*J.King Saud Univ.*, Vol.8, *Computer & Information Sci.*, pp.1-19 (A.H 1416/1996)

# Survey of Knowledge Based Approaches to Automating Program Formation

**Pavol Navrat**

*Computer Science and Engineering Dept., Slovak Technical University*
*Ilkovicova 3, 812 19 Bratislava, Slovakia*

(Received, 06 Sept. 1993; accepted for publication, 20 June 1994)

**Abstract**. The paper presents a critical overview of the development in the area of automating the program formation, seen in a broader perspective from automatic program synthesis to computer aided software engineering. The fundamental problems are identified in the area of automatic program synthesis, intelligent support of software development, computer aided software engineering. The trends in related fields, especially in artificial intelligence are surveyed and the possible influence is evaluated. Need for urgent development of theory of programming is recognised. In parallel, knowledge engineering methods should be considered because they allow building models of the respective domain.

## Introduction

Practiclly as soon as people started to use computers an effort originated to include computer in the process of programming it. People experienced how difficult task it is to write a program that describes how to solve the given problem by a computer. But they very soon realized that computer is indeed an extremely powerful tool and as such it can be helpful also in the process of forming a program. In a more ambitious wording, researchers discussed as early as in 1954 the topic of "Automatic Programming" [1].

The motivation for automating the programming or more general the software

development process is twofold: to improve quality of software and to improve quality of the process of its development. Of course, the notion of quality is understood here in quite a general sense, encompassing many dimensions: efficiency, correctness, reliability, etc.

The reason why it has been so often very difficult to achieve an acceptable level of quality is the great complexity of the programming task and the insufficient human ability to cope with it.

The vision of automatic synthesis of programs is based on an assumption that we can separate what is the problem from how it can be solved. What is to be solved is expressed in a problem specification. How it can be solved is a matter of algorithmisation.

Of course, to expect from the computer to devise an algorithm for the given problem completely by itself is a very optimistic vision. There have been several other more realistic directions of research, ranging from automatic programming language translation to non automatic computer assistance to various software engineering tasks.

Similarly to shifting attention from automating entirely the problem solving process to providing support to it as we witnessed in artificial intelligence, notably in form of "expert systems", there has also been taking place a shift in the program development process from automating it to providing support to it. Of course, there have been other research directions besides on automatic program sysnthesis before and this should not be understood in such a way that recently this research direction has been abandoned entirely. In fact, there is quite a lot of work done in very different directions, all of them aiming to provide support to the entire software development porocess. Even the more traditional approaches like computer aided software engineeing (CASE) seek progress in incorporating knowledge based techniques e.g., [2].

There are several directions of research all within the general endeavour to create methods and tools to support the software development process in a form of computer assistance to humans. The research directions can be classified according to different criteria, but we have adopted for the presentation the methodology based on focusing on the human-computer interplay. In the fundamentals of this aspect is the amount of automation, i. e. to what extent the human not only plans and devises the formula manipulating operations in the widest sense in the program formation process, but also

actually performs them. Our hypothesis is that it is not necessary for the human actually to perform all the operations. After having programmed them, the computer can perform them with the possible gain in efficiency, reliability etc. This is quite obvious for such operations with programs as program translation, optimisation and other transformations. In this paper, we would like to concentrate more on forming a program. We shall show that also in the process of program formation there is some automation possible, but shall critically review such attempts. We shall survey briefly the area of automatic program synthesis in part 2.

On the other side of the spectrum of possible approaches there are techniques which limit the use of computer to storing, bookkeeping, and other program manipulation operations all of which are transformations that are algorithmically defined. Being consistent with our survey focus, we shall not discuss these approaches to support program formation. they are extrenmely useful in practice but the automation takes place on a relatively low level. Rather we concentrate on those which attempt to provide the support on a possibly higher level of automation. Approaches to such " intelligent " support to software development are surveyed in part 3.

From both the above parts it will become evident that the crucial concept is that of knowledge. The more human knows about how to solve some class of problems, the more he/she can shift onto computer, provided the knowledge can be suitably represented and the operations for its manipulation can be formulated. As a consequence for program formation, any progress can only be achieved if we can come to a deeper undestanding of it. Again, the work towards this goal proceeds in several directions, most notably perhaps within studies of the theory of programming, but from our point of view attempts to capture the programming knowledge in a way that would allow using it for a computerized support to program formation are most interesting. We identify the work being done within the area of learning programming of special importance here, precisely because both programming knowledge and programming supporting tools are the main focus. A short survey is given in part 4. For a more detailed elaboration of all these questions, see [3] .

## Automatic Program Synthesis

The task of automatic program synthesis essentially means solving automatically a specified problem in such a way that the solution is comprehensible to the computer,

i.e. it is a program. It is often considered as a very promising area in computer science, if for nothing else so because it has formulated such an ambitious taks. It is closely related to artificial intelligence because they both share a common task - to automate certain creative human activities. The close relation is often responsible for utilising similar methods to fulfil similar tasks. As an example, in certain automatic program synthesis methods heuristic search is employed in very similar way to that of using heuristics in automatic problem solving. On the other hand, automatic program synthesis maintains also a close connection to the methodology of programming because both study the process of program construction. We cannot automate the process of program construction if we do not understand how to program deeply enough to be able to formalise our knowledge. And again on the other hand, an effort to automate programming can bring us to new programming methods which might supersede the traditional methods of manual programming in the future. This might be a useful side-effect of the endeavour to automate construction of programs even if the complete automation would never be reached.

A word of caution in this connection need not be superfluous, however. The area of automatic program synthesis cannot be believed to be the only candidate for providing a cure to all problems that have been arising in the field of software engineering. In fact, it has been quite clear to the researchers in this area that their goal is a very challenging one, and perhaps too ambitious at the same time.

One very serious difficulty is connected with the form in which specification is expressed. There are various ways to specify a program. As far as the contents are concerned, the problem specification describes how the program output should be related to the program input. 0000As far as the form is concerned, the problem can be specified by a set of examples of input/output pairs or by a set of examples of its computation traces. Both of these ways are essemtially incomplete because the set of examples does not, of course, include all the possible inputs and the corresponding outputs. The other form of specification, which consists of the input and output conditions expressed as formulae of the first-order predicate calculus is complete to serve as a starting point of an inference process of the program. The task of automatic program synthesis can be viewed on this level as the one of proving a theorem.

for all x, there exists y such that: Input(x) -> Output(x,y)

in a constructive way, i.e. as a function f such that

for all x: Input(x) -> Output(x,f(x))

Usually, however, more practically suitable languages are involved. Frequently, the specification has two parts, headed by the words COMPUTE and WHERE for the output and input conditions, respectively. In general, any specification language based directly on predicate calculus inherits from it its accuracy and elegance which are without doubt important properties of any document involved in program synthesis. The big difficulty is, of course, that it is still quite unusual to specify real problems in the language of predicate calculus formulae, to say the least. It is perhaps more realistic to acknowledge that this is extremely rare. We should note, however, that there have been investigated also other formal approaches and devised other formal languages, besides the pure predicate calculus. Let us note at least the Z language [4]. There have been reported works where quite realistic size problems were attempted to be described in that specification language [5].

Another important issue connected with writing a specification of the given problem is that it is a problem solving process in itself. Stated in other words, it is too idealistic to assume a complete problem specification can be written in a single step, before the actual problem solving begins that would eventually lead to forming a desired program. It is much more realistic to recognise that the specification will be developed gradually and it may be incomplete at least initailly. The process of creating it should be considered a part of the problem solving process [6].

Several methods trying to automate the programming task entirely were published in the seventies. They are capable of automatically synthesising programs of the size not exceeding several lines of text. The methods use different forms of program specification and they also employ different problem solving approaches. Synthesis of a program from a few or just one example of its intended behaviour must rely on some generalisation ability, performed usually by means of induction [7] ; [8]. Synthesis of a program from a formula specifying the general form of its input and the corresponding output by inferring it using suitable rules was clearly a case of deducation [9] or of its special kind, transformation [10].

Although induction and deduction are general problem solving principles and using them has been appropriate, the way they were formulated and used was too general to provide for effective construction. Most of the specific knowledge, used by a human programmer when constructing a program, was overlooked by the "automation"

attempting to construct program automatically [11]. This was identified as perhaps the principal difficulty with all the above methods. The cure was proposed in abandoning the requirement of the synthesis to be fully automatic and seeking an advice from the programmer in two ways: on-line, during the synthesis process, viewing it as a continuous interaction with the human programmer assisting the system to overcome the most difficult parts of the synthesis by giving experienced hints or advices [12], and off-line, by preparing bases of formally expressed patterns of programming, programming skills and techniques and bases describing the domain of the problem being solved [13]. Incidentally, this approach was in accord with development in related areas of automatic problem solving, where the shift from general problem solvers toward specific problem solvers relying on specific expert level knowledge gave rise to the so called expert systems. The need of expressing knowledge used in the solution searching process explicitly became clear [14].

It is perhaps worth noting that the issue is in fact of deeper nature. We can explain it by drawing our attention to a similar task of automatic theorem proving. Many methods are based on the resolution principle. They perform a proof by refutation in a way which allows quite efficient mechanisation. On the other hand, such procedures are quite unnatural for humans when proving theorems. Humans do natural deduction more often. But if we tried to mimic the human behaviour when proving theorems, i.e. to mechanise natural deduction methods, we would not be able to take advantage of the particular properties of the resolution method which manipulates formulae in a special clause form. Although the above argument is somewhat simplified, it presents the basic question for artificial intelligence. Is its aim achieving "intelligent" behaviour regardless of the underlying method, which can be totally different from the one used by humans? or, is its aim much more studying human intelligence e.g. by building appropiate models which would describe the so called "intelligent" behaviour and allow experimenting with them on computer? We shall not enter here into such a fundamental discussion for the field of artificial intelligence. We note, however, that it has methodological implications for any attempt to automate creative human activities.

If we wished to discuss what are the actual theoretical techniques concepts in the background of all these apparently differing methods, we would find - after having abstracted from a sufficient amount of detail - that the basis for all of them is search. Indeed, search is one of the fundamental concepts studied in the theory of cumputation, and from a different corner also in the artificial intelligence. All the methods are very specific instances of searching because not enough information is available to formulate

an algorithmic solution. Hopefully it is clear, that this part of the discussion refers to the meta - level of program formation. This (meta -) problem is not algorithmically solvable in a general case. On the other hand, exactly the opposite is assumed on the object level, viz. the object level problem is algorithmically solvable. Otherwise, there would not in general exist the program to be formed.

In the area of automatic programming, no significant breakthrough can be recorded in general. It has, however, produced various results in specific areas e.g., [15]. (We have also made several attempts in this direction, e.g., [16,] ; [17pp139-146] ; [18,pp412-423] ; [19]), Also, there have been achieved interresting results in various related areas which are often remarkable and surprisingly close to the goal of automatic programming, even if not intended or brought in connection as automatic programming aides explicitly. Computer access has become, mainly due to widespread use of personal computers and networking, possible to millions of non-programmers wishing to solve their problems, but lacking in programming "literacy" . When restricted to specific kind of problems from specific problem domain, most solutions can be pre-programmed in one way or the other, so taht forming the ultimate program in man-machine interaction becomes possible. Typical examples are problems concerned with processing large volumes of data. The tools to be named here are fourth generation languages, spreadsheets, hypertext and multimedia processors, and of course various database systems, providing not only database management, but also support for forming procedures of required data processing.

Automatic program synthesis research continues with many lessons learned. [20] have published an interesting method very recently. They propose the following approach to manage the complexity of the search. Synthesis must be confined to a single domain, which allows extensive use of domain-spefific knowledge. Moreover, design space must be separated into smaller problems using abstraction. Knowledge must be used to prune the design space and hence the complexity of design. They developed an automatic program synthesis prototype that integrates domain-specific knowledge with a variety of automatic program synthesis techniques, called ELF, and concentrates on knowledge representation and reasoning issues. The domain chosen is CAD tools that route the paths for wires in VLSI circuits, and printed circuit boards. Typically, such tools are programs of complexity of 1000 to 2000 lines of C code. Comparing with many previously published works on the automatic program synthesis, this research is an example that demonstrates usefulness of its results in solving real engineering problems.

As another fruitful direction of research, program formation based on reuse of existing software is to be mentioned. Indeed, it is very likely that the largest body of programming know-how is embodied in the already existing and running software. Also similarly to other branches of engineering, a situation when a completely new solution is devised from scratch, is in fact very rare. All in all, rather than attempting to solve each new problem anew, as much of the existing software should be reused as possible. The idea can certainly be accepted on such an abstract level, but on more concrete levels, many practical difficulties arise. Reusing any existing program requires understanding it, which is of course much more difficult on an operational level of the concrete programming language, than on a level of high level specification of what it does. Formation of software from reusable components studied e.g., [21]. Formation of a new program that reuses an existing one (or more of them) relies on similarity of specifications, which allows the reuse. But similarity is not identity, so formation includes modification as it involves more precisely speaking only partial reuse. Again, modification on a concrete language level is much more difficult than on an abstract level. And rather than reusing the result of program development, it may be more fruitful to reuse (replay) the development itself for different but similar problem. Even before the actual reuse, there is perhaps the most difficult problem to be mastered: how to retrieve the approapriate existing suitable software (development) and establish the proper correspondence, which amounts to recognising similarity. The problem of program development based on reuse was studied e.g., [22]. Synthesis of programs using derivational analogy is presented e.g., [23,pp7-55].

To conclude the critical discussion on automatic program synthesis and related approaches, the question does not seem to be how to automate (fully) the program formation process, but rather how high can be raised the level on which the human participates in the process. Abandoning the dream of a full automation, the approaches to a more "intelligent" computer support to program formation are a more realistic alternative, as we shall see in the next part.

## Intelligent Support for Software Development

The progress in software development methodology has been sought in at least two directions. The "classical" line originates from the developments of more systematic analysis and design methodologies and their possible support by various tools, and could be covered approximately by the CASE umbrella. The "sci-fi" line originates in the automatic programming endeavour which was trying to eliminate any direct presence of

a human programmer in the program formation process. So, while the latter attempted to restrict the role of a human to that of specification source (and originally, of course, creator of automatic programming procedure), the former did not attempt to see the role of a machine any bigger than that of a tool, restricted to realise some non-creative tasks in a passive way. Seen from such a perspective, it is most likely that none of these lines has utilised, or not even attempted to utilise all available potential. Of course, there exists a wide range of possible research directions which combine ideas from both these lines. Among those who were able to identify the crossroad and outline a realistic vision, we find the work of [11] of much importance. The vision of an intelligent, interactive software development tool (they call it the Programmer's Apprentice) which will assist software engineers in all phases of the programming process has proved to be a very fruitful ambition.

We should always have in mind that the range of methods and tools offering computerised support to software developers covers also many very practically important achievements. They may have been presented with less scientific ambition, but their value is in their wide everyday use. From the process of program formation, not only translation of its high language formulation into an executable code, but also creating the program text, manipulating with stored form in the computer, maintaining it, etc. were automated. Such is a notion of a syntax directed editor e.g., [24,pp125-136]. These services were made available not only by "stand alone" general tools like editors, but also in a more or less intergrated form centred usually around a specific programming language, like Basic or Lisp. The case of Lisp was especially significant. The very nature of Lisp as a functional language implemented via interpretation defined in Lisp itself allows for relatively easily defined and implemented manipulations of Lisp language texts, including its evaluation in special modes. There were sophisticated utilities included in such Lisp programming systems, as e.g., INTERLISP [25]. By providing access to tools supporting virtually all the routine, non-creative parts of the programmer's task in a uniform manner, such systems were becoming programming environments.

Recently, an outcome of the work on the Programmer's Apprentice project, a prototype knowledge representation and reasoning system called Cake has been described [26]. It serves in support of developing knowledge-intensive, evolutionary, intelligent assistants for software development. (It is thus a meta development tool).

The original vision remains as for today, more than fifteen years later, still a

vision to be fulfilled. And this despite the fact, that there were made numerous other attempts along the suggested lines, many of them not less important or innovative than the one mentioned above. Indeed, we see incomparably more works reported on the subject than let's say fifteen years ago. Obviously, the problem is approached from various corners. Some rely almost entrirely on artificial intelligence techniques, e.g. [27] ; [26], others use AI techniques to cope with certain issues [2] and still others do not explicitly use any AI techniques at all [28,pp180-188] ; [29,pp257-271].

It is important to identify basic orientations in the field of knowledge based software engineering. Besides the aforementioned shift in emphsis from automatic programming to augmenting human as a basis in building human-computer cooperative problem solving tool, [30] put stress on domain orientation. They hope domain orientation would allow to reduce the large conceptual distance between problem-domain semantics and software.

In the area of computer aided software engineering (CASE), much research is oriented toward taking into account the recent developments in knowledge engineering. [2] presented work named 'the project ASPIS' which is based on an idea of explicit representation of the methods employed (the meta level knowledge) as well as of the domain knowledge (the object level knowledge).

There are other projects, on the other hand, which do not make an explicit reference to any special artificial intelligence technique. Project PROSPECTRA [31, pp187-190] is based on a rigorous methodology for developing correct ADA Programs are developed from an initial formal requirements specification in a process of step by step transformation. Because the specification is expressed formally, correctness of the design can be verified during the whole process. The aim of the project is to make software development an engineering discipline. The project hopes to provide not only coherent methodology, but also comprehensive support system.

Currently, a strong shift toward domain specific knowledge seems to be dominant. While its importance is clearly beyond doubt, we should nevertheless like to stress a need for balance here among different kinds of knowledge. In particular, programming knolwedge, or more generally knowledge on software development, plays equally important role in our opinion. It is embodied in all systems presented implicitly as the supporting tool iteslf. An important exception which we should like to mention, is the Programmer's Apprentice project [26]. It uses the programming knowledge expressed

also explicitly, providing special representation formalism for that. Indeed, there is no reason why this kind of knowledge ought to be excluded from the use in the reasoning process. On the contrary, there are good reasons for incorporating such knowledge e.g., in design, implementation, or debugging phases, for explanation or justification purposes, in guiding the development process.

In providing intelligent support, besides especially tailored knowledge based tools like those above, there could be found useful also results from kngwledge engineering aiming to devise more general tools. For our attempts in that area, cf., [31, pp187-190] ; [32, pp351-360] ; [33, pp65-73].

On the other hand, a critical discussion should not avoid the fact that by taking simply the methodology of knowledge based systems and using it to fulfil some of the tasks in the software development process a substantial progress is hardly to be expected. In the heart there is the problem of building bases of explicitly represented knowledge. Rather than concentrating solely on that, more emphasis should be given to techniques of supporting reuse of software. In order to create software that allows easy reuse, special languages and methodologies should be sought, perhaps in the direction of "object oriented" ones.

## Support to Learning Programming

One of the crucial problems connected with use of knowledge based approaches is capturing the knowledge involved. This includes not only choice and availability of informed knowledge sources (e.g., domain experts, literature, experience), but also method of acquiring it. Besides the traditional way of asking human experts to formulate their know-how as knowledge pieces in a knowledge representation language, there are attempts to elicit them from examples or other information automatically in a process of machine learning.

Once a base of knowledge relevant to a certain problem is created and it is sufficiently complete in some suitable sense, it can be considered a model of that domain. As such, it can serve as a source of knowledge for those who do not know about it much and/or wish to learn about it. Now, the situation is reversed comparing to the one above: human learning is the issue.

In the following, we whould like to concentrate briefly or the question of support

to learning programming or software development. Of course, in some sense we can consider any tool used by the pupil during programming attempts as providing support. This, of course, is too broad a definition for the specific purposes related to the learning process. Indeed, there are several research directions which concentrate on following special questions, among others:

1- what knowledge (languages, skills, methods) should be chosen to be acquired by the learner (the contents)
2- what representation of the knowledge is suitable for acquiring by the learner (the form)
3- what approach should be taken to the learning process (the method)

There have been a lot of work in the indicated areas. Psychological aspects of the learning process are studied by [34] ; [35] investigated knowledge involved in learning elementary programming. [36] seek an alternative method to the more ' traditional' rule based approach to learning, stressing active problem solving involvement of the learner from the beginning.

From among the several more recent research projects aiming to develop intelligent tutoring systems for learning programming, there are at least the following ones to be mentioned : The Lisp Tutor [37] for learning introductory programming in Lisp and PROUST [38] for the problem of program analysis which is however based on inferring a plausible design process.

PROUST [38] is a system that analyses novices' Pascal programs which can contain "bugs" Programs are supposed to be from a limited class given implicitly by the available knowledge. The analysis is based on reconstructing intentions from the problem specifications given to the programmer and from the analysed program itself. The system derives automatically a non-algorithmic description of the program. The description is based on so called plans and strategies. The analysis is then in fact a comparison of intended functions and structures to actual ones.

The knowledge base on programming includes the following main kinds: goals, plans, and code. Goals arise from the problem specification. Plans describe how to achieve them. To get the actual program code, transformation rules and also so called "bug rules" (explaining known bugs) are applied. So the system "knows" about the usual ways novices make errors when solving the specified problem. Using the described

knowledge, PROUST searches for implementations of the given specification for which there is evidence in the actual program. It can then not only recognise a bug occurring in the program, but also provide a hypothesis on the programmer's misconception that led to it.

The Lisp Tutor [37] follows the student during forming a program step by step. So contrary to PROUST which gets a syntactically correct complete program, ignoring the actual intermedicate steps but having to reconstruct them, Lisp Tutor is present at and aware of every step that has led to the final program. Each step is in fact an application of a rule. This comes from an assumption of the underlying theory of cognitive psychology [39] that cognitive functions can be represented as sets of production rules. Lisp Tutor has available a fairly large base on programming knowledge expressed as rules. It closely monitors every student's action during the program formation process and intervenes as soon as it identifies an error. To be able to do so, it has available also "buggy" variants of the rules. Further research related to this project's results has been done e.g., on using analogy.

It should be noted, however, that no matter how appealing appears to be the idea of expressing programming knowledge in form of plans as employed in e.g., PROUST, there are several questions open. Some of them were discussed very recently in [40]. They note that e.g., the process of programming is in fact more complex and besides programming plans, strategies of their use must be taken into account. There is also a question whether any 'sufficiently complete' knowledge base, taken as externalisation of the programming theory, would be suitable for learning programming, without being at the same time an implementation of a sound psychological theory of skill acquisition.

An interesting framework for learning programming techniques provides programming in Prolog. As a logical programming language, it facilitates reasoning on a high level of abstraction. Work on teaching prolog techniques, similar to programming plans, is reported [41].

One of the central underlying concepts in acquiring and representing knowledge seems to be the one of programming plan. A more critical look allows us to recognize finer categories of this rather fundamental concept. Besides the already mentioned programming techniques, researchers study also semantically augmented programming primitives [42], programming cliches [26] and units [43].

Of course, research in the area of supporting learning programming has attracted many other researchers, as well. To name just one more example, architecture and knowledge representation of an intelligent environment for learning programming has been described in [44, pp114-124]. For ourselves, methodological aspects were our concern e.g., [45, pp149-152] ; [3] , architectural aspects of the supporting tool e.g., [46, pp375-379] ; [18] ; [32], the programming knowledge itself in e.g., [31] ; [47].

## Conclusions

We can recognise the steady progress in the endeavour to incorporate the computer in the task of programming. Our common everyday experience provides engough evidence about the difficulty of the programming task. We understand now that the reason lies in the complexity of the task and immaturity of the field. On the other hand, the same experience shows us many fascinating possibilities hidden in computer use. Is has been only natural to try to use computer in the task of programming it. There have been achieved results on many fronts. In fact, the area has become very broad, encompassing attempts of full automation as well as tools not attempting automation of any but routine, mechanical manipulations; or methods completely relying on artificial intelligence techniques as well as such which reject any explicit influence of artificial intelligence.

Looking back, it is indeed fascinating to see the wealth of various methods aiming to incorporate the computer into the process of program formation, that were developed over nearly four decades.

It appears that this particular area has put too much stress on automation. The original motivation might have been at least partially a selfish one, i.e. programmers who by writing programs made available the computer's powerful support to users from all kinds of different fields, should at least attempt to make the computer to support them in their own program-forming task. It was perhaps not very fortunate to give such endeavour the name of automatic programing. Being imprecise and sounding too ambitious, its only advantage has been that it was sufficiently attractive to serve as a challenge to many researchers. In this sense, the name experienced a similar fate to that of artificial intelligence. The field with such name has become vulnerable to misinterpretations of its fundamental aims. In [48], automatic programming was described as a contradiction in terms, because he sees programming as human symbol manipultion contrary to computing seen as mechanized symbol manipulation. However,

it is not clear why humans, once they realize the potential of the mechanism they have, i.e. its capability to manipulate symbols without errors and efficiently at the same time, should not use it to support them in their own symbol manipulations

There can be made several conclusions on the background of the survey given above:

1- Further development of the theory of programming is urgently needed. It should give uniform formalisation of the process e.g., [49], providing basis for construction models of the process and investigating its properties, as well as proposing methods to achieve desired properties e.g., [50]. Especially, methods of combining different theories should be sought. This is extremely important because in the process of program derivation, there are frequently several theories involved: one decribing the programming language level, another describing the problem domain, yet another describing the solution etc.

2- Formalisation allows devising formal methods and formal methods allow possible automation e.g., [51] ; [52].

3- There are imposed different important requirements on software construction, be it concerned with the process itself, like efficiency of using resources including proggrammer(s), or be it concerned with result of the process, i.e. the program constructed like its correctness, relaibility, efficiency. If it is to meet them, it must follow more elaborated methods which would free the programmer from tasks which are mechanical, non-creative, clerical and which guide him/her through the whole process in such a way that meeting the requirements in not only possible, but guaranteed.

4- Besides purely theoretical approaches, more practical (engineering) approaches to constructing model of the programming process must be tried as well. Here, it is advisable to make use of knowledge engineering methods, and of artificial intelligence in more general terms. However, it should be understood that these approaches do not present a fundamentally different alternative, but rather an attempt to propose more suitable ways of representing the axioms, therorems and inference rules, and techniques of organizing the inferences. The engineering is concerned with both the suitability to human ( in the above sense ) and the suitability to increasing the efficiency of processing.

## References


[1] Office of Naval Research.*"Symposium on Automatic Programming for Digital Computers."* Washington, D.C.: Office of Naval Research, Dept. of Navy, 1954.

[2] Aslett, M.J. *"A Knowledge Based Approach to Softwere Development"*. Amsterdam, The Netherlands: North Holland Publishing Co., (1991).

[3] Navrat, P. "Intelligent Support for Software Construction, and Higher Education in Informatics at The Slovak TU : teh DEC Connection." In : *Proc. DECSYM 92 Latest Trends in Computing*, Side-Antalya, (1992), 253-263.

[4] Spivey, J.M. "*The Z Notation*." Hemel Hampstead: Prentice Hall,1989.

[5] Milnes, B.G.; Pelton, G.; Doorenbos, R. ; Hucka, M.; Rosenbloom, P., and Newell, A. "A Specification of The Soar Cognitive Architecture in Z." *Rept. CMU-CS-92-169, School of Computer Science*, Carnegie Mellon University, Pittsburgh, (1992).

[6] Fickas, S. "A Knowledge Based Approach to Specification Acquisition and Construction." *TR -85-13, Dept. of CIS*, University of Oregon, (1985).

[7] Biermann, A.W. and Krishnaswamy, R. "Constructing Programs from Example Computations." *IEEE Transactions on Computers*, C-24, No.2 (1976), 141-153.

[8] Siklossy, L. and Sykes, D.A. "Automatic Program Synthesis from Example Problems." In: *Advance Papers IJCA14*, Tbilisi, (1975), 268-273.

[9] Manna, Z. and Waldinger, R. "A Deductive Approach to Program Synthesis." *ACM Trans. on Programming Languages and Systems*, 2, No.1 (1980), 90-121.

[10] Burstall, R.M. and Darlington, J. " A Transformation System for Developing Recursive Programs." *Journal of the ACM*, 24, No.1(1977), 44-67.

[11] Hewitt, C.and Smith, B. "Towards a Programming Apprentice." *IEEE Transactions on Software Engineering*, SE-1, No.1 (1975), 26-45.

[12] Vojtek, V. ; Molnar, L. and Navrat, P. "Automatic Program Synthesis Using Heuristics and Interaction." *Computers and Artificial Intelligence*, 5, No.5 (1986), 427-442.

[13] Green, C. and Barstow, D.R." On Program Synthesis Knowledge." *Artificial Intelligence*, 10, (1978), 241-279.

[14] Molnar, L.; Navrat, P.and Vojtek, V. "Heuristic Search with Global and Local Heuristics." *Computers and Artificial Intelligence*, 5, 417-126.

[15] Cacace, F. ; Ceri, S. ; Tanca, L. and Crespi; Reghizzi, S. "Designing and Prototyping Data-Intensive Applications in The Logres and Algres Programming Environment", *IEEE Transactions on Software Engineering*, 18, No.6 (1992), 534-546.

[16] Molnar, L. and Navrat, P. "Automation of Program Creation and Methodology of Programming." (In Slovak), *Elektrotechnicky Casopis*, 36, No.4 (1985), 316-323.

[17] Navrat, P.; Molnar, L. and Vojtek, V. "Using Automatic Program Synthesizer to Generate Solutions from Diverse Problem Environments." *Computers and Artificial Intelligence*, 7, No.2 (1988).

[18] Navrat, P.; Molnar, L. and Vojtek, V. "Using Automatic Program Synthesizer as a Problem Solver: Some Interesting Experiments."In: J. Davenport (Ed.) *Proc. EUROCAL'87, LNCS 378*, Berlin: Springer, (1989).

[19] Molnar, L. ; Navrat, P. and Vojtek, V. "A System for Automatic Implementation of Abstract Data Types." *Computers and Artificial Intelligence*, 6, No.5 (1987), 481-488.

[20] Setliff, D. and Rutenbar, R. A. " Knowledge Representation and Reasoning in a Software Synthesis Architecture." *IEEE Transactions on Software Engineering*, 18, No.6 (1992), 523-533.

[21] Neighbors, J.M. "The Draco Approach to Constructing Software from Reusable Components." *IEEE Transactions on Software Engineering*, SE-10, No.9 (1984), 564-574.

[22] Wile, D.S. "Program Developments : Formal Explanations of Implementations" *Communications of the ACM*, 26, No.11 (1983), 902-911.

[23] Bhansali, S. and Harandi, M.T. "*Synthesis of UNIX Programs Using Derivational Analogy.*" Machine Learning, 10, No.1 (1993).

[24] Navrat, P. and Klaudiny, M. "A Syntax Directed Editor of Pascal Programs." (In Slovak) In: *Proc. Modern Programming 1983*, Part 2, Zilina (1983), 125-136.

[25] Teitelman, W. "INTERLISP Reference Manual." *Xerox Palo Alto Research Center*, Palo Alto (1978).

[26] Rich, C. and Feldman, Y.A. "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development." *IEEE Transactions on Software Engineering*, 18, No.6 (1992), 451-469.

[27] Fickas, S. and Helm, B.R."Knowledge Representation and Reasoning in The Design of Composite Systems." *EEE Transactions on Software Engineering*, 18, No.6 (1992), 470-487.

[28] Keiser, G.E. and Feiler, P.H. "An Architecture for Intelligent Assistance in Software Development." In : *Proc. ACM*, (1987).

[29] Krieg-Bruckner, B. " The PROSPECTRA Methodology of Program Development." In : J. Zalewski, and W. Ehrenberger (Ed.) : *IFIP 1989*, Amsterdam: Elsevier, (1989).

[30] Fisher, G. ; Girgensohn, A. ; Nakakoji, K. and Redmiles, D. "Supporting Software Designers with Integrated Domain Oriented Design Environments." *IEEE Transactions on Software Engineering*, 18, No.6 (1992), 511-522.

[31] Navrat, P. and Mlada, I. "What Knowledge is The Knowledge Based Programming Based On? : An Inquiry into Knowledge Sources." In:I. Plander (Ed.) *Proc. Artificial Intelligence and Information-Control Systems of Robots 89*, Amsterdam: North-Holland, (1989).

[32] Navrat, P. and Fric, P. "A Tool for Knowledge-Based Systems Development." In: V. Marik (Ed.) *Proc. Artificial Intelligence Applications Conference AI'91*, Prague, (1991).

[33] Gasparovic, L. and Navrat, P. "Extalk - Smalltalk Based Expert Systems Development Tool." In: A. Mrazik (Ed.): *Proc. East EurooPe'91, Short Papers*, Bratislava (1991).

[34] Anderson, J.R.; Farrell, R, and Sauers, R. "Learning to Program in LISP." *Cognitive Science*, 8, (1984), 87-129.

[35] Soloway, E. and Ehrlich, K."Empirical Studies of Programming Knowledge." *IEEE Transactions on*

*Software Engineering*, 10, No.5 (1984), 305-321.

[36] Boyle, T. and Margetts, S. "The CORE Guided Discovery Approach to Acquiring Programming Sills." *Computers and Education*, 18, No. 1-3 (1992), 127-133.

[37] Anderson, J.R. and Skwarecki, E. "The Automated Tutoring of Introductory Computer Programming."*Communications of the ACM*, 29, (1986), 842-849.

[38] Johnson, W. L. and Soloway, E.M, "PROUST : An Automatic Debugger for Pascal Programs." *Byte*, 10, No.4 (1985), 179-190.

[39] Anderson, J.R.*The Architecture of Cognitio*, Combridge: Harvard University Press, 1983.

[40] Davies, S.P. and Castell, A.M. "Embodying Theory In Intelligent Tutoring Systems." An Evaluation of Plan-Based Accounts of Programming Skill, *Computers and Education*, 20, No.1(1993), 89-96.

[41] Brna, P. " Teaching Prolog Techniques."*Computers and Education*, 20, No.1(1993), 111-117.

[42] Bertels, K.; Vanneste, P., and De Backer, C. "A Cognitive Model of Programming Knowledge for Procedural Languages." In: I. Tomek (Ed.) *Computer Assisted Learnings 92, Proceedings of ICCAL* 92, Springer Verlage, (1992), 124-135.

[43] Jonckers, V, "A Framework for Modeling Programming Knowledge." *PhD. Thesis, Vrije Universiteit Brusel*, Dienst Artificiele Intelligentie, (1987).

[44] Brusilovsky, P.L. "Towards an Intelligent Environmentt for Learning Introductory Programming." In: E. Lemut; B. du Boulay, and G. Dettori (Eds.) *Cognitive Modesls and Intelligent Enviornments for Learnign Programming.* Springer -Verlag, (1992).

[45] Molnar, L. and Navrat, P. "Problem Solving Aspects in The Education of Programming." In: *Proc. 27. Internazionales Wissenschaftliches Kolloquium*, Ilmenau, Technische Hochschule, (1982).

[46] Navrat, P. "Towards a Master Programmer : A Paradigm for Automated Tutoring of Programming." In : I. Plander (Ed.) *Proc. Artificial Intelligence and Information- Control Systems of Robots 87*, Amsterdam: North-Holland, (1987).

[47] Navrat, P. and Rozinajova, V. "Making Programming Knowledge Explicit." *Computers and Education*, 21, No.4 (1993), 281-299.

[48] Dijkstra, E.W. "On Cruelty of Really Teaching Computing Science." *Communications of the ACM*, 32, No.12 (1989), 1398-1404.

[49] Turski, W.M. and Maibaum, T.S.E. "*The Specification of Computer Programs.*" Wokingham: Addison Wesley, (1987), 287.

[50] Smith, D.R. and Lowry, M.R. "Algorithm Theories and Design Tactics." *Science of Computer Programming*, 14, (1990), 305-321.

[51] Reddy, U.S. "Formal Methods in Transformational Derivation of Programs." In: *Proc. ACM SIGSOFT Intern.* Workshop on Formal Methods in Software Development, Napa, (1990), 104-114.

[52] Cooke, D.E. "Towards a Formalism to Produce a Programmer Assistant CASE Tool." *IEEE Trans. on Knowledge and Data Engineering*, 2, No.3 (1990), 320-326.

# مسح عام لطرق مكننة تطوير البرامج باستخدام قواعد المعرفة

**بافول نافرات**

*قسم علوم هندسة الحاسب، جامعة سلوفاكيا التقنية*
*الكوفيكوا، براتشلوفا، سلوفاكيا*

(قدم للنشر في ٠٦/٠٩/١٩٩٣م، وقبل للنشر في ٢٠/٠٦/١٩٩٤م)

**ملخص البحث** . تعطى الورقة مسح ناقد عام للتطورات في مجال مكننة البرمجة بصورة واسعة من التكوين الأوتوماتيكي للبرامج إلى هندسة البرمجيات بمساعدة الحاسوب. كما تحدد الورقة المشاكل الأساسية في مجالات التكوين الأوتوماتيكي للبرامج، الدعم الذكي لتطوير البرامج والهندسة البرمجية بمساعدة الحاسوب.

كذلك تقوم الورقة بمسح وتقويم للاتجاهات العامة في المجالات ذات العلاقة، خاصة مجال الذكاء الاصطناعي. من خلال الدراسة، توضح الورقة أن هناك حاجة ماسة للتطوير في مجال نظريات البرمجة كما توصى الورقة بأنه ينبغي الأخذ في الاعتبار أساليب هندسة المعرفة ودورها في تطوير نماذج في المجالات ذات العلاقة.