From Implementations to a General Concept of Evolvable Machines

Lukáš Sekanina

Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
e-mail: sekanina@fit.vutbr.cz phone: +420 541141215

Abstract. This paper introduces a little bit different view on evolvable computational machines than it is usually presented. Evolvable machines are considered as mathematical machines. Traditional tools of theoretical computer science are then employed in order to obtain qualitatively new understanding the evolvable machines. In particular the questions related to formal definition and computational power are discussed. The concept is proposed in framework of traditional software and hardware implementations of evolvable machines.

1 Introduction

From a machine learning perspective, *genetic programming* is very often considered as a technique allowing the automatic design of *computational machines* [8, 1]. In the most popular approach, a *program* is evolved. In order to create the computational machine, the evolved program is uploaded into a universally programmable computer and the program is executed. In another approach, referred to as *Cartesian genetic programming* (CGP) [12], digital circuits are evolved directly. The resulting circuits can be considered as computational machines too.

It could theoretically be possible to evolve using genetic programming whatever computational machine based on an arbitrary model of computation. However only relatively simple computational machines (such as small programs [8], circuits [11], cellular automata [15] or Turing machines [20]) were designed successfully because of scalability problems.

Genetic programming also enables the design of *adaptive computational machines*. In this case, opposite to the evolutionary computational machines design, the evolutionary algorithm is inherent part of a target (e.g. embedded) system. The evolutionary algorithm has to autonomously produce computational machines according to requirements represented via dynamic fitness function which reflects a changing environment. Adaptive computational machines, which utilize the evolutionary approach, are known as *evolvable (computational) machines*.

Computer engineering concerns itself with implementation of a given computational machine in reality (in software, hardware etc.). On the other hand computer science usually defines theoretical models of computation for these computational machines. It is advantageous to have such models because then various methods might be applied to investigate machines' properties, limits

and classes. Furthermore, computer engineers can prepare an effective design strategy easily if suitable theoretical models exist.

This paper compares three viewpoints on computational machines designed using evolutionary methods: a software implementation viewpoint, a hardware implementation viewpoint and a formal approach viewpoint. While software implementations are well developed in the genetic programming community and hardware implementations are realized in the *evolvable hardware* community, a formal approach seems to be quite overlooked. Hence this paper introduces some formal definitions as a potential way in which software as well as hardware implementations of evolvable machines can systematically be integrated, studied and understood. In particular we emphasize a *computational scenario* of evolvable machines that differs from conventional computational scenarios.

The approach used in this paper represents a high-level insight to the problem. The method can show what we really do in the evolutionary machine design and how evolvable machines perform the computation. In particular it becomes important for (semi)automatic design tools in which we are looking for a universal description suitable for specification of evolvable systems.

The paper is organized as follows. Section 2 deals with software, hardware and formal viewpoints related to the evolutionary computational machines design. A formal definition of a dynamic environment and computational power in dynamic environment are discussed in Section 3. Section 4 provides summary of the obtained results. Finally conclusions are given in Section 5.

2 Evolutionary computational machines design

In case of the evolutionary computational machine design, genetic programming is only to assist (or "replace") the designer and thus a genetic programming system is utilized only in the design phase. Only a single fitness function is usually constructed. We are interested in *innovative* designs. The resulting machine is much more important than the design method applied since only the resulting machine is interesting for potential customers.

2.1 A software viewpoint

Let us recall a programmer's perspective. In genetic programming, (variable length) chromosomes represent either trees or machine language instructions. All candidate programs are executed in order to obtain their fitness values. Every new population is formed using genetic operators working over the chromosomes. The algorithm is terminated when a perfect solution is produced or a pre-defined number of populations are generated.

From a theoretical computer science viewpoint, tree representation can be modeled using expressions of λ-calculus. Machine language instructions directly represent programs of the RAM (Random Access Machine) computer model [7]. Hence a target computational machine looks like a *universal computer*; the evolutionary algorithm is only to supply a program which is uploaded into a memory of the computer and which is executed.

In another approach, genetic programming was employed to construct cellular automata rules. (Non-uniform) *cellular automaton* is a d-dimensional grid of finite automata (known as *cells*) that operate according to their *local transition functions* (also known as *rules*). The cells work synchronously—a new state of every cell is calculated from its previous state and the previous states of cell's "neighbors" in each time step. Local transition functions can also be defined as syntactic trees. These trees can be evolved using genetic programming. Koza [8], Ferreira [6], Sipper [15] and others evolved a number of high-quality rules in tasks where human approach has led to poor results.

The previous paragraph has demonstrated that genetic programming can be applied to design not only a program for a universal computer, but also to design a *component* of computational machine definition (i.e. cellular automata rules in our example). Note that such a computer model (i.e. cellular automaton) differs from traditional universally programmable computers (e.g. von Neumann's computer organization) substantially. However the cellular automaton is typically simulated on a machine of traditional organization.

Finally, genetic programming in cooperation with a software simulator of practically whatever behavior might be able to discover innovative designs. That is clearly demonstrated, for instance, on the automatic design of analog circuits using genetic programming and an analog circuit simulator [16].

2.2 A hardware viewpoint

In case of hardware we have to distinguish hardware implementations of genetic programming and evolvable hardware.

In order to speed up the evolution, parallel genetic programming implementations (employing hundreds processors like in [16]) or hardware implementations are constructed. As an example of conventional hardware approach, Martin implemented a simple variant of parallel genetic programming in an FPGA (Field Programmable Gate Array) [10]. The programs, which can be evolved directly in an FPGA, consisted of a few types of instructions and were evaluated in p small "processors" constructed and distributed inside the FPGA. We would like to mention that Martin's approach couldn't be classified as evolvable hardware because no circuits were actually evolved.

In case of evolvable hardware [23], chromosomes are considered as circuit configuration bitstreams which are used to configure a *reconfigurable circuit*. Every candidate configuration is evaluated either in a circuit simulator (i.e. *extrinsic evolution*) or in a physical hardware (i.e. *intrinsic evolution*). It is crucial to distinguish these approaches. In case of so-called *unconstrained evolution* discovered by A. Thompson [19] (it is intrinsic evolution too), the evolution is free to exploit physical properties of a chip as well as other environmental characteristics (like temperature during experiment) for building the resulting circuit. Because of the "side effects" of the unconstrained evolution, it is possible to obtain two *different* fitness values (related to the same configuration) when formally the same fitness calculation process is carried out in a software circuit simulator and then in a physical reconfigurable circuit. Note that the software simulator and

the constrained evolution in hardware always yield the same fitness value for the same configuration. Although unconstrained evolution causes a number of problems in practice, it is a tool how to discover really innovative circuits [19].

Miller's CGP is probably one of the most developed models of FPGA-based genetic programming [12]. In CGP a reconfigurable circuit is modeled as an array of n_c (columns) $\times$ n_r (rows) programmable nodes (see Fig. 1). The number of circuit inputs n_i and outputs n_o are fixed. A node's input can be connected to the output of some element in the previous columns or to some of circuit inputs. A node has up to n_n inputs and a single output. Every node is programmed to implement one of n_f functions defined in set F. Finally, circuit interconnectivity is defined by *levels back parameter* L, which determines how many previous columns of nodes may have their outputs connected to a node in the current column. For example, if $L = 1$, only neighboring columns may be connected; if $L = n_c$, the full connectivity is enabled. Nodes in the same column are not allowed to be connected to each other, and any node may be either connected or disconnected. Circuit outputs can be taken from any node output. A configuration of every node is represented in a chromosome using $n_n + 1$ integer values, which define connection of node's inputs, and a function realized in the node. While chromosomes are of fixed length, phenotypes are of variable length.

From a theoretical computer science viewpoint, Boolean circuits represent a model of computation too. While Turing machine and RAM are examples of uniform and infinite computer models (in sense that each particular computer can process inputs of an arbitrary size), a single Boolean circuit computes only a single Boolean function. In order to make a universal computer model of the same power as Turing machines out of Boolean circuits, uniformly designed families of Boolean circuits have to be considered [7].

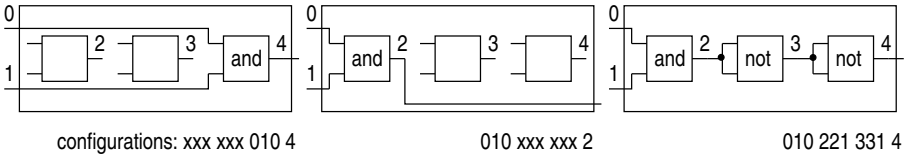

Fig. 1. Three different phenotypes with identical behavior. Parameters of CGP: $n_c = 3, n_r = 1, n_i = 2, n_o = 1, n_n = 2, n_f = 2, F = \{\text{and, not}\}$.

2.3 A formal viewpoint

Up to now we mentioned various models of computational process in this paper. These models are formally defined in many textbooks, e.g. [7]. Genetic programming was employed in order to design computational machines based on the discussed models. Let us try to formalize this well-known concept.

Any evolutionary algorithm E (and thus genetic programming too) can be considered as a stochastic population-based search algorithm. For instance, Surry

formally defines a stochastic search algorithm as recursive function Ψ which, when given a sequence of points from the representation space and the corresponding sequence of fitness function values for those points, generates a new point in the representation space [17]. The definition assumes the existence of genotype-phenotype mapping (sometimes referred to as a *growth function*) which is important from (our) machine design viewpoint.

Classical evolutionary algorithms utilize fitness function of the form $\Phi : \mathcal{C} \to \mathbb{R}$ where $\mathcal{C}$ denotes a set of chromosomes (representation space). Conceptually, chromosomes are not evaluated. Only machines (more precisely, behaviors of these machines) are and can be evaluated. Hence it is reasonable to define a set of machines $\mathcal{M}$ which can be constructed from chromosomes for a given problem domain using *growth function* $g : \mathcal{C} \to \mathcal{M}$. It is supposed that g is surjective. The machines are then evaluated using "machine" fitness function $f : \mathcal{M} \to \mathbb{R}$. Finally, Φ is expressed as composition $\Phi = f \circ g$.

In order to illustrate the proposed formal approach, assume that a definition of cellular automaton A consists of four components $A = (c_1, c_2, c_3, R)$, where $R = \{0,1\}^n$ denotes cellular automaton rules encoded as n-bit string. Then a set of all machines that can be evolved corresponds to a 2^n-element set of cellular automata of the form

$$\mathcal{M} = \{(c_1, c_2, c_3, R) \mid c_1 = k_1, c_2 = k_2, c_3 = k_3\}$$

where k_1, k_2, and k_3 are invariable objects. Genotype-phenotype mapping is constructed as $g : \{0,1\}^n \to \mathcal{M}$. Cellular automata are then evaluated using f.

According to the previous analysis, we can summarize that any evolutionary design of computational machines is fully defined in terms:
E – evolutionary algorithm employed (with fitness function $\Phi : \mathcal{C} \to \mathbb{R}$);
$\mathcal{M}$ – a set of possible machines which can be created;
g – a surjective growth function of the form $g : \mathcal{C} \to \mathcal{M}$;
f – a "machine" fitness function of the form $f : \mathcal{M} \to \mathbb{R}$;
$\Phi = f \circ g$

We can see that only four mathematical components $(E, \mathcal{M}, g, f)$ are needed in the definition. Note that f is a problem specific function, g can cover any type of constructional process (e.g. such as a development of phenotypes from genotypes described and implemented initially in Dawkins's biomorphs [5]) and $\mathcal{M}$ is defined implicitly or explicitly before the evolution is executed. Looking via the proposed definition all evolvable machines operate in the *same way*. We could perhaps say that they are (iso)morphic each other in some sense. Some other properties have been investigated in [13].

2.4 Relation of the approaches

When software, hardware and theoretical views (together with underlying models) are formulated, we can investigate their relations and mutual translatability. However, if we accept the proposed definition as a general paradigm, we could easily get into troubles.

Theoretically, $\mathcal{M}$ is infinite but enumerable set. Practically, $\mathcal{M}$ is always finite because of finite resources of any implementable universal computer or any electronic circuit. If we accept that $\mathcal{M}$ is a finite set then software implementations can always correspond to the proposed formal model. Furthermore, all Martin's style hardware implementations of genetic programming always correspond with the proposed formal model too.

However it is *not* case of evolvable hardware. There is not any problem in the constrained evolution: the number of possible distinguishable phenotypes (circuits) is given by the number of possible different circuit configurations. For instance, we derived using a simple combinatorial analysis that CGP enables

$$P = n_f^{n_c n_r}(n_i + n_c n_r)^{n_o}(n_i + L n_r)^{n_n n_r (n_c - L)} \prod_{j=0}^{L-1}(n_i + j n_r)^{n_r n_n}$$

different configurations and so circuits to emerge, i.e. $|\mathcal{M}| = P$. Note that the circuits, which perform the same logical behavior, are treated as different phenotypes because they are not placed in physically identical nodes as seen in Fig. 1.

Although the unconstrained approach supports P configurations too, the number of different circuit behaviors which can appear is P', but P' is greater than P. It is due "analog" nature of the evolved circuits and various side effects (like variability of material characteristics of "the same" chips or changing temperature during evolution). Hence it is practically impossible to specify $\mathcal{M}$ in this case. Hypothetically, if one would like to specify $\mathcal{M}$, (s)he has to create a detailed model for a given piece of silicon (chip), to explore all possible configurations and to take in account all working conditions (so-called *operational envelope* in [18]).

This problem can also be interpreted in the following way: f is not a function, but f is a relation. Hence two (or more) different fitness values might be assigned to a single machine in the formally same fitness calculation process (see Fig. 2CD). The situation traditionally leads in theoretical computer science to definition of *non-deterministic* variant of the computational model. In our case, we can speak about *non-deterministic evolvable machines.*

In practical designs of evolvable hardware, growth function g has always been a bijection. This means that a single configuration bitstream encodes a single physical circuit. In case of software, redundancy of encoding is sometimes introduced as shown in Fig. 2BD. It yields genotype *neutrality.* Note that the mappings in Fig. 2A exactly correspond with the circuits depicted in Fig. 1.

3 Evolvable machines

Evolvable machines operating in a dynamic environment (e.g. with a changing fitness function in scheduling tasks [2] or in dynamic hashing [4]) introduce in embedded systems the following situation: Problem representation and genetic operators (and architecture of a reconfigurable circuit in case of evolvable

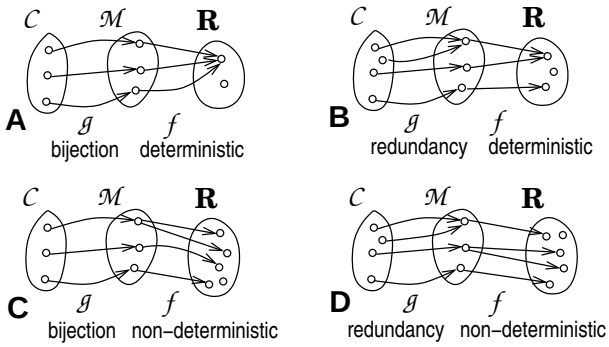

Fig. 2. Properties of g and f in four different types of evolvable machines.

hardware) remain unchanged and cannot be altered when a new fitness function is specified. Formally, $E, \mathcal{M}$, and g are *invariable*. This is unpleasant from a performance (No Free Lunch theorem [22]) point of view. Hence it is important to define a problem domain carefully in order to outperform random search for a reasonable class of fitness functions.

3.1 Formal approach

In order to provide a simple formal framework for machine evolution in a dynamic environment, the following specification of *machine context* (environment) is proposed. The machine context is considered as a set of fitness functions (we can call them *context functions*) together with a mechanism of transition between them. Context functions are changed in discrete time points modeled as natural numbers $\mathbb{N} = \{1, 2, \ldots\}$. A set of all mappings from $\mathcal{M}$ into $\mathbb{R}$ will be denoted $\mathbb{R}^{\mathcal{M}}$. Formally, machine context is defined in terms:
$\Gamma \subseteq \mathbb{R}^{\mathcal{M}}$ – a set of context functions ($\varphi_i \in \Gamma$ specifies fitness function in environment i).
$\varphi_0 \in \Gamma$ – an initial context function.
$\varepsilon : \Gamma \times \mathbb{N} \to \Gamma$ – a relation that determines successive context function.

The following example illustrates the proposed formal definitions: Consider a real-time adaptive image filtration realized using an evolvable computational machine. The evolutionary algorithm is applied to automatically design image filters. The filters should suppress a noise presented in images taken from a camera. The evolvable machine in fact produces a (potentially endless) sequence of filters (i.e. machines from $\mathcal{M}$), each for more recent type of noise. Assume, for instance, that the machine is a part of a traffic control system. Type of noise is reflected in context function φ_i and hence φ_i depends on daytime, weather and other factors. A change of type of noise (i.e. a change of context function described by ε) is unpredictable as weather is.

3.2 Computational power

The example has demonstrated that if we put together the definition from Section 2.3 with the definition of machine context (i.e. $f = \varphi_i$), we obtain a formal definition of an arbitrary evolvable machine working in a dynamic environment.

It was shown in an emerging field—hypercomputation (or super-Turing computation) [3, 9, 21]—that some theoretical models and modern computational systems do not share the computational scenario of a standard Turing machine and hence they can not be simulated on Turing machines. Let us note that a standard Turing machine supposes that input data are available before a computation is started (no interaction is enabled later) and that a uniform algorithm (which is invariable and never changed during execution) processes them.

Nevertheless van Leeuwen and Wiedermann have shown that such computations may be realized by an *interactive Turing machines with advice* [9]. Interactive Turing machine with advice is a classical Turing machine endowed with three important features: advice function (it is a weaker type of oracle [7]), interaction and infinity of operation. The same authors have proposed the following extension of the Church-Turing thesis [9]: *Any (non-uniform interactive) computation can be described in terms of interactive Turing machines with advice.*

For example, a model of Internet possesses the same computational power as an interactive Turing machine with advice. However, only in the case that its life-span is *infinite*. Otherwise, the computation is finite and remains in the scope of standard Turing machine and the standard Church-Turing thesis [9].

We can observe that evolvable computational machines operating in a dynamic environment show simultaneous non-uniformity of computation, interaction with an environment, and infinity of operations. Furthermore, relation ε is in general *uncomputable*. It was proven that computational power of an evolvable computational machine operating in a dynamic environment is equivalent with the computational power of an interactive Turing machine with advice, however, only in the case that evolutionary algorithm is tuned to the problem [14].

Also physical implementations of evolvable machines (for instance, the evolvable image filter mentioned in the previous subsection), Internet and other similar devices are very interesting from a computational viewpoint. At each time point they have *finite* description. However, when one observe their computation in time, they represent *infinite* sequences of reactive devices computing non-uniformly. The "evolution" of machine's behavior is supposed to be endless. In fact it means that they offer an example of real devices (physical implementations!) that can perform computation that no single Turing machine (without oracle) can. Nevertheless they can be modeled *a posteriori* by interactive Turing machine with advice [9].

4 Summary

We can now summarize the most important results of the paper.

- It is reasonable and useful to support various approaches in order to describe and understand the evolvable machines.

- General formal definition of an evolvable machine can be introduced.
- One-to-one mapping exists between the formal definition, software and "conventional" hardware implementations of any evolvable machine.
- If a hardware implementation of an evolvable machine is realized using the unconstrained evolution then it is practically impossible to define set of machines $\mathcal{M}$.
- The unconstrained evolution can be modeled using a non-deterministic evolvable machine.
- Evolvable machines operating in a dynamic environment exhibit a super-Turing computational power.

Some open problems for future research are as follows:

- How (iso)morphism of evolvable machines should be defined.
- How classes of evolvable machines (comprising machines of the same computational power) should be defined.
- How to reflect all the proposed ideas in practical design tools.

5 Conclusions

The paper presents an original contribution to the theory of evolvable machines. We showed that theoretical computer science has methods for effective description of evolvable machines. Application of these methods allowed interesting results to appear. We are going to go on in the proposed approach in future research. The ultimate goal is (1) to establish a widely acceptable mathematical theory of evolvable computational machines and (2) to apply the results in practical designs.

Acknowledgment

The research was performed with the financial support of the Grant Agency of the Czech Republic under No. 102/03/P004 and 102/01/1531, and from the Research Intention No. CEZ: J22/98:262200012.

References

1. Banzhaf, W., Nordin, P., Keller R., E., Francone, F., D.: Genetic Programming – An Introduction. Morgan Kaufmann Publishers, 1998
2. Branke, J.: Evolutionary Approaches to Dynamic Optimization Problems – Updated Survey. In: Proc. of the GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems, 2001, p. 27–30
3. Copeland, B., J., Sylvan, R.: Beyond the Universal Turing Machine. Australasian Journal of Philosophy. 77, 1999, p. 46–66
4. Damiani, E., Liberali, V., Tettamanzi, A.: Dynamic Optimisation of Non-linear Feed-Forward Circuits. In: Proc. of the 3rd Conference on Evolvable Systems: From Biology to Hardware ICES'00, LNCS 1801, Springer-Verlag, 2000, p. 41–50

5. Dawkins, R.: The Blind Watchmaker. Penquin Books, London, 1991
6. Ferreira, C.: Discovery of the Boolean Functions to the Best Density-Classification Rules Using Gene Expression Programming. In: Proc. of the 5th European Conference on Genetic Programming EuroGP2002, LNCS 2278, Springer, 2002, p. 50–59
7. Gruska, J.: Foundations of Computing. International Thomson Publishing Computer Press, 1997
8. Koza, J., R., Bennett III., F., H., Andre, D., Keane, M., A.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, 1999
9. van Leeuwen, J., Wiedermann, J.: The Turing Machine Paradigm in Contemporary Computing. In: Mathematics Unlimited – 2001 and Beyond, Springer-Verlag, 2001, p. 1139–1155
10. Martin, P.: A Hardware Implementation of a Genetic Programming System Using FPGAs and Handel-C. Genetic Programming and Evolvable Machines, Vol. 2(4), 2001, p. 317–343
11. Miller, J., Job, D., Vassilev, V.: Principles in the Evolutionary Design of Digital Circuits – Part I. Genetic Programming and Evolvable Machines, Vol. 1(1), Kluwer Academic Publisher, 2000, p. 8–35
12. Miller, J., Thomson, P.: Cartesian Genetic Programming. In: Proc. of the 3rd European Conference on Genetic Programming EuroGP2000, LNCS 1802, Springer-Verlag, 2000, p. 121–132
13. Sekanina, L.: Evolvable Computational Machines: Formal Approach. In: Intelligent Technologies – Theory and Applications, 2nd Euro-International Symposium on Computational Intelligence, 2002, IOS Press Amsterodam 2002, p. 166–172
14. Sekanina, L.: Component Approach to Evolvable Systems. PhD thesis, Brno University of Technology, 2002, p. 132
15. Sipper, M.: Evolution of Parallel Cellular Machines – The Cellular Programming Approach. Springer-Verlag Berlin, 1997
16. Streeter, M., J., Keane, M., A., Koza, J. R.: Routine Duplication of Post-2000 Patented Inventions by Means of Genetic Programming. In: Proc. of the 5th European Conference on Genetic Programming EuroGP2002, LNCS 2278, Springer-Verlag, 2002, p. 26–36
17. Surry, P., D.: A Prescriptive Formalism for Constructing Domain-specific Evolutionary Algorithms. PhD thesis, University of Edinburgh, 1998
18. Thompson, A.: On the Automatic Design of Robust Electronics Through Artificial Evolution. In: Proc. of the 2nd Conference on Evolvable Systems: From Biology to Hardware ICES'98, LNCS 1478, Springer-Verlag, 1998, p. 13–35
19. Thompson, A., Layzell, P., Zebulum, S.: Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution. IEEE Transactions on Evolutionary Computation, Vol. 3(3), 1999, p. 167–196
20. Vallejo, E., E., Ramos, F.: Evolving Turing Machines for Biosequence Recognition and Analysis. In: Proc. of the 4th European Conference on Genetic Programming EuroGP2001, LNCS 2038, Springer-Verlag, 2001, pp. 192–203
21. Wegner, P., Goldin, D.: Computation Beyond Turing Machines. Communications of the ACM. Accepted (2002) `http://www.cs.brown.edu/people/pw/home.html`
22. Wolpert, D., H., Macready, W., G.: No Free Lunch Theorems for Optimization. IEEE Transactions on Evolutionary Computation, Vol. 1(1), 1997, p. 67–82
23. Yao, X., Higuchi, T.: Promises and Challenges of Evolvable Hardware. In: Proc. of the 1st International Conference on Evolvable Systems: From Biology to Hardware ICES'96, LNCS 1259, Springer-Verlag, Berlin, 1997, p. 55–78