Roots of Computing in Austria

Contributions of the IBM Vienna Laboratory and Changes of Paradigms and Priorities in Information Technology [1]

Kurt Walk
Vienna, Austria walk@via.at

Key words: word processing

It was in the late fifties when we ran one of the first application programs on the computer we just had finished at the Technical University of Vienna. We were a group of young people led by Heinz Zemanek, engaged in one of the early university computer projects of those times. The computer became known under the poetic name of 'Mailuefterl' and was the first one in Europe built entirely with transistor technology. Mailuefterl had fifty words of core store, 10.000 words of drum store, paper tape input and typewriter output, and an internal speed of 2.000 words per second [16] [04].

The program in question was written and run for a musician, a composer interested in the set of those twelve-tone sequences that satisfied certain constraints he had established. Mailuefterl, by the way, ran overnight to complete the job, despite clever optimization Peter Lucas, the programmer, had designed into the program.

The program was run only once, which makes this an extreme example of a computer usage pattern: one programmer transforming a class of problems out of a certain universe (music) into computer digestible form (the program), solving a particular case by computer, interpreting the results, handing them over to the one interested party (the composer).

Mailuefterl was used for a number of applications in similar patterns and also for a number of more durable ones. It was moved to IBM Vienna in 1961, and members of the University group then formed the kernel of a newly established IBM Science Group in Vienna, again under the leadership of Heinz Zemanek. This group of two dozen people developed into the IBM

[1] Supported by OEGIG, the Austrian Society for the History of Computing

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35609-9_29

K. Brunnstein et al. (eds.), *Human Choice and Computers*

Vienna Laboratory, which until a few years ago has served various missions in different IBM organizations, ranging from programming technology to advanced system development and program product development with roughly 120 people peak plus auxiliary work force. I was manager of the laboratory from 1976 to 1992 and I am glad now to give you my personal views on some of the highlights of our work.

Goals and achievements changed over the years. The stable factor, maybe, was the urge to remain current in the advances of at least certain sections of information technology, if not trying to be ahead of the main stream. Paradigm changes in information technology used to set new priorities.

1. PARADIGM CHANGES

The above mentioned application out of musical theory belongs to the **independent computing** paradigm featuring a human user, a problem world and a computer system as three separate domains. Solutions for classes of problems out of the problem world were formulated as programs and, characteristically, executions of a program for different specific cases were independent of one another. A programming language expressing the program (an assembler language in our early case) was the formal carrier of information from the human user to the computer. It was the only formal carrier. The transformation of a case out of the problem world to a program, and the interpretation of the results of the program run on a computer into a form meaningful for the problem world, was human effort and not grasped by any formalism. The programming language here was the sole mediator between humans, their problems, and the computer. This made programming languages and their compilers the central subjects of interest and research in (the software section of) information technology. Language design and language definition methodology were the key disciplines.

The vast expansion of application areas in the early sixties boosted the development of 'machine independent' languages, general purpose as well special languages intended to serve special fields of application. The decisive development, however, was the emergence of file and data base systems which preserved data in between the computer runs. **Data based computing** became the paradigm of expansion and success of the computing field.

A data base encodes statements about facts in a certain universe of concern. Facts and rules do model the that universe. In certain cases the computer not only contains statements about facts, but it embodies the facts: think of a stored bank account - changing it does not just change information

about a fact, it does change the fact. The problem universe of concern and the computer system partially overlap (which imposes a new dimension of reliability requirements on computer systems).

Data base systems had their own definition and access languages to which programming languages had to establish interfaces. But, alas, programming language development was earlier than data base development and so in most cases we just saw bridges from one to the other rather than integral language solutions.

The emerging every day life significance of computing added another dimension: the significance of the computer interface to the user. A business area normally is served by a centrally managed computer system, containing a data base, programmed transactions corresponding to standard business transactions, and work places of users. The **transaction processing** paradigm has to serve the operating requirements of human users who are not computer professionals. Human properties with respect to the digestible amount of information handled in a transaction, the adequacy of information layout for human comprehension, and acceptable response time did enter information technology. Design and development of the user interfaces of terminals and, later on, of workstations received attention.

The picture of the business professional initiating and handling transactions naturally extends to that of a team of people working together to handle complex problems. People may assume different roles within the group. Their work does depend on the work results of other people and the timing of their actions may depend on conditions arising during the work process. In general they will share common data as well as use their own private data. I call this the **workflow** paradigm. Systems have been developed the last several years which support these working scenarios. They offer facilities for defining workflows in workflow models, used for the control of the actual workflows, for binding appropriate services to the workflows, and for building user workplaces corresponding to the roles the users play. This paradigm includes users and their workplaces into the problem area of concern. The world of users, the problem universe, and the computer system overlap. The overall working system is one of asynchronously cooperating human users and computer services. Difficult questions do arise here: what are adequate workflow models for certain application areas, which type of interface and interaction is acceptable to users, how can a running workflow be changed to adapt to changed conditions, or recover from an unforeseen situation, how can workflow management cope with time and resource planning?

Paradigm changes have changed work content and work style of the Vienna laboratory over time. Certain areas, nevertheless, formed continuous threads of activity over many years. I would like to sketch some of them.

2. THE PROGRAMMING LANGUAGE THREAD

Programming language development started with machine and assembler languages, whose structure was fairly simple, programs consisting of equally formed statements whose meaning was determined by the status and the changes of the hardware components they referred to. (We, of course, had an assembler on our Mailuefterl machine.) Higher level languages then were aimed at referring to elements of a problem area rather than to the components of a computer. The situation of early language designers was more difficult than they themselves thought at that time. They did not have to start from nothing: there was the mathematical notion of an algorithm, and numerical mathematics was an established field. Many basic elements could be drawn from there: variables, arithmetic expressions, functions, iteration. Yet we know that variables, expressions, and functions in mathematics are not quite the same as variables, expressions, and functions in computing. In addition, running programs depend on certain properties of the real computer: the finite precision of number representation, the implemented order of expression evaluation, storage properties, the available input/output operations, exception handling, and others. To which extent should these properties be visible or controllable in the programming language? Languages were designed following different philosophies, ranging from: keep the language definition 'clean' and implementation properties simply open, to at least partially addressing implementation properties within the language. Were these questions ever resolved to satisfaction? I think not. Only more recently, the internet begins to enforce more fully defined languages with compatible implementations across systems.

We concentrated on a different aspect in Vienna. The question was how to define the meaning of a programming language, whether well-designed or not. The languages were announced as being machine independent, but the manuals failed to say on what these languages now depended, if it was not the real machine. Understanding language manuals relied on obvious analogies with arithmetics or real machine constructs, and on manually following and interpreting the program text. The question began to be recognized in the early sixties. The community was in search for well founded language definitions to serve as input to compiler writing, to investigate the correctness of programs, and to improve the conceptual understanding and teaching of programming languages. The feeling for the lack of these foundations was building up in Vienna in the course of writing a compiler for the ALGOL60 language, completed in 1961.

The syntax of languages was the first area for which solutions appeared, it even triggered the interest of mathematicians and a wealth of publications was the consequence. New concepts for the semantic description of

languages came manyfold. Various attempts were made to use well known explications of the notion of an algorithm as the basis. Certain language features could be mapped on the Lambda calculus, on Markoff chains or abstract automata, leading to natural correspondences in some cases, unsatisfactory ones in others. It was John McCarthy who first formulated the concept of an abstract machine, to replace the real machine as the definitional vehicle [11].

This concept was taken up by Peter Lucas and colleagues in Vienna and successfully applied to various examples. In 1964 the IFIP Working Conference on Language Description Languages, organized by Heinz Zemanek and members of the Vienna Laboratory, surveyed the various proposed approaches in that area, all still within the philosophy of 'independent computing'. For us the real challenge came with the emergence of the programming language PL/I in 1965, the most complex language seen so far, meant to encompass numerical as well as business applications and giving access to all the system capabilities available at that time (file systems, input/output, exception handling, tasking, etc.), clearly supporting the data based computing paradigm. The Vienna laboratory accepted the challenge and started the project of developing a complete, formal description of PL/I. We constructed an abstract machine which was characterized, like a real machine, by the set of states it can assume, and by the transitions from one state to the next. States were mathematical objects, which had components serving the definition of the various language aspects (stores, name scopes, evaluation stacks, etc.). Given a state, the state transition function defined the successor state (or in general: the set of possible successor states). Programs were represented in tree form according to an 'abstract syntax', which defined its meaningful parts, independent of the oddities of the concrete program representation as a character string. The program to be interpreted defined the initial state of the abstract machine, and the successive application of the transition function defined the computation steps until, hopefully, an end state is reached. The behavior of the abstract machine and, in particular, the end state defined the 'meaning' of the program [10].

The first version of the formal definition of PL/I (called ULD: Universal Language Definition) was finished by the end of 1966. Two more versions followed which improved readability and included new extensions of PL/I. Completing these definitions was a major engineering effort. We had to cooperate internationally with other IBM departments (Language Development and Control, Compiler Development, User Representatives) distributed over various locations (before the advent of the internet). This cooperation had immediate value leading to clarifications, changes and simplifications of the language. Technically, it was important to take the

freedom to design the underlying abstract mechanism in a way best suitable for the problem, which means not to be constrained by a preconceived formalism. The notation invented for defining the abstract syntax, states and the state transitions became known as the VDL (the Vienna Definition Language) [09].

The ULD project demonstrated that formal definition of a language the size of PL/I was actually possible and it gave insights into properties of PL/I that were difficult to grasp without a formal notation. To note, the main cause of difficulties and tediousness to cope with was the fact that the language existed first, before the universe, or better call it the abstract system, was defined it was supposed to address. In hindsight, it is the wrong order to create notation first, and then to make the attempt to clarify its meaning by inventing an appropriate abstract system. I want to underline this here because the same mistake is being made again and again until today, just see some of the many design and modeling languages invented in recent years.

VDL was not the final answer. A number of critical questions remained:

- If the behavior of the abstract machine defines the meaning of a program, the meaning to an extent depends on the specific design of the machine. Is it sensible to define an 'essential meaning' based on a subset of the states or of the computational steps?
- Is the definition adequate for attempts to formally prove general properties of the defined language? The principal answer is yes and practical feasibility was shown with smaller examples. Proofs, however, may become unsurmountably lengthy and tedious for a language the size of PL/I.
- Is the definition adequate for proving properties of individual programs written in the language defined? A similar answers applies.

These questions gave rise to other approaches. The approach often called 'denotational semantics' (as opposed to the 'operational semantics' of the VDL) was triggered be Dana Scott [03] and Christopher Strachey [12] and investigated and used in Vienna by Hans Bekic, Peter Lucas, later by Dines Bjoerner, Cliff Jones and colleagues. It avoids the concept of computational steps and associates meaning to a language term in the form of a mathematical object, where the meaning of a composite term in general is built up from the meanings of the components of the term. Denotational semantics offers elegant explications for certain language concepts (e.g. procedures), less elegant ones for others (e.g. goto's), and in many cases makes it easier to construct mathematical proofs. The definition of a PL/I subset was completed in Vienna using a notation built from elements of VDL but also influenced by denotational semantics [01]. It was done in preparation for a compiler project for a new machine architecture, so it was

critical to find the right style for the definition to serve multiple purposes: as communication vehicle among language designers and machine architects, as a basis for arguing formally about programs, and as the precise input to compiler design.

Another school of thought advocated the 'axiomatic approach'. R.W. Floyd [06] and C.A.R. Hoare [07] were the main proponents and important contributions came from E.W. Dijkstra [05]. The aim was to be able to talk about the effects of executing a program, or the component of a program, without setting up a mechanism for doing the execution. The building blocks of this scheme are sentences of the form „if P is true before the execution of a piece of a program then Q is true after the execution, provided the execution terminates", where P and Q are propositions about the state of the computation. This builds up a methodology for proving the correctness of programs. The proof theory then consists of a proof rule for each syntactic form of the language. This thinking also influenced the projects in Vienna.

After the PL/I project emphasis in Vienna shifted from language definition to advanced system development and to program development. Formal methods carried over to programming methodology.

To be mentioned along the programming language thread are several language compilers which were developed in Vienna over time, ranging from an experimental ALGOL 60 compiler to compilers for RPG II, for a subset PL/I, and for REXX, the IBM command language, for various systems and for commercial use. Correctness proofs for the design of certain central compiler features were performed especially for ALGOL 60 and for PL/I.

3. THE PROGRAM DEVELOPMENT THREAD

Programming methodology developed out of an art performed by individuals. A considerable skills base had developed already by the end of the fifties, primarily for applications in numerical mathematics and string manipulation, supporting the independent computing paradigm, but writing correct programs was not yet really a teachable discipline. Testing was considered the verification of program correctness by many.

The problem of correctness of programs was made a theme in Vienna early on. The thinking was: if a program is supposed to realize a mathematical function, and if the meaning of the program is formulated in mathematical terms, it must be possible to mathematically prove the correctness of the program by proving the equivalence of the two. Initial steps were taken to show the practical feasibility for non-trivial, yet small programs.

The application of these methods to larger programs soon hit limits of size and complexity. Also, whereas programming already was supported by tools like editors, compilers, and debuggers, this was not the case for the development of formal proofs. This is one of the reasons why attempting correctness proofs of larger programs was not a practical proposition. But, maybe, there were ways for writing correct programs in the first place? Given a formal specification of what a program is supposed to achieve, we could look for ways of transforming this specification into a form which is closer to the terms of the chosen programming language. These steps then could be repeated until a valid program is reached. This is the idea of stepwise refinement, where the correctness of each individual step should be verifiable.

Main application area in Vienna was the formal description of compiler concepts and the verification of their correctness (sometimes even more significantly: their incorrectness) with respect to the language definition, also proofs of the equivalence of compiler concepts. Significant practical results were achieved. The method, however, is not restricted to compilers. It was broadly applied and became a subject taught at universities [02], [08].

Appropriate notation was developed in Vienna for expressing program specifications at an abstract level and for arguing the correctness of development steps. The approach together with the specific notation became known as the Vienna Development Method (VDM). The Vienna laboratory did not work in isolation, VDM was extensively discussed in the scientific community, in particular also in the IFIP Working Groups 2.2 and 2.3.

Around mid of the seventies, further developments of this methodology moved out of Vienna, together with the key contributors Peter Lucas, who joined IBM Research in San Jose, Dines Bjoerner and Cliff Jones, who accepted positions as University professors. There is an active, international scientific community in this area. New developments and applications of formal methods are being discussed in annual conferences (International Symposia of VDM Europe, later called Formal Methods Europe) which bring together active researchers in the area of formal methods and representatives of the industry. Formal methods are a living subject and there is a tendency now that it is returning from academia to industry.

4. THE PROGRAM PRODUCT DEVELOPMENT THREAD

The development of programs for commercial use, in Vienna started in the seventies, added new dimensions to our understanding of the programming profession:

- Interfaces to system facilities and resources, key to data based and transaction computing, were not defined within the scope of programming languages. They needed separate investigation.
- Starting points of development more often than not were general requirements rather than formal specifications. Development of the specifications was the first phase in development, which in most cases is an iterative process. The specification method had to serve more than one purpose: communicating with controlling parties and prospective users, providing the basis for the planning of programming tasks and the communication among programmers, and being the formal input to programming development. This often led to adopting formal and informal methods that seemed best suitable just for the case at hand, no generally acceptable method for expressing requirements was developed.
- Project planning, setting of checkpoints and committing to deadlines, controlling and re-planning, was seen as an art more than as a scientific discipline and personal experience was a key success factor.
- Human aspects like communication in a project team and the motivation of team members are hardly less significant for success than ´ are technical education and experience.

We soon learned that ignoring any one of these dimensions would lead to disaster. An appropriate skills base could be built up and a series of products were developed by the Vienna laboratory. Besides compilers and utilities, there was a focus on the development of tools for the development and support of complex applications, specifically in two areas. Transaction computing was supported by products for the definition and maintenance of user interfaces on terminals and workstations, and workflow scenarios were the target of workflow products.

The goals of using workflow products in an organization are to ensure the reliable and repeatable execution of established business processes, and to improve the efficiency of the organization by improving the processes over time [13]. Workflow products can improve the workplaces of users, offering tools and data for use just at the right time, and they can be used to enforce the rules established for the organization. Business processes can be embodied in workflow models, which define the relative positioning of activities performed either by people or by computer services. Workflow models are automatically interpreted by the workflow product. They can also

be used early on in the development cycle for the analysis and structuring of complex applications. This leads to a 'traceable' development, which means that components of the final implementation can be traced back to the terms and requirements introduced in the analysis phase. This is a property the workflow approach shares with object-oriented development, which allows tracing back implementation components to the objects meaningful during analysis (and for the user). One of our results was that the workflow and the object-oriented approach complement each other and must not be seen as alternative, or even conflicting disciplines.[14].

Some of the Vienna products, not all of them, were commercially highly successful, and most of them were of outstanding quality. Leading edge quality even was reached by certain products being virtually defect-free. Still, the planning and control of complex development projects always remained a subject of concern. A project is a complex system of people playing various different roles, of development tools, computing resources and communication lines, and of development results in different states of completion. The system behaves in response to acting people and the performance of system services, guided by project plans and general development procedures.

A development project, therefore, is itself a workflow system. Actual projects are usually supported by development environments providing tools at the workplaces of developers, but project guidance is normally not automated. A first step in this direction is the formal modeling of projects. Systems of asynchronously cooperating objects and activity flow models have been used for modeling standard project situations. They are the starting point for investigating the automation of projects [15]. Workflow products and object directories combined are the suggested way to go for formally controlling development project processes.

Formal methods again will advance the art of programming.

5. REMARK

The Vienna laboratory is history. Paradigm changes in information technology have coined its proceedings. It has produced advanced technology and products, and also has shaped people who continued work in other places. There were also failures among the successes. Occasionally, projects were even too early for the general state of the art. For example, we had speech processing projects early on. It was too early and we returned to that subject only in the nineties, developing speech recognition products.

What is the next computing paradigm? Maybe we shall see us all more and more as passive users in all-encompassing systems, interacting with the

system just by being at a certain place, or by changing position? Technology is certainly around, but the increasing dependency of most aspects of life (and of life itself) on information technology will further increase the significance of a solid foundation and of rigorous methodologies.

REFERENCES:

[01] H. Bekic, D. Bjoerner, W. Henhapl, C.B. Jones and P. Lucas: A formal definition of a PL/I subset. Techn. Report 25.139, IBM Laboratory Vienna, 1974

[02] The Vienna Development Method: The Meta-Language. Eds. D. Bjoerner and C.B. Jones, Springer Verlag Lecture Notes in Computer Science, Nr. 61, 1978

[03] J.W.de Bakker and Dana Scott: A theory of programs. Manuscript notes for IBM Seminar, Vienna, 1969

[04] N.S.Bachmann: European Electronic Data Processing. Comm. ACM 2 (1959),No. 9

[05] E.W. Dijkstra: A Discipline of Programming. Prentice-Hall 1976

[06] R.W. Floyd: Assigning Meanings to Programs. In Proc. Symp. in Applied Mathematics, Vol 19: Mathematical Aspects of Computer Science. American Mathematical Society, 1967

[07] C.A.R. Hoare: An axiomatic basis for computer programming. Comm. ACM 12, October 1969

[08] C.B. Jones: Software Development: A Rigorous Approach. Prentice Hall International, 1980

[09] J.A.N. Lee: The Vienna Definition Language. SIGPLAN Symposium on Programming Language Definition, August 1960

[10] P. Lucas and K. Walk: On the Formal Description of PL/I. Annual Review in Automatic Programming, Vol.6, Part 3, Pergamon Press 1969

[11] J. McCarthy: A formal description of a subset of ALGOL. In T.B. Steel (Ed), Formal Language Description Languages for Computer Programming, North-Holland, 1966

[12] C. Strachey: Towards a formal semantics. In T.B. Steel (Ed), Formal Language Description Languages for Computer Programming, North Holland, 1966

[13] K. Walk: Workflow Management Concepts. A White Paper. IBM Laboratory Vienna, November 1993

[14] K. Walk: Object-Oriented Development of Workflow Applications. A Development Approach. IBM Laboratory Vienna, July 1994

[16] K. Walk and R. Messnarz: Object Oriented Modeling Strategies. Software Process 96 Conference and Exhibition, Brighton, 1996.

[14] H. Zemanek: „Mailuefterl", ein dezimaler Volltransistor-Rechenautomat. Elektrotechnik und Maschinenbau, Vol. 75, 1958