Genetic Evolution of Protocol Implementations and Configurations

Lidia Yamamoto and Christian Tschudin
Computer Science Department, University of Basel
Bernoullistrasse 16, CH-4056 Basel, Switzerland
Email: Lidia.Yamamoto@unibas.ch, Christian.Tschudin@unibas.ch

Abstract— **One of the biggest challenges in obtaining truly self-managed networks is to automate the process of software evolution, and in particular, the evolution of protocol implementations and configurations. In this paper we explore an approach to network evolution that works *inside the network software* to manage and which operates directly at the code level. We investigate related code steering techniques in two directions: One is the fully *automatic selection of protocol service elements* where, depending on device characteristics and current operation environment, each communication entity has to select among a potentially wide variety of protocol implementations providing similar services. The other direction relates to the *automatic synthesis of new protocol elements* which are the result of optimizing existing implementations for a specific context. We use genetic programming as a tool to generate new configurations and new code automatically. In this paper we present a framework for injecting such code into a running environment in a non-disruptive way and report on first exploratory results on resilient protocol evolution.**

I. Introduction

Many scenarios for future networks include cases where explicit network management is not desired (home network with some hundred networked gadgets), is difficult (arbitrary mesh networks with wireless and wired links) or even is impossible (intermittently connected sensors spread over a wide area). Here, management should be handled by the network itself in an autonomic way, without human intervention. The network's functional building blocks themselves must become self-managed and must be able to organize into a coherent network that exhibits purposeful behavior. While an "autonomic network element" might still offer an external management interface, in a first place it has to adapt to changing requirements and environmental conditions and yet be resilient. Malicious or erroneous entities could then try to disturb its building blocks in any possible way, but ideally, these blocks would detect and defeat attacks and would recover and heal themselves to continue providing the required services. In case of failures, alternative service blocks would replace the non-functioning ones in a reactive and non-supervised way.

Some of the environmental changes like new user goals, new software or hardware constraints or patterns in dynamic resource availability, can be predicted by the network software engineers at design time and can be built into adaptive algorithms able to steer the network to the desired operating point. However, not all kinds of changes can be foreseen. There will always be cases that require modifications in the adaptive algorithms themselves: These new algorithms must be engineered, then programmed, and redeployed in the network. Today this process is slow and requires the effort of many people (network managers, engineers, programmers), which is out of scope for autonomic networks. Ultimately, protocols and algorithms for these networks should evolve during their own execution, with minimum service disruption.

In this paper we describe our framework for protocol evolution based on genetic programming. We concentrate on two research directions: the first one is to automatically select combinations of protocol modules adapted to given network conditions; the second is the automatic synthesis of new protocols optimized for a specific context. The contribution of this paper is to show the feasibility of automatic network software selection based on service agnostic target functions. This result is based on the introduction of competition at the level of functional blocks and the use of genetic algorithms to steer the selection process. We summarize our initial experimental results, using simple case studies, still in a simulated, off-line environment, but with considerations and parameters intended to progressively detach the framework from the off-line simulation out into the real world.

This paper is structured as follows: Section II summarizes the state of the art in program and protocol evolution techniques. Section III states our position and describes our framework for protocol evolution. Section IV summarizes the experimental results. Section V concludes the paper with our outlook for this new area.

II. State of the Art

Automatic programming or *program synthesis* refers to any method for automated generation of a computer program that is able to solve a given problem expressed in a high-level form. Examples include variations of meta-programming, deductive program synthesis [1], and evolutionary methods such as genetic programming.

Genetic Programming (GP) [2] evolves computer programs automatically from random initial code, using genetic operations such as crossover and mutation, and evolution by natural selection ("survival of the fittest") to select the solutions that best satisfy specified criteria. GP is typically employed when the solution to a problem is not known or very difficult to program by hand.

GP applications generally rely on off-line code generation. Once in operation, the generated code does not continue to modify itself to adapt to changing situations. One of the reasons for that is that genetic operations cannot guarantee error-free code for each generated program. Another drawback of off-line generation is that the code evolved in a simulated environment will not necessarily work as expected in the real world.

Genetic programming has been applied to evolve new programs at run-time in domains such as evolvable hardware and robotics. However, to the best of our knowledge, the on-line evolution of networking protocol code has not been tried yet. In [3] genetic algorithms are applied in a decentralized way to evolve agents that provide network services. Their results show that evolution can improve agent performance. However, in their scheme, the code itself does not change. They focus on the evolution of parameters that trigger certain predefined behaviors.

Protocol synthesis methods [4] aim to generate a valid protocol specification that satisfies a supplied service specification. Since these methods must guarantee error-free code, they are still not feasible for on-line evolution.

An iterative deepening search method is proposed in [5] to find protocol specifications that satisfy a given set of security properties. In [6] genetic search is used to synthesize from scratch protocol implementations expressed as communicating finite state machines. This research is extended in [7] and shows that relatively complex protocols can be synthesized in this way, and in certain cases these protocols can even outperform a reference protocol designed and validated by human beings. However in most cases the fitness of synthesized protocols is significantly lower than the reference protocol.

III. Evolving Communication Protocols

The main premise underlying our work is that software in an autonomic network must be *self-modifying*. Otherwise, humans have to cater for the software's adaption every time a case is encountered which was not anticipated at design time.

We envisage different levels at which self-modification of software takes place and different time scales at which such modifications can happen. A first step, aimed at a shorter time scale, is the configuration of function blocks, where the challenge consists in selecting the right combinations from ready-made modules. This is the focus of this paper. Today, this is mostly controlled by the standardization process and interoperability tests. Although several systems able to dynamically reconfigure software have been proposed, for instance [8], most of these systems still rely on humans to program exactly what kind of reconfiguration should be performed under which circumstances. In the future we imagine that a network "settles" by itself on different protocol sets without having humans to intervene. At a longer time scale, self-modification can be extended down to the level of single instructions where the autonomic network has the power to create new implementation variants, instead of just configuring coarse grained functional blocks.

A. Resilience and Competition

For such a self-managing process to work we need a modus operandi that permits adaption (medium time scale) as well as evolution (long term). Adaption relates to the configuration of existing functionality while evolution refers to the modification of old and generation of new functions. We believe that two attributes of such a system are key for its viability: resilience and competition. *Inherent resilience* is needed, otherwise there is a risk that (malicious or erroneous) function blocks can be inserted that disrupt the network's operation. The second attribute is *competition*: the autonomic network operates in a constant optimization mode where best suited code variants are selected.

B. Software Hardening and Genetic Programming

We have started to explore the feasibility of self-modifying communication software by demonstrating protocol resilience, where protocol implementations can survive the removal of an arbitrary code line [9]. In the current paper we explore genetic programming for modifying, recombining and erasing protocol modules. Other machine learning methods or heuristics could also be envisaged [5]. However, plain genetic programming lends itself for our project because it is agnostic to the functions adapted, and naturally extends to the finer grained code evolution that enables long-term synthesis and evolution.

The execution environment for the protocol software should be amenable to genetic programming. Section III-E explains our choice of the Fraglet model [10] for this purpose.

C. A Framework for Automated Code Steering

Ideally, a software environment for an autonomic network would feature continuous adaption and evolution: Alternative code variants would co-exist in parallel with the currently best selection of protocol implementations. In terms of code steering, there would be a mechanism in place for on-line evaluation and selection of the alternatives. This on-line evolution would be a continuously ongoing process that is decentralized and asynchronous, working on each node and many levels inside the graph of functional modules.

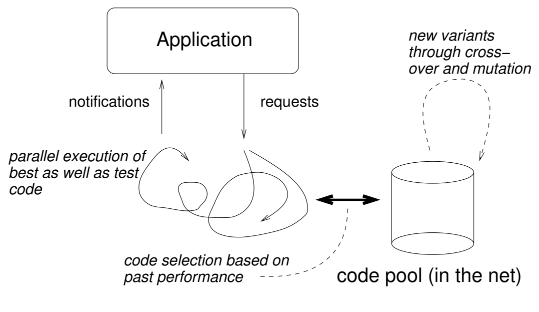
Application
notifications
requests
new variants through cross–over and mutation
parallel execution of best as well as test code
code selection based on past performance
code pool (in the net)

Fig. 1. Conceptual framework for automatic protocol evolution

Figure 1 shows a conceptual model of how resilience and competition work together to enable the automatic evolution of protocol implementations and configurations. Applications (or any client protocol) delegate service provisioning to a resilient protocol implementation, and from time to time or in parallel give a chance to test candidates. Based on their performance, new service implementation variants can increase their chance to be selected a next time. Service variations do include different ways of combining sub-services. Because the evaluation and selection mechanism takes into account the overall performance of a service implementation, it will give preference to the service with the most optimal internal composition and configuration of sub-services.

Our current implementation of the model of Figure 1 is still limited to off-line evolution, i.e. to the case of synchronous evaluation and selection, so there are no concurrent services yet. However we plan to progressively detach it from the off-line sphere in favor of the long-term goal of on-line evolution.

D. Genetic Programming Set-up for Protocol Evolution

Genetic Programming is used to evolve computer programs, which in our case correspond to communication protocols or protocol structures. Each program is an individual in a GP population. Each individual or program is regarded as genetic material in an evolutionary plan. In our case, an individual could correspond to a configuration of protocols to be evaluated (e.g. forming a protocol stack or other arbitrary protocol structure). In the simplest case, the configuration corresponds to a single protocol, which is the starting case we consider for this paper. The metaphor for the code is the genotype, which encodes the functioning of an individual and is manipulated through genetic operations. In our case, the protocol genotype is divided into modules, that make up the "genes" of the individual.

The population evolves from one generation to the next by the use of well-known genetic operators such as crossover, mutation, and cloning. Cloning simply produces an identical copy of an individual. Crossover combines the genetic material of two individuals by swapping a segment of the first one with a segment of the second one. Mutation randomly modifies a small portion of an individual. The way in which genetic operators are combined to produce new individuals determines the speed of the search and extent of the search space explored as the GP run progresses. The general rule is to submit the best individuals of the population to crossover, and apply mutation with a small probability.

The crossover operator in our set-up is a simplified implementation of the genetic concept of homologous recombination. Homologous recombination states that the exchange of genetic material can only occur between functionally compatible DNA segments, and is only triggered when the two DNA strands are completely aligned. This form of recombination preserves gene functionality, promotes genetic stability, and increases the probability of producing viable offspring. In our set-up, crossover may only occur at gene (module) boundaries and between functionally equivalent modules. Modules are identified by a name, and crossover operations are only allowed to exchange modules with identical names.

The fitness measure is the performance of the protocol as perceived by the applications. They reward correct behavior and punish incorrect one when detected. For instance, the score of an individual is incremented when it performs the correct operation (e.g. successfully delivering a packet), and it is decremented when an error is detected (e.g. an acknowledgment is issued for a data item that has never been actually received). Resource consumption, in terms of memory occupied by the genotype, is proportionally penalized.

The GP algorithm used follows a fairly standard tournament selection mechanism, in which a number of individuals is selected for a tournament. The winning individuals and/or their descendants replace the losing ones.

E. Fraglets

The Fraglet paradigm [10] has been proposed as part of our search for feasible ways to achieve automated synthesis of protocol implementations. It is an instance of Gamma systems [11], a chemical model where "molecules" interact with each other or undergo some internal transformation. A fraglet is a string of symbols $[s_1 : s_2 : \ldots : s_n]$ representing data and/or protocol logic. It is a fragment of a distributed computation, that may be carried in packets or stored inside a network node. The fraglet processing engine continuously executes tag matching operations on the fraglets in the store, in order to determine the actions that should be applied to them. The fraglet instruction set contains two types of actions: transformation of a single fraglet, and "chemical reaction" between two fraglets. The instruction set is described in [10], [9], along with examples of processing and protocol functions.

The fraglets model has many relevant properties that must be highlighted in connection with automated protocol synthesis and evolution. First of all, any string of symbols is a valid fraglet, therefore fraglets can be split at arbitrary places and merged with other fraglets to produce different code. A second property is the ability to express code and data in a uniform way. Code is manipulated just like any other form of data, and it is easy to express rules that generate and delete code from the running pool. A third aspect is the ability to express code mobility in a natural way: any fraglet can be regarded as either a set of packet header tags that can be processed by a header processing engine, or as a program fragment that is executed at a given node. This facilitates the dynamic deployment of new code logic.

A fourth property of the fraglet environment stems from its roots in Gamma systems: it enables programs to be expressed in a highly parallel way that is very close to their specification, without artificial sequentiality constraints. This is relevant for automated program synthesis and evolution, in two ways: first, this parallelism can be used to produce resilient programs as shown in [9], which tolerate the loss of parts of their code stream, due to fallback alternatives running in parallel. This can be used to diminish the impact of malfunctioning code. Secondly, the fact that programs are relatively compact and close to their specification could open up potential avenues for deterministic synthesis techniques based on specification.

IV. Summary of experiments

We have performed a few experiments using the fraglet environment to verify whether software configurations can adapt to their environment, by the mere application of generic and service agnostic GP methods. A simple case is considered where a reliable delivery service must be provided over different channel characteristics. The task is to transmit all packets from the client application, with acknowledgment of correct delivery. Two types of underlying transmission channels are considered:

- *Perfectly reliable channel*: In this case, the protocol does not need to retransmit packets. A simple implementation of this is the confirmed delivery protocol (CDP) presented in [10].
- *Unreliable channel*: In this case, the protocol must retransmit lost packets. A reliable delivery protocol (RDP) has been implemented for this purpose. It takes an input payload from the application, sends it to the destination, stores a copy locally, and sets a waiting timer. When the timer expires, and the corresponding local copy of the information is still stored, the packet is retransmitted. When an acknowledgment is received, the local copy is destroyed; this cancels any pending retransmissions scheduled for the item.

Both CDP and RDP are very simple protocols able to handle only one packet at a time. But they suffice to illustrate the concepts of evolutionary protocol module selection, code adaptation and re-adaptation.

The quantitative results of the experiments are reported in detail in [12]. We summarize their qualitative aspects below.

A. Adaptation

The goal of this experiment is to verify whether a mixed population of protocols is able to adapt to a given environment. Two identical instances of a mixed population consisting of CDP and RDP variants are created. One is inserted into a lossy channel environment and the other into a non-lossy one.

Figure 2 shows the adaptation of the initially mixed population to the lossy environment (the non-lossy one is omitted due to space constraints). The upper part shows the absolute fitness scores, and the lower part shows the percentage of high and low-score individuals. A high-score individual is an individual that has achieved a score equivalent to at least 80% of the best score from its generation. A low-score one scores less than 40% of the best of its generation. In this experiment, the best individual is also the optimum (hand-designed), and the GP selection process succeeds to keep it in the population through the successive generations.

The population on the non-lossy channel (not shown) starts with a low average score, but after a few generations most of the individuals have a score close to the best. After four or five generations the retransmission code is eliminated, and at the end of the experiment the surviving individuals are all instances of CDP.

In the lossy case (Fig. 2), after roughly 15 generations, more than 80% of the population is made up of high-score individuals. The retransmission code spreads very quickly through the entire population on the lossy channel: all the individuals contain it after the first couple of generations, and at the end of the experiment all individuals are variations of RDP.

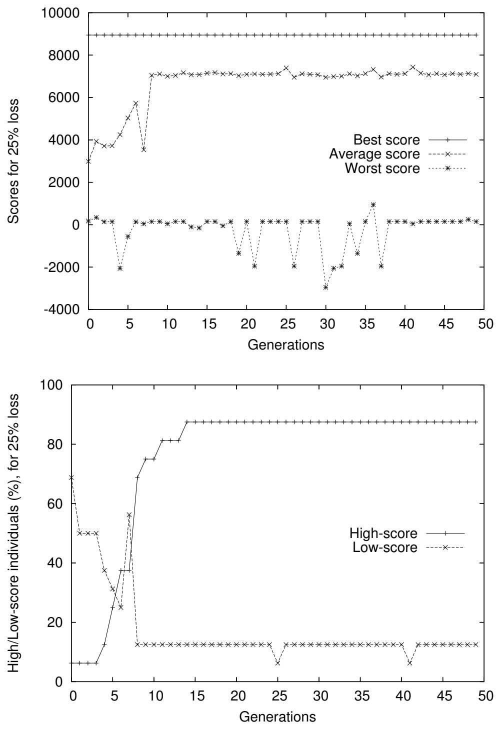

Fig. 2. Absolute scores and percentage of high/low scores for the lossy environment

In both lossy and non-lossy cases, mutations are mostly responsible for the recurrence of low-performance individuals, even after these have been eliminated by the fitness selection process. The purpose of mutations is to introduce genetic variability. However, it is well known that most mutations are harmful. In our case, mutations are kept in the system in order to test its capacity to produce new code, and its resilience to potentially disrupting code. The production of new useful code has not been verified in such short runs. On the other hand, the fact that the system can still adapt in spite of harmful mutations is an indication that resilience at the population level is possible even with a relatively high rate of mutation. However this system is obviously not perfect. There are still clients affected by low-performance individuals. Resilience is not achieved at the individual level. Furthermore, as it adapts, the population loses genetic variability, which hinders its capacity to readapt. This problem is also present in nature, when genetic variability is lost in small populations adapted to a fairly stable environment.

B. Re-adaptation

In this experiment we investigate the capacity of a population to readapt to an environment different from the one where it has originally evolved.

We inject a population evolved in a lossy environment into a non lossy one and vice-versa. The results (graphs not shown for conciseness) clearly show that the population cannot readapt. The lost retransmission modules cannot be recreated in such a short time by genetic operators only.

Crossover only recombines existing modules, and mutations of individual symbols is simply a too slow and randomized process. The search space for the solution is far too vast, spanning all the combinations of about 30 symbols at around 200 positions. Nevertheless, if we inject a single optimally adapted individual in the population, it instantly redeploys and the entire population readapts after a few generations.

C. Discussion

We can extract several lessons from these early experiments. We first discuss the aspects related to genetic operators and other GP parameters. We then discuss future issues of resilience and on-line evolution.

We have modeled homologous recombination which is generally overlooked in classical GP. By restricting crossover to functionally compatible genes only, we have a high probability of producing viable individuals. In a few earlier experiments we had tried crossover at arbitrary points, and the result was the well-known GP phenomenon of intron bloating [2]. This phenomenon is the accumulation of useless code in individuals, leading to populations of individuals with very large genotypes, containing portions of code that serve no functional purpose, analogous to junk DNA in living organisms. These junk portions however protect the individual from destructive crossover operations, as the probability of crossover points falling inside an intron – and therefore not breaking existing useful functionality – increases with the percentage of introns in the individual. As soon as we introduced homologous recombination, the intron growing problem disappeared.

However, homologous recombination in a limited population of simple individuals with few genes, as shown in the experiments, leads to low genetic variability, and after a few generations most of the variability is lost.

Mutation is usually regarded as the main source of genetic variability in GP populations [2]. However, the benefits of mutation can only be observed at the very long run, since most mutations are lethal. In our short-run experiments, we have not been able to observe really productive mutations. We have to interpret these very preliminary results with caution; nevertheless, they seem to indicate that new, more intelligent techniques for evolving populations of genetic protocols need to be devised to make on-line evolution a reality.

Fitness evaluation still has a centralized component in our current set-up. This prevents the propagation of cheat programs, e.g. programs that lie about transmitted or acknowledged packets. Decentralized fitness evaluation is a non-trivial issue. Perhaps redundancy and reputation mechanisms could be combined to provide a safe and reliable way to evaluate the behavior of protocols at run-time.

We are starting to investigate how to add resilience at the level of individuals, as opposed to the level of entire populations as described in the experiments above. The idea is to combine our previous resilience work [9] with genetic programming, by modelling each protocol as tuples of redundant genetic code, analogous to chromosome pairs. This should in principle improve resilience, and help preserving genetic variability in small populations.

V. Conclusions and Outlook

In this paper we propose an intrinsic approach to the automated evolution of network software that can be used for self-deployment, self-configuration of functional modules and even automatic synthesis of protocol implementations. These are important elements of future self-managing, autonomic networks in which protocol code must be self-modifying. We argue that automated protocol evolution becomes feasible if the networking code is *resilient* such that we can have *competing* protocol variants running in parallel.

We have carried out adaption experiments using genetic programming to evolve code fragments based on a chemical execution model for protocols. These early experiments show that a system can automatically and gradually evolve depending on the environment it is confronted with, provided that a minimum variability of code instances is kept.

A more complex task, that has yet to be demonstrated, is an on-line version where software evolution is a continuous activity. Our experiments have provided some insights on the obstacles that have to be overcome before this objective can be realized. For instance, fitness evaluation in a decentralized and competitive environment is a non-trivial issue. Furthermore, we still lack a solution for keeping variability and recreating lost code fragments that suddenly become useful. A fundamental issue is how to improve genetic operators to obtain clever program transformation functions able to evolve genuine new code suited for unforeseen situations.

References

[1] Z. Manna and R. Waldinger, “Fundamentals of Deductive Program Synthesis,” *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 674 – 704, Aug. 1992.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming, An Introduction*. Morgan Kaufmann Publishers, 1998.

[3] T. Nakano and T. Suda, “Adaptive and Evolvable Network Services,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, ser. Springer-Verlag LNCS 3102, 2004, pp. 151–162.

[4] R. L. Probert and K. Saleh, “Sythesis of Communication Protocols: Survey and Assessment,” *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 468 – 476, Apr. 1991.

[5] A. P. D. Song and D. Phan, “AGVI – Automatic Generation, Verification, and Implementation of Security Protocols,” in *Proc. 13th Conference on Computer Aided Verification (CAV 2001)*, July 2001.

[6] N. Sharples and I. Wakeman, “Protocol construction using genetic search techniques,” in *Real-World Applications of Evolutionary Computing – EvoWorkshops 2000*, ser. Springer-Verlag LNCS 1803, Edinburgh, Scotland, Apr. 2000.

[7] N. Sharples, “Evolutionary Approaches to Adaptive Protocol Design,” PhD dissertation, University of Sussex, UK, Aug. 2001.

[8] K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer, “A system model for dynamically reconfigurable software,” *IBM Systems Journal*, vol. 42, no. 1, pp. 45–59, 2003.

[9] C. Tschudin and L. Yamamoto, “A Metabolic Approach to Protocol Resilience,” in *Proceedings of 1st Workshop on Autonomic Communication (WAC 2004)*, ser. Springer-Verlag LNCS 3457, Berlin, Germany, Oct. 2004.

[10] C. Tschudin, “Fraglets - a Metabolistic Execution Model for Communication Protocols,” in *Proceeding of 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*, Menlo Park, USA, July 2003.

[11] J.-P. Banâtre, Y. Radenac, and P. Fradet, “Chemical Specification of Autonomic Systems,” in *Proc 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, July 2004.

[12] L. Yamamoto and C. Tschudin, “Experiments on the Automatic Evolution of Protocols using Genetic Programming,” University of Basel, Switzerland, Technical Report, Apr. 2005.