Evolving Algebras*

Yuri Gurevich†

March 5, 1998

Keyword Codes: D.2.2; F.1.1; F.3.2
Keywords: Software Engineering, Tools and Techniques;
Computation by Abstract Devices, Models of Computations;
Logics and Meanings of Programs, Semantics of Programming Languages

1 Introduction

I am delighted to open the first workshop devoted entirely to evolving algebras. For the sake of brevity, I shorten the term "evolving algebra" to "ealgebra" (pronounce e-algebra) or "EA".

The main goal of the evolving algebra methodology is to provide succinct and executable formal specifications for algorithms. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and real-time protocols, etc.. Some of my engineering colleagues say that formal methods had their chance and failed. This is not true. Some particular methods might have been over-advertised and failed to live up to the great promises. I believe in a great future of formal methods and in particular the EA method in designing, testing, verifying, documenting, maintaining, understanding and teaching algorithms.

There are so many things I would like to tell you. For the EA experts, I would like to survey the main ideas, to draw their attention to open problems and to congratulate them on their achievements. Keep up with the good work. We have come quite a way in so short a while.

For the skeptical engineer, I would like to demonstrate that we have a practical tool. Do not be turned off by terms like "algebra" and "algorithms". EA methodology works and it may help you. Tell us your algorithm-related problems. Give us challenges.

For the computer science theoretician, I would like to explain the EA approach from first principles and to show that this computation model bridges between complexity theory and formal methods.

But I cannot do all that in a talk. All I can do is a brief overview. I will not even

*IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations, eds. B. Pehrson and I. Simon, Elsevier, Amsterdam.

†Partially supported by ONR grant N00014-91-J-1861 and NSF grant CCR-92-04742. EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA. gurevich@umich.edu

define ealgebras here; see* the tutorial [Gurevich91], henceforth called the tutorial, or the Lipari guide [Gurevich94b]. I will not examine the complexity theoretic aspects of the EA model; this is the topic of the next lecture. And I will skip many issues likely to be covered in the survey lectures by Egon Börger and Dean Rosenzweig; in particular I will not say much about logic programming where the EA methodology was so successfully applied and developed.

2 The EA Thesis

When you use evolving algebras, you develop a strong feeling that ealgebras can model nicely any algorithmic process. The EA thesis can be seen as an attempt to express that feeling in terms of the EA computation model. The EA thesis was also the initial goal of the EA project.

2.1 Sequential Computations

At the beginning, there was Turing's thesis. Turing's machine model allowed Turing to formalize the notion of computable functions. But are there computing devices that allow one to formalize the notion of algorithm? The idea of the EA thesis is that every algorithm is essentially an ealgebra.

The EA thesis can be stated as follows: Every algorithm can be simulated *on its natural abstraction level* by an appropriate ealgebra in strict lock-step, so that the ealgebra needs only one step to simulates a step of the algorithm.

In the case of sequential computations, we are talking about the basic evolving algebras of the tutorial. The case of non-sequential computations is discussed below.

How can one argue for such a thesis? I see two ways: speculative philosophy and experimentation. A speculative argument for the EA thesis in the case of sequential algorithms is sketched in the tutorial. A fair amount of experimentation is mentioned in the tutorial as well. More experimentation — both sequential and non-sequential algorithms — has been done in the meantime as the annotated bibliography shows. In particular, evolving algebras have been constructed for various programming languages including C, C++, Occam and major versions of Prolog including its extensions by constraints, types and parallelism.

The confidence in the EA thesis that some of us feel comes not only from the amount of experimentation. You look at as many different kinds of algorithms as you can and you see that a strict lock-step simulation is always possible and after a while you run out of challenges. During a discussion in a 1993 Dagstuhl workshop, Andreas Blass remarked that the situation reminds him of the situation with formalizing mathematics within the standard set theory in the beginning of this century.

2.2 Non-Sequential Computations

The situation with non-sequential computations is different. A weaker form of the EA thesis holds: For every well defined class of algorithm, there is a generalization of the basic EA

*The references in this abstract refer to "Annotated Bibliography on Evolving Algebras" in "Specification and Validation Methods", ed. E. Börger, Oxford University Press, to appear. The book is related to a 1993 summer school on Lipari Island, Italy, and will be called the Lipari volume.

model that "captures" that class. The problem is that at the current stage of computer science we do not yet clearly understand, for example, what parallel computations are, what concurrent computations are, or what real-time computations are.

The evolving algebra methodology seems to be well adaptable to any of these computation paradigms. Evolving algebras with free variables and numerous daemons can capture the concurrent asynchronous computations of Occam (see [GurMos90] by Larry Moss and myself) or Parlog, a parallel version of Prolog (see [BoeRic93] by Egon Börger and Elvinia Riccobene).

By the way, Egon Börger, Igor Durdanović and Dean Rosenzweig returned recently to Occam and used advances in the EA methodology, like stepwise refinement and hiding syntax in a flowchart, to construct another ealgebra for Occam from which they were able to make a provably correct, smooth transition to the Transputer Instruction Set architecture. A more abstract study of concurrent evolving algebras has been undertaken by Paola Glavan and Dean Rosenzweig in [GlaRos93].

A group membership protocol involving real-time constraints has been formally specified and verified in [GurMan94] by my student Raghu Mani and myself. You will find a number of other instructive EA simulations of non-sequential algorithms in the annotated bibliography; some of them will be presented later in this workshop and some are found in the Lipari volume. As I said above, we are running out of obvious challenges.

In the Lipari guide, I attempt to extend the definition of ealgebras and provide a more solid foundation for the quickly growing EA field. Hopefully, ealgebras will be used not only to model, say, concurrent computations but also to understand what concurrent computations are.

3 Remarks

A survey talk is appropriate for reflection, but I philosophized on evolving algebras before in various articles, so here I will restrict myself to a few remarks.

My fellow logicians often sniff at semantics provided by computing devices because of the ad hocish character of devices and because devices often implement algorithms at a lower abstraction level. I believe that there isn't much ad hocish about ealgebras. The two main ingredients of the concept of evolving algebra belong to the well established tradition of mathematical logic. They are the notion of structure (as in first-order logic) and the notion of transition system. The novelty is in the way the two are put together. In my eyes, ealgebras were discovered, not invented.

Of course, a benefit of semantics provided by computing devices is that, as they say, you don't need a PhD in logic to understand it. EA specifications indeed are readable. I use them without problems in my classes whether my students are good in mathematics or not. The important thing is that they know programming.

Concerning the abstraction levels, the EA thesis mentions the natural abstraction level of the algorithm. Indeed, we are not interested in specifying an algorithm by implementing it. We are interested in specifying the algorithm on its natural abstraction level. Fortunately, the EA approach allows us to do that. Why is this so important? Well, let me raise another question. How do you know that your model faithfully reflects reality? This is a sticky question. You can't prove that a formal model fits informal reality. A good strategy is to

keep your model as close to the original as possible, so that it is obvious that the modeling is correct. The ability to specify algorithms on their natural abstraction levels allows one to use ealgebras for requirement specifications.

Let me stress the universality of the EA method. Many competing systems are good in specific domains. The EA methodology seems to work across domains. As you work with ealgebras you begin to see other computational models through the EA lens and they begin to look like special evolving algebras.

Another important issue is separation of concerns. Separating statics and dynamics (that is, static and dynamic aspects of computation) is at the heart of the EA approach. Statics plays a minor role in the definition of ealgebras and often is taken for granted; we concentrate attention on dynamics. That is why the definition of basic ealgebras is so simple. Separating statics and dynamics a little further, the syntax of the C programming language has been hidden into a flowchart in [GurHug93] by Jim Huggins and myself; this helped to simplify the ealgebra.

Of course, statics may be important and nontrivial; you cannot ignore it completely. Indeed, any EA interpreter will require not only the program of an ealgebra but also a presentation of the initial state.

By the way, what is static and what is dynamic may be in the eye of the beholder. "We suggest ... that many grammatical frameworks are static formalizations of intuitively dynamic ideas", write David Johnson and Larry Moss in their paper [JohMos93] whose name speaks for itself: Grammar Formalisms Viewed as Evolving Algebras.

Another relevant separation of concerns was suggested and discussed recently in [Gurevich94a]: First establish your claim (say, some property of an algorithm, e.g., correctness) mathematically and only then translate it into a formal system and check it on a computer if necessary. Computer verification of mathematical proofs should be a separate science or at least a separate branch of logic; I call it pedantics for the time being. I know that many disagree with me on that separation of concerns. One advantage of that separation is that you can use freely any necessary mathematics at the first stage when you are searching for a proof.

A few words on the method of stepwise refinement in the EA context. Because of the freedom to choose the appropriate abstraction level inherent in the EA approach, the method of stepwise refinement is extremely useful. The use of stepwise refinement was pioneered by Egon Börger and Dean Rosenzweig. An impressive hierarchy of abstraction levels and the simplicity of single refinement steps allowed them to establish the correctness of a general compilation schema of Prolog programs to the Warren Abstract Machine which underlies most of the current Prolog implementations and incorporates crucial optimization techniques [BoeRos94a].

Sometimes an algorithm comes with a built-in hierarchy of abstraction levels which guides the refinement process. A good example of this is the well known Kermit communication protocol specified and proved correct by my student Jim Huggins in [Huggins94]. But usually one has to analyze the algorithm in order to separate concerns and figure out the right abstraction levels. A relatively simple (comparative to [BoeRos94a]) and instructive example of stepwise refinement is found in [GurHug93] where Huggins and I constructed an ealgebra for the C programming language. Compare this ealgebra to an earlier ealgebra for Modula-2 in [Morris88] to see the benefits of stepwise refinement and hiding syntax in a flowchart.

A simple and instructive example of the use of multiple abstraction levels for verification purposes is found in [BoGuRo94].

4 Future Work

There are plenty of things to do. Here are some of them.

Explore new application areas and subareas (and in this way continue to test the broader EA thesis). Regarding hardware, I should mention the paper [BoGlMu93] by Egon Börger, Uwe Glässer and Wolfgang Müller on VHDL; Jim Huggins is looking at Verilog, another hardware description language. Speaking about software, ealgebras may for example be a good way to formalize active databases. And so on.

In connection with new application, let me mention testing of algorithms. Testing may seem an unlikely application for a specification method even if the specifications are executable. Why would you execute the specification if you can experiment with the algorithm itself? First, you may experiment with a specification before the algorithm is implemented or when no implementation of the algorithm is available. Second, various environmental forces may be represented by parameters in the specification. This may give you have more opportunities to inject faults. You may play with more operating systems than you have around. You may play with physical parameters like gravitation. Third, you may combine testing and verification.

Develop a toolkit for writing and using ealgebras. We have an interpreter for sequential ealgebras in Michigan which is being upgraded; in particular we would like to be able to interleave more elegantly multi-agent ealgebras. We have also a partial evaluator [GurHug94]. Much more is needed.

Use the EA computation model for complexity investigation. Such investigation started with a paper [BlaGur94] by Andreas Blass and myself.

Focusing on the theory of ealgebras, develop a model theory of ealgebras. Isolate the principal refinement notions. What are the main ways to combine ealgebras? The investigation of the second problem started in [GlaRos93]. Develop the proof theory to go with ealgebras. Temporal logic may be relevant. A step in that direction is made in [Poetzsch94]. By the way, it is customary in temporal logic to start with a proof system and analyze the models. It may be fruitful to explore the other direction. Start with a class of evolving algebras and see what proof system is appropriate to it.

Acknowledgment

I thank Egon Börger, Dean Rosenzweig and Jim Huggins for help with this abstract; portions of the text are direct quotations from Egon Börger.