Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 317–325
doi: 10.14794/ICAI.9.2014.2.317

Quo vadis, self-* software?*

Veronika Szabóová, Csaba Szabó

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Veronika.Szaboova@tuke.sk, Csaba.Szabo@tuke.sk

Abstract

The research on self-* software began with Laddaga's DARPA report [1]: "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do." As time went, there was significant growth in the field of self-adaptive systems, many self-* properties were defined to allow better focused research. Guinea presented in [2] in 2013 a very important process characteristics as well: "Its life-cycle cannot be stopped after its development and initial setup." Which means that the challenge of self-* systems is to "live and change." Changes alter the software itself but could reconfigure its operating environment as well. Or implement the adaption using both ways. We focus on internal changes. Regarding to these self-modifications, Salehie and Tahvildari wrote in [3]: "there is still a lack of powerful languages and frameworks that could help realize adaptation processes." Our goal is to address this problem in this article. We focus on implementation of self-modification and related issues of self-testing of the changes introduced.

Keywords: self-adaptive, self-testing, self-* properties

MSC: 68N30; 68N20

1. Introduction

The idea of self-adaptive systems is very old. We can find it in early science-fiction novels of Isaac Asimov or in Frank Herbert's Dune as well, where a rule-based system controlled the system behavior and allowed adaption to changing situations. "Thinking machines" and "robots" characterize the years 1939-72, but the idea continues until today.

*This work was supported by the Cultural and Educational Grant Agency of the Slovak Republic, Project No. 050TUKE-4/2013: "Integration of Software Quality Processes in Software Engineering Curricula for Informatics Master Study Programme at Technical Universities – Proposal of the Structure and Realization of Selected Software Engineering Courses."

Software and hardware implementations needed to wait for required technologies. In them, there could be easily found the influence of science-fiction novels or their authors, respectively.

Self-* systems first occurred in the second half of 1990's. Several companies such as IBM, Sun, HP and Microsoft started the research – at the beginning, the focused on military applications, later network and operating system oriented solutions appeared. For commonly used software, the technology needs improvement. Actually, each implementation is architecture dependent mostly having a distributed architecture. Distributed architecture is similar to network architectures, which allows the adoption of several computer network and hardware related algorithms. These algorithms deal with a subset of self-* properties, mainly related to robustness or self-configuration.

We think that besides existing (mostly Java-based) implementations, there is an interpreted language as alternative in self-* systems development. Creation of a framework for interpreted self-healing (FISH) seems to be possible using interpreted or scripting languages. We focus our paper on this innovative idea. Our goal is to present background and history, which we think directly lead to FISH.

2. Background

Before Laddaga's DARPA report [1], there is a long evolution of software architectures and viewpoints between small programs and self-healing software systems.

2.1. Development of software architecture

Parnas [4] started software architecture research as the evolving separate discipline of software engineering with defining the term modular decomposition of software systems.

Later, Shaw expressed in [5] the need of higher level of abstraction than the one provided by modular decomposition.

The term *software architecture* was first used by Perry and Wolf in [6], their most referred article presenting fundamentals is [7]. Since these publications, software architecture becomes a well known phrase, later a standalone discipline of software engineering.

2.2. Software architectural views

The viewpoint on software architecture also changed and developed in time. While Parnas [4] identified three structures for modules, usage and processes, Perry and Wolf [7] talk about views:

- data,
- computation,

- connectors.

The above views started a new thinking about software, many development methods and strategies rely on them. The first significant improvement was developed by Kruchten [8] when he described the 4+1 architectural views approach, which uses the following views:

- logical,
- development,
- process,
- physical,
- scenarios.

Rozanski and Woods extended in [9] Kruchten's approach by new views and a new dimension called perspectives. Views and perspectives separate and collect selected aspects of software. We state that this approach is the key to self-* system design and implementation.

2.3. Software architecture for self-adaptive systems

Specific software architectures for self-* systems exist too. At the beginning, mostly existing architectures were modified to achieve a subset of self-* properties. Gorlick [10], Taylor [11] and Magee [12] worked with component-based and distributed systems.

Oreizy et al. [13] were the first addressing the architecture of a self-adaptive software as a different and new architecture. Kramer and Magee in [14] presented the architectural challenge on self-managing systems. Latest paper to mention, there is a definition of architectural style for distributed architectures for component coordination in the work of Baresi et al. [15], where self-organizing group topologies are discussed. The presented level of self-adaptiveness is very high, because groups can change their leads autonomously by voting, which is a new approach to rule management in such systems.

2.4. Categories of self-systems

Miller presents the categories of self-* systems in [16] using the self-CHOP (configure, heal, optimize, and protect) abbreviation, the shortest explanation of the topic is possible by Fig. 1.

At the lowest implementation level, one could consider self-adaptation in a `switch` or `if-then` statement already. Higher levels are represented by exception handling. The actual top level includes autonomic resource management and automatic updates. All that is mainly implemented using compiled languages. This also implies two significant limitations. The first limitation is that there is a lack of

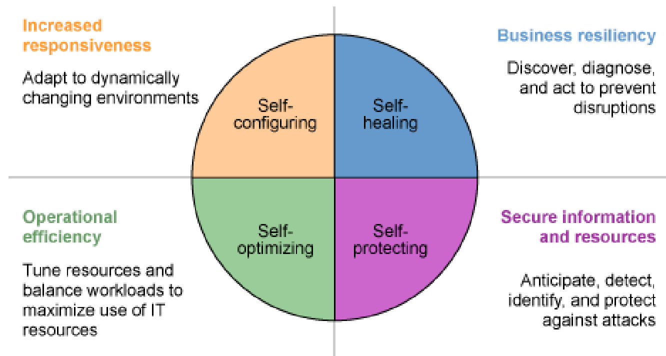

Figure 1: Self-* system classification [16]

self-modification of components; only the whole component can be replaced in the best situation. Second limitation is that the mechanisms require an extra rule base and rule inference to determine the action; this action planning is using developer-defined symptoms instead of deriving the symptoms from the architecture and requirements automatically, which strongly limits self-adaptiveness.

All of the system categories presented above require a very flexible management. There are overlapping parts of implementation too. We see that adaptation management is the most critical problem in autonomous computing. I.e. everyone would like to use a system adopting to her/his changing requirements, but no one wants to fight a self-willing software.

3. Motivation

Our motivation is to create a self-healing system architecture, which is a real self-adaptive system – not a self-willing one. The architecture will be then evaluated in numerous case studies and will undergo a theoretical proving mechanism. All these goals are actually distant.

In this article, we aim to get closer to a solution from a different angle. This point of view is technology.

Why? Because Salehie and Tahvildari wrote in [3]:

> There is still a lack of powerful languages and frameworks that could help realize adaptation processes.

We show in this article that there is a language with the power to be used here. A language with possibility of runtime changes of the implementation source codes. The changes are also evaluated before they got introduced. The most significant limitation is that the language is not used in every application domain and therefore, there is a lack of support of several application domains.

4. Perl in self-* systems

When selecting a language for self-adaptive systems, one needs to consider support for several implementation-related attributes:

- lazy execution, which allows fast reaction to changed environment,
- managed, which means that a garbage collector or equivalent is always performing a controlling mechanism over the system,
- interpreted, because interpreted modules can be changed during execution of other modules, which makes the change implementation faster (no compilation is needed, which fastens evaluation as well)
- interactive, which property is needed for explicit management (system administration) and also supports module updates.

These properties need to be included with the language – mostly as native language constructs or libraries, because the general self-healing process implements the following:

- changing the code,
- compiling the new version,
- checking or testing the new version for the production environment,
- introducing or deploying the new version, which also means removing the old one.

Perl allows explicit lazy evaluation of statements. Dynamic processing of user inputs is significantly different to this approach, because the `eval` function allows delaying or denial of execution that is impossible when dynamically processing user inputs, which run in a so-called interactive mode. Lazy evaluation in Perl language is frequently used. There are two alternatives [17]:

eval EXPR In the first form, the return value of `EXPR` is parsed and executed as if it were a little Perl program. The value of the expression is first being parsed, and if there were no errors, executed in the lexical context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. The value is parsed every time the `eval` executes. This form is used to delay parsing and execution of `EXPR` until run time, that could be considered as the lazy property.

eval BLOCK In the second form, the code within the `BLOCK` is parsed only once – at the same time the code surrounding the eval itself was parsed – and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first one, while also providing the benefit of checking the code within `BLOCK` at compile time.

In both forms, the return value is the value of the last expression evaluated inside the mini-program. If there is a syntax error or run-time error, or a `die` statement is executed, `eval` returns `undef`, and `$@` is set to the error message. If there was no error, `$@` is guaranteed to be the empty string.

With an `eval`, there exist four cases to remember what is being looked at when:

```
1 eval $x;          # CASE 1
2 eval "$x";        # CASE 2
3 eval '$x';        # CASE 3
4 eval { $x };      # CASE 4
```

Cases 1 and 2 above behave identically: they run the code contained in the variable `$x`.

Cases 3 and 4 likewise behave in the same way: they run the code `'$x'`, which does nothing but return the value of `$x`. (Case 4 is preferred for visual reasons, but it also has the advantage of compiling at compile-time instead of at run-time).

In the field of self-* systems, both alternatives of usage of the `eval` function could find their place – all four cases presented above. The second pair for exception-handling while the first one for lazy evaluation of self-healed code or its parts, respectively.

5. Examples

The property of introducing new class members on-demand is used to introduce the attribute `rotx` for rotation angle. In the next listing, if the attribute already exists, its value is incremented. The construct triggers both cases of non-existance, which is a kind of basic self-healing strategy. Self-modification part is located in lines 1-2.

```
1 unless ( defined $v -> {'rotx'} )  {
2  $v -> {'rotx'} = 0;
3 }
4 else {
5  $v -> {'rotx'} ++;
6 }
```

The second example is a typical self-modifying routine. The `EDIT ME` tag is used to identify change location. In that place, a new code is being placed until the stop condition is met.

```
1 #!/usr/bin/perl
2 use strict;
3 use warnings;
4
5 my $Gen = 1024; #EDIT ME
6 print $Gen . "\n";
7
```

```
 8 &modify_code;
 9 exec('perl', $0) if $Gen < 500;
10
11 sub modify_code {
12         local $/;
13         local *CODE;
14         open CODE, "+<$0" or die "Can't␣open␣for␣r/w:␣$!";
15         $_ = <CODE>;
16
17         my ($ext) = /\$Gen = (\d+);\s+\#EDIT ME/;
18         $ext*=2;
19         s/\$Gen = \d+;\s+\#EDIT ME/\$Gen = $ext; \#EDIT ME
               /;
20
21         seek CODE, 0, 0;
22         truncate CODE, 0;
23         print CODE;
24 }
```

Our third example of self-adaptive Barycentric interpolation [18] demonstrates the power and weakness of self-modification in one. Self-modification could be distributed into selected lines of code (here line 3 and 7-10). Identification of change locations is getting difficult due to the need of different tags. This kind of annotated code would be very hard to maintain.

```
 1 sub l($) {
 2      my ($x) = @_;
 3      my $l_string = "1*($x-0)*($x-10)*($x-20)*($x-30)"; #
            EDIT LSTRING
 4      return eval $l_string;
 5 }
 6
 7 my $diff = 1; #EDIT DIFF
 8 my $Gen = 40; #EDIT ME
 9 my %points = ( '0' => '0', '10' => '100', '20' => '400', '
      30' => '900',); #EDIT POINTS
10 my @weigths = [-0.1, 0.1, 0]; #EDIT WEIGTHS
11 ...
12 &modify_code if $diff;
13 exec('perl', $0) if $Gen < 10;
14
15 sub modify_code () {
16 ...
17 }
18
19 sub barycentric ($) {
20 ...
21 }
```

6. Conclusion

We focused this paper on the possibilities of self-modification and self-healing implementation using interpreted languages. We selected Perl to demonstrate that this language provides the required constructs and mechanisms to implement a self-modifying system. This system would become a self-healing by a proper definition of self-modification goals. Lower levels of related self-* properties such as exception handling or branching are also present in Perl, but these properties are included in compiled languages as well. The real power is in the ability of flexible self-modification, which makes Perl a perspective candidate for FISH implementation.

FISH, the framework for interpreted self-healing, could utilize Perl::Inline, XS and other modules to include existing systems implemented in different programming languages. Especially the latter will be the main aim of our future research.

References

[1] Laddaga, R., Self Adaptive Software SOL BAA 98-12, *DARPA/ITO* (1998).

[2] Guinea, S., Software Architecture for Adaptive Systems, *Self-Adaptive Software Systems PhD Course with Carlo Ghezzi at Politecnico di Milano* (2013).

[3] Salehie, M., Tahvildari, L., Self-Adaptive Software: Landscape and Research Challenges, *ACM Trans. Autonom. Adapt. Syst.* Vol. 4 No. 2 Article 14 (2009), 42 p.

[4] Parnas, D. L., *On the criteria to be used in decomposing systems into modules* (1971). Computer Science Department. Paper 1980. `http://repository.cmu.edu/compsci/1980`

[5] Shaw, M., Abstraction techniques in modern programming languages, *IEEE Software*, Vol. 1, No. 4, p. 10, (1984)

[6] Perry, D. E., Wolf, A. L., *Software Architecture*. August 1989. The original paper.

[7] Perry, D. E., Wolf, A. L., Foundations for the Study of Software Architecture, *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992)

[8] Kruchten, P.B., The 4+1 view model of architecture, *IEEE Software*, 12 (6), pp.42–50 (1995)

[9] Rozanski, N., Woods, E., *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley (2011)

[10] Gorlick, M.M., Razouk, R.R., Using weaves for software construction and analysis, *Software Engineering, Proceedings., 13th International Conference on*, pp.23,34, 13-16 May 1991 (1991)

[11] Taylor, R.N., Medvidovic, N., Anderson, K. M., Whitehead Jr., E.J., Robbins, J. E., Nies, K. A., Oreizy, P., Dubrow, D.L., A component- and message-based architectural style for GUI software, *Software Engineering, IEEE Transactions on*, vol.22, no.6, pp.390,406, (Jun 1996)

[12] Magee, J., Dulay, N., Kramer, J., Regis: a constructive development environment for distributed programs. *Distributed Systems Engineering* 1(5): 304-312 (1994)

[13] Oreizy, P., Medvidovic, N., Taylor, R. N., Architecture-Based Runtime Software Evolution. *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, pages 177-186, Kyoto, Japan, April 19-25, 1998.

[14] Kramer, J., Magee, J., Self-Managed Systems: an Architectural Challenge. *FOSE 2007*: 259-268

[15] Baresi, L., Guinea, S., Saeedi, P., Achieving Self-adaptation through Dynamic Group Management. *Assurances for Self-Adaptive Systems 2013*: 214-239

[16] Miller, B., *The autonomic computing edge: Can you CHOP up autonomic computing?*, IBM, 2008.

[17] *Perl Programming Documentation.* [Online]. Available: `http://perldoc.perl.org/`

[18] Berrut, J.-P., Trefethen, L. N., Barycentric Lagrange Interpolation, *SIAM Review*, Vol. 46, No. 3, pp. 501–517 (2004)