Harnessing Self-Modifying Code for Resilient Software

Christian Tschudin and Lidia Yamamoto

Computer Networks Group, Computer Science Department
University of Basel, Bernoullistrasse 16, CH–4056 Basel, Switzerland
{christian.tschudin|lidia.yamamoto}@unibas.ch

Abstract. In this paper we argue that self-modifying code can become a better strategy for realizing long-lived autonomous software systems than static code, regardless how well it was validated and tested. We base our discussion on three facets – self-repairing software, adaptive software and networked systems – for which we point out ongoing and related work before presenting a roadmap towards a controlled framework for self-modifying code.

Keywords: resilient software, self-healing protocols, computational agents, autonomic communication.

1 Introduction

In a famous article series in Scientific American from 1984, Dewdney introduced the "core wars" game [1]: Imprisoned in a small memory bank, two programs fight against each other by peeking and poking into the adversary's memory cells and by taking care of retaining the own software's integrity through repair actions or dislocation. On the hardware side, von Neumann considered unreliable logical gate implementations and showed in 1956 how to use redundancy to achieve fault tolerance despite defective execution elements [2].

These examples illustrate the simple idea that the road towards robust software starts with the humble assumption that no system or software is perfect and therefore any of its parts may fail. While mission-critical systems have considered robustness and fault-tolerance techniques at the core of their design, solutions are still lacking for light-weight and widely deployable *autonomous software* that is resilient and able to survive unsupervised in a large spectrum of situations.

One can imagine scenarios requiring autonomous distributed systems, where large networks of small devices must run unsupervised during a long period of time, such as sensor networks and embedded networks of pervasive computing devices, but also constellations of spacecrafts sent to distant outer space regions where ground control from Earth becomes impractical even for non real-time actions [3, 4].

Two notions are combined here: adaptability and resilience. Ideally, a fully autonomous software system system should continuously update itself, recovering from failures and adapting to new situations. Since no realistic software

To appear in: Proc 2nd IEEE Workshop on Radical Agent Concepts (WRAC), NASA Goddard Space Flight Center Visitor's Center, September 2005.
Springer LNCS/LNAI 3825, 2006.

system could possibly anticipate all future situations, such updates necessarily include changes to the system's own code base. Today these changes are largely performed off-line either by humans or in a semi-automated way with the help of software engineering tools able to generate parts of the code. Hardware oriented hardening techniques have received continued attention since von Neumann's work – most recently in quantum computing where inevitable execution noise has to be compensated through error correcting quantum execution circuitry. On the other hand, software oriented solutions have targeted fault tolerance by adding instead of changing components: A *fully* software based solution that considers memory, communication and execution errors and which provides self-healing capabilities is still lacking. We believe that self-modifying code, although banned from computer science because it is difficulty to debug and to assess its correctness, will play an important role in this quest.

Before discussing a roadmap towards a better mastering of self-modifying adaptive and resilient software, we show in three fields (soft errors, software adaption, mobile code) where software resilience has been envisaged and how self-modifying code comes into the picture.

2 Handling Soft Errors in Software

Soft errors are transient faults due to radiation (particles discharging a semiconductor capacitor) or other low level physical phenomena (noise in a quantum computing device). The effect is that either state is altered or instructions are incorrectly executed. An important counter measure against such incidents is to apply systems-level i.e., hardware changes. For example, memory cells store data in an error correcting code format that can revert a limited number of bit flips. Similarly, quantum computing circuits have been proposed that permit error correcting execution, hence masking away low level errors.

However, it is also possible to mitigate or even neutralize soft errors by software techniques alone. In [5], a compiler is described that schedules redundant instructions such that additional comparison instructions can detect whether an execution thread encountered an error or not. This code transformation approach relies on a reliable memory subsystem, but other proposals have been made that also include memory in the "sphere of replication". The SWIFT approach in [5] targets bit failures and applies to programs that can be recompiled. Low-level errors are again masked away and the program itself is not made aware of them. A small time window remains just after validation where validated state could be corrupted unnoticed, hence the sequential character of the execution environment becomes visible. Another weak point is that (erroneous) store operations are not observable.

Clearly, these software-implemented fault tolerance techniques can be applied to a software system at design time. We note however, that at run-time, the probability of fatal failure after an uncatched soft error (e.g., because too many bits were flipped etc) is high as the program itself relies on the installed safe guards. Such unnoticed errors will accumulate if no other self-checking mech-

anisms are put into place. In addition to run-time only detection of execution errors, a software system should periodically verify its integrity and, if necessary, regenerate its own code base and memory (self-healing). This is further complicated if the software is capable of adapting to new situations (see next section), or is extended with imported functionality (see section 4 on networking), so that there is no constant reference point to which the current state of the system could be compared to. Overall, changed, imported or regenerated functionality are all special cases of self-modifying code, although at different levels of programmability[1].

3 Beyond Planned Adaptation – Run-time Software Evolution

The ability to adapt to new situations is an essential property of any system that is expected to operate unsupervised. Adaptive software systems capture information from their environment and make changes in their internal structure in order to achieve a desired performance in that environment. These changes are performed according to an adaptive plan [6], which determines the transformation operations to be applied to the current structure to obtain a new one.

No matter how comprehensive an adaptive plan might be, in terms of covering the space of possible combinations of environment state variables, there will always be cases that have been left out of the plan, or for which the plan is not sufficiently reactive. Those cases generally call for humans to intervene and upgrade the plan.

A change in the environment that had not been predicted at design time causes one or more system components to become imperfect because they are no longer well adapted to the task at hand. The system must then diagnose and correct these imperfections, making the components suitable again. The same behavior leads to both adaptability and resilience. The central question is then how to automate this diagnose and repair process, not only for the system configuration but also for the running software. In other words, how to build software that genuinely evolves to correct itself and to match new expectations.

Structures that are subject to adaptation also include programs. A common technique applied to adapt programs is genetic programming (GP), in which programs are modeled as genetic material (genotype) and the adaptation plan consists in the way to apply genetic operations (e.g. crossover, mutation) to a population of programs to obtain new ones. GP can potentially cover a large environment space with a very simple plan, due to its problem-agnostic adaptive plan and high parallelism of solution search. However, it is slow to react due to the random nature of the code transformation operations, which cannot

[1] Note that the SWIFT approach collides with self-modifying code (at the level of CPU instructions): the compiler will not be able to apply appropriate resilient code transformation at compile time as the code to protect will only become available at run time.

ensure viable programs. Non-viable programs can only be detected after they obtain a low fitness value during execution. This limitation has prevented GP from being applied to running programs. However, in a resilient system as the one we describe below, invalid programs could be detected and eliminated with minimum impact in the system. GP would then become feasible in these systems, and we have started to explore this path in [7]. The question remains open whether improved genetic operators could be found to accelerate the search and to increase the likelihood of viable results, while at the same time keeping the plan to a minimum.

Formal techniques have been applied to generate code from specifications [8, 9], however the focus has been on software engineering where tools are not fully automated and not directly applicable to running code. The adaptive plans resulting from these techniques are likely to become complex.

The quest for minimum plans is directly related to the quest for fully resilient software, with no "fixed" parts that cannot be adapted. The plan is one such fixed part, since the plan itself is not part of the structures subject to adaptation. Although GP plans are already small, researchers have taken steps towards reducing them even further by incorporating GP parameters into the genotype of the individuals being adapted [10], therefore including part of the plan into the adaptive structures themselves.

4 Networking and Mobile Software

Now consider the case of self-modifying distributed software, such as a distributed system or a network protocol stack. One can easily anticipate the difficulties in keeping the different pieces of the software fully functional and compatible, such that they can run in a consistent way to provide the intended service. Once a new software has been generated, it must deploy itself, i.e. ship its own constituent parts to the regions of the network where they are necessary. Code mobility is therefore a natural and essential ingredient of distributed and adaptive software.

The desired joint resilience/adaptability property discussed so far for a single system needs to be extended to the distributed context as well, creating robust distributed execution circuits that can recover from several disturbances such as message loss, losses of parts of their code base in different parts of the network, transient inconsistencies and temporarily unsuitable code portions.

Attempts on code mobility include active networking and mobile software agents. The main aspect of these approaches is the late binding of functionality which permits the adaptability to network conditions and user requirements at run time. Because they focus on the run-time-deployment of new (but fixed) functionality, existing mobile code systems like [11, 12] cannot be easily used to build the robust self-healing execution circuits that we are aiming at. The main reason for that is the lack of access to the representation of the code itself, which would permit self-modification. Mobile code systems like M0 [13] have been proposed where a program is able to access the representation level of mobile

code units, leading to the possibility to create (compute) new code pieces at run-time. Such an expressiveness permits a much deeper degree of self-modification and enables otherwise not implementable protocols like e.g., self-compressing or self-fragmenting mobile code units. Moreover, because of safety concerns (and the implicit virus-like nature of mobile code), programming languages and execution environments like [11, 12] have been deliberately restricted which removes the ability to write self-modifying mobile programs.

More advanced yet more secure functionality based on self-modifying code could be permitted if there was a methodology to generate safe self-modifying programs. One currently hypothetical option would be to design an environment for mobile *specifications* (instead of mobile code) that can capture the essence of self-modification but leaves the actual implementation to a safety-property preserving run-time code generation mechanism. In the following section we come back to this aspect of run-time code generation, where code is not only generated once but every time before it is executed.

5 A Roadmap for Self-Modifying Code

We see three related research challenges where more insights for using self-modifying code are needed:

- execution models,
- software regulation techniques, and
- self-modifying code generation methods.

A crucial element and research target for resilient software is an appropriate *execution model* and the question how such an environment supports self-modifying code. The execution model encompasses the properties of the memory, the execution part as well as the communication primitives. From results in fault tolerance it seems that the parallel execution of activities is an essential element of an execution model. However, first results (discussed in Section 2) show that soft error resilience can also be achieved inside a single-threaded sequential execution environment (although preventing self-modifying code). Regarding the effective implementation of resilient software, more research is needed to understand the requirements on the execution model.

In our research so far we have been working with tiny computational agents that we call Fraglets [14]. Fraglets are concurrent computation fragments similar to chemical molecules such that they can operate on each other. Code and data are processed by reactions between Fraglets, thus putting code self-modification at the heart of the execution model.

In Figure 1 we present a Fraglet inspired execution model that shows essential processes for a purely software-based resilience approach. The software system is placed in a hostile environment where memory faults (soft error, but also loss of memory content due to programming errors or contention for memory), execution errors (altered code, either transiently or permanently) and transmission errors will occur. Talking in networking terms, it suffices that this hardware

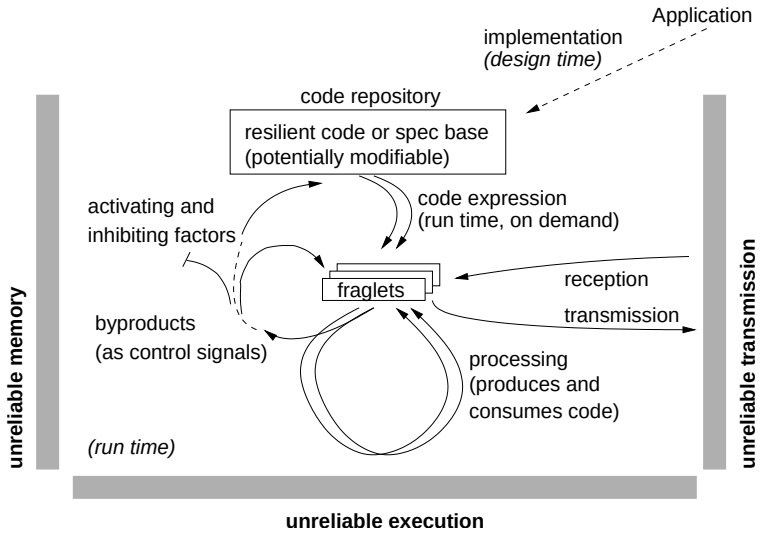

Fig. 1. Execution model for resilient and self-modifying software

offers *best effort services*, memory as well as execution wise. The software is organized in parallel activities, shown as separate execution circles. Activities are basically execution circles because we are working with continuously operating autonomous software.

One hypothesis in our model is that "code" is not permanently installed but is explicitly generated when it is needed. Moreover, Fraglet code is consumed (by executing it), which has the advantage that execution of accidentally modified fraglets is not repeated. Special copy processes (dashed line) are responsible for procuring the code when it is needed[2].

A resilient code repository, shown as a box in Figure 1, could be implemented as specially hardened read-only memory. However, due to our interest in online software evolution, we envisage that the repository is implemented with Fraglets, too: One or more continuous processes watch over the code's integrity and interact with the code expression requests from the copy processes.

The challenging part of such an execution model is the design of the *code generation control loop* that regulates code expression. Fraglet execution should be shadowed by signals (in form of Fraglets) which permit to assess its success or failure. A lost Fraglet execution would be compensated by the recreation of the corresponding Fraglet instance. But also excess executions due to accidentally or

[2] When comparing this model to the SWIFT approach (or any classical von Neumann CPU, see the discussion in Section 2 on soft error resilience with SWIFT), we note the similarity in the code copy operation which happens between the memory and the CPU: in SWIFT the code is copied instruction wise. In classical CPUs the code copy process (instruction fetch) is hardwired and cannot be altered.

deliberately interfering processes (bugs or viruses) could be handled by lowering the rate at which Fraglets are expressed. Thus, we believe that for continuously running software we have to develop code regulation techniques for highly interdependent computations such that all code related errors can be handled in a much more elastic way than the current run-or-break property of sequential programs.

The third research challenge relates to the safe exploitation of the available code self-modification mechanisms of such an execution environment. Ideally, one wants to let a system evolve through its internal code modification procedures while still being assured about long term invariants. Research in programming languages has produced "tamed" versions of self-modification e.g., in form of reflection or the lambda calculus. However, unreliable execution as well as code mobility are not accounted for. We are also not aware of a theoretical framework for code transformations that target self-modifying code and which could, for example, capture self-fragmentation or the dynamics of code regulation.

6 Conclusions

Autonomous software which runs unattended and perpetually should be able adapt to unforeseen situations and to cope with drastic error conditions at all levels, including external memory and execution faults as well as buggy software components. In this paper we pointed to the properties of adaptability and resilience that such a system must have and opt for a fully software based solution. We argued that important techniques for implementing it boil down to mastering self-modifying code, for which we presented a framework and research roadmap.

References

1. Dewdney, A.K.: Recreational Mathematics – Core Wars (May 1984) Scientific American. See also `http://www.koth.org/`.
2. von Neumann, J.: Probabilistic logics and the synthesis of reliable organisms from unreliable components. Automata Studies (1956) 43–98
3. Truszkowski, W., Hallock, H.: Agent Technology from a NASA Perspective. In: Proceedings of the 3rd International Workshop on Cooperative Information Agents (CIA'99). Lecture Notes in Artificial Intelligence (LNAI 1652), Springer-Verlag (1999) 1–33
4. Robertson, P., Williams, B.: 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 (2005)
5. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: Proceedings of the Third International Symposium on Code Generation and Optimization. (2005)
6. Holland, J.: Adaptation in Natural and Artificial Systems. MIT Press (1992)
7. Yamamoto, L., Tschudin, C.: Genetic Evolution of Protocol Implementations and Configurations. In: IFIP/IEEE International Workshop on Self-Managed Systems and Services (SelfMan 2005), Nice, France (2005)

8. Manna, Z., Waldinger, R.: Fundamentals of Deductive Program Synthesis. IEEE Transactions on Software Engineering **18**(8) (1992) 674 – 704
9. Probert, R.L., Saleh, K.: Synthesis of Communication Protocols: Survey and Assessment. IEEE Transactions on Computers **40**(4) (1991) 468 – 476
10. Spector, L., Robinson, A.: Multi-type, Self-adaptive Genetic Programming as an Agent Creation Tool. In: Proceedings of the Workshop on Evolutionary Computation for Multi-Agent Systems (ECOMAS-2002), International Society for Genetic and Evolutionary Computation. (2002)
11. Wetherall, D., Guttag, J., Tennenhouse, D.: ANTS: Network Services Without the Red Tape. IEEE Computer (1999) 42–48
12. Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A., Nettles, S.: PLAN: A Packet Language for Active Networks. In: Proceedings of the 3rd. ACM SIGPLAN International Conference on Functional Programming Languages (ICFP'98). (1998) 86–93
13. Tschudin, C.: The Messenger Environment MØ - A Condensed Description. In: Mobile Object Systems - Towards the Programmable Internet (MOS'96). Springer LNCS 1222, Linz, Austria (1996) 149–156
14. Tschudin, C.: Fraglets - a Metabolistic Execution Model for Communication Protocols. In: Proceeding of 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA (2003)