Software as Learning: Quality Factors and Life-Cycle Revised*

José Hernández-Orallo and Mª José Ramírez-Quintana

Universitat Politècnica de València. Dep. de Sistemes Informàtics i Computació
Camí de Vera s/n, E-46071, València, Spain
E-mail: {jorallo, mramirez}@dsic.upv.es

Abstract. In this paper Software Development (SD) is understood explicitly as a learning process, which relies much more on induction than deduction, with the main goal of being predictive to requirements evolution. Concretely, classical processes from philosophy of science and machine learning such as hypothesis generation, refinement, confirmation and revision have their counterpart in requirement engineering, program construction, validation and modification in SD, respectively. Consequently, we have investigated the appropriateness for software modelling of the most important paradigms of modelling selection in machine learning. Under the notion of incremental learning, we introduce a new factor, predictiveness, as the ability to foresee future changes in the specification, thereby reducing the number of revisions. As a result, other quality factors are revised. Finally, a predictive software life cycle is outlined as an incremental learning session, which may or may not be automated.

1 Introduction

Software engineering was considered a pure science just two or three decades ago. Theoretical and formal methods were prevalent. Nowadays, we have a much more realistic conception of software engineering as an experimental science [6]. Empirical studies of real problems are encouraged and their conclusions are usually much more successful in practice than theoretical results. Moreover, many times theoretical studies are not applicable because in the end they do not model the software construction process.

In our opinion, the formal methods in software engineering cannot be fully exploited due to still frequent conceptions of software as being "from specification to final product". This does not take maintenance nor the generation of that specification into account.

Fortunately, there is an increasing interest in requirements elicitation and evolution as the most important topics in software engineering. *Requirements Engineering* has made some important things patently clear: the need to take the *context* of a computer system into consideration, i.e., "*the real-world environment in which the system*

---

* This work has been partially supported by CICYT under grant TIC 98-0445-C03-C1.

operates, including the social structure and the people therein" and the fact *"that requirements are always incomplete; each stage involves identifying new requirements based on experiences from previous stages… and requirements and design affect each other*" [8].

The other two fundamental (and more classical) areas of research for improving the economics of software have been reusability and modifiability, the latter being more relevant when a particular system is already implemented. The maintenance cost is greatly reduced by improving the modifiability and/or extensibility software quality factors. Another neglected but fundamental question is whether we are able to reduce the modification probability. The idea is to 'predict' requirement evolution as much as possible, in order to minimise the remaking of software as a trace of this evolution. In order to gain greater benefits, this "predictive model of requirements" should be made upon previous models by *reusing* parts of other specifications and taking context into account.

It should be explicitly stated that this predictive character of the model must be preserved during the remainder of the life-cycle: the design must be conceived to maintain the generality of the model, validation must be made according to this general model, and, more importantly, future modifications must consist of coherent revisions, not extensional 'patches' to the model.

With the appearance of new approaches, such as adaptive software [28] or intelligent software [30], which include techniques and languages for further generalisation, an empirical and theoretical study of when a generalisation of the model is useful and how it should be done seems necessary. The flippancy here would be to start from scratch or to reinvent the wheel. As we will see in the following sections, predictive modelling in particular and philosophy of science in general historically been able to provide us very useful terminology and tools to select the most likely or the most informative model.

This paper tries to emphasise the benefits of adapting the paradigm of theory construction to software, and to recognise and situate the role of induction in software engineering. In fact, recent popular subjects in the field such as adaptive software or intelligent software are in essence inductive. However, none of them use inductive tools and techniques in an explicit way.

Induction, as we use throughout this paper, is the process of theory abstraction from facts. Karl Popper proposed [36] the concept of *verisimilitude* as the level of agreement with the facts. Since there are an infinite number of theories which cover a finite number of examples, the question of verisimilitude must be contrasted with all the possible future examples that may appear in a given context. The quandary of whether there is any way to know, a priori, if a given hypothesis will be followed by future experiences in that context is obvious. If we know the initial distribution of hypotheses in that context, the plausibility of the hypothesis can be obtained in a Bayesian way. Since this initial distribution is generally unknown, many different measures of the quality of theories have been proposed in philosophy of science and Machine Learning (ML), generally in an informal way. From these, there are two main trends ([39]): *descriptional induction* ([5]), which is usually related to the simplicity criterion (or Occam's Razor) and the view of learning as compression; and *explanatory induction*

([39]), which is more closely related to coherence, cohesion or 'consilience' criteria ([41]).

In 1978, Rissanen formalised Occam's Razor under the Minimum Description Length (MDL) principle, quickly spreading over the theory and practice of ML and predictive modelling. In his later formulation ([5]), the MDL principle advocates that the best description of a given data is the shortest one. Apart from all the methodological advantages of simplicity, the major reason for using the MDL principle is that it usually avoids over-specialisation (underfitting) and over-generalisation (overfitting). From here it is usually argued that "*the shorter the hypothesis the more predictable it is*". On the contrary, 'consilience' or coherence refer to the idea that the data must be covered by the same general rule. Thagard ([41]) postulated that "*explanatory coherence*" is more important for durability than prediction and confirmation: "*a hypothesis exhibits explanatory coherence with another if it is explained by the other, explains the other, is used with the other in explaining other propositions, or if both participate in analogous explanations*". Another related notion is that of intensionality ([19]), which is based on the avoidance of extensional patches to the theory.

The convenience of both these trends will be studied for the case of software, in order to obtain predictive and coherent models for the requirements which will improve software quality factors. In the ML literature [31], there is a classical paradigm that is necessary for problems of medium or large complexity: incremental learning. The evidence is obtained incrementally and new evidence can appear which may force the revision of the model. Revision is then the most important process in incremental learning and is motivated by two kinds of errors: anomalies (cases which are not explained by the current theory) and novelties (new cases which are not covered). The incremental paradigm is the most appropriate one for software.

The paper is organised as follows. In Section 2 we introduce an analogy between software development and theory induction, which we contrast with previous (and very debated) analogies between software development and deduction and mathematics.

Section 3 reviews many software quality factors under the inductive paradigm. A new quality factor, 'predictiveness', is defined and is related to other software quality factors such as functionality, validation, reusability, modifiability, …

Section 4 introduces a new life-cycle as an incremental learning session which attempts to reduce prediction errors. The automation is discussed, as applied to declarative programming (logic programming), because the techniques and stages required for the new life-cycle (ILP, evaluation, transformation, revision, etc.) are much more mature than in any other paradigm.

Finally, Section 5 concludes the paper with a discussion of the practical relevance of this work and future directions.

2 Programs as Scientific Theories.

The statement "programs are scientific theories" or, alternatively, "software is a learning process" summarises the main idea developed in this section. Before presenting this analogy, we will review the most well-known analogies in order to understand software development. In our opinion, these analogies have failed to capture the essence of software, although partial mappings have been useful.

2.1 Existing Analogies

The use of analogies with pure mathematics to formalise the processes of computer programming was greatly influenced and promoted by the seminal paper from Hoare [24]. This first analogy, known as "Verifiers' Analogy", established that proofs are to theorems as verifications are to programs:

Verifiers' Analogy

Mathematics		*Programming*
theorem	↔	program
proof	↔	verification

After the first successes of some simple programs of algorithmic nature, the debate began precisely from the side of mathematics, exemplarised by the influential paper by De Millo, Lipton and Perlis [13]. The preceding analogy was criticised and replaced by the following one, which is based on the idea that programs are formal whereas the requirements for a program are informal:

De Millo-Lipton-Perlis Analogy

Mathematics		*Programming*
theorem	↔	specification
proof	↔	program
imaginary formal demonstration	↔	verification

The new trends in automated programming, popularised by the logic programming community, were very difficult to conciliate with an analogy with mathematics as a social science. Thus, the analogy was revised once again [38]:

Automated Programming Analogy

Mathematics		*Programming*
problem	↔	specification
theorem	↔	program
proof	↔	program derivation

However, Colburn [11] affirms that "for unless there is some sort of independent guarantee that the program specifications, no matter how formally rendered, actually specify a program which solves the problem, one must run the program to determine whether the solution design embodied by the specification is correct". In this way, he postulates that software is the final test for validating the specification.

Finally, Fetzer reopened the debate and introduced an analogy between computer programs and scientific theories [17]:

Fetzer's Analogy

	Mathematical Proofs	*Scientific Theories*	*Computer Programs*
Syntactic Entities:	Yes	Yes	Yes
Semantic Significance:	No	Yes	Yes
Causal Capability:	No	No	Yes

In our opinion, the difference between scientific theories and programs in *causal capability* disappears if we consider software to be a learning process (which shares the same paradigm with philosophy of science but *has* causal capability), where the learned theory (the program) can be used to interact with the environment.

2.2 Scientific Theories and Programs

The assumption that there is an "independent guarantee" that the specification is correct for a given problem is rather optimistic in the context of modern software development, where requirements and, consequently, applications are very complex. It is impossible to completely and definitely establish the intended behaviour that the system should show.

Current methodologies end up accepting the dynamic character of requirements, and include loops back to each of the stages, including a loop to requirement elicitation. Then it is finally recognised that requirements are never completely stated, that they evolve and that they are related to the environment, the context of specifications, the 'reality'.

The same idea is implicitly suggested by the maxim that "*testing can be used to show the presence of bugs, but never to show their absence*" [15], although applied to a restricted view of software. The extended maxim "requirements cannot be fully validated, just invalidated" conforms perfectly with Popper's conception of scientific methodology [36] where scientific hypotheses can possibly be shown to be false but can never be shown to be true.

This impels us to looking for more realistic conceptions of programming. We propose a new analogy between scientific methodology and programming methodology which shows the numerous links that can be established:

Programs as Scientific Theories Analogy

Science		*Programming*
reality	↔	requirements context
problem	↔	problem
experimentation data	↔	cases / interviews / scenarios
construed evidence	↔	requirements
evaluation	↔	analysis
best hypothesis	↔	specification
refinement	↔	transformation
theory	↔	program
verisimilitude	↔	correctness
anomalies	↔	exceptions
confirmatory experiments	↔	testing
confirmation	↔	validation
revision	↔	modification
background knowledge	↔	SW. repositories
technical books	↔	technical/programmer's doc.
science text books	↔	user documentation

The difference between best hypothesis and theory in Science is less than the difference between specification and program in software engineering. In both cases, however, a hypothesis-specification is not usually predictive/operational and it must be refined/transformed into a more manageable and applicable form, by the use of mathematisation/formalisation and a proper engagement with the background knowledge / repositories. As we have stated, the analogy should be well understood by regarding programs not only as simple scientific theories which predict the outputs for given inputs but also as systems that interact with an environment or reality according to the ontology and hypotheses that have been learned, i.e., interactive learning systems. This 'new' analogy offers many equivalences to work on and many of the results in one field can be applied to the other. Therefore, following this comparison, Figure 1 shows how deduction and induction are more or less used depending on the stage of the development of a scientific theory or a software system.

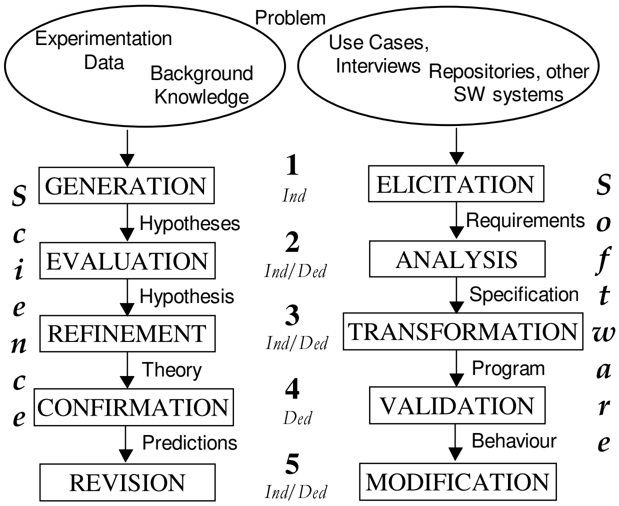

Fig. 1. Main stages in scientific theories and software systems development.

It is remarkable that, in some way, the philosophy of science and software engineering are complementary in experience and techniques because they have focused primarily on different stages.

For instance, the third and fourth stages (of a more engineering character) have been thoroughly addressed on the software side, especially the stage that is called 'transformation', which includes design and codification. In *automated programming*, this transformation stage is the most important one, because it is not convenient to *directly* execute the specification, and it is necessary to transform it by using program transformation techniques and further analyses [35]. The final program *evolves* from the specification to better performance characteristics, while still preserving semantics.

On the contrary, the first and second stages have been traditionally addressed by philosophy of science. Only recently these stages have been taken into consideration and they are included in the software construction paradigm under the banner of "requirement engineering". However, it is not usually recognised in the literature that the techniques should be mainly inductive. The information transmitted from the user to the developer is incomplete and inexact. The developer must complete and explain it by inductive methods. This inference process has recently given name to a different approach to software engineering: inductive programming [33]. According to Partridge: "*The science of creating software is based on deductive methods. But induction, deduction's ignored sibling, could have a profound effect on the future development of computer science theory and practice*". In our opinion, this effect will come true if both inferences can be effectively combined.

Finally, our concern is not to present software engineering as an experimental science but to show that each program can be seen as a scientific theory and that each software problem can be seen as a learning problem. We will work under this analogy for the rest of the paper, mostly investigating the implications from theory formation and revision to program generation and modification.

From theory formation we will translate the debate between descriptive and explanatory approaches to theory construction in science (see [39] for some contrasted positions in this debate). From theory revision (and abduction) we will try to identify which kind of software modifications are preferable: minimal extensional modifications or deeper coherent ones.

3 A Revision of Software Quality Factors

As we stated above, the role of the specification as a tentative hypothesis forces a revision of many software quality factors. The factors that are more directly related to the compliance with the specification are functionality, completeness, correctness, reliability and robustness. Other factors which are indirectly related such as testability, traceability, adaptability, flexibility, reusability, generality, maintainability /modifiability, practical modularity/granularity, ideal modularity or module independence, coupling, cohesion, efficiency/performance, comprehensibility and intelligibility will be discussed later[1].

The main factors are classically defined in terms of "the specification" or "requirements" ([26], [27]):

- *Functionality*: the degree to which a system "satisfies stated or implied needs".
- *Completeness*: usually assumed in functionality, it is the degree to which a system or component implements all required capabilities.
- *Correctness*: is the fundamental part of functionality (jointly with completeness): the degree to which software, documentation, or other items *meet specified requirements* (classical view) or *meet user needs and expectations*, whether specified or not (modern view).
- *Reliability*: "the ability of a component to perform its required functions under stated conditions for a specified period of time".
- *Robustness*: "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions".

[1] For the factors that are not explicitly defined here, we refer to [26] and [27].

Most of them deal with “stated or implied needs”, “required capabilities”, “specified requirements”, “expectations, whether specified or not” and “required functions”. However, the question arises of what software feature measures the correctness of this specification or specified requirement wrt. the “stated or implied needs”.

In our opinion, a new factor is required to measure the goodness of the requirement elicitation stage and the whole process of specification (hypothesis) revision during the overall life-cycle:

Definition 3.1. Software Predictiveness

Predictiveness is the degree to which the software system predicts present and future requirements in the context where the requirements are originated.

The key issue is that the behaviour of a program is seen as a prediction given from the hypothetical specification. The analogy from incremental learning is now as follows: software construction is an *incremental* process. The new goal is not necessarily to achieve the highest accuracy at the end of a first prototype or version (or even with the ‘last’ version), but to maximise the cumulative benefits (prediction hits) obtained *throughout* the entire life of the software.

Consequently, some concepts must be redefined. If we regard functionality equivalent to predictive accuracy, we must reconsider the components of functionality.

Functionality or *predictiveness* includes:

- *correctness* (prediction for normal situations),
- *robustness* (prediction for environment or abnormal situations),
- *reliability* (minimisation of anomalies), and
- *completeness* (minimisation of novelties).

Since a *modification* is required when there is a lack of *functionality*, modifiability (which includes extensibility) should cover both prediction errors (anomalies) and failure to predict (novelties). The former are motivated by a lack of correctness or reliability and the latter by a failure of robustness or completeness.

Finally, maintainability is redefined as considering both the predictiveness and modifiability factors. That is to say, it weights the frequency and scope of modifications. For instance, a software system can be non predictive at all and highly modifiable, resulting in a maintainable software. Conversely, a software system can be not modifiable at all but, if it has been predictive for changing requirements, then the resulting cost of maintenance could still be low.

Next we deal with how the learning/science analogy for software helps to redefined many of these and other quality factors in a more detailed way.

3.1 Functionality and Validation

In the reorganisation made after the introduction of predictiveness, one may still question why we have included reliability inside functionality. The reason that has been argued is that, since the requirements are never definite, reliability depends more on the accuracy of the requirements elicitation than on the rest of the design and implementation phases. However, the relationship between these later phases and reliability is, at first glance, not related to the predictiveness factor. We will see that this is not the case, and even the later phases of the software life-cycle can be better understood using the analogy with a learning session.

For example, it is widely accepted that redundancy compromises reliability because inconsistencies can easily arise, and checking them is *sparse* and consequently more difficult. An initial reflection would suggest that the removal of redundancies is the key to reliability. In [42], Gerard Wolff establishes the correlation between software and information compression in a direct way, arguing that "*the process of designing well structured software may be seen as a process of information compression*". Although the parallel between automated programming and pattern recognition was recognised by Banerji in the eighties [4], Wolff reminds us that patterns such as iteration, function, procedure or sub-routine, and the idea of recursion are all forms of information compression. The same applies to object-oriented abstraction through the use of inheritance.

The paradox arises when conditional statements (*if.. then.. else* and cases) are well justified by this approach. However, it has been recognised that the number of "cases" or "exceptions" in software increases the possibility of errors (and makes modifiability difficult). Moreover, there is a maintainability measure, known as *cyclomatic complexity* [25], which measures exactly this, the number of conditions in the source.

In this way, an intensional model (with few exceptions) seems much easier to check. In particular, apart from reusability, object-oriented methodologies (and especially the polymorphism technique) improve reliability because they generally eliminate long cases and exceptions with a considerable increase in code length.

As the software becomes more complex and requirements evolve during software validation, validation is finally applied to a whole which can be influenced by requirements elicitation errors, design errors or implementation errors. However, the idea of intensional and coherent models and structures applies to all the stages and must be preserved from specification to final code.

3.2 Reusability

The coherence and simplicity criteria are have long been known to improve reusability (keep in mind the claims *keep methods coherent* and *keep methods small* [37]). However, intensionality is more important for reusability than simplicity. For instance, we can make a very simple program work for a given specification or data, but if we have no foresight, the software will be not useful to slight changes in the specification. In this case, the problem resides in selecting the 'easy' or 'extensional' solutions, the most specific ones instead of the most general ones.

This avoidance of application specific procedures, methods, modules, etc, is directed by the following classical guidelines [37]:

- *provide uniform coverage*: if input conditions can occur in various combinations, write methods for all combinations, not just the ones that you currently need.
- *broaden the method as much as possible*: try to generalise argument types, preconditions and constraints, assumptions about how the method works, and the context in which the method operates. Take meaningful actions on empty values,

extreme values, and out-of-bound values (anomalies). Often a method can be made more general with a slight *increase* in code.

In addition, reusability is based on again taking advantage of the effort made in previous software components. Although there are some approaches where the reused parts can be modified (adapted) from project to project [12], the ideal option is to reuse it exactly as it was, benefiting from all the acquired validation, something that is guaranteed by encapsulation, restricting that the module could be modified at the moment of reusability.

3.3 Modifiability

Whatever the software system is, predictiveness cannot be complete and, unavoidably, prediction errors will sometimes occur. In that case a question we must ask ourselves is whether making predictive software, i.e., reducing modification frequency, entails an increase in the cost of modification, thus eventually compromising overall maintenance. To answer this, we would need to know what kind of software is most maintainable in the following way:

- it *predicts* specification changes in order to reduce the number of future changes to software.
- once a failure in prediction occurs, the *modification* can be made smoothly.

The question is how are these two properties compatible and to what extent.

It is well known that redundancy also compromises modifiability, but, at the same time, every software developer knows that excessive compression (cryptic models, code or documentation) also hinders modifiability. Wolff's new concept of software [42] is based on compression, based on the fact that short software is more manageable and that the reduction of redundancy eases modifiability and extensibility. In the end, despite its predictive shortcomings, the MDL principle (as a preference for compressed models) is not valid for software development because it has been experimentally proved that extremely compressed models are not appropriate for reusability, as for other software quality factors, such as comprehensibility, testability and modifiability itself.

Is there then any good compromise between compression and avoidance of redundancy? The answer is again to realise that avoidance of redundancy can be achieved without excessive compression. In other words, there are an infinite number of irreduceable models (without redundancy), such as intensional models, which are not the most compressed models.

The explanatory paradigm of ML and philosophy of science is the most appropriate one to ensure functionality, reusability, modifiability and maintenance. Some other factors such as traceability, modularity, cohesion, comprehensibility and intelligibility can also be partially redefined under the paradigm of machine learning or philosophy of science (especially the explanatory view). For instance, performance can be seen in terms of lazy vs. eager learning methods [29]. Eager learning works with a model, whereas lazy learning predicts each new instance by comparing it with old cases. In the case of software, eager learning obviously requires more effort at development time and revision is more complicated, but performance and reusability are not compromised.

Nevertheless, from the ML experimental and theoretical results on the complexity of lazy methods, a medium or large size predictive system must necessarily be based on eager learning methods, and, in the following, we will take for granted that this is the case.

4 Predictive Software Life-Cycle

In section 2, the five main common stages between science and software were presented just as they are, without explicit influence between them. The analogy is now exploited to re-design the software life-cycle with the goal of making it predictive, and introducing model revision as one of the most important (and reiterative) stages.

4.1 A New Life Cycle

Figure 2 represents a mixture between an automated software construction cycle and scientific theory evolution. The terminology is used indistinctly, by either borrowing a term from philosophy of science (or ML) or by using a term from software engineering.

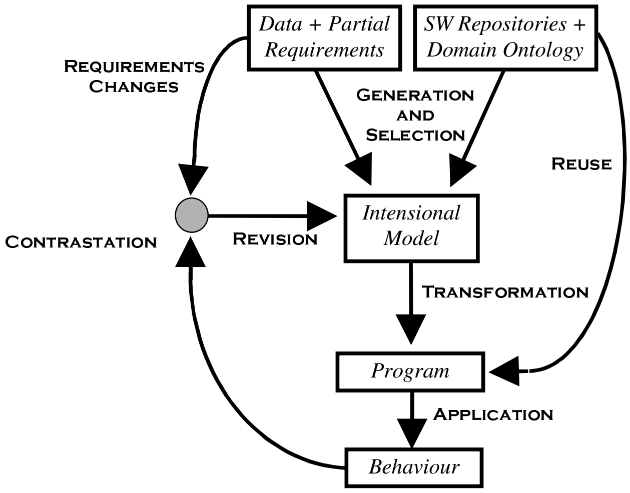

Fig. 2. *Predictive Software Life-Cycle*

The process begins by gathering the data and partial requirements of the problem and *inducing* a first model from these examples usually with the help of background knowledge which consists of reusable software components of previous projects and some other information about the domain of the reality from which the problem originates. The next stages are quite similar to the classical or automated life-cycles, depending on the degree of formalisation of the model and the automation of the transformation stage. The first result of the process is a program which can be contrasted / validated with more use examples or with real operation. This comparison may trigger a partial or whole revision of the model, which is followed by a rederivation of the program.

Obviously, this cycle could be more detailed depending on the automated or non-automated character of each stage. For instance, in a non-automated developing schema, an analysis stage could be introduced between an induced partial specification and the model, without using previous software. The design would convert this initial model into a refined model by using the repositories.

4.2 Towards the Automation of the Predictive Life-Cycle

As the analogy suggests, at present, the goal would be to (partially) automate the process by using techniques from ML. However, automated inductive methods are not yet ready for most of the complex software problems we face. Nonetheless, specifications are getting increasingly more complex and more data-based, and ML techniques are becoming more powerful to justify the inductive software paradigm practically and economically. Partridge [33] presents some successful cases of automated construction of software from sample data.

There are two generic approaches for research towards this difficult goal: 1) to evolve simple, fully-automated software systems into more complex systems; and 2) to develop semi-automated systems. Both approaches highlight a revival of declarative paradigms, because declarative languages are more mature for automating the stages of the previous life-cycle, and more intelligible.

For the first *via*, logic programming is clearly the most appropriate paradigm at present. Inductive Logic Programming (ILP) [32] represents the automation of the stage of generation and selection from examples and background knowledge. The automation of the transformation stage is ensured by many years of research in transformation techniques (see e.g. [35]). The automation of the revision of the model can be found in works usually originated in non-monotonic approaches inside AI, by using logical theories [9] or more software specific approaches [1]. Finally, the application stage is performed directly through SLD-resolution or after a specialisation stage (see e.g. [2]) for improving performance.

The problems of scalability appear at all the stages, but more critically in the first stage. From the first recovery of specifications by using ILP [10], and after the application for inducing simple algorithms shown in part III of [7], the prospect of ILP for program synthesis has sometimes been discredited and biased. Nowadays, ILP is addressing more complex problems, so the predictive declarative programming paradigm can address medium-complexity problems, such as web-agents, controllers in changing environments, software assistants, and the like. Nonetheless, we must recognise the main problems of ILP for software engineering: background knowledge usage bottleneck and knowledge mobilisation [18].

For the second *via*, the most important issue is the intelligibility of software, i.e., if some part of the model (or the code) is automatically generated from the user's requirements, it must be intelligible to humans, in order to ensure good coordination with manually generated software and allowing for manual maintenance and revisions. In this case, the use of declarative languages is even more justified.

5. Conclusions and Future Work

This paper has focused on the view of programs as scientific theories or, more precisely, software development as learning. This analogy forces a reconsideration of the quality factors of software and the software construction life-cycle. New software characteristics are distinguished, mainly that software must be predictive, in order to minimise future modifications, and other software factors are redefined. Induction will be more important than deduction in the future, when automation is possible for complex systems.

Software systems must receive feedback from the user about the quality of its task: adequacy to user's needs, efficiency, robustness, etc., and they must update to the user's demands dynamically. In other words, software systems must learn from the environment and user's needs, in an interactive way quite similar to query or interactive learning [3].

The recent aim for automation of induction has driven us to highlight and encourage the productive translations of ML techniques to software engineering. Although full automation is not possible at the moment, the analogy, the revision of factors and the new life-cycle are useful for traditionally developed software. This is the keypoint of the paper, to highlight that a change in attitude (or paradigm) in software construction can be useful in practice even without any automation at all.

In a broader context, many historical traits of the short life of software engineering (in contrast to the long life of philosophy of science) can also be better understood. For instance, many techniques and paradigms of software engineering in the last decades can be seen as tools and mechanisms to ease compression while preserving intensionality (avoidance of exceptions), such as structured programming, object-oriented programming, encapsulation, polymorphism, etc.

Moreover, the emphasis placed on the inductive phase of modelling to make software more predictive matches the increasing relevance that requirement elicitation has been acquiring in software engineering theory and practices in the last decade.

At present, the authors are developing inductive algorithms and systems for other more powerful declarative languages, for which transformation and specialisation techniques are also developed [35], [2]. In particular, the induction of functional logic programs has been attempted [20][21] in order to allow the acceptance of more expressive and complex requirement cases as examples, which is usual in software applications. The system FLIP [16] is specially designed to induce recursive programs and it also supports the use of background knowledge, with the long-term goal of automating [22] more and more parts of the whole process.

As future work, from a much more practical point of view, software is a very appropriate place to experiment new techniques from AI and ML (see e.g. [40]). Moreover, AI and ML can expand their commercial applications in this area. Many ML paradigms and techniques [31] can be used in different processes and stages of software construction: evaluation criteria like cross-validation, query learning [3], reinforcement learning applied to constructive languages [23], explanation-based learning, [14], data mining for knowledge-based software, analogical reasoning, case-based reasoning, genetic computation, etc.

In summary, our analogy also shows that until machine intelligence (and ML) approaches human ability more closely, fully automated programming will remain a fallacy. In the meantime, in accordance with the analogy presented here and in an effort to reach the Utopia of "intelligent software" [30], a more prosperous methodology for software construction could be devised from the nascent "predictive software".

Acknowledgements

First of all, the authors would like to thank Jesús Alcolea for the introduction to Fetzer's work. Other people have helped to develop these ideas: Jaume Agustí, Vicent Pelechano and Enrique Hernández. Finally, this work also benefited from a short discussion held with John Lloyd, Steve Muggleton and Luc de Raedt about the possibilities of the use of ILP for software practice at the the "*JICSLP'98 CompulogNet Area Meeting on Computational Logic and Machine Learning*" [20].

References

1. Alferes, J.J.; Leite, J.A.; Pereira, L.M.; Przymusinska, H.: Przymusinski, T.C. "Dynamic Logic Programming", in Freire et al. (eds) Proc. Joint Conf. of Declarative Prog., pp. 393-408, 1998.
2. Alpuente, M.; Falaschi, M.; Vidal, G. "Partial Evaluation of Functional Logic Programs" *ACM Trans. on Programming Languages and Systems*, 20(4):768-844, 1998.
3. Angluin, D. "Queries and concept learning" *Machine Learning* 2, No. 4, 319-342, 1988.
4. Banerji, R.B. "Some insights into automatic prog. using a pattern recognition viewpoint" in Biermann et al. (eds.): *Automatic program construction techniques*, Macmillan, 1984.
5. 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, 1998.
6. Basili, V.R. "The Experimental Paradigm in Software Engineering" in Rombach et al. (eds.) *Experimental Software Engineering Issues*, LNCS no. 706, Springer-Verlag, 1993.
7. Bergadano, F.; Gunetti, D. *Inductive Logic Programming*, The MIT Press, 1996
8. Berry, D.M.; Lawrence, B. "Requirements Engineering" *IEEE Software*, 26-29, Mar. 1998.
9. Botilier, C.; Becher, V. "Abduction as belief revision" *Art. Intelligence* 77, 43-94, 1995.
10. Cohen, W. "Recovering Software Specifications with Inductive Logic Programming" in K. Ford, (ed.), *Proc. of the AAAI Conference*, pages 142-148, Seattle, WA, 1994.
11. Colburn, T.R. "Program Verification, Defeasible Reasoning, and Two Views of Computer Science" in *Minds and Machines* 1, 97-116, 1991.
12. Davis, M. J. "Adaptable, Reusable Code" Symp. on Sw Reusability, Seattle, apr. 1995, ACM.
13. 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.
14. Dietterich, T.G.; Flann, N.S. "Explanation-Based Learning and Reinforcement Learning: A Unified View" *Machine Learning*, 28, 169-210, 1997.
15. Dijkstra, E.W. "Notes on Structured Programming" in O.Dahl *et al.* (eds.) *Structured Programming*, New York, Academic Press 1972.
16. Ferri, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. "The FLIP System. From Theory to Implementation", submitted to Machine Learning, Special Issue on ILP'99.
17. Fetzer, J.H. "Philosophical Aspects of Program Verification" *Minds* & *Machines* 1, 197-216, 1991.

18. Flener, P. and Yilmaz, S. "Inductive synthesis of recursive logic programs: achievements and prospects", *The Journal of Logic Programming*, 41, 141-195, 1999.
19. Hernández-Orallo, J.; García-Varea, I. "Explanatory and Creative Alternatives to the MDL Principle", *Foundations of Science*, Kluwer, to appear.
20. Hernández-Orallo, J.; Ramirez-Quintana, M.J. "Induction of Functional Logic Programs", Lloyd (ed.) JICSLP'98 CompulogNet Meeting on Computational Logic and ML, 49-55, 1998.
21. Hernández-Orallo, J.; Ramírez-Quintana, M.J. "A Strong Complete Schema for Inductive Functional Logic Programming", in Flach, P.; Dzeroski, S. (eds.) *Inductive Logic Programming'99* (ILP'99), in LNAI 1634, pp. 116-127, Springer-Verlag 1999.
22. Hernández-Orallo, J.; Ramírez-Quintana, M.J. "Predictive Software", submitted to Automated Software Engineering Journal, Special Issue on Inductive Programming.
23. Hernández-Orallo, J. "Constructive Reinforcement Learning", *International Journal of Intelligent Systems*, Wiley, to appear.
24. Hoare, C.A.R. "An Axiomatic Basis for Computer Programming" *Communications of the ACM* 12, 576-580, 583, 1969.
25. IEEE Std. 982.1-1988 "IEEE Standard Dictionary of Measures to Produce Reliable Software" *The IEEE*, June, 1988.
26. IEEE/ANSI Std. 610.12 "IEEE Standard Glossary of Software Engineering Terminology" *The IEEE*, February, 1991.
27. ISO/IEC 9126, "Information technology. Software Product Evaluation. Quality characteristics and guidelines for their use" *Intl. Org. for Standardization and Intl. Electrotechnical Commission*, 1991.
28. Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996.
29. López de Mántaras, R.; Armengol, E. "Machine Learning from examples: Inductive and Lazy Methods" *Data & Knowledge Engineering* 25, 99-123, 1998.
30. Maes, P. "Intelligent Software" *Scientific American* 273 (3), September, 1995.
31. Mitchell, Tom M., *Machine Learning*, McGraw-Hill International Editions, 1997.
32. Muggleton, S.; De Raedt L. "Inductive Logic Programming — theory and methods" *Journal of Logic Programming*, 19-20:629-679, 1994.
33. Partridge, D. "The Case for Inductive Programming" *IEEE Computer*, pp. 36-41, Jan 1997.
34. Pettorossi, A.; Proietti, M., "Rules and Strategies for Transforming Functional and Logic Programs" *ACM Computing Surveys,* Vol. 28, no. 2, June 1996.
35. Pettorossi, A.; Proietti, M., "Developing Correct and Efficient Logic Programs by Transformation" *Knowledge Engineering Review*, Vol. 11, No. 4, December 1996.
36. Popper, K.R., *Conjectures and Refutations: The Growth of Scientific Knowledge* Basic Books, New York 1962, Harper, New York, 1965, 1968.
37. Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall 1991.
38. Scherlis, W.L.; Scott, D.S. "First Steps Towards Inferential Programming" in R.E.A. Mason (ed.) *Information Processing* 83, pp- 199-212, 1983.
39. Shrager, J.; Langley, P., *Computational Models of Scientific Discovery and Theory Formation*, Morgan Kaufmman, 1990.
40. Srinivasan, K.; Fisher, D. "Machine Learning Approaches to Estimating Software Development Effort" *IEEE Transactions on Software Engineering*, Vol. 21, No. 2, Feb. 1995.
41. Thagard, P. "Explanatory coherence", *Behavioural & Brain Sciences*, 12 (3), 435-502, 1989.
42. Wolff, J.G. "Towards a new concept of software", *Software Engineering J.*, IEE, Jan. 1994.