Topologies 1 and 5 may have an overlap in the coverings of the bottom set, i.e., $\forall\ b_i \in Bot_P$, $\Sigma\rho\rho(b_i) > n$. This kind of redundancy is usually eliminated in software systems (except when a voting method is used to increase reliability), so we will consider $\forall\ b_i \in Bot_P, \Sigma\ \rho\rho(b_i) = n$.

Given n examples, it is shown that a program P of m components with $n \gg m$ such that $\forall r \in P$, $\text{Cost}_{\text{mod}}(r) = U_{cost}$, it is shown in the appendix that model 2 brings forward the following maintenance costs:

Topology	*Maintenance cost*
1a) Horizontal compensated	$O(2^{-n/m} \cdot m)$
1b) Horizontal with Exceptions	$O(m)$
2) Vertical	$O(2^{-n} \cdot m^2)$
3b) Lattice with depth = 3	$O(2^{-n} \cdot m)$
3c) Binary Lattice	$O(2^{-n} \cdot m \cdot \log_2 m)$
4b) Tree with depth = 2	$O(2^{-n} \cdot m)$
4c) Binary Tree	$O(2^{-n} \cdot m \cdot \log_2 m)$
5b-i) Inverse Tree with depth = 2 and compensated	$O(2^{-n/m} \cdot m)$
5c-i) Binary Inverse Tree and compensated	$O(2^{-n/m} \cdot m \cdot \log_2 m)$
5b-ii) Inverse Tree with depth = 2 with exceptions	$O(m)$
5c-ii) Binary Inverse Tree with exceptions	$O(m \cdot \log_2 m)$

Fig 6.1. Results of Maintenance Costs for Different Topologies
***n**= no. of examples / tests cases, **m**= no. of components*

Having in mind the assumptions and approximations that have led us to use $\text{Cost}^2_{\text{maint}}(S)$ for approximating the maintainability of a software system, we can extract some conclusions:

First of all, when the software system has exceptions or patches, which are used for few examples (topologies 1b, 5b-ii and 5c-ii), which have not been validated, the maintenance cost depends almost exclusively on them, in the way that the cost is asymptotically independent from *n*, the factor that reduces the cost.

From all the rest of compensated topologies, where reinforcement is distributed uniformly, the results are not so despair. However there is a great asymptotic difference between a wide tree or a lattice and a binary inverse tree. This remarks that topologies should be confluent or 'conciliated' at the bottom, much more like a tree than like a root. In other words, components at the bottom should behave in a broad way and not in a specialised way, something that may be interpreted very differently depending on what one could think of a component. Finally, other more intuitive consequence is that wide topologies are better than thin ones, because of modification propagation.

The strongest result derivable from figure 6.1 is that compression, i.e. increasing *n* over *m*, is an excellent way to reduce maintenance cost. In relation to the same sample, simple systems are more reinforced because the ratio of examples by piece of software is greater, so validation is higher. In the other way, modification is much easier. Although this is well known since long ago, recently, however, there have been claims about considering software engineering as compression [Wolff 1994], supported by the idea of learning as compression. However, a very compressed model can be spoilt by some patches, something that it is plainly seen in topologies 1b, 5b-ii and 5c-ii.

Finally, as we have said, more things can be inferred from this study if the components are particularised to real objects: classes, functions, modules, etc. For instance, if the components are classes, one can identify these results with four OO software quality metrics such as "lack of cohesion of methods" (increase granularity when possible) , "coupling between objects" (decrease granularity when possible), "depth of inheritance tree" and "number of children". Other interpretations are at first sight less intuitive or even contradictory. Inheritance, which is widely accepted, determines a topology of type 5c-i). However, the dependency relation is not only conditioned by inheritance relationship but also by associations, aggregations, etc. Moreover, multiple inheritance helps to change the topology to types 3c) or 4c). Ultimately, polymorphism represents the previous idea of confluence or avoidance of specialisation at the bottom. In some way, polymorphism tries to 'pump up' reinforcement.

VII. REFINEMENTS

After the interpretation of the results of the previous section, it is still possible to make some considerations and extensions to make the models more realistic. As usual, this increased accuracy entails less manageability, and hence, less general results can be obtained if the following refinements are considered. In particular, we will briefly assess three important issues in a topology with propagation: how large the components are, how the propagation can be parameterised bottom-up and how the propagation can be parameterised top-down.

A. Modelling granularity

The idea of increasing n over m may suggest gross granularities over finer ones. This was avoided in section 4.2.2, by including in the reinforcement measures a factor inversely related to the size of the components. However, the cost of modifying one component is not usually linearly related to its size. In the previous section we used the assumption on $\forall r \in S$, $\mathrm{Cost}_{\mathrm{mod}}(r) = U_{cost}$. In general, it would be interesting to introduce another variable in addition to n and m, namely $s = size(S)$. In this way, we can use the following more accurate approximation for $\mathrm{Cost}_{\mathrm{mod}}(r)$:

Definition 7.1. The cost of modification of a component can be related to its size in an exponential way: $\mathrm{Cost}_{\mathrm{mod}}(r) = a_{\mathrm{mod}} \cdot b_{\mathrm{mod}}^{size(r)}$ where a_{mod} and b_{mod} should be experimentally or theoretically adjusted.

Consequently, the incidence of granularity in the previous topologies could be studied assuming that all components are of the same size, giving: $\mathrm{Cost}_{\mathrm{mod}}(r) = a_{\mathrm{mod}} \cdot b_{\mathrm{mod}}^{size(S)\,/\,m}$.

Finally, if granularity is considered, reusability could be studied too. Assertions such as *keep methods small* (Rumbaugh et al. 1991) are more related with granularity than with total length of the program.

B. Bottom-up use rate

In the previous section we made the assignment between the bottom-up and top-down topologies. However, both of them were saturated, i.e., all the 'links' had total capacity.

In this subsection we extend the propagation bottom-up; any dependency link $r_1 \lrcorner r_2$ is assigned a real value $0 \le s \le 1$ known as saturation or mean use rate, and represents the percentage of examples that use r_1 that still require the use of r_2. We denote this extension by $r_1 \lrcorner^{s} r_2$.

Definition 7.2. The total saturation of a component r, denoted by $Sat(r)$ can be defined as

$$Sat(r) = \Sigma_{r_i \in DAsc(r)} \{s_i : r \lrcorner^{s_i} r_i\}$$

A more realistic study of reinforcement propagation can be done by using the mean total saturation of a component and the deviations as parameters. In practice, and if the topology structure is given by modification dependencies, the values of saturation could be acquired (or *learned*). Learning techniques can be inspired in neural networks where the 'wiring' is given and the weights are learned using the training sample.

The study of the saturation parameter could reveal that there can be two kinds of 'losses' of reinforcement (which are compatible with each other). A simple dependency can be *weakened* because a component is able to make do for some examples without some of its dependencies. The other loss is more natural, and it happens when a component has more than one direct ascendant (i.e, $DAsc(r) > 1$), then the reinforcement is usually *shared* among the dependencies. This clearly restricts high fan-ins and fan-outs.

Finally, this extension allows the appearance of exceptions at higher levels (with components that are scarcely used). In this way, the results of topologies 3 and 4 can vary a great deal by changing the mean saturation of the dependencies, because exceptions are much worse on the top. This explains why inheritance, which is a kind of specialisation, tries to place exceptions (i.e. subclasses) on the bottom, each time that there is an impossibility (or inability) to 'conciliate' an anomaly. Under this view it seems that multiple inheritance presents some advantages over single inheritance, because multiple inheritance improves saturation (whereas single inheritance uses other dynamic dependencies to solve the same problem).

C. Top-down information hiding

As we have seen, there are two ways to decrease maintenance cost. We have centred our measurement in the first way, the reduction of the probability of modification. The other way is to reduce the consequence of each modification. Throughout this paper, we have presented modification propagation as a snowball. This means that a modification at the top propagates to the bottom, without *friction*.

In other words, we have not parameterised the effect of each modification or, more properly, the modification factor of software. Almost three decades ago, (Parnas 1971) introduced “information hiding” precisely for this. Modules are generated to encapsulate “*the design decisions which are questionable and likely to change under many circumstances*”. This is obtained in two steps:

1) One must evaluate the “design decisions which are likely to change”.
2) “Each module is then designed to hide such a decision from the others”. This is done by generalising the prototype, hiding these decisions or weak parts of the model from the interface.

If this is made judiciously, most of the changes (the more likely) will not propagate outside the component. Precisely, objects or concepts (in the modern conceptual modelling fashion), from a cognitive point of view are used to model those sets of features from reality that are coherent and represented by some properties which are not likely to change. The drift from procedural, modular, to object-oriented programming and beyond has centred on moving the interface into more stable components.

We have different possibilities for defining a hiding (or friction) factor h: for each link, for each component or for the whole system. If we use the second possibility, each component a_i must be assigned a different value h_i:

Definition 7.3. The probability of modification of a component denoted as $\mathrm{Ph}_{\mathrm{mod}}$, can be extended as: $\mathrm{Ph}_{\mathrm{mod}}(r) = 1 - \mathrm{P}_{\mathrm{mod}}(r) \cdot \{ \prod_{a_i \in DAsc(r)} h_i \cdot \overline{\mathrm{Ph}}_{\mathrm{mod}}(a_i) \}$

In general, it may be difficult to estimate these h_i, because it depends on the ability to distinguish the lasting parts from the ephemeral ones (or more likely to change). A good solution to this would be the study of the resulting topologies from finer topologies to grosser ones.

VIII. Conclusions

This paper has shown that a great number of characteristics of software can be theoretically studied using ML analogies and techniques. In our case, reinforcement is used to obtain a probability of modification, where many other measures are derived from, such as system stability and maintainability measures for different topologies. The goodness of different topologies can be theoretically established as it has been done and empirically refined by implementing automated measuring tools in each component.

In order to apply these measurements, it is required the establishment of the notions of accordance, component granularity and example for a particular system. A proper choice of these parameters will surely depend on experience.

Appendix

This appendix includes the results of maintainability for the different topologies presented in section VI.

<u>Topology 1</u>. $\mathrm{card}(Bot_P) = \mathrm{card}(P)$. So $\Sigma_{i=1..m}\, \rho\rho(c_i) = n$.

- For the case a) we have that $\rho(c) = 1 - 2^{-n/m}$ and for every example [illegible] course $\chi(e_i) = (1 - 2^{-n/m})$. The isolated and related probabilities of modification are the same, exactly, $P_{\mathrm{mod}}(c) = P^{*b}_{\mathrm{mod}}(c) = 2^{-n/m}$. From here, $\mathrm{P}_{\mathrm{mod}}(P) = 1 - (1 - 2^{-n/m})^m$.
  $\forall i\ \mathrm{AcCost}_{\mathrm{mod}}(c_i) = \Sigma_{a \in Des(r)}\, \mathrm{Cost}_{\mathrm{mod}}(a) = \mathrm{Cost}_{\mathrm{mod}}(a) = \mathrm{U}_{\mathrm{mod}}$ and $\mathrm{Cost}^2_{\mathrm{maint}}(P) = \Sigma_{r \in S}\, \mathrm{P}_{\mathrm{mod}}(r) \cdot \mathrm{AcCost}_{\mathrm{mod}}(r) = m \cdot 2^{-n/m} \cdot \mathrm{U}_{\mathrm{mod}} \in \mathbf{O}(\boldsymbol{m} \cdot \mathbf{2}^{-\boldsymbol{n/m}})$.
- For the case b) we have that $\rho(c_j) = 1 - 2^{-(n-m+1)}$ and $\rho(c_i) = 0.5$ iff $i \neq j$. We have the course $\chi(e_j) = (1 - 2^{-(n-m+1)})$ and $\chi(e_i) = 0.5$ iff $i \neq j$. The mean course is $[(m-1)/2 + (n-m+1) \cdot (1 - 2^{-(n-m+1)})\,] / n = [(1-m)/2 + n + (m-n-1) \cdot (2^{-(n-m+1)})\,] / n$. If $n >> m$ this is approximately $(1 - 2^{-n})$. The isolated probability of modification is, $P_{\mathrm{mod}}(c_j) = 2^{-(n-m+1)}$ and $P_{\mathrm{mod}}(c_i) = 0.5$ iff $i \neq j$. From here, $\mathrm{P}_{\mathrm{mod}}(P) = 1 - (1 - 2^{-(n-m+1)})^{\,n-m+1} \cdot$

$(1/2)^{m-1} = 1 - (1 - 2^{-(n-m+1)})^{n-m+1} \cdot 2^{1-m}$. Again, since $n >> m$ and n is great then $P_{mod}(P) \cong 1 - 2^{1-m}$.

$\forall i$ $AcCost_{mod}(c_i) = \Sigma_{a \in Desc(r)} Cost_{mod}(a) = Cost_{mod}(a) = U_{mod}$ and $Cost^2_{maint}(P) = \Sigma_{r \in S} P_{mod}(r) \cdot AcCost_{mod}(r) = ((n-m+1) \cdot 2^{-(n-m+1)} + (m-1) \cdot 1/2) \cdot U_{mod}$. Since the first term $(n-m+1) \cdot 2^{-(n-m+1)}$ is always ≤ 1 if $n > m$, then $Cost^2_{maint}(P) \in \mathbf{O(\mathit{m})}$

<u>Topology 2</u>. From theorem 6.5, $\rho\rho(c) = n$, and $\rho(c) = 1 - 2^{-n}$, the course $\chi(e_i) = (1-2^{-n})^m$ for all e_i. For all c_i, the isolated probability of modification $P_{mod}(c_i)$ is 2^{-n} and $P_{mod}(P) = 1 - (1 - 2^{-n})^m$.

- Without loss of generality in this topology, $c_1 = b$ and $c_m = t$, with $c_i < c_{i+1}$ $\forall i$ $1 \leq i < m$.

$\forall i$ $AcCost_{mod}(c_i) = \Sigma_{a \in Des(c_i)} Cost_{mod}(a) = i \cdot U_{mod}$ and $Cost^2_{maint}(P) = \Sigma_{r \in S} P_{mod}(r) \cdot AcCost_{mod}(r) = 2^{-n} \cdot \Sigma_{i=1..m} i \cdot U_{mod} = 2^{-n} \cdot m \cdot (m+1)/2 \cdot U_{mod} \in \mathbf{O(2^{-\mathit{n}} \cdot \mathit{m}^2)}$

<u>Topology 3</u>. From theorem 6.5, $\rho\rho(c) = n$, and $\rho(c) = 1 - 2^{-n}$, the course $\chi(e_i) = (1-2^{-n})^m$ for all e_i. For all c_i, the isolated probability of modification $P_{mod}(c_i)$ is 2^{-n} and $P_{mod}(P) = 1 - (1 - 2^{-n})^m$.

- For the case b) $AcCost_{mod}(b) = U_{mod}$ and $AcCost_{mod}(t) = m \cdot U_{mod}$ and $AcCost_{mod}(r) = 2 \cdot U_{mod}$ iff $r \neq t$ and $r \neq b$. So, $Cost^2_{maint}(P) = \Sigma_{r \in S} P_{mod}(r) \cdot AcCost_{mod}(r) = 2^{-n} \cdot (1 + m + (m-2) \cdot 2) \cdot U_{mod} \cong 2^{-n} \cdot 3m \cdot U_{mod} \in \mathbf{O(2^{-\mathit{n}} \cdot \mathit{m})}$
- For the case c) it is obvious that $AcCost_{mod}(t) = m \cdot U_{mod}$. Since $m = 2^k - 1 + 2^{k-1} - 1$, we have $2k-1$ levels: k levels with increasing width and $k-1$ levels with decreasing width. For the first k levels, j-numbered top-down from 1 to k, we have 2^{j-1} components on each level, and $AcCost_{mod}(r^j) = (2^{k-j+1} - 1 + 2^{k-1-j+1} - 1 + (j-1)) \cdot U_{mod} = (3 \cdot 2^{k-1-j+1} - 3 + j) \cdot U_{mod}$

  For the next $k-1$ levels, i-numbered top-down from $k-1$ to 1, we have 2^{i-1} components on each level, and $AcCost_{mod}(r^i) = i \cdot U_{mod}$

Finally, $\text{Cost}^2_{\text{maint}}(P) = \Sigma_{r \in S} \text{P}_{\text{mod}}(r) \cdot \text{AcCost}_{\text{mod}}(r) = 2^{-n} \cdot (\Sigma_{j=1..k} 2^{j-1} \cdot [3 \cdot 2^{k-1-j+1} - 3 + j] + \Sigma_{i=1..k-1} 2^{i-1} \cdot i) \cdot \text{U}_{\text{mod}} = 2^{-n} \cdot (\Sigma_{j=1..k} [3 \cdot 2^{k-1} - 3 \cdot 2^{j-1} + j \cdot 2^{j-1})] + \Sigma_{i=1..k-1} 2^{i-1} \cdot i) \cdot \text{U}_{\text{mod}} = 2^{-n} \cdot ([k \cdot 3 \cdot 2^{k-1} - 3 \cdot 2^k + 2 + \Sigma_{i=1..k-1} j \cdot 2^{j-1})] + \Sigma_{i=1..k-1} 2^{i-1} \cdot i) \cdot \text{U}_{\text{mod}}$

By using the approximation $\Sigma_{l=1..p} 2^{l-1} \cdot l \cong p \cdot 2^p$., we have: $\text{Cost}^2_{\text{maint}}(P) \cong 2^{-n} \cdot ([k \cdot 3 \cdot 2^{k-1} - 6 \cdot 2^{k-1} + 2 + (k-1) \cdot 2^{k-1})] + (k-1) \cdot 2^{k-1}) \cdot \text{U}_{\text{mod}} = 2^{-n} \cdot ((k \cdot 3 + 2k - 2 - 6) \cdot 2^{k-1} + 2) \cdot \text{U}_{\text{mod}} = 2^{-n} \cdot ((5k - 8) \cdot 2^{k-1} + 2) \cdot \text{U}_{\text{mod}}$

Since $m = 2^k - 1 + 2^{k-1} - 1$, then $2^{k-1} = (m+2)/3$ and $k = \log_2 [(m+2)/3 + 1]$ then, $\text{Cost}^2_{\text{maint}}(P) \cong 2^{-n} \cdot ((5 \log_2 [(m+2)/3 + 1] - 8) \cdot (m+2)/3 + 2) \cdot \text{U}_{\text{mod}} \cong 2^{-n} \cdot 5/3 \cdot m \cdot \log_2 (m/3) \cdot \text{U}_{\text{mod}} \in \mathbf{O(2^{-n} \cdot \mathit{m} \cdot \log_2 \mathit{m})}$

Topology 4. From theorem 6.5, $\rho\rho(c) = n$, and $\rho(c) = 1 - 2^{-n}$, the course $\chi(e_i) = (1-2^{-n})^m$ for all e_i. For all c_i, the isolated probability of modification $P_{\text{mod}}(c_i)$ is 2^{-n} and $P_{\text{mod}}(\text{P}) = 1 - (1 - 2^{-n})^m$.

- For the case b) $\text{AcCost}_{\text{mod}}(b) = \text{U}_{\text{mod}}$ and $\text{AcCost}_{\text{mod}}(t) = 2 \cdot \text{U}_{\text{mod}}$ for all $t \in Top_P$. So, $\text{Cost}^2_{\text{maint}}(P) = \Sigma_{r \in S} \text{P}_{\text{mod}}(r) \cdot \text{AcCost}_{\text{mod}}(r) = 2^{-n} \cdot (1 + (m-1) \cdot 2) \cdot \text{U}_{\text{mod}} \cong 2^{-n} \cdot 2m \cdot \text{U}_{\text{mod}} \in \mathbf{O(2^{-n} \cdot \mathit{m})}$.
- For the case c) we directly have that $\text{AcCost}_{\text{mod}}(t) = k \cdot \text{U}_{\text{mod}}$. Since $m = 2^k - 1$, there are k levels with decreasing width, i-numbered top-down from k to 1, and 2^{i-1} components on each level, so $\text{AcCost}_{\text{mod}}(r^i) = i \cdot \text{U}_{\text{mod}}$

  Finally, $\text{Cost}^2_{\text{maint}}(P) = \Sigma_{r \in S} \text{P}_{\text{mod}}(r) \cdot \text{AcCost}_{\text{mod}}(r) = 2^{-n} \cdot (\Sigma_{i=1..k} 2^{i-1} \cdot i) \cdot \text{U}_{\text{mod}}$

  By using again the approximation $\Sigma_{l=1..p} 2^{l-1} \cdot l \cong p \cdot 2^p$., we have: $\text{Cost}^2_{\text{maint}}(P) \cong 2^{-n} \cdot (k \cdot 2^k) \cdot \text{U}_{\text{mod}}$

  Since $m = 2^k - 1$, then $2^k = m+1$ and $k = \log_2 [m+1]$ then,

  $\text{Cost}^2_{\text{maint}}(P) \cong 2^{-n} \cdot \log_2 [m+1] \cdot (m+1) \cdot \text{U}_{\text{mod}} \in \mathbf{O(2^{-n} \cdot \mathit{m} \cdot \log_2 \mathit{m})}$

Topology 5. Cases b) and c) will be studied with these two extreme conditions:

i) Compensated: $\forall c_i \in \text{Bot}_\text{P}$ then $\rho\rho(c_i) = n \,/\, Card(Bot_P)$.

ii) With exceptions: $\exists j \in \text{Bot}_\text{P}$ $\rho\rho(c_j) = n - Card(Bot_P) + 1$ and the rest $c_i \in \text{Bot}_\text{P}$ have $\rho\rho(c_i) = 1$, $i \neq j$.

- For the case b)-i) we have that $Card(Bot_P) = m\text{-}1$, so $\rho(t) = 1 - 2^{-n}$ for t and $\rho(r) = 1 - 2^{-n/(m-1)}$ iff $r \neq t$. For every example e_i the course $\chi(e_i) = (1-2^{-n/(m-1)}) \cdot (1 - 2^{-n})$. The isolated probabilities are $P_{\text{mod}}(t) = 2^{-n}$ and $P_{\text{mod}}(r) = 2^{-n/(m-1)}$ iff $r \neq t$. $P_{\text{mod}}(P) = 1 - (1 - 2^{-n})^{\text{m}-1} \cdot (1 - 2^{-n/(m-1)})$.

  On the other hand, $\text{AcCost}_{\text{mod}}(t) = 2\cdot\text{U}_{\text{mod}}$ and $\text{AcCost}_{\text{mod}}(b) = \text{U}_{\text{mod}}$ iff $r \neq t$

  So, $\text{Cost}^2_{\text{maint}}(P) = \Sigma_{r \in S}\, \text{P}_{\text{mod}}(r) \cdot \text{AcCost}_{\text{mod}}(r) = (2^{-n} \cdot 2 + (m-2)\cdot 2^{-n/(m-1)}) \cdot\text{U}_{\text{mod}} \in$ $\mathbf{O}(\boldsymbol{m}\cdot \mathbf{2}^{-\boldsymbol{n/m}})$

- For the case c)-i) we have that $\text{Card}(Bot_P) = (m + 1) / 2$. We have $\rho(t) = 1 - 2^{-n}$ for t and for the k levels of the inverse tree, numbered top-down from 1 to k, we have $\rho(r^j) = 1 - 2^{-n/(2\text{^}(j-1))}$. For every example e_i the course $\chi(e_i) = \Pi_{j=1..k}\, (1 - 2^{-n/(2\text{^}(j-1))}) \leq 1 - 2^{-n/(2\text{^}(k-1))} = 1 - 2^{-2n/(m-1)}$.

  We have that the probabilities of modification are $P_{\text{mod}}(t) = 2^{-n}$. Since $m = 2^k - 1$, we have k levels with increasing width, numbered top-down from 1 to k, we have 2^{j-1} components on each level, and $P_{\text{mod}}(r^j) = 2^{-n/(2\text{^}(j-1))}$. We have $P_{\text{mod}}(P) = (1 - \Pi_{j=1..k} \cdot (1 - 2^{-n/(2\text{^}(j-1))})^{2\text{^}(j-1)})$.

  It is obvious that $\text{AcCost}_{\text{mod}}(t) = m\cdot\text{U}_{\text{mod}}$. Since $m = 2^k - 1$, we have $2k-1$ levels: k levels with increasing width, j-numbered top-down from 1 to k, we have 2^{j-1} components on each level, and $\text{AcCost}_{\text{mod}}(r^j) = (2^{k-j+1} - 1 + 2^{k-1-j+1} - 1 + (j-1)) \cdot\text{U}_{\text{mod}} = (3 \cdot 2^{k-1-j+1} - 3 + j) \cdot\text{U}_{\text{mod}}$

  Finally, $\text{Cost}^2_{\text{maint}}(P) = \Sigma_{r \in S}\, \text{P}_{\text{mod}}(r) \cdot \text{AcCost}_{\text{mod}}(r) = \Sigma_{j=1..k}\, 2^{j-1} \cdot 2^{-n/(2\text{^}(j-1))} \cdot[3 \cdot 2^{k-1-j+1} - 3 + j] \cdot\text{U}_{\text{mod}} = \Sigma_{j=1..k}\, 2^{-n/(2\text{^}(j-1))} \cdot[3\cdot 2^{k-1} - 3\cdot 2^{j-1} + j\cdot 2^{j-1}] \cdot\text{U}_{\text{mod}}$

  Since both factors increase very quickly with j, and using the approximation $\Sigma_{l=1..p}\, 2^{l-1} \cdot l \cong p \cdot 2^p$, we have that we can roughly approximate to: $\text{Cost}^2_{\text{maint}}(P) \cong 2^{-n/(2\text{^}k)} \cdot k\cdot 2^k \cdot\text{U}_{\text{mod}}$

  Since $m=2^k-1$, then $2^k=m+1$ and $k=\log_2(m+1)$, so $\text{Cost}^2_{\text{maint}}(P) \cong 2^{-n/m} \cdot m\cdot\log_2 m \cdot \text{U}_{\text{mod}} \in \mathbf{O}(\mathbf{2}^{-\boldsymbol{n/m}} \cdot\boldsymbol{m} \cdot \mathbf{log_2}\boldsymbol{m})$

- For the case b)-ii) we have that Card(Bot_P) = m–1, so $\rho(t) = 1-2^{-n}$ for t and $\exists j \in \mathrm{Bot_P}$ $\rho\rho(c_j)$ = n–Card($\mathrm{Bot_P}$)+1 = n–m+2 and the rest m–2 components $c_i \in \mathrm{Bot_P}$ have $\rho\rho(c_i)=1$, i≠j. There are m–2 examples with course $\chi(e) = (1-2^{-1})\cdot(1-2^{-n})$. The rest n–m+2 examples have $\chi(e) = (1-2^{-(n-m+2)})\cdot(1-2^{-n})$.

  The isolated probabilities are $P_{\mathrm{mod}}(t) = 2^{-n}$ and $\exists j \in \mathrm{Bot_P}$ $P_{\mathrm{mod}}(r) = 2^{-(n-m+2)}$ and the rest $m-2$ components $c_i \in \mathrm{Bot_P}$ have $P_{\mathrm{mod}}(r) = 0.5$. We have $P_{\mathrm{mod}}(P) = (1-(1-2^{-n})\cdot(1-2^{-(n-m+2)})\cdot(1-0.5)^{(m-2)})$.

  On the other hand, $\mathrm{AcCost_{mod}}(t) = 2\cdot\mathrm{U_{mod}}$ and $\mathrm{AcCost_{mod}}(b) = \mathrm{U_{mod}}$ iff $r \neq t$

  So, $\mathrm{Cost}^2_{\mathrm{maint}}(P) = \Sigma_{r \in S}\, \mathrm{P_{mod}}(r)\cdot\mathrm{AcCost_{mod}}(r) = (2^{-n}\cdot 2 + 2^{-(n-m+2)} + (m-2)\cdot 0.5\cdot 2)\cdot\mathrm{U_{mod}}$. Since the first two terms are always ≤ 1 if $n > m$ and n great, then $\mathrm{Cost}^2_{\mathrm{maint}}(P) \in \mathbf{O}(\boldsymbol{m})$.

- Case c)-ii) would be lengthy to study directly. However, the results are very similar to the case of considering a vertical propagation like topology 2 on one side and the biggest subtree of topology c)- i) on the other side. This latter part, composed of ((m+1) / 2) –1 nodes, will dominate the whole result because it is reinforced by (m+1)/4 examples only, one for each component of Bot_P. (The former part is $O(2^{-n}\cdot m^2)$).

  Using the results of topology c)- i) and changing n by (m+1)/4 we have:

  $\mathrm{Cost}^2_{\mathrm{maint}}(P) \cong 2^{-(m+1)/4m}\cdot m\cdot\log_2 m\cdot\mathrm{U_{mod}} \in \mathbf{O}(\boldsymbol{m}\cdot\mathbf{log_2}\boldsymbol{m})$.

ACKNOWLEDGEMENTS

I am in debt with Jesus Alcolea who motivated my interest in Fetzer's work by suggesting the relation between software and philosophy of science. The relevance of ML results and techniques to *interactive* software was recognised by Peter Wegner in a visit to my department in Spring 1998. Finally, I would also like to thank M.Carmen Juan and Rose Barreiro for their comments and corrections.

REFERENCES

Aha, D.W.: Lazy Learning, in Editorial Special Issue about "Lazy Learning" *AI Review*, v.11, 1-5, Feb. (1997).

Banerji, R.B.: Some insights into automatic programming using a pattern recognition viewpoint in Biermann, A.W.; Guiho, G.; and Kodratoff, Y. (Eds.): *Automatic program construction techniques*, Macmillan, (1984).

Barron, A.; Rissanen, J.; Yu, B.: The Minimum Description Length Principle in Coding and Modeling, *IEEE Transactions on Information Theory*, Vol. 44, No. 6, 2743-2760, October (1998).

Basili, V.R.; Selby, W.; Hutchens, D.H.: Experimentation in Software Engineering, *IEEE Transactions on Software Engineering*, vol. SE-12, no.7, pp. 733-743, July (1986).

Basili, V.R.: The Experimental Paradigm in Software Engineering, in Rombach, H.D.; Basili, V.R; Selby, R. *Experimental Software Engineering Issues: Critical Assessment and Future Directives*, LNCS 706, Springer-Verlag, August (1993).

Berry, D. M.; Lawrence, B.: Requirements Engineering, *IEEE Software*, March/April 1998, pp. 26-29.

Bertalanffy, L.V.: *Teoria generale dei sistemi*, Instituto Librario Internationales, Milano 1971.

Biermann, A.W.; Guiho, G.; and Kodratoff, Y. (Eds.): *Automatic program construction technique*, Macmillan, 1984.

Boehm, B.W.: *Software Engineering Economics,* Prentice Hall, 1981.

Booch, G., *Object-oriented Analysis and Design with Applications,* The Benjamin/Cummings Publishing Co., Inc., 1994.

Brooks, F.P.: No Silver Bullet, in *The Mythical Man-Month: Essays on Software Engineering* (2nd ed.) Addison Wesley Longman, Reading, Mass. 1995, pp. 179-203.

Chidamber, S.; Kemerer, C.: A Metrics Suite for Object-Oriented Design, *IEEE T. Software Eng.*, June, pp.476-492 (1994).

Colburn, T.R.: Program Verification, Defeasible Reasoning, and Two Views of Computer Science, *Minds & Machines* 1, 97-116, 1991.

Conte, S.D.; Dunsmore, H.E.; Shen, V.Y.: *Software Engineering Metrics Models*, The Benjamin/Cummings Pub. Co., 1986.

Cox, B.: *Object Oriented Programming, An Evolutionary Approach*, Addison Wesley 1987.

De Millo, R.; Lipton, R.J, Perlis, A.J.: Social Processes and Proofs of Theorems and Programs, *Communications of the ACM* 22 (5), 271-280, (1979).

Dijkstra, E.W.: Notes on Structured Programming, in O.Dahl *et al.* (eds.) *Structured Programming*, Academic Press 1972.

Fetzer, J.H.: Program Verification: The Very Idea, *Communication of the ACM* 31(9), 1048-1063, (1988).

Fetzer, J.H.: Philosophical Aspect of Program Verification, *Minds and Machines* 1, 197-216, (1991).

Finkelstein, A.: Re-use of formatted requirements specifications, *Software Engineering J. 3*, 3, 186-197, Sept. (1988).

Fouqué, G.; Matwin, S.: A case-based approach to software reuse, *J. Intelligent Inform. Systems*, 2 (2): 165-197, (1993).

Genesereth, M.; Ketchpel, S.P.: Software Agents. *Communications of the ACM*, 37(7):48-53, (1994).

Gold, E.M.: Language Identification in the Limit, *Inform. and Control.*, 10, pp. 447-474, (1967).

Harrison, W.; Magel, K.; Kluczney, R.; DeKock, A.: Software Complexity Metrics and their application to maintenance, *IEEE Computer*, pp. 65-79, Sept. 1982.

Harrison W.: An Entropy-Based Measure of Software Complexity, IEEE T. Software Eng. 18, No.11, 1025-34, Nov (1992)

Hernández-Orallo, J.; Hernández-Orallo, E.: *Programación en C++*, Paraninfo 1993, 2nd Ed., Intl. Thompson Publishers, 1995.

Hernández-Orallo, J.: Constructive Reinforcement Learning, *International Journal of Intelligent Systems*, 15(3), 241-264, 2000.

Hernández-Orallo, J.; Ramírez-Quintana, M.J. A Strong Complete Schema for Inductive Functional Logic Programming. in Dzeroski, S.; Flach, P. (eds) "Inductive Logic Programming" Volume 1634 of the Lecture Notes in Artificial Intelligence (LNAI) series, pp. 116-127, Springer-Verlag 1999.

Hernández-Orallo, J.; Ramírez-Quintana, M.J. Software as Learning: Quality Factors and Life-Cycle Revised (English, 15 pp.), to be presented in Foundational Aspects of Software Engineering, Berlin (FASE'2000). To appear in the Lecture Notes in Computer Science (LNCS) series, Springer-Verlag 2000a.

Hernández-Orallo, J.; Ramírez-Quintana, M.J.: Predictive Software, *submitted*, 2000b

Hoare, C.A.R.: An Axiomatic Basis for Computer Programming, *Communications of the ACM* 12, 576-580, 583, (1969).

Jacobson, I.: The Use-Case Construct in Object-Oriented Software Engineering, *Scenario-Based Design: Envisioning Work and Technology in System Development*, J.Carroll, ed., John Wiley & Sons, New York, 1995, pp. 309-336.

Kaelbling, L.; Littman, M.; Moore, A.: Reinforcement Learning: A survey, *J. of AI Research*, 4, 237-285, (1996).

Kearns, M.; Mansour, Y.; Ng, A.Y.; Ron, D.: An experimental and theoretical comparison of model selection methods, *Machine Learning*, to appear.

Kuhn, T.S.: *The Structure of Scientific Revolution*, University of Chigago 1970.

Lieberherr K.J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Pub. Co, 1996.

Lorenz, M.; Kidd, J.: *Object-Oriented Software Metrics*, Prentice Hall Publishing, 1994.

Loucopoulos, P.; Karakostas, V.: *System Requirements Engineering*, McGraw-Hill, New York, 1995.

Maes, P.: Intelligent Software, *Scientific American* 273 (3), September, (1995).

Merhav, N.; Feder, M.: Universal Prediction, *IEEE Transactions on Information Theory*, Vol. 44, No. 6, 2124-2147, October (1998).

Mitchell, T.M.: *Machine Learning*, McGraw-Hill Series in Computer Science, 1997.

Muggleton, S.; De Raedt, L.: Inductive Logic Programming — theory and methods, *J.Logic Prog.*, 19-20, 629-679, (1994).

Nishida, F.; Takamatsu, S.; Fujita, Y.; and Tani, T.: Semi-automatic program construction from specifications using library modules, *IEEE Trans. on Software Eng.*, 17, (9), pp. 853-871, (1991).

Nwana, H.: Software Agents: an overview, *Knowledge Engineering Review*, 11(3), pp.1-40, Sept. (1996).

Parnas, D.L.: Information distribution of design methodology, Tech. Rept., Depart. Computer Science, Carnegie Mellon U., Pittsburgh, Pa., 1971. Also presented at the IFIP Congress 1971, Ljubljana, Yugoslavia.

Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems into Modules, *Communication of the ACM*, Vol. 15, no. 12, December 1972, pp. 1053-1058. http://www.acm.org/classics/may96.html

Partridge, D.: The Case for Inductive Programming: *IEEE* Computer, January, pp. 36-41, (1997).

Popper, K.R.: *Conjectures and Refutations: The Growth of Scientific Knowledge*, Basic Books, 1962.

Pressman, R.S.: *Software Engineering: A practitioner's Approach*, McGraw-Hill, 1992.

Rissanen, J.: Modelling by the shortest data description, *Automatica-J.IFAC*, 14, 465-471, (1978).

Rissanen, J.: Fisher Information and Stochastic Complexity, *IEEE Trans. Inf. Theory*, 1(42): 40-47, (1996).

Rumbaugh, J.: Getting Started: Using Use Cases to Capture Requirements, *J. Object-Oriented Prog..*, Sept. p.8, (1994).

Sommerville, I.: *Software Engineering. Fourth Edition*, Addison-Wesley, 1992.

Sutton, R.S.: Special issue on reinforcement learning, *Machine Learning*, 1991.

Szyperski, C.: *Component Software – Beyond Object-Oriented Programming*, Addison Wesley Longman Limited, 1998.

Wegner, P.: Interactive Software Technology, *Handbook of Computer Science and Engineering.*, CRC Press, Dec. 1996. URL: http://www.cs.brown.edu/~pw/

Weidenhaupt, K.; Pohl, K.; Matthias, J.; Haumer, P.: Scenarios in System Development: Current Practice, *IEEE Software*, March-April, 34-45, (1998).

Weyuker, E.: Evaluating software complexity measures, *IEEE Trans Software Eng.*, vol. 14, pp. 1357-1365, Sept. (1988).

Wolff, J.G.: Towards a new concept of software, *Software Engineering Journal*, IEE, January (1994).

Wooldridge, M.; Jennings, N.: Intelligent Agents:Theory and Practice, *Knowledge Eng. Review* 10 (2), 115-152, (1995).

Zuse, H.: *Software Complexity: Measures and Metrics*, Berlin, Germany: Walter de Gruyter, 1991.