Self-Evolving Network Software

Christian Tschudin and Lidia Yamamoto
Computer Science Department
University of Basel, Switzerland

Abstract

Future network systems with myriads of elements must become self-managed: Beyond autoconfiguration, they will have to evolve by themselves as context and elements change and to manage their code base by themselves in a self-organized way. Building such autonomic communication networks requires a radical shift in current network technology. In this paper we discuss the issues involved in network software self-repair and self-optimization with regard to automated network evolution. We survey current work in this area and point towards new research directions like autonomic communications. We then introduce the concept of *autocatalytic software* for computer networks that is based on a chemically-inspired execution model. In this framework, software is constantly regenerated as it runs, permitting to evolve by itself for reasons of self-repair, self-optimization as well as for innovation.

1 Introduction

As networks grow and become more complex, and networked devices embed themselves in a myriad of unsuspected places, routine operations traditionally performed by humans, such as surveillance, maintenance and software adjustments now require automation. Automation of all these management tasks is essential because the devices are expected to outnumber the human network managers by several orders of magnitude, while the network complexity will continue to grow. Moreover, some devices such as smart dust escape direct management. Traditional Internet-based networks would also benefit from increased automation, since today they still rely on considerable human intervention.

Automation in a decentralized network context necessarily implies self-organization, as the different parts of the system must interact to form a coherent whole that exhibits the desired properties, without resorting to a global supervision or management system.

In:
Praxis der Informationsverarbeitung und Kommunikation (PIK Magazine)
28 (2005) 4, December 2005, pp. 206-210.

Network self-management involves two cases: the first is a predefined, closed system of fixed rules able to manage itself independently, but which is not necessarily self-organizing. This system would only be useful in static situations that require few changes that can be done centrally and propagated to the rest of the network. We shall not mention this case any longer. The second and most useful case is an open system: open with respect to time, size, and types of changes performed on its elements: it must deal with a wide variety of changing conditions: self-management in this case implies self-organization in a distributed context.

More recently, routing solutions for wireless ad hoc network have been called "self-organizing" because of their use of local information (e.g. neighbor information) for achieving global tasks (multi-hop routing). For an overview article on self-organization in communication networks we refer to [22]. In this paper we target a much deeper meaning of self-organization that also covers the long-term evolution of the network's software. In section 2 we first review software evolution in a general sense, looking at software engineering, program synthesis and transformation techniques. We then elaborate on the difference between adaptation and evolution before focusing on the networking context in Section 3. There, the concept of autocatalytic software is introduced as well as the importance of resilience for the self-organization of network evolution.

2 Self-Evolving Software

Up to now the evolution of computer network systems has been mainly triggered by humans. However, in autonomic networks where manual configuration is difficult or impossible – for example sensors spread over a wide area, which cannot be accessed by humans – a system that does not allow on-line evolution is condemned to stagnation. In this section we first look at software evolution outside the context of networking.

The dream of software that evolves by itself in an unsupervised way can be traced back to the early days of artificial intelligence and machine learning: Eurisko [16] was a rule system that included heuristics on how to change its own heuristics. Holland [11] introduced genetic algorithms and classifier systems as tools for adaptation. Koza [13] later proposed to apply genetic algorithms to computer programs, creating the genetic programming field. Programs synthesized by genetic programming are known that have even generated patented or patentable inventions [14]. However, comparatively little has been achieved in the evolution of running systems.

Software evolution is still mostly carried out inside a software development process and by advances in the tools that help to create new software at design time. Recently, automatic reconfiguration and self-adaptive software systems [19, 15] allow

a controlled but relatively limited form of evolution during runtime. In this section we describe some of this research work, and outline our definition of self-evolving software systems, which extends previous definitions to encompass self-adapt and self-repair cycles in a composite system made up of several potentially competing subsystems.

2.1 Synthesizing New Programs

Several approaches to *Program synthesis* currently exist, which mostly focus on the off-line automatic generation of new programs. In *deductive program synthesis* [17], programs are formally derived as part of a theorem proving process. Synthesis through *generative programming* [5] helps the software developer to generate the most repetitive parts of the code from templates describing code patterns, thus partially automating the software manufacturing process. In [7] *schema-based programming* is proposed as a hybrid between deductive and generative approaches.

Automatic Design of Algorithms Through Evolution (ADATE) [18] is a synthesis method for functional programming, able to generate code as complex as to include recursive functions. An integrated method for program synthesis, transformation and verification for logic programming is proposed in [21]. *Dynamic software update* [10] is a method to generate and apply software patches at runtime.

None of these methods provide full automation, nor are able to deal with fuzzy or incomplete specifications. The same holds for automatic code writers [24] and runtime code generators [4], which generate code that is correct by construction. These are specialized compilers which translate from a higher level language (e.g. a graphical language or a formal specification) to a lower level one. As such, they rely on the correctness of the input high-level program or specification. The same applies to model-based systems, even when model transformations can also occur on-line [25].

2.2 Engineering Software Evolution

The view of software evolution as a spiral movement is well established in software engineering. Although this metaphor captures the software life cycle as humans work on it at design time, it can also be adapted to more automated contexts at runtime. In [9] an architecture that supports an *evolutionary software life cycle* is presented. It makes use of computational reflection to extend the software life cycle spiral to the usage phase, such that software continues to evolve on-site and automatically during its own operation, without going back to the software factory. However, this architecture is limited to reconfiguration of design information and as such is not really able to create new solutions.

In [6] a so-called *self-evolving software system* is able to automatically swap software modules according to observed conditions. Reconfiguration techniques based on module replacement are now extensively known. In this case, no new algorithms are really invented, as the system merely swaps existing modules according to a predefined recipe.

In [1], self-evolving systems are defined as those which automatically adapt to changing external situations and internal conditions. The paper proposes a framework for formally modeling time-critical self-evolving systems to deal with change in a reactive and reliable way. In this framework, the exact way to react to each change must be designed into the system, which is then prepared to handle only a handful of situations that have been predicted at design time.

Self-adaptive software [19, 15] is defined as software able to evaluate its own behavior and modify it when necessary. The most common method to achieve self-adaptation is the reconfiguration of the component graph in response to system observations, with selection and replacement of components at runtime.

Among all these approaches, the distinction between between self-evolving and self-adaptive software is blur. The common aspect among them is that a system's configuration can change at run time in an automated way. Moving configuration decisions from design to run time is a first step towards a self-evolving system, although its evolution potential remains limited.

2.3 Evolution vs. Adaptation

According to [19], the simplest level of self-adaptation are conditional expressions that are rather application specific because of explicit encoding of e.g., events and corresponding actions. At an intermediate level we have parametrized algorithms and algorithm selection, which already provide a better separation of adaptivity and functionality concerns; at the highest level we have the most generic adaptivity, where actual evolution of the code base can occur via evolutionary programming. An important question is whether adaption occurs only inside a predefined configuration space, or whether true inventions are possible that permit a system to change, and to find itself with new functionality and configuration options.

We define self-evolving software as a self-organizing software system designed to pursue its multiple goals in a fully autonomic way, able to perform all its operations unsupervised, including creating its own code parts for software update and modifying its configuration envelope.

This extends self-adaptability as defined in [19, 15] which refers to the ability of a single, self-contained software system to reconfigure its constituent parts in response to external or internal changes. An example from [19] is a guidance system for unmanned vehicles: each system component is integrated into a unique goal, so

conflicts arising within the system are either due to external changes or to internal errors. The system then constantly monitors itself to resolve such transient conflicts.

Our definition aims at a network setting where interactions occur among many different adaptive systems. Some of these interactions may be constructive, others destructive; some gently cooperative, others aggressively competitive. In such a context it is not sufficient to let (sub-)systems adapt according to their single goals, and it is impossible to impose a global goal. Instead, adaption has to be attempted through evolution that permits for the occurrence of new functionality, which we do not know in advance. Competition and selection will then act as driving forces for software evolution and are a way of sorting out conflicting goals; this is analogous to the way evolution occurs in biological ecosystems.

This view resonates with the interest in bio-inspired solutions for self-evolving systems. For example, self-evolution is shown in an Artificial Life system [8] based on a chemical reaction metaphor. Although the system has no explicitly programmed genetic operators, nor a selection strategy, a crossover operator is shown to emerge, indicating that evolutionary mechanisms can appear in a self-organizing way, out of simpler reaction operations. Similar work has been pursued within organic computing [20], such as [2], where regulatory processes that act on themselves are studied.

In the next section we are going to expand on these bio-inspired concepts due to their inherent potential for self-evolving network software.

3 Self-Evolving Networking Software

The difference between mere adaption and evolution is the starting point of this section. Before we introduce the concept of autocatalytic networking software, we discuss the role of resilience and adaptivity for developing self-organized and self-evolving networking software.

3.1 Resilience and Adaptivity

Many algorithms in networking are adaptive: the TCP protocol adjusts its transmission rate depending on measured network properties; routing protocols are able to adapt to arbitrary network topology. Also, networking protocols can be made resilient, for example reliable data transport continues to be provided despite packet loss and corruption, as well as broken links and crashing nodes. However, this is a restrictive form of adaptivity referred to as conditional expressions in [19]. Moreover, resilience is limited to cope with those events for which a protocol was designed.

In a context of a self-managing – or autonomic – network, these levels of adaptivity and resilience are not sufficient anymore. Baring the possibility of manual inter-

vention, a network has to be able to adapt to situations which are outside its original specification envelope. At the same time a network has to be resilient against yet unknown disturbances and changes. At this level, these two properties become antagonistic goals because any adaption, e.g. in form of an evolutionary step, could be considered as a harmful development and would be suppressed. As an example of this antagonism we refer to the problem of fighting network viruses while at the same downloading software upgrades over the network that alter the very core of a computer. Note that today these two vectors of change both require human invention and involve manual intervention.

The antagonism between adaptation and resilience can be used to the benefit of network software evolution. The idea is to have an automatic selection process in place that is able to choose among several possible configurations and algorithms. On the other hand, the system's resilience will make sure that "odd" combinations do not harm the network's operation. Based on feedback from users or applications, an autonomic network will adapt to new situations by combining service elements and trying them out in a resilient mode. This approach is similar to the ones presented in [6, 1, 19, 15] for classical software systems, except that the resilience property permits to select new combinations in a agnostic way, thus does not require goal-specific and pre-defined re-configuration rules. In [28], we have shown an automatic selection of protocols according to such a scheme. In the future, the resilience property should permit to integrate new service inventions, hence letting a system to truly evolve beyond adaption and its original configuration envelope. With resilience in place, adaption and evolution can be carried out in a self-organized way.

3.2 Autocatalytic Networking Software

We refer to a network that steers and caters for its own software production as an autocatalytic networking system. In cellular biology, autocatalytic proteins are substances which promote their own production i.e., they act like a catalyst for themselves. In network evolution, we are interested in software that controls its own generation, configuration and its deployment. In other words: we seek software that organizes itself in a networking context.

The model behind autocatalytic networking software is that of a dynamic code system that is constantly rewriting itself. This is a desirable behavior for a distributed code system where parts of its code might be lost (and must be redeployed), where parts might be corrupted (due to malicious activities), and where parts of the code have to be changed (for incorporating beneficial changes). First efforts in this direction are reported in [26], where we show how a protocol implementation can be made resilient against the deletion of an arbitrary code fragment. The peculiar aspect of such a code system, which is based on a chemical execution model for commu-

nication protocols, is that the code must survive a deletion attack (robustness), must detect tampering (self-diagnosis) and be able to recover (self-healing) for resilience. Moreover, any logic that is added for providing these features is also potential subject to the deletion attack, hence must also cater for its own protection, tampering diagnosis and self-healing capacity. So far, no such system could be demonstrated that would cover all aspects.

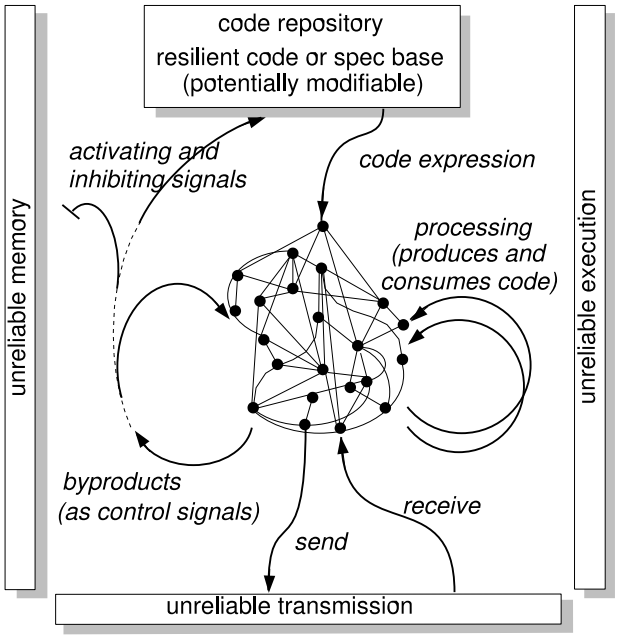

Figure 1: Autocatalytic Software Execution Model

Figure 1 shows a possible framework for the operation of autocatalytic software and the hostile environment it has to operate in. This framework is an extension of our previous work [27] on self-modifying code as a tool for resilient software. Figure 1 depicts the execution of autocatalytic software in a single node, where unreliable storage, execution and transmission form its environment. A resilient specification or implementation source contains the initial code base that is inserted into a proper execution environment. We call "code expression" the action of running the code in the environment, in analogy to gene expression in biology. This is because *running* the code in this environment requires to *produce* the code in a first place, in a process similar to the way chemical reactions in cells lead to consumption and production of substances. A concrete example of this chemical execution of protocols will be

shown in Section 3.3. Such execution is heavily based on byproducts of chemical reactions that serve as activator or inhibitor signals to control code expression, by blocking, boosting or modifying code consumption-production cycles. These execution byproducts could also affect the initial code base introducing persistent code transformations and thus long-lasting changes.

Note that the execution flow naturally extends itself to a distributed context with several nodes. In Figure 1, this is represented by the send and receive arrows, through which code is shipped between nodes.

Network software traditionally copes with unreliable transmissions, as well as complete node failures or connectivity changes, but still assumes reliable execution and storage. The reason why autocatalytic software would be able to run despite transient packet loss, memory errors and unfaithful execution lies in its constant regeneration of its own code base which – with appropriate coding – converts the software into a perpetually "moving target" that cannot be harmed by spurious incidents.

3.3 Prologue to Autocatalytic Code: A Metabolic Protocol Diagram

In [28] we show experiments on genetic programming to automate protocol evolution, using a simple Reliable Delivery Protocol (RDP). An execution diagram for this protocol is shown in Figure 2. RDP was implemented using a chemical execution language that we call Fraglets. Fraglets, which stand for "computation fragments", are used for representing both data carriers and executable rules. Like molecules, fraglets operate on each other and undergo transformations like splitting up or transmitting themselves to a remote node. This forms a metabolic network where the execution of one fraglet rule leads to the creation of the next fraglet to be executed.

In Figure 2, each node of the graph contains a token that represents a stage in the chemical execution process, while arrows represent transitions that trigger further reactions. The main execution cycle is activated by data payloads obtained from the application, which must be transmitted from one node to another. The token *data* represents the start of this cycle. A series of reactions then proceed until node *d3* splits into an acknowledgment waiting path (starting with token *m0*) and a retransmit cycle (token *retransmit*). The ack waiting path ends up with the production of a rule that waits for the arrival of an acknowledgment, and reacts to it by producing an inhibitor for the retransmission cycle. This inhibitor kills the pending retransmission request (which holds a copy of the data to be retransmitted, the *datacopy* token) by reacting with it and generating a *null* (empty) fraglet as a byproduct. If there is a packet loss, after a timeout the stored copy of the data payload will be retransmitted, and a new wait cycle is initiated.

Although showing first traits of autocatalytic software, our system so far still requires a constant code base that steers the whole metabolic process. Indeed, most

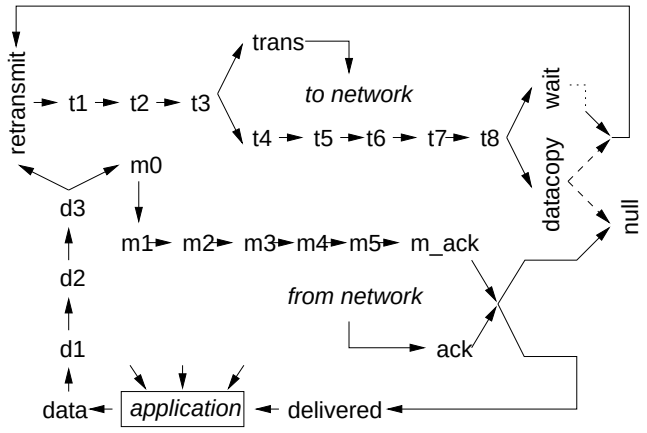

Figure 2: Metabolic execution diagram of a simple reliable delivery protocol

transitions indicated in Figure 2 require a persistent fraglet to trigger the reaction. In the future we would like to produce a system that does not depend on fixed fraglets but creates those code fragments that it needs for execution as it goes: Most of the code would then be regenerated at run-time according to various inhibition and activation code fragments that regulate code expression, according to local constraints and code exchange with the other nodes in the network.

3.4 New research directions

Self-organization in a deep sense (including adaptivity *and* evolution) is the core vision of autonomic communications. However, this research field has just emerged and has yet to produce first breakthrough results. In this section we restrict ourselves to three interesting problems of automated network software evolution and the fields that might be relevant for making progress on these questions.

A first topic are execution models which permit the expression of robust protocol implementations. Results from the domain of compiler construction point to the possibility of making sequential programs more resilient in a mechanical way [23]. A more natural approach with a firm theoretical foundation could be multiset processing systems like Gamma [3] (on which fraglets are based) which permit to have a multiplicity of rules working in parallel. In both cases we need a new design methodology that exploits these execution primitives for obtaining more resilient protocol implementations at different levels of granularity.

A related research question is the encoded execution of protocols. Instead of direct encoding of protocol actions, a protocol could be transformed into a multi-step

procedure that spreads over multiple execution entities. Such a technique has been used in quantum computing to permit the execution of logical gate operations despite noisy execution at the physical level [12]. The problem solved there is similar to the one mentioned on self-diagnosing protocol implementations: error recovery logic executes on the same noisy substrate wherefore it is subject to the "robustification" itself. While the solution found in quantum computing works fine for fixed processing actions, we must be able to apply such encoded execution to dynamic settings and at higher granularity than logical gates.

Finally we point to the problem of designing a self-organized distributed evolution core that is able to steer a network's software and configurations in a perpetual way. Instead of depending on handwritten code, the autonomic network would be able to produce new functionality by itself, for example by incorporating genetic programming capabilities. One problem is the design of the corresponding online evolution environment where, unlike in simulated ecosystems, it is not possible to impose evolution in "generation steps" due to the fully distributed network. The other problem is the design of a robust selection procedure and the assessment whether a protocol provides a service as claimed or not. Because this assessment is key for the selection it must not be possible to deceive it. Both problems contain a wealth of research questions and implementation considerations, which can be put in relation with ways of producing a variety of protocol implementations in more deterministic ways and with assertable properties.

4 Conclusions

Research in autonomic communications has the formidable goal of self-organizing the operations of computer networks. We have made the point that such networks ultimately are in charge of handling their own evolution: Beyond mere adaptation to changing topologies and packet loss rates, networking software has to be permitted to evolve. An intermediate quantitative step in this direction is the classical self-organization in terms of automated configuration procedures (for example as in ad hoc networks) and the shift of algorithm selection decisions from design to run-time. Eventually, autonomic networks will become self-organized evolving systems with the ability of qualitative internal changes. Autocatalytic software is a key concept in this context that captures the dynamics of the self-referential aspect. Unlike current networking software that clearly separates generation from installation, we envisage a mode of operation where the software produces its next instantiation for the sake of resilience and evolvability.

References

[1] Vangalur S. Alagar, Ramesh Achuthan, M. Haydar, D. Muthiayen, Olga Ormandjieva, and Mao Zheng. A rigorous approach for constructing self-evolving real-time reactive systems. *Information & Software Technology*, 45(11):743–761, 2003.

[2] Wolfgang Banzhaf. Artificial Regulatory Networks and Genetic Programming. In R. Riolo and B. Worzel, editors, *Genetic Programming - Theory and Applications*, pages 43–61. Kluwer Academic, Boston, MA, 2003.

[3] Jean-Pierre Banâtre and Daniel Le Métayer. A new computational model and its discipline of programming, September 1986. Technical Report RR0566, INRIA.

[4] Alessandro Coglio. Toward Automatic Generation of Provably Correct Java Card Applets. In *Proc. 5th ECOOP Workshop on Formal Techniques for Java-like Programs*, July 2003.

[5] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] Chrysanthos Dellarocas, Mark Klein, and Howard Shrobe. An Architecture for Constructing Self-Evolving Software Systems. In *Proc. Third International Workshop on Software Architecture*, pages 29–32, Orlando, Florida, USA, 1998.

[7] Ewen Denney and Jon Whittle. Combining Model-driven and Schema-based Program Synthesis. In *Proc. 1st International Conference on the Applications of UML and MDA to Software Systems (UMSS'2004)*, Las Vegas, Nevada, USA, June 2004.

[8] Peter Dittrich and Wolfgang Banzhaf. Self-Evolution in a Constructive Binary String System. *Artificial Life*, 4:203–220, 1998.

[9] Ahmed Ghoneim, Sven Apel, and Gunter Saake. Evolutionary Software Life Cycle for Self-Adapting Software Systems. In *Proc. 7th Conference on Enterprise Information Systems (ICEIS'05)*, Miami, Florida, USA, May 2005.

[10] Michael Hicks and Scott M. Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, September 2005.

[11] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992. First Edition 1975.

[12] E. Knill. Quantum computing with realistically noisy devices. *Nature*, 434:39–44, March 2005.

[13] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[14] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV : Routine Human-Competitive Machine Intelligence*. Springer, July 2003.

[15] Robert Laddaga and Paul Robertson. Self Adaptive Software: A Position Paper. In *International Workshop on Self-* Properties in Complex Information Systems*, Bertinoro (Forlì), Italy, May-June 2004.

[16] D. B. Lenat. EURISKO: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, 21:61–98, March 1983.

[17] Zohar Manna and Richard Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18(8):674 – 704, August 1992.

[18] J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–83, March 1995.

[19] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[20] Organic computing home page. http://www.organic-computing.org/.

[21] Alberto Pettorossi and Maurizio Proietti. Program Derivation = Rules + Strategies. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond (Essays in Honour of Robert A. Kowalski - Part I)*, Springer, LNAI 2407, pages 273–309, 2002.

[22] C. Prehofer and Christian Bettstetter. Self-Organization in Communication Networks. *IEEE Communications Magazine*, 43(7):78–85, July 2005.

[23] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proc. 3rd International Symposium on Code Generation and Optimization*, March 2005.

[24] John Thomas Riley. Automatic Code Writing Agent. In Walt Truszkowski, Mike Hinchey, and Chris Rouff, editors, *Innovative Concepts for Agent-Based Systems: First International Workshop on Radical Agent Concepts (WRAC 2002)*, Springer, LNCS 2564, Berlin, Germany, pages 451–452, McLean, VA, USA, January 2002.

[25] Paul Robertson and Brian Williams. A Model-Based System Supporting Automatic Self-Regeneration of Critical Software. In *IFIP/IEEE International Workshop on Self-Managed Systems and Services (SelfMan 2005)*, Nice, France, May 2005.

[26] Christian Tschudin and Lidia Yamamoto. A Metabolic Approach to Protocol Resilience. In *Proc. 1st Workshop on Autonomic Communication*, Springer-Verlag LNCS 3457, pages 190–205, Berlin, Germany, October 2004.

[27] Christian Tschudin and Lidia Yamamoto. Harnessing Self-modifying Code for Resilient Software. In *Second NASA GSFC/IEEE Workshop on Radical Agent Concepts (WRAC)*, Greenbelt, MD, USA, September 2005. To appear in LNCS/LNAI 3825.

[28] Lidia Yamamoto and Christian Tschudin. Experiments on the Automatic Evolution of Protocols using Genetic Programming. In *Proc. 2nd Workshop on Autonomic Communication*, Athens, Greece, October 2005.