Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 53–61

On the role of the reusability concept in automatic programming research

Ladislav Samuelis
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: `Ladislav.Samuelis@tuke.sk`

Abstract. The central question of this paper is: "How the concept of reusability influences the research in automatic programming?" After discussing briefly the historical background of the question we analyze the research of automatic programming from the software reuse point of view. We try to show the presence of the reusability concept in the automatic programming research throughout its relatively short history. Based on observations, we argue that the concept of software reuse is an inherent idea of the automatic programming research. Finally, we stress the necessity to introduce the historical view into the curricula of teaching informatics at universities.

1 Introduction

The motif of automatic programming is spreading over the history of software engineering in various dimensions. It has been a moving target, which is constantly shifting to reflect increasing expectations. We observe recently the incredible increase in power of the hardware. This increase is itself the reason for the incredible growth of the software complexity. The Wirth's [1] "law": *Software is getting slower, faster than hardware is getting faster*, allegorically points to this fact.

The big challenges for automatic program construction are presented (e.g.) by Laurianne McLaughlin [2] in the following manner:

AMS 2000 subject classifications: 68N19
CR Categories and Descriptors: I.2.2 [Automatic programming]: Subtopic - Program synthesis; D.2.13 [Reusable Software]: Subtopic - Reuse models.
Key words and phrases: software reusability, automatic programming

- To produce good runtime performance;
- To produce code that someone can look at, deal with, and understand;
- To ensure that the code is provably correct.

The aim of this contribution is highlighting the role of the reusability concept within the automatic program construction efforts in the emergent science of software engineering. In particular, to show how the concept of reuse is interwoven into the automatic programming research. We demonstrate selected research ideas in automatic programming in chronological order.

In order to address the question: "How the concept of reusability influences the research in automatic programming?", we have collected several leading ideas, which are grouped around the term of *automatic program construction or program synthesis.* This selection is in no way a complete and representative survey of the field. We hope that the reflections and discussions to the selected concepts will attract minds and in this way enable more deep comprehension of the notion of automatic program construction.

We believe that students studying informatics have to study the history of software engineering very seriously. In particular they have to study deeply the intellectual movements and to read the original works of informaticians whose ideas mostly influenced the field. Students may observe the blind alleys in the history, steps backward, and to follow the evolution of ideas. Pondering and discussions about the ideas could encourage students to create their own ideas, standpoints, and in this way to join the community of educated informaticians. This historical view should be also an essential part of teaching informatics at the universities. This is considered as nutriment that is a necessity for being educated and later being expert in informatics.

This paper begins with a brief history of the idea of automatic programming in Chapter 2. It continues by discussing the reusability principle in the era before object-oriented programming, in Chapter 3. Obstacles to software reuse are referred in Chapter 4 and finally we make conclusions in Chapter 5.

2 Where does the idea of automatic program construction come from?

Looking backward into the history, we observe that the first mentioning and the first discussions around the *industrial production of software* appeared formally in 1968 and were presented at a NATO conference by McIlroy [3].

Many expert-like papers and books concerning automatic software construction have been already presented. However, only few of them tried to discover "laws" which are beyond the software construction or program synthesis along the history of computing [4].

This is an exciting story to follow the interpretations of the notion of automatic program construction and to observe the interleaving of the more-less well-known ideas. This endeavor involves also an attempt to untangle ideas and concepts, which appear suddenly in various contexts.

The term "automated programming" (automatic program construction or program synthesis) is used to refer to the study and implementation of methods for automating a significant part of the process of creating and enhancing software. Its meaning somewhat varies, but often includes several aspects of the programming processes.

A broader goal of this field is to make computer programs much easier by means of automation of the software creation process. More particular goals include increasing software productivity, lowering costs, increasing reliability, making more complex systems tractable, as mentioned in the Introduction.

The *theoretical* question addressed in research in program synthesis is the discovery and articulation of the principles (or appropriate programming languages) underlying the creation of software. The important *practical* question addressed is how to implement systems that embodies a particular knowledge and applying it to assist the programmer or end user.

The history shows us that there is freedom toward the use of more declarative and less procedural specification where appropriate. Another way of description is that the specification is closer to *what* than to the *how* end of the spectrum. The implementation is closer to the how end.

Methods for automated program synthesis vary. One approach is to view synthesis as a set of transformations that perform the steps of successive refinement of the specification into the implementation, performing data structure refinement and control structure refinement of the program into the final implementation. Other methods of program synthesis focus more on theorem proving, where an inference engine derives the steps of the program and proves the necessary facts along the way, e.g. the PROLOG program is such a set of declarations. In particular, a prover proves that an implementation exists, and the proof is made constructive, that is, it constructs the implementation that exists. We note, that both of the above mentioned formulations describe the same goal: to allow users focusing more on solving problems than on the details of implementation.

We have to be aware that programming cannot be fully automated since the computer must at least be told what to do. There is no way for a computer "to invent an idea". The automation may refer only to the way how "to execute an idea".

3 Automatic program construction before the object-oriented programming

Nowadays it is almost forgotten that before the object oriented approach, within the frame of the classical procedural programming, the automatic program construction was associated (e.g.) with the "construction of programs by examples" or with the "construction of loops".

In the 1950s, the term "automatic computing" referred to almost anything related to computing with a computer. The biggest problem of automatic coding systems was efficiency, or the lack of it. Human-computer interaction was very inefficient. This resulted in an atmosphere in which the idea of automatic coding was conceived as fundamentally wrong: "efficient programming was something that could not be automated" was an often-heard statement.

In the period of "structured programming tide" (1967-1977) the optimizers mattered because they free programmers from the need to deal with specific details to focus on larger issues. In this way developers are able to do more important things. One research stream was dedicated to the loop optimizers. Typical representative of the loop optimizer is the programming by example approach. This technology is based on the induction principle.

In the next section we outline the idea, which represents the automatic program construction efforts in the 1980th. In other words, how the automatic programming by example was perceived before the object orientation [5], [6], [7]. The research before the object-orientation was influenced in great extent by the results gained in the artificial intelligence research. Algorithm described below was discussed in detail by the author elsewhere [8].

The principle of programming by examples

There exist many areas when the demonstration is a suitable tool for automating tasks. For example, paths of robots represent linear plans and the task is to construct program; or the sequence of learning objects represent the progress of the student in the learning material and the task is to construct the navigation plan (learning by watching). The structures of the systems devoted to synthesis of programs by examples are similar also to the structure of linguistic pattern recognition systems.

In the next paragraphs we describe formally the idea of synthesis the loops from examples [9]. The aim of the synthesis is to construct a minimal final deterministic automaton with branches and loops, which are expressed as:

1. $I_1 \overset{c_1}{\rightarrow} I_2$
2. $I_2 \overset{c_1}{\rightarrow} I_3$

where I_1, I_2 and I_3 are the instructions of the trace and c_1 and c_2 represent the conditions for the execution of the respective instructions. In this model the program equals to the regular grammar:

- $(V_n,\ V_t,\ D,\ I_0)$

where

- V_n - is the set of program instructions (non terminal symbols),
- V_t - is set of conditions, which belong to the appropriate transitions between the program instructions (set of terminal symbols),
- D - is set of rules, which does not contain 2 or more rules with the same left side,
- I_0 - is the start non terminal symbol.

The algorithm for building the model is summarized as follows. Let the symbol P be a set of available instructions, which are necessary for constructing the example.

$$P = (I_1, I_2, \ldots, I_K) \tag{1}$$

We introduce notation $[I_j]$, for the set of equal instructions $I_j|1 \leq j \leq K$. We introduce, that

$$[I_j] = 1I_j, 2I_j, \ldots, X_jI_j \tag{2}$$

where the integers in front of I_j are called labels. Let the overall number of I_js in the model equal to $|[I_j]|$ and the $[I_j]^*$ is the actual number of $[I_j]$. The number of the total instructions c_2 in the model is:

$$L = \sum_{j=1}^{K} |[I_j]| \tag{3}$$

where the number of various types of instructions is K. Because the value of the L is varying during the synthesis, we introduce the L^* for the actual value of L. Then

$$L^* = \sum_{j=1}^{K} |[I_j]^*| \tag{4}$$

"Step"in the example is defined as a pair of (c_p, I_q). Different steps (l) may contain the same pairs, i.e. the same condition c_p and same instruction I_q. That is why we introduce the notion of N_l for the condition and O_l for the instruction in certain step l. The $u(l)$ will denote the label of the instruction $O(l)$. The principle of the program synthesis is in searching the value of $u(l)$, which will fulfill the following conditions:

1. The number of instructions L in the program is minimal and it is true that $K \leq L \leq M$, where M is the maximum number of instructions of the example.

2. If the M is the maximum number of instructions of the example, then during the synthesis it is necessary to assign a label $u(l)$ to every instruction $O(l)$ of the example and at the same time to achieve deterministic flow of control. I.e. for every step i where $i \leq l$, and $O(i-1) = O(l-1), u(i-1) = u(l-1)$ and $N(i) = N(l)$, then either

   - $O(l) = O(i)$ and in this case it is possible to provide merging, i.e. $u(l) = u(i)$ or
   - the above-mentioned conditions are not true and $O(l) = O(i)$, then new node has to be created, i.e. $u(l) \neq u(i)$ for the respective instructions in $O(i)$ and $O(l)$. This creation of the new node has to be done in order to secure the deterministic control of flow.

It is evident that when there does not exist a node in the model, which is merge-able with the given instruction in the example, then new node has to be created.

4 Obstacles to software reuse

The book H. Mili et. al. [12] defines the reusability as follows: *Software reuse is the process whereby an organization defines a set of systematic operating* *procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in its development activities.*

Software reusability is an attribute of software that facilitates its incorporation into new application programs. Reusable software shares many attributes in common with "good software" (i.e. transportability, maintainability, flexibility, understandability, usability and reliability).

The status and the future of the software reuse research is described exhaustly in the paper of W.B.Frakes [11]. The work of Z.Porkoláb [13] points to the relation between metaprogramming and reusability. Interesting fact (or friction) is that "reusability" is not usually a distinguished attribute of artifacts in other engineering disciplines. This induces the following question: "Why do we emphasize so intensively reusability in software engineering?"

If we follow the idea of W. Wang [10] *"The key reason is that software is a tangible form of mathematics that lends itself to being engineered.... This tangibility is both software's strength and Achilles heel."*, then it is clear that this "executability" feature is the driving force behind the "software engineering" activities (e.g. software testing).

It is also often argued that the reusable software assets are "information rich". What does it mean that "information rich"? In fact it means that: software assets represent written ideas and the "customization" of these representations within other context requires excessive mental effort. This idea is valid also for other kinds of representations, as software patterns and models.

The history shows that all contemporary techniques always contained some mechanism of reusability. For example: data encapsulation, information hiding, polymorphism, abstract data types, classes and methods, pipes and filters, inheritance, parametrization and generality, etc. All these techniques could help in certain implementations and domains. More detailed analysis on these issues is out of scope of this contribution.

5 Conclusion

We have discussed some selected ideas around the notion of automatic programming research. We tried focusing on the importance of raising students' awareness to sustainable ideas, which are beyond the "fashionable" ones. This paper could be a basis for formulating further questions in this direction. We expect that the formulation of adequate questions is the first step toward discovery of relevant knowledge in the emergent science of software engineering. We think that this approach could support more thorough understanding of other software engineering principles too.

Acknowledgments

This work was supported by project VEGA No. 1/0350/08 “Knowledge-Based Software Life Cycle and Architectures” and project VEGA No. 1/0175/08 “Behavioral categorical models for complex programme systems.”

References

[1] N. Wirth, Plea for Lean Software. *Computer*, **28**, 2 (1995) 64–68.

[2] L. McLaughlin, The Next Wave of Developer Power Tools, *IEEE Software*, **23**, 3 (2006) 91–93.

[3] D. McIlroy, Mass-Produced Software Components *Proceedings of the 1st International Conference on Software Engineering, Garmisch Partenkirchen, Germany*, pp. 88–98, 1968.

[4] D. King, C. Kimble, Uncovering the epistemological and ontological assumptions of software designers. *Paper presented at the conference Proceedings 9e Colloque de I”AIM, Evry, France*, 2004.

[5] J. S. Poulin, Technical opinion: reuse: been there, done that. *Comm. ACM* **42**, 5 (1999) 98–100.

[6] Z. Manna, R. J. Waldinger, Toward automatic program synthesis, *Comm. ACM* **14**, 3 (1971) 151–165.

[7] C. H. Smith, The Power of Pluralism for Automatic Program Synthesis. *Journal ACM* **29**, 4 (1982) 1144–1165.

[8] L. Samuelis, Programming by examples, *Technical University of Budapest,1990*, PhD thesis.

[9] A. W. Biermann, Automatic programming, In *Stuart C. Shapiro, editor, Encyclopedia of Artificial Intelligence.* John Wiley and Sons, January 1992.

[10] Wei-Lung Wang, Beware the Engineering Metaphor, *Comm. ACM*, **45**, 5 (2002) p. 29.

[11] W .B. Frakes, K.Kang, Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, **31**, 7 (2005) 529–535.

[12] H. Mili, A. Mili, S. Yacoub, E. Addy, *Reuse based software engineering (techniques, organization, and measurement)*, John Wiley & Sons, 2002.

[13] Z. Porkoláb, Debugging C++ Template Metaprograms, *Generative Programming and Component Engineering*, The ACM Digital Library pp. 255–264.

Received: October 13, 2008