Why Use Evolving Algebras for Hardware and Software Engineering? *

Egon Börger

Università di Pisa, Dipartimento di Informatica, Corso Italia 40,
I-56125 Pisa, Italy

Abstract. In this paper I answer the question how evolving algebras can be used for the design and analysis of complex hardware and software systems. I present the salient features of this new method and illustrate them through several examples from my work on specification and verification of programming languages, compilers, protocols and architectures. The definition of a mathematical model for Hennessy and Patterson's RISC architecture DLX serves as a running example; this model is used in [24] to prove the correctness of instruction pipelining. I will point out the yet unexplored potential of the evolving algebra method for large-scale industrial applications.

Ich habe oft bemerkt, dass wir uns durch allzuvieles Symbolisieren
die Sprache für die Wirklichkeit untüchtig machen.
Christian Morgenstern[2]

It needs some courage to come after three decades of intensive research in the area of formal methods and to advocate yet another general method for software and hardware design. However the huge gap between much of academic theory and the prevailing software and hardware practice is still with us, as is a wide-spread scepticism about the industrial benefit of formal methods (see the discussion in the literature about what are the right methods and criteria [33, 68, 69, 47, 78, 31, 32]) for software engineering to become a "mature engineering discipline"[54]. I accept with pleasure the invitation to explain to this audience how the new evolving algebra approach contributes to bridging this gap. I will try to convince you that it offers a mathematically well founded and rigorous but nevertheless simple discipline practical and scalable to industrial applications.

The notion of *evolving algebra* has been discovered by Gurevich in an attempt to sharpen Turing's thesis by considerations from complexity theory (see [39]) where the notion has led to important new developments [6]. At that time I was trying to develop a mathematical model for the programming language Prolog; this led me to the idea to use the notion of evolving algebras for defining transparent and simple specifications of complex dynamic systems at various

* In: Miroslav BARTOSEK, Jan STAUDEK, Jiri WIEDERMANN (Eds.), SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics, Springer LNCS

2 Translation: I have often observed that by over-symbolizing we make the language inefficient to use in the real world.

levels of abstraction and to relate such specifications through hierarchies of provably correct stepwise refinements. I understood that using this notion one can develop a powerful and elegant specification method which a) supports the way programmers work and and b) provides a rigorous basis for it. Through numerous real-world case studies this idea has been confirmed.

I will illustrate the main features of this new approach through examples from evolving algebra specifications and verifications of real-life programming languages, compilers, protocols and architectures (see [11] for an annotated bibliography complete up to 1994). The running technical example is an abstract mathematical definition of the well known RISC architecture DLX [49]. For DLX the notion of sequential evolving algebra suffices; note however that in [41] this notion has been extended to that of distributed evolving algebras which turned out to be natural and powerful for modeling distributed systems. I start from scratch without presupposing any knowledge of DLX. It is not the worst argument for a specification method that for this architecture one can define a clear mathematical model which is easy to manipulate; the model has been used in [24] in a correctness proof of current instruction pipelining techniques.

The discussion leads also to the question what constitutes a *proof* and what is the role of *ground models* when we *apply* our mathematical notions and methods to the physical and technical world.

More precisely I will explain in sections 1–9 the following 9 features of the evolving algebra approach to the specification and verification of complex computer systems:

1. *freedom of abstraction* by which evolving algebras offer hierarchical structuring combined with systematic use of stepwise refinement and by which they support the software life-cycle phases from initial specifications to executable code,
2. powerful but simple mechanism for *information hiding* and defining *precise interfaces* which makes evolving algebra models easily adaptable,
3. *locality principle for dynamics*,
4. *separating specifications from verifications*,
5. *avoiding the "formal system straitjacket"*,
6. satisfactory links to application domains by *appropriate ground models*,
7. *support of abstract operational views*,
8. *scalability* to large complex systems including hardware/software co-design,
9. *easy learning by the practitioner* without presupposing any previous theoretical training, although the method has a *rigorous mathematical foundation*.

Other formal methods (VDM, Z, RAISE, B, … [3]) share some of these features, but to my knowledge only evolving algebras combine them all.

[3] This is not the place for the interesting task to compare evolving algebras to other formal methods in the literature.

1 Freedom of Abstraction

It is well known that general abstraction principles are needed to cope with the complexity of large systems. Whereas the algebraic specification theory [79] shows a way to deal with abstract data types and the action semantics approach [64] proposes a scheme for constructing complex operations out of basic components, evolving algebras offer the possibility to choose both, the data and the basic actions, at any level of abstraction and independently of each other. The way this is done is simple and corresponds to common practice in systems engineering: when specifying a software or hardware system one has to define its basic *objects* and the elementary *operations* which the system uses for its actions (dynamical behaviour). In other words one has to define the basic domains and functions of a system. This leads in a natural way to the mathematical notion of *structures* as formalization of system *states*, as I am going to explain now.

1.1 Universes

Each system S deals with certain basic *objects* which might be classified into different categories. This is reflected in an evolving algebra model of S by corresponding *sets* (also called universes or domains), one for each category of objects. These sets can be completely abstract—this is the case if no restriction is imposed on the corresponding category of objects. In case that the objects are assumed to have certain properties or to be in certain relations with other objects, we formalize these properties and relations by corresponding conditions (integrity constraints) which the objects in those sets are required to satisfy. The evolving algebra approach accepts any precise formulation of such conditions, in whatever language or framework they are given. The domains might also be equipped with certain functions providing some basic structure which can be used in the operations to be performed by the system.

In the remaining part of this subsection I illustrate this data abstraction principle by a discussion of basic DLX domains. I will try to show that the evolving algebra approach allows one to make well known ideas rigorous following simple patterns of reasoning every programmer is familiar with.

DLX is a general purpose register machine,.i.e. the operands of the basic operations are stored internally in CPU registers. Thus registers are basic objects of the architecture and are formalized as elements of a set REG. The content of registers is represented abstractly by a function $reg : REG \rightarrow WORD$ where $WORD$ is not furthermore specified at this stage but is intended to be implemented by a set of words. One use of registers in DLX is to contain addresses for memory access. This can be reflected abstractly by a set $ADDR$ satisfying $ADDR \subset WORD$ and coming with a function $mem : ADDR \rightarrow WORD$ which yields the content of the memory at the given address. We will see below that various interesting features of DLX can be described appropriately and succinctly on the basis of such an abstract notion of memory.

DLX is a load–store machine, i.e. the operands of ALU–instructions are not memory addresses but are taken from registers or from the instruction itself.

This design decision for the location of operands can be reflected by an abstract set $INSTR$ of instructions on which functions are defined which provide the operands. The design decision for DLX that no instruction can have more than two operands and that some instructions have a so called immediate value as (part of) an operand is reflected by the fact that we can formalize by three abstract functions how the machine gets its operands from an instruction, namely $fst\_op, scd\_op : INSTR \rightarrow REG$ and $ival : INSTR \rightarrow VALUE$. These functions are partial because the R–type instructions have two register operands whereas the I–type instructions take only one operand from a register and have the other one directly encoded into the instruction as immediate value and the J–type instructions (J for jumps) have only an immediate operand. We can use once more an abstract function $iop : INSTR \rightarrow BOOL$ to formalize the mode of addressing operands without imposing any details for the instruction format. In accordance with this view we also use an abstract function

$$opcode : INSTR \rightarrow ALU\_SET \cup TRANSFER \cup CONTROL$$

to split the instructions into three disjoint sets which have the obvious intended interpretation as set of arithmetical and logical, data transfer or jump instructions. At a certain abstraction level we will define these sets explicitly following [49, p.165]. For example ALU_SET is split into ALU and SET where ALU contains the usual arithmetical, logical and shift instructions and SET contains the usual operations for zero-test and $\leq, \geq$ etc. tests (in R–type and in I–type form), i.e.

$$ALU = \{ADD, SUB, \ldots, AND, OR, \ldots, SLL, SRL, \ldots, ADDI, \ldots\}.$$

Similarly TRANSFER is split into sets LOAD, STORE, INTERRUPT and CONTROL into JUMP and BRANCH. The principles of pipelining in DLX can be described without further specification of the instruction format. This means that we also do not formulate here any conditions which assure that no instruction needs both to calculate an address and to perform an operation on data. This feature will become apparent from the formalization of the instruction interpretation below; it is this feature which allows us in the pipelined version of the architecture to combine the execution step proper with the effective address calculation for memory access and jump instructions.

The above mentioned universes are static—i.e. they do not change; in general, the evolving algebra framework permits universes also to be dynamic. Since the main intention of the concept of evolving algebras is to reflect the dynamical system behaviour in a direct and simple way, there is a construct for growing of universes, namely **extend** A **by** $x_1, \ldots, x_n$ **with** — **endextend** where '—' is used to define certain properties or functions for (some of) the new objects x_i of the universe A. We use it in our Prolog model [27] which reflects the underlying resolution tree. Upon calling a user-defined activator the system creates as many new children nodes to start alternative computations as there are candidate clauses in the procedure definition of the activator in the user's database db. The child relation is needed for the backtracking behaviour of the system and is formalized using a function *father*. To each new node t_i the corresponding clause occurrence c_i in the program *db* is associated using a function *cll* and the current node records the candidates to be selected as possible alternatives to continue the current computation. This is formalized by the following Call Rule of Prolog:

$$
\begin{array}{l}
\textbf{if} \, \mathit{is\_user\_defined}(\mathit{act}) \wedge \mathit{mode} = \mathit{Call} \\
\textbf{then extend } \mathit{NODE} \textbf{ by } t_1, \ldots, t_n \textbf{ with} \\
\qquad \mathit{father}(t_i) := \mathit{currnode} \\
\qquad \mathit{cands}(\mathit{currnode}) := [\, t_1, \ldots, t_n \,] \\
\qquad \mathit{cll}(t_i) := c_i \\
\quad \textbf{endextend} \\
\quad \mathit{mode} := \mathit{Select} \\
\textbf{where} \, [\, c_1, \ldots, c_n \,] = \mathit{procdef}(\mathit{act}, \mathit{db})
\end{array}
$$

By the way domain extension and deletion of elements can be reduced to function updates [41].

1.2 Dynamic Functions

Once it has become clear what are the basic objects of a system S, one has to think about what are the elementary operations which are performed on those objects in S. Typically, a basic operation consists of setting a certain value, given the values of certain parameters. The most general framework of such operations is the following *function update*:

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n–ary function and $t_1, \ldots, t_n$ represent the parameters at which the value of the function is set to t. Evolving algebras allow function updates with arbitrary functions f and expressions t_i, t of any complexity or level of abstraction. Functions whose values can change are called *dynamic* in contrast to *static* functions which do not change.

Function updates provide the basic notion of *destructive assignment at any level of abstraction.* The above Call Rule for Prolog updates the dynamic functions *father*, *cands*, *cll*, *currnode*, *mode*. The last two functions have arity 0 and thus correspond to (global) variables in programming which can be updated dynamically and at different moments can assume different values. Also the typical fetch–execution mechanism in an architecture can be described in terms of two 0–ary functions, say $IR, PC \in REG$ for the instruction register (containing the currently executed instruction) and the program counter (containing the address of the next instruction to be fetched). The updating of the program counter is expressed by the function update $reg(PC) := next(reg(PC))$ where the function $next : INSTR\_ADDR \rightarrow INSTR\_ADDR \subset ADDR$ determines for a given instruction address (in PC) the next instruction address (to be stored in PC). The fetching of the next instruction to be executed is described by the function update $reg(IR) := mem(reg(PC))$. Clearly we rely here upon the obvious integrity constraint that the range of the restriction of mem to INSTR_ADDR is a subset of INSTR.

Function updates are the mechanism by which the dynamics of arbitrary systems can be described in an explicit way. In accordance with usual practice the execution of updates in evolving algebras can be conditioned by guards, giving rise to *transition rules* of the form **if** *Cond* **then** *Updates*. *Cond* is an arbitrary boolean valued expression (first–order logic formula) and *Updates* a finite set of updates. If *Cond* is true the rule can be executed by simultaneously executing each update in the set *Updates*. (The simultaneous execution of more than one update helps to avoid an explicit description of intermediate storage, see for example the updates $a := b, b := a$.)

The above Call Rule for Prolog is a typical example of an evolving algebra transition rule. This type of rules suffices to describe the sequential control of DLX following elementary ideas familiar to every programmer; see section 1.4.

1.3 States as Static Algebras

To speak about a system means to talk about its objects in terms of functions and relations defined on them. Domains, functions, and relations constitute what in mathematics is called a *structure*. Structures without relations are traditionally called *algebras*. Since relations (and in particular sets) can be represented by their characteristic functions, for simplicity we deal only with algebras.

A sequential *evolving algebra* can be defined as a finite set of transition rules **if** *Cond* **then** *Updates*. The effect of a transition rule R when applied to an algebra $\mathcal{A}$ is to produce another algebra $\mathcal{A}'$ which differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by the rule R.

The *consistency* of updates is the responsibility of the programmer who may use special tools for the purpose. For the simple evolving algebra model for the sequential control of DLX defined in the next subsection it is obvious from the form of the rules that they are consistent.

Note that no rule changes the type of the functions; only the incarnation (the concrete interpretation) of a function changes by changing some of its values. We speak therefore of algebras also as *static* algebras, to distinguish them from evolving algebras. Evolving algebras are transition systems which transform static algebras.

Thus, the abstraction principle which is built into the notion of evolving algebra can be summarized as follows: (static) algebras as "states" and guarded destructive assignments for abstract functions as basic dynamic operations. This is the most general notion of state and of dynamic changes of states modern mathematics offer. As a consequence evolving algebras are the most general notion of a (discrete) dynamic system. A priori no restriction is imposed on the abstraction level where one might want to place an evolving algebra description of a system.

This freedom explains the success of the simple and transparent evolving algebra models for the semantics and the implementation of numerous complex programming languages like Prolog [7, 8, 27, 28], C [43], VHDL [20, 21], Occam [17, 16], for protocols [22, 55], architectures [14, 13, 24], real–time algorithms

[45, 46], etc. which have been developed in a relatively short time by a relatively small number of persons who used only evolving algebras as tool. It is also the feature of evolving algebras which assure *extensibility* and *reusability*; see the ease with which the Prolog model in [7, 8, 27] could be modified to provide models for various well known extensions of Prolog by parallelism [25], constraints [30, 29], types [2], functional [23] or object–oriented features (see [9] for a detailed survey).

The importance of the freedom of abstraction is also confirmed by a common experience in the design of algorithms. Namely, the need to model phenomena of the real world, which are given a priori, leads the designer of programs to use 'abstract structures', as has been well expressed a long time ago by N. Wirth: *"... Data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages."* [80, p.10]

The reciprocal dependency of algorithms and data structures makes it important for the designer not to be hindered by inappropriate restrictions of the framework; in Wirth's words: *"It is clear that decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often depend strongly on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined."* [80, p.9]

By their power of abstraction evolving algebras offer the freedom the designer needs to 'tailor' his models to the given level of abstraction and to express his ideas without introducing any extraneous formal overhead. This allows one to build an arbitrarily complex system as a hierarchy of appropriate simpler evolving algebras , making systematic use of stepwise refinement. As a consequence the evolving algebra method supports all the software life-cycle phases from initial specifications to executable code; see for example [3, 28, 16].

1.4 The Sequential Control for DLX

The sequential control of DLX is formalized as in Prolog's Call rule by the values of *mode* for the different execution phases. In mode OPERAND the operands of the previously fetched instruction are decoded by putting them into the two register file exits (0–ary functions) A, B from where they are taken as input by the ALU. The next value of *mode* is determined by the result *opcode*(IR) of decoding the operation code.

For ALU or SET instructions the ALU computes the value of the function $opcode(IR)'$ for the given operands and outputs it to the register file entry C from where this result is written in mode WRITE_BACK into the corresponding destination register *dest(IR)* in the register file.[4]

For LOAD or STORE instructions first—i.e. in mode MEM_ADDR—the Memory Address Register MAR is used for the result of the MEM_ADDR calculation (which involves the immediate value and the value in A); for STORE

[4] For the sequential model of DLX an additional register TEMP appears in [49] which temporarily stores the right second operand (namely $fst\_op(IR)$ or $ival(IR)$, depending on the type of the instruction); this call for an intermediate mode ALU'.

instructions also the value to be transfered to the memory is passed from B to the Memory Data Register MDR. Then—i.e. in mode MEM_ACC—the memory is accessed for the transfer of data between MDR and the calculated memory position. In case of LOADing the data are then—i.e. in mode WRITE_BACK—written from MDR into the destination register in the register file.[5]

The special interrupt address register IAR is accessed only by instructions in INTERRUPT = {MOVS2I, MOVI2S} (for moving the interrupt information from or to IAR) or by the system jump TRAP (for moving PC into IAR).

The set JUMP splits into PLAINJ, LINKJ and {TRAP}. The characteristic difference of LINKJ instructions is that they record the current value of PC in the destination register.

These explanations should suffice for an understanding of the following evolving algebra rules which formalize the diagrams in [49] for the sequential control of DLX; the model is complete except for the floating point related DLX instructions which are left out for reasons of space.

For notational succinctness the function *reg* is systematically suppressed and the abbreviation $\alpha \leftarrow \beta \equiv \alpha' := \beta'$ is used where γ' is obtained from γ by substituting $reg(R)$ for R. For the same reason the standard argument IR for the static decoding functions is dropped and an abbreviation *new_mode* is used with AS standing for $ALU \cup SET$, M for $LOAD \cup STORE$, I for INTERRUPT, J for $JUMP \cup BRANCH$.

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{FETCH}} \\
\textbf{then } IR \leftarrow mem\ (PC) \\
\qquad PC \leftarrow next\ (PC) \\
\qquad mode := OPERAND
\end{array}$$

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{OPERAND}} \\
\textbf{then } A \leftarrow fst_op \\
\qquad B \leftarrow scd_op \\
\qquad mode := new_mode
\end{array}$$

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{ALU}} \\
\textbf{then if } iop(opcode) \\
\qquad \textbf{then } TEMP \leftarrow ival \\
\qquad \textbf{else } TEMP \leftarrow B \\
\qquad mode := ALU'
\end{array}$$

$$new\_mode = \begin{cases} ALU & \textbf{if } opcode \in AS \\ MEM\_ADDR & \textbf{if } opcode \in M \\ IAR & \textbf{if } opcode \in I \\ JUMPS & \textbf{if } opcode \in J \end{cases}$$

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{ALU'}} \\
\textbf{then } C \leftarrow opcode'(A, TEMP) \\
\qquad mode := WRITE_BACK
\end{array}$$

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{WRITE_BACK}} \\
\textbf{then } dest \leftarrow C \\
\qquad mode := FETCH
\end{array}$$

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{MEM_ADDR}} \\
\textbf{then } MAR \leftarrow A + ival \\
\qquad \textbf{if } opcode \in STORE \\
\qquad\quad \textbf{then } mode := pass_B_to_MDR \\
\qquad\quad \textbf{else } mode := MEM_ACC
\end{array}$$

$$\begin{array}{l}
\textbf{if } mode = \boxed{\text{pass_B_to_MDR}} \\
\textbf{then } MDR \leftarrow B \\
\qquad mode := MEM_ACC
\end{array}$$

[5] Note that in DLX it is supposed that depending on the operation code only a subword of what has been loaded from the memory to MDR is copied into the destination register; this calls for an intermediate mode SUBWORD.

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \boxed{\text{MEM\_ACC}} \\
\quad \wedge\ \mathit{opcode} \in \mathit{STORE} \\
\textbf{then } \mathit{mem}(MAR) \leftarrow MDR \\
\qquad \mathit{mode} := FETCH
\end{array}
$$

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \boxed{\text{MEM\_ACC}} \\
\quad \wedge\ \mathit{opcode} \in \mathit{LOAD} \\
\textbf{then } MDR \leftarrow \mathit{mem}(MAR) \\
\qquad \mathit{mode} := SUBWORD
\end{array}
$$

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \boxed{\text{SUBWORD}} \\
\textbf{then } C \leftarrow \mathit{opcode}'(MDR) \\
\qquad \mathit{mode} := WRITE\_BACK
\end{array}
$$

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \boxed{\text{IAR}} \\
\textbf{then if } \mathit{opcode} = MOVS2I \\
\qquad \textbf{then } C \leftarrow IAR \\
\qquad\qquad \mathit{mode} := WRITE\_BACK \\
\qquad \textbf{if } \mathit{opcode} = MOVI2S \\
\qquad \textbf{then } IAR \leftarrow A \\
\qquad\qquad \mathit{mode} := FETCH
\end{array}
$$

$$
\begin{array}{l}
\textbf{if } \mathit{opcode} \in \boxed{\text{BRANCH}} \\
\quad \& \mathit{mode} = JUMPS \\
\textbf{then if } \mathit{opcode}'(A) \\
\qquad \textbf{then } PC \leftarrow PC + \mathit{ival} \\
\qquad \mathit{mode} := FETCH
\end{array}
$$

$$
\begin{array}{l}
\textbf{if } \mathit{mode} = \boxed{\text{JUMPS}} \\
\textbf{then if } \mathit{opcode} = TRAP \textbf{ then } IAR \leftarrow PC \\
\qquad\qquad\qquad\qquad\qquad\qquad PC \leftarrow \mathit{ival} \\
\qquad \textbf{if } \mathit{opcode} \in PLAINJ \cup LINKJ \\
\qquad \textbf{then if } \mathit{iop}(\mathit{opcode}) \textbf{ then } PC \leftarrow \mathit{ival} + PC \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } PC \leftarrow A \\
\qquad \textbf{if } \mathit{opcode} \in \{TRAP\} \cup PLAINJ \textbf{ then } \mathit{mode} := FETCH \\
\qquad \textbf{if } \mathit{opcode} \in LINKJ \textbf{ then } C \leftarrow PC \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mode} := WRITE\_BACK
\end{array}
$$

2 Information Hiding and Interfaces

Information hiding, introduced by D. Parnas [67], calls for modular structuring of systems. In a practical specification method, information hiding has to go hand in hand with a good discipline to handle interfaces. The evolving algebra approach offers both in a most general way through the concepts of *oracle functions* and *externally alterable functions*.

Let f be a function of an evolving algebra $\mathcal{A}$. If f has no updates of the form $f(t_1, \ldots, t_n) := t$ in any transition rule of $\mathcal{A}$, it is called an *oracle function* of $\mathcal{A}$; if f does appear in an update $f(t_1, \ldots, t_n) := t$ of a transition rule of $\mathcal{A}$ it is called *internally updatable* or internally alterable. Dynamic functions whose values can be affected by the environment are called *externally alterable*.

Oracle functions are completely determined by the environment. They can be static or dynamic; if they are dynamic they are externally but not internally alterable. For the description of distributed systems it is convenient that in the evolving algebra framework one can speak about functions which are both externally and internally alterable; for such functions it depends on the system which discipline is imposed to avoid conflicts between external and internal changes of the function.

Usually the rules of $\mathcal{A}$ give some crucial information on the dynamical behaviour of internally updatable functions f of $\mathcal{A}$. In contrast the rules of $\mathcal{A}$ give no information on how an oracle function f of $\mathcal{A}$ operates; such a function cannot be modified ('written') by $\mathcal{A}$, but it can be used ('read') in the rules of $\mathcal{A}$ to determine arguments at which an internally updatable function is changed dynamically or to determine the new value in such updates. Oracle functions and externally alterable functions f are used to represent influences of the environment in which the given evolving algebra is intended to work. It is the task of the system designer to provide the information on f which he wants the programmer to know and to use. In the evolving algebra approach this interface information can range from nothing at all—this is the case of a function for which only the number and the types of its arguments and values are known—to a full specification by some axioms or by a set of equations or by another evolving algebra (module), etc. Note that due to the abstraction principle explained in the previous section the evolving algebra approach imposes no restriction at all on the choice of externally alterable or oracle functions and the way they are described. The use of evolving algebras does not trivialize the difficult task of "designer control of the distribution of information" ([67]:p. 344), but at least it does not hinder this task by extraneous overhead of formalities and offers a flexible and open framework to guarantee information hiding and the definition of precise abstract interfaces.

As a consequence, the evolving algebra approach helps to ensure that programs, once developed, can be extended, maintained and reused as components of larger systems in a systematic and reliable way. Also it integrates well into existing development environments without requiring a complete revision of the latter. Another pragmatically important feature in this context is that through skilful introduction of the appropriate abstractions high–level evolving algebra specifications allow one often to produce precise definitions of modules which are considerably shorter than implementations of those modules in a programming language.

In the following subsection I explain some examples showing the power of abstraction which is offered through the introduction of externally alterable or oracle functions in evolving algebra descriptions of complex systems.

Examples of Externally Alterable and Oracle Functions. The code related functions *fst_op*, *scd_op*, *iop*, *ival*, *opcode* and *next* in DLX are static, determined by an initialization (not given here) which contains the given program to be executed. The four simple rules [27] which define the complete behaviour of Prolog for user–defined predicates make crucial use of two oracle functions *procdef* and *unify*.

The function *procdef* is supposed to provide for given literal l and program db the clauses in db which are relevant for l, in the order in which they have to be applied. The whole backtracking behaviour of Prolog (including optimizations like determinacy detection) can be described on the basis of this abstract function *procdef*. If one considers Prolog without program modifying operations like *assert, retract*, then *procdef* is a static oracle function. If one wants to model also Prolog's program modification features then *procdef* becomes a dynamic and internally updatable function (see [15, 26]). Through the refinement process by which the Prolog model of [27] is linked in a provably correct way to the WAM implementation model in [28] *procdef* receives an explicit definition.

unify is supposed in [7, 8, 27] to provide for each pair of literals either a unifying substitution or the information that there is no such unification. The function describes the abstract behaviour of unification without being bound to any concrete unification algorithm. It also hides from the programmer the details about the representation of terms which appear in the refined WAM models of [28]. As a result the abstract PROLOG model of [27] and its refinement to the WAM model of [28] could easily be extended to constraint logic programming languages with or without types where unifiability appears as a particular case of constraints (see the evolving algebra definitions of PROLOG III [30], Protos–L [2], and CLP(R) [29]).

The oracle function *find-catcher* for which a recursive definition is given in [27] leads to a concise formalization of the error–handling predicates *catch* and *throw* of Prolog.

In the evolving algebra model [20, 21] for the IEEE VHDL Standard we have obtained a simple and uniform rule set for signal assignments by introducing for the inertial delay an oracle function *reject* for which we give a natural and easily understandable recursive definition. Similarly, a transparent description is obtained for the propagation of signal values by introducing oracle functions for the so–called driving and effective values; the former is determined by a recursion on the signal sources, the latter by a recursion on port association elements from ports to signals. In both cases the recursive definitions replace rather complex algorithmic characterizations in the VHDL'93 language reference manual [56].

In the abstract evolving algebra models of Occam (see [17]) which are the starting point for the correctness proof of a compilation scheme into Transputer instructions in [16] we have taken great advantage of the usual flowchart layout of programs; we define it by oracle functions which in the later refinement steps are replaced by recursive definitions of the compiling function. Considerable simplifications for both the specifications and the proofs have also been obtained there by leaving the evaluation and compilation of expressions and the implementation of values abstract, realized by appropriately restricted oracle functions.

For Lamport's mutual exclusion protocol, known as the Bakery Algorithm, a considerable simplification of the correctness proofs in the literature has been achieved in [22] by introducing two externally alterable functions, namely *Ticket* and *Go*, on which three natural conditions and an induction principle are imposed which imply the correctness of the protocol.

The widely used parallel virtual machine PVM realizes a distributed computation model which is characterized by the *reactive behaviour* of concurrently operating PVM daemon processes, each residing on one of several host computers. The daemons are triggered by the environment; they carry out the PVM instructions of the local tasks they have to manage and interact with each other through asynchronous message–passing communication. No daemon can influence when, from where and which request or message will reach him, rather he has to wait for the next such event to come whenever he is idle. We have modelled this intuition faithfully by introducing an externally alterable oracle function *event* which for a given daemon might yield a PVM instruction or a message as value. If $event(pvmd)$ is defined and has the value $instr/mssg$, then the daemon *pvmd* is going to execute/read $instr/mssg$. This is formalized in our PVM model [18, 19] by a rule of form

$$\textbf{if } event(pvmd) = instr/mssg \textbf{ then } execute\_instr/read\_mssg$$

for each individual PVM instruction *instr* or PVM message *mssg*, where $execute\_instr/read\_mssg$ represents the corresponding updates. An integrity constraint on the function *event* is that a defined value of $event(pvmd)$ remains stable until the PVM daemon pvmd has evaluated the function. However, we assume 'destructive reading' such that $event(pvmd)$ is reset to *undef* or indicates the next event as soon as the pvmd has read the current value. The dynamic oracle function *event* thus directly reflects the way in which tasks interact with their local PVM daemon when they want PVM routines to be invoked. Using the dynamic oracle function *event* we abstract from the specific way how the daemon's walk through his sequence of instructions/messages is determined by the activities of his tasks.

In all these cases the externally alterable or static functions allowed us to define a precise interface with respect to which the model under discussion works in a simple and transparent way. If for such an abstract model one wants to prove general properties about the behaviour of the system where externally updatable or oracle functions play a role, one has to state and assume the properties which are used. In order to guarantee an unchanged interface behaviour these properties have to be proved to be satisfied when those functions are defined explicitly or implemented in later refinement steps or modified by changing requirements.

The most general *concept of modularity* which is present through the notions of externally alterable and oracle functions is deliberately kept open in the definition of evolving algebras. The resulting flexibility in using and dealing with different module structures is an advantage for real–life specification endeavors. Nothing prevents us from restricting this notion to specific and even syntactic concepts of compositionality where the need arises; an example where it turned out to be useful to stick to a simple and well known automaton–theoretic concept of composition of evolving algebras through sequencing, juxtaposition, and feedback can be found in [13].

3 Locality Principle for Dynamics

It is typical for large systems that their overall behaviour is determined by the actions of their components, i.e. by *local* changes. Even for the dynamic performance of large *sequential* systems it is characteristic that at each moment and in a given context only a few things do change whereas the rest remains unchanged.

The complexity of the behaviour of a large system is due to the overall effect of a lot of small local changes. Often descriptions which build upon the locality phenomenon are considerably simpler and more natural than global descriptions which try not to refer to single computation steps. Exploiting a locality property can lead often to a modular design. In contrast to a widely held view not only the description of the system behaviour, but also the mathematical reasoning about it can become considerably simpler if the locality principle is used. It allows one to concentrate on the parameters which *do* change and not to have to worry about the other state components which *do not* change under the considered transition.

The evolving algebra method allows one to reflect this characteristic interplay between global and local dynamic system behaviour in a direct and faithful way by viewing (global) system states as static algebras and by providing the possibility to express local updates $f(t_1, \ldots, t_n) := t$ in a uniform way *at any level of abstraction.* We have used the locality principle with advantage for modelling parallel or distributed systems. See for example the succinct formalization given in [19] for the PVM message–passing interface. Another example is the parallel version DLX^p of DLX which we have defined as starting point for several further refinement steps leading to the pipelined version of DLX and a mathematical proof of its correctness [24].

3.1 The Basic Parallel Control for DLX with Pipelining

The refinement of DLX to a parallel model DLX^p in this section is taken from [24]. It resolves structural pipelining hazards and serves as link between DLX and its further refinements to the fully pipelined model in which also data and control hazards are resolved by the architecture.

The intuitive idea of pipelining is simple. The execution of an instruction is done in *DLX* in five stages, namely Instruction Fetch (IF), Instruction Decode which includes reading the operands (ID), EXecution proper for ALU operations which include (data or branch) address calculations using the ALU (EX), MEMory access (MEM) and Writing the computed result Back into the final registerfile destination (WB). Since at each stage different actions are taken, ideally one can pipeline *DLX* by letting the processor execute during each clock cycle simultaneously one different *pipe stage* for each of five instructions. Roughly this can be described by eliminating the sequential control and by replacing where necessary the *mode* guards by *opcode(I)* guards which correspond to the pipe stage of the instruction I in question. The resulting rule system should then be interpreted under the *lock-step parallelism* semantics of evolving algebras; under this semantics at each moment each rule whose guard is true is applied.

Special care has to be taken however to avoid conflicts resulting from dependencies between instructions which can occur when some of their execution stages overlap. Structural hazards arise when during one clock cycle two instructions compete for resources, each functional architectural unit of *DLX* being available at each step only once. For example fetching a new instruction on each clock cycle would create a *mem* access conflict with load/store instructions. This can be avoided by increasing the memory bandwidth. We introduce therefore an additional memory access function mem_{instr} which is used only for fetching instructions and which is supposed to be a subfunction of *mem*; in this way we abstract from any particular implementation feature related to using separate instruction and data caches.

Similarly one avoids to use the ALU for incrementing the program counter *PC* by providing a separate PC-incrementer. This is reflected by the constraint that our abstract function *next* has to be implemented without using the ALU. Note that the ALU is also not needed for the zero test in BRANCH–instructions because this test can be done using the standard exit of registers.

Some of the values which appear during the execution of an instruction at a certain pipe stage are needed also at later pipe stages and therefore should not be overwritten by the corresponding values of a subsequent instruction occurring in the pipeline. An example is the immediate value *ival(IR)* which is used by jump instructions in their EX stage. For reasons of simplicity we abstract from possible optimizations and provide three additional 0–ary functions (latches) *IR1, IR2, IR3* to keep full copies of a fetched instruction through the three pipe stages *EX, MEM, WB*. Similarly two latches *PC1* and *C1* are used to save the values of *PC* and *C* respectively for one pipe stage; *PC1* provides at pipe stage *EX* of an instruction *I* a copy of the value of *PC* after the FETCH stage of *I* (serving in case *I* is a jump instruction the execution of which triggers a transfer or an update of that *PC*–value); *C1* provides at pipe stage *WB* of an instruction *I* a copy of the ALU output value *C* computed in the pipe stage *EX* of *I* (which is the case for instructions with *ALU/SET*–operations, for *LINKJ* instructions and for the interrupt instruction *MOVS2I*).

In *DLX* the register *MDR* is the only interface between the register-file and the memory and serves for both loading and storing to the memory. In the pipelined version for *DLX* however a load instruction I which in the pipeline immediately precedes a store instruction I' would compete with I' for writing into *MDR* in its pipe stage *MEM* (when consequently I' in its pipe stage *EX* wants to write *B* into *MDR*). This resource conflict is resolved by doubling *MDR* into two registers *LMDR* and *SMDR* and by refining the *DLX*–rules correspondingly. (Note that the refined rule *pass_B_to_MDR* requires a new direct link from the exit of *B* to the entry of *SMDR* in order to avoid the use of the ALU for this data transfer.)

Since all pipe stages proceed simultaneously and the time which is needed for moving an instruction one step down the pipeline is a machine cycle, the length of the latter is determined by the time required for the slowest pipe stage. In order to balance the length of the pipeline stages the two ALU–rules of DLX are combined into one DLX^p–rule. Similarly the SUBWORD–rule (which selects and outputs to *C* the required portion of the word just loaded from the memory) is incorporated into the WRITE_BACK–rule under the guard that the value to be written comes through a loading instruction from the memory, i.e. has not been computed by executing an *ALU/SET, LINKJ, MOVS2I* instruction in which case the value comes from *C1*. This is done at the expense of linking the exit of *LMDR* directly (without passing through *C1*) to the entry of the register-file and adding to the latter a selector for choosing among *C1* and (the required portion of) *LMDR*.

Note also that transferring a subword of *LMDR* into a destination register can be realized without using the ALU by relying upon the usual register shift functions.

We prove in [24] that the evolving algebra DLX^p obtained from DLX as described above resolves structural conflicts and therefore is a correct implementation of DLX under the assumption that the compiler takes care about avoiding data and control hazards. (In the further refinement steps we show how this assumption can be dismissed by additional architectural changes.)

The **IF stage** is described by the following $\boxed{\text{FETCH}}$ updates:

$$IR \leftarrow mem_{instr}(PC) \quad \textbf{if } \neg jumps \textbf{ then } PC \leftarrow next(PC)$$

where the condition $jumps \equiv$ opcode(IR1) $\in$ JUMP or (opcode (IR1) $\in$ BRANCH and opcode(IR1)'(A)) prevents the inconsistency with updates of PC in the EX stage of jump instructions (see the JUMP rule below).

The **ID stage** is described by the following $\boxed{\text{OPERAND}}$ and latch updates:

$$\begin{array}{lll} A \leftarrow fst\_op(IR) & B \leftarrow scd\_op(IR) & PC1 \leftarrow PC \\ IR1 \leftarrow IR & IR2 \leftarrow IR1 & IR3 \leftarrow IR2 \end{array}$$

The **EX stage** is described by the following nine rules:

$\boxed{\text{ALU}}$ **if** $opcode(IR1) \in ALU \cup SET$
then if $iop(opcode(IR1))$
  then $C \leftarrow opcode(IR1)'(A, ival(IR1))$
  else $C \leftarrow opcode(IR1)'(A, B)$

$\boxed{\text{MEM\_ADDR}}$
if $opcode(IR1) \in LOAD \cup STORE$
then $MAR \leftarrow A + ival(IR1)$

$\boxed{\text{pass\_B\_to\_MDR}}$
if $opcode(IR1) \in STORE$
then $SMDR \leftarrow B$

$\boxed{\text{INTERRUPTS2I}}$
if $opcode(IR1) = MOVS2I$
then $C \leftarrow IAR$

$\boxed{\text{INTERRUPTI2S}}$
if $opcode(IR1) = MOVI2S$
then $IAR \leftarrow A$

$\boxed{\text{TRAP}}$
if $opcode(IR1) = TRAP$
then $IAR \leftarrow PC1$
  $PC \leftarrow ival(IR1)$

$\boxed{\text{JUMP}}$
if $opcode(IR1) \in PLAINJ \cup LINKJ$
then if $iop\ (opcode\ (IR1))$
  then $PC \leftarrow ival(IR1) + PC1$
  else $PC \leftarrow A$

$\boxed{\text{LINKJ}}$
if $opcode(IR1) \in LINKJ$
then $C \leftarrow PC1$

$\boxed{\text{BRANCH}}$
if $opcode(IR1) \in BRANCH$
then if $opcode(IR1)'(A)$
  then $PC \leftarrow PC1 + ival(IR1)$

The **MEM stage** is described by the following three rules:

$$\begin{array}{lll} \boxed{\text{STORE}} & \boxed{\text{LOAD}} & C1 \leftarrow C \\ \textbf{if } opcode(IR2) \in STORE & \textbf{if } opcode(IR2) \in LOAD & \\ \textbf{then } mem(MAR) \leftarrow SMDR & \textbf{then } LMDR \leftarrow mem(MAR) & \end{array}$$

The **WB stage** is described by the following $\boxed{\text{WRITE\_BACK}}$ rules:

$$\begin{array}{l} \textbf{if } opcode(IR3) \in ALU \cup SET \cup \{MOVS2I\} \cup LINKJ \\ \textbf{then } dest(IR3) \leftarrow C1 \\ \textbf{if } opcode(IR3) \in LOAD \\ \textbf{then } dest(IR3) \leftarrow opcode'(IR3)(LMDR) \end{array}$$

4 Separating Specifications from Verifications

The use of evolving algebra allows one to tune specifications to a given application domain without having to care about the peculiarities of an a priori given verification system.

To many this feature of the evolving algebra approach will appear as either trivial or not desirable at all. They will insist that a good specification system has to come together with specific proof and implementation principles and techniques. Surprisingly one finds here representatives from both proof theory and software engineering. Many proof–theoretically inclined researchers tend to identify a semantic definition of a computing system with a (deductive system for a) logic and the reasoning on the so defined objects with proving theorems within that system of logic. But one finds here also software engineers who are looking for specifications produced using some general design calculus which leads to (mechanically verified) transformations into executable code. Let us look at two well-known witnesses.

Dijkstra's calculus for program design is a representative example of such a program development method. In [10] I have given my arguments why such a method, even if applied not to system development in the large but only to programming in the small, limits in an unacceptable way our ability to turn mathematical insight into good programs. Another example is the program synthesis approach which is based on the so–called formulae–as–types interpretation. It advocates the construction of provably correct programs by building proofs in a calculus for which it has been shown that the programs which are extracted from those proofs are correct with respect to formal specifications in the language of the system. Systems like NUPRL, COQ, ALF, LEGO have been implemented to support this researchwise productive approach [50] which encounters however great difficulties where it attempts to synthesize complex real–life programs. The method relies on intuitionistic logic and does not work for classical logic. Furthermore the proofs have to be constructed within a fixed formal system which does not support the use of simplifying heuristics coming from concrete application domains. Thereby the user is forced into a rigid corset in the same way as the programmer is who has to write code in an *a priori* fixed programming language and in a representation which is determined by an underlying machine architecture. Such restrictions can hinder the free exploration of corners of the design space which are difficult to reach and where interesting efficient solutions may be waiting for being discovered.

How important it is for a general design method to separate the concerns of specification and verification can be seen also from the impact this distinction has on building verification tools. Each particular specification language and tool relies upon a certain number of built-in assumptions and design decisions. Therefore one has to be careful to use it only for applications which share those assumptions. For example Lamport's TLA is based on a fixed notion of equivalence. If the system is used for proving equivalence concerning an application with a coarser notion of equivalence, one is forced to prove more than required; if the application has a finer notion of equivalence, one has to identify items which one really wants to distinguish (see [44]). Evolving algebras can be used in connection with *any* notion of equivalence.

Another illustration comes from the success obtained by tools for program development and verification which have been tailored to particular application domains. As outstanding recent examples one can cite here D. Smith's method for synthesis of high–performance transportation scheduling programs [77, 76] and Clarke and Dill's machinery for the verification of programs which can conveniently be mapped to finite state transition systems of manageable size and can be dealt with by automated model checking. Another remarkable example is Russinoff's [73] functional definition of an important subset of VHDL—the IEEE standard for a hardware description language; it comes together with related procedures for deriving and verifying behavioral specifications of combinatorial and sequential devices which can be formally encoded in Boyer–Moore Computational Logic for mechanical proof checking.

In such situations it is the precise knowledge of a well defined application domain which provides the insight into how to tune the design and verification—or analysis and synthesis—principles to each other and to the given application domain. Our experience shows that evolving algebras make it possible to take advantage from the separation of concerns: they assure the highest degree of flexibility in adapting the means of description to the specific features of any application domain and they allow one to incorporate into the proof methods all the knowledge which is available in the application. Indeed in complex design situations drastic simplifications can be obtained if one *first* circumscribes the conceptual constituents of a specification and only then starts to look for possible means of proof. But more importantly the evolving algebra approach allows one to use the brain where the muscles of even the strongest mechanized proof system ("brawn methods") cannot really help any more.

This claim is supported by our experience in working with evolving algebras for proving run-time properties of complex systems. For example during our work on the correctness proofs for the compilation of Prolog and Occam compilers [28, 17, 16], at various occasions we avoided getting stuck only because the evolving algebra framework allowed us to invent intermediate models which had not been expected at the beginning. By introducing these intermediate specification levels we could reduce the complexity of each single refinement step in such a way that we were enabled to formulate and prove the relevant correctness statements. There is a trade–off between the abstraction difference of two specification levels and the difficulties one encounters when trying to prove properties which relate specifications at those levels. My guiding principle for breaking complex statements into simpler ones has been to stop only where the proofs become routine inductions and case distinctions which can be carried out by an automatic or interactive theorem proving system. Indeed at present two research groups at the universities of Karlsruhe and Munich are using their theorem proving systems KIV and ISABELLE respectively to provide a machine verification of our mathematical WAM correctness proof.

The flexibility of evolving algebras has also the effect that their use can easily be integrated into existing development systems—a vital feature for a practical design method. Such an integration might easily turn out to be impossible for a fixed formal system which could require, by its rigidity, to completely change the entire already existing development system—a pragmatically speaking unrealistic request.

It is still unclear to what degree complex systems can be satisfactorily let alone completely verified. The evolving algebra approach allows one to give *relative* correctness proofs, i.e. proofs that specific system parts function well under precisely stated assumptions on a well behaving environment. Such environmental parameters and their properties enter the specification and the proof as abstract interface, technically speaking as externally alterable or as oracle functions. Since evolving algebras can be appropriately tailored to any desired interface they permit to satisfactorily solve the design and analysis problems of an embedded system *modulo the context* into which this system is embedded.[6] For solutions of practical problems in our real word we cannot expect more from a scientific method as Popper explains with a good picture [71]:

> Die Wissenschaft baut nicht auf Felsengrund. Es ist eher ein Sumpfland, über dem sich die kühne Konstruktion ihrer Theorien erhebt; sie ist ein Pfeilerbau, dessen Pfeiler sich von oben her in den Sumpf senken—aber nicht bis zu einem natürlichen, 'gegebenen' Grund. Denn nicht deshalb hört man auf, die Pfeiler tiefer hineinzutreiben, weil man auf eine feste Schicht gestossen ist; wenn man hofft, dass sie das Gebäude tragen werden, beschliesst man, sich vorläufig mit der Festigkeit der Pfeiler zu begnügen.

[6] I am far from saying that this tailoring of a model to a desired interface is trivial: "The trickiest part is in explicitly stating the assumptions about the environment in which each critical piece is placed."[78, p. 13] Evolving algebras allow one to make such assumptions explicit in a direct way, without creating additional overhead which deals only with the formalism and not with the reality to be modeled. If there are problems in the subject matter, they are not solved by using evolving algebras; but use of the latter avoids the introduction of additional problems.

No formal method can guarantee the absolute correctness and safety of computer systems. The best one can obtain is relative correctness. Evolving algebras allow us to push the frontier of this effort much beyond what is widely believed—in particular in engineering and industrial circles—to be an inherent limitation of mathematical methods.

5 Avoiding the Formal System Straitjacket

Instead of building upon just one particular formal system the evolving algebra approach is an open framework into which a variety of systems can be incorporated. Evolving algebras assure the greatest possible *freedom of language* and *freedom of proof.*

Freedom of language. The freedom to choose how to represent the basic objects and operations of the system under consideration is crucial in two respects, namely for constructing satisfactory ground models and for building hierarchies of system levels. Related to this is the need to distinguish between concepts (*mathematical modelling*) and notation (*formalization*) and to remain flexible enough to be able to "choose the right notation".

A special "correctness" problem arises in those cases where the original requirement specification is given informally. This problem is a case of the general problem about the applicability of scientific methods to the real world and is relevant for the design of computer systems. I will explain in the section on *ground models* that using evolving algebras one can satisfactorily settle this problem thanks to the possibility to *tailor a description to the given application domain.*

The methods of *abstraction* and of *stepwise refinement* have been rightly recommended as a way to cope with complexity by *building and crossing hierarchies of system levels.* The two methods are intimately connected: only a specification framework which allows one to freely choose the appropriate abstractions for a given problem can provide the full freedom of refining abstract descriptions to intended lower level implementations. How else can we freely move among different language levels in computer design? This is well illustrated by an example suggested by Simon Read [72]: the industrial hardware design process at present uses separate languages for specification (behavioural VHDL), for design (Register Transfer Level VHDL), for implementation (cells), for verification (BDD's) and for mathematical reasoning (Boolean algebras). The example shows that any a priori restriction of the language by a fixed formal framework either results in restrictions of the application domain or forces you to work on encodings which are extraneous to the problems under investigation and usually are responsible for the well known combinatorial explosion encountered so often when specification frameworks are applied to real systems.

The freedom of language offered by evolving algebras assures that they can be integrated into arbitrary contexts at arbitrary system level without encountering any compatibility problems. Technically speaking this is based upon the choice of the language of mathematics (i.e. classical logic) as the underlying language.

This does not represent an a priori restriction of the specification language because the language of logic is as broad as a precise algorithmic language can be; as explained above there is no more basic algorithmic device than a transition rule system with guarded abstract function updates. The language of evolving algebras is flexible enough to deal with any practical algorithmic language of the working computer scientist. The choice of the language of mathematics guarantees that in the evolving algebra framework one can adapt the formalization of a model or a mathematical concept to the need of what has to be modelled. There is no special notation one has to learn; no peculiar syntax is imposed on the designer.

Freedom of proof. The evolving algebra approach offers the necessary freedom for the choice of the proof techniques used to establish properties of computer systems. This has to do with the fundamental *distinction between formal and rigorous* descriptions (or proofs), and more generally between various *degrees of precision.*

Traditional mathematical proofs aim at understanding by humans, at revealing structure which guides the comprehension; as a result they typically lead to further investigations. The essential ingredient of this process is creativity in *finding* proofs; creativity is the heart of mathematical progress. Formalized proofs provide *verifications* within a particular proof system. They belong to what Gurevich [40] proposed to call *Pedantics*, a respectful and badly needed scientific discipline of validating mathematical proofs. It is here that we will find machine checked proofs providing all the details which are suppressed in traditional mathematical proofs. Machine checked proofs can contribute to the pragmatic acceptance of theorems. Progress in pedantics will provide more freedom for writing less formal creative proofs. The role of interactive proof checkers is to couple in a practically fruitful manner the checking abilities of automated theorem provers with the user's creativity in finding proofs.

In connection with the need to develop correct programs for safety-critical applications, Pedantics has a still more important and (with respect to traditional mathematics) new role. Professional deontology obliges the mathematician to check the proof of any theorem he is going to use. This obligation was sufficient until the middle of this century to guarantee for mathematical results a very high (although not absolute) degree of reliability. The slow pace of this proof checking process by the mathematical community, a process which involved generations, was in equilibrium with the pace of discovery and of applications of new theorems. This has changed radically in the second half of this century. Statistics show us that nowadays on average there are only 2.5 readers per publication in mathematics. This fits the often quoted anecdotal claim that about one third of the published mathematical papers contain serious errors. But such statements say more about the refereeing process than about the inherent unreliability of mathematical proofs. Often despite errors in the proof, the theorem is correct and the proof can be corrected; a good mathematical proof creates the right images in the reader which allow him to fill the details and to repair the flaws.

The discipline of pedantics has to provide the laboratory conditions in which the traditional and slow proof checking process by humans is if not replaced then at least enhanced and speeded up by machine assisted interactive proof checking procedures; see the interesting report [66]. This is a rationale for programs like Beta testing. Pedantics has to create appropriate methods and criteria which an applied computer scientists can use when they need to decide whether and to what degree a proposed proof is reliable.

The notion of proof is not absolute. There is a hierarchy of notions of proof, each level having its own degree of precision. There are not only classical, intuitionistic, constructive proofs, but also flat and deeply structured ones. Mathematical proofs are always to be understood within a context of concepts, methods, groups of experts, background of previously developed theories. Their role, as analyzed by Plato who made the fundamental discovery of the underlying notion of universally valid law, is to establish *valid arguments*. Aristotle made this concept operational by inventing specific proof principles, even a proof format which has deeply influenced the occidental culture until it was again revolutionized by Frege and Hilbert. It is well known that the progress in mathematics is intimately related to the discovery of new notions and proof principles. Gödel's incompleteness theorem shows that this is not a historical accident but that it is necessarily so when we codify mathematical knowledge in concrete and well delimited formal frameworks.

The same holds for mechanical proofs. They are surely useful but also not absolute; their reliability depends on the correctness of the environment where they are executed and on the adequacy of the formalization. Fully formalized proofs are certainly less convincing for humans than well checked traditional mathematical proofs (see the discussion in [42]). It is for a reason that theorem proving researchers try hard to make machine produced proofs readable to humans.

Thus we have to accept that the notion of mathematical proof has a pragmatic component and thereby is limited by the progress of mathematics and pedantics.[7] On the other side one can say that the mathematical rigour has the highest degree of reliability which has been reached in occidental culture. I do not want to enter here into the discussion whether it is good to identify "formal" with "mathematical". We may be well advised to equate formal with other forms of rigour, as encountered in natural or engineering sciences or in jurisprudence. But in any case, a mathematical verification method has better chances to prove interesting theorems for real systems if it is not restricted to a fixed deductive system but can make use of all what mathematics has to of-

[7] To recognize the limitation of knowledge acquisition by mankind does not imply "that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem", as has been claimed in [36, p. 271]. Here is not the place to enter the philosophical discussion on the status of objectivity of mathematical knowledge, a dispute which is with us since the ancient Greeks. One should keep in mind that a rigorous foundation of science must and can be *self-evident* in the Aristotelian sense, neither psychological nor social, see [1].

fer. This includes deductive systems, the discipline of Pedantics and the use of crisp and powerful mathematical techniques for the stepwise development and verification of complex software and hardware systems. The freedom to choose proof methods freely is a key to the success of the evolving algebra approach in proving non–trivial run–time properties.

6 Appropriate Ground Models

Ground models[8] play a crucial role in attempts to face the general problem of applicability of mathematical concepts and methods to the physical world. In this section I illustrate the notion of ground models in relation to computer science and explain the extraordinary potential of the evolving algebra approach to build satisfactory ground models.

The fundamental question is about the relation between our mind and the inexact real–world phenomena. How can we relate our theoretical scientific models to the reality of our world? Typically what is given is a system S of problems in the real world for which I want to find a scientific solution. In computer science the solutions we are looking for are algorithms (in the general sense of the term, including interpreters for programming languages, protocols, architectures, etc.).

6.1 The foundational problem.

In the realm of our models we have mathematical methods to solve precisely defined classes of problems. The given system S itself is not a formal one but incorporates our intuitions about the basic objects and the basic operations which are the constituents of the given problem. What we have to relate to them in our mathematical models are definitions of concepts and of functions. We orient ourselves in the real world by the expertise in the application area to which S belongs. In the world of scientific methods we have instead formal manipulations of models, say sequences $S_0 \rightarrow \ldots \rightarrow S_n$ of models S_i where typically the final element of such a chain is an executable version of the proposed problem solution and where—at least ideally—the transformations $\rightarrow$ which lead from S_i to S_{i+1} are provably correct. Aristotle pointed out that such a chain of provably correct stepwise refined specifications has to start somewhere and has to be finite. The whole specification chain will remain an intellectual exercise if we do not "know" that S_0 is "correct".

How can one establish the *correctness* of such a first model S_0 in a specification chain? The question is how we can relate the non–formal system S to the formal model S_0. By definition there is no provable relation between the mathematical object S_0 and the loosely given informal system S. Therefore the only thing we can hope for is a pragmatic foundation: we have to grasp the

[8] In [9, 27] I had called them primary models. This was to stress that these models are not unique but are naturally thought of as those models from which others are derived by formal transformations. Ground models in this sense should not be confused with the notion of ground models as it is used in logic programming.

correctness of S_0 with respect to S *by inspection.* To say it in computer science terms: we have to understand the semantics of S through the model S_0.[9] Or to say it from the software engineering perspective (see [48, p. 346]): "a specification must be validated against the intuitive understanding". In order to make such a pragmatic foundation safe, the *ground model* S_0 (which formalizes the basic intuitions, concepts and operations of S) has to satisfy a certain number of requirements.

Before describing these requirements I want to stress once more that this correctness problem is part of the general problem how to relate rigorous scientific methods to the real world. Since building satisfactory ground models is the working computer scientist's daily bread the correctness problem cannot be solved once and for all for the whole body of the current basic concepts of the discipline. The computer scientist has to solve an instance of this problem each time he has to implement a system which satisfies a given requirement specification. In classical engineering disciplines, the experience accumulated through one hundred years of work has produced a body of "right" modelling concepts and methods which in general permit the satisfactory resolution of the issue of finding good ground models. There is no such well established body of engineering knowledge in computer science yet. Remember that as we know from statistics, 80% of the errors in system programming do occur at the level of requirement specifications, i.e. at the place where we have to relate the given system S to a ground model S_0 for it.

6.2 Properties of ground models.

There are at least three requirements which must be satisfied by ground models in order to serve as safe basis for a specification chain which provides a correct implementation of a given application system S. Namely ground models must be *precise*, *abstract* and must have a *rigorous foundation.*
Precision. A ground model must be precise in order to become subject to mathematical analysis at all. In connection with the distinction to be made between *rigorous* and *formal* (see above) it has to be required that despite of its precision, a ground model has to be *flexible, simple, concise* and *falsifiable.*

The **flexibility** requirement wants to make sure that ground models are adaptable to the characteristics of different application domains. Flexibility has also to permit ground models to meet the important software engineering principle which asks the models to be easily modifiable and in particular extendable. Extensibility is necessary for ground models to serve as prototypes and also to become reusable in the design process.

[9] This is the typical situation of a standardization effort for a programming language: S is what the standardization wants to abstract from the existing descriptions and implementations of the language. Once the standard is defined— ideally by a ground model—it should be possible to understand the behaviour of a proposed implementation through that model. For Prolog I have developed an evolving algebra ground model, see [7, 12, 27]; it defines the semantics of the ISO Prolog standard. An IEEE VHDL'93 Standard evolving algebra ground model appears in [21].

The condition of **simplicity** and **conciseness** wants to make sure that ground models are understandable by the user. This is crucial from the software engineering point of view (see [48, p. 346]): "To be useful in industrial practice, specifications must be comprehensible". I am not saying that say aircraft specifications should be comprehensible by the future passenger or by the Airline executive who buys them; they must be comprehensible to the engineer who understands the application. [10]

The user typically is not a computer science specialist but an expert in the given application area. He is the one who has to check by inspection that the ground model S_0 faithfully reflects the application system S of problems formalized by S_0. That is why the model S_0 must be as close as possible to the reality it is going to formalize, i.e. its elementary objects and functions must represent directly, without encoding, the basic concepts and operations which appear in the application system S. "A significant engineering project begins with a specification describing *as directly as possible*[11] the observable properties and behaviour of the desired product"[53, p. 4]. This formalization task "... is not simple: it requires a careful choice of those aspects of the real world to be described in the formal language and an understanding of both the detailed practical problems of the application and of the formal language. Errors would likely be introduced during this process ..."[59, p. 41]. Therefore the ground model must reflect the user's domain expert knowledge, it must capture the domain intrinsics and go hand in hand with the requirements capture in the software cycle (see [5]).

The ground model represents the precise interface where the discussion between the user and the designer of the system has to take place and where the contract between the customer and the implementor is formulated. All the furthermore refined models $S_i (i > 0)$ belong to the world of the designer, not to the world of the client. The ground model S_0 has to express what the system

[10] From my work for the ISO Prolog standardization I can cite an interesting example which might serve as illustration here. In the ISO Prolog standardization working group (ISO/IEC JTC1 SC22 WG17) it has been discussed for years whether the semantics of the language should be defined by executable stratified control-free Prolog code proposed for this purpose by Deransart [37]. The alleged advantage was that whenever a question comes up about what the standard requires, it can be answered in a definite way by running the program which defines the standard. But one has to be conscious about the difference between explaining and executing a program. What if we have difficulties in understanding the program? Then running the code to decide a question is like querying an oracle and just believing the answer and behaving accordingly, even if the answer is cryptic. As a matter of fact Deransart's program is a rather large one (of "about 500 clauses" [37, p. 30]) which has two crucial problems: a) it is written in a sublanguage of Prolog whose semantics and therefore implementation depends upon subtle problems having to do with the non classical treatment of negation in Prolog (see [7, section 4] for a detailed discussion of this problem); b) "it is not very easy to understand by anyone unfamiliar with their (viz. the authors') methods and notations"[60, p. 1]—a reason why it had to be accompanied by an extra "Explanation of the Formal Definition" [60]. This confirms the need of a comprehensible ground model for a standardization effort.

[11] My italics.

is supposed to do, the refined models $S_i (i > 0)$ define how this behaviour is achieved. Clearly the *what* has to precede the *how*. The practical software engineering importance of the simplicity and the comprehensibility of the language for ground models comes out clearly also from statistical evidence: two thirds of the development time is spent for communication between user and designer and one quarter of the failures of software projects are due to communication problems between user and designer.

The ground model S_0 must satisfy the Popperian criterion of being *falsifiable*. This means that a) the ground model must assure the possibility to make statements about the design that are either verifiable or falsifiable and that b) the user must have the possibility to test the appropriateness of the model by experiments with reproducible results, using S_0 or executable prototypes[12]. In [54] these consequences of the Popperian falsifiability criterion are postulated as quality standard for any verification method.

The falsifiability request for *ground* models does not contradict but complements the role of the use of formal methods during the—ideally provably correct—development of the *refined* formal models $S_i (i > 0)$. Thus it is natural that in the GEC Alsthom project of software development for speed– and switching control of the French railways, the entire design is based upon Abrial's formal method B, but nevertheless also functional tests are done. Fernando Mejia who directs this formal development project indicates two reasons which illustrate my analysis (see [38, p. 77]): "First, programmers do occasionally make mistakes in proofs." As explained above the discipline of *pedantics* creates the laboratory conditions which have to enhance or to replace the traditional form of the mathematical proof checking process. "Secondly, formal methods can guarantee only that software meets its specification, not that it can handle the surprises of the real world". This expresses the fundamental distinction between the pragmatic scientific foundation of ground models and the mathematical justification of refined models.

Abstractness. Ground models must be abstract. They have to reflect the intrinsic functionalities which are constituents of what appears in the non–formal problem formulation and therefore will be needed for an algorithmic solution of S, but they should not contain any irrelevant representation or optimization features. The latter will have to be considered only in the specification chain which leads from the ground model to an implementation. Being abstract has to go together with being *complete*. Completeness means that the semantically relevant parameters of S must be present in S_0 although they might appear there only as interface relating to a possibly abstract environment.

Ground models with the right abstractions satisfy the postulates in [54, p. 36] for a mature engineering discipline, namely that one is *"able to make explicit formal statements about the correctness requirements for a new design that are independent of the design itself"*, that one can *"discriminate between requirements and implementations"*, that one can predict the essential characteristics

[12] Simple and easily implementable ground models are also important for simulation purposes; they allow one to test performance criteria at an early development stage.

of a product before it is built and that one is *"able to build an abstraction of the behaviour ... which is susceptible to formal analysis"*. The last mentioned postulate is in accordance with the view explained below that practical (in particular ground) models better support the process-oriented understanding of a dynamic system.

Rigorous foundation. The necessity of the third requirement for ground models, namely to have a rigorous foundation, is obvious. But note that this requirement does not only satisfy an academic intellectual desire. It also corresponds to a practical need when it comes to build reliable tools which help for design and transformation of ground models in connection with prototyping. In [54, p. 36] a sound formal framework for capturing prototypes is even considered as the first quality standard for any verification method.

Also through this requirement one can see how fundamental the notion of evolving algebra is: the foundation of evolving algebras [41] is in first order logic, the most general and simplest basis occidental culture knows for mathematics. Paraphrasing a famous slogan I am tempted to comment this by saying that *Simplicity is our business.*

7 Support of Abstract Operational Views

Evolving algebra specifications directly support the process-oriented understanding of the behaviour of dynamic systems and they allow one to build operational models at arbitrarily high or low levels of abstractions. If for given dynamic features one looks for mathematical descriptions which are simple and concise, easy to understand and manipulate (whether by humans or by machines), one is well advised to try to express dynamic changes *directly* by dynamic concepts. As explained above, the notion of evolving algebra incorporates directly the most basic dynamic concept we know in computer science, namely destructive assignment, and it does it in the most abstract form one can conceive. By supporting such abstract operational views evolving algebras allow one to deal in an explicit and transparent way with non–trivial run–time properties.

For example the analysis of the dynamic properties of Prolog database operations in [15, 26] is based upon a simple but precise model of the backtracking behaviour of Prolog for user–defined predicates which abstracts away from irrelevant features by working with abstract interfaces for terms, goals, clauses, procedure definitions, substitution and unification. Whereas the analysis in [15] is at the user's level of observation, the model in [26] covers the behaviour of pseudo–compiled Prolog code which reflects all the features of the database related WAM instructions. In our evolving algebra specification of the Warren Abstract Machine [28] we develop various abstract models for different WAM layers (predicate structure, clause structure, term structure, etc.) in order to state and prove the intended correctness theorem for compilation of Prolog programs. These models make a mathematical (implementation independent) study of WAM-related execution or implementation issues possible. Also the evolving algebra definition of Prolog [27] yields several modules around the simple Prolog nutshell for user-defined predicates, each for a different group of built-in predicates. These modules provide the mathematical basis to extend abstract analysis from Horn clauses to real Prolog programs which contain built-in predicates formalized in those modules.

The claims about abstract operational specifications will surprise those who share the widely believed view that *operational* and *abstract* are conflicting properties of formal methods, that they exclude each other. For a long time it has been a common place in theoretical computer science that in comparison to the equational or axiomatic approaches, the operational approach to semantics is scientifically not so respectable. It is considered to be of lower level of abstraction which may be good for producing implementations but not for defining succinct high–level descriptions of desired functionalities of a system. It is supposed to deal with dirty control features or efficiency considerations which belong to code execution but should be hidden from the system designer.

Only Plotkin's structural operational semantics SOS [70] is sometimes considered as an acceptable style of defining the meaning of programs. In that approach, the program constructs guide the formulation of axioms and rules which define the meaning of programs and serve for proof schemes. Indeed SOS has been useful. The formal definition of Standard ML makes crucial use of Kahn's *Natural Semantics* [58]—a version of structural operational semantics. The semantic rules are used to answer questions about the meaning of program constructs. Theorems justify particular design decisions and are typically proved by induction on the programs, see [61, 62].

In the communities of functional and logic programming and of artificial intelligence it is still rather common to identify *abstract* with *equational* or *declarative* as opposed to *operational*. Surprisingly enough not only among theoreticians but even among researchers in system design and analysis one encounters such an attitude. For example to my surprise this view was held by a distinguished VHDL specialist in a panel discussion on the use of formal methods at the 1994 *European Design Automation Conference* in Grenoble. Also some advocates of "pure" programming styles have contributed to discrediting operational (imperative) methods as "impure" by banning "side effects".

I explain in the rest of this section why I believe it to be misleading to look at *operational* concepts as being in conflict with the method of *abstraction*.

Abstract versus operational. Let me start by recalling that when the attractive and fruitful concept of denotational semantics was discovered, its pioneers explicitly aimed at providing mathematical models and tools to deal in a precise and safe way with real languages and systems. In [75, p. 40] we read:

> An essential topic will be the discussion of the relation between the mathematical semantics for a language and the implementation of the language. What we claim the mathematics will have provided is the standard against which to judge an implementation.

Only later it has been claimed that for defining the semantics of a program one should forget the notion of *state* and the notion of *individual computation step*.

Both notions have been relegated to the world of implementations as not abstract enough. Also in the stateless process algebra we can observe the high price paid for abstracting from states; for example, the proposed general refinement theory does not capture many refinements that are crucial for the analysis of complex real systems which do refer to state constituents; see e.g. the refinements in the evolving-algebra-based correctness proofs in [27, 17, 16, 55, 43, 22].

Purely functional (equational) definitions of the semantics of programs have been advocated which are global and do not refer to single computation steps. By now it is acknowledged by many [63, p. 626] that such semantics has problems in coping with data abstraction mechanisms in programming languages (descriptive inadequacy) and with reactive programs. The notions of state and single computation step, which such pure approaches want to avoid, need not to reflect irrelevant details of execution but serve as the basis for the important *principle of locality* discussed above. The vast literature on models of logic programming languages offers an interesting illustration of the difference between "local" and "global" specifications. Compare the evolving algebra definition of the full programming language Prolog in [27] with various equational, or abstract algebraic, or axiomatic, or "pure" logic specifications for the sublanguage of definite Horn clauses extended by some control elements (see [65] and more references in the introduction to [27]).

In a sense fixed point based descriptions of programming languages reintroduce both the 1–step computation and the state notion; but they do this in an implicit and technically more involved way. Single computation steps reappear in the form of successive approximations to the fixed points for the given equations. The relevant state components reappear in the form of continuations. Again an example is provided by some denotational definitions of the semantics of Prolog programs in the literature. For each abstract domain which appears in the evolving algebra definition of the core of Prolog for user–defined predicates [27], the denotational descriptions in [57, 34, 35] introduce a continuation: for terms, goals, (occurrences of) clauses, programs, substitutions, etc. The difficulty is that the denotational Prolog models have to deal with these continuations explicitly—as constituents of the global "object" which encodes the "state"—via complicated and numerous equations which describe the effect of the Prolog computation. In the evolving algebra description of Prolog [27], the corresponding elements of abstract domains appear without any encoding; no overhead is needed for their representation, they occur directly in the four simple rules which define the complete Prolog machine for user–defined predicates. These abstract elements are subject to further refinement at the moment when their implementation becomes the issue, e.g. in the WAM [28].

Models versus Syntax. It is true that in the theoretical computer science community serious attempts are made to overcome the practical problems of purely denotational approaches to semantics.[13] The development of *Action Semantics*

[13] I do not want to throw doubt on the many achievements which have been obtained in the area of denotational semantics. The development of denotational semantics has brought us a good understanding of types and higher–order functions (in particular

[64] is an example. In general, during the last years there seems to be a trend in theoretical computer science to make semantics more operational. The interest in full abstraction and observational equivalences is a related phenomenon.

It is my belief that people put too much emphasis on syntax. In Plotkin's Structural Operational Semantics, the proof rules for actions are directly derived from the syntactical structure of the programs. Such semantics is compositional which allows one to establish many useful properties, but there is a price. Not all programming constructs lend themselves to such treatment which limits the applicability of the method. Similar restrictions apply to abstract interpretations of run–time investigations. In this connection compare the following:
1. According to Hoare, it should be the case that "all the properties of a program and all the consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning" [51, p. 576].
2. Von Henke, one of the fathers of the successful PVS system [66], points out that most tools which have been developed up to now "in dealing primarily with syntactic and structural aspects of software ... fail to address major issues of software quality having to do with *semantic* aspects of software" [48, p.345].

The evolving algebra approach allows one to model and analyze the run–time program behaviour at the desired level of abstraction. What is considered as one step in a computation depends on the level of abstraction at which we want to build the computation model. Evolving algebras overcome the presumed insufficient abstractness of operational semantics by allowing one to have as 1–step computation an arbitrary (intendedly finite) number of conditional updates of functions of arbitrary level of abstraction. The use of guarded function updates for local transformations of structures provides a tool to build precise yet simple, abstract yet operational models for complex real-life systems which support the intuitive process-oriented understanding of the system. It gives the notions of state and state transformation a mathematical status which allows one to use powerful mathematical techniques for characterizing the dynamic behaviour of complex real systems.

Evolving algebras utilize all the traditional means for the description of static features. For example the initial states from which evolving algebra computations start can be often described adequately by algebraic (purely equational or axiomatic) specifications. Evolving algebras take advantage of abstract data types. The explicit dynamic behaviour is defined on top of all that.

The evolving algebra approach to program design and analysis is semantical. It allows one to study in precise and abstract mathematical terms the *effect* of

---

of recursive and polymorphic types) for functional programming, especially for the λ–calculus which plays the role of the underlying canonical computation model. I am discussing only whether these results gave as much to practical applications as the pioneers intended. Who does still remember the goal stated in [74, p. 2] that the mathematical definition of a language should enable one to determine whether a proposed implementation is *correct*? I certainly do not deny that in particular contexts denotational definitions can be both elegant and useful. I have mentioned already Russinoff's work on VHDL [73] as an example. Another instructive example is Björner's application domain model for the Chinese railway computing system [4].

programs; it permits one to concentrate on those *behavioral* features for which the program has been or is to be developed. There is no limitation imposed by, say, viewing programs as logical formulae or viewing programming as carrying out proofs (or normalizations of proofs) within a fixed deductive system. However a proof theoretical analysis is *also* possible[14].

8 Scalability

I have often heard the statement that formal methods had their chance and failed. It cannot be denied indeed that several traditional formal methods are not scalable. They work well for small examples, often invented to illustrate the method. But when it comes to real–life large systems many of them face the well known combinatorial explosion or simply fail. *Developing techniques for crossing abstraction levels* [52] represents one of the challenging goals of current computer science research and is of vital importance for the development of reliable safety critical systems (see the title *Developing abstraction for coping with complexity* of a section on safety in the IFIP 1994 World Computer Congress, op.cit.). The use of evolving algebras allows one to cope with the complexity of real systems by building hierarchies of system levels.

I have mentioned already several real-world case studies through which I wanted to test whether they support the preceding claim. One is the mathematical definition [27] of the semantics of the real programming language Prolog which went into the ISO standard (see [12]) and has been refined by a hierarchy of intermediate models to a definition—coming with a correctness proof—of its implementation on the Warren Abstract Machine. Another example is the recently finished project of a formal definition of Occam at the level of the user [17] which has been refined through a hierarchy of intermediate models to the Transputer level [16], again coming with a mathematical correctness proof for the compilation of Occam programs into Transputer code. A recent example is the reverse engineering project [13] where evolving algebras have been used for a mathematical specification (leading to only four pages of abstract parallel code) of the VLSI implemented microprocessor which controls the successful dedicated massively parallel architecture APE100.

The experience gained in those projects gives me the conviction that the evolving algebra method scales up to complex systems. I believe that it can play a particular role in hardware/software co-design. Through the freedom of abstraction together with the information hiding and interface mechanism evolving algebras can serve as tool to develop and analyze a design without committing to a particular technology for realizing the design; by the flexibility to formally represent all the system parts as evolving algebras one can postpone to a late design stage the decision about which parts to realize in hardware and which ones in software. I am seriously interested in finding a challenging industrial size problem where these convictions could be proved.

[14] Indeed there are attempts to analyze evolving algebras as systems of rewrite rules or a form of Horn clause programs.

9 Easy Learning

It is easy for the practitioner to learn to use evolving algebras in his daily design and analysis work. Evolving algebras use only standard mathematical notation; one can work with them using any knowledge or technique from existing practice and avoiding the straitjacket of any particular formal system. As a consequence the use of special application domain knowledge or design expertise is supported and provides the possibility to decompose complex systems by *familiar* techniques into simpler subsystems in such a way that this decomposition is formalized in a rigorously controllable manner.

During the last years I have experienced with numerous programmers, implementors and hardware designers that evolving algebra models—i.e. abstract assembler-like code— can be understood by the working computer scientist without any formal training in theory. It needs not more than a day or two of explanation, through simple examples, to convey to a hardware or software engineer the idea and a precise definition of evolving algebras in such a way that he can start to produce his own well defined evolving algebra models. Look at the two evolving algebra models for DLX. It is true that this RISC architecture is simple by itself, but nevertheless the example illustrates I hope how one can formalize or construct a real system following a natural path of explanation and using standard mathematical notation in such a way that the resulting evolving algebra model becomes simple and transparent and easy to read, to understand and to manipulate.

This refutes, for the evolving algebra approach, an objection which is often put forward against large scale industrial use of formal methods. It is said that the average programmer has not enough mathematical skill to be able to apply a formal method which needs a PhD to be understood. I cannot judge whether this claim is true, but for sure no PhD is needed to understand and to correctly use evolving algebra models; just experience with algorithmic (programming) phenomena is sufficient. If "mathematical abilities" are needed for the system development process, this is due to the fact that the development of any algorithm aimed at solving a given problem constitutes among others also a mathematical (combinatorial) achievement and includes a formalization (representation) task. The question therefore is not whether "mathematics" is needed or not, but how the intrinsically mathematical part of the programming activity is supported by a method, be it "formal" or not. This is expressed also by Wing [78, p. 10]: *"Programs, however, are formal objects, susceptible to formal manipulation ... Thus, programmers cannot escape from formal methods. The question is whether they work with informal requirements and formal programs, or whether they use additional formalism to assist them during requirements specification."*

Evolving algebras support the system development activity ideally because they give a chance also to the non-theoretically drilled programmer to express the system features appropriately at the desired level of abstraction. This feature can also be put to use for writing informative manuals. As a side effect of a stepwise refined system development (ideally coming with proofs which relate the different levels) one gets a systematic documentation of the system for free.

Thus the use of evolving algebras can not only make the system design process more reliable, but it can also speed up the whole process and thereby make it less expensive. This is good news which however has still to be understood by industrial circles.

The ease with which the working computer scientist can build simple and clean evolving algebra models is one of the reasons why evolving algebra models can be delivered, not only promised or built "in principle"; this holds even under industrial constraints.[15] I would be very much surprised indeed should this potential of the evolving algebra method not influence the fate of Industrial–strength Formal specification Techniques[16].

10 Conclusions

I have illustrated some important features of the evolving algebra method to design and analysis of complex computer systems. I have shown that this method satisfies the conditions which are widely required for a "mature engineering discipline" ([54, p. 36]). After having finished this text I became acquainted with [31]; I must confess that the evolving algebra method seems to be an orthodox one: it obeys all the *Ten Commandments* of [31]. I hope to have revealed its practicality for the working computer scientist and also its potential to become a viable mathematical method especially under industrial constraints.

At this point the reader might wonder what are the limitations of the evolving algebra approach. In principal they seem to coincide with the limitations of the human capabilities of mathematical formalization and structuring. I do *not* advocate the reformulation in terms of evolving algebras of all the successful techniques which have been developed by the use of other approaches, especially not for the description and analysis of static phenomena. I want to suggest however that wherever dynamic behaviour is at stake, evolving algebras will be helpful and are there at their best.

Important future achievements will be a) to marry the concept of evolving algebra with control theory and b) to develop the "Pedantics" and the tool side of the approach. I hope that some readers will feel challenged to contribute to this work.

Acknowledgment. I am grateful to the following colleagues for stimulating conversations on the subject, for valuable criticism and last but not least for helpful comments on previous versions of this paper: Dines Björner, Jonathan Bowen, Uwe Glässer, Yuri Gurevich, Nils Klarlund, Leslie Lamport, Jim Lipton, Erich Marschner, Peter Mosses, Alan Mycroft, Peter Päppinghaus, Lutz Plümer,

[15] For example the four simple rules which constitute my evolving algebra model in [7, 8, 27] for the Prolog kernel for user–defined predicates have been implemented in two leading Prolog companies, one in the US and one in Europe. Such implementations can be used as running prototypes for industrial experiments with new developments.

[16] WIFT is the acronym for the Workshop on Industrial–Strength Formal Specification Techniques which is sponsored by the IEEE Technical Committee on Software Engineering.

Simon Read, David Russinoff, Britta Schinzel, Peter Schmitt, Kirsten Winter. A few paragraphs in this paper are direct quotations from [19]. Thanks to Franz Rammig for the occasion offered to discuss some of the ideas presented here to the panel on *Formal Semantics: Practical Need or Academic Pleasure?* at the annual *European Design Automation Conference with EURO–VHDL* in Grenoble (19.–23.9.1994). Last but not least my thanks go to BRICS at the University of Aarhus for the invitation to an intensive and pleasant summer month which allowed me also to finish this paper.

References

1. D. Barnocchi. L"Evidenza" nell'assiomatica aristotelica. *Proteus*, II,5 (1971), pp. 133–144.
2. Ch. Beierle and E. Börger. A WAM extension for type-constraint logic programming: Specification and correctness proof. Research report IWBS 200, IBM Germany Science Center, Heidelberg, December 1991.
3. Ch. Beierle, E. Börger, I. Đurđanović U. Glässer, and E. Riccobene. An evolving algebra solution to the steam-boiler control specification problem. Seminar on *Methods for Specification and Semantics* (Dagstuhl, June 1995), Report, 1995.
4. D. Björner. A Formal Model of the Railway Application Domain System. UNU/IIST PRaCoSy Document no. SP/5/3, January 7, 1994, pages 1–19.
5. D. Björner. Domain Analysis, a Prerequisite for Requirements Capture. UNU/IIST Document, 1995.
6. A. Blass and Y. Gurevich. *Evolving Algebras and Linear Time Hierarchy.* In B. Pehrson and I. Simon, editors, *Proc. of the IFIP 13th World Computer Congress 1994, Vol. I*, pp. 383–390. Elsevier, 1994.
7. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. *CSL'89.* Springer LNCS 440, 1990, 36–64.
8. E. Börger. A logical operational semantics for full Prolog. Part II: Built-in predicates for database manipulations. *MFCS'90. Mathematical Foundations of Computer Science* (B. Rovan, Ed.). Springer LNCS 452, 1990, 1–14.
9. E. Börger. Logic Programming: The Evolving Algebra Approach. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology and Foundations*, Elsevier, Amsterdam, 391–395.
10. E. Börger. Review of: E.W. Dijkstra & C.S. Scholten: *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989. *Science of Computer Programming* 23 (1994) 1–11 and *The Journal of Symbolic Logic* 59 (1994) 673–678
11. E. Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
12. E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog standardization document no. 58, NPl, Middlesex, 1990, pp. 92–114.
13. E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (The APE100 reverse engineering project). In *Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95).* See also BRICS NS-95-4, pp. 195–222, University of Aarhus, 1995.
14. E. Börger, G. Del Castillo, P. Glavan and D. Rosenzweig. Towards a mathematical specification of the APE100 architecture: The APESE model. In B. Pehrson and

I. Simon, editors, *Proc. of the IFIP 13th World Computer Congress 1994, Vol. I*, pp. 396–401. Elsevier, 1994.

15. E. Börger and B. Demoen. A framework to specify database update views for Prolog. In M. J. Maluszynski, editor, *PLILP'91.* LNCS 528, 1991, 147–158.
16. E. Börger and I. Đurđanović. Correctness of compiling Occam to Transputer code. BRICS NS-95-4, pp. 153–194 , University of Aarhus, 1995.
17. E. Börger, I. Đurđanović, and D. Rosenzweig. Occam: Specification and compiler correctness. Part I: The primary model. In E.-R. Olderog, editor, *Proc. of PRO-COMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.
18. E. Börger and U.Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology and Foundations*, Elsevier, Amsterdam, 402–409.
19. E. Börger and U.Glässer. Modelling and analysis of distributed and reactive systems using evolving algebras. BRICS NS–95–4, pp. 128–153, University of Aarhus.
20. E. Börger, U. Glässer and W. Mueller, The Semantics of Behavioral VHDL'93 Descriptions. In: *EURO-DAC'94 European Design Automation Conference with EURO-VHDL'94.* Proc. IEEE CS Press, Los Alamitos/CA, 1994, 500-505.
21. E. Börger, U. Glässer and W. Mueller. Formal definition of an abstract VHDL'93 simulator by EA–machines. In C. Delgado Kloos and Peter T. Breuer, editors, *Semantics of VHDL.* Kluwer, 1995.
22. E. Börger, Y. Gurevich and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
23. E. Börger, F.J. Lopez-Fraguas and M. Rodrigues-Artalejo. A Model for Mathematical Analysis of Functional Logic Programs and their Implementations. in: B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Vol. I, pp. 410-415, 1994, Elsevier. See the full version *Towards a Mathematical Specification of Narrowing Machines*, Report DIA 94/5, Dep. Informática y Automática, Universidad Complutense, Madrid, March 1994, pp.30.
24. E. Börger and S. Mazzanti. A correctness proof for pipelining in RISC architectures. Manuscript, 1995.
25. E. Börger and E. Riccobene. A Formal Specification of Parlog. In: *Semantics of Programming Languages and Model Theory* (M. Droste, Y. Gurevich, Eds.), Gordon and Breach, 1993, pp.1-42.
26. E. Börger and D. Rosenzweig. An analysis of Prolog database views and their uniform implementation. *Prolog. Paris Papers–2.* ISO/IEC JTC1 SC22 WG17 Prolog Standardization Report no.80, July 1991, pp. 87-130.
27. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1995.
28. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In L. C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications.* Elsevier Science B.V./North–Holland, 1995.
29. E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
30. E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. Springer LNCS 533, 1991, 67–79.
31. F.P. Bowen and M.G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer* 28(4):56–63, April 1995.

32. F.P. Bowen and M.G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software* 12(4):34–41, July 1995.
33. F. P. Brooks, No Silver Bullet—Essence and Accidents of Software Engineering. *IEEE Computer* 20, 1987, 10–19.
34. S.K. Debray and P. Mishra, Denotational and Operational Semantics for Prolog. In: *Journal of Logic Programming* 5, 1988, 61–91
35. A. de Bruin and E. P. de Vink, Continuation semantics for Prolog with cut. In: *Theory and practice of software engineering*, Springer LNCS 351, 1989, 178–192.
36. R. DeMillo, R. Lipton and A. Perlis, A social process and proofs of theorems and programs. In: *Comm. ACM* 22 (5), 271–280, 1979.
37. P. Deransart and G. Ferrand, An operational formal definition of Prolog.INRIA RR 763. See *Proc. 4th. Symposium on Logic Programming*, San Francisco 1987, 162—172 and *New Generation Computing*, 10.2, 1992, 121–171.
38. W.W. Gibbs, Software's Chronic Crisis, *Scientific American*, Sept 1994, 72–81.
39. Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pp. 1–57. CS Press, 1988.
40. Y. Gurevich. Logic Activities in Europe. in: ACM SIGACT NEWS, 1994.
41. Yuri Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
42. Yuri Gurevich. Platonism, Constructivism, and Computer Proofs vs. Proofs by Hand. In Bulletin of the EATCS, October 1995.
43. Y. Gurevich and J. Huggins. The semantics of the C programming language. Springer LNCS 702, 1993, 274–308 and LNCS 832, 1994, 334–336.
44. Y. Gurevich and J. Huggins. Equivalence is in the eye of the beholder. CSE TR 240-95, University of Michigan at Ann Arbor.
45. Y. Gurevich, J. Huggins and R. Mani. The Generalized Railroad Crossing Problem: An Evolving Algebra Based Solution. CSE-TR-230-95, University of Michigan.
46. Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
47. D. Harel, Biting the Silver Bullet. *IEEE Computer* 25, 1992, 8—20.
48. F.W. von Henke, Putting Software Technology to Work, In: K. Duncan and K. Krueger (Eds.) *IFIP 13th World Computer Congress 1994*, Vol. III, pp.345-350, 1994, Elsevier, Amsterdam.
49. J. Hennessy and D.A. Patterson. Computer Architecture: a Quantitative Approach. Morgan Kaufman Publisher, 1990.
50. H. Herbelin. Types for Proofs and Programs. Note on the ESPRIT Basic Research Action 6453 in: *Bulletin of the EATCS* vol.54, 1994, 105–116.
51. C.A.R. Hoare. An axiomatic basis for computer programming. In: *Comm. ACM* 12, pp. 576–580 and 583, 1969.
52. C.A.R. Hoare, ProCoS Working Group Meeting, Gentofte/DK 18.-20.1.1994.
53. C.A.R. Hoare, *Mathematical Models for Computing Science.* Manuscript, August 1994, 65 pp.
54. G.J. Holzmann, The Theory and Practice of a Formal Method: NewCoRe, in: B. Pehrson and I. Simon (Eds.), *Proc. of the IFIP 13th World Computer Congress 1994*, Vol.I, pp. 35–44, Elsevier, Amsterdam.
55. J. Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
56. *IEEE Standard VHDL Language Reference Manual—IEEE Std 1076–1993*, The Institute of Electrical and Electronics Engineering. New York, NY, USA, 1994.

57. N.D. Jones and A. Mycroft, Stepwise development of operational and denotational semantics for Prolog. In: *Proc. Int. Symp. on Logic Programming* 2/84, Atlantic City, IEEE, 289–298
58. G. Kahn, *Natural Semantics*, INRIA Rapport de Recherche No. 601, Février 1987.
59. B. Littlewood, L. Strigini, The Risks of Software, in: *Scientific American.* November 1992, p.38-43.
60. A. J. Mansfield, An Explanation of the Formal Definition of Prolog. NPL Report DITC 149/89, Teddington, 1989, p. 1–9.
61. R. Milner and M. Tofte, and R. Harper. *The definition of Standard ML.* Cambridge, Mass.: MIT Press, 1990.
62. R. Milner, M. Tofte, *Commentary on Standard ML.* MIT Press, 1990.
63. Peter D. Mosses. Denotational Semantics. In: Jan van Leeuwen (Ed.), *Handbook of TCS.* Elsevier 1990.
64. Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992.
65. N.North, A denotational definition of Prolog. NPL, Teddington, TR DITC 106/88.
66. S. Owre, J. Rushby, N. Shankar and F. von Henke, Formal Verification for Fault-tolerant Architectures: Prolegomena to the Design of PVS. In: *IEEE Transactions on Software Engineering*, vol. 21, no. 2, February 1995, pp.107–125.
67. D. L. Parnas. Information distribution aspects of design methodology. In C. V. Freiman, editor, *Proc. of IFIP Congress 1971, Volume 1: Foundations and Systems*, pp. 339–344. North-Holland, 1972.
68. D. L. Parnas, Software Aspects of Strategic Defense Systems. In: *Comm. ACM*, 28 (12), 1985, 1.326–1.335.
69. D. L. Parnas, Education for Computer Professionals. In: *IEEE Computer* 23, 1990, 17—22.
70. G. Plotkin, A structural approach to operational semantics, Internal Report, CS Department, Aarhus University, DAIMI FN-19
71. K. Popper, *Logik der Forschung.*1935.
72. S. Read, e-mail 25 Oct 1994, Compass Design Automation, Columbia/MD.
73. D. M. Russinoff, Specification and verification of gate-level VHDL models of synchronous and asynchronous circuits. In: *Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1995, pp. 411–459.
74. D. Scott, Outline of a Mathematical Theory of Computation, PRG-2, November 1970, Oxford Univ. Comp. Lab., Progr. Res. Group, pp.1–24
75. D. Scott, C. Strachey, Toward a Mathematical Semantics for Computer Languages. Proc. 21st Symp. Computers and Automata, Polyt.Inst. of Brooklyn, 1971, 19–46.
76. D.R. Smith, Classification Approach to Design, TR KES.U.93.4, Kestrel Institute, Palo Alto, CA, November 1993, pp.24
77. D.R. Smith and E.A. Parra, Transformational Approach to Transportation Scheduling, in: *Proc of the Eighth Knowledge-Based Software Engineering Conference*, IEEE Computer Society Press, September 1993, 60–68.
78. J. M. Wing, A Specifier's Introduction to Formal Methods. In: *IEEE Computer*, 23 (9), 1990, 8–24.
79. M. Wirsing. Handbook of Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B*, pages 675–788, Elsevier, 1990.
80. N. Wirth. Algorithms & Data Structures. Prentice-Hall, 1975.

This article was processed using the LaTeX macro package with LLNCS style