On the Use of Inductive Reasoning in Program Synthesis: Prejudice and Prospects

Pierre Flener
Department of Computer Engineering and Information Science
Faculty of Engineering
Bilkent University
TR-06533 Bilkent, Ankara, Turkey
pf@bilkent.edu.tr

Luboš Popelínský
Department of Computing Science
Faculty of Informatics
Masaryk University
Burešova 20
CZ-60200 Brno, Czech Republic
popel@fi.muni.cz

Abstract – In this position paper, we give a critical analysis of the deductive and inductive approaches to program synthesis, and of the current research in these fields. From the shortcomings of these approaches and works, we identify future research directions for these fields, as well as a need for cooperation and cross-fertilization between them.

1 Introduction

Many Software Engineering tasks—such as algorithm design, algorithm transformation, algorithm implementation into programs, and program transformation—are usually perceived as requiring sound reasoning—such as deduction—in order to achieve useful results. We believe that this perception is based on some faulty assumptions, and that there *is* a place for not-guaranteed-sound reasoning—such as induction, abduction, and analogy—in these tasks.

In this position paper, we analyze this case within the research of the Logic Programming community, and more specifically within logic program synthesis research. Moreover, we concentrate on the use of inductive inference as a complement to sound reasoning. This paper is then organized as follows. In Section 2, we take a critical look at the use of deductive reasoning in synthesis, so as to identify the shortcomings of that approach. In Section 3, we do the same thing with the use of inductive reasoning in synthesis. This allows us, in Section 4, to plead for a cooperation and cross-fertilization between the two approaches. In Section 5, we conclude this apology of the use of not-guaranteed-sound reasoning in Software Engineering.

2 On the Use of Deductive Reasoning in Program Synthesis

The Software Engineering tasks of algorithm design, algorithm transformation, algorithm implementation into programs, and program transformation are often believed to be the exclusive realm of sound reasoning, such as deduction. Note that the last two tasks are often merged into the first two, provided the focus is on some form of pure logic programs. However, it is clearly advantageous to separate algorithms from programs, that is to distinguish between the declarative aspects (such as logical correctness) and the procedural aspects (such as control, operational correctness, and efficiency) of software. This distinction is often blurred by the promises of Logic Programming, promises that have however not been fulfilled by languages such as Prolog. It is but a recent trend to dissociate algorithms and programs in Logic Programming

[16] [24] [32] [56] [86]. This corresponds to recasting Kowalski's equation "Algorithm = Logic + Control" [55] as "Program = Algorithm + Control". Since there is no consensus yet on these issues, and in order to keep our terminology simple, the word "algorithm" stands in the sequel for "algorithm or program", hence encompassing all viewpoints.

In the Logic Programming community, this trend of deduction-based approaches is clearly dominant in the proceedings of dedicated workshops such as the LOPSTR series (LOgic Program Synthesis and TRansformation, where "synthesis" stands for some form of (semi-)automated algorithm design) [19] [25] [57] [this volume], and of dedicated sessions at most Logic Programming conferences (ICLP [50], SLP, META, …).

Let's focus our attention now on the most challenging of the four tasks enumerated above, namely algorithm design, and, more precisely, on algorithm synthesis. It should be noted that the line between synthesis and transformation is a very subjective one, and that the synthesizers of some researchers would be transformers for other researchers. For the sake of this paper, we assume the following purely syntactic criterion for distinguishing between synthesis and transformation: if the input and output languages of a tool are the same, then it is a transformer, otherwise it is a synthesizer. Usually, synthesis amounts to the translation from a rich, high-level input language (the specification language) into a less rich, lower-level language (the algorithm language). It is in this sense that we consider synthesis more challenging than transformation. Our restriction of the focus on algorithm synthesis can now be motivated as follows: synthesis generates the objects that can be transformed, so synthesis could just as well generate the transformed version right away. In this sense, it suffices to discuss synthesis here.

The rest of this section is now organized as follows. Section 2.1 briefly relates the various approaches to using deductive reasoning in synthesis. This allows us, in Section 2.2, to enumerate the problems of such deduction-based synthesis. Finally, Section 2.3 contains a partial conclusion.

2.1 Approaches to Deduction-based Program Synthesis

In Logic Programming, synthesis is usually the process of (semi-)automatically translating a specification (usually in a language quite close to full first-order logic plus equality) into a logic algorithm (usually in a language that is a proper subset of the specification language). Such a first-order axiomatization is often just assumed to be a faithful formalization of the intentions[1], though the synthesis process may provide feedback to the specification elaboration process [60]. From such a formal specification, a natural thing to do is to proceed with sound reasoning, so as to obtain an algorithm that is logically correct with respect to the specification. Research from this mind-set can be divided into two main categories [27] [31]:

- *Deductive Synthesis* (also known as *Transformational Synthesis*): meaning-preserving transformations are applied to the specification, until an algorithm is obtained. Sample works are those of Clark [18], Hogger [48], Bibel *et al.* [6], Sato and Tamaki [73] [74], Lau and Prestwich [62], Kraan *et al.* [56], and so on. The theoretical foundations to deductive synthesis are being laid out by Lau and Ornaghi [58]–[61].

1. Let's ignore here the problem that the intentions tend to be unknown or to change over time.

- *Constructive Synthesis* (also known as *Proofs-as-Programs Synthesis*): an algorithm is extracted from a (sufficiently) constructive proof of the satisfiability of the specification. Sample works are those of Tärnlund *et al.* [29] [46], Takayama [83], Bundy *et al.* [16], Wiggins [86], Fribourg [35], and so on.

These two categories are not as clear-cut as one might think, as some synthesis mechanisms can be successfully classified in the two of them [56]. The two approaches are probably only facets of the same technique.

A third category, namely *Schema-guided Synthesis*, should be added: an algorithm is obtained by successively instantiating the place-holders of some algorithm schema, using the specification, the algorithm synthesized so far, and the integrity constraints of the chosen schema. This usually involves a lot of deductive inference sub-tasks, hence the justification for viewing this as deduction-based synthesis. Curiously, this category seems almost absent from the Logic Programming world, although spectacular results are being obtained with this approach in Functional Programming by D.R. Smith [78] [79]. The work of Fuchs [40] and Kraan [56] is definitely schema-based, but their logic program schemas are not finegrained enough to more effectively *guide* the synthesis.

2.2 The Problems with Deduction-based Program Synthesis

Now, what are the problems with deduction-based synthesis, and the current approaches to it? We see basically two problems.

The first problem is related to the current synthesis approaches. A lot of wonderful theoretical effort is being directed at building *theories of algorithms* that may underlie synthesis. Almost any algorithm can be synthesized using these approaches. Another crux is that although these approaches show the derivability of many algorithms (which is in itself a very positive result), the search spaces are exponentially large, making these synthesizers impossible to use in automatic mode. So interactive usage is recommended. But a close look at such syntheses shows that one almost needs to know the final algorithm if any serious speedup is to be expected, which is not what one would expect to have to do with a synthesizer. What is needed *now* is also a *theory of algorithm design* in order to allow efficient traversal of these huge search spaces. The investigations on proof planning [17] and lemma generation [36] are first steps into that direction. But we believe that even more can be done, maybe along the lines of the schema-guided synthesis paradigm mentioned above. Indeed, one of the conclusions of the early efforts at automatic programming was that algorithm design knowledge and domain knowledge are essential if synthesis wants to scale up to realistic tasks [42]. Curiously, few people in the Logic Programming community seem to pay heed to that recommendation by Green and Barstow.

The second problem with deduction-based synthesis is related to the formality of the specifications. Where do the formal specifications come from? If deduction-based synthesizers guarantee total correctness of the resulting algorithms with respect to their specifications, what guarantee do we have that these specifications correctly capture our intentions? These questions are often either dismissed as irrelevant or considered as intractable issues that are left for future research. But what use are these synthesizers if we don't even know whether they solve our problems or not?

Before continuing, we must define the concept of specification, as there is little consensus on such a definition. For the purpose of this paper, a *specification* is a description of *what* a program does, and of *how to use* that program. Such specifications are totally declarative in that they don't express how the program actually works, and they thus don't bias the programmer in her/his task. Specifications have been alternately required to be not (necessarily) executable [47], (preferably) executable [37] [38], and so on.

So what are the problems with formal specifications? As seen above, there is no way to construct formal specifications so that we have a formal proof that they capture our intentions. So an informal proof is needed *somewhere* (at worst by testing a prototype, although testing never amounts to proving), as the purpose of Software Engineering is after all to obtain programs that implement our informal intentions. Writing formal specifications just shifts the obligation of performing an informal proof from the program *vs*. specification verification to the specification *vs*. intentions verification, but it doesn't rid us of that obligation. So specifications could just as well be informal, to prevent a delaying of an informal proof that has to be done anyway.

So we are actually hoping for somebody to write a new paper stating that specifications ought to be informal! Informal specifications should not be mixed up with natural language specifications: they are just specifications written in a non-formal language (without a predefined syntax and semantics). So natural language statements augmented with *ad hoc* notations, or statements in a subset of natural language with a "clear" semantics, would constitute an informal specification [63].

Another indicator why specifications ought to be informal can be obtained by observing the history of program synthesis research [72]: the first assemblers and compilers were seen as automatic programmers, as they relieved the programmers from many of the burdens of binary programming. Some programmers felt like they were only writing some form of specifications, and that the compilers took care of the rest. But after a while, the new "specification languages" were perceived as programming languages! The same story is happening over and over again with each new programming paradigm. One of the latest examples is Prolog: initially perceived by many as a specification language (and still being perceived as such by some people), the consensus now seems that it is a programming language. So how come that one-time specification languages are sooner or later perceived as programming languages, or that there never is a consensus about the difference between formal specifications and programs? The formal specifications of some people are indeed often very close to what other people would call programs. Moreover, formal specifications are often as difficult to elaborate and to maintain as the corresponding programs.

Some researchers don't require formal specifications to be totally declarative, but only as declarative as possible. Indeed, if a specification language allows the procedural expression of knowledge, practice shows that specifiers will use these features. But what does it mean for a formal specification to be declarative? As *writing* recursive statements seems to reflect a very procedural way of thinking[2], a possible criterion for declarativeness could be the absence of explicit recursion in the specification itself as well as in the transitive closure of the union of the specifications of the predicate-symbols used in that specification. But such recursion-free specifications are (possibly?) im-

2. However, recursive statements can be *understood* declaratively: the meaning is "… and so on".

possible to write, as sooner or later one gets down to such fundamental concepts as integer-addition or list-concatenation, which don't have non-recursive finite-length descriptions that completely capture them. So if the most evident symptom of "procedurality", namely recursion, seems impossible to avoid in formal specifications, this would imply that declarative formal specifications don't exist. And since specifications ought to be declarative, this would in turn imply that formal specifications cannot be written.

In our opinion, the solution to all these problems with formality is that formal specifications and programs are intrinsically the same thing! The inevitable intertwining between the formal specification elaboration process and the algorithm design process has already been pointed out by Swartout and Balzer [82].

As algorithm synthesis research aims at raising the level of language in which we can interact with the computer, synthesizers and compilers perform intrinsically the same process. In other words, the "real" programming is being done during the formalization process while going from an informal specification to a formal specification/program (which is then submitted to a synthesizer/compiler). In Logic Programming, there is little research about this formalization process, a laudable exception being Deville's work on hand-constructing logic algorithms from informal specifications [24], a process for which some mechanization opportunities have been pointed out [26].

Note that we are not saying that formal specifications are useless: of course it is important to be able to check whether a formal specification is internally consistent, and to generate prototypes from executable specifications, because this allows early error detection and hence significant cost-cutting. At any given time, formal specifications, even if written in a programming language, will have different purposes (validation, prototyping, contracts, …) and attributes (readability, efficiency, …) than programs [39]. We here just say that formal/executable specifications are already programs, though not in a conventional sense. But conventions change in time, and the specifications of today may well be perceived tomorrow as programs. To understand this, it helps to define programming from a process-theoretic viewpoint (that is, as an activity of carefully crafting, debugging, and maintaining a formal text) rather than from a product-theoretic viewpoint (programming yields a formal text).

2.3 Partial Conclusion about Deduction-based Program Synthesis

Let's summarize in a very synoptic way the situation about deduction-based synthesis:

- Deduction-based synthesis translates a formal specification (with assumed-to-be-complete information about the intentions) into an algorithm.
- Deduction-based synthesis research, in Logic Programming, should incorporate (more) explicit algorithm design knowledge, such as algorithm schemas.
- The deduction-based synthesis approach suffers from the following problems:
  - where do the formal specifications come from?
  - it's impossible to have a formal guarantee that a formal specification correctly captures the intentions;
  - formal specifications are often as difficult to write as programs;
  - there is even no consensus on the difference between formal specifications and programs; in fact, the expression "formal specification" is a contradiction in

terms, and formal specifications and programs are intrinsically the same thing; so synthesis and compilation also are intrinsically the same process.

Let's now move on to the use of inductive reasoning in algorithm synthesis.

3 On the Use of Inductive Reasoning in Program Synthesis

Human beings often understand a new concept after just seeing a few positive (and negative) examples thereof. Machine Learning is the branch of Artificial Intelligence that explores the mechanization of concept learning (from examples). Important subbranches are Empirical Learning (from a lot of examples, but only a little background knowledge) and Analytical Learning (from a few examples, but a lot of background knowledge), the latter being also known as Explanation-Based Learning (EBL) or Explanation-Based Generalization (EBG) [85]. Machine Learning was long cast in the framework of propositional logic, but since the early 1980s, the results are being upgraded to first-order logic. These efforts are nowadays collectively referred to as ILP (Inductive Logic Programming, a term coined by Muggleton [65]), because concept descriptions are there written as logic programs. ILP is somehow a cross-fertilization between Logic Programming and Machine Learning, and between Empirical Learning and Analytical Learning, and is divided into Empirical ILP (heuristic-based learning of a single concept from many examples) and Interactive ILP (algorithmic and oracle-based learning of many concepts from a few examples). The base cycle of every learner is that it reads in examples from a teacher and periodically turns out hypotheses (conjectures at concept descriptions).

An important distinction needs to be made here. Algorithm synthesis from examples is but a niche (albeit a significant one) of ILP. Indeed, algorithm synthesis in general is only useful if the algorithm actually performs some "computations", via some looping mechanism such as recursion or iteration. Straight-line code is always very close to its full specification, and its synthesis is thus a mere rewriting process. So, in particular, algorithm synthesis from examples is only useful if the algorithm actually performs some "computations". But recursive algorithms are only a subclass of all possible concept descriptions, so algorithm synthesis from examples effectively is a niche of ILP. In the following, by "algorithms" we mean recursive concept descriptions, and by "identification procedures" we mean non-recursive concept descriptions. Other differences between ILP in general and induction-based algorithm synthesis in particular are summarized in Table 1 [31].

The central column of Table 1 shows the spectrum of situations covered by ILP research, but it doesn't mean to imply that all learners do cover, or should cover, this full spectrum. The right-hand column however shows the most realistic situation for induction-based algorithm synthesis, that is a situation that *should* be covered by every synthesizer. Let's have a look now at these two columns.

In ILP, the agent who provides the examples can be either a human being or some automated device (such as a robot, a satellite, a catheter, …). It is possible for this agent not to know the intended concept, which means that it may give examples that are not consistent with the intended concept, or that it may give wrong answers to queries from the learner. Examples can be given in any amounts: Empirical ILP systems expect numerous examples, while Interactive ILP systems expect only a few examples, and often

	Inductive Logic Programming	Induction-based Algorithm Synthesis
Class of hypotheses	any concept descriptions	algorithms
Specifying agent	human or machine	human
Intended concept	sometimes unknown	always known
Consistency of examples	any attitude	assumed consistent
# examples	any	a few
# predicates in examples	at least 1	exactly 1
Rules of inductive infer.	selective & constructive	necessarily constructive
Correctness of hyp.s	any attitude	total correct. is crucial
Existence of hyp. schemas	hardly any	yes, many
# correct hypotheses	usually only a few	always many

Table 1: Induction-based Algorithm Synthesis as a Niche of Inductive Logic Programming

construct their own examples so as to submit them to the teacher. Examples may involve more than one predicate-symbol: the instance "Tweety" of the concept "canary" could yield the example:

```
mouth(tweety,beak) ∧ legs(tweety,2) ∧ skin(tweety,feather) ∧
      utterance(tweety,sings) ∧ color(tweety,yellow),
```

which involves many predicate-symbols, but not a *canary*/1 predicate-symbol. The used rules of inductive inference can be either selective (only the predicate-symbols of the premise may appear in the conclusion) or constructive (the conclusion "invents" new predicate-symbols). Selective rules are often sufficient to learn concepts, such as "canary", from multi-predicate examples. There are many learning situations where an approximately correct concept description is sufficient, whereas in other situations a totally correct description is hoped for. For general concept descriptions, there are hardly any useful schemas (template concept descriptions): indeed, such schemas tend to spell out the entire search space, and thus don't decrease its size. For general concepts, there are usually only a few correct hypotheses: for instance, there is probably only one correct definition of the "canary" concept, in any given context.

But in induction-based algorithm synthesis, the most realistic setting is where the specifier is a human being who knows the intended concept and who is assumed to provide only examples that are consistent with that intended concept.[3] "Knowing a concept" means that one can act as a decision procedure for answering membership queries

3. There are of course other settings for induction-based synthesis, such as the intelligent system that re-programs itself in the face of new problems [11]. We think that in such cases a general Machine Learning approach is more adequate as the system can't know in advance whether the new concept has an algorithm or an identification procedure.

for that concept [2], but it doesn't necessarily imply the ability to actually write that decision procedure.[4] Such a specifier cannot be expected to be willing to give more than just a few examples. Examples only involve one predicate-symbol, namely the one for which an algorithm is to be synthesized: for instance, an example of a sorting algorithm could be *sort*([2,1,3],[1,2,3]). The used rules of inductive inference thus necessarily include constructive rules, as algorithms usually use other algorithms than just themselves. Total correctness of the synthesized algorithm with respect to the intended concept is crucial in induction-based synthesis. Algorithms are highly structured, complex entities that are usually designed according to some strategy, such as divide-and-conquer, generate-and-test, global search [79]: algorithm synthesis can thus be effectively guided by an algorithm schema that reflects some design strategy. The existence of many such schemas, and the existence of many choice-points within these strategies entail the existence of many correct algorithms for a given "computational" concept. For instance, sorting can be implemented by Insertion-sort, Merge-sort, Quicksort algorithms, and many more.

So there is a dream of actually synthesizing algorithms from specifications by examples. Since many intentions are covered by an infinity of examples, finite specifications by examples cannot faithfully formalize such intentions, and the synthesizer needs to extrapolate the full intentions from the examples. This is necessarily done by not-guaranteed-sound reasoning, such as induction, abduction, or analogy.

The rest of this section is now organized as follows. Section 3.1 briefly relates the various approaches to using inductive reasoning in synthesis. This allows us, in Section 3.2, to enumerate the problems of such induction-based synthesis. In Section 3.3, we tackle the most commonly encountered prejudice about induction-based synthesis. Finally, Section 3.4 contains a partial conclusion.

3.1 Approaches to Induction-based Program Synthesis

In the early 1970s, some researchers investigated how to synthesize algorithms from traces of sample executions thereof. However, traces are very procedural specifications, and constructing a trace means knowing the algorithm, which rather defeats the purpose of synthesis. Sample work is related in [7] and, more recently, [52]. Regarding induction-based synthesis from examples, there are basically two approaches [31] [27]:

- *Trace-based Synthesis*: positive examples are first "explained" by means of traces (that fit some predefined algorithm schema), and an algorithm is then obtained by generalizing these traces, using the above-mentioned techniques of induction-based synthesis from traces. Sample works are those of Biermann *et al.* [8] [12], Summers [81], and so on, and they are surveyed by D.R. Smith [77]. This research was a precursor to the EBL/EBG research of Machine Learning.
- *Model-based Synthesis*: a logic program is "debugged" with respect to positive and negative examples until its least Herbrand model coincides with the intentions. This is the ILP approach. Sample works are those of E.Y. Shapiro [76], and many others are compiled by Muggleton [66] and surveyed in [67].

4. It would be interesting to examine specifiers (oracles) that are capable of answering other kinds of queries (subset, superset, … [2]) and to investigate other meanings of the phrase "knowing a concept".

Historically speaking, the two approaches barely overlap in time: trace-based synthesis research took place in the mid and late 1970s, whereas model-based synthesis research is ongoing ever since the early 1980s. Indeed, in the late 1970s, trace-based synthesis research hit a wall and partly declared defeat considering that the techniques found didn't seem to scale up to realistic problems. But then, E.Y. Shapiro [76] and others published their first experiments with model-based approaches, and model-based synthesis took over, not only for induction-based algorithm synthesis, but for inductive concept learning in general.

"Linguistically" speaking, the two approaches also barely overlap: trace-based synthesis was pursued by the Functional Programming community, whereas model-based learning is being investigated by the Logic Programming community. Revivals of trace-based synthesis in the Logic Programming community have been suggested by Flener [31] [32] and Hagiya [45].

3.2 The Problems with Induction-based Program Synthesis

Now, what are the problems with induction-based synthesis, and the current approaches to it? We see basically two problems.

The first problem is related to the current synthesis approaches: there seems to be little dedicated induction-based synthesis research any more in the Logic Programming community, as most research seems directed at model-based learning in general. However, as conveyed by Table 1, induction-based synthesis is a sufficiently restricted sub-area of induction-based learning to justify very dedicated techniques. It is illusory to hope that very general learning techniques carry over without major efficiency problems to particular tasks such as induction-based synthesis: since synthesis is akin to compilation (see Section 2.2), this illusion amounts to looking for a universal programming language. The phrase ILP is ambiguous in that it can be understood in two different ways: ILP could mean "writing Logic Programs using Inductive reasoning" (I-LP), or it could mean "Programming (in the traditional sense of the word) in Logic using Inductive reasoning" (I-L-P). The bulk of ILP research accepts the first interpretation.

Some good ideas of trace-based synthesis (such as schema-guidance) haven't received much attention by model-based learning research. Indeed, as seen above, for general concepts there are hardly any schemas that wouldn't spell out the entire search space. Of course, one can use application-specific schemas, but then the question arises as to the acquisition of these schemas. Now, for the particular task of model-based synthesis, there *is* room for schemas [80] [84]: algorithm schemas significantly reduce the search space, they bring "discipline" into an otherwise possibly anarchic debugging process, and they convey part of the algorithm design knowledge called for by Green and Barstow [42]. The other entries of Table 1 provide an agenda for future, dedicated research in model-based synthesis.

Also, there is a fundamental difference between a teacher/learner relationship and a specifier/synthesizer relationship. A teacher usually is expected to know *how* to compute/identify the concept s/he is teaching to the learner, whereas a specifier usually only knows *what* the concept is about, the determination of *how* to compute it being precisely the task of the synthesizer. So a teacher can guide a learner who is "on the wrong track", but a specifier usually can't. A teacher can, right before the learning session, set the

learner "on the right track" by providing carefully chosen examples and/or background knowledge, but a specifier often can't. For instance, most ILP systems can learn the Quicksort algorithm from examples of *sort*/2 plus logic procedures for *partition*/3 and *append*/3 as background knowledge. But this amounts to a "specification of *quicksort*/2", which is a valid objective for a teacher, but not for a specifier: one specifies *sort*/2, a problem, not *quicksort*/2, a solution! We really wonder about the efficiency of ILP-style learners in a true specifier/synthesizer setting, where *a lot of* relevant *and* irrelevant background knowledge is provided. A solution to the ensuing inefficiency would be structured background knowledge, such as classifying the *partition*/3 procedure as a useful instance of the induction-parameter-decomposition placeholder in a divide-and-conquer algorithm schema.

The second problem with induction-based synthesis is that examples alone are too weak a specification approach. Incompleteness results are indeed abundant [3] [10] [41] [51] [68] [69]. It is true that in Machine Learning in general, examples are often all one can hope for. But, as conveyed by Table 1, in synthesis, we usually have the setting of a human specifier who *knows* the intended relation. So s/he probably knows quite a bit more about that relation, but can't express it by examples alone. For instance, it is unrealistic that somebody would want a sorting program and not know the reason why [2,1] is sorted into [1,2] rather than into [2,1]. The reason of course is that $1 \leq 2$, but the problem here is that the $\leq/2$ predicate-symbol cannot appear in the examples.

More generally, the problem is about the lack of provision of domain knowledge to the synthesizer (another recommendation by Green and Barstow [42]), and has been perceived a while ago. Various proposed solutions are type declarations for the parameters [76], type assertions about the intended relation [28], properties of the intended relation [31] [32], integrity constraints about a set of intended relations [20] [21], and bias (all knowledge potentially useful for narrowing the search space [71]), as generally used in ILP. Note that special care needs to be taken not to *require* complete knowledge about the intentions in the assertions/properties/constraints/bias, because otherwise a deduction-based synthesizer would be more appropriate. This is a problem with some of the proposed solutions [23]. Of course, if someone wants to give complete knowledge about the intentions, then the synthesizer should be able to handle it.

Some other often mentioned "problems" with induction-based synthesis are, in our opinion, no problems at all, and we discuss them in the next sub-section.

3.3 Induction-based Program Synthesis: Prejudice and Reality

When faced with research about synthesizing algorithms from examples, some deduction-based synthesis researchers react somewhere in between the paternalistic smile of a father at his child who just completed her/his first Lego house and aggressive attacks about the uselessness of such research. Let's have a look at the most frequently encountered prejudices, and debunk them in the face of reality.

Prejudice: Induction-based synthesis researchers think that they will provide *the* solution to synthesis.

Reality: Induction-based synthesis researchers are fully aware of the limitations of their research. They view it as just the provision of components and tools for software engineering environments. In the synthesizer-as-a-workbench-of-powerful-mini-

synthesizers approach advocated by A.W. Biermann [9] and schema-guided synthesis researchers such as D.R. Smith [78] [79], there *is* a place for induction-based synthesizers, because certain classes of algorithms can be reliably synthesized with little effort from a few examples. As an illustration, the first author estimates that about 50% of the code of his induction-based SYNAPSE synthesizer [31] falls into such categories of algorithms (divide-and-conquer algorithms in this case), and could thus have been written by SYNAPSE itself.

Prejudice: Induction-based synthesis can at most pretend to aim at programming-in the-small. So it is useless as such algorithms are trivial and can often be written faster than the specifications by examples.
Reality: Even though deduction-based synthesis can effectively hope to scale up to programming-in-the-medium (though probably not to programming-in-the-large?), this doesn't mean that strictly less powerful approaches are not useful. One should not forget that synthesis aims at helping all sorts of programmers, not only the skilled ones. Moreover, synthesis aims at the design of any algorithms, not only the complex ones. Finally, synthesis aims at raising the level of language in which the programmer can communicate with the computer, and thinking in terms of examples seems to us of a higher level than thinking in terms of recursion. During the implementation of his SYNAPSE system, the first author felt many times that he would rather use SYNAPSE (if only it existed already!) than work out himself the recursive calls and other more low-level details. In ILP research on algorithm synthesis, a lot of denotation is now being paid to minimizing the number of examples, and the first results are promising [1] [5] [43] [70].

Prejudice: Induction-based synthesis research is useless because it offers no guarantee that the synthesized algorithms are correct with respect to our intentions.
Reality: In *both* the deduction-based and the induction-based synthesis approaches, it takes specification debugging and maintenance to achieve correctness with respect to the intentions. In the *two* approaches, completeness of the algorithm with respect to the specification is guaranteed, but only in the deduction-based approach does partial correctness with respect to the specification make sense. The problem does not lie in the use of not-guaranteed-sound *vs*. sound reasoning, but in the fact that synthesis starts from formal specifications. Whereas with example-based specifications one knows that the specification is but a fragmentary description of the intentions, such is usually not the case with axiomatic specifications, where one only knows that there is a problem when something goes wrong during the synthesis or during the execution of an implementation of the synthesized algorithm. The line of reasoning for the prejudice above could thus also be used to claim that deduction-based synthesis research is useless because it also doesn't offer a guarantee that the synthesized algorithms are correct with respect to our intentions, this because we have no guarantee that our formal specifications are correct with respect to our intentions.

There certainly are other prejudices, but let's leave it at these for now.

3.4 Partial Conclusion about Induction-based Program Synthesis

Let's summarize in a very synoptic way the situation about induction-based synthesis:

- Induction-based synthesis generalizes a formal specification (with known-to-be-fragmentary information about the intentions) into an algorithm.
- Induction-based synthesis research, in Logic Programming, is a niche of ILP, but its specifics are not being catered for. The results are usually inefficiency and inadequateness. For instance, synthesizers should incorporate (more) explicit algorithm design knowledge, such as algorithm schemas.
- The induction-based synthesis approach suffers from the problem that examples alone are too weak a specification approach: additional domain and problem knowledge must be provided.
- There *is* a place for the induction-based synthesis approach.

This finishes our discussion of the deduction-based and induction-based approaches to algorithm synthesis. Let's now plead for a cooperation and cross-fertilization between the two approaches, and actually also with abduction- and analogy-based approaches.

4 Towards a Cooperation and Cross-Fertilization between Deduction-based Synthesis and Other Approaches to Synthesis

From the beginning, ILP has sought cross-fertilization with other fields, be it by definition (ILP is an attempt at cross-fertilizing Machine Learning and Logic Programming) or by "charter" (ILP aims at the cross-fertilization of Empirical Machine Learning and Analytical Machine Learning). Other opportunities for cross-fertilization have been discovered and added to the ILP "charter". Some successful attempts have been with:

- *Data/Knowledge-Base Updating*: cross-fertilization resulted in that learning can now be done from (clausal) integrity constraints, a generalization of examples, and that non-unit clauses can now be asserted [20] – [22]. The extended field is known as Belief Updating or Theory Revision. The discovery of data dependencies in relational and deductive databases has also been examined [30] [53] [75].
- *Theorem Proving*: a procedure may be constructed by an analysis of a failed proof of a formal specification of its predicate-symbol [34] [49] [64]. However, in some cases, an inductive theorem prover is not able to process a formula and thus fails to finish a proof. Induction-based learning methods are then used for inventing new predicates and a new formula is built. It is shown that even in the case where the new formula is not equivalent to the original one, the prover is able to make the next step and to finish the proof [33].
- *Logic Program Transformation*: cross-fertilization with Analytical Learning (EBL/EBG) resulted in Explanation-Based Program Transformation (EBPT) [14], where sample concrete transformations guide the overall abstract transformation process.

So the question now arises as to whether cooperation and cross-fertilization are possible with (other) branches of deduction-based Software Engineering? Some attempts at solving Software Engineering tasks with induction-based techniques have been made, such as logic program synthesis and debugging [5] [23] [28] [43] [76], test case generation [4], and program verification [13], but there was no sign of actual cross-fertilization. So what is the potential of such cooperation and cross-fertilization?

One of the major problems we pointed out with the deduction-based synthesis paradigm is due to the formality of the needed specifications: where do the formal specifications (logical axiomatizations, that is) come from? Following Muggleton's summary of the importance of inductive reasoning in scientific discovery [65], the popular answer is that axioms, representing generalized beliefs, can be constructed from particular facts, which are in turn derived from the senses. Turing is reported to have believed that the problems due to Gödel's incompleteness theorem could be overcome by learning from examples. So a possible cooperation would be mixed-inference specification acquisition followed by deduction-based synthesis/transformation. Deduction-based synthesis can provide feedback to the specification elaboration process [60], but this doesn't assist in the *initial* formalization process and only detects inconsistencies *within* the specification, but not inconsistencies with respect to the intentions.

Another opportunity for cooperation lies in the synthesizer-as-a-workbench-of-powerful-mini-synthesizers approach advocated by A.W. Biermann [9] and schema-guided synthesis researchers such as D.R. Smith [78] [79]: such a workbench should include induction-based synthesizers that are known to reliably converge very quickly to correct algorithms from just a few examples. This would be handy synthesis tools, and we believe they would be used very often. Similarly for tools based on other kinds of inference.

A little-explored avenue for cross-fertilization is the use of deductive reasoning *within* induction-based synthesis, and vice-versa. Indeed, synthesis (especially if schema-guided) can often be broken down into very different sub-tasks (such as instantiating some place-holder of an algorithm schema). So it is likely that some sub-tasks are easier to solve by deductive reasoning, whereas others are more amenable to other kinds of reasoning.

For instance, the SYNAPSE system [31] [32] is schema-guided, starts from specifications by examples and some strong form of axioms (called properties), and features deductive *and* inductive reasoning, according to whichever is preferable for each place-holder of a divide-and-conquer schema. The given properties are used in a constructive way: formulas are extracted from an explanation of the failure of a deductive proof that the current algorithm satisfies the properties, and these formulas are then added to that algorithm. This is a different approach from the usage of assertions [28] or integrity constraints [21], which are used in a destructive way (to reject parts of the current hypothesis) and without any actual deductive reasoning.

This technique is related to abduction [54], which plays an important role when the incorporation of incoming information to an existing theory is impossible or inconvenient: for example in deductive databases and knowledge bases, where adding a new piece of knowledge may cause an inconsistency [15] [44], or in a fault diagnosis, where we are interested in a cause of failure rather than in the failure itself. By abduction, we look for an explanation of the new knowledge, consistent with the existing theory, and then add it to the theory.

Abductive reasoning is non-monotonic, as many explanations may exist for a given piece of knowledge. Another evidence for non-monotonicity is that explanations of two different pieces of incoming knowledge may contradict each other.

In the field of algorithm synthesis, De Raedt [20] pointed out that Shapiro's MIS [76] actually also performs abduction: in the context of multiple predicate learning, abduced facts are not added to the theory, but rather used as the starting point for a synthesis phase. In [20], an abductive technique for the inductive learner CLINT is described, making it capable of learning multiple predicates. The use of abduction in interactive algorithm synthesis from examples is also explored in [70].

We believe that deductive/abductive inference from, and constructive usage of, oracle-answers and extensions of example-based specifications will play an important role in induction-based synthesis (and learning). Conversely, we also believe that deduction-based synthesis will greatly benefit from the inclusion of other kinds of reasoning.

5 Conclusion

In this essay, we have given a critical analysis of the deduction-based and induction-based approaches to algorithm synthesis, and of the current research in these fields within the Logic Programming community. We have identified some future research directions for these approaches, as well as a clear need for cooperation and cross-fertilization between them.

The two approaches and their associated current research efforts have their shortcomings, and, upon close inspection, they even share the most fundamental shortcomings:

- the two current efforts suffer from the fact that (more) algorithm design knowledge (such as algorithm schemas) ought to be injected into the synthesizers;
- the two approaches suffer from the fact that there is no formal guarantee that the synthesized algorithms correctly cover our intentions; so in the two approaches an informal proof of correctness is needed somewhere (usually via specification debugging and maintenance).

We hope to have convinced initially suspicious readers that the intuitive argument of the superiority of the deduction-based approach is based on some faulty and prejudiced assumptions.

Acknowledgments

The first author gratefully acknowledges stimulating discussions with Baudouin Le Charlier (University of Namur, Belgium), Yves Deville (Catholic University of Louvain, Belgium), and Norbert Fuchs (University of Zürich, Switzerland). Thanks of the second author are due to his supervisor Olga Štěpánková (Czech Technical University, Prague, Czech Republic) and to Pavel Brazdil (University of Porto, Portugal) for useful discussions and suggestions. The first author also benefitted from the feedback of the students of his Automatic Program Synthesis course at Bilkent University.

References

ICLP = International Conference on Logic Programming
ILP = International Workshop on Inductive Logic Programming
LOPSTR = International Workshop on LOgic Program Synthesis and TRansformation
LPAR = International Conference Logic Programming and Automated Reasoning
SLP = Symposium on Logic Programming

[1] David W. Aha, Stéphane Lapointe, Charles X. Ling, and Stan Matwin. Inverting implication with small training sets. In F. Bergadano and L. De Raedt (editors), *Proceedings of the* 1994 *European Conference on Machine Learning*, pages 31–48. LNCS 784, Springer-Verlag, 1994.

[2] Dana Angluin. Queries and concept learning. *Machine Learning* 2(4):319–342, April 1988.

[3] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Surveys* 15(3):237–269, September 1983.

[4] Francesco Bergadano *et al.* Inductive test case generation. In *Proceedings of ILP'*93, pages 11–24.

[5] Francesco Bergadano and Daniele Gunetti. Inductive synthesis of logic programs and inductive logic programming. In [25], pages 45–56.

[6] Wolfgang Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence* 14(3):243–261, October 1980.

[7] Alan W. Biermann. On the inference of Turing machines from sample computations. *Artificial Intelligence* 3(3):181–198, Fall 1972.

[8] Alan W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics* 8(8):585–600, 1978.

[9] Alan W. Biermann. Dealing with search. In Alan W. Biermann, Gérard Guiho, and Yves Kodratoff (editors). *Automatic Program Construction Techniques*, pages 375–392. Macmillan, 1984.

[10] Alan W. Biermann. Fundamental mechanisms in machine learning and inductive inference. In W. Bibel and Ph. Jorrand (editors), *Fundamentals of Artificial Intelligence*, pages 133–169. LNCS 232, Springer-Verlag, 1986.

[11] Alan W. Biermann. Automatic programming. In S. C. Shapiro (editor), *Encyclopedia of Artificial Intelligence*, pages 59–83. John Wiley, 1992. Second, extended edition.

[12] Alan W. Biermann and Douglas R. Smith. A production rule mechanism for generating LISP code. *IEEE Transactions on Systems, Man, and Cybernetics* 9(5):260–276, May 1979.

[13] Ivan Bratko and Marko Grobelnik. Inductive learning applied to program construction and verification. In *Proceedings of ILP'*93, pages 279–292.

[14] Maurice Bruynooghe and Danny De Schreye. Some thoughts on the role of examples in program transformation and its relevance for explanation-based learning. In K. P. Jantke (editor), *Proceedings of the* 1989 *International Workshop on Analogical and Inductive Inference*, pages 60–77. LNCS 397, Springer-Verlag, 1989.

[15] François Bry. Intensional updates: Abduction via deduction. In D. H. D. Warren and P. Szeredi (editors), *Proceedings of ICLP'*90, pages 561–578. The MIT Press, 1990.

[16] Alan Bundy, Alan Smaill, and Geraint Wiggins. The synthesis of logic programs from inductive proofs. In J. W. Lloyd (editor), *Proceedings of the ESPRIT Symposium on Computational Logic*, pages 135–149. Springer-Verlag, 1990.

[17] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence* 62(2):185–253, 1993.

[18] Keith L. Clark. *The synthesis and verification of logic programs*. Technical Report DOC-81/36, Imperial College, London (UK), September 1981.

[19] Tim Clement and Kung-Kiu Lau (editors), *Proceedings of LOPSTR'*91. Workshops in Computing Series, Springer-Verlag, 1992.

[20] Luc De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, 1992.

[21] Luc De Raedt and Maurice Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence* 53(2–3):291–307, February 1992.

[22] Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In *Proceedings of ILP'*93, pages 25–40.

[23] Nachum Dershowitz and Yuh-Jeng Lee. Logical debugging. *Journal of Symbolic Computation* 15(5–6):745–773, May/June 1993. Early version, entitled "Deductive debugging", in *Proceedings of SLP'*87, pages 298–306.

[24] Yves Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming, Addison Wesley, 1990.

[25] Yves Deville (editor), *Proceedings of LOPSTR'*93. Workshops in Computing Series, Springer-Verlag, 1994.

[26] Yves Deville and Jean Burnay. Generalization and program schemata: A step towards computer-aided construction of logic programs. In E. L. Lusk and R. A. Overbeek (editors), *Proceedings of the North American Conference on Logic Programming'*89, pages 409–425. The MIT Press.

[27] Yves Deville and Kung-Kiu Lau. Logic program synthesis: A survey. *Journal of Logic Programming, Special Issue on* 10 *Years of Logic Programming*, 1994.

[28] Wlodek Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M. H. Rogers (editors), *Meta-Programming in Logic Programming: Proceedings of META'*88, pages 501–521. The MIT Press.

[29] Agneta Eriksson and Anna-Lena Johansson. Computer-based synthesis of logic programs. In M. Dezani-Ciancaglini and U. Montanari (editors), *Proceedings of an International Symposium on Programming*, pages 105–115. LNCS 137, Springer-Verlag, 1982.

[30] Peter Flach. Predicate invention in inductive data engineering. In *Proceedings of the 1993 European Conference on Machine Learning*, pages 83–94. LNAI 667, Springer-Verlag, 1993.

[31] Pierre Flener. *Logic Algorithm Synthesis from Examples and Properties*. Ph.D. Thesis, Université Catholique de Louvain, Louvain-la-Neuve (Belgium), June 1993.

[32] Pierre Flener and Yves Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation* 15(5-6):775–805, May/June 1993.

[33] Marta Fraňová. Fundamentals of a new method for inductive theorem proving: CM—construction of atomic formulae. In *Proceedings of the* 1988 *European Conference on Artificial Intelligence*.

[34] Marta Fraňová and Yves Kodratoff. *Predicate synthesis from formal specifications or using mathematical induction for finding the preconditions of theorems*. Technical Report 646, LRI, Université Paris-Sud, 1991.

[35] Laurent Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D. H. D. Warren and P. Szeredi (editors), *Proceedings of ICLP'90*, pages 685–699. The MIT Press, 1990. Updated and revised version in [50], pages 39–66.

[36] Laurent Fribourg. Automatic generation of simplification lemmas for inductive proofs. In V. Saraswat and K. Ueda (editors), *Proceedings of SLP'91*, pages 103–116. The MIT Press, 1991.

[37] Norbert E. Fuchs. Hoare logic, executable specifications, and logic programs. *Structured Programming* 13:129–135, 1992.

[38] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal* 7:323–334, September 1992.

[39] Norbert E. Fuchs. Private communications, March–June 1994.

[40] Norbert E. Fuchs and Markus P. J. Fromherz. Schema-based transformations of logic programs. In [19], pages 111–125.

[41] E. Mark Gold. Language identification in the limit. *Information and Control* 10(5):447–474, 1967.

[42] Cordell Green and David R. Barstow. On program synthesis knowledge. *Artificial Intelligence* 10(3):241–270, November 1978.

[43] Marko Grobelnik. Induction of Prolog programs with Markus. In [25], pages 57–63.

[44] A. Guessoum and John W. Lloyd. Updating knowledge bases. *New Generation Computing* 8:71–88, 1990.

[45] Masami Hagiya. Programming by example and proving by example using higher-order unification. In M. E. Stickel (editor), *Proceedings of the 1990 Conference on Automated Deduction*, pages 588–602. LNCS 449, Springer-Verlag, 1990.

[46] Åke Hansson. *A Formal Development of Programs*. Ph.D. Thesis, University of Stockholm (Sweden), 1980.

[47] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal* 4(6):330–338, November 1989.

[48] Christopher J. Hogger. Derivation of logic programs. *Journal of the ACM* 28(2):372–392, April 1981.

[49] Andrew Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov (editor), *Proceedings of LPAR'92*, pages 178–189. LNCS 624, Springer-Verlag, 1992.

[50] Jean-Marie Jacquet (editor). *Constructing Logic Programs*. John Wiley, 1993.

[51] Klaus P. Jantke. Algorithmic learning from incomplete information: Principles and problems. In J. Dassow and J. Kelemen (editors), *Machines, Languages, and Complexity*, pages 188–207. LNCS 381, Springer-Verlag, 1989.

[52] Alípio M. Jorge and Pavel Brazdil. Learning by refining algorithm sketches. *Proceedings of ECAI'94*. 1994.

[53] Jyrki Kivinen and Heikki Mannila. Approximate dependency inference from relations. In *Proceedings of the 1992 International Conference on Database Theory.*

[54] Andonakis C. Kakas, Robert A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation* 2:719–770, 1992.

[55] Robert A. Kowalski. *Logic for Problem Solving*. North Holland, 1979.

[56] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In D.S. Warren (editor), *Proceedings of ICLP'*93, pages 441–455. The MIT Press, 1993.

[57] Kung-Kiu Lau and Tim Clement (editors), *Proceedings of LOPSTR'*92. Workshops in Computing Series, Springer-Verlag, 1993.

[58] Kung-Kiu Lau and Mario Ornaghi. An incompleteness result for deductive synthesis of logic programs. In D.S. Warren (editor), *Proceedings of ICLP'*93, pages 456–477. The MIT Press, 1993.

[59] Kung-Kiu Lau and Mario Ornaghi. A formal view of specification, deductive synthesis and transformation of logic programs. In [25], pages 10–31.

[60] Kung-Kiu Lau and Mario Ornaghi. On specification frameworks and deductive synthesis of logic programs. In this volume.

[61] Kung-Kiu Lau, Mario Ornaghi, and Sten-Åke Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck (editor), *Proceedings of ICLP'*94, pages 665–683. The MIT Press, 1994.

[62] Kung-Kiu Lau and S. D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D. H. D. Warren and P. Szeredi (editors), *Proceedings of ICLP'*90, pages 667–684. The MIT Press, 1990.

[63] Baudouin Le Charlier. *Réflexions sur le problème de la correction des programmes*. Ph.D. Thesis (in French), Facultés Universitaires Notre-Dame de la Paix, Namur (Belgium), 1985.

[64] Raul Monroy, Alan Bundy, and Andrew Ireland. Proof plans for the correction of false conjectures. In F. Pfenning (editor), *Proceedings of LPAR'*94. LNCS, Springer-Verlag, 1994.

[65] Stephen Muggleton. Inductive logic programming. *New Generation Computing* 8(4):295–317, 1991.

[66] Stephen Muggleton (editor). *Inductive Logic Programming*. Volume APIC-38, Academic Press, 1992.

[67] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming, Special Issue on* 10 *Years of Logic Programming*, 1994.

[68] Leonard Pitt and Leslie G. Valiant. Computational limits on learning from examples. *Journal of the ACM* 35(4):965–984, October 1988.

[69] Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie (editors). *Machine Intelligence* 5:153–163, 1970. Edinburgh University Press, Edinburgh (UK).

[70] Luboš Popelínský, Pierre Flener, and Olga Štěpánková. ILP and automatic programming: Towards three approaches. Submitted to *ILP'*94, Bonn (Germany).

[71] *Proceedings of the International Workshop on Machine Learning'*92 *Workshop on Biases in Inductive Learning*. Aberdeen (Scotland, UK), 1992.

[72] Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *IEEE Computer* 21(8):40–51, August 1988.

[73] Taisuke Sato and Hisao Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth-Generation Computer Systems*, pages 195–201, 1984.

[74] Taisuke Sato and Hisao Tamaki. First-order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation* 8(6):605–627, 1989.

[75] Iztok Savnik and Peter Flach. Bottom-up induction of functional dependencies from relations. In *Proceedings of the AAAI*'93 *Workshop on Knowledge Discovery in Databases*.

[76] Ehud Y. Shapiro. *Algorithmic Program Debugging*. Ph.D. Thesis, Yale University, New Haven (CT, USA), 1982. Published under the same title by The MIT Press, 1983.

[77] Douglas R. Smith. The synthesis of LISP programs from examples: A survey. In Alan W. Biermann, Gérard Guiho, and Yves Kodratoff (editors). *Automatic Program Construction Techniques*, pages 307–324. Macmillan, 1984.

[78] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.

[79] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering* 16(9):1024-1043, September 1990.

[80] Leon S. Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In [50], pages 127–140.

[81] Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM* 24(1):161–175, January 1977.

[82] William R. Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM* 25:438–440, 1982.

[83] Yukihide Takayama. Writing programs as QJ proof and compiling into Prolog programs. In *Proceedings of SLP*'87, pages 278–287.

[84] Nancy L. Tinkham. *Induction of Schemata for Program Synthesis*. Ph.D. Thesis, Duke University, Durham (NC, USA), 1990.

[85] Axel van Lamsweerde. Learning machine learning. In A. Thayse (editor), *From Natural Language Processing to Logic for Expert Systems*, pages 263–356. John Wiley, 1991.

[86] Geraint Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. Apt (editor), *Proceedings of the Joint International Conference and Symposium on Logic Programming*'92, pages 351–365. The MIT Press, 1992.