The Role of Induction in (Semi-)Automated Life-Cycles

José Hernández-Orallo and Mª José Ramírez-Quintana
Unversitat Politècnica de València
Dep. de Sistemes Informàtics i Computació, C/ de Vera s/n, E-46071, València, Spain
E-mail: {jorallo, mramirez}@dsic.upv.es

Abstract. Inductive inference, defined as the hypothetical reasoning process from specific to general\textsuperscript{1}, matches with the view of software construction as a hypothetical (tentative) process from specific examples, requirements cases, scenarios, etc., to the general behaviour of a program. In this paper, the analogy is further exploited and the discussion is centred on the transfers that can be made to software engineering, especially the selection of a predictive model of requirements, and the placement of other inductive stages in semi-automated life-cycles. Then, software development is re-understood as an incremental learning session, where both induction and deduction play different and complementary roles. The distance between the degree of automation which would be required and the current development of inductive techniques is analysed. For bridging this gap and for a smooth engagement between deductive and inductive techniques, declarative programming languages, especially (functional) logic programming, are recognised as clearly advantageous.

1 Introduction

The role of inductive inference in software development has usually been neglected, due to several reasons. First, inductive inference is hypothetical, in the sense that the results of the inference process can be wrong, and even worse, an inductive inference can never be shown to be correct. The idea that automated software construction should avoid error (which was exclusively attributed to human intervention \textsuperscript{2}) excluded the use of inductive techniques. Secondly, the classical analogies of programs as proofs promoted the view of programming as an exclusively deductive process. Thirdly, the understanding and existing techniques for induction have been by far less developed than those for deduction, mainly because induction is a much more complex process than deduction. Thus, initial ideas and trends in this direction, like the 'Programmer’s Apprentice', an automated programming system designed by \textsuperscript{3} [21], fell soon into oblivion.

These reasons have come into question recently, though. First of all, it has been recognised that the main source of errors is not a wrong manual derivation from specification to final program but a wrong elicitation of the specification.

\textsuperscript{1} This work has been partially supported by CICYT under grant TIC 98-0445-C03-C1.
\textsuperscript{2} In contrast to what is called mathematical induction, which is truth-preserving, and that will be considered here a deductive process.
Moreover, it is well known that modifications are costlier to be made the earlier the stage. Consequently, requirement elicitation has been given the most prevalent position in software development [5], because errors in this early stage are brought in its train through all the process.

The analogy between programs and proofs, or other mathematical entities, have also fallen into disuse, mainly because these analogies have not been able to capture the real nature of software as an incremental, evolutionary process, which can never be shown to be finished, precisely because specifications are seldom absolutely correct and they are almost never definitive.

Finally, the understanding of induction and the state of the art of machine learning are much more positive for their application to software. Although full automated techniques are not ready for medium and large scale applications, much terminology and the results on model selection criteria can easily and favourably applied to software.

In particular, after elaborating the analogy between software development and an incremental learning session, we study the choice of the most convenient model in order to improve the economics of software. As is well known, there are two factors that mainly affect this economics: maintainability and reusability. Maintenance cost is especially reduced by improving the modifiability and/or extensibility software quality factors, but also reducing the modification probability. The idea is to ‘predict’ and prepare for requirement changes rather than patch them when they occur. And, finally, according to the second factor, this “predictive model of requirements” should be made upon previous confirmed models by reusing parts of other specifications and taking context into account.

The paper is organised as follows. In Section 2 we introduce an analogy between software development and theory induction. In section 3 a new quality factor, ‘predictiveness’, is defined, and it is related to other software quality factors. Section 4 introduces a new life-cycle as an incremental learning session which tries to reduce prediction errors. Its automation is discussed, particularised to declarative programming, more specifically to (functional) logic programming, because the techniques and stages required for the new life-cycle (JLP, evaluation, transformation, revision, etc.) are much more mature than in any other paradigm. Then we show how machine learning (ML) selection criteria can be used to compare models in order to distinguish the most predictive one as well as to automatically induce some simple models. Finally, section 5 concludes the paper with a discussion of the practical relevance of this work and points out future directions.

2 Software Development as an Incremental Learning Session

In [11] we reviewed previous analogies that were essayed to better understand the software process, and come up with a final analogy between programs and scientific theories. We showed the benefits of adapting the paradigm of theory construction to software which suggested many results and techniques in one field
to be used into the other. The analogy between software development and theory construction can be summarised as follows:

<table>
<thead>
<tr>
<th>Science</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>reality</td>
<td>requirements context</td>
</tr>
<tr>
<td>problem</td>
<td>problem</td>
</tr>
<tr>
<td>experimentation data</td>
<td>cases / interviews / scenarios</td>
</tr>
<tr>
<td>unstructured evidence</td>
<td>requirements</td>
</tr>
<tr>
<td>test hypothesis</td>
<td>specification</td>
</tr>
<tr>
<td>evaluation</td>
<td>analysis</td>
</tr>
<tr>
<td>refinement</td>
<td>transformation</td>
</tr>
<tr>
<td>theory</td>
<td>program</td>
</tr>
<tr>
<td>verisimilitude</td>
<td>correctness</td>
</tr>
<tr>
<td>anomalies</td>
<td>exceptions</td>
</tr>
<tr>
<td>confirmatory experiments</td>
<td>testing</td>
</tr>
<tr>
<td>information</td>
<td>validation</td>
</tr>
<tr>
<td>revision</td>
<td>modification</td>
</tr>
<tr>
<td>background knowledge</td>
<td>SW, repositories</td>
</tr>
<tr>
<td>technical books</td>
<td>technical/programmer’s documentation</td>
</tr>
<tr>
<td>science text books</td>
<td>user documentation</td>
</tr>
</tbody>
</table>

This 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.

The most important utility of this analogy is that philosophy of science / ML and software engineering have focused primarily on different stages, and, hence, they are complementary in experience and techniques.

In figure 2.1, the use of inductive and deductive techniques are placed in the different stages of the corresponding scientific and software processes.

![Figure 2.1. Main stages in scientific theories and software systems development](image-url)
The first stages have been mainly addressed by the inductive sciences (Philosophy of Science, ML, statistics) and neglected by software engineering. Recently, these stages have fortunately been taken into consideration and they are included in the software construction paradigm, under the banner of “requirement engineering”.

In the ML literature [17], there is a classical paradigm necessary for problems of medium or large complexity: incremental learning. The evidence is fed up 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 it 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 for software, an the analogy is set between software development and an incremental and interactive learning session.

3 Software Predictiveness

The conception of software development as a hypothetical process, which can be refuted but never confirmed, could motivate the thought that few can be done in obtaining a good and durable specification, and the techniques and effort should be then devoted to further stages, the more deductive ones, from the specification to the final product. Never more on the contrary, most can be done in order to improve the quality and durability of the specification.

The first thing is to realise that software quality factors should not be understood wrt. the specification, because a great deal of the software quality depends on the previous process of obtaining that specification. For instance, classical factors such as functionality, completeness, correctness, reliability and robustness are defined in terms of “the specification” or “requirements” ([14], [15]). But the reference must be previous. Hence, there is need to introduce a measure of the degree of validity of this specification or specified requirement wrt. the “stated or implied needs”. Such a factor can be defined from the well-known concept of model predictiveness.

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. Functionality is defined in terms of prediction rather than execution. Moreover, the analogy with incremental learning settles software construction as an incremental process. The 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 all the software life.
Some concepts, consequently, must be re-understood. If we see functionality as an equivalent to predictive accuracy, we must re-consider 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 cannot be predictive at all and highly modifiable, resulting in a maintainable software. Conversely, a software system cannot be modifiable at all but it can be predictive for changing requirements, and the resulting maintainability cost could still be low.

4. **Predictive Software Life-Cycle**

In section 2, we compared five main common stages between science and software. In [12] this analogy is 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**

A mixture between an automated software construction cycle and scientific theory evolution is shown in Figure 4.1. The terminology is used indistinctly, by either borrowing a term from philosophy of science (or ML) or by using a term from software engineering.

Certainly, 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 using the repositories.

The predictive life cycle is similar to the transformational one [4] with regard to the last stages of the development: definition of a model or specification of requirements and its transformation into the final program. Also, the maintenance stage consists of a revision and re-derivation of the model. Nevertheless, the difference between both life-cycles lies in the way in that intensional models or
formal specifications (respectively) are built, and the criteria used for generating and selecting them.

The adoption of this life-cycle also depends on the choice of the representational language. In the same way as many modern non-declarative methodologies were not adapted to the automated programming paradigm, because automated deductive techniques required a well-established and manageable semantics, the first inductive stages are even more difficult to apply to non-declarative languages, because high-level and model-based languages are required for the automated induction of expressive and understandable models.

4.2 Towards the Automation of the Predictive Life-Cycle

Suggested by the analogy, the goal seems to be to (partially) automate the process, by using techniques from ML. However, automated inductive methods are not ready yet for most complex software problems of today. Nonetheless, specifications are increasingly getting more complex and more data-based, and ML techniques more powerful to justify practically and economically the inductive software paradigm for some kinds of applications. Partridge [19] presents some successful cases of automated construction of software from sample data.

Two generic approaches for research can be established towards the goal of automation for more varied and complex systems: 1) to evolve simple, fully-automated software systems into more complex systems; and 2) to develop semi-automated software development 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, if human intervention is necessary to fill gaps between stages.
With regard to the first approach, logic programming is clearly the most appropriate paradigm at present. Inductive Logic Programming (ILP) [18] represents the automation of the stage of generation and selection of hypotheses from examples and background knowledge. The automation of the transformation stage is ensured by many years of research in transformation techniques (see e.g. [20]). The automation of the revision of the model can be found in works usually originated in non-monotonic approaches inside AI using logical theories [6] 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. Consequently, we can chain all these techniques in a unique paradigm. Using logic programming, our cycle is practically the same as the generic one, but now using concrete (and in many cases automated) techniques, as it is illustrated in figure 4.2.

The extension to other declarative languages seems difficult at the moment, except for the functional logic programming case, because the generation and evaluation of inductive hypotheses and transformation processes are now available ([10],[20]). In the end, the functional extension may be crucial for the acceptance of the predictive declarative programming paradigm, because the definition of functions is a profoundly established custom in software engineering. Furthermore, the examples which are more appropriate for the use of ML techniques are classification problems, which are better handled as functions than as predicates.

![Diagram of Predictive Logic Programming Cycle](image)
In the following subsections, we show how ML selection criteria can be used in software engineering to compare models in order to distinguish the most predictive one. Moreover, in some special cases, ML techniques can be used as well as to induce the models under the chosen criterion.

4.3. Model Comparison Example

A university library gathers the information about the journals it is subscribed to and the articles that appear in them. The system allows the search by journal name, paper title and year, as well as author(s) name. However, no information about the area or field of each paper is included in the database. A new query system is to be developed to allow for the search of journals and papers by field.

Suppose that part of the journal papers have been classified manually. Now the goal of the application is to suggest the field for the rest of the papers. Part of the initial database (scientific papers in computer science journals) is read and converted into the schema illustrated in Figure 4.3.

![Diagram](image)

Figure 4.3. A Database Schema for The Field Query Problem

To better illustrate the point and trace the subsequent models, let us show a brief excerpt from the data:
<table>
<thead>
<tr>
<th>J.jcod</th>
<th>J.title</th>
<th>P.pcod</th>
<th>authors²</th>
<th>F.jcod</th>
<th>F.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASE</td>
<td>Automated Software Eng. 1</td>
<td>Penix; Alexander</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>ASE</td>
<td>Automated Software Eng. 2</td>
<td>Chatzoglou; M.</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>ASE</td>
<td>Automated Software Eng. 3</td>
<td>Couso; Couso</td>
<td>LP</td>
<td>Logic Programming</td>
<td></td>
</tr>
<tr>
<td>JLP</td>
<td>Journal of Logic Prog.   4</td>
<td>Muggleton; DeRaedt</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>JLP</td>
<td>Journal of Logic Prog.   5</td>
<td>Lloyd</td>
<td>LP</td>
<td>Logic Programming</td>
<td></td>
</tr>
<tr>
<td>JLP</td>
<td>Journal of Logic Prog.   6</td>
<td>Sammelia; Waleen</td>
<td>LP</td>
<td>Logic Programming</td>
<td></td>
</tr>
<tr>
<td>JLP</td>
<td>Journal of Logic Prog.   7</td>
<td>Couso; Couso</td>
<td>LP</td>
<td>Logic Programming</td>
<td></td>
</tr>
<tr>
<td>AAI</td>
<td>Applied Artificial Intell. 8</td>
<td>Blockeel; DeRaedt</td>
<td>DM</td>
<td>Data Mining</td>
<td></td>
</tr>
<tr>
<td>TCS</td>
<td>Theo. Computer Science 9</td>
<td>Roscoe; Hoare</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>MM</td>
<td>Minds and Machines 10</td>
<td>Felzer</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>DK</td>
<td>Data &amp; Knowledge Eng. 11</td>
<td>Lopez; Armengol</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>CAO</td>
<td>Comm. of the ACM 12</td>
<td>Valiant</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>CAO</td>
<td>Comm. of the ACM 13</td>
<td>Hoare</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>CAO</td>
<td>Comm. of the ACM 14</td>
<td>Fayyad; Uluruymy</td>
<td>DM</td>
<td>Data Mining</td>
<td></td>
</tr>
<tr>
<td>CAO</td>
<td>Comm. of the ACM 15</td>
<td>Felzer</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>CAO</td>
<td>Comm. of the ACM 16</td>
<td>Genesereth; Ketchpel</td>
<td>SW</td>
<td>Software Engineering</td>
<td></td>
</tr>
<tr>
<td>CAO</td>
<td>Comm. of the ACM 17</td>
<td>Muggleton</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>DEB</td>
<td>Data Engineering Bulletin 18</td>
<td>Fayyad</td>
<td>DM</td>
<td>Data Mining</td>
<td></td>
</tr>
<tr>
<td>NGC</td>
<td>New Generation Computing 19</td>
<td>Muggleton</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>MLJ</td>
<td>Machine Learning Journal 20</td>
<td>Lopez de Mantaras</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>MLJ</td>
<td>Machine Learning Journal 21</td>
<td>Muggleton</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>MLJ</td>
<td>Machine Learning Journal 22</td>
<td>Angluin</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence 23</td>
<td>DeRaedt; Dehaspe</td>
<td>ML</td>
<td>Machine Learning</td>
<td></td>
</tr>
</tbody>
</table>

This data is generated by the following query:

```sql
SELECT J.jcod, J.title, P.pcod, A.name, F.jcod, F.name
FROM JOURNAL J, PAPER P, WRITTEN W, AUTHOR A, FIELD F
WHERE J.jcod = P.jcod AND P.pcod = W.pcod AND W.acod = A.acod AND
P.jcod = F.jcod;
```

The first relation to be learned is `is_topic`, which is easily extracted as follows:

```prolog
% topics of journals
is_topic(J, F) :- paper(_, _), J, _, F).
```

which means that the topics of a journal are all the topics of its papers. This is possible because the relation between journal and topic is many-to-many, by the use of the extra relation `is_topic`. On the other hand, the problem of classifying the main field of a paper is more complicated because of the uniqueness restriction. Moreover, apart from the sample, there will be papers which will be manually classified and others will have to be classified automatically. To differentiate them, the classified papers are denoted by the predicate `paper_`. From here, the following models can be induced:

² A.name is shown aggregated.
MODEL A

$\text{Learned rules}$

\begin{itemize}
  \item Muggleton's articles are about ML.
  \item paper\(_{(P, A)}\) : written\(\text{(P, A)}\), author\(\text{(A, 'Muggleton')}\), !.
  \item 'JLP' papers field is 'LP' but Muggleton's articles handled above.
  \item paper\(_{(P, A)}\) : 'JLP', 'LP'.
  \item ML\(\text{J}' articles are always about ML.
  \item paper\(_{(P, A)}\) : 'ML\(\text{J}'', 'ML').
  \item Valiant's articles are about ML.
  \item paper\(_{(P, A)}\) : written\(\text{(P, A)}\), author\(\text{(A, 'Valiant')}\), !.
  \item Fayyad's articles are about DM.
  \item paper\(_{(P, A)}\) : written\(\text{(P, A)}\), author\(\text{(A, 'Fayyad')}\), !.
  \item 'C ACM' papers field is 'SW' but Valiant's, Fayyad's and Muggleton's:
    paper\(_{(P, A)}\) : 'C ACM', 'SW'.
\end{itemize}

$\text{Exceptions // non-predictable}$

\begin{itemize}
  \item paper\(_{(P, A)}\) : 'SW'.
  \item paper\(_{(P, A)}\) : 'SW'.
  \item paper\(_{(P, A)}\) : 'LP'.
  \item paper\(_{(P, A)}\) : 'DM'.
  \item paper\(_{(P, A)}\) : 'SW'.
  \item paper\(_{(P, A)}\) : 'ML'.
  \item paper\(_{(P, A)}\) : 'ML'.
\end{itemize}

MODEL B

$\text{Learned rules ordered by priority (strength)}$

\begin{itemize}
  \item dependency between authors’ other papers field:
    paper\(_{(P, A)}\) : written\(\text{(P, A)}\), written\(\text{(P2, A)}\), paper\(_{(P2, A)}\), paper\(_{(P3, A)}\), written\(\text{(P3, A)}\), paper\(_{(P3, A)}\), written\(\text{(P3, A2)}\), paper\(_{(P3, A2)}\), written\(\text{(P3, A2)}\), paper\(_{(P3, A2)}\) \(\text{F} \rightarrow \text{F2, F} \rightarrow \text{P3, A} \rightarrow \text{A2}.
  \item dependency between author's other paper field:
    paper\(_{(P, A)}\) : written\(\text{(P, A)}\), written\(\text{(P2, A)}\), paper\(_{(P2, A)}\), paper\(_{(P2, A)}\).
  \item dependency with journal field:
    paper\(_{(P, A)}\) : is_topic\(\text{(J, F)}\).
  \item Unclassified papers follow these rules:
    \item 'JLP' papers field is 'LP':
      paper\(_{(P, A)}\) : 'JLP', 'LP'.
  \item 'ASE' papers field is 'SW':
    paper\(_{(P, A)}\) : 'ASE', 'SW'.
\end{itemize}

$\text{Uniqueness restrictions for the field of a paper}$

\begin{itemize}
  \item paper\(_{ul}\) : paper\(_{(P, F1)}\), paper\(_{(P, F2)}\), F1 != F2, fail.
  \item paper\(_{u2}\) : paper\(_{(P, F1)}\), paper\(_{(P, F2)}\), F1 != F2, fail.
  \item paper\(_{u3}\) : paper\(_{(P, F1)}\), paper\(_{(P, F2)}\), F1 != F2, fail.
\end{itemize}

$\text{Priorities}$

\begin{itemize}
  \item paper\(_{(P, A)}\) : ! paper\(_{(P, A)}\).
  \item paper\(_{(P, A)}\) : ! paper\(_{(P, A)}\).
  \item paper\(_{(P, A)}\) : ! paper\(_{(P, A)}\).
\end{itemize}
In a semi-automatised ILP environment, both models would possibly need some meta-information to be generated (mainly the cuts for uniqueness in the second model), but they are not far from the possibilities of some ILP systems. Clearly, model A is much shorter and simpler than B. However, model A is full of exceptions (extensional patches) and will fail to classify many new papers. In contrast, model B is much more complex, but it is also much more intensional, in the sense that it does not include many particular cases. Hence, the second model should be preferable, and it is indeed selected by several intensional evaluation measures (e.g. [13]).

Finally, the valuation stage must be made before the transformation stage, because these transformations can modify the intensionality of the model in order to make it more operative, i.e. to make it an efficient program. Moreover, specialisation must carefully be done, because the predictiveness of the model could be affected.

4.4. Automated Selection Example

Let us show with a second example how this model selection can be done automatically. We will use the FLIP system for this. The FLIP system [8] is an application built in C which implements the Inductive Functional Logic Programming framework [9][10]. At its actual stage of development, the FLIP system allows conditional functional logic programs for background knowledge and hypothesis. The system works with two sets of facts: the positive examples and the negative ones, and optionally an initial set of theories and a background knowledge. Hypothesis selection is guided by simplicity, avoidance of exceptions and greatest coverage.

The FLIP system is a versatile application that can operate as a pure induction system, a theory reviser and a theory evaluator. For this example we use FLIP as a theory evaluator. Consider the following set of examples of good clients of an insurance company:

```
goodc(p(p(v,woman),tall))=t,
goodc(p(p(v,nurse),woman))=t,
goodc(p(p(p(v,has_children),woman),joan),speaks_spanish)=t,
goodc(p(p(p(v,jane),woman),speaks_portuguese),tall))=t,
goodc(p(p(p(p(v,susan),has_children),teacher),woman),highincome)=t,
goodc(p(p(v,married),teacher),cellularphone)=t,
goodc(p(p(v,teacher),atheist),cellularphone),married))=t,
goodc(p(p(v,teacher),atheist),cellularphone),married))=t,
goodc(p(p(v,browniee),likes_coffee),has_children))=t,
goodc(p(p(p(v,has_children),nurse),cellular_phone))=t,
goodc(p(p(v,jane),plays_chess),has_children)=t,
goodc(p(p(p(v,mary),speaks_spanish),has_children),cellularphone)=t
```

where ‘p’ is the list constructor, ‘v’ denotes the empty list and ‘t’ denotes true.

From here and some negative evidence, the following two theories have been generated:

\[ \text{See its webpage, http://www.dsic.upv.es/~jorallo/FLIP.} \]
where ‘and’ is the ‘A’ logic connective and ‘member’ is an incomplete function which returns true if the element is in the list. Both functions are defined in the background theory.

FLIP automatically chooses model A as the best solution by following its selection criterion wrt. the previous examples. In fact, it is to be expected that model B would be revised soon, because the last equation only covers one example and, in the best case, will only be applicable to clients whose name is ‘Anthony’. It is apparently a patch which should be avoided in a predictive software. Accordingly, FLIP rates the first model much more favourably than the second one.

5. Conclusions

Under the analogy between programs and scientific theories or, more precisely, the view of software development as an incremental learning, we have re-understood many software quality factors and we have placed the role of inductive techniques into the software life-cycle.

Software must be predictive, and this can be done with the same tools and under the same limitations that model induction and model selection criteria have been studied in the ML literature. Model selection criteria are directly and easily applicable, as it has been shown. On the other hand, the progress in the automation of induction in the recent years suggests the applicability to software engineering. However, the automation of the whole cycle for complex problems has also been shown to be infeasible at the present moment.

These limitations and the need of the conjunction with deductive techniques justify the use of declarative languages for increasing and broadening the range of applications. In our opinion, the recent interest in “inductive programming” [19] and the expected (commercial) applications of AI and ML in software engineering can fall into a new decay once again if the proper representational formalisms and programming languages are not used. In this sense, there is a need of adapting evaluation criteria like cross-validation, query learning [3] and reinforcement learning to high-declarative languages [13], rather than the use of non-comprehensive and limited representations for software, such as neural networks or attribute learning, as suggested by [19]. This must not preclude other different techniques, such as explanation-based learning [7], data mining, analogical
reasoning, case-based reasoning, genetic computation, etc., to also be included as usual techniques in software development.

References

3. Angluin, D. “Queries and concept learning” Machine Learning 2, No. 4, 319-342, 1988