Embedding Adaptivity in Software Systems using the ECSELR framework [*]

Kwaku Yeboah-Antwi
INRIA
Rennes, France
kwaku.yeboah-antwi@inria.fr

Benoit Baudry
INRIA
Rennes, France
benoit.baudry@inria.fr

ABSTRACT

We present, ECSELR, an ecologically-inspired approach to software evolution that allows for environmentally driven evolution in extant software systems at runtime without relying n any offline components or management. ECSELR embeds adaption snd evolution inside the target software system allowing such systems to transform themselves via darwinian evolutionary mechanisms and adapt in a self contained manner. This allows such software system to benefit from the useful byproducts of evolution like adaptivity, bio-diversity without having to worry about problems involved in engineering and maintaining such properties. ECSELR allows the software systems to address changing environments at runtime enabling benefits like mitigation of attacks, memory-optimization among others while avoiding time-consuming and costly maintenance and downtime. ECSELR differs from existing work in that, 1) adaption is embedded in the target system, 2) evolution and adaptation happens online(ie in-situ at runtime) and 3) ECSELR is able to embed adaptation inside systems that have been already started and are in the midst of execution. We demonstrate the use of ECSELR and present results on an experiments in using the ECSELR framework to slim a software system.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous ; D.2 [**Software Engineering**]: Management—*Life cycle, Software configuration management*; D.2 [**Software Engineering**]: Search-based Software Engineering—*software evolution*

General Terms

Theory, Software Engineering, Self-Adaptive Systems, Genetic Improvement

[*]This work is supported by the EU FP7-ICT-2011-9 No. 600654 DIVERSIFY project

Keywords

Software Evolution, Diversification, Software Slimming, Search Based Software Engineering, Genetic Improvement, Self Adaptive Software Engineering

1. INTRODUCTION

The field of software engineering has been straining to deal with the exponential growth of software systems. As computer systems get more and more complex, maintaining them requires a huge amount of human effort. Software Engineering may soon hit a "*complexity wall*" where our efforts will not be able to scale up to highly complex software systems [2, 6, 14]. This has led to the rise in the amount of research being done in automating software design, maintenance, with software engineers increasingly seeking to design software systems that autonomously evolve and adapt. Designing such self-optimising adaptive systems has been described as one of the great challenges of the software engineering field [4]. The holy grail sought at the end of this challenge by software engineers would be a system that is self-contained and is be able to self-modify, evolving itself autonomously at runtime with minimal or no human input thereby allowing it to self- optimise and self-adapt to its environment. One of the most promising approaches to this challenge has been focused around the agenda of Dynamic Adaptive Search Based Software Engineering (DASBSE). Research in DASBSE focuses on embedding adaptivity inside software systems to allow self-contained evolution and adaption in these software systems. This is referred to as *dynamic adaptivity* [4, 10]. Implementation of dynamic adaptivity in a software system will result in the system being able to autonomously evolve and adapt *in-vivo* and *in-situ*,(ie. in place and online at runtime) without relying on external aids from the programmer [5]. The mechanisms for evolution and adaptation would be in essence, self-contained inside the application. Such systems would be able to autonomously perform actions like, mitigation of virii attacks, resources-optimization among others while avoiding time-consuming and costly maintenance and downtime and also reducing the amount manpower needed to supervise such activities as compared to current systems.

Currently, one of the main focuses of DASBSE has been Genetic Improvement(GI). Under Genetic Improvement(GI), heuristic search based evolutionary algorithms/systems are applied to extant software programs directly transforming them [4]. The instructions making up the software systems are treated as the donor genetic material to be evolved thereby *genetically improving* the software system. Research into different GI methodologies is currently very young with most work in the field currently falling short of full embedded adaptivity. Current research in DASBSE in general falls short of the fully conquering "dynamic adaptivity". Most approaches so far still rely on offline adaptation and offline evolution components and steps. Adaptation is not fully embedded inside the software system; adaptation happening outside the software system being evolved. Adaptation also usually tackles either the non-functional or the functional properties of the system and but rarely both. There exists a need for new solutions.

We present ECSELR, a new ecologically-inspired dynamic software evolution framework model that aims to help fill the gap in methodologies that enable true dynamic adaptivity. ECSELR is a java based platform that allows extant software systems to evolve themselves by autonomously self-modifying their code at runtime, transforming and evolving the target software system in-vivo and in-situ via darwinian evolutionary mechanisms. ECSELR takes as input, the system process of the running program and inserts itself into the process, imbuing the software system with the ability to monitor its actions and its environment during the program lifecycle. Based on these observations, the system stochastically effects actions on its basic computational units(e.g. bytecode) transforming the structure and function of the software system and thereby evolving it and allowing it to dynamically self-adapt to its environment. The environment of a software system is defined as the state of the software system and the interactions between the components of the software system. Evolution and self-adaptation is embedded in the system, with everything happening *online* (in-vivo and in-situ) without any external dependencies and/or any human input. This makes the whole adaptation process self-contained and fully dynamic/autonomous. A software system running the ECSELR framework is able to evolve and adapt autonomously online without any pauses or breaks in system execution, thereby maintaining expected system availability through the software systems lifecycle. ECSELR can also be easily coupled with other frameworks to give it new capabilities among others.

We demonstrate the ease of use of the ECSELR platform by using it to reduce the resource footprint of an existing software system. We show that, no domain knowledge is needed to operate ECSELR and and that, it can be used by both software programmers and the software endusers.

We believe that ECSELR is a major novel contribution to DASBSE and GI and provides another avenue for future research into the uses of dynamic adaptivity.

The paper is structured as follows. In Section 2 we present ECSELR, describing its components and its mode of utility. In Section 2.1, we present one(1) use scenario for ECSELR. In Section 3, we describe our experiment in utilising ECSELR to evolve a system to optimise process resources and change its attack surface, demonstrating the use of ECSELR and presenting and discussing the results. In Section 4, we present previous work and distinguish between ECSELR and these previous methodologies and frameworks. In Section 5, we reiterate our contribution and expand on possible avenues of future work.

2. ECSELR

To implement dynamic adaptivity in a target complex software system, the software system is first given the ability to monitor and observe itself, its actions and environment at runtime. This allows the system to be self aware [8] about its functioning and its environments and the effects of the environment on itself. Subsequently, the system is given the mechanisms to effect stochastic transformations upon itself. The software system applies the transformation to itself based on its relationship to its environment and also, its understanding of its own state garnered from its ability to monitor itself and its environment. Just like in natural evolution, these transformatory changes are solely based on environmental pressure and the stochasticity of the environment.

The ECSELR framework provides all the necessary components needed for this to happen. The framework is comprised of four main components, an evolution driver called *EvoAgent*, an evolution utility called *EvoDaemon*, transformation utilities called *Evolution Operators* and transformation guides called *Evolution Strategies*.

Given a running target software system, EvoAgent attaches itself to the process, living in the same native memory address space as the process. The attachment can be done natively through the OS level hooks or via java language mechanisms. EvoAgent is comprised of a series of native system functions and data structures that allow it to monitor and keep track of the state of the software system and its environment at any point in time. Upon attachment, EvoAgent probes and sets up its set of functions to profile the basic computational blocks in the software system and record relevant information about the blocks. These blocks refer to the lowest level of instructions that comprise the system (ie. a set of bytecode instructions in the JVM). This gives the ECSELR platform the ability to monitor and observe the system and its actions and interactions with its environments. ECSELR observes when blocks are called/executed, what inputs the blocks are given and what they act on and when they complete execution/exit. Since blocks correspond to methods and fields in the JVM, ECSELR is thusly able to observe the execution flow of the software system. ECSELR partitions the software system execution lifecycle into discrete time periods, called epochs, representing a generation in an evolutionary timeline. At the end of each epoch, EvoAgent appraises itself of the state changes in the lifecycle of the software system and its environment and then invokes EvoDaemon to evolve the system based on the changes.

EvoDaemon contains an engine for rewriting java bytecode according to a set of given instructions/strategies. EvoDaemon evolves the software system according to a set of evolutionary strategies that are supplied by EvoAgent.

The evolutionary strategies are stochastic conditions that are created based on the history of observed system interaction. They decide which parts of the system are transformed and how they're transformed. An evolutionary strategy probabilistically selects which parts of the software system it wants to evolve and an evolutionary operator is then used to effect this evolution. An example evolutionary strategy is one which probabilistically decides a method to remove based on observed frequency of method usage.

The evolutionary operators are defined as the set of generic transformatory mechanisms that modify the software blocks, transforming them from one state to another and in essence evolving the system over the course of the program lifecycle. They are analogous to the evolutionary mechanisms/operators that act on the genome of biological organisms and drive/effect evolution. They are distinguished by their effect on the system blocks and structure of the system as a whole. The operators are grouped into four basic types:

Amorphic Operators which act by removing blocks from the system.

Geomorphic Operators which act by combining blocks in numerous ways.

Translocation Operators which change the structure and execution flow of the program by moving blocks around.

Hypomorphic Operators which change the availability of blocks, suppressing and inciting different functionalities of the system.

Listed below is the set of evolutionary operators that are currently implemented in ECSELR.

1. *Merge Operator:* It takes two software blocks/methods, A and B, naively appends block B to block A and replaces the original block A with this new version of Block A. Block B is then deleted from the software system.
2. *Deletion Operator:* This operator takes as input, a method, and deletes the byte code of the method from the software system, thereby removing the method from the system.
3. *Inversion Operator:* This operator takes as inputs, locations for two software blocks (eg. blocks in a method) and invertes their positions, placing the first block in the second's position and the second in the first's position. This changes the execution flow of the software system.
4. *Duplication Operator:* This operator takes as input, a software block and a location. It duplicates the input software block and inserts the generated copy of block at the specified location.
5. *Loop Perforation Operator:* This operator takes as input, a block of code that runs in a loop and perforates the loop, reducing the number of times the block is iterated.
6. *PassThrough Operator:* This operator returns an unmodified clone of a given block.

Platforms that enhance software systems by providing a measure of adaptability can be described and taxonomized according to 4 facets/properties. These properties characterise the level of adaptability and evolvability in the enhanced software systems, mapping adaption to evolution and describing the degree to which adaptation is implemented, governed and reasoned about via evolution. These properties are based on similar properties widely used by the self-adaptive systems community [11] . The properties are presented as follows, emphasising how specifically they can be used to describe self-adaptation:

• **Temporality:** Describes the temporal property of the adaptivity, specifying *when* the adaptivity changes are applied. The temporality of self-adaptivity is either Online(ie. the changes happen at runtime) or Offline(ie. changes happens before runtime).
The ECSELR platform has an online temporality. Evolution and adaption happens online at runtime.

• **Locality:** Describes the locality of software evolution and adaptivity, showing *where* in the system evolution happens to lead to adaptivity. The options encompass localities like source code, binary or an AST/model of the system.
The ECSELR platform acts directly upon the binary. It modifies and transforms the execution instructions ind the binary.

• **Artifacity:** Describes the scope of evolution, showing *what about/in* the system is being evolved and benefitting from adaptation. (ie. Functional Properties vs Non-Functional Properties).
ECSELR is able to evolve and adapt both functional and non-functional properties of the target system. It is able to change system functionality in addition changing the on-functional aspects of system design.

• **Dynamicity:** Describes how dynamic evolution is, showing how automated evolution is. It quantifies the degree of human intervention in evolution. A system with full dynamicity evolves and adapts with no human input while a system with no dynamicity has its evolution and adaptation managed by a human programmer(ie. the hand of god).
ECSELR is able to autonomously evolve and adapt the target application with minimal human intervention needed or required. All though the framework supports input from the human programmer, its does not require this and is able to function without any help. ECSELR can be said to therefore have full dynamicity.

2.1 ECSELR Use Case

The ECSELR platform can be used to achieve tackle many problems that benefit from dynamic adaptivity. In this section, we present one(1) such use case.

2.1.1 Software Slimming

Software Slimming is the removal of unused and/or underutilised parts of software systems at runtime for various purposes.

A library is always built to serve a large number of use cases and therefore contains a wide range of components for each of these various use cases. The engineers of the library can rarely fully anticipate in which combinations the different components will be used and so ship the library with its full complement of components. Since the rate of utilisation of any of the components depend a lot on the application that is built on top of it, each software engineer is responsible for narrowing down the sets of components that are necessary for their application and discarding the rest. Most software engineers usually just bundle the whole library with their application. For any given application, a number of these components from the library will certainly never be used at runtime and some of them can be detected statically. However, it is not always possible or efficient to detect before runtime, all components that will never be used due to a number of reasons that include uncertainty about the projected use profile, lack of access to the source of the system, lack of familiarity with the system, lack of time, among others. This results in software systems that are distributed with superfluous components of libraries that are rarely used. These components take up space and use resources and represent unknown possible attack vectors.

Since ECSELR is able to observe the interactions of the components in the software system, it is able to identify unused and superfluous packages and instructions and remove them, thereby reducing the space and memory resources needed for the application and also removing possible attack entry points in the program. These software systems are subsequently considered to be *slimmed* based on the system use profile. ECSELR is also able to slim software services according to other different criteria such as power consumption, etc.

Another example of software slimming would be a software system that is going to be used in a low-resource environment where the system engineer has no pre-conceived knowledge about which part of the library is going to be more utilised. When the software system is loaded and is being executed with the ECSELR framework embedded in it, ECSELR is able to remove unused components reducing the systems resource usage and optimizing the software system.

Other concrete examples of software slimming can be found in the literature here in [9] and in [7].

3. EXPERIMENTS

In order to test how capable ECSELR is at Software Slimming, we designed a test platform, jEvoTester, to simulate a large java software system, and run ECSELR on the jEvoTester platform. We subsequently present this platform in the next section.

3.1 Experimental Platform

jEvoTester was designed to micmic and reproduce the interdependent relations and interactions present in software systems. It simulates a software system platform that offers services to a set of applications. It can also be considered as a large library or framework that offers a wide range of services to applications that need it.

jEvoTester is composed of 360 classes each offering 1000 methods. jEvoTester provides these methods and classes as services to applications.

To simulate the interactions between applications and the software system/library and also in between the components of the system, we sampled the WorldCup98 website requests dataset [1] [1]. This data captures the interactions between users of the World Cup '98 web site. We equate the visits by users in the dataset to interactions between the classes and methods and the apps that request them, with each visit being comprised of requests to objects that in turn depend on other objects. Each user's visit in the dataset consists of a request for a combination of multiple objects. Each unique user is equated to an application and the objects requested are equated to methods.

An application therefore requests a combination of methods that it requires, with the methods considered to be interdependent on each other since they all need to be available for the request to succeed. If a method is unavailable, the application's request fails. The applications were implemented in jEvoTester as agents in a multi-agent system, each with a non-static collection of methods required over the course of its lifetime. All application requests are performed asynchronously and independently of other applications.

[1]http://ita.ee.lbl.gov/html/contrib/WorldCup.html

Table 1: Evolutionary Operators in jEvoTester

Operator Name	Function
Merge Methods Operator	Merges two given methods
Method Deletion Operator	Deletes a method.
Passthrough Operator	Return an unmodified clone of the object

During each run of jEvoTester, ECSELR is inserted into jEvoTester in order to evolve and adapt it. Table 1 lists the subset of operators used by ECSELR.

jEvoTester is unaware of the applications and also works asynchronously and independent of system functioning(i.e. ECSELR does not pause or half any system instructions whilst evolving the system).

3.2 Results

We chose to utilise the jEvoTester platform to test ECSELR on the Software Slimming use case detailed in Section 2.1.1. We tested whether ECSELR would be able to slim jEvoTester during jEvoTester execution. Our test harness was run 200 times with the same interaction data in order to verify the significance of our results.

For each test, the jEvoTester platform was executed. During the course of execution, the ECSELR platform was inserted into jEvoTester. Each application recorded the response from jEvoTester to its requests, noting if jEvoTester was able to provide a method being requested or not. In jEvoTester, when a deleted method is called, a runtime exception is generated which is caught and returns a negative answer to the application. In order to test how well ECSELR performs at software slimming, the jEvoTester platform was designed to have a whole complement of methods, some of which are utilised by applications and some of which aren't. The goal of the experiment was to test if ECSELR would be able to remove a percentage of the unused methods while ignoring methods that were being utilised. The rate of usage of the methods were not static with some methods used more frequently than others during different points in jEvoTester's lifetime.

Our results are aggregated in Table 2.

For each test, there were 27,500 applications calling 4,907 methods. These methods were called a total of 1,522,111 times. The applications called these methods in 68,084 different configurations. The most popular method combination was called 15,517 times. It was called by 10,071 unique applications.

355,093 out of 360,000 methods were never called during the duration of our experiments. ECSELR was able to remove an average of 140,206 methods during the course of each run. On average, 57 of these methods were requested by applications after their removal resulting in errors.

3.3 Discussion

We used our results to answer three key questions which are elucidated below.

• **RQ1:** ***Can ECSELR slim jEvoTester during its lifecycle?***
Our results confirmed that, yes, ECSELR is able to suc-

Table 2: jEvoTester Data

Description	Value
Classes	360
methods per class	1000
Total Methods	360000
Unique Applications	27500
Methods Called	4907
Number of method calls	1522111
Total method combinations called	68084
Uncalled Methods	355093
Percentage of methods never called	98%
Average Number of Methods slimmed	140206
Percentage of uncalled methods slimmed	39.5%
Percentage of total methods slimmed	38.9%
Methods called after removal	57
Percentage of methods removed that were called after removal	0.04%
Number of experiments	200

cessfully slim jEvoTester. ECSELR was able to remove an average of 140206 methods in the jEvoTester software system during the runs.

• **RQ2:** ***How successful is ECSELR at slimming a software system?***
Our result show that ECSELR was able to remove approximately 39% of all unused methods. This shrunk the application reducing its footprint (ie. amount of memory needed by the JVM to store the application byte code, class data, etc).

• **RQ3:** ***Does software slimming through ECSELR negatively affect the availability and usage of the software system?***
Analysis of our data showed that, during each run, approximately 57 methods that were required by applications were slimmed and thus made unavailable to the applications requiring them. A maximum of 4 applications were affected by this problem. This represents an application request failure rate of 0.01% due to ECSELR.

Our results show and confirm that, ECSELR is able to autonomously slim software systems with slimming controlled and mediated by the state of the software system during its lifecycle.

4. PREVIOUS WORK

Schulte, Forrest and Weimer in [13] and with DiLorenzo in [12] presented a method of evolving assembly code and binary code to effect automatic software repair. Their method evolved an AST of the extant program, converting that AST back to a binary executable. Goues, Nguyen, Forrest and Weimer in [3] presented GenProg, a platform for evolving C source code to automatically fix bugs. Unlike ECSELR, these approaches do not embed evolution and adaption in the software systems being evolved. Therefore evolution and adaptation doesn't happen online. Also, unlike ECSELR, these approaches are unable to evolve a software system that already running.

Harman et al in [5] presented an outline of an approach called Online Genetic Improvement(OGI), a methodology in which an existing program is improved online by modifying its existing code using evolutionary search mechanisms. Their approach consists of an online and offline phase where program execution is monitored online, this data is sent to an offline component to learn from and expose implicit parameters that are sensitive to operational properties of interest. These parameters are optimised and then deployed online. Swan et al in [15], presented Gen-O-Fix, a framework which consists of an agent that resides outside the binary and proxies the instructions to the binaries serving as the evolution agent. These two approaches differ from ECSELR in that, adaptivity isn't not embedded inside the application. OGI also relies on an offline phase unlike ECSELR. Although these frameworks are able to evolve running software systems, they require the target software system to be started with the framework. ECSELR on the other hand is able to target software systems that have already started execution.

5. CONCLUSION

In this paper, we presented the ECSELR platform, platform for achieving dynamic adaptivity, i.e.. embedding adaptivity in a software system. We presented Software Slimming as a use case for such a system and showed that, ECSELR is able to perform admirably well in this use case. ECSELR is demonstrated to be able to evolve and adapt already executing extant software systems with no human input. ECSELR shows that it is infact possible to embed adaptivity in software systems.

Although we describe the ECSELR framework in this paper, we chose not to fully expand on all aspects of the frameworks design. We aim to present a more detailed exposition and study about the design decisions behind ECSELR in a subsequent paper.

We hope that, with this paper, ECSELR will be seen as a platform for more experiments and research into dynamic adaptivity.

6. ACKNOWLEDGMENTS

The authors of this paper would like to note that, work done for this paper is supported by the EU FP7-ICT-2011-9 No. 600654 DIVERSIFY project. More details can be found at http://diversify-project.eu/

7. REFERENCES

[1] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical report, IEEE Network, 1999.

[2] P. Feiler, K. Sullivan, K. Wallnau, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, and D. Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future.* Software Engineering Institute, Carnegie Mellon University, June 2006.

[3] C. L. Goues, T. Nguyen, S. Forrest, S. Member, and W. Weimer. Genprog: A generic method for automatic software repair.

[4] M. Harman, E. Burke, J. Clark, and X. Yao. Dynamic adaptive search based software engineering. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 1–8, New York, NY, USA, 2012. ACM.

[5] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. Genetic improvement for adaptive software engineering (keynote). In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 1–4, New York, NY, USA, 2014. ACM.

[6] S. Herold, H. Klus, D. Niebuhr, and A. Rausch. Engineering of it ecosystems: Design of ultra-large-scale software-intensive systems. In *Proceedings of the 2Nd International Workshop on Ultra-large-scale Software-intensive Systems*, ULSSIS '08, pages 49–52, New York, NY, USA, 2008. ACM.

[7] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*. Accepted.

[8] M. Mowbray and A. Bronstein. What kind of self-aware systems does the grid need? *HP Laboratories*, 2005.

[9] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. Garcia-Sanchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 137–149, Granada, Spain, 23-25 Apr. 2014. Springer.

[10] C. Raibulet and L. Masciadri. Evaluation of dynamic adaptivity through metrics: an achievable target? In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 341–344. IEEE, 2009.

[11] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[12] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 317–328, New York, NY, USA, 2013. ACM.

[13] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[14] M. Sipper. Ubiquity symposium: Evolutionary computation and the processes of life: Darwinian software engineering: The short term, the middle ground, and the long haul. *Ubiquity*, 2012(December):2:1–2:6, Dec. 2012.

[15] J. Swan, M. Epitropakis, and J. R. Woodward. Gen-o-fix: An embeddable framework for dynamic adaptive genetic improvement programming. Technical report, Technical Report January, Department of Computing Science and Mathematics, University of Stirling, Stirling, UK, 2014.