Evolving Distributed Algorithms with Genetic Programming

Thomas Weise, *Member, IEEE*, Ke Tang, *Member, IEEE*

***Abstract*—In this article, we evaluate the applicability of Genetic Programming (GP) for the evolution of distributed algorithms. We carry out a large-scale experimental study in which we tackle three well-known problems from distributed computing with six different program representations. For this purpose, we first define a simulation environment in which phenomena such as asynchronous computation at changing speed and messages taking over each other, i.e., out-of-order message delivery, occur with high probability. Second, we define extensions and adaptations of established GP approaches (such as tree-based and Linear Genetic Programming) in order to make them suitable for representing distributed algorithms. Third, we introduce novel rule-based Genetic Programming methods designed especially with the characteristic difficulties of evolving algorithms (such as epistasis) in mind. Based on our extensive experimental study of these approaches, we conclude that GP is indeed a viable method for evolving non-trivial, deterministic, non-approximative distributed algorithms. Furthermore, one of the two rule-based approaches is shown to exhibit superior performance in most of the tasks and thus can be considered as an interesting idea also for other problem domains.**

***Index Terms*—Genetic Programming, SGP, LGP, Rule-based Genetic Programming, Fraglets, Distributed Algorithms, Election, Critical Section, Mutual Exclusion, GCD**

I. Introduction

COMPUTER networks are one of the most vital components in today's economy and society on the whole. Virtually every computer is either already part of a distributed system or may become one in future [2]. While many internet technologies, protocols, and applications grew into maturity and have widely been researched, new forms of networks and distributed computing have emerged during the recent years. Amongst them, we can find large-scale wireless networks, mobile ad hoc networks (MANETS) [3], sensor networks [4], wireless sensor networks [5], Smart Home environments [6], Ubiquitous Computing [7], and experimental ideas like Amorphous Computing [8].

The devices interacting in such networks often have scarce resources, such as little memory, low processing power, and a limited battery capacity. Additionally to these restrictions, new requirements such as self-adaptation and self-organization arise. A software architect designing a sensor network, for instance, must not only be concerned with functional aspects like data aggregation [9] associated with the purpose of the system, but also consider non-functional criteria such as energy consumption throughout the complete software engineering process. This results in the development of new algorithms for traditional problems such as election [10–12], often with changed requirements, priorities, and framework conditions.

A. Motivation

The design of a program for a distributed system is the transformation of an expected behavior of a network as a whole to a program which must be executed on each of its nodes in order to achieve this behavior. In other words, a (known) globally beneficial target configuration is translated into local rules. Investigating new methods for this purpose and evaluating their utility can be considered as a good idea in the current time.

In this context, many researchers are drawing inspiration from biological systems in order to face the new challenges of the emerging network technologies [13]. Natural swarming behaviors [14], for example, have been adapted as a paradigm for manually designing routing algorithms [15].

At second glance, we find that the ability of organisms to form swarms is the result of evolution. Over millions of years, this process transformed a beneficial global system configuration to local behavior rules. This way, many efficient natural distributed systems were created along with all local interaction patterns needed to maintain them.

Existing natural systems often have desirable features such as resilience and scalability. Facets of these systems are thus often *emulated* in order to equip distributed systems with similar features, as is the case in Swarm Intelligence-based routing. By imitating the process which *created* these facets instead, we consider a more basic approach in this article: the *evolution* of complete distributed algorithms with Genetic Programming.

B. Evolving Distributed Algorithms

Evolutionary Algorithms (EAs) [16–18] are metaheuristics which are inspired by natural evolution in order to solve complex problems. With the foundations laid in the 50s of the past century [19], they have successfully been applied to an incredibly wide range of problem domains. The set

T. Weise (tweise@ustc.edu.cn) and K. Tang (ketang@ustc.edu.cn) are with the Nature Inspired Computation and Applications Laboratory (NICAL), School of Computer Science and Technology, University of Science and Technology of China; Hefei, Anhui, China, 230027.

Manuscript received July 30, 2009.

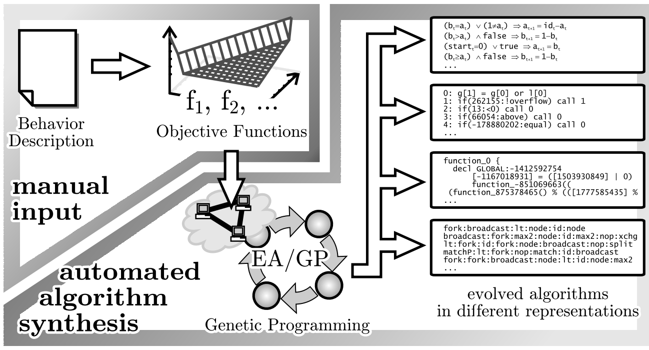


Fig. 1. Evolving distributed algorithms with Genetic Programming [22].

of Evolutionary Algorithms for deriving programs is called Genetic Programming (GP) [18, 20, 21]. In this article, we outline our results on utilizing GP in order to automate the transformation from global to local behavior of distributed systems, i. e., the design of distributed algorithms [22]. We specifically describe the synthesis of algorithms for systems with asynchronous computation and heterogeneous processing speeds of the nodes and aim for non-approximative, discrete computations.

Fig. 1 illustrates this new design process. It starts with a specification of the desired (global) system behavior. Objective functions, i. e., optimization criteria, which rate "how close" the behavior of an evolved program comes to the target behavior are derived manually from the specification. They fuel the Genetic Programming process which begins with an initial population consisting of randomly created programs. These programs are refined iteratively. In every iteration (generation), each program is executed on all nodes of a simulated distributed system. The objective functions access the simulations and evaluate the observed behaviors. Their results are then the basis of a subsequent selection step during which only best candidate programs are preserved. These programs become subject to the reproduction operations (mutation and crossover) and with their offspring, the cycle starts again. Step by step, distributed algorithms emerge which fulfill the initial behavior specification. Finally, the software engineer receives the set of candidate solutions which satisfy the objective functions best.

C. Contribution

With this article, we make the following contributions to the area of applying Genetic Programming to distributed computing.

1) By thoroughly evaluating the approach proposed in the previous section and based on our preliminary work [23–26], we answer the question whether Genetic Programming is an appropriate method for synthesizing distributed algorithms positively.
2) We define a model for distributed systems in which phenomena such as parallelism, messages which take over each other, and asynchrony occur with high probability. We implement this model in a high-speed simulation environment suitable for Genetic Programming and use this environment to
3) extend our previous work [23] with a large-scale experimental study by tackling three well-known problems from distributed systems with multi-objective Genetic Programming using six different program representations. Our work is the first application of GP to two of the three problems (the distributed GCD and the critical section problem). Also, as far as we know, no study has yet been performed which involves multiple different program representations for synthesizing distributed algorithms by other researchers. As we will show in Section II, only very little work has been contributed to this area by the evolutionary computation community in general. This holds especially for close-to-reality scenarios with, for instance, realistic parallelism and message latency assumptions, as used in our experiments.
4) Two of the test program representations combine a rule-based syntax with an implicit form of "transactional" memory. They have been developed by us especially with the characteristic difficulties of the problem area in mind [24]. We show that one of them, the Extended Rule-based Genetic Programming method, exhibits superior performance in comparison with the other approaches which makes it interesting for other application areas as well.
5) We furthermore define extensions to three of the other representations (two standard GP and one linear GP approach) which make them useful for evolving distributed algorithms, such as two forms of memory and a support for concurrency.

The remainder of this article is structured as follows. First, we provide a discussion of work related to our approach in Section II. In Section III, a clear definition of both, the model of the distributed systems we wish to evolve programs for and the simulations in which we test them is given. The idea of synthesizing distributed algorithms with GP is very intriguing. Yet, there is a set of features which make actual success in this class of optimization problems less likely. We investigate these features in Section IV, where we also propose countermeasures against them. Our novel program representations, developed especially with respect to this difficulty analysis, are introduced in Section V. Generally, programs can be defined in a variety of different ways and in Section VI, we also introduce adaptations of four other, well-known concepts to our problem domain. In this section we describe the results of three large-scale experiment series: evolving leader election algorithms (Section VI-C), distributed mutual exclusion methods (Section VI-D), and algorithms for computing the greatest common divisor of a set of numbers (Section VI-E). We draw conclusions from the gathered experiences, summarize our findings, and give pointers for future work in Section VII.

II. Related Work

Many aspects of distributed systems are configurable or depend on parameter settings, such as the topology, security, and routing. Using metaheuristic optimization algorithms for this purpose has huge potential.

A. Metaheuristics for Distributed Computing

And indeed, this potential is widely utilized. The study by Sinclair [27] from 1999 reported that more than 120 papers had been published on work which employed Evolutionary Computation for routing, optimizing network topologies, dimensioning, node placement, and other system design decisions. The comprehensive master's thesis by Kampstra [28] from 2005 builds on this study, classifies over 400 papers, and identifies the area as the most active research field in Evolutionary Computation at that time. In the year 2000 alone, two books ([29] and [30]) have been published on the topic of evolutionary telecommunications and further summary papers appeared [31, 32]. The recent studies [33–36] as well as the special journal issue [37] and the high number of papers published every year show that the interest in applying metaheuristic optimization techniques in this problem domain has by no means decreased.

Most of the research on the application of optimization methods to distributed systems focuses on routing (with, for instance, Ant Colony Optimization [15] or Genetic Algorithms (GAs) [38] or offline GP [39]), network design problems (with GAs [40, 41], GP [42], Estimation of Distribution Algorithms (EDAs) [43], or Tabu Search [44], for example), security issues (e.g., with linear GP [45]), or system configuration [46] (using, for instance, Hill Climbing and Simulated Annealing [46], EAs [47], or GP [48]).

An especially interesting application in this field is the optimal configuration of software parameters of protocols. Here, the work by Tate et al. [47] on the sensor network tuning problem may serve as a notable example: The parameters of a TTL-bounded gossiping protocol are optimized with respect to five criteria concerning performance, reliability, and efficiency. Tate et al. [47] carried out experiments with both, traditional design of experiment (DoE) methods and an EA to solve this multi-objective problem. They showed that both methods have specific merits and that EAs can outperform DoE in some quality objectives while losing in others.

Regardless of the success in the area of algorithm parameter *adaptation*, only few researchers have considered the *synthesis* of distributed algorithms and even less work has been contributed to this domain under close-to-reality conditions such as asynchronous communication and asynchronous process execution on the nodes of the distributed systems.

B. Evolving Protocols and Algorithms

The transition between distributed algorithms and protocols is seamless. Both require information exchange following specific patterns, but algorithms additionally involve some computation on the nodes of the networks.

In [49], Yamaguchi et al. define the problem of transforming a *service specification* given as a Petri Net with registers to a *protocol specification* in the same format. The service specification is structured like a program for a centralized system and hence, does not detail any message exchange [50]. The protocol specification defines how the different entities involved in the computation communicate with each other. El-Fakihy et al. [50] show how to synthesize such protocol specifications with 0-1 integer linear programming and a hybrid Genetic Algorithm under the objective of minimizing communication costs.

An especially interesting approach to protocol synthesis has been contributed by de Araújo et al. [51]. The starting point of their process is a finite state machine describing the interaction of a sender and a receiver – again as it would happen on a local system. Transitions between the states of the FSM are triggered by output and input events to and from both user processes. This specification is then transformed with a Genetic Algorithm to FSMs describing the protocol interactions locally for the sender or receiver.

El-Fakihy et al. [50] and de Araújo et al. [51] hence describe methods to automate the design of data and protocol flows. Their starting point is a *localized* view on the system behavior which is then completed to a *global* specification with the synthesized interactions between system entities. Our goal, on the other hand, is to translate given global behaviors to local rules. Furthermore, we do not solely wish to find communication sequences and message structures, but complete algorithms for distributed computations (which subsume both).

During the advent of distributed multi-agent systems in the mid to late 1990s, various researchers considered the automatic generation of communication patterns [52–54]. Most of them used Genetic Programming approaches to evolve cooperation based on information exchange but only concerned synchronous systems or control tasks. A popular example is the pursuit scenario [53], where two or more predator agents have to exchange information about their prey and use this information efficiently in order to capture it. Communication in the agent-centric related works is usually instantaneous [52, 53] and the simulations are synchronous [52, 53]. Our goal is to evolve algorithms for asynchronous systems, which poses different requirements onto the program representation (see Section VI-A5). Furthermore, the tasks in this kind of related work are rather fuzzy – agent movements which roughly go into the right direction for most of the time e.g., will lead to success in predator/prey scenarios. In our work, we investigate whether it is possible to evolve algorithms like computing the distributed greatest common divisor which do not permit approximate results or wrong intermediate results.

Tschudin [55] contributed a new way for representing protocols in 2003: *Fraglets*. We considered the Fraglets representation as one approach for evolving distributed algorithms in our experiments too and therefore, discuss it in more detail in Section VI-A4. One aspect of the structure of Fraglets is that the language is simple and all constructs are always syntactically correct. Tschudin [55] therefore anticipated that it would be suitable for evolving protocols. In his initial experiments where he aimed to create a confirmed message delivery protocol, no satisfying results could be reported. As reason for these problems, he identified the needle-in-a-haystack (NIAH) nature of the evolution of algorithms which is also one of the problems discussed in our analysis (see Section IV-C).

Tschudin's research group then shifted its focus from the *offline* evolution of protocols (which is the subject of this article) to their *online* adaption in running systems. Yamamoto and Tschudin [56, 57] create populations containing a mix of different confirmed delivery and reliable delivery protocols for messages. These populations were then confronted with either reliable or unreliable transmission channels and were able to adapt to these conditions quickly. Re-adaptation in a later stage, however, proved again to be harder.

The experiments on protocol evolution and adaptation by Tschudin alone and in cooperation with Yamamoto both focus mainly on protocols, on the way information is delivered from one node to another in a distributed system. They do not involve additional computations on the nodes.

Artificial life approaches have been applied to the evolution of communication patterns as well. First results in this area have been reported by Werner and Dyer [58] in the early 1990s. Amongst the ALife methods, the recent Digital Evolution approach by McKinley et al. [59–61] based on the Avida platform [62] deserves most attention here. In our algorithm synthesis approach, Genetic Programming evolves a population of programs, each evaluated in separate simulations and well-known reproduction operators are used to explore the search space. In Avida, on the other hand, programs are self-reproducing organisms which co-exists in the same (simulated) distributed system. Apart from these conceptual differences, both concepts also exhibit an interesting duality. The first experiments with them were on distributed leader election [26, 63] and were conducted independently at around the same time. Again independently and simultaneously, different forms of interaction with Model-Driven Development environments and UML model generating facilities were developed [25, 60].

A general weakness common to all the related work listed in this section is the lack of comparisons with different Genetic Programming techniques. In the Digital Evolution, for instance, the instruction set and virtual CPUs of the programs may change, the linear Genetic Programming-like structure of the programs, however, remains the same. The research papers mentioned above rarely raise the question whether there may exist different forms of program representations exhibiting more favorable traits. In the case that a new representation has been developed, the direct comparison with other methods is usually omitted. Furthermore, experiments tend to be limited to simple tasks such as leader election.

While the first problem results from the architectural challenge and the high effort that cross-representation comparisons require, the second one is the result of the hardness of Genetic Programming of distributed algorithms. Indeed, the problem difficulty rises quickly with the complexity of the global behavior to be created, which leads to unsatisfying experimental results – both issues have been mentioned by Tschudin [55] and experienced by the authors themselves [22]. Nevertheless, in this article we provide a comparison of six different program representations [22] applied to three different problems. We furthermore propose one possible answer to the question for better program representations and deliver evidence clearly supporting this claim.

III. Distributed Systems Model

As pointed out in the introduction, there exists a wide variety of distributed systems, each having their own special features and peculiarities. In this section, we want to detail the class of distributed systems for which we want to synthesize algorithms.

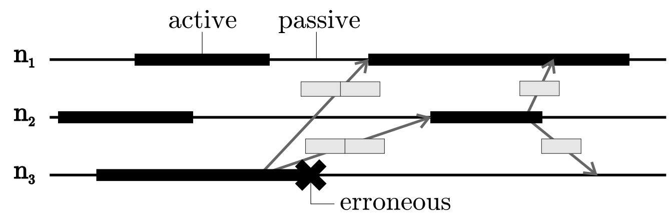


Fig. 2. A timing diagram annotated with the states of the nodes/processes.

A distributed system is a set of autonomous systems (nodes) which are connected by a network and communicate via the exchange of messages [2, 64–66]. Distributed algorithms [66, 67] are algorithms which are executed by multiple computers in a distributed system and cooperatively try to solve a given problem. There usually exists no shared global state information and each node has only knowledge about the information locally available on it. Information from other nodes can only be obtained by communication via message exchange. Our goal is to evolve algorithms which can become *part* of the software controlling all the nodes of a distributed system, modules which are suitable for specific task.

A. Algorithm Class

Most of the algorithms we want to synthesize are non-approximative. In other words, they compute distinct results which are, in our case, integer-valued. In Section VI-E, for instance, we discuss the evolution of the algorithms for computing the greatest common divisor of n numbers. A result of such a computation is either right or wrong, and even if it is only 1% off, it is still wrong. Such algorithms differ greatly from aggregation protocols [9, 68], for example, where real approximations of actual values are computed.

B. State Model

One of the key points of our work is that we consider the evolution of distributed algorithms for *asynchronous* systems [65]. By doing so, the results can clearly be distinguished from, for instance, the works presented in [52], and also reflect the applicability of Genetic Programming to distributed systems in real-world scenarios.

In Fig. 2, we sketch an example of how a distributed algorithm could proceed using a notation similar to Tanenbaum and van Steen's in [2]. A node (and the process running on it) according our network model is always in one of the three states illustrated in Fig. 3[1]:

1) A node is ***active*** if it is currently executing instructions of the distributed algorithm. An active node may send or receive messages.
2) A node is ***passive*** when it is currently not executing any instruction but waiting for incoming messages. It cannot

[1] In [22], we additionally consider explicit termination.

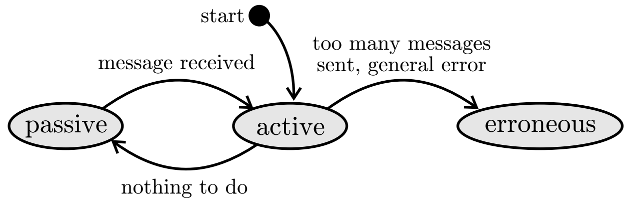


Fig. 3. The state diagram of the nodes in our simulations (simplified from [22]).

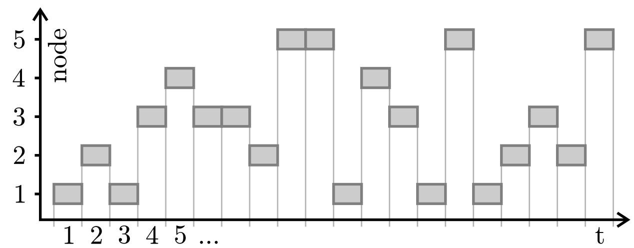


Fig. 4. Asynchronous parallelism in a model of a network of five nodes.

send messages in this state but may become ***active*** again and perform some computations when receiving one.

3) A node is ***erroneous*** if it has performed an erroneous or forbidden action.

C. Parallelism Model

One feature of the systems we consider is that their nodes proceed asynchronously. This means that the local progresses of the processes running on them may differ greatly and that the processors on the nodes may run at different speeds.

For evaluating the evolved programs, we use a network simulation in which time proceeds in discrete steps. In each time step, one ***active*** node is picked randomly according to the uniform distribution to perform one execution step. On average over infinite time, all nodes will perform the same amount of algorithm steps. For a given time period however, it is possible that one node can execute five steps while another one just proceeds by two, for example, as sketched in Fig. 4.

The network simulations run for a fixed number of time steps and are terminated thereafter. Partly because of this runtime limitation, in two of the performed experiments, programs evolved that iteratively approximate and refine the computation results. They store their best-guess of the result in a certain memory cell and update it during their run.

D. Network Model

A message in our system is a fixed-length sequence of integers (except for the `Frag` program representation discussed in Section VI-A4 where it corresponds to a Fraglet). For each node $\mathbf{n}$ in a network $\mathbf{N}$, the neighbor set $\mathrm{neighbors}(\mathbf{n}) \subseteq \mathbf{N} \setminus \mathbf{n}$ is the set of all other nodes to which $\mathbf{n}$ can send messages. Whenever a node $\mathbf{n}$ transmits a message, it is treated as a multicast to all the nodes in $\mathrm{neighbors}(\mathbf{n})$. This scenario basically holds for many wireless and wired distributed systems alike. Like in many real systems, unicasts could be emulated by message filtering methods and it lies in the responsibility of Genetic Programming to evolve such behavior if needed.

In our simulations, we also consider transmission latencies since we aim for asynchronous systems. We simulate this by splitting each single message into $|\mathrm{neighbors}(\mathbf{n})|$ identical transmissions, each targeting exactly one node and having its own latency. In the experimental settings of each of the three analyzed problems, we will choose a different topology for investigation.

The communication latency in a real network depends on many factors [65, 69] and it is hard to estimate it correctly with any probability distribution. We therefore decided to randomly pick message delays according to the uniform distribution. With this distribution, the probability of phenomena such as messages taking over each other is very high. Thus, simulations based on this model are more challenging for the algorithms being evaluated and the chances of discovering errors or deficiencies is the highest.

Although our system can simulate packet loss, in the experiments presented here we assumed a reliable communication medium. Yet, long message delays are possible in the simulation and such delays come, to a certain degree, close to the loss of a message. Interestingly, many of the synthesized algorithms turned out to be invulnerable to the loss of certain messages, see, for instance, the evolved mutual exclusion algorithms presented in Section VI-D3.

In the simulations, there is a limit to the maximum number of messages a node can send in order to prevent memory overflows in the simulator. It is usually set to values in $\mathbf{O}\left(n^{2.5}\right)$ or $\mathbf{O}\left(n^{3}\right)$ where n is the number of nodes in the network. If a node exceeds this limit, it is terminated as ***erroneous***.

IV. Difficulties in Evolving Algorithms

As outlined in our related work study given in Section II-B, there apparently are only few contributions to the area of evolving distributed algorithms which perform non-approximative computations in asynchronous systems. The reason for this absence of research is that the associated optimization problems are very difficult [55]. Hard enough that the optimization process may degenerate to a random walk if no measures are taken. For this difficulty, there are several reasons [22, 70, 71] which we will shortly outline in this section.

A. Ruggedness and Weak Causality

Optimization algorithms generally depend on some form of gradient in the objective space. The objective functions should be continuous and exhibit low total variation. If they are unsteady or fluctuating, i. e., rugged, optimization becomes more complicated.

Strong causality means that small changes in the properties of an object also lead to small changes in its behavior [72]. In fitness landscapes with weak causality, small modifications of the individuals instead lead to large changes in the objective values and make the fitness landscape rugged.

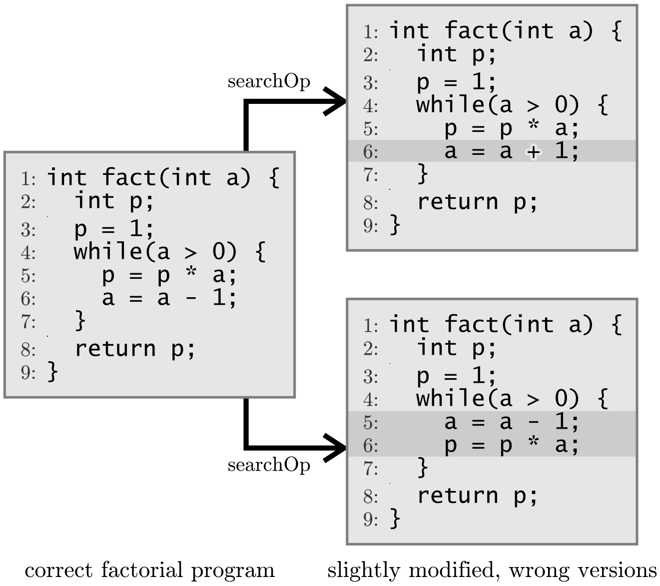


Fig. 5. Example for weak causality in Genetic Programming.

Exactly this is the case in algorithm synthesis problems. In Fig. 5, we sketch a program for computing the factorial $\mathtt{p} = \mathtt{a}!$ of a natural number $\mathtt{a} \in \mathbb{N}_1$ in `Java` or `C` notation on the left hand side. Such a program could have evolved with Genetic Programming in a tree or a linear representation. Assume that the program illustrated on top of the right hand side, where the `-` in line 6 is replaced by a `+`, was the result of the application of a search operation to the left program. This modification then clearly constitutes a violation of causality, since its result is a program behaving very differently and hence, induces a jump in the objective functions. Notice that such phenomena are especially intense in the problem class we envisaged in Section III-A.

B. Epistasis

This lack of causality is rooted in the high epistasis inherent to most of the program representations applied in Genetic Programming. In biology, *epistasis* is defined as a form of interaction between different genes [73]. In optimization, epistasis is the dependency of the contribution of one gene (an instruction of a program in our case) to the value of the objective functions on the allelic state of other genes [18].

In a conventional computer program, not only the presence of an instruction and its semantics are important, but also its position. Strong *positional* epistasis exists in both, linear and tree-based forms of Genetic Programming which have been designed with conventional program structures in mind [24]. This issue is illustrated at the bottom right of Fig. 5.

Let us consider a program P as a function $P : I \mapsto O$ that connects the possible inputs I of a system to its possible outputs O [24]. Two programs P_1 and P_2 can be considered as equivalent if $P_1(i) = P_2(i)\ \forall i \in I$. For the sake of simplicity, we further define a program as a sequence of n statements $P = (s_1, s_2, \dots, s_n)$. There are $n!$ possible permutations of these statements. We define $\xi(P)$ as the fraction $\xi(P) = \frac{v}{n!}$ of v permutations that lead to programs equivalent to P. If we assume that recombination operators often effectively permutate instructions, the probability of creating highly-fit offspring from highly-fit parents in program representations where ξ is usually low will be low too. Thus, exhibiting high values of ξ for many programs, i. e., having low positional epistasis, is a beneficial feature of a program representation [24].

Avoiding epistatic effects should be a major concern of the design of program representations [24] which, unfortunately, has been neglected in the past. Besides the rule-based approaches presented in this paper, the authors have knowledge of only two other methods which reduce epistasis in GP on representation/execution model-level:

1) Algorithmic Chemistries: Lasarczyk and Banzhaf [74–76] developed a Genetic Programming approach called Algorithmic Chemistry where positional epistasis is circumvented. It basically is a variant of linear Genetic Programming (see Section VI-A3 on page 9) where the execution order of the single instructions is defined by some random distribution instead of being fixed as in normal programs. Of course, if the instructions of a program are always executed in a random order, there can be no positional dependencies between them ($\xi \to 1$) and they can freely be permutated. The drawback of this approach is that the programs are no longer deterministic and their behavior and results may vary between two consecutive executions. Therefore, this method does not lend itself to the evolution of deterministic distributed algorithms.

2) Soft Assignment: Another approach for reducing the epistasis is the soft assignment method (*memory with memory*) by McPhee and Poli [77]. It implicitly targets epistasis by weakening the way values are assigned to variables. In traditional programs, instructions like `x=y` or `mov x, y` will completely overwrite the value of `x` with the value of `y`. McPhee and Poli replace this strict assignment semantic with $\mathtt{x}_{t+1} = \mathtt{y}_t \quad \equiv \quad \mathtt{x}_{t+1} \longleftarrow \gamma \mathtt{y}_t + (1-\gamma)\mathtt{x}_t$ where $\mathtt{x}_{t+1}$ is the value that the variable `x` will have after and $\mathtt{x}_t$ its value before the assignment. $\mathtt{y}_t$ is the value of an arbitrary expression which is to be stored in `x`. The parameter γ is "a constant that indicates the *assignment hardness*" [77].

For mathematical or approximation problems, this approach is very beneficial. The drawback of programs using soft assignment is that, although they are deterministic, they are approximative and cannot compute precise values as required in some discrete problems. One example for such a problem where soft assignments cannot be applied is the Greatest Common Divisor experiment discussed in Section VI-E.

Besides the new rule-based approaches, we furthermore strengthen the causality in linear program representations (`LGP`, `Frag`) by applying homologous crossover [78].

C. Correctness

The epistasis-induced ruggedness in the fitness landscape of Genetic Programming of non-approximative, deterministic, distributed algorithms makes it hard to find good candidate solutions. Another problem is the definition of *good* itself.

Determining the correctness of programs in Turing-complete representations will never be generally possible [79, 80], although model checking approaches such as SPIN [81, 82] with which asynchronous distributed algorithms can be processed [83] made large progress in the recent years.

Instead, the only general way to determine a program's behavior is by simulating it. In our case, this means executing the program in a simulated network. Here, a *training case* is characterized by the values of all parameters of the network, including the number of nodes, the network topology, the message latencies, and the assignments of execution steps to the nodes. It is easy to see that exhaustive testing of all possible training cases is not possible. Hence, the program behavior is *approximated* instead of *determined*.

Therefore, we use the notion of *functional adequacy* as defined by Gleizes et al. [84] instead of correctness in our work: When a system has the "right behavior – judged by an external observer knowing the environment – we say that it is functionally adequate." In our GP system, the objective functions act as external observers.

It should be noticed that an objective function solely rating the correctness of a program for a given problem – maybe returning 0 for *wrong* and 1 for *correct* – would be of low utility in any metaheuristic optimization process anyway. It would lead to a needle-in-a-haystack problem, also known as the all-or-nothing-feature of Genetic Programming [55], since it provides no gradient information at all and the performance of the optimizer degenerates to the one of a random walk. What is needed instead is a formulation which allows rewarding also "partial adequacy" which can be done by evaluating simulations. For our experiments, we define objective functions with exactly this feature in Section VI-C1, Section VI-D1, and Section VI-E1.

D. Overfitting

When evolving algorithms by using training cases for the fitness evaluation, there is a high chance of overfitting [85]. Programs may emerge which have learned the right response to each scenario instead of being general solutions. Such programs are not correct and only behave adequately for *exactly* the scenarios used for training but will fail in scenarios with even only slightly changed parameters.

We apply two measures against the problem of overfitting: First, we introduce a non-functional objective function putting pressure into the direction of smaller programs. Since overfitted programs often resemble large decision tables, this reduces the probability of producing them. At the same time, this measure also reduces *bloat* (uncontrolled growth in program size) and *introns* (program parts which do not contribute to the functional fitness) [20, 86]. Second, we generate multiple, randomized training cases which are replaced after each generation of the EA and set the final objective value of a program to be the arithmetic mean of its scores achieved in the scenarios. It should be noted that using multiple training cases leads to more stable objective values and reduces the probability of outliers, but has to be paid for with an increase in runtime [47].

V. Rule-based Representations

In Section IV-B we have argued that epistasis is one of the key problems in Genetic Programming. There exists one class of Evolutionary Algorithms that elegantly circumvents positional epistasis: the (Learning) Classifier Systems (LCS) family [87]. In the Pittsburgh LCS approach [88], a population of rule sets is evolved with a Genetic Algorithm. Each individual in this population consists of multiple classifiers (the rules) which transform input signals into output signals. The evaluation order of the rules in such a classifier system plays no role except maybe for rules concerning the same output bits, hence $\xi \approx 1$. The idea behind our new GP approaches described in the following text is to use this knowledge to create a new program representation that retains high ξ-values in order to become more robust in terms of reproduction operations [22, 24].

A. Rule-based Genetic Programming [RBGP]

The first step into this direction is the Rule-based Genetic Programming (`RBGP`) method introduced in its original form in [24]. A `RBGP` program consists of arbitrary many rules, each divided into two conditions and an action which is executed if the conditional part evaluates to `true`. The conditions each compare the values of two symbols and are concatenated with either the $\vee$ or $\wedge$ operator. In RBGP, each symbol identifies an integer variable, which is either read-only *(r/o)* or read-write *(r/w)*. Some *r/o* symbols are defined for constants such as `0` and `1`. The *r/w* symbol `start` is only `1` during the first application of the rules and reset to `0` afterwards (it can, however, be set by the program itself). Furthermore, a program can be provided with some general-purpose variables (`a`, `b` ...). Symbols with special meanings are introduced for evolving distributed algorithms: input symbols $\texttt{in}_1$, $\texttt{in}_2$, ... where the contents of incoming messages will occur and variables $\texttt{out}_1$, $\texttt{out}_2$, ... which are mapped to the integer fields of an outgoing message on transmission are added. A message is sent with a special `send` action and a special symbol `incomingMessage` is automatically set to `1` whenever a message arrives.

1) Execution of RBGP Programs: The value of symbols can either change because of data incoming from the outside when messages are received or by the actions of the program itself. In `RBGP`, actions do not directly modify the values of the symbols but rather write their results to a temporary storage. After all rules have been processed, the temporary storage is committed to the actual memory as sketched in Fig. 6. The symbols in the condition part and in the computation parts of the actions are annotated with the index t and those in the assignment part of the actions are marked with $t+1$ in order to illustrate this issue.

2) Levels of Independence: This approach generates two levels of independence which are not available in normal program representations. First, it allows for great amount of disarray in the rules since the only possible positional dependencies left are those of rules which write to the same symbols. All other rules can be freely permutated without any influence on the behavior of the program. Therefore, the positional epistasis in RBGP is very low and $\xi \approx 1$.

Second, the cardinality of the rules plays no role either. The evolutionary process may, for instance, duplicate one rule in a

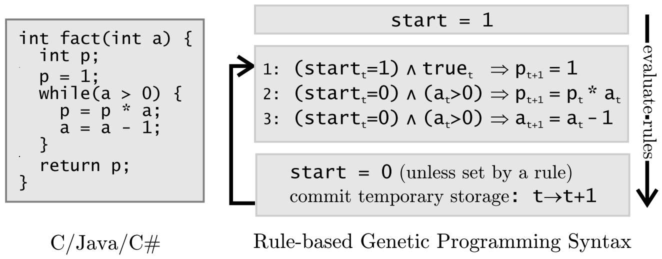


Fig. 6. A program computing the factorial of a natural number in `Java`, `C`, or `C#` and RBGP syntax.

Symbol	Encoding
0	0000, 1100
1	0001, 1101
start	0010, 1110
id	0011, 1111
netSize	0100
in	0101
out	0110
enterCS	0111
csLeft	1000
a	1001
b	1010

Comp.	Enc.
$>$	000
$\geq$	001
$=$	010
$\leq$	011
$<$	100
$\neq$	101
true	110
false	111

Concat.	Enc.
$\wedge$	0
$\vee$	1

Action	Enc.
$=$ x+y	00
$=$ x−y	01
$=$ x	10
$=$ 1−x	11

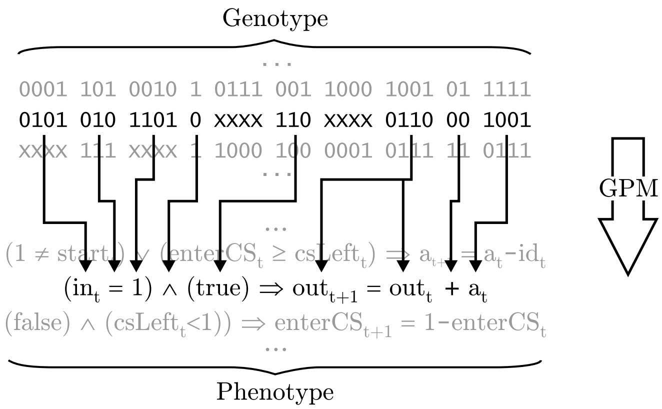


Fig. 7. An example for the encoding in RBGP similar to the one used in the critical section experiments in Section VI-D.

reproduction step without direct influence on the functional fitness. Subsequent mutations may then specialize the two rules and lead to the evolution of new functionality. In biology, similar processes are assumed to significantly contribute to evolution [89, 90]. `RBGP` is thus one possible answer to Hu and Banzhaf's [91] question for transposing this biological mechanism to GP.

3) Binary Encoding: The sets of symbols and actions are specified before the Genetic Programming process starts. Based on the fixed structure, a straightforward binary encoding is constructed as sketched in Fig. 7 and the programs can be evolved with a normal GA [24].

B. Extended Rule-based Genetic Programming [eRBGP]

Rule-based Genetic Programming is designed with the goal to lower epistasis and hence, to increase the causality which in turn should lead to a reduction of the ruggedness in the fitness landscape. If this could be achieved, the Genetic Programming

```
(start_t > 0) ∧      true      ⇒  a_t+1 =  0
(start_t > 0) ∧      true      ⇒  b_t+1 =  0
    (a_t < l_t) ∧ ([a_t]_t < [b_t]_t) ⇒ [a_t]_t+1 =  [b_t]_t
    (a_t < l_t) ∧ ([a_t]_t < [b_t]_t) ⇒ [b_t]_t+1 =  [a_t]_t
    (b_t ≥ a_t) ∧    (a_t < l_t)    ⇒  a_t+1 = a_t + 1
    (b_t < a_t) ∧      true      ⇒  b_t+1 = b_t + 1
    (b_t ≥ a_t) ∧    (a_t < l_t)    ⇒  b_t+1 =  0
```

Listing 1. A simple selection sort algorithm written in the eRBGP language.

```
if( (a<b) && (c>d) && (a<d) ) {
  a += c;
  c--;   }
```

Listing 2. A complex conditional statement in a `C`-like language.

processes would likely result in better solutions. However, `RBGP` is still limited in two aspects: power and expressiveness.

1) Power: The original `RBGP` method is not Turing-complete. Teller [92] and Woodward [93] both argued that this feature is present in program representations with indexed memory. Hence, we introduced such an extension to `RBGP` in order to test whether Turing completeness is helpful for the evolution of distributed algorithms also in situations where it is not strictly required. We define the notation $[a_t]_t$ which stands for the value of the a_tth symbol at time step t in the ordered list of all symbols. In this, it is equivalent to a simple pointer dereferentiation (`*a`) in the `C` programming language.

With this extension alone, it now becomes possible to use the RBGP language for defining list sorting algorithms, for instance. Assume that the following symbols $(i_0, i_1, .., i_{n-1}, l, a, b)$ have been defined and arranged in that order (starting with i_0 at index 0) in memory. The symbols i_0 to i_{n-1} constitute the field which is used to store the list elements and l is initialized with the length of the list. Listing 1 then represents a variant of selection sort.

2) Expressiveness: Another restriction of the initial `RBGP` approach is that its rules always consist of exactly two conditions. Since logical operators and variables for storing values are available, arbitrarily complex conditions can be expressed. This expressiveness is, however, achieved by allocating variables for temporary evaluation results and additional rules. It comes with the trade-off of rising epistasis and decreasing causality, hence jeopardizing the purpose of Rule-based Genetic Programming, the reduction of epistasis.

If the statement in Listing 2 was translated to `RBGP` syntax, we would need four rules and an additional variable e_t. The result presented in Listing 3 clearly shows the increase in complexity. We therefore extend the expressiveness in `eRBGP` by dropping the constraints on the structure of the rules and allow the formulation of arbitrarily complex expressions [22]. Thus, the `eRBGP` version of Listing 3 (given in Listing 4)

```
  true   ∧   true   ⇒ e_t+1 =  0
(a_t < b_t) ∧ (c_t > d_t) ⇒ e_t+1 =  1
(a_t < d_t) ∧ (e_t = 1)  ⇒ a_t+1 = a_t + c_t
(a_t < d_t) ∧ (e_t = 1)  ⇒ c_t+1 = c_t − 1
```

Listing 3. The RBGP version of Listing 2.

```
((a_t < b_t) ∧ ((c_t > d_t) ∧ (a_t < d_t)))  ⇒ a_{t+1} =  (a_t + c_t)
((a_t < b_t) ∧ ((c_t > d_t) ∧ (a_t < d_t)))  ⇒ c_{t+1} =  (c_t − 1)
```

Listing 4. The eRBGP version of Listing 3.

becomes much simpler.

Because of this increase in expressiveness, the eRBGP programs cannot be encoded in fixed-length binary strings anymore and we use tree genomes instead.

VI. Experiments

We applied a set of six Genetic Programming methods to three problems from distributed computing in order to obtain a good understanding of their utility in this domain. Additionally to our two new rule-based approaches, we test tree-based Standard Genetic Programming, linear Genetic Programming, and Fraglets, a programming paradigm inspired by bio-chemical metabolisms. The three problems to which we will apply these approaches are the evolution of election algorithms, of mutual exclusion algorithms, and of algorithms for computing the greatest common divisor. The problems have characteristic properties and differ in hardness, the number and structure of objective functions, and network topologies. In this section, we will first describe the additional program representations and then discuss our experiments in depth.

A. Program Representations for Comparison

Besides the two rule-based methods, we tested four other program representations in order to cover a wide range of different GP approaches in our experiments. These additional approaches are based on well-known representations and contributions from related work. Like in the rule-based programs, data (such as the content of memory cells) is always in 32bit signed integer format in them and similar to the C programming language, Boolean expressions are also integer-valued, i. e., false if 0 and true otherwise.

1) Standard Genetic Programming with Memory [SGP]: The baseline approach for Genetic Programming is to use a standard, tree-based genome. Koza [21] defined such genomes back in 1992. In our experiments, we use the tree representation with some modifications in order to facilitate the requirements of cooperative computations based on asynchronous network communication:

Each program consists of at least two automatically defined functions (*ADFs*, [21]). The first one is called on the startup of the program. The second function is invoked as an *asynchronous function call* whenever a message is received by the node, similar to an interrupt service routine (ISR) in the Intel 80x86® architecture [94, 95].

Each node has a global (process-scope) memory that resembles the data segment of a program which is accessible by from all function scopes. It allows the message handler, for instance, to store permanent information. Additionally, there is local memory private to the scope of each function call. This is essential to allow message handlers which were asynchronously invoked to process data without interfering with other procedures or each other [22].

All parameters of a function (such as the incoming message in case of the second ADF) are stored in its local memory. The instruction send which causes a message transmission to all nodes in reach has between one and two parameters which denote the contents of the message to be sent. Besides normal arithmetic expressions, the SGP language allows the definition of alternatives and while loops which take an expression and two (respectively one) blocks of instructions as parameters.

2) Extended SGP [eSGP]: Like RBGP, the SGP approach is not Turing-complete. We therefore introduce an extension similar to the one in Section V-B1. Here we also test another concept, an additional layer of indirection transparent to the GP system. With the special instruction decl, memory locations can be marked "for use". Instead of accessing memory directly, programs now use *virtual* addresses which are indices into a translation table. This way, they are resolved to *real* addresses which are either direct or indirect.

An additional construct in the eSGP language is a for loop which takes a minimum and a maximum value of the loop counter as well as a block of instructions as parameter. The counter variable is automatically declared by the loop.

3) Linear Genetic Programming [LGP]: Trees are not the only way for representing programs. Indeed, a computer processes programs as sequences of instructions (which may contain branches realized by jumps to other places in the code) instead. The area of Genetic Programming concerned with such instruction string genomes is called *linear Genetic Programming* or LGP for short [96–98].

The advantage of LGP lies in the straightforwardness of evaluation and the simplicity of limiting the runtime and simulating parallelism since one instruction can be assigned to each time step which are distributed as shown in Fig. 4. We therefore chose such a format with extensions for supporting ADFs and the memory features of eSGP (but without the declaration feature and indirection) as the third approach for comparison. An LGP program is a variable-length list of integer strings – each list standing for one function.

The nodes executing these programs are three-address machines, i. e., machines where arithmetic instructions have up to three parameters: the target address and the addresses of two operands. In this, our LGP language is very similar to the one used by Lasarczyk and Banzhaf [74] in the Algorithmic Chemistries approach discussed in Section IV-B1. Conditional jumps (jmp) and function calls (call) use an internal flag register filled by a comparison instruction cmp with exactly the same semantics as in the Intel architecture [95]. For sending messages, a buffer similar to a stack is provided whose contents can be multicasted with a send instruction.

Because of the mentioned positive aspects, executing a LGP program in a simulation is much easier than doing the same with a SGP or eSGP program. We therefore automatically compiled all SGP and eSGP programs to the LGP representation before executing them in our network simulations. Since SGP and eSGP programs are practically incomprehensible[2], this has the second advantage that the LGP phenotypes have

[2]due to the levels of memory indirection and the fact that memory indices are actually integers which have to be normalized with modulo operations

a better readability than the SGP and eSGP genotypes. The third advantage is that LGP, SGP, and eSGP are now directly comparable.

4) Fraglets [Frag]: The Fraglet language by Tschudin [55, 99] is an execution model for communication protocols which resembles the chemical reactions in living organisms. Fraglets are symbolic strings of the form $[\mathtt{s}_1 : \mathtt{s}_2 : \ldots : \mathtt{s}_n]$. The symbols s_i either represent control information or payload. Each node in the network has a *Fraglet store* which corresponds to a reaction vessel in chemistry. Fraglet stores are implemented as multisets keeping track on the multiplicity of the Fraglets they contain.

The instruction set defined by Tschudin [55] comprises transformation and reaction rules for Fraglets. In the former, the first symbol of a Fraglet issues a change to its tail and in the latter, two Fraglet strings are combined according to the operation defined by the first symbol of one of them. In our experiments, we utilize a subset of the Fraglet language as of September 2007 [100] (encompassing point 1, broadcast, and lt) with problem-specific extensions. For a discussion of the Fraglets language we refer the interested reader to [55, 56, 99, 101, 102], or [22].

5) Summary of the Approaches: All in all, we defined six different representations for Genetic Programming and adapted them to the evolution of distributed algorithms. The asynchronicity of our system model defined in Section III poses some specific requirements onto the program representation. Our system model, for instance, permits messages to arrive at nodes in short succession. In other words, a node might be busy processing one message while the next one is already received.

The Frag approach solves this problem in a very natural way: messages and the modules of the programs are both artificial molecules injected into the same “reaction vessel”. Multiple messages just mean more molecules ready to react. In the rule-based programs, a symbol signals incoming messages (and the $\mathtt{in}_t$-symbols take on non-zero values). Since all rules are applied at once in each time step, programs may process one message per iteration and hence, comply with the systems model per default.

Normal tree-coded or linear programs known from off-the-shelf SGP or LGP implementations are not suitable for such a scenario. If messages are simply mapped into memory, they may be overridden too fast if the process is busy with other things, say executing a loop. Therefore, we introduced the concurrency model by defining interrupt-like message handler ADFs. With the local memory concept, they can handle messages and perform computations without interfering with the main routine or concurrently running instances. This work hence also explores three different mechanisms for message handling in Genetic Programming.

In Table I we summarize the six program representations and list their genomes, phenomes (if different from the genomes), whether they are Turing-complete (TC) or not, and what their equivalent of one single execution step is. The basic instruction sets utilized are given in Listing 5. Its full specification can be found in [22] which will be permanently online available. Division operations are not protected, a division by zero leads to a transition to ***erroneous*** of the issuing node.

	genome	**phenome**	**TC**	**1 step**
SGP	trees	LGP	no	1 instruction
eSGP	trees	LGP	yes	1 instruction
LGP	genome ≡ phenome, multiple functions where each function ≡ variable-length integer list		yes	1 instruction
Frag	genome ≡ phenome, multiple fraglets where each fraglet ≡ variable-length lists of integer-encoded symbols		? [102]	1 reaction
RBGP	variable-length bit strings	rule sets	no	1 full evaluation
eRBGP	trees		yes	1 full evaluation

Table I
THE SIX GENETIC PROGRAMMING APPROACHES UTILIZED IN THIS WORK.

```
SGP: function_declaration, call(func,params),
    blocks, while, var=<expr>, return(<expr>),
    send(<expr>,..), comparison, +, -, *, /, and,
    or, not

eSGP: SGP + variable declaration, indirect memory
    access, for_loop

LGP: +, -, *, /, %, not, and, or, xor, =, xchg,
    jmp, call, send, comparison

Fraglets: dup, exch, fork, nop, null, pop2, split,
    lt, max, broadcast, node, id, terminate, match,
    matchP

RBGP: +, -, *, /, %, not, terminate, send

eRBGP: direct/indirect memory access, =, +, -, *,
    /, and, or, not, comparison, send, terminate
```

Listing 5. The instruction sets of the applied GP approaches.

B. Experimental Configuration and Evaluation

1) General Configuration: All of our experiments are based on multi-objective Genetic Programming which we realize by a plain Pareto-ranking based fitness assignment procedure [18] which assigns one scalar fitness to each candidate solution representing its *relative* utility in comparison with the other members of the population. This assignment process also considers the Euclidean distances of the candidate solutions in the objective space and punishes individuals located very close to each other in order to further diversity. For the same purpose, we delete individuals with exactly the same objective values with a certain probability from the population [18, 22]. The utility of these settings were tested on small-scale GP experiments and on a suitable benchmark model [103] beforehand.

Generally, we apply all the measures outlined in Section IV for mitigating the difficulties arising in the area of synthesizing distributed algorithms. In the EAs, we furthermore used steady-state populations consisting of 512 individuals and applied tournament selection with five contestants. The mutation rate was set to the rather high value of 40% (distributed as follows: 45% node/gene modification, 15% deletion, 15% insertion, 25% re-arranging) and the crossover rate to 70% (homologous multi-point crossover for integers/bit strings and sub-tree crossover for trees) as these settings turned out to be efficient in our previous experiments. Homologous crossover corresponds to the exchange of sub-routines of a program.

The utility of the evolved distributed algorithms is determined in network simulations which obey the models introduced in Section III. Random numbers influence many parameters of our simulations, such as the assignment of computation time to nodes, the latency of every single message, and the number of nodes in the network. We refer to the set of all random numbers used during a simulation as one *training case*. For every generation of the EA, multiple training cases are newly created and each program in the population is applied to all of them. By using the same training cases for each individual, we ensure equal chances for every candidate solution and also that equal programs receive the same objective values.

2) General Evaluation: For evaluating the experiments, we on one hand provide tables denoting the key performance indicators of each configuration, such as the fraction of runs in which adequate algorithms evolved and the arithmetic mean of the best achieved objective values. However, such values can only provide limited insight into which algorithm is *actually* better and may even be deceptive. The same holds for diagrams illustrating the convergence of the optimization processes (which have been omitted for space reasons in this article). If the goal is to make profound statements about which approach is superior in a given scenario, only statistical tests [104] provide useful answers. Therefore, we analyze our results with such tests, choosing non-parametric variants in order to not make false assumptions about the distribution of the compared variables. The outcomes of the tests define partial orders which can easily be visualized with diagrams (such as Fig. 11).

C. Experiment 1: Election

Election algorithms have many applications in distributed systems. They are used to determine the coordinators in several routing [105] or group communication protocols [12], for instance. According to Le Lann [106], a distributed election algorithm can be initiated by any number of nodes in the system and will reach a terminal configuration in which exactly one node is elected as *leader* and all nodes agree to this choice. Many different ways to perform distributed elections have been developed, such as Le Lann's original approach for ring topologies [106], the message extinction algorithm by Chang and Roberts [107], and special methods for MANETs [12].

We adapt the assumptions of Le Lann about the network $\mathbf{N}$ of nodes $\mathbf{n}$ performing the election as follows: *(1)* The IDs of the nodes are unique numbers drawn from $\mathbb{N}_0$ and the order imposed on them is the $<$-relation. *(2)* A node *does not* know the IDs of the other nodes. *(3)* At startup, a node $\mathbf{n} \in \mathbf{N}$ only knows its own ID $\mathrm{id}(\mathbf{n})$ (which is stored in a dedicated variable, symbol, or memory cell). *(4)* During the election, each node $\mathbf{n}_1$ in $\mathbf{N}$ will decide for a node $\mathbf{n}_2 \in \mathbf{N}$ which it thinks has won the election. It will store the ID of this node in another dedicated memory cell $(\mathrm{elected}(\mathbf{n}_1) = \mathrm{id}(\mathbf{n}_2))$.

The novelty of the election task defined here and compared to the "standard election problem" from [106] is Point 2. In the domain of novel distributed systems as described in the introduction, a possibly large number of nodes are deployed and the assumption from [106] that each node knows the IDs or the number of other nodes in the network will generally not hold.

In the area of Genetic Programming, a few attempts to solve the election problem have been recorded [26, 63]. Only in our work, however, a comparison between different Genetic Programming approaches in the election domain is performed [23].

1) Objective Functions: In order to derive such algorithms, we apply an evolutionary process governed by two objective functions: An objective f_{el} which furthers *functional adequacy* and a non-functional criterion f_{ps} which minimizes the size of the synthesized programs. We propose two possible definitions of *functional adequacy* for election algorithms: *(a)* without restrictions on the node to be elected and *(b)* the elected node should either have the maximum or minimum ID, as it is the case in some well-known election schemes [106, 107].

The programs we want to evolve here converge to the correct result. The problem definitions are likely to lead to the emergence of algorithms that keep the application or the operating system up-to-date about what the current guess about the leader is. If a node thinks that it is not the leader but receives messages for the leader, it would simply propagate them to the node which it assumes to be the elected one.

In Fig. 8, we specified the functional objective function $f_{el.a}$ for category-*a* algorithms. This function will always take on values between zero and one, where 0 is the optimum and 1 is the worst case.

$$\left(\max_{x \in ids} x = \max_{\mathbf{n} \in \mathbf{N}} \mathrm{id}(n)\right) \vee \left(\min_{x \in ids} x = \min_{\mathbf{n} \in \mathbf{N}} \mathrm{id}(n)\right) \qquad (1)$$

The objective function $f_{el.b}$ is computed exactly like $f_{el.a}$, but adds a penalty of 1 to r if Equation 1 does not hold. The result of $f_{el.b}$ is then $r/3$ so that again $0 \leq f_{el.b}(x) \leq 1$.

2) Experimental Settings: For each algorithm evaluation, 20 randomized scenarios with networks consisting of between 4 and 20 virtual machines were executed. The networks were organized in a linear topology where each node can only communicate with its direct predecessor and successor – the topology where the highest number of messages for finding the leader is to be expected. The message sizes were limited to two memory words except in the `Frag` approach, where complete Fraglets are exchanged.

The `LGP`, `SGP`, and `eSGP` programs were provided with two cells of global and local memory each. Nodes executing `RBGP` or `eRBGP` programs were equipped with two multi-purpose variables and the length of Fraglets was limited to 15.

The IDs of the nodes were stored in the first global memory cell (`LGP`, `SGP`, `eSGP`), a dedicated (writable) symbol (`RBGP`, `eRBGP`), or available via a special Fraglet. It is possible that a node can lose its ID during the execution of a program. Due to

Fig. 8. $f_{el.a}(x) \longleftarrow \text{fitnessElectionA}(x, \mathbf{N})$

Input: x*:* the simulated program
Input: $\mathbf{N}$*:* the network after the simulation has ended
Output: $f_{el.a}(x)$*:* the objective value of the algorithm x simulated in $\mathbf{N}$

begin
  $ids \longleftarrow \emptyset,\ valid \longleftarrow 0$
  for all $\mathbf{n}_1 \in \mathbf{N}$ **do**
    if $(\exists \mathbf{n}_2 \in \mathbf{N} : \text{id}(\mathbf{n}_2) = \text{elected}(\mathbf{n}_1)) \wedge (\mathbf{n}_1.state \neq \textbf{\textit{erroneous}})$ **then**
      $valid \longleftarrow valid + 1$
      //aggregate all different *valid votes*
      $ids \longleftarrow ids \cup \{\text{elected}(\mathbf{n}_1)\}$
    end if
  end for
  $r \longleftarrow 0$
  //compute proportion of valid votes
  if $ids \neq \emptyset$ **then**
    $r \longleftarrow r + \frac{|ids|-1}{\text{numNodes}(\mathbf{N})-1}$
  end if
  //punishment in case each valid ID was only voted for once
  if $|ids| = valid$ **then**
    $r \longleftarrow r + 1$
  end if
  //punish invalid votes
  $r \longleftarrow r + \frac{\text{numNodes}(\mathbf{N})-valid}{\text{numNodes}(\mathbf{N})}$
  return $0.5r$
end

Fig. 9. `electionA`

begin
  Subalgorithm `main`
  begin
  $\text{elected}(\text{self}) \longleftarrow \text{id}(\text{self})$
  $\text{sendMessage}(\text{elected}(\text{self}))$
  end
  Subalgorithm `onMessage`(message)
  begin
  $\text{elected}(\text{self}) \longleftarrow \text{message}_{[0]}$
  $\text{sendMessage}(\text{elected}(\text{self}))$
  end
end

Fig. 10. `electionB`

begin
  Subalgorithm `main`
  begin
  $\text{elected}(\text{self}) \longleftarrow \text{id}(\text{self})$
  $\text{sendMessage}(\text{elected}(\text{self}))$
  end
  Subalgorithm `onMessagemessage`
  begin
  if $\text{message}_{[0]} > \text{elected}(\text{self})$ **then**
    $\text{elected}(\text{self}) \longleftarrow \text{message}_{[0]}$
    $\text{sendMessage}(\text{elected}(\text{self}))$
  end if
  end
end

the selection pressure during the optimization, however, only such programs will survive that still function correctly, i. e., only lose the IDs that belong to nodes which will not become the leader, if any. Notice that it is not important that a node preserves the own ID. The evolved algorithms are assumed to become a module of a software system and that a copy of the ID exists outside of their scope. They just need to be able to name the ID of their best guess on the winner of the election, the executing process will then know whether it is the leader or not.

We repeated the experiments with the two functional objectives combined with f_{ps} in order to find out about the "GP hardness" of the different aspects of this problem.

3) Evolved Algorithms: In the problem definition *a)*, the goal of the evolution was to find an election algorithm which is able to name a winner after a certain amount of simulated time steps. All GP approaches were able to solve this problem driven by $f_{el.a}$ and f_{ps} by producing adequate programs, although largely differing in the fraction of successful runs. With the exception of the `Frag`[3] approach, the obtained programs most often belonged to the same algorithm classes presented in Fig. 9 and Fig. 10, which were manually-derived from the evolved solutions by removing unnecessary instructions and homologous transformations. Interestingly, the algorithm given in Fig. 9 behaves similarly to a Moran process [108] in biology and works in almost all scenarios perfectly well since it relies on the randomness in the message latencies. All nodes repeatedly send the IDs they vote for and immediately change their decision for the IDs they receive. Since the whole system is asynchronous, votes may be overwritten by messages before being propagated. Over time, IDs get extinct and sooner or later, only one prevails. Regardless of the size of the network $\mathbf{N}$, it will eventually converge to a situation where $\text{elected}(\mathbf{n}_1) = \text{elected}(\mathbf{n}_2)\ \forall \mathbf{n}_1, \mathbf{n}_2 \in \mathbf{N}$. The algorithm is not *correct* but works if the delay in the communication is sufficiently random.

In its behavior, this algorithm is very different from all traditional approaches to the election problem. In its structure, it is probably the simplest solution possible, realizable with only a few machine code instructions, and yet sufficient for many applications, especially in the novel network types listed in the introduction.

With the exception of the `SGP` approach, all Genetic Programming methods could also find solutions to the problem variant *b)* – again, with largely different success rates. The evolved programs generally followed the scheme defined in Fig. 10. An example for this behavior is the `RBGP` program specified in Listing 6, where the symbol $\mathtt{id}_t$ contains the node's ID, the result of the election is expected to appear in $\mathtt{a}_t$, and the $\mathtt{in}_t$ and $\mathtt{out}_t$ symbols are used for incoming and outgoing message content[4]. The evolved Fraglet algorithm in Listing 7 is another example for this structure. (Notice that, because of the different computational properties of Fraglets, multiple `elected` Fraglets may occur in one node's Fraglet store. In this case, we assume that the id corresponding to the most frequent `elected` Fraglets was voted for, i.e., make a majority decision.)

```
   false    ∨    true     ⇒ out_{t+1} = in_t
 (in_t ≥ id_t) ∧  true     ⇒  id_{t+1} = in_t
 (in_t ≥ id_t) ∧  true     ⇒ out_{t+1} = in_t
    true    ∨    false    ⇒   a_{t+1} = id_t
    true    ∧ (id_t ≠ a_t) ⇒  send
```

Listing 6. A `RBGP` program solving case *b)*.

It extends Fig. 9 by imposing a condition on forwarding the votes, reducing both the number of messages sent as well as

[3]All adequate `Frag` programs for problem *a)* belonged to that later class.

[4]the $\mathtt{in}_t$-symbols are non-zero only in case a message was just received

```
[broadcast : node : elected] //produce [node : elected]
[broadcast]                  //useless
[nul]                        //useless
/* compute maximum of two [elected:NN] fraglets and make a
[match:...] which creates [broadcast:...] fraglets */
[max2 : elected : match : elected : broadcast : elected]
[elected : match : pop2 : pop2 : lt : max2] //useless
```

Listing 7. A Frag program solving case *b)* (more information: [22, 100]).

Case		$\#r$	$\#s$	s/r	$\breve{st}$	$\overline{st}$	$\widehat{st}$	$\breve{f}_{el}$	$\overline{f_{el}}$	$\breve{f}_{ps}$	$\overline{f_{ps}}$	T[s]
SGP	**a**	37	35	0.95	76	198	782	0.000	0.052	10.0	17.8	316
	b	32	0	0.00	–	–	–	0.216	0.339	10.0	15.6	207
eSGP	**a**	55	43	0.78	114	354	997	0.000	0.147	10.0	17.3	227
	b	57	1	0.02	981	981	981	0.000	0.310	10.0	17.2	202
LGP	**a**	42	21	0.50	92	325	675	0.000	0.331	10.0	18.5	55
	b	39	2	0.05	488	637	786	0.000	0.542	10.0	10.5	67
Frag	**a**	44	8	0.18	231	362	514	0.000	0.020	15.0	31.1	485
	b	23	5	0.22	479	677	958	0.000	0.039	12.0	35.7	413
RBGP	**a**	31	3	0.10	240	397	615	0.000	0.768	8.0	14.2	150
	b	54	10	0.19	204	477	648	0.000	0.501	11.0	22.4	142
eRBGP	**a**	31	31	1.00	30	141	742	0.000	0.000	17.0	24.2	225
	b	59	59	1.00	35	197	950	0.000	0.000	18.0	21.6	226

Table II
THE GENETIC PROGRAMMING APPROACHES IN DIRECT COMPARISON IN THE ELECTION EXPERIMENTS.

the time needed for convergence and removing the dependency on randomness in the message delays. Its behavior exhibits a certain resemblance with Chang and Roberts's message extinction algorithm [107], with the difference that it does not terminate. The evolution of near-adequate terminating election algorithms is discussed in [22].

4) Results: We define a run as successful if it yielded at least one adequate program, i. e., an individual with optimal values in the functional criteria. In this evaluation we take all experimental runs into consideration which completed at least 750 generations or were successful earlier.

In Table II we show the number $\#s$ of such successful runs in relation with the number $\#r$ of total runs.[5] We furthermore distinguish the minimum ($\breve{st}$), mean ($\overline{st}$), and maximum generation ($\widehat{st}$) in which the first adequate program was found in the successful runs. Additionally, Table II shows the optimal (minimal, $\breve{f}_{el}$) and mean ($\overline{f_{el}}$) values of the functional and the non-functional objective f_{ps}.

For the sake of completeness, the mean T (in seconds) of the time consumed by all runs of a configuration is also illustrated. Although the different GP approaches have significantly different runtime requirements resulting from the complexity of simulating the corresponding virtual machines, this only becomes important when performing many runs. Even the slowest GP method (Frag) with around eight minutes per run is feasible for practical purposes from this perspective.

From Table II, it becomes obvious that eRBGP is a very strong Genetic Programming approach for the election problem. We used statistical tests in order to verify this hypothesis

[5] Because of the different runtime of the experiments and the way in which we utilized the cluster, $\#r$ is not the same for all configurations. This plays no role in the statistical evaluation.

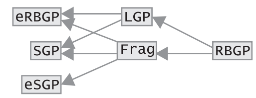


Fig. 11.a: According to $f_{el.a}$ in case *a*.

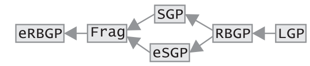


Fig. 11.b: According to $f_{el.b}$ in case *b*.

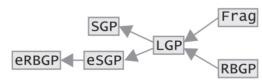


Fig. 11.c: According to *s/r* in case *a*.

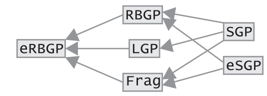


Fig. 11.d: According to *s/r* in case *b*.

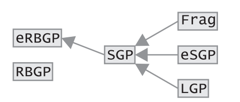


Fig. 11.e: According to *st* in case *a*.

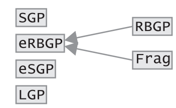


Fig. 11.f: According to *st* in case *b*.

Fig. 11. Partial orders of the GP approaches according to their performance in the election experiments.

and illustrated the results in Fig. 11. The diagrams there represent partial orders where an arrow from an approach *A* to approach *B* means that *B* beats *A* in the corresponding criterion with a probability to err of less than 2% in a two-tailed test. The success rates were compared with Fisher's exact test and for comparing *st* we used the Mann–Whitney U test.

From these diagrams, it becomes obvious that eRBGP is never beaten by any other approach in terms of the number of generations *st* needed to find a solution and the solution quality in terms of the functional objective values. In three of the compared criteria, it significantly outperforms all other approaches.

5) Summary: In this first experiment, all six Genetic Programming approaches were able to evolve the desired distributed algorithms. The synthesized programs are fully adequate and would work perfectly well in practical scenarios.

The experiment also shows that different program representations lead to different results and success probabilities. The two standard Genetic Programming methods SGP and eSGP, for instance, both solved the problem *a)* with high success rates but could not deal with *b)* properly. The additional condition required for the transition from the solution for *a)* (Fig. 9) to the algorithm for the latter task (Fig. 10) leads to an increase in problem hardness which could not efficiently be dealt with at the small population size 512. The partial orders of the GP approaches according to their performance given in Fig. 11 give no clear preference to either SGP or eSGP.

The relation of the two rule-based approaches draws a very different picture. eRBGP solves all problems in all cases and practically dominates all other methods which, in turn, outperform plain RBGP most of the time. The former shows that the targeted design of representations can indeed lead to significantly better results. The ability of rule-based GP to duplicate rules and subsequently specializing them as well as the robustness against rearrangement of the instructions clearly pays off. Forcing it into the rigid two-condition rule structure of RBGP seemingly nullifies these advantages, as we anticipated in Section V-B2.

The `Frag` and the `LGP` method show moderate success rates. The reason why the `Frag` programs never exhibited a stochastic behavior as described in Fig. 9 is that an adequate algorithm with such a behavior cannot be expressed with the instruction set chosen in our experiment. Each adequate program must multiply `elected` Fraglets (at least via communication) and therefore, must possess a corresponding consuming rule. The deterministic `max` Fraglet is the only reaction in the instruction set which can be used for that purpose and it will always lead to the node with highest ID being elected.

D. Experiment 2: Critical Section

In the next set of experiments, we tackle a problem where, on one hand, the corresponding man-made solutions are more complex than those for the election problem. On the other hand, these solutions only implicitly involve computing and ordering of certain numbers, their quality is purely based on behavioral aspects.

This second problem is the mutual exclusion at the distributed critical section. The term critical section was coined by Dijkstra for the program code accessing a shared resource. He realized that software engineers must ensure *mutual exclusion*, i. e., guarantee that at most one process may execute its critical section at a time. Developing mutual exclusion algorithms for distributed systems is cumbersome since the processes are running concurrently on different nodes and have to communicate by the means of message exchange in order to cooperatively decide which node may enter its critical section. The first efficient distributed algorithms for mutual exclusion at a critical section were introduced by Lamport [64] in 1976 and by Ricart and Agrawala [110] in 1981, followed by Maekawa's optimal solution in terms of the number of exchanged messages in 1985 [111].

A simple way to implement mutual exclusion would be to first elect a leader node in the network and then let this node decide who can utilize the critical section. Also, the network could repeatedly elect nodes which then can enter the critical section. Evolving such algorithms would mean to provide functionality surpassing election capability and hence, be harder than solving a single election problem. Therefore, one would expect that it should be a harder task than election. Furthermore, it requires two functional objectives, as we will show in the following section.

1) Objective Functions: The goal of this experiment is to evolve algorithms which *(a)* ensure mutual exclusion of the access to a shared resource as good as possible and *(b)* allow the processes to access this resource as often as possible. The synthesized programs are to follow the scheme used by Dijkstra and try to access the critical section in an infinite loop. Therefore, the program representations listed in Section VI-A are extended with the instruction `enterCS`. `enterCS` places the invoking node into ***passive*** mode for a randomly selected number w of time steps. This sleeping time is a simulation for accessing the shared resource and doing something useful with it. The node cannot be woken up during the w steps by any other event (such as incoming messages). Afterwards, it will resume execution normally. Hence, Genetic Programming

Fig. 12. $f_{use}(x) \longleftarrow \text{csFunctionalObjectiveUse}(x, \mathbf{N})$

Input: x: the simulated program
Input: $\mathbf{N}$: the network after the simulation has ended
Data: $check, k, total$: temporary variables
Output: $f_{use}(x)$: the objective value x simulated in $\mathbf{N}$

begin
$total \longleftarrow 0$
$check \longleftarrow (0,0,0,0,0)$
//iterate over the nodes in the network
for all n $\in$ **N do**
  //erroneous networks receive worse possible fitness
  if n$.state =$ ***erroneous*** **then**
    return $+\infty$
  end if
  //extract number of resource accesses
  $k \longleftarrow \text{csTimes}(\mathbf{n})$
  $total \longleftarrow total + k$
  //the cs accesses relevant for fairness are limited to 5
  for $k \longleftarrow \min\{k, 5\}$ **down to** 1 **do**
    $check[k] \longleftarrow check[k] + 1$
  end for
end for
$res \longleftarrow 0$
for $k \longleftarrow 5$ **down to** 1 **do**
  if $check[k] > 1$ **then**
    $res \longleftarrow res + check[k] - 1$
  end if
end for
//compute final objective value
return $1 - \frac{res + 1 - \frac{1}{total+1}}{5*(\text{numNodes}(N)-1)+1}$
end

is to find a way to embed calls to `enterCS` into properly synchronized loops.

In this experiment, two objectives (f_{col} and f_{use}) focusing on functional adequacy are used together with the non-functional program size criterion f_{ps} defined in Section VI-C1.

f_{col}: A collision has occurred in a time step i when two processes A and B both have entered the critical section (by calling `enterCS`) without leaving it yet. k processes can cause $0.5\,k(k-1)$ collisions in each such step. We define the objective function f_{col} as the total number of collisions during the simulated time divided by the maximum possible number of collision, i. e., normalized into $[0,1]$. This function is again subject to minimization and we added a penalty of 1 before normalization if the critical section is not utilized during the simulation.

f_{use}: Still, an optimal value of f_{col} may be reached by electing a leader and only allowing this node to access the shared resource. A criterion for encouraging frequent usage of the critical section is required. The algorithm specified in Fig. 12 examines the observed behavior of a program x in a simulated network $\mathbf{N}$. f_{use} is a mapping of a m-dimensional space to the real interval $[0,1]$ (where $m = \text{numNodes}(\mathbf{N})$ is the number of nodes in $\mathbf{N}$). f_{use}, subject to minimization, penalizes exactly the two non-solutions mentioned above. Trivial programs only achieve objective values very close to 1. Assume that $total \to +\infty$ in Fig. 12 but $check[k] = 1\ \forall k \in 1..5$, then f_{use} evaluates to $\frac{5m-5}{5m-4}$. For a network size $m = 4$, f_{use} becomes $\frac{15}{16} = 0.9375$ and for $m = 23$, $f_{use} = \frac{110}{111} = 0.\overline{990}$. The worst possible fitness value in cases where at least two nodes enter

```
function_0:          //invoked on startup
 push 0              //push parameter for function 3
 call 3              //call function 3
 l[2] = pop          //obtain return value of funct. 3
 push 0              //push parameter for function 3
 call 3              //call function 3
 l[0] = pop          //obtain return value of funct. 3
 l[4] = 1 or l[0]    //waste time computing something
 l[3] = not l[4]     //waste time computing something
 l[1] = l[2] - l[3]//waste time computing something
// pattern repeated multiple times: push something,
// call 3, pop something, compute something
 l[1] = pop          //obtain return value of
      function 3
 push l[0] * l[1]    //push data to be sent
 send                //send message
 enterCS             //enter critical section
 push l[0]           //push return value
```

```
function_1:          //invoked when a message comes in
 // 43 lines of useless computation for delaying
 // the execution
```

```
function_2: //invoked when cs is left,
            //postponed if message is pending
 enterCS             //enter critical section (1)
 enterCS             //enter critical section (2)
 enterCS             //enter critical section (3)
 enterCS             //enter critical section (4)
 push l[0]           //push return value
```

```
function_3:          //no ADF, added by GP
 push -1             //push value to be sent
 send                //send message
 push 0              //push value to be sent
 send                //send message
 push l[0]           //push return value
```

Listing 8. The `LGP` phenotype of the `eSGP` genotype: ($f_{col} = 0.0$, $f_{use} = 0.1529$, $f_{ps} = 98$).

the critical section ($total = 2$, $res = 1$) is $\frac{5m-5.5}{5m-4}$ which is $\frac{29}{32} = 0.906\,25$ for a network consisting of $m = 4$ nodes and $\frac{73}{74} = 0.9\overline{864}$ for $m = 23$ nodes.

2) Experimental Settings: The six Genetic Programming approaches defined in Section VI-A were applied to the critical section problem under the three objective functions $\vec{f} = \{f_{col}, f_{use}, f_{ps}\}$. The `SGP`, `eSGP`, and `LGP` approaches were provided with an additional automatically defined function which is asynchronously called after `enterCS` returns. `RBGP` and `eRBGP` received a `csLeft` symbol set to `1` when the critical section was left and in the Fraglet stores of the `Frag` approach, a symbol [`csLeft`] was injected in this case.

For each algorithm evaluation, twenty scenarios with networks consisting of between four and 23 virtual machines were executed. We define a program x as *marginally fair* if it reaches $f_{use}(x) \leq 0.906\,25$. This threshold is the lowest boundary for a network with four nodes where at least two nodes have accessed the critical section, as previously shown. We structured the networks in the simulations in a fully-meshed topology where each node can directly communicate with every other one.

3) Evolved Algorithms: In Listing 8 we specify the fairest program which evolved from all experimental runs. In the syntax used, access to the ith cell in local memory is denoted by `l[i]` and to global memory as `g[i]`. Like virtually all evolved adequate solutions for this problem, it follows the scheme of mutual stalling and delaying given in Fig. 13. Here,

Fig. 13. `criticalSectionProtect`

begin
Subalgorithm `main`
begin
//stall other nodes by sending lots of messages
enterCS()
end
Subalgorithm `onMessage`(message)
begin
//perform length computation
//maybe send messages
//invoke procedures which either do "enterCS" *or call* `main`
end
end

the communication medium is used as signaling device for synchronization. A node only enters the critical section if it did not receive any message for some time. The other nodes try to prevent this by frequently broadcasting messages. Whether a node can access the "shared resource" therefore again at least partly depends on the randomness of the message latency and parallelism.

At first glance, the evolved algorithms do not equal any other common method for protecting the critical section in distributed systems. This, however, is not true: They are *Carrier Sense Multiple Access* (CSMA, [65, 112]) protocols where the message sending in case of a free communication medium has been replaced with entering the critical section and listening whether the channel is busy is exchanged with checking whether messages were received.

One of the interesting features of the evolved algorithms in the `SGP` and `eSGP` representation is that they do not involve explicit loops although special language constructs for such structures were available. Even more interesting is that `SGP` and `eSGP` most often use code without any *conditional* branches. This trend, which similarly has been reported by Paterson [113] and Wàn et al. [114], leads to the impression that these programs seem to be trivial or overfitted.

Yet, they are not. Despite never making use of any sophisticated feature, they achieve full functional adequacy in more than twenty randomly created scenarios, usually over many generations in the EA, and perform adequately if tested in scenarios (with a similar framework of parameters) not used for training. Functional adequacy here involves both, proper protection of the critical section and a fair resource utilization – the value $f_{use} = 0.1529$ for Listing 8 is indeed very good. Still, these programs are not correct solutions in the Dijkstra-sense, although – if configured properly – they would lead to satisfactory results if the application scenario allows a certain, low degree of uncertainty.

4) Results: A run of the critical section experiment was considered as successful if it yielded at least one individual x with the optimal value in the collision-minimizing objective function and which is at least marginally fair. Here we take all runs into consideration which have finished 700 generations.

In Table III, $\#r$ is again the number of total runs for each configuration. We further list the number $\#p$ (and proportion p/r) of runs which achieved to evolve individuals which could protect the critical section although not necessarily in a fair

Case	#r	#p	p/r	#s	s/r	$\overline{[f_{col}]}_z$	$\widetilde{[f_{use}]}_z$	$\overline{[f_{use}]}_z$	T[s]
SGP	38	38	1.00	37	0.97	$3 \cdot 10^{-10}$	0.185	0.364	286
eSGP	46	46	1.00	45	0.98	$1 \cdot 10^{-7}$	0.153	0.367	299
LGP	32	29	0.91	8	0.25	$4 \cdot 10^{-4}$	0.478	0.811	75
Frag	27	14	0.52	0	0.00	0.002	0.884	0.899	446
RBGP	23	6	0.26	0	0.00	0.004	0.742	0.870	424
eRBGP	37	5	0.14	0	0.00	0.004	0.807	0.893	321

Table III
THE GENETIC PROGRAMMING APPROACHES IN DIRECT COMPARISON IN THE CRITICAL SECTION EXPERIMENTS.

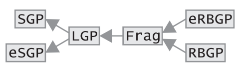


Fig. 14. Partial order of the GP approaches according to $\widetilde{[f_{col}]}_z$.

way. T is again the mean runtime.The two Standard Genetic Programming approaches achieved this in all experiments and LGP has a very high p/r rate. Both Rule-based Genetic Programming methods fall behind the Frag approach in this measure.

$\#s$ is the number and s/r is the proportion of successful runs, i. e., runs where at least one individual evolved for which p holds and where the critical section was accessed by more than one node. Such programs were only found with SGP, eSGP, and LGP.

We use the subscript z to annotate values belonging to individuals which possess at least marginal fairness. We determined the minimally-fair individuals with the best values of f_{col} for every single run and listed their mean value in the collision objective functions $\overline{[f_{col}]}_z$ and the minimum $\widetilde{[f_{use}]}_z$ and mean $\overline{[f_{use}]}_z$ of the fairness-of-use criterion. Again, SGP, eSGP, and LGP perform best in these measures.

We checked the significance of the trends reported above using a two-tailed Mann–Whitney U test with 2% significance level and illustrated them in Fig. 14. Here, the Standard Genetic Programming approaches dominate LGP which, in turn, dominates the other approaches.

5) Summary: The outcomes of the critical section experiments were surprising at the first glance, especially in the light of the results of the election experiments.

SGP and eSGP have effectively changed places with eRBGP and RBGP and now perform significantly better. The cause for these overtakes is the ability of SGP, eSGP, and LGP to create long sequences of instructions for slowing down the execution. Twelve out of the 15 instructions of the LGP language (in which the phenotypes of SGP and eSGP are specified) can be randomly inserted into the code in order to do this. Therefore, the Genetic Programming process, once it has identified the CSMA communication scheme, only has to adjust their number to the right amount in order to achieve good fitness. RBGP, eRBGP, and to a lesser amount, the Frag method, cannot do this. There is no such thing as *sequential* instruction processing in these approaches. Rules in Rule-based Genetic Programming are triggered by conditions and Fraglets react with each other. Thus, stalling and delaying as used by the algorithm defined in Fig. 13 becomes much more complicated. The experiment also provided the second indicator that tree-based SGP methods outperform pure linear Genetic Programming in this problem domain.

This experiment has also shown that Genetic Programming can evolve distributed algorithms in a multi-objective scenario. It is well known that the Pareto front may grow exponentially [115] with the number of objectives. We already considered the critical section task itself to be harder than the election problem. Additional to this basic complexity, the number of functional objectives has increased to two while the non-functional criterion f_{ps} was retained. A general requirement for evolving programs for sensor networks, for instance, would be to also minimize the energy consumption of the nodes. With solving the critical section, it became apparent that Genetic Programming is able to handle more than two objective functions.

This experiment is the second account for the evolution of algorithms which are adequate and may even work sufficiently well in practical scenarios. It is, however, also the second account for the evolution of algorithms which are not correct and differ much from what an engineering approach would yield.

E. Experiment 3: DGCD

For both, the election and the critical section problem, manually derived solutions exhibit a certain complexity. They involve iterative, distributed computations which come to precise results. Yet, Genetic Programming has dodged this complexity by evolving behaviors which – although being robust and functioning adequately – have simpler structure. On one hand, this showed that it is possible to solve common tasks in distributed systems. On the other hand, it did not lead to results anticipated from an engineering perspective (which is not necessarily bad). With the third series of experiments, we want to find whether problems can be solved where the actual computation cannot be simplified, emulated, or mocked-up with simplified behavior. We therefore picked the problem of the distributed computation of the greatest common divisor.

For two integer numbers $a, b \in \mathbb{N}_1$, the greatest common divisor (GCD) is the largest number $c \in \mathbb{N}_1$ that divides both, a and b. The GCD of two numbers can be computed with the Euclidian algorithm [116]. Mattern used a distributed version of this procedure (specified in Fig. 15) as an example in his foundational book [66]. Here, each node $\mathbf{n}$ of the network $\mathbf{N}$ starts with an own number $\mathbf{n}.num$ which will be its first guess about what the GCD of all numbers distributed over the network is. Step by step, the $\mathrm{gcdVal}(\mathbf{n})$ values of all nodes $\mathbf{n} \in \mathbf{N}$ will converge to the real GCD.

1) Objective Functions: The initial situation of a network $\mathbf{N}$ be that each of its nodes $\mathbf{n} \in \mathbf{N}$ knows exactly one number $\mathbf{n}.num \in \mathbb{N}_1$. We wish to evolve programs which, if executed on these nodes, compute the greatest common divisor $corr = \mathrm{gcd}_{\forall \mathbf{n} \in \mathbf{N}}\, \mathbf{n}.num$ of all the numbers distributed over the network. This number should be stored in a special variable or symbol $\mathrm{gcdVal}(\mathbf{n})$ on each of the nodes $\mathbf{n}$. The non-continuous nature of the GCD problem prevents any approximative results and strictly limits the set of possible solutions. We therefore

Fig. 15. $\text{gcdVal}(\text{self}) \longleftarrow \text{distributedGCD}$

Input: $\text{self}.num \in \mathbb{N}_1$: the own number
Output: $\text{gcdVal}(\text{self}) \in \mathbb{N}_1$: the result of the GCD computation

begin
Subalgorithm `main`
begin
$\text{gcdVal}(\text{self}) \longleftarrow \text{self}.num$
$\text{sendMessage}(\text{gcdVal}(\text{self}))$
end

Subalgorithm `onMessage`(message)
begin
$\text{gcdVal}(\text{self}) \longleftarrow ((\text{gcdVal}(\text{self}) - 1) \bmod \text{message}_{[0]}) + 1$
$\text{sendMessage}(\text{gcdVal}(\text{self}))$
end
end

can expect that it will be the hardest one amongst the three tasks presented in this article.

Another aspect of GP is that we generally can derive multiple objective functions for the same behavior specification. In the critical section experiment, for instance, we could have replaced f_{use} with a function which returns the arithmetic mean of the number $\text{csTimes}(\mathbf{n})$ of times the nodes $\mathbf{n} \in \mathbf{N}$ have executed their critical sections. In the distributed greatest common divisor experiments, we want to also test how different functional criteria for the same behavior influence the results of the evolution, whether effort put into defining a criterion which also rewards partial solutions actually pays off or not. We therefore define two functional objective functions gcd.1 and gcd.2:

gcd.1 (specified in Fig. 16) rewards programs which decide for return values "gcdVal" divisible by the correct result *corr*. This reward increases when the algorithms get closer to the real result. All values greater or equal to the minimum initial number $\min_{\forall \mathbf{n} \in \mathbf{N}} \mathbf{n}.num$ receive the same default fitness in order to prevent the evolution of algorithms which simply converge to this number.

gcd.2 (specified in Fig. 17) provokes the *all-or-nothing* problem by only giving rewards if a node has found the correct GCD. $f_{gcd.2}$ provides little more information than a Boolean decision criterion about the correctness of a program. Although it can take on more than two values since all nodes in the simulated networks are considered separately, it can be assumed that only correct (or close-to-adequate) algorithms can score results lower than 1. Hence, f_{ps} should be the driving force of the evolution and many very small programs are likely to occur. Since all programs solving the GCD problem adequately have a certain minimum size, we set a lowest boundary of 25 for f_{ps} under which it cannot drop.[6]

2) Experimental Settings: We used the same settings as for the election experiment (see Section VI-C2) except that we evolved the algorithms in a rectangular topology where each node had up to four neighbors to communicate with and provided four variables to the `RBGP` and `eRBGP` approaches and four global and local memory cells to the `SGP`, `eSGP`, and `LGP` Genetic Programming methods.

[6]This boundary was determined in the first experimental series with *1)* where it was not yet applied.

Fig. 16. $f_{gcd.1}(x) \longleftarrow \text{gcdFunctionalObjective1}(x, \mathbf{N})$

Input: x: the simulated program
Input: $\mathbf{N}$: the network after the simulation has ended
Output: $f_{gcd}(x)$: the objective value x simulated in $\mathbf{N}$

begin
$values \longleftarrow ()$
$ownNum \longleftarrow 0$
$minNum \longleftarrow +\infty$
$corr \longleftarrow 0$
for all $\mathbf{n} \in \mathbf{N}$ **do**
  if $\mathbf{n}.num < minNum$ **then**
    $minNum \longleftarrow \mathbf{n}.num$
  end if
  if $\mathbf{n}.state \neq$ ***erroneous*** **then**
    //store gcd value
    $values \longleftarrow \text{addListItem}(values, \text{gcdVal}(\mathbf{n}))$
    //count nodes which think gcd = own number
    if $\text{gcdVal}(\mathbf{n}) = \mathbf{n}.num$ **then**
      $ownNum \longleftarrow ownNum + 1$
    end if
  else
    //errors are treated like gcd=own number
    $ownNum \longleftarrow ownNum + 1$
  end if
  if $corr = 0$ **then**
    $corr \longleftarrow \mathbf{n}.num$
  else
    $corr \longleftarrow \gcd(corr, \mathbf{n}.num)$
  end if
end for
$div \longleftarrow \frac{1}{2} * \frac{1}{minNum - 1 - corr}$
$tv \longleftarrow \frac{1}{2} * \frac{\text{numNodes}(\mathbf{N}) - ownNum + 1}{\text{numNodes}(\mathbf{N}) + 1}$
$r \longleftarrow 0$
for $i \longleftarrow \text{len}(values) - 1$ **down to** 0 **do**
  //if the value could be some sort of correct interim result...
  if $(values_{[i]} > 0) \wedge (values_{[i]} \bmod corr = 0)$ **then**
    if $values_{[i]} \geq minNum$ **then**
      $r \longleftarrow r + tv$
    else
      $r \longleftarrow r + 1 - (div * values_{[i]} - corr)$
    end if
  end if
end for
return $1 - \frac{r}{\text{numNodes}(\mathbf{N})}$
end

Fig. 17. $f_{gcd.2}(x) \longleftarrow \text{gcdFunctionalObjective2}(x, \mathbf{N})$

Input: x: the simulated program
Input: $\mathbf{N}$: the network after the simulation has ended
Output: $f_{gcd}(x)$: the objective value x simulated in $\mathbf{N}$

begin
$corr \longleftarrow 0$
for all $\mathbf{n} \in \mathbf{N}$ **do**
  if $corr = 0$ **then**
    $corr \longleftarrow \mathbf{n}.num$
  else
    $corr \longleftarrow \gcd(corr, \mathbf{n}.num)$
  end if
end for
$r \longleftarrow 0$
for all $\mathbf{n} \in \mathbf{N}$ **do**
  if $(\mathbf{n}.state \neq$ ***erroneous***$) \wedge (\text{gcdVal}(\mathbf{n}) = corr)$ **then**
    $r \longleftarrow r + 1$
  end if
end for
return $\frac{\text{numNodes}(\mathbf{N}) - r}{\text{numNodes}(\mathbf{N})}$
end

$$
\begin{array}{rcrl}
(\mathtt{start}_t + (\mathtt{16} * \mathtt{in2})) & \Rightarrow & \mathtt{send} & \\
\mathtt{out2}_t & \Rightarrow & \mathtt{a}_{t+1} = & ((\mathtt{-4} \geq \mathtt{in2}_t) + \mathtt{out2}_t) \\
\mathtt{id}_t & \Rightarrow & \mathtt{out2}_{t+1} = & (\mathtt{in2}_t + (\mathtt{in2}_t + \mathtt{id}_t)) \\
\mathtt{a}_t & \Rightarrow & \mathtt{out2}_{t+1} = & (\mathtt{in2}_t \,\%\, \mathtt{a}_t)
\end{array}
$$

Listing 9. One evolved eRBGP program for problem case *1)* in the 512 individual population $\left(f_{gcd.1} = 0, f_{ps} = 30\right)$.

Fig. 18. gcdVal(self) $\longleftarrow$ distributedGCDEvolved

Input: self.$num \in \mathbb{N}_0$: the own number
Output: gcdVal(self) $\in \mathbb{N}_0$: the result of the GCD computation

begin
Subalgorithm `main`
begin
gcdVal(self) $\longleftarrow$ self.num
sendMessage(gcdVal(self))
end

Subalgorithm `onMessage`(message)
begin
$tmp \longleftarrow$ gcdVal(self) mod message$_{[0]}$
if $tmp \neq 0$ **then**
  gcdVal(self) $\longleftarrow tmp$
  sendMessage(gcdVal(self))
end if
end
end

```
function_0:            //called at startup
 g[1] = g[0]           //g[1]=own number given in g[1]
 call 2                //this call+pop instruction is
 l[0] = pop            //for stalling, so g[1]=g[0]
 push not l[0]         //push -1
 send                  //send -1
 push l[0]             //useless
function_1:            //invoked when message comes in
 l[2] = 0 + g[1]       //l[2]=own number g[1]
 l[1] = l[0] - g[1]    //l[1]=received-own number
 l[0] = not l[1]       //l[0]=ones complement of l[1]
 l[1] = g[1] * 1       //l[1]=own number g[1]
 l[3] = l[0] mod l[1]//l[3]=l[0] mod own number g[1]
 g[1] = l[2] - l[3]    //own number=own number-l[3]
 call 2                 //useless
 l[3] = pop            //useless: result of func_2=0
 l[2] = l[0] + l[3]    //useless
 push not l[2]         //useless
 call 2                 //useless
 l[3] = pop            //useless
 l[2] = l[0] + l[3]    //useless
 push not l[2]         //push ones compl. of l[2]
 send                  //send ones compl. of l[2]
 push l[0]             //useless
function_2:            //additional function
 l[0] = not l[0]       // <delaying code>
 l[1] = g[1] * 1       //this function is basically
 l[3] = l[0] mod l[1]//used for stalling the
 l[2] = not l[3]       //execution  of the program
 l[1] = 3 * g[1]       //in order to ensure that
 l[0] = l[1] mod l[1]//g[1]=g[0] in function_0 for
 l[3] = l[0] * l[0]    //all nodes of the network
 l[1] = l[2] + l[3]    // </delaying code>
 push l[0]             //return the input or 0
                       //if no parameter was supplied
```

Listing 10. The LGP phenotype of an evolved eSGP solution $\left(f_{gcd.1} = 0, f_{ps} = 31\right)$.

We repeated the experiments for the different functional optimization criteria *gcd.1* and *gcd.2*. While ensuring that in each simulated scenario, the correct result of the GCD was different from one. We did not include the `Frag` approach in our experiments because this would have required too much of a deviation from the instruction set given in Listing 5 and in [100] used until now.

3) Evolved Programs: `eRBGP` solved the distributed GCD problem most often. One of the adequate programs it was able to synthesize is shown in Listing 9, where the initial number of a node is its ID $\mathtt{id}_t$ and the result of the distributed computation is expected to occur in variable $\mathtt{a}_t$. $\mathtt{in}_t$ and $\mathtt{out}_t$ are again symbols where incoming and outgoing message content will be put in.

In Mattern's method, the values of gcdVal($\mathbf{n}$) were prevented from becoming zero by adding one to the result of the modulo division used to compute the GCD step by step (line 11 in Fig. 15). Like the program in Listing 9, most of the algorithms evolved with `eRBGP` follow a different approach given as Fig. 18. They store the modulo of the current estimate of the GCD and the received value in a temporary variable. This variable is then written back to the estimate if it is not zero.

Listing 10 reveals a problem with which a reader may find herself confronted when analyzing an evolved program. Even with a simple syntax, Genetic Programming may produce incomprehensible code because of the lack of targeted design and intention. For the `SGP`, `eSGP`, larger `RBGP`/`eRBGP`, and `Frag` programs, this is even much worse which is also the reason why we stated the evolved results in form of algorithms and only gave a few, obvious examples for the original code.

Although being a valid solution, the inner workings of Listing 10 are camouflaged by the way in which variables and modules are utilized. In the `LGP`/`SGP`/`eSGP` approaches to the GCD problem, the own number of a node is supplied to the program in the first cell `g[0]` of global memory and the result is expected in the second cell `g[1]`. The evolved program connects all four local and the relevant two global variables with none-obvious calculations. Finding out which of the instructions in Listing 10 are useful and which are not is actually complicated.

At first glance, `function_2`, for instance, seems to play an important role in the GCD computation since it is invoked from multiple locations, receives values extracted from the received messages as parameters, and contains modulo division operations. It possibly was important during the early phase of the evolution and became degraded as more efficient code evolved. Now, it is just used to delay the execution in order to ensure that the first instruction of `function_0` (`g[1]=g[0]`) has taken place before the real distributed computation begins, which is performed by the first six lines of `function_1`. Furthermore, the GCD estimates are exchanged between the nodes in their ones complements for no particular reason. Because of the vulnerability of the first value assignment, this program (unlike Listing 9) is not correct but only adequate.

4) Results: In Table IV, we have noted the same measurements as provided for the election experiment in Table II. The number $\#s$ and proportion s/r of successful runs in relation with the number $\#r$ of total runs is very low in all configurations.

	Case	#r	#s	s/r	$\breve{st}$	$\overline{st}$	$\widehat{st}$	$\frown{f_{gcd}}$	$\overline{f_{gcd}}$	T[s]
SGP	1	30	0	0.00	-	-	-	0.496	0.500	117
	2	153	0	0.00	-	-	-	0.630	0.997	18
eSGP	1	53	1	0.02	912	912	912	0.000	0.483	160
	2	153	0	0.00	-	-	-	1.000	1.000	17
LGP	1	41	0	0.00	-	-	-	0.445	0.499	36
	2	161	0	0.00	-	-	-	0.986	1.000	1
RBGP	1	54	0	0.00	-	-	-	0.500	0.500	129
	2	149	0	0.00	-	-	-	0.990	1.000	37
eRBGP	1	74	5	0.07	236	406	649	0.000	0.440	57
	2	147	1	0.01	880	880	880	0.000	0.985	20

Table IV
THE GENETIC PROGRAMMING APPROACHES IN DIRECT COMPARISON IN THE GCD EXPERIMENTS.

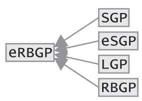


Fig. 19. Partial orders of the GP approaches according to $\frown{f_{gcd.1}}$ for case *1)*.

Only eSGP and eRBGP have a non-zero success ratio and only eRBGP finds an adequate program more than once. For the second way of expressing the functional optimization criterion ($f_{gcd.2}$), only eRBGP – seemingly accidentally – finds one single solution.

In terms of the minimum ($\breve{st}$), mean ($\overline{st}$), and maximum generation ($\widehat{st}$) in which the first adequate program was found in the successful runs, eRBGP is therefore again best. The same goes for the minimum and mean functional objective values ($\frown{f_{gcd}}$, $\overline{f_{gcd}}$) of the individuals with the best functional objective values of each run. Generally, these values are much better for case *1* than for case *2* of the GCD problem.

Like in the election experiment, we have compared the success rates of the different Genetic Programming approaches with Fisher's exact test and the best values $\frown{f_{gcd.1}}$ of the functional objective functions with the Mann–Whitney U test. For both tests, we again used the two-tailed variants with a significance level of 2%. Because of the low success rates of the experiments, most trends turned out to be insignificant and only few, strong relations (illustrated in Fig. 19) could be confirmed.

In order to clarify the question whether Genetic Programming indeed utilizes the information gained from sampling the search space *efficiently*, we specified the second functional criterion $f_{gcd.2}$. Since the value of f_{ps} was bounded below, we would expect the search to behave like parallel random walks in the space of programs which are not much longer than the specified minimum size and to find solutions only very rarely.

The result of the experiments with $f_{gcd.2}$ fully meet this expectation. Only one solution was found in more than 760 runs. We compared the success rates for $f_{gcd.1}$ and $f_{gcd.2}$ and found that there only was a significant difference for eRBGP. This does not necessarily mean that the other approaches not perform any better than (bounded) random walks – with more runs, the differences may become significant – but it means that eRBGP definitely does.

5) Summary: Our experiments targeting the evolution of algorithms for computing the greatest common divisor in a distributed way led us to two conclusions. First, the GCD task seems to be the most complicated one of all the problems which we have tested. Although two GP approaches found solutions for it, the success rates of the experiments are very low. Second, only eRBGP is significantly better than a (bounded) random walk in this domain. Here, it is not sufficient to combine random instructions in a way similar to the CSMA method created by eSGP in Section VI-D3. Instead, a well defined computation has to be assembled. Similar to the second election experiment where clear criteria for the computation result existed, eRBGP dominated the other approaches, although it still had a low success rate.

On one hand, these results reveal the hardness of needle-in-a-haystack problems which we anticipated in Section IV-C and to which the GCD task seems to belong for the traditional GP representations. On the other hand, it is a further indicator for the strength of eRBGP and justifies its design outlined in Section V. Whether a problem has or has not NIAH characteristics depends on the program representation to a large degree and, for the rule-based program structure, the GCD problem is indeed easier to tackle. Furthermore, the results evolved by eRBGP, such as the program illustrated in Listing 9, this time not only are *adequate* but *correct*.

VII. Conclusions and Future Work

In the introduction, we motivated that Genetic Programming could be used as a foundation for a new design approach for distributed algorithms. We analyzed the possible difficulties of this problem domain and developed two program representations from which we expected that they may have beneficial features in this context. We applied these representations to three example problems from distributed computing and compared their performance to four other GP approaches.

A. Choice of Scenarios

The reasons for applying our method to three different problems were twofold. First, when evaluating new techniques – be it for software design or in any other area – testing them on one or two scenarios only cannot be considered a significant sample and only provides weak indications for their utility. By presenting three test problems, we aimed at striking a balance between providing more evidence for the utility of GP and retaining a suitable length for an article. Second, the three scenarios each have characteristic features and pose different problems.

The election task outlined in Section VI-C is a problem of moderate hardness which has already been tackled in the past [26, 63]. Yet, this is the first study which compares the performance of different GP approaches in this domain. The second advantage of this problem is that the spectrum of achieved success rates is wide enough to allow fine-grained and yet statistically significant comparisons amongst them (see Fig. 11 as opposed to Fig. 19).

Of all three problems, the critical section task is the only one which does not necessarily involve computing numerical values, although all practical solutions for fully-meshed networks do this. Also, in this scenario we tested the influence of multiple functional criteria on the optimization process. The results of this experiment were surprising at first glance, since only here the Standard Genetic Programming approaches outperformed Rule-based Genetic Programming significantly. Analyzing it revealed that the strength of rule-based programs in some cases can also be their weakness: They do not have explicit execution sequences.

Finally, in the distributed GCD experiment, we had a very tight specification of what *adequate* means. Simple solution structures circumventing the complexity of man-made solutions here cannot achieve adequacy which turns the problem into a needle-in-a-haystack setting. In this scenario, we also introduced an all-or-nothing objective which allowed us to check whether GP actually can beat random walks in such scenarios.

The focus of our work presented here was on running large-scale experiments with significant outcomes. Even though using large populations is quite common in Genetic Programming [117–120], we intentionally utilized relatively small populations of 512 individuals. This makes our results highly relevant, because even better outcomes can be expected if bigger populations are used for actual applications.

B. Experimental Results

At the end of each experiment section, we provided a summary of conclusions drawn from it. Here, we present five general and important lessons learned.

First, with the work outlined here we have shown that *adequate* algorithms for distributed computations can be evolved with Genetic Programming. This is true despite the fact that such optimization tasks exhibit a variety of problematic facets discussed in Section IV and are certainly amongst the most difficult ones.

Second, our study is the only one which compares the performance of diverse GP approaches in the area of evolving distributed algorithms. Based on our experiments, we found that the choice of Genetic Programming approach has a tremendous influence on the chance of success. Different program representations can lead to different results, to different chances of success, and may need different numbers of generations to converge. Also, there likely is no single best program representation [121]. While our `eRBGP` was very powerful in the tasks where algorithms computing single numbers were to be evolved, it failed in the critical section domain. Here, `SGP`, `eSGP`, and `LGP` performed much better since they, unlike the rule-based or Fraglets approaches, are based on the concept of sequential instruction processing established in virtually all of today's computers. Yet, such structures have a high positional epistasis which renders them less efficient or even infeasible in situations where more complex algorithms are to be synthesized.

This leads over to the third lesson, to the targeted design of program representations. We designed the two new representations `RBGP` and `eRBGP` especially with this goal in mind, as described in Section V. `RBGP` has a well-defined structure which allows us to encode the programs in binary form which can be evolved with a normal GA. However, in Section V-B we pointed out that well-defined can also mean rigid and may restrict the freedom of the evolution. And indeed, lifting the structural limitations and using a tree genome for encoding the programs was extremely beneficial, even though the search space was further increased by introducing indexed memory – a feature which turned out to be only rarely used by the evolution in our experiments[7]. Still, `eRBGP` was the most successful approach (except in the critical section experiment) and is likely to perform well in many other problem domains whereas the performance of `RBGP` was sub-par.

Fourth, especially the DGCD experiment in Section VI-E substantiates the assumption that the choice of the objective functions has an extreme impact on the chance of success. All-or-nothing criteria should be avoided by all means and instead, objectives which reward partial solutions should be developed. Hence, before starting large-scale experiments, some small-scale runs should be performed for testing different criteria.

Finally, there is the question of whether indexed memory, used to achieve Turing completeness in some of the program representations, is beneficial or not. `eRBGP` often achieved much better results than `RBGP`. This may have either been rooted in its Turing completeness or in the higher degrees of freedom for constructing complex expressions. We also tested two other GP approaches which are similar and between which the main distinction is the availability of indexed memory respectively the lack of it: `eSGP` and `SGP`. Between these two methods, no significant difference in performance could be detected in most of the experiments. Therefore, we believe that the versatility provided by the tree genome is the decisive difference between `RBGP` and `eRBGP`. From this perspective, our results do not allow us to decide whether the usage of indexed memory is beneficial (in cases where it is not necessarily needed, such as in our experiments). However, we could not detect any disadvantage either – although the presence of indexed memory leads to more degrees of freedom and thus, enlarges the search space. More experiments on this issue would thus be interesting.

It should further be noted that runtime is also a concern when evolving distributed algorithms. In the work presented here, we focused on carrying out a large-scale experimental study which provides conclusive results. Therefore, we performed more experiments than what would actually be needed in a real application. This led to a runtime of over 1280 processor days distributed over a cluster in the election experiment, for instance. Time consumptions of six to ten minutes per generation for a population consisting of 512 individuals would, however, also be expected in practical scenarios.

C. Criticism of the Idea

In Section IV-C, we raised the most severe objection against the idea of evolving (distributed) algorithms: the results are

[7] when comparing `eSGP` (with indexed memory) and `SGP` (without), the differences are marginal

not necessarily correct. However, some of the larger-scale distributed system types named in the motivation of our approach tolerate a certain amount of malfunction. In sensor networks, for instance, some of the nodes will inevitably fail due to depleted batteries. In MANETs, temporary network partitions may occur. Especially for networks of these kinds, evolving algorithms may indeed be a useful software design approach. When a critical section protection scheme can fail due to network partitions, an additional small chance of failure of a CSMA-scheme due to the structure of the algorithm may not be a big problem.

For other scenarios where correctness is required, applying a subsequent model checking step *after* the evolutionary algorithm synthesis in order to weed out incorrect solutions would be a viable way to make use of our method. Generally, GP can help to open our eyes for strange but effective solutions: In the literature known to the authors, CSMA has not been considered for critical sections.

With the work presented in this article, we made a first step towards a new algorithm design method for distributed systems. We do not aim at replacing the existing approaches, but believe that we will be able to complement them. In order to achieve this goal, we openly discussed both the strengths and the weaknesses of our idea and hope that this way, an open and fruitful debate can be started.

D. Future Work

Although we provided the necessary optimization and Genetic Programming frameworks, a certain learning curve is still unavoidable. In [25, 122], we already discussed the integration of the results of Genetic Programming into a model-driven development (MDD) process. However, we would like to use MDD tools not only as backend for GP but also as frontend. The utility of our method will increase very much if it becomes possible to model the behavior of the anticipated system by defining and combining optimization criteria in a more graphical and straightforward way.

Another point worthy of further investigation is the design of low-epistatic program representations. Our Rule-based Genetic Programming method is different from Standard Genetic Programming in two aspects: 1) Its execution model is based on rules evaluated in parallel and not on instruction sequences and 2) it uses a temporary storage committed after all rule evaluations instead of "normal" memory. With our experiments, we cannot be sure which of these two points contributed most to the dominance of `eRBGP` in many of the experiments or whether it was their combination.

Especially with regard to the results of the critical section experiment, we therefore plan to introduce "transactional" memory into SGP, maybe with a special `commit` instruction. By allowing the Genetic Programming process to decide when to commit the changes to the variables, we would reduce the positional epistasis of the Standard Genetic Programming approaches. An alternative would be to introduce an automatic variable commit at the end of instruction groups (such as at the bottoms of loops). Both methods could then enable GP to produce solutions for this problem which comply better with the engineering perspective. With the presented work, we also have plenty of data to compare the performance of such new "transacted" `SGP` or `eSGP` methods with.

Acknowledgements. The authors wish to thank Prof. Xin Yao for his helpful comments on the article. Also, we wish to thank Prof. Kurt Geihs, Prof. Christian Tschudin, Dr. Lidia Yamamoto, and Thomas Meyer for their suggestions regarding the presented work. This work was partially supported by the China Postdoctoral Science Foundation (No. 20100470843) and the National Science Foundation of China (No. U0835002 and 60802036).

REFERENCES

[1] Thomas Weise and Kē Táng. Evolving Distributed Algorithms with Genetic Programming. *IEEE Transactions on Evolutionary Computation (IEEE-EC)*, 16(2):242–265, April 2012. doi: 10.1109/TEVC.2011.2112666.

[2] Andrew Stuart Tanenbaum and Maarten van Steen. *Distributed Systems. Principles and Paradigms*. Prentice Hall International Inc.: Upper Saddle River, NJ, USA and Pearson Education: Upper Saddle River, NJ, USA, March 2003. ISBN 0130888931, 0131217860, 0132392275, and 8178087898.

[3] Stefano Basagni, Marco Conti, Silvia Giordano, and Ivan Stojmenović, editors. *Mobile Ad Hoc Networking*. Institute of Electrical and Electronics Engineers (IEEE) Press: New York, NY, USA and Wiley Interscience: Chichester, West Sussex, UK, 2004. ISBN 0-471-37313-3 and 0-471-65689-5.

[4] Mohammad Ilyas and Imad Mahgoub, editors. *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*. CRC Press, Inc.: Boca Raton, FL, USA, July 28, 2004. ISBN 0849319684.

[5] Edgar H. Callaway, Jr. *Wireless Sensor Networks: Architectures and Protocols*. Internet and Communications. Auerbach Publications: Boca Raton, FL, USA, August 23, 2003. ISBN 0849318238.

[6] Richard Harper, editor. *Inside the Smart Home*. Springer-Verlag London Limited: London, UK, 2003. ISBN 1-85233-688-9.

[7] John Krumm, Gregory D. Abowd, Aruna Seneviratne, and Thomas Strang, editors. *Proceedings of the 9th International Conference on Ubiquitous Computing (UbiComp'07)*, volume 4717/2007 of *Information Systems and Application, incl. Internet/Web and HCI (SL 3), Lecture Notes in Computer Science (LNCS)*, September 16–19, 2007. Springer-Verlag GmbH: Berlin, Germany. ISBN 3-540-74852-0 and 3-540-74853-9. doi: 10.1007/978-3-540-74853-3.

[8] Hal Abelson, editor. *Workshop on Amorphous Computing*, September 13–14, 1999. MIT Artificial Intelligence Laboratory: Cambridge, MA, USA. URL http://www-swiss.ai.mit.edu/projects/amorphous/workshop-sept-99/.

[9] Thomas Weise, Kurt Geihs, and Philipp Andreas Baer. Genetic Programming for Proactive Aggregation Protocols. In Bartłomiej Beliczyński, Andrzej Dzieliński, Marcin Iwanowski, and Bernardete Ribeiro, editors, *Proceedings of the 8th International Conference on Adaptive and Natural Computing Algorithms, Part I (ICANNGA'07)*, volume 4431/2007 of *Theoretical Computer Science and General Issues (SL 1), Lecture Notes in Computer Science (LNCS)*, pages 167–173. Springer-Verlag GmbH: Berlin, Germany, 2007. doi: 10.1007/978-3-540-71618-1_19.

[10] Navneet Malpani, Jennifer L. Welch, and Nitin Vaidya. Leader Election Algorithms for Mobile Ad Hoc Networks. In *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM'00)*, pages 96–103. ACM Press: New York, NY, USA, 2000. doi: 10.1145/345848.345871. URL http://courses.csail.mit.edu/6.885/spring06/papers/MalpaniWV-dialm00.pdf.

[11] Kostas P. Hatzis, George P. Pentaris, Paul G. Spirakis, Vasilis T. Tampakas, and Richard B. Tan. Fundamental Control Algorithms in Mobile Networks. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 251–260. ACM Press: New York, NY, USA, 1999. doi: 10.1145/305619.305649. URL http://courses.csail.mit.edu/6.885/spring06/papers/Hatzis-etal.pdf.

[12] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)*, pages 350–360. IEEE Computer Society: Piscataway, NJ, USA, 2004. doi: 10.1109/ICNP.2004.1348124. URL ftp://gaia.cs.umass.edu/pub/Vasu03_LdrElec.ps.gz.

[13] Özalp Babaoğlu, Geoffrey Canright, Andreas Deutsch, Gianni A. Di Caro, Frederick Ducatelle, Luca Maria Gambardella, Niloy Ganguly, Márk Jelasity, Roberto Montemanni, Alberto Montresor, and Tore Urnes. Design Patterns from Biology for Distributed Computing. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1

(1):26–66, September 2006. doi: 10.1145/1152934.1152937. URL http://dit.unitn.it/~montreso/pubs/papers/TAAS06.pdf.

[14] Julia K. Parrish and William M. Hamner, editors. *Animal Groups in Three Dimensions: How Species Aggregate*. Cambridge University Press: Cambridge, UK, December 1997. ISBN 0521460247. doi: 10.2277/0521460247.

[15] Gianni A. Di Caro and Marco Dorigo. AntNet: Distributed Stigmergetic Control for Communications Networks. *Journal of Artificial Intelligence Research (JAIR)*, 9:317–365, December 1998. URL http://www.jair.org/media/544/live-544-1748-jair.pdf.

[16] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Inc.: New York, NY, USA, January 1996. ISBN 0-19-509971-0.

[17] David Edward Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1989. ISBN 0-201-15767-5.

[18] Thomas Weise. *Global Optimization Algorithms – Theory and Application*. it-weise.de (self-published): Germany, 2009.

[19] David B. Fogel, editor. *Evolutionary Computation: The Fossil Record*. IEEE Computer Society Press: Los Alamitos, CA, USA and Wiley Interscience: Chichester, West Sussex, UK, May 1998. ISBN 0-7803-3481-7.

[20] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises UK Ltd: London, UK, March 2008. ISBN 1-4092-0073-6. URL http://www.gp-field-guide.org.uk/. With contributions by John R. Koza.

[21] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Bradford Books. MIT Press: Cambridge, MA, USA, December 1992. ISBN 0-262-11170-5. 1992 first edition, 1993 second edition.

[22] Thomas Weise. *Evolving Distributed Algorithms with Genetic Programming*. PhD thesis, University of Kassel, Fachbereich 16: Elektrotechnik/Informatik, Distributed Systems Group: Kassel, Hesse, Germany, May 4, 2009. Won the Dissertation Award of The Association of German Engineers (Verein Deutscher Ingenieure, VDI).

[23] Thomas Weise and Michael Zapf. Evolving Distributed Algorithms with Genetic Programming: Election. In Lihong Xu, Erik D. Goodman, and Yongsheng Ding, editors, *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation (GEC'09)*, pages 577–584. ACM Press: New York, NY, USA, 2009. doi: 10.1145/1543834.1543913.

[24] Thomas Weise, Michael Zapf, and Kurt Geihs. Rule-based Genetic Programming. In *Proceedings of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS'07)*, pages 8–15. IEEE Computer Society: Piscataway, NJ, USA, 2007. doi: 10.1109/BIMNICS.2007.4610073.

[25] Thomas Weise, Michael Zapf, Mohammad Ullah Khan, and Kurt Geihs. Genetic Programming meets Model-Driven Development. In Andreas König, Mario Köppen, Ajith Abraham, Christian Igel, and Nikola Kasabov, editors, *Proceedings of the 7th International Conference on Hybrid Intelligent Systems (HIS'07)*, pages 332–335. IEEE Computer Society: Piscataway, NJ, USA, 2007. doi: 10.1109/HIS.2007.11.

[26] Thomas Weise and Kurt Geihs. Genetic Programming Techniques for Sensor Networks. In Pedro José Marrón, editor, *5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, volume 2006/07, pages 21–25. Universität Stuttgart, Fakultät 5: Informatik, Elektrotechnik und Informationstechnik, Institut für Parallele und Verteilte Systeme (IPVS): Stuttgart, Germany, 2006. URL http://elib.uni-stuttgart.de/opus/volltexte/2006/2838/.

[27] Mark C. Sinclair. Evolutionary Telecommunications: A Summary. In Mark C. Sinclair, David Wolfe Corne, and George D. Smith, editors, *Proceedings of Evolutionary Telecommunications: Past, Present and Future – A Bird-of-a-Feather Workshop at GECCO'99 (GECCO'99 WS)*, pages 209–212, 1999. URL http://uk.geocities.com/markcsinclair/ps/etppf_sin_sum.ps.gz.

[28] Peter Kampstra. Evolutionary Computing in Telecommunications – A Likely EC Success Story. Business mathematics and informatics (bmi), Vrije Universiteit Amsterdam, Faculty of Sciences: Amsterdam, The Netherlands, August 2005. URL http://www.few.vu.nl/stagebureau/werkstuk/werkstukken/werkstuk-kampstra.pdf.

[29] David Wolfe Corne, Martin J. Oates, and George D. Smith, editors. *Telecommunications Optimization: Heuristic and Adaptive Techniques*. John Wiley & Sons Ltd.: New York, NY, USA, September 2000. ISBN 047084163X and 0471988553. doi: 10.1002/047084163X.

[30] Witold Pedrycz and Athanasios V. Vasilakos, editors. *Computational Intelligence in Telecommunications Networks*. CRC Press, Inc.: Boca Raton, FL, USA, September 15, 2000. ISBN 084931075X.

[31] Y. Ahmet Şekercioğlu, Andreas Pitsilides, and Athanasios V. Vasilakos. Computational Intelligence in Management of ATM Networks: A Survey of Current State of Research. In *European Symposium on Intelligent Techniques (ESIT'99)*. European Network for Fuzzy Logic and Uncertainty Modeling in Information Technology (ERUDIT), 1999. URL http://www.cs.ucy.ac.cy/networksgroup/pubs/published/1999/CI-ATM-London99.pdf.

[32] Athanasios V. Vasilakos, Kostas G. Anagnostakis, and Witold Pedrycz. Application of Computational Intelligence Techniques in Active Networks. *Soft Computing – A Fusion of Foundations, Methodologies and Applications*, 5(4):264–271, August 2001. doi: 10.1007/s005000100100.

[33] Yong Xu, Sancho Salcedo-Sanz, and Xīn Yáo. Metaheuristic Approaches to Traffic Grooming in WDM Optical Networks. *International Journal of Computational Intelligence and Applications (IJCIA)*, 5(2):231–249, June 2005. doi: 10.1142/S1469026805001593. URL http://www.cs.bham.ac.uk/~xin/papers/IJCIA_TrafficGrooming2005.pdf.

[34] Pablo Cortés Achedad, Luis Onieva Giménez, Jesús Muñuzuri Sanz, and José Guadix Martín. A Revision of Evolutionary Computation Techniques in Telecommunications and an Application for the Network Global Planning Problem. In Ang Yang, Yin Shan, and Lam Thu Bui, editors, *Success in Evolutionary Computation*, volume 92/2008 of *Studies in Computational Intelligence*, pages 239–262. Springer-Verlag: Berlin/Heidelberg, 2008. doi: 10.1007/978-3-540-76286-7_11.

[35] Thomas Weise and Raymond Chiong. Evolutionary Approaches and Their Applications to Distributed Systems. In Raymond Chiong, editor, *Intelligent Systems for Automated Learning and Adaptation: Emerging Trends and Applications*, chapter 6, pages 114–149. Information Science Reference: Hershey, PA, USA, 2009. doi: 10.4018/978-1-60566-798-0.ch006.

[36] Sancho Salcedo-Sanz, Yong Xu, and Xīn Yáo. Hybrid Meta-Heuristics Algorithms for Task Assignment in Heterogeneous Computing Systems. *Computers & Operations Research*, 33(3):820–835, March 2006. doi: 10.1016/j.cor.2004.08.010. URL http://www.cs.bham.ac.uk/~xin/papers/SalcedoSanzXuYao2006.pdf.

[37] Yong Xu, Sancho Salcedo-Sanz, and Xīn Yáo. Editorial to the Special Issue on Nature Inspired Approaches to Networks and Telecommunications. *International Journal of Computational Intelligence and Applications (IJCIA)*, 5(2):iii–viii, June 2005. doi: 10.1142/S1469026805001532.

[38] Masaharu Munetomo, Yoshiaki Takai, and Yoshiharu Sato. An Adaptive Network Routing Algorithm Employing Path Genetic Operators. In Thomas Bäck, editor, *Proceedings of The Seventh International Conference on Genetic Algorithms (ICGA'97)*, pages 643–649. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1997.

[39] I. M. A. Kirkwood, S. H. Shami, and Mark C. Sinclair. Discovering Simple Fault-Tolerant Routing Rules using Genetic Programming. In George D. Smith, Nigel C. Steele, and Rudolf F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms (ICANNGA'97)*, SpringerComputerScience. Springer Verlag GmbH: Vienna, Austria, 1997. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/kirkam_1997_dsftrr.html.

[40] Sancho Salcedo-Sanz, José Antonio Portilla-Figueras, Emilio G. Ortiz-García, Angel M. Pérez-Bellido, Christopher Thraves, Antonio Fernández-Anta, and Xīn Yáo. Optimal Switch Location in Mobile Communication Networks using Hybrid Genetic Algorithms. *Applied Soft Computing*, 8(4):1486–1497, September 2008. doi: 10.1016/j.asoc.2007.10.021. URL http://nical.ustc.edu.cn/papers/asc2007b.pdf.

[41] Faris N. Abuali, Dale A. Schoenefeld, and Roger L. Wainwright. Terminal Assignment in a Communications Network using Genetic Algorithms. In Dawn Cizmar, editor, *Proceedings of the 22nd Annual ACM Computer Science Conference on Scaling up: Meeting the Challenge of Complexity in Real-World Computing Applications (CSC'94)*, pages 74–81. ACM Press: New York, NY, USA, 1994. doi: 10.1145/197530.197559. URL http://euler.mcs.utulsa.edu/~rogerw/papers/Abuali-Terminal-CSC-94.pdf.

[42] Mark C. Sinclair. Optical Mesh Network Topology Design using Node-Pair Encoding Genetic Programming. In Wolfgang Banzhaf, Jason M. Daida, Ágoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark J. Jakiela, and Robert Elliott Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 2, pages 1192–1197. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1999. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/sinclair_

1999_OMNTDNEGP.html.

[43] Jianyong Sun, Qingfu Zhang, Jin Li, and Xīn Yáo. A Hybrid Estimation of Distribution Algorithm for CDMA Cellular System Design. *International Journal of Computational Intelligence and Applications (IJCIA)*, 7(2):187–200, June 2008. doi: 10.1142/S1469026808002235.

[44] Samuel Pierre and Fabien Houéto. Assigning Cells to Switches in Cellular Mobile Networks using Taboo Search. *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, 32(3):351–356, June 2000. doi: 10.1109/TSMCB.2002.999810.

[45] Dawn Xiaodong Song, Malcom Ian Heywood, and A. Nur Zincir-Heywood. A Linear Genetic Programming Approach to Intrusion Detection. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian Francis Miller, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, Part II (GECCO'03)*, volume 2724/2003 of *Lecture Notes in Computer Science (LNCS)*, pages 2325–2336. Springer-Verlag GmbH: Berlin, Germany, 2003. URL http://users.cs.dal.ca/∼mheywood/X-files/Publications/27242325.pdf.

[46] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A Smart Hill-Climbing Algorithm for Application Server Configuration. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web (WWW'04)*, pages 287–296. ACM Press: New York, NY, USA, 2004. doi: 10.1145/988672.988711. Session: Server performance and scalability.

[47] Jonathan Tate, Benjamin Woolford-Lim, Ian Bate, and Xīn Yáo. Comparing Design of Experiments and Evolutionary Approaches to Multi-Objective Optimisation of Sensornet Protocols. In *10th IEEE Congress on Evolutionary Computation (CEC'09)*, pages 1137–1144. IEEE Computer Society: Piscataway, NJ, USA, 2009. doi: 10.1109/CEC.2009.4983074. URL ftp://ftp.cs.york.ac.uk/papers/rtspapers/R:Tate:2009a.pdf.

[48] Paul Grace. Genetic Programming and Protocol Configuration. Master's thesis, Lancaster University, Computing Department: Lancaster, Lancashire, UK, September 2000. URL http://www.lancs.ac.uk/postgrad/gracep/msc.pdf.

[49] Hirozumi Yamaguchi, Kozo Okano, Teruo Higashino, and Kenichi Taniguchi. Synthesis of Protocol Entities' Specifications from Service Specifications in a Petri Net Model with Registers. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 510–517. IEEE Computer Society: Washington, DC, USA, 1995.

[50] Khaled El-Fakihy, Hirozumi Yamaguchi, and Gregor von Bochmann. A Method and a Genetic Algorithm for Deriving Protocols for Distributed Applications with Minimum Communication Cost. In *Proceedings of the Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, pages 863–868. International Association of Science and Technology for Development (IASTED): Calgary, AB, Canada and ACTA Press: Calgary, AB, Canada, 1999. URL http://www-higashi.ist.osaka-u.ac.jp/∼h-yamagu/resource/pdcs99.pdf.

[51] Sérgio Granato de Araújo, Aloysio de Castro Pinto Pedroza, and Antônio Carneiro de Mesquita Filho. Evolutionary Synthesis of Communication Protocols. In Pascal Lorenz, Petre Dini, and Vladimir Uskov, editors, *Proceedings of the 10th International Conference on Telecommunications (ICT'03)*, volume 2, pages 986–993. IEEE Computer Society: Piscataway, NJ, USA, 2003. doi: 10.1109/ICTEL.2003.1191573. URL http://www.gta.ufrj.br/ftp/gta/TechReports/AMP03a.pdf.

[52] Mohammad Adil Qureshi. Evolving Agents. In John R. Koza, David Edward Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proceedings of the First Annual Conference of Genetic Programming (GP'96)*, Complex Adaptive Systems, Bradford Books, pages 369–374. MIT Press: Cambridge, MA, USA, 1996. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/AQ.gp96.ps.gz.

[53] Hitoshi Iba, Tishihide Nozoe, and Kanji Ueda. Evolving Communicating Agents based on Genetic Programming. In Thomas Bäck, Zbigniew Michalewicz, and Xīn Yáo, editors, *IEEE International Conference on Evolutionary Computation (CEC'97)*, pages 297–302. IEEE Computer Society: Piscataway, NJ, USA, 1997. doi: 10.1109/ICEC.1997.592321.

[54] Kenneth J. Mackin and Eiichiro Tazaki. Emergent Agent Communication in Multi-Agent Systems using Automatically Defined Function Genetic Programming (ADF-GP). In *IEEE International Conference on Systems, Man, and Cybernetics – Human Communication and Cybernetics (SMC'99)*, volume 5, pages 138–142. IEEE Computer Society: Piscataway, NJ, USA, 1999. doi: 10.1109/ICSMC.1999.815536.

[55] Christian F. Tschudin. Fraglets – A Metabolistic Execution Model for Communication Protocols. In *The Second Annual Symposium on Autonomous Intelligent Networks and Systems (AINS'03)*. Center for Autonomous Intelligent Networks and Systems (CAINS), University of California (UCLA): Los Angeles, CA, USA, 2003. URL http://cn.cs.unibas.ch/pub/doc/2003-ains.pdf.

[56] Lidia A. R. Yamamoto and Christian F. Tschudin. Experiments on the Automatic Evolution of Protocols Using Genetic Programming. In Ioannis Stavrakakis and Mikhail I. Smirnov, editors, *Revised and Selected Papers from the Second IFIP Workshop on Autonomic Communication (WAC'05)*, volume 3854/2006 of *Lecture Notes in Computer Science (LNCS)*, pages 13–28. Springer-Verlag GmbH: Berlin, Germany, 2005. doi: 10.1007/11687818_2. URL http://cn.cs.unibas.ch/people/ly/doc/wac2005-lyct.pdf.

[57] Lidia A. R. Yamamoto and Christian F. Tschudin. Experiments on the Automatic Evolution of Protocols Using Genetic Programming. Technical Report CS-2005-002, University of Basel, Computer Science Department, Computer Networks Group: Basel, Switzerland, April 21, 2005. URL http://cn.cs.unibas.ch/people/ly/doc/wac2005tr-lyct.pdf.

[58] Gregory M. Werner and Michael G. Dyer. Evolution of Communication in Artificial Organisms. In Christopher Gale Langdon, Charles E. Taylor, Doyne J. Farmer, and Steen Rasmussen, editors, *Proceedings of the Workshop on Artificial Life (Artificial Life II)*, volume X of *Santa Fe Institue Studies in the Sciences of Complexity*, pages 659–687. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA and Westview Press: Boulder, CO, USA, 1990. URL http://www.isrl.uiuc.edu/∼amag/langev/paper/werner92evolutionOf.html.

[59] Philip K. McKinley, Betty H. C. Cheng, Charles A. Ofria, David B. Knoester, Benjamin Beckmann, and Heather J. Goldsby. Harnessing Digital Evolution. *Computer*, 41(1): 54–63, January 2008. doi: 10.1109/MC.2008.17. URL http://www.cse.msu.edu/∼mckinley/digital-evolution.pdf.

[60] Philip K. McKinley, Betty H. C. Cheng, and Charles A. Ofria. Applying Digital Evolution to the Development of Self-Adaptive ULS Systems (Position Paper). In *Proceedings of the International Workshop on Software Technologies for Ultra-Large-Scale Systems (ULS'07)*, pages 3–3. IEEE Computer Society: Piscataway, NJ, USA, 2007. doi: 10.1109/ULS.2007.1. URL http://www.cs.virginia.edu/∼sullivan/ULS1/ULS07/mckinley.pdf.

[61] Heather J. Goldsby, Betty H. C. Cheng, Philip K. McKinley, David B. Knoester, and Charles A. Ofria. Digital Evolution of Behavioral Models for Autonomic Systems. In *Proceedings of the 5th International Conference on Autonomic Computing (ICAC'08)*, pages 87–96. IEEE Computer Society: Piscataway, NJ, USA, 2008. doi: 10.1109/ICAC.2008.26.

[62] Charles A. Ofria and Claus O. Wilke. Avida: A Software Platform for Research in Computational Evolutionary Biology. *Artificial Life*, 10(2):191–229, 2004. doi: 10.1162/106454604773563612. URL http://www.cs.bham.ac.uk/∼wbl/biblio/gp-html/ofria_2004_AL.html.

[63] David B. Knoester, Philip K. McKinley, Benjamin Beckmann, and Charles A. Ofria. Evolution of Leader Election in Populations of Self-Replicating Digital Organisms. Technical Report MSU-CSE-06-35, Michigan State University, Department of Computer Science: East Lansing, MI, USA, December 2006.

[64] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978. doi: 10.1145/359545.359563. URL http://research.microsoft.com/users/lamport/pubs/time-clocks.pdf. Also: Report CA-7603-2911, Massachusetts Comptuter Association, Wakefield, Massachusetts, USA, March 1976.

[65] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education: Upper Saddle River, NJ, USA and Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 4th rev. edition, June 2005. ISBN 0201180596 and 0321263545.

[66] Friedemann Mattern. *Verteilte Basisalgorithmen*, volume 226 of *Informatik-Fachberichte (IFB)*. Springer-Verlag GmbH: Berlin, Germany, October 1989. ISBN 0-387-51835-5 and 3540518355. Based on his dissertation at Prof. Dr. J. Nehmer's group / Sonderforschungsbereich "VLSI-Entwurf und Parallelität" at the computer science department of the University of Kaiserslautern.

[67] Valmir C. Barbosa. *An Introduction to Distributed Algorithms*. MIT Press: Cambridge, MA, USA, October 1996. ISBN 0262024128.

[68] Thomas Weise, Michael Zapf, and Kurt Geihs. Evolving Proactive Aggregation Protocols. In Michael O'Neill, Leonardo Vanneschi,

Steven Matt Gustafson, Anna Isabel Esparcia-Alcázar, Ivanoe de Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming – Proceedings of the 11th European Conference on Genetic Programming (EuroGP'08)*, volume 4971/2008 of *Theoretical Computer Science and General Issues (SL 1), Lecture Notes in Computer Science (LNCS)*, pages 254–265. Springer-Verlag GmbH: Berlin, Germany, 2008. doi: 10.1007/978-3-540-78671-9_22.

[69] Grenville Armitage, Mark Claypool, and Philip Branch. Network Latency, Jitter and Loss. In *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*, chapter 5, pages 69–82. John Wiley & Sons Ltd.: New York, NY, USA, 2006. doi: 10.1002/047003047X.ch5.

[70] Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio Jesús Nebro Urbaneja. Why is optimization difficult? In Raymond Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193/2009 of *Studies in Computational Intelligence*, chapter 1, pages 1–50. Springer-Verlag: Berlin/Heidelberg, 2009. doi: 10.1007/978-3-642-00267-0_1.

[71] Thomas Weise, Raymond Chiong, and Kē Táng. Evolutionary Optimization: Pitfalls and Booby Traps. *Journal of Computer Science and Technology (JCST)*, 27(5):907–936, September 2012. doi: 10.1007/s11390-012-1274-4. Special Issue on Evolutionary Computation, edited by Xin Yao and Pietro S. Oliveto.

[72] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technische Universität Berlin: Berlin, Germany, 1971.

[73] Patrick C. Phillips. The Language of Gene Interaction. *Genetics*, 149(3):1167–1171, July 1998. URL http://www.genetics.org/cgi/reprint/149/3/1167.pdf.

[74] Christian W. G. Lasarczyk and Wolfgang Banzhaf. An Algorithmic Chemistry for Genetic Programming. In Maarten Keijzer, Andrea G. B. Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming (EuroGP'05)*, volume 3447/2005 of *Theoretical Computer Science and General Issues (SL 1), Lecture Notes in Computer Science (LNCS)*, pages 1–12. Springer-Verlag GmbH: Berlin, Germany, 2005. doi: 10.1007/978-3-540-31989-4_1. URL http://ls11-www.cs.uni-dortmund.de/downloads/papers/LaBa05.pdf.

[75] Wolfgang Banzhaf and Christian W. G. Lasarczyk. Genetic Programming of an Algorithmic Chemistry. In Una-May O'Reilly, Gwoing Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II, Proceedings of the Second Workshop on Genetic Programming (GPTP'04)*, volume 8 of *Genetic Programming Series*, pages 175–190. Kluwer Publishers: Boston, MA, USA, 2004. doi: 10.1007/0-387-23254-0_11. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/banzhaf_2004_GPTP.html.

[76] Christian W. G. Lasarczyk and Wolfgang Banzhaf. Total Synthesis of Algorithmic Chemistries. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantú-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llorà, Spiros Mancoridis, Martin Pelikan, Günther R. Raidl, Terence Soule, Jean-Paul Watson, and Eckart Zitzler, editors, *Proceedings of Genetic and Evolutionary Computation Conference (GECCO'05)*, volume 2, pages 1635–1640. ACM Press: New York, NY, USA, 2005. doi: 10.1145/1068009.1068285. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1635.pdf.

[77] Nicholas Freitag McPhee and Riccardo Poli. Memory with Memory: Soft Assignment in Genetic Programming. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev P. Kumar, Fernando G. Lobo, Julian Francis Miller, Jason H. Moore, Frank Neumann, Martin Pelikan, Jordan B. Pollack, Kumara Sastry, Kenneth Owen Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'08)*, pages 1235–1242. ACM Press: New York, NY, USA, 2008. doi: 10.1145/1389095.1389336. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/McPhee_2008_gecco.html.

[78] Frank D. Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. Homologous Crossover in Genetic Programming. In Wolfgang Banzhaf, Jason M. Daida, Ágoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark J. Jakiela, and Robert Elliott Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 2, pages 1021–1026. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1999. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-463.pdf.

[79] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. URL https://www.fdi.ucm.es/profesor/fraguas/CC/church-An%20Unsolvable%20Problem%

[80] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: 10.1112/plms/s2-42.1.230. URL http://www.abelard.org/turpap2/tp2-ie.asp.

[81] Gerald J. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, September 3, 2003. ISBN 0321228626.

[82] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer-Verlag GmbH: Berlin, Germany, January 2008. ISBN 1-84628-769-3 and 1-84628-770-7. doi: 10.1007/978-1-84628-770-1.

[83] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall International Series in Computer Science. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA and Pearson Education: Upper Saddle River, NJ, USA, 2nd edition, March 6, 2006. ISBN 032131283X.

[84] Marie-Pierre Gleizes, Valérie Camps, and Pierre Glize. A Theory of Emergent Computation based on Cooperative Self-Organization for Adaptive Artificial Systems. In Lorenzo Ferrer-Figueras, editor, *Fourth European Congress of Systems Science (CES-4)*. Spanish Society of General Systems (Sociedad Española de Sistemas Generales): València, Spain, 1999. URL ftp://ftp.irit.fr/pub/IRIT/SMAC/DOCUMENTS/PUBLIS/ECS99.pdf.

[85] Thomas G. Dietterich. Overfitting and Undercomputing in Machine Learning. *ACM Computing Surveys (CSUR)*, 27(3):326–327, 1995. doi: 10.1145/212094.212114.

[86] Stefan Bleuler, Martin Brack, Lothar Thiele, and Eckart Zitzler. Multiobjective Genetic Programming: Reducing Bloat Using SPEA2. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'01)*, pages 536–543. IEEE Computer Society: Piscataway, NJ, USA, 2001. URL ftp://ftp.tik.ee.ethz.ch/pub/people/zitzler/BBTZ2001b.ps.gz.

[87] John Henry Holland and Judith S. Reitman. Cognitive Systems based on Adaptive Algorithms. In Donald Arthur Waterman and Frederick Hayes-Roth, editors, *Selected Papers from the Workshop on Pattern Directed Inference Systems*, pages 313–329. Academic Press: New York, NY, USA, 1977.

[88] William M. Spears and Kenneth Alan De Jong. Using Genetic Algorithms for Supervised Concept Learning. In *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence (TAI'90)*, pages 335–341. IEEE Computer Society Press: Los Alamitos, CA, USA, 1990. doi: 10.1109/TAI.1990.130359. URL http://www.cs.uwyo.edu/~wspears/papers/ieee90.pdf.

[89] Susumu Ohno. *Evolution by Gene Duplication*. Springer New York: New York, NY, USA and Allen & Unwin: Crows Nest, NSW, Australia, 1970. ISBN 0045750157.

[90] Jianzhi Zhang. Evolution by Gene Duplication: An Update. *Trends in Ecology and Evolution (TREE)*, 18(6), June 1, 2003. doi: 10.1016/S0169-5347(03)00033-8. URL http://www.umich.edu/~zhanglab/publications/2003/Zhang_2003_TIG_18_292.pdf.

[91] Ting Hu and Wolfgang Banzhaf. Evolvability and Acceleration in Evolutionary Computation. Technical Report MUN-CS-2008-04, Memorial University, Department of Computer Science: St. John's, Canada, October 2008. URL http://www.cs.mun.ca/~tingh/papers/TR08.pdf.

[92] Astro Teller. Turing Completeness in the Language of Genetic Programming with Indexed Memory. In Zbigniew Michalewicz, James David Schaffer, Hans-Paul Schwefel, David B. Fogel, and Hiroaki Kitano, editors, *Proceedings of the First IEEE Conference on Evolutionary Computation (CEC'94), 1994 IEEE World Congress on Computation Intelligence (WCCI'94)*, pages 136–141. IEEE Computer Society: Piscataway, NJ, USA, IEEE Computer Society: Piscataway, NJ, USA, 1994. doi: 10.1109/ICEC.1994.350027. URL http://www.astroteller.net/pdf/Turing.pdf.

[93] John R. Woodward. Evolving Turing Complete Representations. In Ruhul A. Sarker, Robert G. Reynolds, Hussein A. Abbass, Kay Chen Tan, Robert Ian McKay, Daryl Leslie Essam, and Tom Gedeon, editors, *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'03)*, volume 2, pages 830–837. IEEE Computer Society: Piscataway, NJ, USA, 2003. doi: 10.1109/CEC.2003.1299753. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/Woodward_2003_

Etcr.html.

[94] William Stallings. *Operating Systems: Internals and Design Principles*. Gradience Online Accelerated Learning Series (GOAL). Pearson Education: Upper Saddle River, NJ, USA and Prentice Hall International Inc.: Upper Saddle River, NJ, USA, 2004. ISBN 0029464919, 0130319996, 0131479547, 0131809776, 0136006329, 0138874077, and 0139179984.

[95] I1997IASDMV1BA. *Intel Architecture Software Developer's Manual – Volume 1: Basic Architecture*, 1997. URL http://developer.intel.com/design/pentium/manuals/24319001.pdf.

[96] Wolfgang Banzhaf. Genetic Programming for Pedestrians. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA'93)*, pages 17–21. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1993. URL ftp://ftp.cs.cuhk.hk/pub/EC/GP/papers/pedes93.ps.gz. Also: MRL Technical Report 93-03.

[97] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient Evolution of Machine Code for CISC Architectures using Instruction Blocks and Homologous Crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter John Angeline, editors, *Advances in Genetic Programming*, volume 3 of *Complex Adaptive Systems, Bradford Books*, chapter 12, pages 275–299. MIT Press: Cambridge, MA, USA, 1996. URL http://www.cs.bham.ac.uk/~wbl/aigp3/ch12.pdf.

[98] Markus F. Brameier and Wolfgang Banzhaf. A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. *IEEE Transactions on Evolutionary Computation (IEEE-EC)*, 5(1):17–26, 2001. doi: 10.1109/4235.910462. URL http://web.cs.mun.ca/~banzhaf/papers/ieee_taec.pdf.

[99] Lidia A. R. Yamamoto and Christian F. Tschudin. Genetic Evolution of Protocol Implementations and Configurations. In Jean-Philippe Martin-Flatin, Joe Sventek, and Kurt Geihs, editors, *IFIP/IEEE International Workshop on Self-Managed Systems and Services (SelfMan'05)*, 2005. URL http://cn.cs.unibas.ch/pub/doc/2005-selfman.pdf.

[100] Christian F. Tschudin and Lidia A. R. Yamamoto. Fraglets Instruction Set, September 24, 2007. URL http://www.fraglets.net/frag-instrset-20070924.txt.

[101] Lidia A. R. Yamamoto, Daniel Schreckling, and Thomas Meyer. Self-Replicating and Self-Modifying Programs in Fraglets. In *Proceedings of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS'07)*. IEEE Computer Society: Piscataway, NJ, USA, 2007. doi: 10.1109/BIMNICS.2007.4610104. URL http://cn.cs.unibas.ch/people/ly/doc/bionetics2007-ysm.pdf.

[102] Paolo Dini. Section 1: Science, New Paradigms. Subsection 1: A Scientific Foundation for Digital Ecosystems. In Francesco Nachira, Paolo Dini, Andrea Nicolai, Marion Le Louarn, and Lorena Rivera Lèon, editors, *Digital Business Ecosystems*. Office for Official Publications of the European Communities: Luxembourg, 2007. URL http://www.digital-ecosystems.org/book/Section1.pdf. See also: Keynote Talk "Structure and Outlook of Digital Ecosystems Research", IEEE Digital Ecosystems Technologies Conference, DEST07, Cairns, Australia, February 20–23, 2007, and.

[103] Thomas Weise, Stefan Niemczyk, Hendrik Skubch, Roland Reichle, and Kurt Geihs. A Tunable Model for Multi-Objective, Epistatic, Rugged, and Neutral Fitness Landscapes. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev P. Kumar, Fernando G. Lobo, Julian Francis Miller, Jason H. Moore, Frank Neumann, Martin Pelikan, Jordan B. Pollack, Kumara Sastry, Kenneth Owen Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'08)*, pages 795–802. ACM Press: New York, NY, USA, 2008. doi: 10.1145/1389095.1389252.

[104] Sidney Siegel and N. John Castellan Jr. *Nonparametric Statistics for The Behavioral Sciences*. Humanities/Social Sciences/Languages. McGraw-Hill: New York, NY, USA, 1956. ISBN 0-07-057357-3 and 070573434X.

[105] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In Ralph H. Sprague, Jr., editor, *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS'00)*, volume 2, page 10. IEEE Computer Society: Washington, DC, USA, 2000. URL http://pdos.csail.mit.edu/decouto/papers/heinzelman00.pdf.

[106] Gérard Le Lann. Distributed Systems – Towards a Formal Approach. In Bruce Gilchrist, H. J. Kugler, and Simon H. Lavington, editors, *Proceedings of the International Federation for Information Processing World Computer Congress on Information Processing (IFIP'77)*, volume 7 of *IFIP Congress Series*, pages 155–160. North-Holland Scientific Publishers Ltd.: Amsterdam, The Netherlands, 1977. URL http://www-roc.inria.fr/novaltis/publications/IFIP%20Congress%201977.pdf.

[107] Ernest J. H. Chang and Rosemary Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM (CACM)*, 22(5):281–283, 1979. doi: 10.1145/359104.359108.

[108] Patrick Alfred Pierce Moran. Random Processes in Genetics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 54(01): 60–71, January 1958. doi: 10.1017/S0305004100033193.

[109] Edsger Wybe Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages (Electronic Computers): Papers presented at a NATO Summer School*, pages 43–112. Academic Press: London, New York, 1968. URL http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF. EWD 123. Discusses Theodorus Jozef Dekker's algorithm.

[110] Glenn Ricart and Ashok K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM (CACM)*, 24(1):9–17, January 1981. doi: 10.1145/358527.358537.

[111] Mamoru Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems (ACM TOCS)*, 3(2):145–159, May 1985. doi: 10.1145/214438.214445.

[112] Andrew Stuart Tanenbaum. *Computer Networks*. Prentice Hall PTR: Englewood Cliffs, Upper Saddle River, NJ, USA, 1981. ISBN 0130384887, 0130661023, 013162959X, and 0131651838.

[113] Norman R. Paterson. *Genetic Programming with Context-Sensitive Grammars*. PhD thesis, University of St Andrews, School of Computer Science: St Andrews, Scotland, UK, September 2002. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/paterson_thesis.html.

[114] Míngxù Wàn, Thomas Weise, and Kē Táng. Novel Loop Structures and the Evolution of Mathematical Algorithms. In Sara Silva, James A. Foster, Miguel Nicolau, Penousal Machado, and Mario Giacobini, editors, *Proceedings of the 14th European Conference on Genetic Programming (EuroGP'11)*, volume 6621/2011 of *Theoretical Computer Science and General Issues (SL 1), Lecture Notes in Computer Science (LNCS)*, pages 49–60. Springer-Verlag GmbH: Berlin, Germany, 2011. doi: 10.1007/978-3-642-20407-4_5.

[115] Hisao Ishibuchi, Noritaka Tsukamoto, and Yusuke Nojima. Evolutionary Many-Objective Optimization: A Short Review. In Zbigniew Michalewicz and Robert G. Reynolds, editors, *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'08), Computational Intelligence: Research Frontiers – IEEE World Congress on Computational Intelligence – Plenary/Invited Lectures (WCCI)*, volume 5050/2008 of *Theoretical Computer Science and General Issues (SL 1), Lecture Notes in Computer Science (LNCS)*, pages 2424–2431. IEEE Computer Society: Piscataway, NJ, USA, 2008. doi: 10.1109/CEC.2008.4631121. URL http://www.ie.osakafu-u.ac.jp/~hisaoi/ci_lab_e/research/pdf_file/multiobjective/CEC2008_Many_Objective_Final.pdf.

[116] Euclid of Alexandria, editor. *Stoicheia, Elements*. self-published: Alexandria, Greece, 300 BC. URL http://aleph0.clarku.edu/~djoyce/java/elements/elements.html. A series consisting of 13 books, to which two books were added later on.

[117] Hugues Juillé and Jordan B. Pollack. Massively Parallel Genetic Programming. In Peter John Angeline and Kenneth E. Kinnear, Jr, editors, *Advances in Genetic Programming II*, Complex Adaptive Systems, Bradford Books, pages 339–358. MIT Press: Cambridge, MA, USA, 1996.

[118] John R. Koza, Forrest H. Bennett III, Jeffrey L. Hutchings, Stephen L. Bade, Martin A. Keane, and David Andre. Evolving Sorting Networks using Genetic Programming and the Rapidly Reconfigurable Xilinx 6216 Field-Programmable Gate Array. In Monique P. Fargues and Ralph D. Hippenstiel, editors, *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers*, volume 1, pages 404–410. IEEE Computer Society: Piscataway, NJ, USA, 1997. doi: 10.1109/ACSSC.1997.680275. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/koza_1997_ASILIMOAR.html.

[119] William B. Langdon and Wolfgang Banzhaf. A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In Michael O'Neill, Leonardo Vanneschi, Steven Matt Gustafson, Anna Isabel Esparcia-Alcázar, Ivanoe de Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming – Proceedings of the 11th European Conference on Genetic Programming (EuroGP'08)*, volume 4971/2008

of *Theoretical Computer Science and General Issues (SL 1), Lecture Notes in Computer Science (LNCS)*, pages 73–85. Springer-Verlag GmbH: Berlin, Germany, 2008. doi: 10.1007/978-3-540-78671-9_7. URL http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2008_eurogp.html.

[120] William B. Langdon. A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. Computer Science Technical Report CSM-470, University of Essex, Departments of Mathematical and Biological Sciences: Wivenhoe Park, Colchester, Essex, UK, July 3, 2007. URL http://cswww.essex.ac.uk/technical-reports/2007/csm_470.pdf.

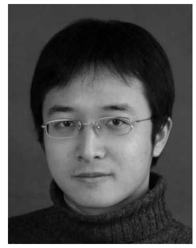
[121] David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation (IEEE-EC)*, 1(1):67–82, April 1997. doi: 10.1109/4235.585893.

[122] Thomas Weise, Michael Zapf, Mohammad Ullah Khan, and Kurt Geihs. Combining Genetic Programming and Model-Driven Development. *International Journal of Computational Intelligence and Applications (IJCIA)*, 8(1):37–52, March 2009. doi: 10.1142/S1469026809002436.

Thomas Weise (M'10) received the Diplom Informatiker (equivalent to M.Sc.) degree from the Department of Computer Science, Chemnitz University of Technology, Chemnitz, Germany, in 2005, and the Ph.D. degree at the Distributed Systems Group of the Fachbereich Elektrotechnik and Informatik, University of Kassel, Kassel, Germany in 2009.

Since 2009, has been researcher at the Nature Inspired Computation and Applications Laboratory, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China. His major research interests include Evolutionary Computation, Genetic Programming, and real-world applications of optimization algorithms. His experience ranges from applying GP to distributed systems and multi-agent systems, efficient web service composition for Service Oriented Architectures, to solving large-scale real-world vehicle routing problems for multimodal logistics and transportation.

Besides being the author/co-author of over 40 refereed publications, Dr. Weise also authors the electronic book *Global Optimization Algorithms – Theory and Application* which is freely available at his website http://www.it-weise.de/.



Ke Tang (S'05–M'07) received the B.Eng. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2002 and the Ph.D. degree from the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, in 2007.

Since 2007, he has been an Associate Professor with the Nature Inspired Computation and Applications Laboratory, School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. He is also an Honorary Senior Research Fellow in the School of Computer Science, University of Birmingham, UK. He has authored/co-authored more than 40 refereed publications. His major research interests include machine learning, pattern analysis, evolutionary computation, data mining, metaheuristic algorithms, and real-world applications.

Dr. Tang is an Associate Editor of IEEE Computational Intelligence Magazine, editorial board member of three international journals, and the Chair of IEEE Task Force on Large Scale Global Optimization.


```
@article{WT2012EVDAWGP,
 author = {Thomas Weise and Ke Tang},
 title = {Evolving Distributed Algorithms with Genetic Programming},
 publisher = {IEEE Computer Society: {Washington, DC, USA}},
 journal = {IEEE Transactions on Evolutionary Computation (IEEE-EC)},
 number = {2},
 volume = {16},
 pages = {242--265},
 year = {2012},
 month = apr,
 doi = {10.1109/TEVC.2011.2112666},
},
```