THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Biomimetic Software Engineering Techniques for Dependability

Robert Feldt

Department of Computer Engineering
School of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, SWEDEN, 2002

Biomimetic Software Engineering Techniques for Dependability
Robert Feldt
ISBN 91-7291-241-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 1923
ISSN 0346-718X

School of Computer Science and Engineering
Chalmers University of Technology
Technical Report 9D

Department of Computer Engineering
School of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Tel. +46(0)31-7721000
www.ce.chalmers.se

Author email: feldt@ce.chalmers.se

Vasa Bokbinderi
Göteborg, Sweden, 2002

for Mirjana

my love, light and passion…

Biomimetic Software Engineering Techniques for Dependability

Robert Feldt

Department of Computer Engineering, Chalmers University of Technology

Abstract

The powerful information processing capabilities of computers have made them an indispensable part of our modern societies. As we become more reliant on computers and want them to handle more critical and difficult tasks it becomes important that we can depend on the software that controls them. Methods that help ensure software dependability is thus of utmost importance.

While we struggle to keep our software dependable despite its increasing complexity, even the smallest biological system in nature shows features of dependability. This thesis applies ideas from and algorithms modeled after biological systems in the research for and development of dependable software.

Based on a theory of software development focusing on the internal models of the developer and how to support their refinement we present a design for an interactive software development workbench where a biomimetic system searches for test sequences. A prototype of the workbench has been implemented and evaluated in a case study. It showed that the system successfully finds tests that show faults in both the software and its specification. Like biological systems in nature exploits a niche in the environment the biomimetic search system exploits the behavior of the software being developed.

In another study we applied genetic programming to evolve programs for an embedded control system. Although the procedure did not show much potential for use in real fault-tolerant software, the program variants could be used to visualize the difficulty of the problem domain, explore the effects of design decisions and trade off requirements.

Taken together the works in this thesis support the claim that biomimetic algorithms can be used to explore requirements, design and test spaces in early software engineering phases and thus help in building dependable software.

Keywords: biomimetic algorithms, software testing, automated testing, software development workbench, evolutionary computation, genetic programming, dependability, software engineering, design exploration, software visualization

List of Papers

This thesis is based on the following papers:

1. Robert Feldt. *A Theory of Software Development,* Technical Report no. 02-22, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002.

2. Robert Feldt. *An Interactive Software Development Workbench based on Biomimetic Algorithms,* Technical Report no. 02-16, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002, *a condensed version of this report has been submitted to* IEEE Transactions on Evolutionary Computation.

3. Robert Feldt. *An Experiment on Using Genetic Programming to Develop Multiple Diverse Software Variants,* Technical Report no. 98-13, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, September 1998.

4. Robert Feldt. *Genetic Programming as an Explorative Tool in Early Software Development Phases,* Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering, pp. 11-21, Limerick, Ireland, 12th-14th April, 1999.

5. Robert Feldt. *Forcing Software Diversity by Making Diverse Design Decisions - an Experimental Investigation,* Technical Report no. 98-46, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, December 1998.

6. Robert Feldt and Peter Nordin. *Using Factorial Experiments to Evaluate the Effect of Genetic Programming Parameters,* In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, Genetic Programming, Proceedings of EuroGP'2000 , volume 1802 of LNCS , pages 271-282, Edinburgh, 15-16 April 2000. Springer-Verlag.

7. Robert Feldt, Michael O'Neill, Conor Ryan, Peter Nordin, and William B. Langdon. *GP-Beagle: A Benchmarking Problem Repository for the Genetic Programming Community,* In Late Breaking Papers at the 2000 Genetic and Evolutionary

Computation Conference, pp. 90-97, Las Vegas, Nevada, USA, July 2000.

Paper 3 includes and extends on the two previously published papers:

- Robert Feldt. *Generating Multiple Diverse Software Versions with Genetic Programming - an Experimental Study*, IEE Proceedings - Software, vol. 145, issue 6, pp. 228-236, December 1998. Special issue on Dependable Computing Systems.

- Robert Feldt. *Generating Multiple Diverse Software Versions with Genetic Programming*, Proceedings of the 24th EUROMICRO Conference, Workshop on Dependable Computing Systems, pp. 387-396, Västerås, Sweden, August 1998.

One component of the WiseR system described in paper 2 was also described in the book:

- Michael Neumann, Robert Feldt, Lyle Johnson, Jonothon Ortiz, Jason Wong and Stephen Legrand. *Ruby Developer's Guide*. Syngress Publishing, Rockland, MA, 2002.

Acknowledgements

I want to thank my advisor professor Jan Torin for his support during these years. Jan never stopped believing in me and allowed me to explore ideas that was not in the mainstream. I'm very grateful for his courage and support.

During most of my years at Chalmers I've been part of the Software Fault Tolerance group headed by Håkan Edler. Co-advisors during the first years were Dr. Jörgen Christmansson and Dr. Marcus Rimén. I thank you all for taking the time to listen to me and for giving advice and encouragement.

I'm also deeply indebted to my colleague and former room-mate Dr. Martin Hiller. Thank you, Martin for stimulating discussions, your comments and advice during these years.

I want to thank all who have taken part in my research advisory committees during these years. In addition to Jan Torin and Håkan Edler, Dr. Jan Jonsson and Dr. Fredrik Dahlgren helped out during the first years. Later on, Professor Bo Bergman joined and had interesting ideas on the border between biology and software.

The rest of the department and former colleagues also deserves many thanks for creating a stimulating atmosphere.

A special credit to Dr. Bengt Månsson for awakening my interest in mathematics and giving me the first glimpses of its inner beauty.

My wife, my daughter and I are fortunate enough to have a large extended family living close nearby. Its hard to find words enough to thank Saara, Srecko and Matilda for helping us out when things get rough and for all the fun we have together. Maja and Mathias are also an important part of this and their 'new kid on the block', Eliot, is a great addition. Thanks for all the fun; we love you all!

I would not be writing this today without the continous support of my fantastic parents Dan and Elisabet. Your love and devotion is amazing! Thanks for putting so much effort into the growth and development of your children. I would also like to thank Helene, the best sister and friend anyone can possibly ask for.

I send warm feelings and love to my little daughter Ebba. She is really the most fantastic little creature I'm part of creating. But Ebba also recreates me and its both stimulating, fun and a little scary. I also want to thank you for more directly showing the explorative and fault-identifying power of a small, biological system[1]. I hope you will get a great and fullfilling life and become happy. I promise that I will do

[1]Ouch, not dad's camera! Watch that wase… :-)

everything I can to make it so.

Finally, my life companion Mirjana. This has been a long journey with both ups and downs. I want to thank you for your creativity, laughs, cries, warm embrace and funny ways. Without you I’m nothing. I love you.

Robert Feldt

Göteborg, November 2002

’The road to wisdom? Well it’s plain
and simple to express:
err
and err
and err again
but less
and less
and less and less…’

A grook by Piet Hein,
Danish mathematician and multi-artist (1905-1996)

Contents

1. Introduction

Software that controls computers is an indispensable part of our modern societies. It permeates our everyday life and we depend on it for tasks ranging from game playing, over the handling of economic transactions to the control of life-support machinery. When software contains faults and fail to provide the service we require it is annoying, costly or even fatal. We need ways to develop and test software so that we can depend on it.

Developers and researchers have stepped up to this task by creating numerous techniques to increase software dependability. We try to prevent faults by reviewing the software, using development processes that minimize defects and applying formal methods to prove that parts of it are correct. We try to remove remaining faults by verifying and validating it. We try to tolerate the remaining faults with redundant units to shield us from sub-system failures. Taken together our increased control over the development process and of the resulting software has taken us a long way.

Despite much progress both on the methods and tools we use we still find it hard to develop software that works without trouble. Partly the reason is that our expectations have risen. We want our computers to do more, do it faster and for a lower price. The result is more complex software with many, inter-related components that affect each other and are used in situations the developers hadn't foreseen. But the reason we can't yet deliver the holy grail of competent, complex, yet fault-free software might still be in the way we think and the methods we use.

While we struggle to develop dependable software even the smallest biological system, be it a single-cellular amoeba, an organism, the human immune system or a whole ecosystem, show signs of robustness and continue to work in spite of faults and failures. Furthermore, biological systems have developed in a dynamic environment with ever changing conditions. The impetus for this work is that there must be something we can learn from biological systems about dependability. If we can mimic what nature does maybe we can extend our arsenal of techniques for acheiving dependable software?

The idea of mimicing nature is not new; humanity has been doing that for ages. Recent examples can be found in material science where researchers have learnt to build stronger and lighter materials by studying and mimicing spiders webs and seashells. In computer science and engineering several of the pioneers had ideas on how to take after nature. John von Neumann, considered as the father of the modern computer, tried to model self-reproduction in his cellular automata. The increased speed of computers have allowed them to simulate more complex models from nature. In recent years we have for example seen algorithms that model the evolutionary process, the human immune system and the neurons in our brains. Can we make use of these algorithms in our quest for software dependability?

1.1. Overall objective and research questions

The overall objective of the work presented in this thesis was to explore how techniques inspired by biological systems can help develop dependable software. More specifically we have adressed the following research questions:

- Q1.1 Can biological systems be called dependable in a technical, dependability engineering sense?

One biomimetic algorithm is genetic programming which searches for computer programs using an algorithm inspired by the evolutionary process in nature. Since genetic programming is a form of automatic programming, ie. automatically finding a program that fulfills a specification, it is potentially very interesting to the field of software engineering.

- Q2.1 Can genetic programming be used to develop software components for fault tolerant software systems?

In addition to using biomimetic algorithms to develop software we would like to know if we can use them to find tests for testing human-developed software:

- Q3.1 Can biomimetic algorithms be used to test software?
- Q3.2 Can the tests be meaningful to a developer so that he can motivate them?

1.2. Main contributions

The contributions put forward in this thesis are:

- **A design of an interactive software development workbench**. Based on a theory of software development focusing on the internal models of the developer we have designed a workbench that supports the simultaneous refinement of a program and its specification. It is called WISE, standing for Workbench for Interactive Software Engineering and applies biomimetic algorithms to find tests that show novel features of the software being developed. The system is unique by searching for test templates from which multiple tests can then be generated.

It is also unqiue by using the feedback from the developer to guide the search process.

- **A prototype of the software development workbench**. The WISE design has been implemented in a prototype called WiseR, WISE for Ruby, for interactive refinement of specifications, programs and test knowledge in the object-oriented programming language Ruby. The focus in the prototype is the module for finding tests, called WiseR-Tests. A case study on the prototype show that it finds test sequences that identifies problems in the specification and implementation of a piece of software.

- **A procedure for developing diverse software variants with genetic programming**. Genetic programming was used to develop multiple software variants for one and the same problem. By varying parameters to the genetic programming systems the variants are forced to be different. The procedure was evaluated empirically by developing 80 variants of a software controller. Although failure diversity was acheived it was limited for the top performing programs. The procedure was also used to assess the validity of assumptions made in an important theorem for N-version programming.

- **A proposal to use genetic programing for problem visualization in early software development phases**. We propose that genetic programming can be used to explore the difficulty of different input data, determine the effects of different requirements and identify design trade-offs inherent in the problem. This can be used early in a software development project to reduce the risc of later phases.

- **A technique for evaluating and tuning parameters to evolutionary algorithms**. We propose to use statistical techniques for design and analysis of experiments to tune the parameters to GP and EC algorithms.

1.3. Thesis structure

The remainder of this thesis is divided into 5 chapters of this thesis introduction and three parts with a total of 7 appended papers.

Thesis introduction:

- **Chapter 2** gives more background on the two main fields related to this work: software dependability and biological systems,

- **Chapter 3** reviews the related work in Biomimetic Dependability,

- **Chapter 4** summarizes the appended papers,
- **Chapter 5** discusses the results,
- **Chapter 6** concludes and points to the future.

There are three parts of appended papers:

- **Part 1** includes two papers on how to support software developers with an interactive software development workbench that uses a biomimetic search system for finding novel tests that highlight important features of the software being developed.
- **Part 2** includes three papers that use genetic programming to evolve software controllers for an aircraft arrestment system.
- **Part 3** includes two papers that propose better experimental methods and test data so that evolutionary computation can be used and evaluated in better ways.

2. Software Dependability and Biological Systems

This chapther introduces the basic concepts and terminology used to describe software dependability. It also defines what we mean with biological systems and note some of their characteristics. It then assesses if and how the dependability concepts and terminology is applicable to biological systems.

2.1. Software Dependability

Dependability is an ’umbrella’ concept introduced to unify the terminology for reliability, safety and security of computer systems. It is the top-level concept defined as *the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers* [20, 5]. In this context, service is the behavior of the system as viewed by the users, which can be humans or other systems.

An assumption in the definitions of the basic dependability concepts is that a specification describes the correct service of the system. When the system violates the specification it has failed. The ultimate cause of a failure is a fault (or several cooperating faults) somewhere in the system. When a fault is activated it leads to an error in the state in some part of the system. It is not until the error reaches the system boundary and affects the service that the failure is said to occur.

Note that since the notion of a failure is not meaningful without a system specification we are vulnerable to faults in the specification. When the service delivered by the system does not comply with the specification it might either indicate an error in the system or in the specification.

There are many different faults that occur in computer systems. A basic classification scheme for faults can be found in [5] and classifies them according to what caused them (physical cause or caused by human), the intent behind (accidental or deliberate), the phase when they were created (development, production or operation), their domain (physical or informational), and their persistence (permanent or transient).

In this thesis we focus on accidental faults caused by humans, ie. we are not concerned with acheiving security against deliberate faults. Since we focus on software the fault domain might be called informational. It is not clear how to differentiate between developmental and production faults for software. When we need to do such distinctions we will use a more detailed model based on traditional phases of a software development process with requirements analysis, design, coding, testing, operation and maintenance. Faults in software are permanent even though the errors they cause might be transient.

When assessing the dependability of a computer system we can use a number of different attributes. Originally the attributes were:[1]

- **availability** - probability that the system is ready to deliver the correct service, ie. its readiness for correct service
- **reliability** - probability that the system will continue to function correctly for a certain amount of time, ie. its continuity of correct service
- **safety** - absence of catastrophic consequences on the user and the environment

Here, safety is really an extension to reliability where we consider both the correct functioning and non-catastrophic failures of a system. In addition to the specification needed to judge the correct service, safety pre-supposes a safety specification, stating the catastrophic failures. In later years maintainability has been added as a dependability attribute. A system is considered maintainable if it supports repairs and modifications [5]. This is important since it will indirectly affect the other attributes: if a system is reliable but not maintainable it will be difficult to retain the reliability when changing the system. Changes are frequent in the real world.

Recently, the attribute of survivability has also been proposed as a dependability attribute [17]. It is interesting since it introduces different levels at which the system can conform to the specification. For availability and reliability the conformance to the specification is judged on a binary scale: the system either delivers or it does not deliver the correct service. A survivable system, on the other hand, has the ability to continue to provide (a potentially degraded or different) service even if it has been damaged. This can be captured in the specifications by ordering the functions (sub-services) that the system should supply from most to least preferred and state the probability with which each function shall be provided.

A concept related to the dependability concepts that is not covered by them is robustness. The reason for this is probably that it has not been given a single meaning but are used in similar but different ways in other research communities [32]. The common pattern between the many different definitions is that a robust system preserves

[1]In the original definition of dependability, [20], security was considered as an attribute of dependability but in later years it has attained a different status [5]. It is now considered a composite concept, much like dependability itself. Instead, new attributes such as confidentiality and integrity has been added. Since we focus on unintentional faults we do not consider the security-related notions any further.

its features and continues to deliver its service even in the prescence of perturberations to it that has not previously encountered and that has not been considered during its design. The general term perturberation is used here since we might mean both input data as well as damages to the system itself.

2.1.1. Means to achieve software dependability

Many techniques has been proposed for how we can develop dependable software. They are frequently classified according to their purpose ([20, 5]):

- to prevent the introduction or occurence of faults - **fault prevention**
- to reduce the number or severity of faults - **fault removal**
- to estimate the number of faults in the system and the consequences they may have if activated - **fault forecasting**
- to deliver correct service even though there are faults - **fault tolerance**

Particular techniques for dependability may not fall cleanly into one of the above categories but adress multiple of them. Below we present some of the techniques for dependability that will be of the largest interest to us in the rest of this thesis. For more detailed coverage see [22, 30, 21].

2.1.1.1. Formal methods

Formal methods is a general term used to describe mathematically based frameworks for specifying, developing and verifying computer systems [41]. By limiting the language used to write specifications and design documents a formal method can give guarantees about the properties of the described system.

The focus is on fault prevention, ie. to prevent that faults are introduced into the system. For example, by writing a specification in a formal language we can find inconsistencies or ambiguities in the requirements and resolve them at the outset. There are no such guarantees if we write the specification in a natural language. The fact that formal methods can help us validate a specification is extra important for dependable systems since they need to be judged against a specification.

But formal methods can also help with fault removal. As an example a formal specification can be used to generate test data for verifying the system. When the testing shows a deviance from the correct behavior there is a fault that needs to be removed. If we have formally specified the behavior of sub-components of the system we would get further indications of the locations of the fault.

2.1.1.2. Redundancy: N-version programming

To be able to tolerate faults that remain when the system is in use we can add redundant components to the system. If we perform multiple computations, either sequentially or concurrently, that lead to the same result or decision we can tolerate faults that only arise in some of them.

The technique of N-version modular redundancy was first used to tolerate harware faults, but was carried over to software in the late 1970'ies [3]. An N-version, or multi-version, system (NVS) should incorporate several software components and include a voter that compares their output and chooses the output of the system. However, the same software component can not simply be replicated several times in an NVS; all replicates will contain the same fault and all components will fail for the same input data.

To overcome this special characteristic of software, Avizienis proposed that several software components, called variants , should be developed. The methodology for developing these variants was called N-version programming (NVP) and it uses several development teams, each developing a variant. The different development efforts should be independent, i.e. no interaction should be allowed between them and different algorithms and programming languages should be used wherever possible. The differences in the development efforts are called design diversity and the term software diversity has been used as a general term to denote differences both between the development efforts and between the resulting variants.

2.2. Biological Systems

According to the Collins English dictionary biology is *'the study of living organisms, including their structure, functioning, evolution, distribution and interrelationships'* [11]. An introductory biology textbook is even more succinct and simply call biology *'the study of life'* [10]. Biological would then mean of or relating to living organisms.

If 'biological' is a very broad term the situation is even worse for 'system'. Collins main description is as *'a group or combination of interrelated, interdependent, or interacting elements forming a collective entity'* but it also lists seven other meanings [11]. In a similar way the Principia Cybernetica Web site devoted to 'System Science' notes that there are dozens of definitions of 'system' [31].

For our purposes we can use these terms in a broad sense. We are seeking new ways to obtain dependable software and can take our inspiration from almost anywhere. Thus, in this thesis we use the term biological system to mean a living entity found in nature that has not been designed by humans. Examples would include a human body, a beehive or the ecology of a small forest with all the plants and animals living next to each other. In all these examples it is clear that we are talking about systems. A human body is an entity made up of different organs, a beehive of its bees and the ecology of its plants and animals.

Biological systems have often been the inspiration when man has designed a new tool or device. In recent years this trend has been stronger and especially in computer science and engineering. Let us look at two authors noting the dependability characteristics of biological systems [16, 2, 4].

In his book 'Out of Control', Kevin Kelly gives examples of how technicians look to nature for inspiration on how to build complex technical systems. His thesis is that we are entering a new era where machines will become more biological and the biological will become more engineered.

Dependability is at the core of Kelly's thesis. He says that as our technical systems grow more complex they will not work at all unless they become more similar to biological systems. We must learn how nature manages its complex systems since we fail to build them ourselves.

But Kelly also says that when we build more life-like systems inspired by nature they will be out of our control.

The most extreme reading of Kelly leads to a viewpoint we can call the biodependability-by-necessity viewpoint:

> **Definition 2.1 (Biodependability-by-necessity):** There are only a few ways to and principles for building functioning, complex systems and nature has found and exploited them. When the systems we humans build grow larger we cannot avoid that they become life-like, if we want them to work.

This may not be so. The biological systems around us are the results of the process of evolution. It can be thought of as a search through a vast search space. Arbitrary 'choices'[1] made early in the search may limit what portions of the space can later be reached. So one cannot rule out the possibility that we can find techniques for building complex systems that nature is not using. For example, some formal method could enable us to build correctly functioning complex systems, but we don't see any formal methods being used in the biological systems we know of.

A milder interpretation of Kellys text would be a pragmatic viewpoint on biodependability:

> **Definition 2.2 (Biodependability-pragmatism):** Some biological systems have features we want our systems to have so they must be doing something right. By

[1]The term choice is a bit unfortunate since it indicates an active decision. However, its hard to find a nice, short substitute.

> studying nature we can understand how these features come about and use the knowledge in our own systems. By imitating nature we may give our systems some of the desired features.

In contrast to Kelly, the author of this thesis considers himself to be a biodependability-pragmatist.

Algirdas Avizienis calls the various species of living creatures that have survived for millions of years *’the most dependable information processing systems in existence’* and challenges us to use them as models for more dependable computer systems [4]. Specifically, he proposes the human immune system as a conceptual model for high-confidence hardware systems. In this model he likens the hardware of a computer system to the human body and the cognitive processes the body supports to the software. He notes four features of the immune system as especially relevant:

- it functions continuously and autonomously independent of the higher cognitive processes
- its elements are distributed to serve all parts of the body
- it has its own communication system
- its elements are redundant and diverse

Avizienis urges the research community to consider how to build such features into our computer systems.

2.2.1. Characteristics of biological systems

Even though biological systems encompasses a very large number of things they share some common characteristics. The system aspect with parts-within-parts is typically present on many different levels and each level build on the level below [10]. Also there are different distinct processes that affect biological systems.

2.2.1.1. Multiple levels of organization

Biological systems have multiple levels of organization. At each level new properties arise from the interactions between components. These emergent properties are not ’magical’, they just reflect that the system has properties that cannot be inferred by looking at its individual components in isolation. An analogy to software might be that the dynamical interaction of software components might not be apparent by reading the source code for its components. An everyday analogy would be that we cannot easily grasp the many different strategies and game states possible in chess from its simple rules.

The basic low-level unit of organization in all living systems is the cell. A human body consists of on the order of 10^{14} cells. The cells are not identical but come in a large variety. They aggregate and form larger structures to perform tasks at the next level in the system. Cells are often specialized to the task they help carry out. Even though cells are mostly studied in their aspect as the building blocks of larger systems they are special since they are the smallest biological system performing all the activities of life [10].

In multi-cellular organisms the next level of organization is tissue. Cells group into tissue like muscle fiber and nervous tissue. Different type of tissue then group into organs. They carry out specialized tasks that emerge from the activity of the constituent tissue. Examples in the human body are liver, heart and lungs.

Sets of organs form systems that take care of several related tasks. Examples are the nervous system consisting of brain, the spinal cord, the sensory system and connecting ’wiring’.

An organism is a single body with many interacting systems. Organisms can reproduce and perform many different tasks. They are open systems that continously interact with their environment and exchange energy and materials.

On the top level the many interactions between organisms and their environment and other organisms form an ecosystem. In an ecosystem nutrients circulate between different organisms. Energy flows into the system from the sun that keeps the circulation going.

2.2.1.2. Three main biological processes

Biological organisms do not arise out of thin air. They are the products of processes that create and affect their development. The first process in the creation of an organism is the *development from an embryo.* It is governed by information inherited from the parents stored in DNA molecules in the core of the cells but it is also affected by the environment in which development takes place.

When an organism has developed it does not stop changing. The process of *learning* now adapts the organism to better function in its environment.

Just as each organism has a history so the species, the type of organism, it belong to has. The species has evolved from its early ancestors to its current form. By going back in time we find that different species have common ancestors. This overarching development of living organisms, on the population level, is governed by the *process of evolution.*

2.2.2. Contrasting biological and human-made systems

2.2.2.1. Complexity

Biological systems are often claimed to be complex. How complex are they in relation to the complexity of the systems we build?[1]

It is not clear what metric to use to compare complexity of different systems. There has been attempts to define information theoretic measures. One such measure would be the minimum number of bits needed to describe the system. This is good since it would solve the problem with simpler metrics such as counting the atoms. We don't need many bits to describe a repetitive pattern such as a crystal even though it may contain very many atoms.

On the other hand a hot gas, has a large number of atoms running around and hitting each other and describing the gas in detail would take many bits. The concept of 'logical depth' tackles this problem by measuring the amount of computation needed to produce the minimal bit pattern that describes a system [34, 8].

In practice it is hard to use information theoretical measures such as logical depth since they have a some input arguments that are generally not known. It is not clear that we are better off by estimating these arguments and using the proposed formulas than using a simpler metric. A basic metric would be to compare the number of components in each system. Even though a case could be made that individual molecules are active components in some systems it is too simplistic to compare the number of atoms. For each system we need to decide on what we consider a system component.

For biological organisms a natural component is the cell. An ant has 10^{10} cells and a human has $10^{14.}$ If we compare this to a 2GHz Pentium 4 processor from Intel it has 42 million transistors, ie. on the order of 10^{7} components.

If we look at software the recently released Windows XP operating system from Microsoft Corp. is rumoured to have around 40 million lines of source code. In one sense this could be seen as a more complex system than the Pentium 4 processor; there are few constraints in software and each line of code could potentially interact with any other.

It would appear that the most complex system humanity has built would be the Internet with its about 10^{8} connected computers. If the Internet was viewed as a meta-computer and we counted the number of transistors in it we would get to the same or-

[1]We are talking about complexity in the sense of an aggregated structure consisting of many parts, and not as in hard to understand. No doubt, the two meanings are related, eg. we have a hard time understanding all the intricacies of the human body since it is such a complex system with many parts.

der of components as the cells in a human body. But we don't consider this meaningful since the individual transistors are not a component in the Internet system.

We conclude that biological systems can be characterized as 'complex' in some loose sense and that the systems we build typically are less 'complex'. However, we note that such statements are hard to formalise. There is not even a consensus on what complexity is.

2.2.2.2. Self-diagnosing and Self-healing

Biological systems are often massively parallel with many independent but interconnected sub-units. The huge number of units makes for extensive redundancy and the systems can use this to diagnose and heal themselves in case of damage.

Based on the self-healing capabilities of biological system a team of material science researchers have developed self-healing polymers (plastic) [40]. The material heals cracks autonomically by releasing a chemical catalysts that bonds the crack faces back together. The findings will lead to materials with longer lifetimes and less requirements for maintenance.

2.2.2.3. Adaptive

Biological systems continously interact with and adapts to their environment. In contrast to human-engineered systems that have a closed structure and behavior once designed biological systems tend to be open and changeable.

An example of adaptiveness is homeostasis, the ability of an organism or cell to maintain internal equilibrium by adjusting its physiological processes. An example is the regulation of body temperature.

2.2.3. A biomimetic algorithm

In this section we give an example of a biomimetic algorithm by describing evolutionary algorithms. This serves both as an introduction to that specific algorithm and highlights the issue of biological plausibility.

Evolutionary algorithms mimic the evolutionary process in nature to find solutions to problems. Genetic programming (GP) is a special form of evolutionary algorithm in which the solution is expressed as a computer program. It is essentially a search algorithm that has been shown to be general and effective for a large number of problems. These algorithms are studied in the area of Evolutionary Computation (EC) in artificial intelligence.

In the classical view of natural evolution, individuals in a population compete for resources [10]. The most 'fit' individuals survive, i.e. they have a higher probability of having offspring in the next generation. This process is modeled in genetic algorithms in which the individuals are objects expressing a certain, often partial or imperfect, solution to the problem investigated. In each generation, each individual is evaluated as to how good a solution it constitutes. Individuals that are good are chosen for the next generation with a higher probability than low-fit individuals. By combining parts of the chosen individuals into new individuals, the algorithm constructs the population of the next generation. Mutation also plays an important part. At random, some parts of an individual are randomly altered. This is a source of new variation in the population.

While a genetic algorithm generally works on data or data structures tailored to the problem at hand, genetic programming works with individuals that are computer programs. This technique was introduced by Koza in [18] and has recently spurred a large body of research [19, 6]. Kozas programs are functions, represented as trees, which are interpreted in software, but a number of other approaches also exist. For example, in [25], Nordin evolved machine language programs that control a miniature robot.

A number of GP systems are available. To use one of them to solve a particular problem, we must tailor it to the problem. This involves choosing the basic building blocks, such as variables and constants, and functions that are to be used in the programs, expressing what are good and bad characteristics of the programs, and choosing values for the control parameters of the system and a condition for when to terminate the evolution of programs [18]. The control parameters prescribe, for example, how many individuals are to be in the population, the probability that a program should be mutated and how the initial population of programs should be created.

This example of a biomimetic algorithm raises an important question: How similar to a biological phenomenon must a computational model or algorithm be for it to be called biomimetic? The genetic programming algorithm is not very biologically plausible; it manipulates trees of symbolic information. We see no clear answer to this question. In any biomimetic work there will be some kind of model that we try to carry over to an algorithm. The resulting system can be more or less biologically plausible.

3. Related Work in Biomimetic Dependability

This section reviews the previous work related to biomimetic dependability. We include hardware-related systems in this review since much more work has been done on that compared to software. However, our coverage of hardware-related research is not in any way complete and should only be taken as a brief overview.

3.1. Hardware-related Biomimetic Dependability Research

Most work on biologically inspired computing has been hardware-related. There are conferences solely devoted to 'Evolvable Hardware'[1] with much of the work applying evolutionary algorithms to the configuration of field-programmable gate arrays (FPGA) circuits.

But inspiration for novel HW architectures is not limited to evolutionary processes. Moshe Sipper et al have introduced the POE model for bio-inspired hardware systems [33]. The model has taken its name from the three main axes used in its classification scheme: Phylogeny, Ontogeny and Epigenesis. The terms are used to denote three major levels at which biological systems change. Phylogeny refers to evolutionary processes, in which the genetic information evolve over time. Ontogeny refers to the development of an individual organism from embryo to adult. Epigenesis is the development that an organism goes through after its basic structure has been fixed, ie. its about the learning processes of the organism. In the following we will refer to these three levels as evolution, development and learning[2]

3.1.1. Embryonics

Embryonic systems are electronic systems inspired by the embryonal development

[1] Examples are the NASA and DoD workshop on Evolvable Hardware and the conference 'Evolvable systems: from Biology to Hardware'

[2] We do not use the three terms taken from the field of biology since their meaning is not clear and their use by Sipper et al differs somewhat from how they are commonly used [33, 10]. Epigenesis is commonly used to denote the theory that an individual is developed by successive differentiation of an unstructured egg rather than by a simple enlarging of a preformed entity. And Sipper et als description of ontogenesis seems to focus on cell differentiation, commonly called morphogeneis. However, when refering to the model as a whole we will call it the POE model.

of stem cells in biological life forms. Stem cells are born identical and each have the same DNA information in their cores. Later they develop to carry out specific tasks. The specialization is based on the cells position in a larger structure; their physical 'address' chooses the function they take on.

Embryonic systems consist of an array of identical processing elements (cells), all of which are controlled by their own configuration register. The registers contains configuration information for each cell in the array. Each cell uses its address to look up its configuration.

The cells also have built-in self-test and can detect when they are faulty. If this happens the cell will pass data and addressing information straight through without processing it. This will render it transparent and the cell addresses will change and affected cells will reconfigure to new functions. As long as there are enough redundant cells to overtake the responsibility of faulty cells the array as a whole can uphold its task.

The reliability of embryonic circuits have been analysed [27, 26, 28] and embryonic systems have been implemented and evaluated both in simulations and in FPGA's.

The properties of biological systems that embryonics exploit is the repeated use of similar elements (cells / processing elements) that specialize according to their position. In the POE model embryonic systems would be characterized as based on the concept of ontogeny. Recently, other properties of multicellular biological systems have been exploited in the embryonics field. For example, Jackson and Tyrrell proposes asynchronous embryonics where the action of the cells are not dictated by a central authority (global clock) [14].

3.1.2. Immunotronics

The human immune system learns to distinguish between cells and molecules that belong to the body and those that are external, ie. bacteria and viruses. It is thus based on the epigenesis level in the POE model. Artificial immune systems tries to mimic similar processes in computers. They have been used for computer security, virus detection and robot control. Recently Bradley and Tyrrell have proposed electronic hardware based on immunological principles [9]. They call their approach immunotronics.

The immunotronic system includes a finite state machine and a detection system that detects faulty states and state transitions. The detection system uses the negative selection algorithm inspired by the human immune system. The detection works on bit strings representing the state, inputs and transitions of a finite-state machine (FSM). These strings are called system strings. It is set-up by generating a set of test criteria and exposing them to correct FSM strings. The ones that do not match correct strings are retained for later use in the detection system. This negative selection of test criteria has given the algorithm its name. During the operation of the FSM the detection system matches the test criteria against the system strings. Upon a match the system is said to have failed.

The benefits of the negative selection algorithm is that the failure probabilities of the detection system can easily be traded off to its memory requirements. A prerequisite to using the negative selection algorithm is that the number of correct system strings are much less than the number of faulty ones.

Bradley and Tyrrell have implemented an immunotronic system for a FSM counter.

3.1.3. Evolving robustness and fault-tolerance

Adrian Thompson and Paul Layzell have used evolutionary algorithms to evolve robust electronic designs in field-programmable gate arrays (FPGA's) [36]. By varying the environmental conditions for the evolutionary system during evolution the resulting designs were robust to temperature variants. Thus the evolutionary algorithm evolved robustness even though it was not explicitly stated in any fitness function.

In a previous study Thompson showed that evolutionary algorithms could evolve solutions that were tolerant to faults [35]. The effect arises naturally from an evolutionary process but can also be explicitly selected for by specifying it in the fitness function.

3.2. Software-related Biomimetic Dependability Research

There are not many studies applying biomimetic algorithms in a software engineering domain to acheive depenadbility. This has been noted by for example Harman and Jones who wrote a 'manifesto' for more research in this area [13].

3.2.1. Evolutionary algorithms for testing

There has been a number of studies that use genetic algorithms (GA's) for structural testing, ie. ensuring that all parts of the implementation are executed by the test set. Jones et al used a GA to generate test-data for branch coverage [15]. They use the control-flow graph (CFG) to guide the search. Loops are unrolled so that the CFG is acyclic. The fitness value is based on the branch value and the branching condition. They evaluated the approach, with good results, on a number of small programs.

Michael and McGraw at RST corporation have developed Gadget - a tool for generating test data that give good coverage of C/C++ code [23]. Gadget work for the full C/C++ syntax and automatically instruments the code to measure the condition/decision coverage. This requires that each branch in the code should be taken and that every condition (atomic part of a control-flow affecting expression) in the code should be true at least once and false at least once. Four different algorithms can be used to search for test data in Gadget: simulated annealing, gradient descent and two different genetic algorithms. One of the GA's was the best on a large (2046 LOC) program which is part of an autopilot system. However, on synthetic programs the GA had problems with programs of high complexity. In all of the experiments random testing fared the worst when the complexity increased.

Pargas et al use a GA to search for test data giving good coverage [29]. They use the control dependence graph instead of the control flow graph since it gives more information on how close to the goal node an execution was. Their system uses the original test suite developed for the SUT as the seed for the GA since it should cover the programs requirements. To reduce the execution time their system employs multiple processors. They compare their system to random testing on six small C programs. For the smallest programs there is no difference but for the three largest programs the GA-based method outperforms random testing.

Tracey et al presents a framework for test-data generation based on optimisation algorithms for structural testing [37]. It is similar to both Jones et al and Michael and McGraw approaches and uses a CFG and branch condition distance functions. They use both simulated annealing and a genetic algorithm for the optimisation. Their tool is automated and works with ADA code.

Tracey have used a similar technique for functional (black-box) testing [38]. The formal specification is described with pre- and post-conditions that each function must obey. The goal is to find indata that will fullfill the pre-condition and the negated post-condition. These expressions are converted to disjunctive normal form. All pairs of single disjuncts from pre- and post-conditions are considered targets for the search since a fault is found when either of them is fulfilled. The search algorithms are guided by similarity measures that judge how close to violating a post-condition a candidate test data set is.

The research by Tracey et al is unique in that they have evaluated their techniques on a real-world safety-critical system [39]. The goal was to assist in exception freeness proofs by finding test data that caused exceptions. The results was very encouraging; the test-data generation system was able to find test data that caused exceptions and covered the exception handling code efficiently.

Mueller and Wegener used an evolutionary algorithm to find bounds for the execution time of real-time programs and compared it to static analysis of the software [24]. Even though the evolutionary algorithm cannot give any safe timing garantuees it is universally applicable and only requires knowledge about the programs interface. Static analysis can give garantuees but only in a theoretical world. It needs extensive knowledge about the actual hardware if we are to trust the results. Such knowledge may not always be available.

Baudry et al have used genetic algorithms for evolving test sequences for mutation testing of Eiffel programs [7]. Their model is similar to ours in that they focus on specification, implementation and tests. Their specifications are written with pre- and post-conditions and invariant. A tool mutates the programs and a genetic algorithm searches for test sequences that kills the mutants. The GA is seeded with test sequences written by the developer. Mutants that are not killed by the GA are analyzed by hand to see if they are mutants that did not change the workings of the software.

3.2.2. Cell-based programming models

Some research groups are studying how programming models inspired by biological systems can be used to build more robust systems. George et al recently introduced such a model where cell programs are automatons containing discrete states and transitions between the states [12]. Cells can sense there immediate neighbourhood and send out chemicals. They can also divide. The authors believe they will be able to build self-healing software with the model.

The ’Amorphous Computing’ group at MIT studies organizational principles and programming languages for coherent behavior from the cooperation of myriads of unreliable parts [1].

4. Summary of papers

4.1. Paper 1. A Theory of Software Development

We present a theory of software development as an incremental learning process. The focus is on the internal models of the developer. There are two main ways in which a development process can make progress: by refining an internal model or by refining an artefact based on an internal model. Refining the internal models is a prerequisite for being able to write a concrete specification and program that show acceptable behaviour. The theory has implications for tools to support software development. By creating novel test cases they can force the developer to question his internal models and realize where they are incomplete or incorrect.

4.2. Paper 2. An Interactive Software Development Workbench based on Biomimetic Algorithms

Based on the theory for software development presented in paper 1 this paper presents a design for an interactive workbench to support the iterative refinement of the internal models of the developer. The goal for the workbench is to expose unknown features of the software being developed so that the developer can check if they correspond to his expectations. The workbench employs a biomimetic search system to find tests with novel features. The search system assembles test templates from small pieces of test code and data packaged into a cell. We describe a prototype of the workbench implemented in Ruby and focus on the module used for evolving tests. A case study show that the prototype supports development of tests that are both diverse, complete and have a meaning to the developer. Furthermore, the system can easily be extended by the developer when he comes up with new test strategies.

4.3. Paper 3. An Experiment on Using Genetic Programming to Generate Multiple Software Variants

Software fault tolerance schemes often employ multiple software variants developed to meet the same specification. If the variants fail independently of each other, they can be combined to give high levels of reliability. While design diversity is a means to develop these variants, it has been questioned because it increases development costs and because reliability gains are limited by common-mode failures. We propose the use of genetic programming to generate multiple software variants by varying parameters to the genetic programming algorithm. We have developed an environment to generate programs for a controller in an aircraft arrestment system. Eighty programs have been developed and tested on 10000 test cases. The experimental data shows that failure diversity is achieved but for the top performing programs its levels are limited.

4.4. Paper 4. Genetic Programming as an Explorative Tool in Early Software Development Phases

Early in a software development project the developers lack knowledge about the problem to be solved by the software. Any knowledge that can be gained at an early stage can reduce the risk of making erroneous decisions and injecting defects that can be expensive to eliminate in later phases. This paper presents the idea of using genetic programming to explore the difficulty of different input data in the input space, determine the effects of different requirements and identify design trade-offs inherent in the problem. Data from a pilot experiment is analysed and the knowledge gained is used to question and prioritize the requirements on the target system. Coping with high-dimensional input spaces and establishing the relationship between GP- and human-developed programs are identified as the major outstanding problems. An extended experimental environment is proposed based on techniques for visual database exploration.

4.5. Paper 5. Forcing Software Diversity by Making Diverse Design Decisions - an Experimental Investigation

When developing software versions for a multi-version system, the probability for coincident failures may be decreased by forcing the development efforts to be different by making diverse design decisions. There are theorems showing that the probability is minimized by making as diverse design decisions as possible but it is not known if the assumptions made in proving the theorems are valid in practice. To investigate this we have developed 435 versions of a software controller for an aircraft braking system. The versions were developed using genetic programming. Analyses of the failure behavior of these versions showed that the assumptions of failure independence among the decisions were valid, on average, for 74% of the test cases. The assumption of indifference between methodologies were not valid in a single case which seems to be the major cause invalidating the theorem. Thus, if we are not indifferent between design decisions, it is not guaranteed that increased diversity of design decisions will decrease the probability of coincident failures.

4.6. Paper 6. Using Factorial Experiments to Evaluate the Effect of Genetic Programming Parameters

Evolutionary algorithms such as genetic programming have very many parameters that affect their effectiveness. Tuning them to the problem at hand can be difficult. This paper presents one approach to this problem based on statistical techniques for design and analysis of factorial experiments. The methodology allows both the stand-alone and combined effects of different parameters on the outcome to be studied.

Three binary classification problems are investigated in a total of seven experiments consisting of 1108 runs of a machine code genetic programming system. The parameters having the largest effect in these experiments are the population size and the number of generations. A large number of parameters have negligible effects. The experiments indicate that the investigated genetic programming system is robust to parameter variations, with the exception of a few important parameters.

4.7. Paper 7. GP-Beagle: A Benchmarking Problem Repository for the Genetic Programming Community

Experimental studies in genetic programming often only use a few, artifical problems. The results thus obtained may not be typical and may not reflect performance on problems met in the real world. To change this we propose the use of common suites of benchmark problems and introduce a benchmarking problem repository called GP-Beagle. The basic entities in the repository are problems, problem instances, problem suites and usage information. We give examples of problems and suites that can be found in the repository.

5. Discussion

It is hard to call biological systems dependable in the technical sense of the word since they have no real notion of a specification. Without a specification we cannot judge when they fail and we have no basis for evaluating the attributes we use to describe dependable systems. The only way forward seems to be to consider the high-level goals of survival, procreation etc common to biological organisms as a kind of specification. However, for biological systems that are not organisms even this approach is not clear-cut. As an example we find it hard to state the goals for an ecosystem or to even say that there are any clear goals. Our answer to question 1.1 is thus that biological systems are too different from the human-made systems we ascribe dependability attributes that we cannot use the dependability nomenclature in any strict sense.

The study reported in paper 3 found that we could use genetic programming to develop diverse software variants for use in fault tolerant systems. However, the failure diversity of the variants was limited so the method is of limited practical value. As an automatic programming technique genetic programming is very immature. The size of the programs that can realistically be developed is very small compared to programs developed by humans. This may change in the future with better genetic programming methods and more powerful computers.

A fundamental problem with using genetic programming to directly develop dependable software is that the solutions found are often orthodox. They make creative use of language constructs in ways we are not used; understanding the resulting programs can be very hard. It is doubtful that we would be willing to trust software we do not understand. However, the question is not clear-cut since we trust systems we do not understand every day. A counterpoint would be that in many cases there at least exists someone that understands them. In any case we do not think that direct development of software for dependable software using genetic programming is an option unless severe restrictions is put in which part of the system the GP-developed software can affect.

The study above was not in vain since it showed that the GP-developed software variants could be used to gain insight about the nature of the problem domain. We think the technique for problem difficulty visualization reported on in paper 4 has clear potential, especially since increased computer power will make simulations increasingly attractive.

The main limitation for using the technique would be how to visualize multi-dimensional problem spaces. However, this problem is quite general and much research focus on resolving it. Even if no suitable techniques for visualization can be found the technique could be of value by giving numerical values for comparing desing decisions and trading off requirements.

Part 1 of this thesis also studies indirect ways of using biomimetic algorithms for software dependability. Even though the WiseR prototype has so far only been evaluated on small problems the approach shows great promise. A major cause of this is that it invites the developer into the loop. The system is designed to be flexible enough so that the developer can easily extend it. This allows the developer to search in parts of the space of all tests that he thinks is relevant. Even though this may limit the creativity of solutions that can be found nothing stops the system from falling back on more open-ended search when the developer is not around or when no more progress is done in the more focused searches.

We think that the behavioural specifications that are the end results from WiseR sessions are a formalism that is attractive to many developers. a possible reason that traditional formal methods have not yet caught on could be that they require much from the developers. Developers need to learn a new language and understand new mathematical concepts. In contrast, the atomic behavioural specifications of WiseR are a formalism-by-example. They are concrete and show the required behaviour in a language the developer is familiar with. They are also flexible and allow the developer that so wishes to add properties that the software should fulfill.

6. Conclusions and Future Work

6.1. Conclusions

Software is an important part of our societies and we need ways to ensure it does not fail. Software failures are ultimately caused by faults in the software. Since the software is developed by humans the cause of the fault can be traced back to a developer having the wrong view of the real world or making an erroneous decision cause there was not enough information available. The goal of this thesis has been to explore how algorithms modeled after biological systems can help the developers avoid or overcome this.

Part 1 of this thesis presents a workbench for interactive software development that directly helps the developer by coming up with novel test cases. By browsing and studying the test cases the developer can identify bugs in the software or problems with the specification. The system is unique in that it interacts with the developer and indirectly uses his choices to guide the search for test cases.

The power of the WiseR system comes from its ability to combine test building blocks in new ways and the ease with which different parts of the system can be extended. By having a biomimetic search algorithm at its core the system can handle this flexibility and learn what combinations of building blocks are valid and meaningful. The biomimetic algorithm is thus used as an explorer.

The explorative capability of evolutionary algorithms was also used in the studies in part 2. Even though the task of developing a program controlling the arrestment of aircraft on a runway proved to hard for the evolutionary algorithm the results could be used to visualize the difficult areas of the problem. Thus, the flexibility and explorative nature of the biomimetic algorithm enabled knowledge about the problem domain to be gained at an early stage. Based on this we described a system which allows the developer to play with different alternatives and study how requirements affect the difficulty of the problem.

In summary, I propose that biomimetic algorithms are a valuable tool in building developer workbenches for exploring requirements, specifications and implementations and thus help avoid faults in software. The algorithms most prominent attributes are that they are easy to set up, are flexible enough to allow for easy extensions, adapt dynamically to changing conditions and show creativity in the solutions they come up with. This fits with the software development process since it is fundamentally a creative task with only partly known or even ill-defined boundaries.

6.2. Future work

There are many ways in which the WiseR prototype developed in paper 2 could be extended. One interesting possibility would be to have it save information about the actual faults the developer commits in programs. By comparing the faulty with the corrected implementation the system could build a knowledge base of faults the developer frequently commits. This information could be used to warn for code that is similar to previously faulty code. It could also be used for mutation testing so that the biomimetic system could search for tests that identified mutants.

More experiments should be carried out with the WiseR prototype. It has currently been used only on small problems. One way to get feedback on the prototype would be to release it as open source. If the system was adopted by several developers one could collect knowledge on how different developers interact with the system.

Having several developers using WiseR would also open for experiments on how to connect the libraries of different WiseR systems. Since the building blocks for test cases are small snippets of code they can simply be shared but having the libraries connect directly to each other could allow for more powerful knowledge transfer and should be investigated.

Even though the goal when designing WISE was for a general system the detailed design of the prototype is fairly tied to the Ruby programming language. It is unclear how the power of WISE would be affected if implemented in another language without Ruby's dynamic features. Investigating this is an important task for the future.

This thesis has not explored how biomimetic algorithms can be used online, in a running software system and how that would affect dependability. Investigating this unexplored area would be a bold future move. Maybe there are hybrid solutions that can ensure safety while still allowing the adaptive and dynamic biomimetic algorithms to play an important role. Safety-caged systems where a a formal safety specification restricts what the biomimetic algorithm can do? Only time and bold researchers will tell…

References

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald J. Sussman and Ron Weiss. Amorphous Computing. *Communications of the ACM* **43** (5), 74-82 (2000).

[2] Algirdas Avizienis. Building Dependable Systems: Can We Keep Up with Complexity?. In *FTCS-25 Silver Jubilee*, 1995.

[3] A. Avizienis. The Methodology of N-version Programming. In Michael R. Lyu (ed.), *Software Fault Tolerance*. WileySons, Chichester, England, September 1995.

[4] Algirdas Avizienis. Towards a systeatic design of fault-tolerant systems. *IEEE Computer*, 51-58 (April 1997).

[5] Algirdas Avizienis, Jean-Claude Laprie and Brian Randell. Fundamental Concepts of Dependability. In *Third Information Survivability Workshop – ISW-2000*, 2000. URL *http://www.cert.org/research/isw/isw2000/index.html*.

[6] Bäck, T., Hammel, U. and Schwefel, H-P.. Evolutionary Computation: Comments on the History and Current State. *IEEE Transactions on Evolutionary Computation* 1, 3–17.

[7] Benoit Baudry, Vu Le Hanh, Yves Le Traon. Testing-for-Trust: The Genetic Selection Model Applied to Component Qualification. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, 2000.

[8] C.H. Bennett. Logical Depth and Physical Complexity. In R. Herken, *The Universal Turing Machine, A Half-Century Survey*. Oxford University Press, Oxford, 1988.

[9] D. W. Bradley and A. M. Tyrell. The Architecture for a Hardware Immune System. In Didier Keymeulen and Adrian Stoica and Jason Lohn and Ricardo S. Zebulum, *Proceedings of the 3rd NASA/DoD workshop on Evolvable Hardware, Long Beach, California*, pages 193–200. IEEE Computer Society, 2001.

[10] Neil A. Campbell. *Biology*. Benjamin Cummings, Menlo Park, CA, 1996.

[11] *Collins English Dictionary*. HarperCollins Publishers, Glasgow, 1994.

[12] Selvin George, David Evans and Lance Davidson. A Biologically Inspired Programming Model for Self-Healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

[13] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology* **43** (14) (2001).

[14] A. H. Jackson and A. M. Tyrrell. Asynchronous Embryonics with Reconfiguration. In *Proceedings of 4th International Conference on Evolvable Systems, Tokyo, Japan*, 2001.

[15] B. Jones, H. Sthamer, D. Eyres.. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal* **11** (5), 299–306 (September 1996).

[16] Kevin Kelly. *Out of control - the new biology of machines, social systems and the economic world*. Perseus books, Cambridge, MA, 1995.

[17] John C. Knight and Kevin J. Sullivan. Towards a Definition of Survivability. In *Third Information Survivability Workshop – ISW-2000*, 2000. URL *http://www.cert.org/research/isw/isw2000/index.html*.

[18] Koza, J. R.. *Genetic Programming - on the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[19] Koza, J. R.. *Proceedings of Second Annual Conf. on Genetic Programming, San Fransisco, California*, July 1997.

[20] Jean-Claude Laprie (ed.). *Dependability: Basic Concepts and Terminology*. Springer Verlag, 1992.

[21] Michael R. Lyu (ed.). *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, NY, 1995.

[22] Michael R. Lyu (ed.). *Software Fault Tolerance*. John Wiley and Sons, Ltd., Somerset, NJ, 1995.

[23] Christoph C. Michael and Gary McGraw. Automated Software Test Data Generation for Complex Programs. In *Proceedings 13th IEEE Conference in Automated Software Engineering*, pages 136–146. IEEE Computer Society, October 1998. URL *citeseer.nj.nec.com/66954.html*.

[24] F. Mueller and J. Wegener. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. In *IEEE Real Time Technology and Applications Symposium*, 1998.

[25] Nordin, P., and Banzhaf, W.. Real-time Evolution of Behavior and a World Model for a Miniature Robot Using Genetic Programming. Tech. Rep. (1995), Department of Computer Science, University of Dortmund.

[26] Cesar Ortega and Andrew M. Tyrrell. Reliability Analysis in Self-Repairing Embryonic Systems. In A. Stoica et al, editors, *Proceedings of 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA*, pages 120-128. IEEE Computer Society, July 1999. URL *citeseer.nj.nec.com/ortega99reliability.html*.

[27] Cesar Ortega and Andrew M. Tyrrell. Self-Repairing Multicellular Hardware: A Reliability Analysis. In *European Conference on Artificial Life*, pages 442-446, 1999. URL *citeseer.nj.nec.com/280270.html*.

[28] C. Ortega, D.Mange, S.L. Smith and A.M. Tyrrell. Embryonics: A Bio-Inspired Cellular Architecture with Fault-Tolerant Properties. *Genetic Programming and Evolvable Machines* **1** (3), 187–215 (July 2000).

[29] Roy P. Pargas and Mary Jean Harrold and Robert Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability* **9** (4), 263–282 (July 1999). URL *citeseer.nj.nec.com/pargas99testdata.html*.

[30] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, 1996.

[31] F. Heylighen, C. Joslyn and V. Turchin. Definition of 'System' at the Principia Cybernetica Web, December 2001. URL *http://pespmc1.vub.ac.be/SYSTEM.html*.

[32] Erica Jen. Working Definitions of Robustness, January 2002. URL *http://discuss.santafe.edu/robustness*.

[33] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer.. A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *IEEE Transactions on Evolutionary Computation* **1** (1), 83–97 (April 1997).

[34] Nick Szabo. Measuring Complexity, 1997. URL *http://szabo.best.vwh.net/complexity.html*.

[35] A. Thompson. Evolutionary Techniques for Fault Tolerance. In *Proc. UKACC Int. Conf. on Control 1996 (CONTROL'96)*, pages 693-698. IEEE, 1996.

[36] A. Thompson and P. Layzell. Evolution of Robustness in an Electronics Design. In *Proc. 3rd Int. Conf. on Evolvable Systems (ICES2000): From biology to hardware*, 2000.

[37] N J Tracey and J A Clark and K C Mander and J A McDermid. An Automated Framework for Structural Test-Data Generation. In *Proceedings 13th IEEE Conference in Automated Software Engineering*. IEEE Computer Society, October 1998. URL *http://www.cs.ukc.ac.uk/pubs/1998/974*.

[38] Nigel Tracey and John Clark and Keith Mander. Automated Program Flaw Finding using Simulated Annealing. In *Software Engineering Notes*, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–81. ACM SIGSOFT, March 1998. URL *http://www.cs.york.ac.uk/testsig/publications/njt-mar98b.html*.

[39] N. Tracey, J. Clark, K. Mander, J. McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience* **30** (2000).

[40] S. R. WHITE, N. R. SOTTOS, P. H. GEUBELLE, J. S. MOORE, M. R. KESSLER, S. R. SRIRAM, E. N. BROWNS. VISWANATHAN. Autonomic healing of polymer composites. *Nature* 409, 794-797 (2001).

[41] Jeanette M. Wing. A specifier's introduction to formal methods. *IEEE Computer* 9, 8-24 (1990).

Part I.

1. Robert Feldt. *A Theory of Software Development*, Technical Report no. 02-26, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002.

2. Robert Feldt. *An Interactive Software Development Workbench based on Biomimetic Algorithms*, Technical Report no. 02-16, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002. *A condensed version of this report has been submitted to IEEE Transactions on Evolutionary Computation.*

Paper 1.

Robert Feldt. *A Theory of Software Development*, Technical Report no. 02-26, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002.

A Theory of Software Development

Robert Feldt

Department of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden

November, 2002

Abstract

We present a theory of software development as an incremental learning process. The focus is on the internal models of the developer. There are two main ways in which a development process can make progress: by refining an internal model or by refining an artefact based on an internal model. Refining the internal models is a prerequisite for being able to write a concrete specification and program that show acceptable behaviour. The theory has implications for tools to support software development. By creating novel test cases they can force the developer to question his internal models and realize where they are incomplete or incorrect.

1. Introduction

Our society increasingly relies on computers. Computers are controlled by software that tells them what to do. It is thus important that we theories of how to develop this software.

The fundamental problem of software development is that of refining mental pictures of how a problem could be solved into a program that a computer can run to actually solve the problem. The problems are typically ill-defined and only partly known. The mental pictures about possible solutions are similarly incomplete and fuzzy. Both problems and the mental pictures of how to solve them change over time. How should a software developer approach this situation?

We wish to consider general problems involving software development. To do this it is first necessary to identify the various elements involved and define them. We can then relate them in a model and define what we mean with a software development process and study which steps advances it.

In this paper we assume that the role of the program is fairly well known, ie. the development task is not about helping a problem owner explore the potential ways how a computer plus a program could solve the problem. We assume this connection between a problem and a computer-based solution has already been established. Even though a more open-ended development situation may better reflect a majority of software development situations we here focus on the simpler problem while noting that more refined theories will be needed in the future.

2. Elements

Our model for software development has five fundamental elements[1]:

1. A *patron* which has a need for a program to solve a problem.

2. A *specification* which states how the program should and should not behave.

3. The actual computer *program* being developed.

4. A *developer* that develops the program.

5. A *library* containing humanity's total accumulated knowledge relevant to the development problem at hand.

In any software development process there are two fundamental types of information: the computer program itself and information about how the program should behave. The software itself is a natural element since we know that the developer has to write it. Another element is information about what the software should do. A specification is where the developer collects this knowledge.

The specification and the implementation are the two main artefacts in a software development process. A third artefact are any tests the developer runs on the software during development. Even though it is thinkable that in some development processes no tests are run, in practice there will always be some tests. The reason that tests are needed is because the implementation is a static artefact. To assess whether it fullfills the specification it has to be executed. However, tests are not fundamental elements of our model since we will see that they are proto-requirements. Requirements are part of the specification.

2.1. Specifications

Informally specifications are any information that specifies the form and behaviour of a computer program and the process by which it is going to be developed. To us

[1]In the rest of the paper we will alter between using he and she when referring to the developer and/or the patron. Of course it does not matter whether they are actually female, male or machines.

the form is not relevant since we are interested in principles and not the specifics of a particular programming language or the naming of parts of the system. Even though the development process is our topic of study it is not fundamental to a specification. Any requirements on the process by which we reach a final software product can only help or hinder development. In theory it should not affect the final behaviour of the system itself.

Fundamental to a specification is that it specifies the behaviour of the running program. Programs are static artefacts we develop to give a computer a certain, useful, dynamic behaviour. Any development effort would be meaningless if it did not pay attention to how the resulting system is supposed to behave. A specification can also describe situations that should never arise, for example inputs that are invalid or outputs that are forbidden. We call such specifications behavioural.

The atomic parts of a behavioural specification are behavioural requirements. They describe an invocation of the program and whether its a valid invocation or not. If its a valid invocation they also state information about different program behaviours in that situation. If the requirement is permissive it states what behaviour is valid. If it is dismissive it states what behaviour is invalid. Formally we have:

Definition 1 (Atomic Behavioural Requirement): An *atomic behavioural requirement*, denoted ρ, is a tuple $\{\sigma, \iota, o, \chi\}$ where

- σ describes the state the program is in before this invocation
- ι the input to the program in this invocation
- o the output of the program in this invocation
- χ the information about this invocation or output where χ is one of
    - InvalidState - this state should never occur ($\iota = \emptyset$, $o = \emptyset$)
    - InvalidInput - this input is not allowed in this state ($o = \emptyset$)
    - InvalidOutput - this output is not allowed in this state for this input
    - DontCare - it is not important how the program behaves in this state and for this input (o can be $\emptyset$)
    - Valid - this is the valid behaviour for the program for this invocation

Note that in practice requirements are often not written in this atomic form. Requirements in traditional specifications often specify a large number of invocations at once. An example would be a mathematical expression relating the outputs to the inputs for some sub-domain of the valid input values. However, for our purposes we simply consider that a notational convenience.

We note how closely related behavioural requirements are to tests. A typical test has one part that sets it up and another part describing the inputs to the program[1]. When a test runs the program gives some output behaviour. A test can thus be described with the tuple $\{\sigma, \iota\}$ and a test run with the tuple $\{\sigma, \iota, o\}$. Thus, a test can be seen as a half-baked requirement; it simply lacks a statement about the validity of the invocation and the program behaviour.

A specification is simply a list of individual requirements.

Definition 2 (Behavioural Specification): A *behavioural specification*, denoted β, is a finite list of behavioural requirements: $\beta = [\rho_{1,} \rho_{2,} \ldots, \rho_n]$.

We also define the space of all possible specifications.

Definition 3 (Specification Space): The *specification space*, denoted $\hat{B}$, is the set of all possible specifications of the dynamic behaviour of computer programs.

Now for a certain development problem there will typically be many specifications in $\hat{B}$ that specify the wanted behaviour in a way that is good enough for the problem at hand. This is because there are situations in which we do not care how the system behaves. Given a development problem we can imagine a set of acceptable specifications that all describe the behaviour of the sought for program in a way that would be acceptable for the patron.

Definition 4 (Acceptable Specifications): The *acceptable specifications*, denoted B, is the set of all possible specifications that describe a behaviour of a program that would solve the problem at hand in an acceptable way.

We use the term *acceptable* since there will always be a trade-off for the resources we have at hand. A basic goal with a development process must be to get enough information about the set of ideal specifications that are acceptable so that we can write a concrete specification that is close enough to the ideal for our purpose and the resources we have at hand. Formally we write:

Development goal 1: Get to know $B \subset \hat{B}$ enough so that you can write a concrete specification $\overline{B} \in B$.

[1]Throughout this paper we use input to refer to the wider notion of stimuli, ie. including both the data and the functions / methods to be called

The ultimate source of information on B must be the patron who has given us the task of developing the program.

2.2. Patrons

Patrons are the owners of the problems that motivate the development process. At the highest level they are some outside agent (customer, boss, contractor etc) which needs a program to solve their problem. However, development processes often have multiple levels and on lower levels the patron might be the developer herself. As an example this would be the case if the developer has identified a sub-component that is needed to solve the customers problem. Even though the ultimate adjudicator of acceptable behaviour is the customer, he might not know how the sub-component must behave. Thus on this lower level of development the developer might be her own patron.

One way to learn about the acceptable specifications is by engaging in Q&A-sessions with the patron. The type of questions that can be answered by the patron range from basic to very complex. The most basic patron can only distinguish invalid from valid invocations and valid from invalid program behaviour. Formally

Definition 5 (Least Knowledgeable Patron): The *least knowledgeable patron*, denoted LKP, is the patron that can only answer questions of the form ’Given a program invocation $\{\sigma, \iota, o\}$ what is χ?’.

At the other end of the scale we could think of an patron that would give a full acceptable specification if we simply ask her.

Definition 6 (Fully Knowledgeable Patron): The *fully knowledgeable patron*, denoted FKP, can give a full acceptable specification, with a complete and consistent list of behavioural requirements, if we simply ask.

The LKP requires a process of interactive specification, where the developer generates questions based on its current knowledge and then updates its knowledge based on the patrons answers. The FKP does not need any questions at all; she can give a complete behavioural specification without any knowledge from the developer.

In practice most patrons are typically between these two endpoints and can give different types of answers depending on the particular invocation. Often they can only reply to direct questions but sometimes give answers for a whole class of invocations. One common patron can give expected outputs when given a state and input.

Definition 7 (Expected Output Patron): The *expected output patron*, denoted EOP, answers with χ and o when asked about the program behaviour for $\{\sigma, \iota\}$.

Patron interaction is even more problematic in practice. The patron might not have the same framework as the developers so might not understand or know the answer to a question. The developer might have a different picture of the problem than the patron leading to misunderstanding and confusion. In this theory we do not model this although we note it as important and a topic for further study.

2.3. Programs

The final product of any software development process is a program. It consists of source code in one or several programming languages that can be compiled to a form that a computer can execute.

In a theory we do not want to care about the details of the programming language or about the form that is needed of the final program so that a computer can understand it. We simply note that a program must be one particular instance in the space of all possible programs. For a particular development problem there is also a set of programs that all show acceptable behaviour.

Definition 8 (Program Space): The *program space*, denoted $\hat{\Pi}$, is the set of all possible computer programs.

Definition 9 (Program): A *program*, denoted π, is a finite list of program instructions that can be executed by a computer.

Definition 10 (Acceptable Programs): The *acceptable programs*, denoted Π, is the set of all possible programs that have a behaviour that would solve the problem at hand in an acceptable way.

In the same way as for specifications we have the obvious goal:

Development goal 2: Get to know $\Pi \subset \hat{\Pi}$ enough so that you can write a concrete program $\overline{\Pi} \in \Pi$.

2.4. The Developer

No theory for a development process can be complete without modeling the developer. We need ways to represent the knowledge she has gained about different parts of the problem and to represent any other knowledge she has that can be put to use during development.

A basic observation is that the developer cannot directly access either B, the acceptable specifications, or Π, the programs showing acceptable behaviour. At any step during the development process she has some knowledge about these sets. This knowledge typically differs in many ways from the ideals she is trying to achieve. Below we formalize this.

> **Definition 11 (Developer's Specification Knowledge at step t):** The *developer's specification knowledge*, denoted $\tilde{B}_i$, is any information about the acceptable specifications, B, that a developer has at development step *i*.

For human developers $\tilde{B}_i$ is the mental picture or idea that the developer has about the acceptable specifications at a certain development step.

Similarly we have for the program space:

> **Definition 12 (Developer's Program Knowledge at step t):** The *developer's program knowledge,* denoted $\tilde{\Pi}_i$, is any information about the acceptable programs, Π, that a developer has at development step *i*.

Informally a step is an atomic action that drives the development forward. To analyze what different steps are possible we also need to define different parts of developer experience.

> **Definition 13 (Developer's Experience):** The *developer's experience*, denoted $\tilde{K}_{dev}$, is any information that the developer has and can brought to bear to help solve the problem at hand.

When such experience relates to a particular type of development artefact we note the artefact with a superscript. $\tilde{K}_{dev}^{spec}$ is any prior knowledge the developer has on how to write specifications, while $\tilde{K}_{dev}^{prog}$ is the knowledge pertaining to writing programs.

2.5. The Library

The library represents the total knowledge that humanity and its machines have accumulated that are relevant to software development processes and that is explicitly available. We must require that the knowledge has been written down in some form since it is of no use if it sits within the head of another developer without any way to access it.

> **Definition 14 (Humanity's Total Experience):** The *humanity's total experience*, denoted K_{total}, is any information that humanity has previously acquired and that can be brought to bear to help solve the problem at hand.

In the same way as with developer knowledge we note the artefact that the experience relates to with a superscript. $\tilde{K}_{total}^{spec}$ for example is any developers prior knowledge on how to define and write specifications, while $\tilde{K}^{prog}_{total}$ is the knowledge pertaining to programs. $\overline{K}_{total}$ would be knowledge written down in a form that other developers can understand.

In fact K_{total} is primarily books and snippets of code for testing and programs that we can look up and use since it is hard to tap into the parts of if that is in the heads of other developers. But there is nothing stopping that the library also has a number of experienced developers that one can ask questions pertaining to the development problem at hand. However, we think formalizing the knowledge in $\tilde{K}_{total}$ so that it can be put to direct use in development projects is an important goal for the software engineering community as a whole.

With the basic elements in place we can now relate them in a model.

3. An Aggregate Model for Software Development

The basic observation on which our theory is built is that a software development process is a learning process in which the developer needs to refine her internal models[1] of the acceptable specifications and the programs showing such behaviour to the point where she can clearly formulate this knowledge in executable form. It is not simply a matter of formalizing existing knowledge. It is about first acquiring that knowledge and then formalizing it.

For both specifications and programs there are three main entities to keep in mind: the ideal set, the internal model of the ideal, and the concrete artefact. In the space of all possible entities $\hat{M}$ there is a sought-for set of ideal enitites M. Through its internal models of M denoted $\tilde{M}$ the developer wants to write a concrete entity $\overline{M}$ that as the development process goes on gets closer and closer to the ideals.

From this abstract picture we can see that there are two basic ways to make progress. The developer can either make $\overline{M}$ better reflect $\tilde{M}$ or she can make $\tilde{M}$ better reflect M. In some sense, the latter is harder and more fundamental. If there is a discrepancy between $\tilde{M}$ and M the goal we are striving for when writing $\overline{M}$ is wrong. We cannot identify such a situation without extending our view of the goal, ie. refining $\tilde{M}$. This needs an outside force that cannot be found when operating[2] inside $\tilde{M}$. Finding discrepancies between $\overline{M}$ and $\tilde{M}$ is easier since it is a matter of reading $\overline{M}$ and comparing it to the internal model.

Lets now incorporate the other elements into this picture. The patron is the adjudicator of what is and is not inside B. To clarify this division we ask her questions about

[1]For human developers we could say mental models, pictures or ideas

[2]For human developers we could say thinking

(sets of) invocations and she classifies the behaviour. Based on such Q&A-sessions with the patron the developer refines $\tilde{B}$. Based on this model she can refine $\overline{B}$ and her model of what acceptable programs must look like, $\tilde{\Pi}$. This model is used to write a concrete program, $\overline{\Pi}$.

The internal models of acceptable specifications and programs depend on the developers previous knowledge. Some of this knowledge may be explicit in the sense that the developer can write it down in a formal way ($\overline{K}_{dev}$). Some of it is tacit knowledge, distilled through previous experience ($\tilde{K}_{dev}$).

The developer can also tap on some of the total knowledge available in the library. By default this knowledge is formal in the sense that it has been written down ($\overline{K}_{total}$). However, the developer might also be able to tap into tacit portions of K_{total} by, for example, asking fellow developers.

From the patrons point of view each development project should include the best practices in K_{total} that are relevant to this problem. An over-arching goal for the software development community as a whole should be to support that.

The developers previous knowledge is a bag of models that affects his thinking. He can widen this knowledge by going to the library and seeking information that is relevant to the problem at hand. By working with the concrete specification and program, and by questioning the patron she can refine her internal models of the ideal specifications and programs. This in turn helps refine the actual artefacts and another round starts.

With this model we can define what we mean by a software development process.

Definition 15 (Software Development Process): A *software development process* is a sequence of steps in which a developer refines $\tilde{B}$ and $\tilde{\Pi}$ to reflect B and Π faithfully enough so that she can write down $\overline{B} \in B$ and $\overline{\Pi} \in \Pi$.

From this definition we see that a development process is a process of incremental learning. No matter what previous experience the developer has he must learn what parts and how to apply his knowledge in this particular case.

To see what propels this process of learning forward we need to see what can happen in each development step.

3.1. Development steps

There are two main ways in which a development process can make progress: refining an internal model or refining an artefact based on an internal model.

Both these refinement processes have two main components: identifying a *discrepancy* and overcoming it. We call the event that makes the developer identify a discrepancy a *trigger.* When the discrepancy has been identified and described the developer decides to take some action to overcome the discrepancy. We call the latter the *remedy* and use the three-part model, of trigger, discrepancy and remedy to describe important development steps.

Triggers are numerous[1]. A common trigger is to do a review of an artefact. By reading the code the developer confronts it with his internal model and can note omissions, inconsistencies and codification faults. Reviewing the specification has similar benefits but with the additional possibility of including the patron during the review. If that is practically possible it is often very valuable since the developers model will confront the patron's view of the system.

Another source for triggers is to use check lists. A check list codifies knowledge that have been important in the past. Each developer has their own internal check lists with common gripes and problematic points. However, writing them down has the benefit that the developer will not forget important findings in later projects. It also has the benefit that check lists can more easily be communicated to others. As an example a developer could get check lists from the 'library' and use any additional items not on their own list when doing a review.

A third common trigger is to write and conduct tests of the system. Since tests concern themselves with the dynamic behaviour of the system they are a natural link between the program and the behavioural specification. When the developer writes a new test she considers how the program behaves in a new situation. Even though she may not know the expected behaviour, the test is valuable since it can be run and the program's output considered. Considering whether a certain output is valid might be simpler than writing down the expected output since the latter requires creativity while the latter is a matter of classification. And if the developer realizes she cannot classify the behaviour of the program she can note that she should ask the patron to help her.

4. Related work

Exploratory programming (EP) focus on developing a prototype to explore the problem domain and to discover good solution strategies [6]. The prototype is then refined until it performs in an adequeate way. The approach is suitable for projects where a detailed requirements specification cannot be written or for research where it is not clear if a solution even exists [6]. The model has some similarities to our model by not forcing developers into writing a specification up-front. However, our model

[1]We do not consider the trivial trigger of a TODO list listing parts of the program or specification that we explicitly know of but have not yet written down.

highlights the importance of the specification and promotes that properties that the program must have are written down as they are found.

Extreme programming (XP) was recently proposed as a lightweight software development process and it has attracted much interest [1]. It is a package of several practices and ideas that in isolation are not new but taken together form a different approach to software engineering [3]. XP focuses on the importance of writing automatically executable tests with tools that support unit testing. In for example JUnit, a tool supporting the writing of XP-style unit tests for and in Java, you specify how to setup and execute the test but also the expected results [2]. Tests written in such a way would be behavioural requirements in our model and considered part of the specification.

An important element of XP is the use of pair programming in which two programmers together program at the same terminal. One takes the driving role and writes on the keyboard while the other 'looks over the shoulder' and thinks about strategic issues. Developers frequently change roles. Both anecdotal evidence and experimental research have shown that pair programing leads to higher productivity and higher programmer satisfaction [8]. There are possibly many reasons for this effectiveness but we note that one possible explanation is that the models of the developers are constantly rubbed against each other so that they are questioned and discrepancies identified. Our model supports this explanation by highlighting the importance of having some external force that triggers the questioning and refinement of internal models.

In his PhD thesis, Andrew Walenstein presents a framework for understanding software engineering tools in terms of how they support the cognitive processes of the developer [7]. His core ideas is that cognition can be usefully modeled as computation and that the cognitive support offered by the tool is the computational advantage it provides. The tool together with the developer is seen as a distributed cognitive system. The stance taken by Walenstein is similar to ours, in that the focus is on the developer and supporting his cognitive processes. However, Walenstein does not place the same emphasis on the possibility of tools to be creative and help the developer 'think outside the box'. In particluar his qualitative theory of cognitive support lists four prinicples of computational advantage: task reduction, algorithmic optimization, distribution and specialization. Even though several of them applies in some sense to the creative abilities of a tool (the tool creates a test invocation so reduces the number of invocations that a developer needs to create) none of them captures it in its entirety.

The capturing and codification of knowledge on software development so that it can be used automatically has been adressed in the past. An example is the CREATOR2 system for automatic software design built by Koono et al [4] which uses an expert system with knowledge about different deisgns to help in designing switching software. At the core the system is based on a unified representation scheme for modeling the design process and the design product and using multiple strategies in applying the knowledge. Even though the authors does not explicitly mention it the knowledge in the system could be exchanged between different sites and thus be a candidate to put in a library.

5. Discussion

5.1. On assumptions

An assumption in our model is that the ideal artefacts are static in the sense that for a certain development project there exists a set of ideal solutions. Over time the environment where the system is used might change as might the user requirements. This will lead to a possible change in the ideal artefacts. This would complicate the task of the developer since he is now targeting a moving target. We do not model this but note it as an important topic for refined models.

5.2. On why tests are not elements of the model

Tests are not as fundamental as specifications and programs. It is thinkable that development processes exist where there is no need for tests. However, in practice any development problem of interesting complexity will have a non-empty set of tests. The reason is that while a program is a static artefact, a specification states its dynamic behaviour. Tests are needed to bridge the static domain of programs to the dynamic domain of specifications.

We have chosen not to include the tests since there are other possible verification and validation activities and it is not clear on what grounds some should be included but not others. For example, inspections is a fundamental way to identify discrepancies between an artefact and our model.

However, tests have a special place by being so closely related to specifications. Tests are incomplete atomic behavioural requirements. They are invocations that lack values for the output and classification of the behaviour shown by the invocation. A test can be written $\{\sigma, \iota, ?, ?\}$ and a test run as $\{\sigma, \iota, o, ?\}$ and by filling in the missing information we get a full requirement.

5.3. On why the specification needs to be written down

There could be situations where it is not needed to write down $\overline{B}$ but the patron would take a risk. If the developer is no longer tied to the patron the knowledge about B is lost. Also opportunities for automation and tool support is lost if it is not written down.

5.4. On implications for tool support

Writing down a $\overline{B}$ that covers all of $\tilde{B}$ can be hard since not all parts of $\tilde{B}$ is conscious. There are parts of our models that are implicit and assumptions we make without thinking about them. However, when shown a particular invocation of a program few developers have a problem classifying it as valid or invalid. If they cannot say if it is valid or invalid it has pointed to a lack of knowledge. An opportunity for a tool to support the incremental learning would be to search for tests that show the current behaviour of the program in novel situations.

5.5. On why there is only a single developer?

We talk about our developer in singularis although software is often developed in teams. Even if the learning process would be more complicated for a team, we see no reason that the principles involved would be different. It would only mean that our developer could be more competent (even though that is not self-evident and might depend on the task and individual developers). Multiple developers have a larger pool of knowledge and experience to draw from and can thus, as a group, make more informed decisions. There is also the possibility of parallelizing tasks which could lead to productivity boosts. However, this assumes that the increased communication burden does not lead to misunderstandings and problems.

5.6. On the connections to Software Engineering

A common factor in definitions of Software Engineering is that they are concerned with teams of developers and includes issues both of technical and non-technical nature. For example it is important to have good documentation. Our model does not deal with documentation and neither with maintenance, user interfaces or any of the many other sub-fields of Software Engineering. We simply note that a fundamental problem in any software development project is to develop a program that shows acceptable behaviour. In relation to this problem many other issues studied in Software Engineering are, although important, secondary.

5.7. On whether knowledge can be separated into different parts

When modeling the user we separate out different parts of his experience such as $\tilde{K}_{dev}^{spec}$ and $\tilde{K}_{dev}^{prog}$. Is that not a gross simplification with many hidden assumptions about the form and structure of human knowledge?

In a sense yes. But we are not claiming that it is plausible to assume that human knowledge can be separated into independent units of information pertaining to different aspects of the world. We simply identify that different parts of human knowledge are relevant to different questions. By collecting that together and giving it a name we can then reason with it. So essentially the actual form and structure of human knowledge is not an issue for the theory. This is similar to the argument in [5] about the ability to delineate a model's boundaries.

6. Conclusions

A theory for software development was presented built from the five elements fundamental to any development process: a *patron* which has a need for a program or program component, a *specification* which states how the program should and should not behave, a program, a *developer* writing it and a *library* containing all of humanities total knowledge relevant to the problem at hand. Based on these elements a software development process is defined as an incremental learning process in which there are two main ways to make progress: refining an internal model of the developer or refining an artefact based on an internal model. Since the former underlies the latter it was identified as more fundamental.

The theory has implications for tools supporting software development. They should trigger the identification of discrepancies between the internal models of the developer and the ideal artefacts that would lead to acceptable behaviour. One way for them to do that would be to create novel test invocations for the developer to consider. If the tool is creative in creating these invocations it could help the developer 'think outside the box' and realize his knowledge is incomplete.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[2] JUnit development team. JUnit - A Cook's Tour. Tech. Rep. (2002). URL *http://junit.sourceforge.net/doc/cookstour/cookstour.htm.*

[3] Daniel Karlström. Introducing Extreme Programming - An Experience Report. In *Extreme Programming Conference 2002 (XP2002)*, 2002.

[4] Z. Koono, B. H. Far, T. Takizawa, M. Ohmori, K. Hatae and T. Baba. Software Creation: Implementation and Application of Design Process Knowledge in Automatic Software Design. In *The 5th International Conference on Software Engineering and Knowledge Engineering, San Fransisco Bay, USA*, 1993.

[5] Marvin Minsky. Matter, Mind and Models. Tech. Rep. (1997), MIT's Artificial

Intelligence Laboratory. URL *http://web.media.mit.edu/~minsky/papers/MatterMindModels.html*.

[6] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1994.

[7] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. Ph.D. thesis, Simon Fraser University.

[8] Laurie Williams and Robert Kessler. All I Really Need to Know about Pair Programming I Learned in Kindergarten. *to appear in Communications of the ACM*.

Paper 2.

Robert Feldt. *An Interactive Software Development Workbench based on Biomimetic Algorithms*, Technical Report no. 02-16, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, November 2002. *A condensed version of this report has been submitted to IEEE Transactions on Evolutionary Computation.*

An Interactive Software Development Workbench based on Biomimetic Algorithms

Robert Feldt

Department of Computer Engineering
Chalmers University of Technology
Gothenburg, SWEDEN

November, 2002

Abstract

Based on a theory for software development that focus on the internal models of the developer this paper presents a design for an interactive workbench to support the iterative refinement of developers models. The goal for the workbench is to expose unknown features of the software being developed so that the developer can check if they correspond to his expectations. The workbench employs a biomimetic search system to find tests with novel features. The search system assembles test templates from small pieces of test code and data packaged into a cell. We describe a prototype of the workbench implemented in Ruby and focus on the module used for evolving tests. A case study show that the prototype supports development of tests that are both diverse, complete and have a meaning to the developer. Furthermore, the system can easily be extended by the developer when he comes up with new test strategies.

1. Introduction

Developing software without faults is an important task in our modern society. The effects of faults in software can range from annoying, over costly to fatal. Having tools that support software developers in avoiding and removing faults is thus important.

One of the most expensive phases of software development is testing. Since testing does not directly add any functionality to the software there is a risk that software developers does not prioritize it enough. This is unfortunate since testing is a crucial step in ensuring the dependability of a piece of software. An important goal for a software development workbench should therefore be to support developers in finding and writing good tests and to automate testing processes.

A candidate for automating testing would be evolutionary computation. Evolutionary algorithms (EA) ruthlessly exploit weaknesses in the scaffolding software needed to support the evolutionary process [5]. In an earlier experiment of ours [9, 10] the genetic programming (GP) algorithm revealed a fault in our simulator. By outputting NotANumber at a crucial step in the simulation the evolving programs could get a perfect score and quickly solve the task. In that study, we fixed the fault, re-reviewed the simulation software for similar or other faults and restarted the experiment. In effect, the GP algorithm had helped us debug our software. In this paper we investigate this ability further and design a system that capitalizes on the effect.

Evolutionary algorithms have previously been used to generate test data for software testing [19, 26, 30, 37]. The focus has been on finding test data for structural testing although one study investigated black-box testing [38] and another one searched for test cases for mutation testing [3].

In this paper we propose a system that supports an incremental learning process for writing code and executable properties of that code. The distinghuishing feature is a biomimetic algorithm that can search for new test cases that highlights previously unshown features of the code and the specification. The biomimetic algorithm employs an evolutionary multi-agent system for the search, an artificial chemistry for communication between entities in the system and a fitness evaluation distributed on multiple evaluators that focus on different aspects. The system is interactive in that the developers actions indirectly affect the search.

In section 2 we give a background to software development and testing and present a theory for software development. The theory has implications for tools to support software development and motivates the workbench for interactive software engineering[1] described in section 3. Section 3 also describes the prototype WiseR of this workbench that we have implemented in the programming language Ruby. The exeperiments we have conducted with WiseR are described in section 4. Sections 5, 6 and 7 then summarizes related work, discusses the results and draws conclusions.

2. Software development and testing

Software Engineering (SE) is the *’application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software’* [18]. Much effort in SE has gone into finding good development processes. The most traditional example is the waterfall model with strict separation between requirements analysis, design, implementation and testing. Other processes have abandoned the strict separation between phases since they are often impossible to withhold in practice; during design we realize we have overlooked or underspecified some requirements and during implementation we realize the design is not complete.

A development process that has received much attention lately takes this stance to

[1]The workbench was originally named after ’workbench for interactive software evolution’ but we changed evolution to engineering so as not muddle the different uses of evolution in this paper.

a new level. Extreme Programming (XP), highlights testing as a fundamental activity to ensure quality and puts it in the front seat [4]. Some XP proponents even use the term ’test-first design’. The tests should be one of the major driving forces and developers start each iteration by writing tests for the code that needs to be implemented. The tests thus constitutes executable examples of the requirements on the system.

A natural companion to the test-first ideas of XP is a unit testing framework that supports the writing of tests and automates the execution and result collection of running them. Kent Beck originally developed SUnit for unit testing of Smalltalk classes but a number of similar systems have now been developed for other languages and they are collectively called XUnit [8]. In practice the XUnit frameworks allow the developer to specify concrete inputs for the software under test and to state the expected outputs. Even though they focus on unit testing they are general enough to support integration and system testing.

Although the unit tests written in a XUnit framework constitutes executable examples they are different from formal specifications. Recently there has been efforts to overcome this by marrying the JUnit Java unit testing framework with the formal specification language JML (Java Modeling Language). The contracts written in JML in the form of pre- and post-conditions that must be valid when calling a method on a class are used as test oracles in the unit testing. This alleviates the developer from the task of writing the expected outputs and thus simplifies the task of writing tests. The developer only need to set up the testing context and specify the inputs and call sequence.

The type of testing supported by the XUnit frameworks is called behavioural or black-box testing. It focus on the behavior of the software under test (SUT) and aims to test the responses of the SUT regardless of its implementation. An example of a black-box testing technique is boundary-value testing which locates and probes points around extrema and discontinuities in the input data domain.

Another type of testing is called structural or white-box testing. It focus on the internals of the implementation, often the control flow. The goal is to find tests that give good coverage of the program, ie. executes all statements or paths in the program.

A special type of testing is mutation testing which creates mutants of the progam. The goal is then to devise tests that reveal the mutants. By choosing mutant operations that resemble faults that programmers frequently introduce the hypothesis is that a test set revealing mutants should also be good at revealing real faults.

With this background let us now summarize our view of the software development problem in a theory and state the implications it has for a development workbench. A detailed description of the theory can be found in [11].

Our theory for software development is built from the five elements fundamental to any development process: a *patron* which has a need for a program or program component, a *specification* which states how the program should and should not behave, a program, a *developer* writing it and a *library* containing all of humanities total knowledge relevant to the problem at hand. Based on these elements a software development process is defined as an incremental learning process in which there are two main ways to make progress: refining an internal model of the developer or refining an artefact based on an internal model. Since the former underlies the latter it is more fundamental.

This theory has implications for tools supporting software development. They should trigger the identification of discrepancies between the internal models of the developer and the ideal artefacts that would lead to acceptable behaviour. One way for them to do that would be to create novel test invocations for the developer to consider. If the tool is creative in creating these invocations it could help the developer 'think outside the box' and realize his knowledge is incomplete. Furthermore the tool should support the sharing of recipes for creating novel test invocations. If a system for sharing such recipes was in wide-spread use it could lead to more general progress in the area of software development.

The theory also describes tests as half-baked requirements; they are requirements without a verdict on the actual behavior the program showed. It would be very powerful to have a tool that generated test sequences and then executed the program on them and presented the test and the program behavior to the developer. This would allow the developer to classify the test as valid, invalid or irrelevant and the behavior as correct or wrong. The tool should then generate the code for setting up and checking this behavioral requirement. The tool should also allow the developer to note that a test is important but that they do not know what the expected behavior should be.

It is also important that tests are clearly documented. If a test identifies a fault we should be able to demonstrate it to others. Also, as the software is completed and further evolved to meet additional user requirements we want to be able to re-run previous tests. This regression testing is important since the new additions may affect previous code so that it fails where it previously worked.

It is also important that the tests are in a form that facilitates automation. Tests are typically numerous and it would be too cumbersome to execute them by hand. If they are written in a form that is easily executed this should lower the barrier for the developers to continously run and monitor the progress on the tests. Such a tight 'feedback loop' is what Extreme Programming and other similar, recent development methods prescribe [4, 2].

In all but trivial cases testing cannot be exhaustive; there are far more possible combinations of indata than we have the time to run. Thus, the tests we choose to run should represent different classes of inputs. If the software correctly handles the few examples from an input class it is likely that it will handle all inputs in the class. This partitioning of the inputs into different classes should be visible from the tests to justify why we have chosen this particular set of tests.

Many tests often have the same structure and only the test data differs between them. Our development tools should allow the programmer to express these recurring patterns in a form so that it can be reused in later projects and by others.

3. WISE - a Workbench for Interactive Software Engineering

WISE is a design for an interactive tool supporting software development based on the theory above. It is an integrated environment for developing an executable, behavioral specification and a program that implements it. It also highlights the importance of tests and their close relation to the behavioral specification. WISE searches for tests with properties different from the tests it already knows of. Interesting tests are presented to the developer which can review them and classify them. This interactivity between the tool and the developer is central to the design of WISE.

A prototype of WISE called WiseR (WISE for and in Ruby) has been implemented in the programming language Ruby. It currently focuses on searching for tests although a simple GUI has been implemented to interact with the system.

WISE draws upon biologically inspired ideas and a running WISE system uses several biomimetic algorithms. Before we describe the philosophy behind WISE, its architecture and the WiseR prototype we motivate why biological ideas are used and the biological processes they resemble.

3.1. Biomimetic ideas in WISE

WISE is based on several ideas inspired by biological systems and uses algorithms modeled after nature:

- It is continously active even if no developer is present. It searches for better and more interesting tests or learns how to use the knowledge in the library.
- Few parts of WISE are cast in stone. When there are alternative solutions WISE implements several of them and then dynamically learns which one works best.
- Templates for tests are built from building blocks resembling cells in biological organisms. They have a membrane with ports that can connect to ports on other cells. In this way cells grow into larger clusters showing more complex behavior.
- Test cells interact within a biochemical system where proteins can be released and sensed. Cells communicate both with other cells, other entities in the system and with the outside world via the biochemical system.

- The basic commodity for cells is energy. Cells compete for energy by producing data or test runs. Evaluators probe the chemical system for data or test runs that are novel and give energy to the entities that produced it.

The reasons for this use of biological ideas are manyfold. From a philosophical viewpoint the problems facing a developer have many similarities with the ones that biological systems are facing. They are ill-defined. If they were not there would be no real development task since it is by definition the formalization of a system from loose beginnings.

The problems facing a developer are also dynamic. As she defines some part of the system, her choices affects other, yet un-defined parts of the system. As she learns more the importance of some parts might decrease while other parts becomes more important. Even worse, the target might change as the patron gets a new idea or changes the requirements.

In any development process there is room for multiple different choices. The developer must identify important trade-offs and study how different decisions affect the behavior of the system. A workbench supporting the developer must support this playing with alternatives and exploring differing avenues.

Central to any development process is creativity and innovation. The developer needs to be innovative in finding solutions, refining the specification and writing tests that show conformance. Above all the developer needs to be creative, and ’think outside the box’ to identify the faults in her own internal models.

Even though biomimetic methods may not be the best optimizers, they have an excellent track record when it comes to ill-defined, dynamic, explorative and creative processes. So from a philosophical viewpoint they are natural candidates as building blocks in a development workbench.

An additional reason for the use of biomimetic ideas is that evolutionary algorithms in previous studies have revealed faults in scaffolding code used during evolution.

One example was in one of our earlier studies where a GP algorithm evolved aircraft brake controllers [9, 10]. In this experiment a simulator was used to evaluate the aircraft controllers. The simulator was faulty by not correctly handling exceptional conditions from the controllers. In particular the GP algorithm found that by returning the float value not-a-number (NaN) in a specific state of the simulation it could trick the simulator into achieving its goal in a non-realistic way without expending any energy.

In the experiment above the goal was not to test the simulator. But since the solutions produced by the evolutionary algorithm interacted with the simulator it was in essence tested. The fault in the simulator was not identified automatically but required human analysis. But it was clearly evident from running the evolutionary system that something was not right. Since the exploitation of the fault in the simulator was such an effective means for the EA to reach its goals all solutions in the population soon used the exploit. By tracing a simulation of one of the solutions the fault was easily spotted. This also points to the important interplay between the system and a human in finding and understanding the cause of a fault.

Other EC researchers have had similar experiences although few report on them in the final papers. In a recent paper [5] the EC researcher Peter Bentley says that when you work with evolution you

> *…get a few glimpses of the creativity of evolution through the bugs in your code: the little loopholes that are ruthlessly exploited by evolution to produce unwanted and invalid solutions…. Each result fascinating, and each prevented by the addition of another constraint by the developer. The bugs are never reported in any publication, and yet they point to the true capabilities of evolution.*

In this paper we extend this fault-revealing ability of EA to the testing of general software.

3.2. Goals and design philosophy

The goals for WISE are to

1. find new knowledge about the software under test (SUT), and
2. allow the developer to specify test building blocks, test strategies, and novelty criteria in a flexible way.

While goal 1 is obvious, goal 2 is explicitly stated since it is what makes 1 possible both for the current SUT but primarily for future development activity. The ’flexibility’ in goal 2 means that WISE should limit the form in which the developer can describe the system components as little as possible. It should also make as few assumptions about them as possible. This leads to the sub-goal that WISE must also find new knowledge about the system components since we cannot assume the developer has stated (or knows) all of it.

Central to WISE’s design is to focus on the interaction between the developer and the system. The developer is the ultimate source of knowledge so if the system is in trouble it should inform him. The system should also encourage feedback on its progress. Since testing can never be exhaustive for non-trivial systems we want to find tests that are meaningful.

Another design principle is to avoid making choices about the values of parameters to the components in the system. With choices we bias what can be expressed and limit creativity. Thus when there is a choice of different alternatives WISE implements several alternatives and let the system choose which ones are effective at run time.

3.3. Basic Architecture

The WISE architecture has four main parts: an interface to the developer (UI), a control module that formulates goals and initiates searches, a knowledge base acting as a central repository for information in and about the system, and compute daemons that perform searches.

The UI is centered around the two artefacts that should be the end results of the development process: the behavioral specification and the program. It can also display tests to the developer and allows him to classify them. If he classifies a test and the output from the program as valid the test is transformed to a behavioral requirement and becomes part of the specification. Central to making this work is the need to make things explicit. In as far as possible the artefacts are stated in a form that the workbench can actively use in later steps. Information in comments or ’outside’ the system is a lost oppportunity since the system has less information to base its decisions on[1].

The UI gives the developer access to the knowledge base. The knowledge base is a local version of the library that is part of the theory presented in [11]. In the future we envisage that the knowledge base could be an interface to central libraries on the Internet or directly linking the knowledge bases of for example the developers in a development team.

The Control module is the main initiator of actions in a WISE system. It can take commands from the developer and set up searches on a compute daemon. If the developer does not give any commands it can formulate goals and sub-goals and initiate actions based on them. As an example, if the user has not written or loaded any new code that needs to be tested the Controller can consult the knowledge base and initiate a search for test sequences that creates a certain type of data.

When the controller initiates a search it sends the search description and any information needed for the search off to a compute daemon. To decouple the WISE front-end from the compute daemons this communication is inter-process via a TupleSpace over TCP/IP. This decouples the UI and control module from the compute daemon and allow one WISE front-end to use multiple compute daemons. Even though high performance is not a goal of this study this separation was deemed necessary since many of the biomimetic algorithms are compute-intensive. Including this in the design from the beginning should make things easier later.

[1]Unless this information can be parsed and made useful…

The compute daemons are independent and may work on separate problems handed out by a WISE front-end. the TupleSpace model was chosen since it allows for very flexible communication between nodes [15, 39]. The daemons are not expected to cooperate to solve problems but the simplicity and power of the TupleSpace model does not disallow it. It decouples the WISE module from the number and type of daemons. The TupleSpace provides a noteboard where information can be published and seen by multiple or only some subset of the daemons. It also allows the daemons to publish information that can be seen both by the WISE front-end and by other drones.

The searches in the daemons is done in a dynamically evolving system that builds test templates that adds novel knowledge. The knowledge can be either about the piece of software under test (SUT) or about the search builing blocks and how to assemble them.

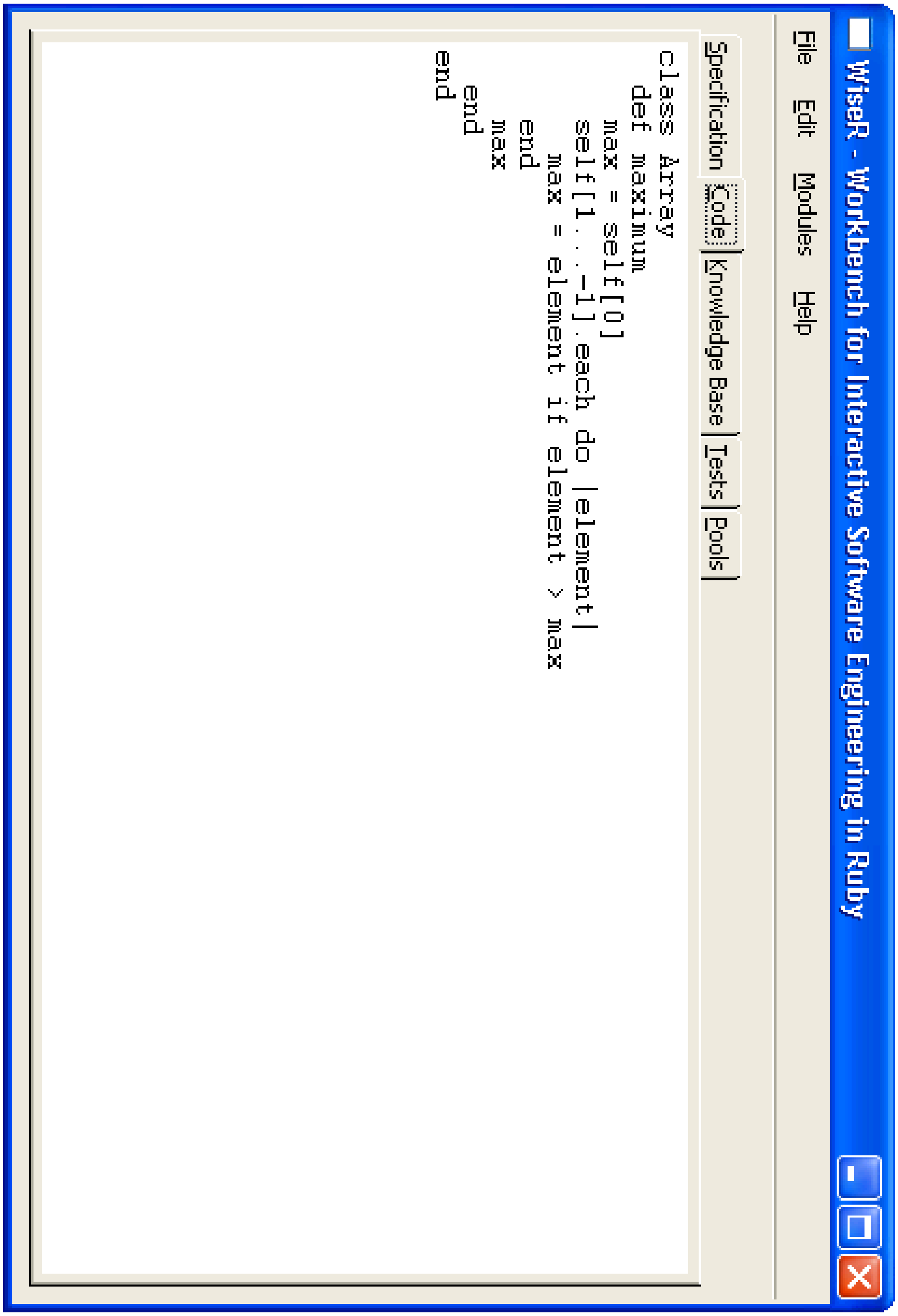
3.4. WiseR - the prototype

A prototype WISE implementation has been implemented in the object-oriented programming language Ruby [36, 29]. It is called WiseR (Wise for Ruby). Ruby was chosen since it is a high-level language with many features that support fast prototyping. It belongs to a new class languages that sports dynamic typing and easy access to all parts of the execution environment. This was deemed necessary in order to allow experimentation with different parts of the system.

Like the popular languages Java and C++ Ruby is object-oriented. Unlike them it is dynamically typed ie. there is no type checking at compile-time. In fact, there is no compile-time since Ruby is not compiled but interpreted[1]. Even though these aspects make Ruby non-typical compared to major languages in use today we do not think it confounds our results. If anything, the dynamic typing makes things harder for a tester. He cannot simply look at the code and see what types are allowed for parameters. In some sense, there is less information available in the code and thus more information needs to be re-discovered. The availability of a compiler would speed up the system, make the client more responsive and allow for more powerful computations. However, in research and for a prototype we think other factors are more important. More detalied information about Ruby can be found in appendix B and in the books [36, 29].

WiseR is both implemented in and supports development of Ruby code. All three artefacts are expressed as Ruby code. Even though this is not a requirement in WISE it makes things easier for our prototype. We can reuse common functionality used in analyzing the artefacts. It is also easier to exchange information between them.

[1]Although Java was originally and is often used as an interpreted language there are numerous compilers available. In any case Java is typed so differs from Ruby in this regard.


WiseR - Workbench for Interactive Software Engineering in Ruby
File
Edit
Modules
Help
Specification
Code
Knowledge Base
Tests
Tools

```
class Array
  def maximum
    max = self[0]
    self[1...-1].each do |element|
      max = element if element > max
    end
    max
  end
end
```

Figure 1. WiseR GUI main window with the Code window active and loaded with the Array#maximum source code

The focus when developing WiseR has been on a module that searches for tests, WiseR-Tests. In addition to WiseR-Tests the system contains of a GUI and small implementations of the control module and a knowledge base. The main window of the WiseR GUI is shown in figure 1.

Before going deeper into WiseR-Tests we briefly describe the WiseR GUI and the support WiseR has for writing specifications.

3.4.1. The WiseR graphical user interface

WiseR’s main window, shown in figure 1, has 5 tab windows for different types of information. The Specification and Code tabs are always present. They are two edit windows for writing the specification and the program, respectively. The ’Knowledge Base’ tab is also always present. It gives an overview of the hierarchical data stored in the knowledge base and allows the developer to change parameters etc.

In the ’Modules’ menu the developer can load a module. After loading the WiseR-Tests module it adds to additional tab windows. The ’Pools’ tab shows all the pools that are currently active and can show some simple statistics for them. The ’Tests’ window is the place where tests found by searches in the pools are reported to the developer.

Examples of how the different tabs look during a run can be found in the case study section below.

3.4.2. Writing specifications in WiseR

Even though the model for software development introduced in [12] and summarized in section 2 above considered requirements as atomic invocations of the program with a state, input stimuli, output and a classification of the invocation, WiseR gives rudimentary support for writing and checking requirements in a more general form.

A specification in WiseR is written as a set of properties in a class inheriting from the class Properties. The superclass Properties add some helper methods for defining the 4 different types of properties supported by WiseR:

- pre - A pre-condition that should hold before a method is called. Gets the arguments to the method as arguments.
- post - A post-condition that should hold after a method is called. Gets the output from the method, the object and the arguments for the call as arguments. Can NOT access the object before the call.
- invariant - A condition that should always hold for objects of the class, both before and after any call.

- raises_exception - Indicates that the method must raise an exception when called. Gets the arguments for the call as arguments.

We see that the first three are simply the ones dictated by Bertrand Meyer in his design by contract method [24, 25]. We added raises_exception since it seemed useful. Not much thought has gone in to this at this stage though and the way specifications are written may have to be updated in the future. For example, the post-conditions in WiseR cannot access attributes of the object before the call was made. This is a rather serious limitation for expressiveness of specifications and needs to be adressed in the future.

All of the three first methods above take a symbol that gives the name of the condition and a block of code that implements the condition. The block must evaluate to true if the condition holds or false if it doesn’t. The latter indicates that the specification is not fulfilled.

The raises_exception property takes a name, the class of the raised exception and a block implementing the condition under which the exception should be raised.

In addition to the four property creating methods above there are two more to indicate which class and method the properties should hold for. They are called for_class and for_method, respectively.

Additionally WiseR can add behavioral atomic requirements as described in section 2 and in paper [12]. They are added in a second class after the properties class at the bottom of the specification.

An example of a full specification is given in the case study section.

3.5. WiseR-Tests

WiseR-Tests is a WiseR module that searches for test templates with novel features. It starts a search by creating a *pool* and filling it with *cells* that are thought relevant to the search. The cells represent recipes for how to create (small) parts of a test template. Cells communicate with each other by sending *proteins* into a *biochemical fluid* in the pool. *Ports* on the cells have protein *sensors* that detect proteins flowing by. If the protein is a product that the port could have use for it saves some of the proteins. When the cell later ’runs’ it can use the protein to search for and connect to the originating cell. As cells connect to more cells and grow into cell clusters they can produce and emit more complex products.

Cells in the pool compete for energy. Without energy a cell hibernates. Cells with high energy levels can execute more actions and have a higher chance of growing into mature cell clusters that produce unique products.

The main source of energy is the *evaluators* on the surface of the pool. Like ports they have protein sensors attached to the biochemical fluid of the pool. Different evaluators sense different types of products. When the sensor of an evaluator fires the evaluator examines the product to assess its novelty. If the product is novel the evaluator assigns an energy score to it and boosts the originating cells energy level. An evaluator also communicates with the outside world by updating the knowledge base with the knowledge gained from the analyzed products.

With this high-level description of the main components of WiseR-Tests let us now go into the details on each one of them.

3.5.1. Cells

To construct a test we need an object of the class to be tested, a sequence of calls to the object, input data for each call and a description of what to do with the generated results. Essentially a test is a small program itself, invoking the class under test (CUT) for a specific purpose.

One approach we could take would be to evolve such test cases directly. For example, a genetic programming algorithm could be used to assemble test programs from the syntactical elements of the programming language. However, there are a number of problems with such an approach.

If we evolve source code directly there is no information about the higher level structure of the test. There is only the code. We will have a hard time writing whole test strategies with only the syntactic elements of the programing language. How should we specify what and how things can vary? Strategies are templates for a piece of source code; not for individual code elements.

When developing tests the goal is not only that they should efficiently test the implementation and show its conformance to the specification. They should also justify for humans (developers or ’customers’) that the system has been thouroughly tested. It is hard to see how such a justification could be built at the same time as evolving the code from atomic syntactic elements. There simply is not enough information to describe the semantics of the test in human-understandable terms. A possible solution would be to analyze the evolved test to produce a description of it. This seems very hard and a backward kind of way. Our representation for evolving tests should support descriptions at its core so that one and the same representation can be used both for running the test and generating a description of what it does.

A further problem with an evolutionary process based on low-level syntactical elements would be that it is unclear how it would scale. Evolutionary algorithms are used on increasingly larger problems and the evolved solutions are more complex but there is still a question of how well they will scale to really complex problems. By evolving the tests from higher-level building blocks we increase the likelihood that it will scale.

Finally, it is often the case that several tests have the same structure. Only some constants or input data differs between tests. The representation we choose must support the easy generation of large number of tests having the same structure.

These problems have led us to define a more powerful representation for test building blocks than what is traditionally used in for example Genetic Programming. In fact the design itself is inspired by cells which are the main building blocks of all biological systems.

The basic building block for our tests are cells. Each type of cell is implemented as one Ruby class. When writing a new cell there are two things the developer must do. He must specify the ports of the cell and he must give one method that implements the functionality of the Cell.

To capture the fact that a single cell can often generate a number of variants. Example of variants are a cell OrderingOfElements that can sort the elements in an array. It has two variants: one for sorting ascendingly and one for sorting descendingly. This allows one and the same cell to capture related variants together. By varying a variant specifier when executing the cell we can choose which variant will be chosen.

In addition to variants cells can often generate random examples of a variant. The OrderingOfElements above cannot be randomized[1] but for example a FixnumGen cell for generating Fixnums would have no variants but many different randomizations. Any combination of variants and randomization is possible.

As an example here is the definition of the ArrayGen cell for creating Array’s filled with objects:

```
class ArrayGen < DataGenerator
  semantics "Array of size v(:size) filled with t(:element)"

  out_port :out, {:type => Array}

  in_port :size, {:semantics => "Size of generated arrays",
           :type => "Positive Fixnum or zero"}

  in_port :element, {:semantics => "Elements for array",
              :type => Object}

  def max_num_variants
    [port(:size).max_num_variants, port(:element).max_num_variants]
  end
```

[1]Well it actually can if it takes input from other cells that can be randomized.

```
  def run(token)
    Array.new(e(:size)).map {e(:element)}
  end
end
```

The ArrayGen cell is a data generator and thus inherits from the DataGenerator base cell class. The semantics line gives the semantics of the cell. Then comes the definitions of one out port and two in ports. The ports are named and assigned meta-data that further defines them. Then comes two methods. The max_num_variants method returns an array with the combined number of variants of the cells connected on the ports. The run method is the one that will be called when the cell produces a product. Here it executes the cell connected to port :size, creates an Array of that size and then fills it by repeatedly calling the cell connected to the element port.

One thing to note is how the string given as the semantics for the cell contains references to the cells connected to the respective ports. The 'v' method call will insert the value received on port size while the 't' method returns the type of the objects returned on port element. This simple scheme allows descriptions of the tests to be built. Since its a simple scheme it will not work in more complex situations but it gives a reasonable hint.

In addition to DataGenerators there are also CodeGenerators. CodeGenerators are cells that implement a piece of Ruby code. They can range from simple cell that call a method to complex test strategies.

A major design goal was to allow flexibility for the developer writing cells. As few requirements as possible should restrict how cells can be written and how they work. A cell should be an isolated building block for building a piece of a test. How to assemble the building blocks to create tests should not be presepcified.

This flexibility is acheived by a flexible cell connection process. Cells connect through ports. A port has a type which specifies what other ports it can connect to. A cell can have zero or multiple ports and the number of ports can change dynamically based on what other cells it has already connected to.

Figure 2 shows an example of a cell cluster for testing the Array#maximum[1] method on arrays of Fixnum's[2]. It is built from four cells. SizeOfDataStructure generates common sizes of datastructures such as array's and hash'es. When executed it generates a size and sends it through its out port to the in port named size on the ArrayGen. The ArrayGen cell creates an array of the given size and fills it with elements

[1]In Ruby C#m indicates the instance method named m on an object of class C.

[2]Fixnum is the Ruby class for 31-bit integers

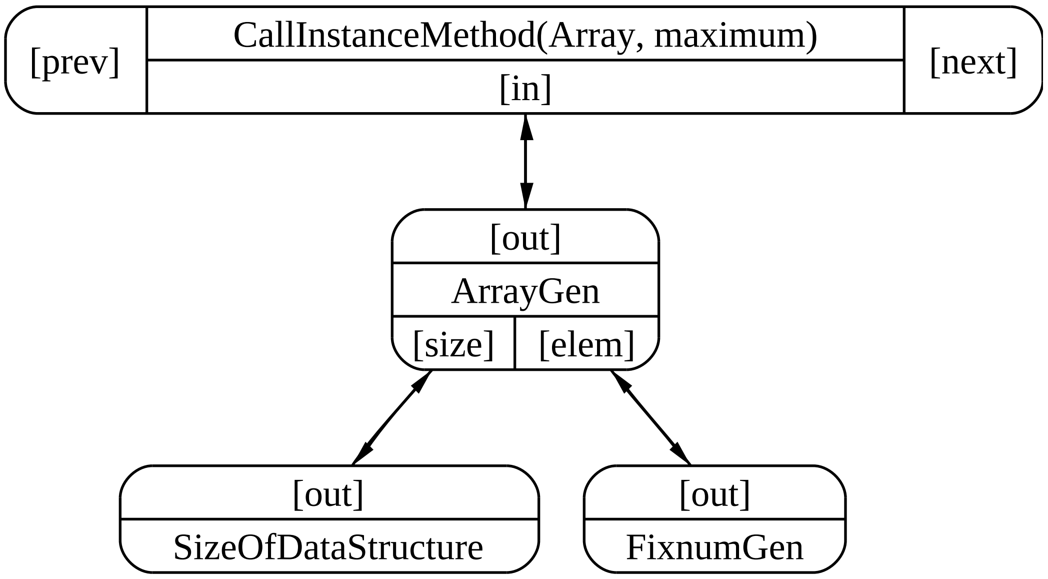


Figure 2. Test cell cluster with four cells for testing the Array#maximum method

generated by the FixnumGen cell. Finally the CallInstanceMethod cell calls the maximum method on the object received through its in port.

The CallInstanceMethod cell in the figure has two additional ports: a prev and a next port. They are used to build more complex tests by linking together several statements.

Cells are active agents in the system and are scheduled to be run depending on how much energy they have. When a cell runs it selects one of a set of cell actions and executes it. Every cell has the same set of cell actions but the parameters that determine the specifics of the action is saved in a genome in the cell. Different cells have different genomes and can thus evolve to perform different actions. In this way we need not explicitly tune parameter values in the system and cells with different functions can evolve different behaviour.

In the WiseR prototype there are currently only 4 different actions:

- Cloner - clones the cell
- Producer - produces a product if we are connected enough to be able to produce something
- Connector - connects to other cells if we have free ports and any of the port sensors have picked up something interesting
- Disconnector - disconnects one or more of the cells in the cluster by breaking up a connection

The actions are implemented as separate Ruby classes inheriting from one and the same base class. This model makes it very easy to experiment with alternative actions; you simply write a new action and hooks it up to the base Cell class. An example would be to implement a crossover action. We have not yet done that since crossover can be said to emerge from the combined actions of a disconnector and a connector.

All actions cost energy based on how long time they execute. A high-resolution timer is used to measure this and deduct the energy level of the cell accordingly.

The cloner cell action is typically activated when the energy of the cell is high. However its activation probability is governed by a constant that is part of the cell genome. Once activated the cloner simply clones the cell and any cells that is connected to it. When a cell is cloned its genome can undergo mutations. After a cell has cloned it must transfer a percentage of its energy to the clone. The percentage is also part of the genome and undergo evolution.

In contrast to the cloner, the producer cell action is activated pretty often given that the cell has matured enough so it can produce a product. A cell is mature if it has only one open port and its either an out port or a left sequence port. The former can produce a Ruby object while the latter can run a test and produce a TestRun product.

When a cells producer action is activated it will create a token and execute itself on the token. The token is passed among the cells during production and records intermediate results, calls to methods and results returned from statements. The token also selects for a certain variant of the variants the cell can produce and specifies the random seed to use for randomizations. Both the variant selector and the random seed are saved and together with a mature cell uniquely determines a test run.

Cell executions are protected in several ways so that invalid connections do not lead to infinite loops or uncaught exceptions. A time out value is used and terminates the cell execution after a constant number of seconds that depends on the speed of the CPU[1]. The cell execution is also invoked within a protected block that will catch any exception. If a test run raises an exception it is used as the return value from the test run. This allows evaluators to check for exceptions and punish the cell clusters that caused them.

When a cell has been executed it packs up the product in a protein, tags it with a unique id and ejects it into the fluid. The process of creating proteins cost energy and that energy is reflected in the concetration of the protein sent out. The amount of energy to spend on the product protein is determined by the cell genome. The amount of energy sent is a sort of gamble the cell makes. By sending out more proteins the potential gain can be higher if an evluator likes the product. However, if they do not the cell will gain no new energy and the energy spent on producing the protein is lost.

[1]WiseR tests the speed of the CPU on startup.

Sending out many proteins also increases the likelihood that other cells i need of the cells product gets the message.

The connector action is pretty straightforward. When activated it checks if any port sensors have detected interesting products that we could have use for. The method used to assess if a product is interesting is a parameter and governed by the genome. One method simply chooses products at random, another ranks the products according to their semantic match with the ports sementics.

The disconnector action is even simpler than the connector. It can be activated by chance but the probability that it is activated raises (by how much is governed by genome) when there are connections in the cluster that cause invalid executions. The disconnector will choose one of the problematic connections randomly and disconnect it.

3.5.2. Ports

Cells connect to each other via ports. A port is a placeholder for a piece of the test coded for by the cell that can vary. Ports are either *connected* to another port or are *free* for new connections. Free ports can be further divided into *open* and *closed*. An open port has to be connected for the cell to be mature. Closed ports does not have to be connected for the cell to be mature. A mature cell can open a closed port so that it can grow into a more complex test template. Cells can add or delete ports dynamically if needed.

There are four common types of ports: in port, out port, left sequence port and right sequence port. In ports are used to receive objects from other cells and out ports are used to send objects to other cells. In ports can only connect to out ports and out ports can only connect to in ports.

Sequence ports are used to chain pieces of code together into more complex templates. Every test template must start with a closed left sequence port; it indicates the first statement of the test. Right and left sequence ports can only connect to each other.

Ports are implemented as a separate class and can be easily extended. This way new type of ports with new semantics can be added.

Ports serve a dual purpose. In addition to being the connection points between cells they are the cells main medium for communication. Cells can send out information through ports and ports have sensors that sense information sent by other entities in the system.

Ports have a simple form of memory. They keep information about which ports on other cells they have been connected to and statistics on what was sent and received by the port during a connection. This information can be exploited by the cell when deciding which ports and cells to connect to. When a cell dies or when the pool is halted the information saved in the ports can be mined and entered into the knowledge base. ’In’ ports are informed by the cell when the information received on the port resulted in an invalid execution of the cell. The ports can thus keep track of which other ports it is fruitful or meaningless to connect to. This way we need not require that the developer specify the exact requirements that needs to be fulfilled for a port to accept a connection.

As an example lets consider the ArrayGen cell from figure 2. Its in port named ’size’ expects a Fixnum value that is larger than equal to 0 since it cannot create arrays with a negative number of elemets. However, since there is no Ruby class for positive Fixnums the type expected on the size port is simply stated to be Fixnum. When WiseR evolves arrays it may happen that NegativeFixnumGen cells connect to the size port. Since they will never result in valid arrays the size port on ArrayGen will be informed of this each time the ArrayGen tries to produce a product. The port saves the type of cell and port it is connected to and calculates an exception rate, ie. the probability that this connection will cause an exception when the cell produces a product. Connections with high exception rates are the primary candidates the Disconnector cell action. Also when the cell dies and the ports are mined for information the knowledge base will be updated. The next time the system runs the knowledge base now shows that it is not a good idea to connect NegativeFixnumGens to the size port on ArrayGen. This will decrease the probability that such a connection happens again.

Often the developer has detailed knowledge about the port and what kind of connections it can accept. He can specify such knowledge as meta-data when adding a port to a cell. It is encouraged that ports at least have meta data describing their semantics ie. their purpose. This allows the selection of candidates for innjection into a pool based on semantic similarity as described above. However semantics are not required. If so the cell will be injected into pools randomly so that the system can learn about the type of connections it can take part in.

3.5.3. Biochemical fluid

Cells communicate by sending proteins into a simulated biochemical fluid surrounding the cells in the pool. Proteins are constructed from a series of acids. An acid is either an ordinary Ruby object or a special acid used when matching proteins. The special acids are:

- DontCareAcid - matches any other acid, used when we don’t care what is in a certain position of the protein

- ClassAcid - wraps a Ruby class, matches any Ruby object that is a kind of the wrapped class, used to match a whole series of proteins that are similar in structure but only differ by the actual object

- MultiClassAcid - same as ClassAcid but matches object that is kind of one of several classes
- NumAcid - encodes a numerical value in the protein, used to indicate the concentration of the protein

The concentration of the protein indicates how many copies of the protein is sent out into the fluid. The implementation is simply to release on protein and have the concentration indicate the number of copies. This saves down on the number of computations the biochemical fluid needs to perform when delivering proteins to sensors.

When a sensor senses a protein there is a reaction that may decrease the concetration of the protein in the fluid. However there are sensor that sense proteins without reducing their concentration. This is for example used to get a notion of time steps in the system. For each iteration of the main pool loop that schedules cells for execution the pool sends out a time chemical in the fluid. Elements in the pool can sense the concentration of the time protein without affecting its concentraion.

It is possible to register reactions with the fluid. This can be used to setup up decaying proteins whose concentration decrease over time. The prototype only supports RateDecayReactions that consume a percentage of the currently available concetration of the protein. They are used for the Frustration protein which is emitted by evaluators when they have seen no progress for a long time. The Frustration protein slowly decays unless more frustration is emitted by evaluators. The level of frustration can be a good indication of the overall progress the search is making. Sensors for frustration can also trigger global pool actions. The prototype does not currently use them but an example would be an Earthquake pool action that triggered on high frustration levels and randomly killed off cells in the pool. After such an earthquake new types of cells could be given more room since previously dominating cells have been killed off.

The fluid is implemented as a simple tuplespace. Sensors register with the fluid object and are organized in a hierarchy depending on the specificity of its matching protein. When proteins are injected into the pool the fluid object look ups the matching sensors and lets them react with the protein in a random order until no more sensors are left or the concetration is zero. So the fluid in the prototype does not model a spatial structure of protein diffusion. It is as if all sensors and emitters where at the same distance from each other and the protein randomly reacted with some of them.

3.5.4. Pool

A pool is a container for cells in different stages of development. It has a biochemical fluid that the cells use to communicate with each other.

Pools allow any entity that has a biochemical sensor or emitter to connect to the fluid. This allows the WiseR front-end to sense the status of the evolution in the pool and injectors to inject new cells into the pool when the proteins indicate that progress is slow. Novelty evaluators connects to the pool to sense products produced by mature cells and to send them energy based on the products novelty.

The pool is the main actor in the system and drives everything by scheduling cells that can execute. The pool keeps a list of the cells ordered from highest to lowest energy. At each time step it randomly select one of the cells from the top 10% of cells having highest energy.

3.5.5. Novelty Evaluators

Novelty evaluators are the main energy sources in the system. They evaluate products produced by the cells and clusters and give them energy if the product is novel.

Novelty evaluators sense products in the fluid with a sensor. The sensor reacts with matching proteins and thus grabs a part of them. If the evaluator likes what they see they will give energy back to the producing cell in accordance with how many proteins they grabbed. Cells that send out much proteins thus have a greater chance of being spotted and getting more energy back.

The actual constants that govern how many proteins an evaluator grab and how much they give back is evolved with an evolutionary strategy algorithm local to the evaluator. This was added to the system since it was hard to set the constants manually and they affected the overall success of the system.

Novelty evaluators was added to the system so that many different criteria for test novelty could be added independently of each other. The approach is similar to [35] and allows multiple criteria and even contradicting criteria to coexist.

Evaluators that evaluate test runs report their findings to the WiseR GUI. The GUI uses the information from the evaluators to sort the tests that are presented to the developer. More novel tests get higher scores and end up higher on the list.

The novelty evaluators that have been implemented for the WiseR prototype are:

- ViolatesPostcondition - violates a post-condition in the specification (only active if there are any postconditions) (10)

- ViolatesInvariant - violates an invariant in the specification (only active if there are any pre-conditions) (10)

- UnseenException - a previously unseen exception was raised when calling a method (9)

- UniqueMethodCalls - the number and order of methods called on the test object differs from previous tests (8)
- UniqueAttributeType - the type of object returned from an attrbiute of the class differs from previous tests (8)
- UniqueReturnTypes - results from methods have different type than previously returned results (5)
- UniqueCellTypes - the number and type of cells in the cluster that produced the test (5)
- UniqueCellConnections - the number and type of connections in the cluster that produced the test (5)
- UniqueAttributeValue - the object returned from an attribute of the class differs from previous tests (4)
- UniqueReturnValues - results from methods have different values than previously returned results (3)
- UniqueParameterNumber - the number of parameters used when calling a method are different than what has been previously used (2)
- UniqueParameterValues - the values of the parameters to a method differs from what has been previously used (1)

The number in parenthesis is an initial weight that the evaluators have. The weight is updated based on which tests the developer investigates and classifies. The weight is used to calculate an aggregate novelty score by summing the individual scores from the evaluators multiplied by their weight.

Many of the evaluators above are of a binary nature and will not change much during a run. However, when they do apply and give high uniqueness scores they ensure that the cell cluster that created the condition gets lots of energy. This promotes the exploration of similar cell clusters since the cells get many changes to run actions that may clone or disconnect cells and connect to other cell combinations.

However the gist of evaluation are the evaluators that evaluate the uniqueness of Ruby objects used as parameters in method calls, returned from method calls and returned from attributes.

These evaluators use distance functions to compare the similarity of Ruby objects. The distance functions for simple objects like numbers is simply the absolute value of their difference. For complex objects the evaluator checks the values of attribute methods on the object. For example for an Array the evaluator would call the length method to compare the lengths of the Arrays. This need not be prespecified since Ruby supplies reflective methods so that the evaluator can dynamically find out which methods are attributes and call them. For objects that can be enumerated (aggrations such as Array's, Hashe's etc) the evaluator will recursively apply itself to compare each subobject.

3.5.6. Feedback from developer interaction

WiseR continously monitors the actions of the developer and uses this information to guide the search. When the developer classifies a test the cells that participated in producing the test gets an energy boost based on how unique the test is according to the evaluators and by counting how many tests with the same classification that are already in the specification.

The weight of evaluators that gave the test a high novelty value are also boosted somewhat.

3.5.7. Controller

The controller in WiseR is very simple. It receives goal statements from the UI, divides them into sub-goals and identifies building blocks that could be useful in searching for a test meeting the goal.

There are only two goal statements supported by the WiseR prototype:

- ’Test method X on class C’
- ’Co-test methods Y1, Y2, …, YN on class C’

The former tells the system to focus on testing one method while the latter indicates that a set of methods should be tested together. One example where the latter goal could be used would be when testing the push and pop methods of a PriorityQueue. With the second type of goal statement above we could tell the system that these methods ’go together’ so its probably a good idea to call them both in a test.

Upon receiving a goal statement the controller checks whether the method(s) to be tested are instance or class methods. An instance method is a method on an instance (object) of a class. A class method is a method on the class itself. The canonical example of a class method is the method used to create instances[1]:

```
# Create an array object (instance of Array) of length 3
a = Array.new(3)
```

[1]In Ruby the ’#’ character indicates that the rest of the line is a comment

If the method under test (MUT) is an instance method the Controller formulates the sub-goal of first creating an instance of the class. So the goal of testing an instance method is broken down into first creating an instance of the class and then testing the method on the instances. After goal analysis the goals enter a goal queue where they are served in turn. The current status and history of the goal queue can be inspected by the developer in the UI.

The controller now takes the next goal from the queue. It first looks in the knowledge base if we already know how to create objects of this type. If not it searches the knowledge base for Cells with meta-data that is relevant to the goal. A cell is deemed relevant if it is known to generate objects of the right type or if its semantics matches the semantics of the goal. The semantic matching is done by a simple heuristic algorithm based on the edit distance between words. The algorithm used is further described in appendix A.

The search for Cells to include as building blocks in the search is repeated in several steps to ensure that the in-ports of previously chosen Cells has some chance to connect to other cells. For example if an ArrayGen Cell has been chosen the controller will in the next round search for Cells with meta-data that is relevant for connecting to the ArrayGen's in-ports.

4. Case Study

In this section we describe a case study that have been carried out on the WiseR system.

4.1. Array#maximum

This experiment is an interactive session with WiseR on a simple example, the Array#maximum method used throughout this paper. The example is somewhat contrived since the implementation of Array#maximum contain a bug on purpose in order to show the workings of the system.

4.1.1. Experimental set-up

For this experiment we start with WiseR in a clean state, ie. it has no knowledge except for the standard cells that come with the basic WiseR system. The cells in such a basic system contains data generators for the common Ruby classes, different ways to call methods with different number of parameters, many different cells for generating boundary cases of arrays since they are so common in Ruby programs and cells to multiplex between datagenerators.

4.1.2. The experiment

The developer needs a method on the Array class that gives the maximum object of all the objects in the Array. He starts WiseR and chooses the specification window. He writes a few properties that the method must obey:

```
class ArrayMaximumProperties < Properties
  in_class Array

  for_method :maximum

  # maximum must be element of array
  post :max_is_an_element do |out, ary|
    ary.include? out
  end

  # maximum must be larger than equal all the elements
  post :max_is_larger_than_equal do |out, ary|
    ary.all? {|element| out >= element}
  end
end
```

He goes on to the Code window to create an implementation and writes:

```
class Array
  def maximum
    max = self[0]
    self[1...-1].each do |element|
      max = element if element > max
    end
    max
  end
end
```

The algorithm simply loops over the elements and keeps the max element in a variable and then returns it. Note that there is an error in the code for the range specifying which elements to loop over (-1 refers to the last element in the array and a…b to the range from a up to but NOT including b). This was introduced for the sake of the experiment.

As soon as he has entered a valid Ruby program (ie. that parses ok), WiseR initiates a search by creating a pool and injecting cells into it. The cells are chosen based on their semantics as described earlier.

As test runs are found by the pool they show up in the Tests window. The developer goes there to check on the findings. Listed on the top is the entry ’Raises NameError: undefined method each for nil’. This indicates that the maximum method has raised an exception. By clicking on the entry it expands to show different test runs for which this happens. By double-clicking on one of them it can be viewed in the lower window. Figure 3 is a screen capture of the WiseR window at this point.

The developer views the source code for the test. It shows the test code and the output from the method in the comment. He realizes he has forgotten about the boundary case of an array of size 0. He decides that the method should return the nil object when called on an empty array since this is the standard way in Ruby to handle empty arrays. He updates the max_is_an_element to

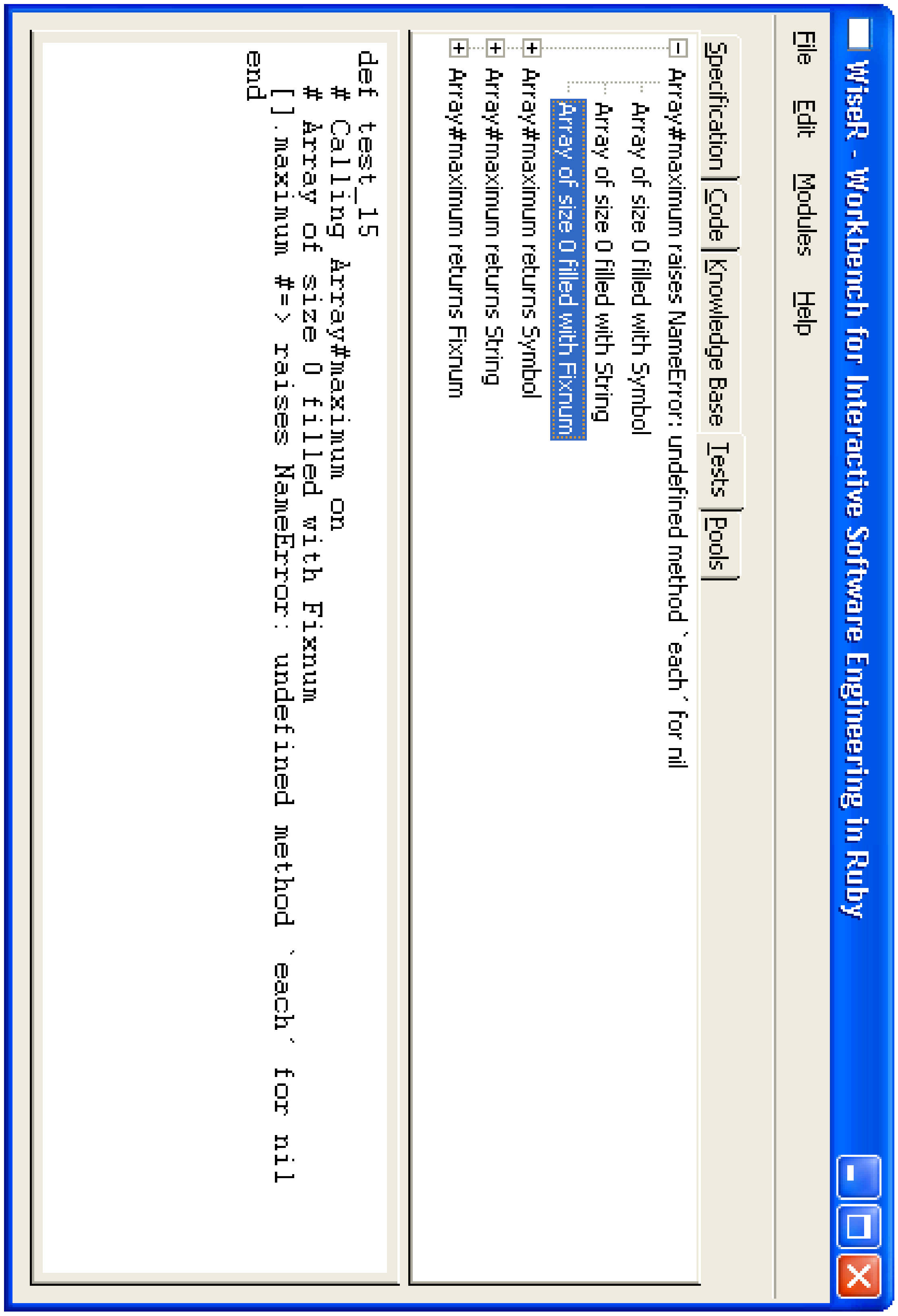
```
# maximum must be element of array or nil if array empty
  post :max_is_an_element_or_nil do |out, ary|
    if ary.length > 0
      ary.include? out
    else
      out == nil
    end
  end
```

and adds a new first statement in the implementation that returns nil if the length is zero. When the program or specification is updated all the evolved tests are reexecuted and the window with tests redrawn.

The previously selected test now returns nil which is the required behavior so the developer classifies it as being a valid test run. WiseR converts the test into a requirement and adds it to the specification after the ArrayMaximumProperties class in the specification window:

```
class ArrayMaximumSpec < Specification
    def req_1
      # Calling Array#maximum on
      # Array of size 0 filled with Fixnum
      # Array of size 0 filled with String
      # Array of size 0 filled with Symbol
      assert_equal(nil, [].maximum)
    end
  end
```

We see that WiseR not only added the test we choose that had Fixnum’s as elements but also the tests that give the same code but are generated in other ways. However, the algorithm WiseR uses for merging tests is simplistic and should be extended. Right


WiseR - Workbench for Interactive Software Engineering in Ruby
File
Edit
Modules
Help
Specification
Code
Knowledge Base
Tests
Tools

- Array#maximum raises NameError: undefined method `each´ for nil
  - Array of size 0 filled with Symbol
  - Array of size 0 filled with String
  - Array of size 0 filled with Fixnum
- Array#maximum returns Symbol
- Array#maximum returns String
- Array#maximum returns Fixnum

```
def test_15
  # Calling Array#maximum on
  # Array of size 0 filled with Fixnum
  [].maximum #=> raises NameError: undefined method `each´ for nil
end
```

Figure 3. The Tests window in WiseR showing the ’Array of size 0 filled with Fixnum’s’ test

now it has only very simple heuristics for how to merge test descriptions that do not work for more complex tests.

At this stage the developer also realizes that the description of this test is not optimal. The test is labeled ’Array of size 0 filled with Fixnum’ but it is not interesting what type of objects are in an empty array. He decides to update the capabilities of the system by adding a specialized test cell for this boundary case. He goes into the knowledge base and clicks down in the hierarchy to WiserTests/Cells/DataGenerators/Array and chooses ’New cell based on…’ from the context menu accessed by right-clicking the mouse. He gets an edit window with the code for ArrayGen except that the ArrayGen name has been removed so he can write a new cell name. He writes ’EmptyArrayGen’ as the name, deletes the two in ports and updates the meta-data and code to execute when running the code. He ends up with (you can compare this to the code for ArrayGen given above):

```
class EmptyArrayGen < DataGenerator
  semantics "Empty Array without any elements (length is 0)"

  out_port :out, {:type => Array}

  def run(token)
    Array.new(0)
  end
end
```

He doesn’t need to define the max_num_variants method since it returns 1 by default as defined in the DataGenerator class.

Going back to the Tests window there are now a new top-level entry ’Array#maximum violates max_is_larger_than_equal_all_elements’. The tests showing this behavior has two Fixnum or String elements. After ensuring himself that the max_is_larger_than_equal_all_elements is a valid property to require and that its implementation is flawless the developer examines the tests in close detail. The one with Fixnum’s look like:

```
def test_64
  # Calling Array#maximum on
  # Array of size 2 filled with Fixnum
  [-196424314, 837355386].maximum #=> -196424314
end
```

Something is obviously not right with the implementation and he goes to review it. Well, the pool has now spotted the problem with the implementation that it never compares to the last element in the array. Before updating the implementation he turns the tests into requirements. He classifies them as valid state and valid input but invalid output so WiseR generates only a skeleton requirement where the developer can fill in the expected output. Here's the requirement above after he has filled in the blank:

```
def req_2
  # Calling Array#maximum on
  # Array of size 2 filled with Fixnum
  assert_equal(837355386, [-196424314, 837355386].maximum)
end
```

He updates the implementation to use the full range 1..-1. Tests are now rerun based on the new implementation and he goes back to the Tests window. On top is now new entries of the form 'Array#maximum raises NameError: undefined method > for nil'. The tests showing this behavior each have two or more elements that are nil or symbols. The developer realises that his implementation will only work for objects that have comparison operators that allow ordering of the elements. He considers adding a property to ensure this but instead decides that raising this exception is the correct behavior in such situations so he classifies the tests as valid and the output as expected. WiseR adds requirements of all the tests. As an example here is the one with 4 symbols:

```
def req_5
  # Calling Array#maximum on
  # Array of size 4 filled with Symbol.
  assert_raises(NameError) {[:vT, :Ho, :ye, :zZ].maximum}
end
```

He also adds the property at the top of the specification:

```
raises_exception :elements_must_be_comparable, NameError do |ary|
  ary.all? {|element| element.kind_of?(Comparable)}
end
```

Going back to the Tests he sees a new type of entry labeled 'Array#maximum raises TypeError: failed to convert Fixnum into String' and examining the test he sees an array with both Fixnum's and String's in it. Aha, not only need they be Comparable they need to be comparable to each other. In a similar way as before he write a new property and converts some tests into requirements.

Going back to the Tests window again he examines some of the 'normal' tests that do not raise an exception or violates any properties. He examines them and turns several of them into requirements by classifying them as valid.

He continues doing this until he feels satisfied there are no more errors.

4.1.3. Analysis

The interactive WiseR session reported above shows that WiseR was effective both in finding bugs in the specification and implementation of the Array#maximum method. The problem with max_is_an_element was an error in the developer internal model of what the method should do. There was a boundary case he hadn’t considered. Seeing the actual test that shows the erroneous behavior gives insight of where the internal model needs to be refined.

5. Related work

5.1. In evolutionary algorithms for testing

There has been a number of studies that use genetic algorithms (GA’s) for structural testing, ie. ensuring that all parts of the implementation are executed by the test set. Jones et al used a GA to generate test-data for branch coverage [19]. They use the control-flow graph (CFG) to guide the search. Loops are unrolled so that the CFG is acyclic. The fitness value is based on the branch value and the branching condition. They evaluated the approach, with good results, on a number of small programs.

Michael and McGraw at RST corporation have developed Gadget - a tool for generating test data that give good coverage of C/C++ code [26]. Gadget work for full C/C++ code and automatically instruments the code to measure the condition/decision coverage. This requires that each branch in the code should be taken and that every condition (atomic part of a control-flow affecting expression) in the code should be true at least once and false at least once. Four different algorithms can be used to search for test data in Gadget: simulated annealing, gradient descent and two different genetic algorithms. One of the GA’s scored the best on a large (2046 LOC) program which is part of an autopilot system but on synthetic programs the GA had problems with programs of high complexity. Simulated annealing fared better here. In all of the experiments random testing fared the worst when the complexity increased.

Pargas et al use a GA to search for test data giving good coverage [30]. They use the control dependence graph instead of the control flow graph since it gives more information on how close to the goal node an execution was. Their system uses the original test suite developed for the SUT as the seed for the GA since it should cover the programs requirements. To reduce the execution time their system employs multiple processors. They compare their system to random testing on six small C programs. For the smallest programs there is no difference but for the three largest programs the GA-based method outperforms random testing.

Tracey et al presents a framework for test-data generation based on optimisation algortihms for structural testing [37]. It is similar to both Jones et al and Michael and

McGraw approaches and uses a CFG and branch condition distance functions. They use both simulated annealing and a genetic algorithm for the optimisation. Their tool is automated and works with ADA code.

Tracey have used a similar technique for functional (black-box) testing [38]. The formal specification is described with pre- and post-conditions that each function must obey. The goal is to find indata that will fullfill the pre-condition and the negated post-condition. These expressions are converted to disjunctive normal form. All pairs of single disjuncts from pre- and post-conditions are considered targets for the search since a fault is found when either of them is fulfilled.

Mueller and Wegener used an evolutionary algorithm to find bounds for the execution time of real-time programs and compared it to static analysis of the software [27]. Even though the evolutionary algorithm cannot give any safe timing garantuees it is universally applicable and only requires knowledge about the programs interface. Static analysis can give garantuees but only in a theoretical world. It needs extensive knowledge about the actual hardware if we are to trust the results. Such knowledge may not always be available.

Baudry et al have used genetic algorithms for evolving test sequences for mutation testing of Eiffel programs [3]. Their model is similar to ours in that they focus on specification, implementation and tests. Their specifications are written with pre- and post-conditions and invariant. A tool mutates the programs and a genetic algorithm searches for test sequences that kills the mutants. The GA is seeded with test sequences written by the developer. Mutants that are not killed by the GA are analyzed by hand to see if they are mutants that did not change the workings of the software.

Genetic algorithms have been used to generate test scripts for GUI testing [20]. Even though the tests generated were simple the authors concluded that the GA could test an application in an unexpected, but not purely random way.

5.2. In evolutionary multi-agent systems

Like WiseR the system developed by Krzysztof Socha and Marek Kisiel-Dorohinicki uses multiple entities, called agents, that together explore a multi-objective search landscape [34]. Agents exchange a non-renewable resource called life energy in transactions based on comparing their behavior against a fitness function. The traditional evolutionary processes of selection and inheritance are not governed by some central authority but happen locally in each agent. When agents have high energy they will reproduce and when energy goes low they die. Agents have a physical location in the world they are in and all actions happen locally. The system has been applied to optimization of some numeric test functions with promising results.

Our system differs from Socha's and Kisiel-Dorohinicki's by not using locality, having heterogenous agents, a renewable 'energy' and a dynamically changing fitness landscape. The agents in our system are cells of diverse constitution. They exist in a pool without a physical location. Communication thus go to many more other cells. This is needed since there may be fewer receivers and too strong locality might hinder progress. More importantly our system does not try to optimize a static fitness function and the fitness function is not smooth. In these conditions we don’t think it would be possible to have a non-renewable ’energy’ resource.

5.3. In cell-based programming models

Some research groups are studying how programming models inspired by biological systems can be used to build more robust systems. George et al recently introduced such a model where cell programs are automatons containing discrete states and transitions between the states [16]. Cells can sense there immediate neighbourhood and send out chemicals. They can also divide. The authors beleive they will be able to build self-healing software with the model.

The ’Amorphous Computing’ group at MIT studies organizational principles and programming languages for coherent behavior from the cooperation of myriads of unreliable parts [1].

5.4. In evolutionary design systems

The Agency GP system is used to let designers explore the design space of 3D objects [35]. It has a very flexible approach to fitness evaluation where agents evaluating one aspect of fitness can be released into the system and affect fitness evaluation. New agents can be added as needed. The authors claim that this model is well suited for fitness evaluation based on conflicting, non-linear and multi-level requirements. Our model with evaluators is very similar to this agent-based fitness model.

Ian Parmee and colleagues have investigated the use of genetic algorithms for conceptual engineering design [31]. Their research has focused on different ways to allow the designer to guide the multi-objective optimization carried out by the genetic algorithm. They have applied their systems to ’traditional’ engineering disciplines such as aerospace and civil engineering.

5.5. In biochemically inspired system

Lones and Tyrell have proposed a new representation for genetic programming inspired by gene expression and enzymes in the metabolic pathways of cells [22, 23]. The building blocks for the GP algorithm are enzymes containing an activity and a set of specificities. The activity is the function the enzyme encodes and the specificities are templates that determine which other components the enzyme can connect to. Genotypes are sets of enzymes and develop into a program by starting the build process from an output enzyme. The system has been evaluated on the evolution of simple, non-recurrent digital circuits. The WiseR-Tests system shares many similarities with Enzyme GP (EGP). Our cells corresponds to EGP's activities and our ports to EGP's specificities.

Many other researchers also build computational models based on modeling cell communication via chemicals. An overview of different approaches is given in [17] which also presents an aggregated model taking different parts from the earlier models. Their system is very similar to ours in that a blackboard is used for communication between autonomous agents. However, their ultimate goal is to model cells for medical research.

5.6. In software testing

The QuickCheck system by Claessen and Hughes is a tool for automatic specification-based testing of programs written in the functional programming language Haskell [6]. The programmer provides a specification of the program by writing properties that the functions in the program must satisfy. The programmer can also combine simple test data generators into more complex ones. The data generators are then used to generate random data for testing the properties of the specification.

The Ballista system can be used for robustness testing of commercial-off-the-shelf (COTS) components [21]. They use the very simple criterion 'Crash or not?' to determine if the response was valid and thus do not require a behavioral specification. The reason is that specifications are often not available for COTS software. The test sets generated by Ballista are exhaustive based on the data types of parameters to each function. There are generators available for each data type and they return extreme or boundary values. Our approach with data generators is very similar to Ballista's with the exception that we allow multiple generators for each type and generators can be combined by connecting to each other.

5.7. In methods for semi-automated software development

The Programmer's Apprentice (PA) was an attempt to build intelligent assistants to support in requirements analysis, design and implementation of a program [32]. They sought to automate the programming process by applying techniques from Artificial Intelligence. As a step towards that long-term goal they built assistants that could help the developer make intelligent decisions. For example, the implementation assistant allowed a programmer to construct programs by combining algorithmic fragments stored in a library.

PA is similar to WISE in that it allows the developer to bypass the system and directly enter code (or tests in WISE). But PA's way to represent knowledge is different.

It is based on finding and encoding knowledge in pre-specified formats. Restrictions are thus put on how people must enter knowledge about the domain. The knowledge is also represented in a form that is different from the implementation language. In contrast the format used to represent knowledge in WISE is the same as the programming language itself. This makes things easier for developer since they do not need to learn another language. Another difference is that the Programmer's Apprentice system does not concern itself with testing.

The approach taken by PA can be called rule-based. A similar approach is the case-based reasoning approach to automated SE taken by some systems [7]. Like WISE they use some fuzzy measure of similarity to find components from a library that are relevant to a task. Like WISE they also learn by allowing the developer to add knowledge to the system. However, none of them have focused on testing and few of them produces artefacts that can easily be read by humans [7]. Since WISE focuses on knowledge about tests and test sequences are often simpler than the software they test the tasks facing WISE is simpler. However, a possible future work can be to investigate if and how more complex meta-data and matching schemes, as used in case-based SE systems, can be used in WISE.

Scheetz et al used an AI planner to generate test cases from an UML class diagram [33]. The UML diagram needs to be augmented with test-specific information.

6. Discussion and future work

6.1. Discussion

On why we did not compare the biomimetic search algorithm to random search. The interactive aspects of WiseR makes it hard to compare the system to a blind random search. If we compare the system without any developer interaction we the system is crippled and it is hard to draw conclusions about the full power of the system. Such an experiment could shed light on the importance and effect of the developer interaction. However, it is not even clear what should be considered a random search to which we could compare. What amount of information should we allow the random search to have? Should it have access to the port memories showing which ports are valid or invalid to connect to? Should it be allowed to use the semantic matching algorithm to select a set of cells that are promising? It is not clear-cut what the answers should be and the limited time available for this study did not allow us to investigate this any further. It is an important point for future work though.

On why it is fast enough even though Ruby is interpreted. Even though Ruby is interpreted and between 5-100 times slower than compiled C (depending on the type of task) the WiseR system can find tests in reassonable time. One reason is that the system can test very many cell clusters while the developer investigates a single test. Another reason is that much knowledge about potentially good tests is captured in the cells. They are high-level building blocks that have shown to be useful in previous testing efforts.

On how tied the system is to Ruby. Even though our goal has been for a generally useful system the WiseR prototype uses many special features of Ruby that may not be available in other languages. Reflection is used in the evaluators to find way to compare unknown objects. The fact that Ruby is interpreted also helps since we can easily reload tests. But there are not only downsides with going for other languages. A statically typed language would simplify things since the types of data would be known.

On WiseR's complexity. The design of WiseR might appear complex on the surface. Even if there is no absolute way to measure and compare the complexity of software systems we think a major cause for WiseR's apparent complexity is that it utilizes concepts not commonly used in software designs. The fact is that the complete WiseR system, including rudimentary graphical interfaces and the code for the test cells, is about 3800 lines of Ruby code[1]. We do not consider that a major software system.

On the risc of using automated methods to search for tests. There is a clear risc with using automated methods such as the one employed by WiseR for finding tests: we get a false sense of security by seeing the mass of tests that can be fairly easily added. However, if the system does not have the right information to base the search on it only searches a small subset of the space of possible input sequences. We note that this is a potential problem and that further development of WiseR should try to find methods to at least partyl overcome this. As an example the QuickCheck system by Claessen and Hughes can summarize the different input data sequences used in the test to show their disitrbution [6]. Something similar would be of value to WiseR, possibly combined with some way of visualizing these distributions.

6.2. Future work

6.2.1. Further experiments on WiseR-Tests

The case study described in section 4 are limited and on a very small development task. To really gauge the power of the developed system we need to perform more experiments on larger development tasks. Since the developer is such an important element in the WISE philosophy experiments should be carried out with several developers on one and the same development task. This could reveal differences in how developers experience and make use of WiseR's capabilities.

[1]Including comments and blank lines

One way to acheive developer feedback about the system could be to release it as open-source. An intriguing possibility would be to develop WiseR-Tests into a plugin for the new FreeRide Ruby integrated development environment [13].

6.2.2. Extending WiseR-Tests

There are a multitude of things that can be changed within the current WiseR prototype.

The connection between what cells are available in the CellPool and the tournaments in the Arena could be tighter. This might add important feedback that more quickly would steer the evolution to interesting areas of the TestSpace. It would also have the potential of trapping the process in local minima, ie. parts of the TestSpace where not much new knowledge is to be found. Exploring this trade-off might be worthwhile.

Improve the descriptions and process of generating test descriptions from a builder. Even though it currently give valuable information to the developer its a bit awkward and might be improved upon.

The controller is not currently part of the evolutionary process in WiseR. We consider this a drawback. This decision was made because we wanted the goal of the current searches to be visible to the developer. It was not clear how a goal could be formulated from an ongoing evolutionary process. An interesting area for future work would be to have a co- or meta-evolutionary search for CellSource's that could attach to the running search and add new cell material. Much of the scaffolding needed for this is already present with the system of triggers that monitor the knowledge base for when to inject new cell material into the pool.

6.2.3. Additional modules and extending WISE

CodeFaultAnalyser. The WISE system knows when you are correcting the source code and can thus save information about error corrections that you do. By saving the faulty and corrected syntax trees these trees can be analyzed. Over time the system can build a knowledge of your common faults and how to correct them. This information can be coupled with the Tester module to allow strengthening the tests based on mutation analysis. By basing the mutations on the actual faults of the user we can assure they are representative. This would be an excellent basis for mutation-based testing in the spirit of [Baudry et al] but with faults that are relevant for the current developer.

Explicitly representing faults also makes it possible to exchange fault sets between different developers. Thus over time this could lead to a common database of faults and how to solve them. Faults and corrections could also be exchanged over the internet etc.

A basic CodeFaultAnalyser module has been implemented in WiseR. However, it has not yet been integrated with Tester so that they can cooperate to strengthen the tests. Future work should strive to integrate these two modules so that they can use each others knowledge. For example, knowledge about the developers common faults could be used to create mutant code that tests would have to identify as faulty. A new novelty evaluator could thus reward tests that killed (new) mutants.

CellExtractor. There are many possibilities for automating the extraction of test cells, ie. test strategies, test code patterns and test data generators. By analysing existing test suites the system could find recurring patterns that can be extracted into new test cells and used for future test evolution.

Extractors could also insert specific data generators tailored to the implementation at hand. A static analysis of the code to be tested could reveal values that are boundary cases for this particular implementation and thus are likely to reveal new information about the system.

7. Conclusions

Based on the theory of software development proposed in [11] we identified opportunities for a workbench to support the development process. Our design for an integrated software development workbench, WISE, tries to follow the ideas indicated by the theory. It explicitly represents both the artefacts to be produced during development and encourage the encoding of meta-information about testing that can be used to derive meaningful tests.

WISE uses biomimetic algorithms to support the development processes. In particular, WiseR, our first prototype of WISE implemented in the programming language Ruby, evolves test templates that generate tests that add interesting information to the system. A common design theme is flexibility. The developer can continously interact with the automatic evolutionary process to guide it and turn it to interesting areas of the design space.

We have performed an initial case study on WiseR. It shows that WiseR can successfully evolve test sets that are both powerful and meaningful.

Appendix A. Algorithm for calculating semantic similarity

The WiseR prototype uses the following heuristic to compare the semantic similarity of two strings:

1. Divide both strings into it constituent words while dropping any non-alphanumeric characters.

2. Delete the short, ’trivial’ words that tend not to carry much information: a, an, the, in, on, of, and, or.

3. Calculate the edit distance (also called the Levenstein distance [14]) for all pairs of words in the two strings that share a prefix at least of length MIN_PREFIX_LENGTH. Inverse this to get a similarity score.

4. Sort all the similarity scores and sum them with a weight that is the inverse of their rank.

5. For each word that do not share a prefix subtract a MISSING_PENALTY from the similarity score

The MIN_PREFIX_LENGTH and MISSING_PENALTY constants was optimized (off-line) with an evolutionary strategy so that the heuristic above gives values that correspond with common sense on a set of strings. However, no evolution is done online on these parameters.

No doubt there are better algorithms for doing the semantic matching and investigating them could be an important future work. However, the above heuristic is simple and gives an indication of semantic similarity. Since the measure is only used to select building blocks it is not fundamental to the success of the system.

Appendix B. Short introduction to Ruby and its syntax

Since Ruby is not very well known we here gives a brief introduction to it and its syntax. This introduction is heavily based on a paper by Michael Neumann [28].

Ruby is an interpreted, object-oriented programming language. It is similar to both Smalltalk, Perl and Python but the syntax is more like Eiffel, Modula or Ada. Like Smalltalk everything is an object[1], there is a garbage collector, variables don’t have type, there is only single-inheritance and code can be packaged into objects. Ruby’s Perl heritage manifests itself in strong support for text-manipulation using regular expressions and substitution but also iterators. In many regards Ruby is very similar to Python although many consider the object-orientedness to be somewhat purer in Ruby than in Python.

In Ruby you declare a class and a method like:

```
class MyClass
  def my_method
    1
```

[1]There are some exceptions to this but they are not important here

```
  end
end
```

and can now get an instance (object) of the class and call the method with

```
o = MyClass.new
o.my_method         # Returns 1!
```

where everything after the # is a comment.

Ruby is dynamic. All classes are open and at any time you can add new methods to classes[1].

Ruby has an eval method so that Ruby code in strings can be evaluated.

```
eval "1"            # Returns 1!
```

References

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald J. Sussman and Ron Weiss. Amorphous Computing. *Communications of the ACM* **43** (5), 74-82 (2000).

[2] Kent Beck, Dave Thomas, Andy Hunt et al. Agilent Development Manifesto. Tech. Rep. (2001). URL *http://citeseer.nj.nec.com/justice93objectoriented.html.*

[3] Benoit Baudry, Vu Le Hanh, Yves Le Traon. Testing-for-Trust: The Genetic Selection Model Applied to Component Qualification. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, 2000.

[4] Kent Beck. *Extreme Programming Explained*, 1997.

[5] Peter J. Bentley. Fractal Proteins. Tech. Rep. (2002), Dept. of Computer Science, University College London.

[6] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, 2000.

[7] H. Dayani-Fard and J.I. Glasgow and D.A. Lamb. A Study of Semi-Automated

[1]Unless you explicitly freeze them in which case they are not allowed to change at all

Program Construction. Tech. Rep. AI memo 933A (1998), MIT's Artificial Intelligence Laboratory.

[8] Stephane Ducasse. SUnit Explained. Tech. Rep. (2000). URL *http://www.iam.unibe.ch/~ducasse/WebPages/Programmez/OnTheWeb/Eng-Art8-SUnit-V1.pdf*.

[9] Robert Feldt. Generating Diverse Software Versions with Genetic Programming: an Experimental Study. *IEE Proceedings - Software Engineering* **145** (6), 228–236 (December 1998). Special issue on Dependable Computing Systems

[10] Robert Feldt. Genetic Programming as an Explorative Tool in Early Software Development Phases. In Conor Ryan and Jim Buckley, *Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering*, pages 11–20, 1999.

[11] Robert Feldt. A Theory of Software Development. Tech. Rep. (2002), Department of Computer Engineering, Chalmers University of Technology, Gothneburg, Sweden.

[12] Robert Feldt. A Theory of Software Development. Tech. Rep. (November 2002), Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden.

[13] Curt Hibbs, Rich Kilmer et al. *The FreeRide Ruby IDE home page*, 2002. URL *http://www.rubyide.org/cgi-bin/wiki.pl?HomePage.*

[14] Hal Fulton. *The Ruby Way*, 2001.

[15] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems* **7** (1) (1985).

[16] Selvin George, David Evans and Lance Davidson. A Biologically Inspired Programming Model for Self-Healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

[17] P.P. González Pérez, M.C. Garcia, C.G. Garcia and J. Lagunez-Otero. Integration of Computational Techniques for the Modelling of Signal Transduction. Tech. Rep. (2001), Instituto de Quimica, Universidad Nacional Autonoma de Mexico. URL *http://www.cogs.susx.ac.uk/users/carlos/doc/GonzalezEtAl-integration-of-computational-techniques.pdf*.

[18] IEEE Standards Team. IEEE Standard Glossary of Software Engineering Terminology. Tech. Rep. (1990).

[19] B. Jones, H. Sthamer, D. Eyres.. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal* **11** (5), 299–306 (September 1996).

[20] David Kasik and Harry George. Toward Automatic Generation of User Test Scripts. In *Proceedings of the Conf. on Human Factors in Computing Systems: Common Ground*, pages 244-251, 1996.

[21] Nathan P. Kropp and Philip J. Koopman Jr. and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the Fault-Tolerant Computing Symposium*, pages 230-239, 1998.

[22] M.A. Lones and A.M. Tyrrell. Biomimetic Representation in Genetic Programming. In *Proceedings of the Workshop on Computation in Gene Expression at the Genetic and Evolutionary Computation Conference 2001 (GECCO2001)*, 2001.

[23] M.A. Lones and A.M. Tyrrell. Crossover and Bloat in the Functionality Model of Enzyme Genetic Programming. In *Proc. 2002 World Congress on Computational Intelligence.*, 2002.

[24] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[25] Bertrand Meyer. Applying 'Design by Contract'. *IEEE Computer* **25** (10), 40-51 (October 1992).

[26] Christoph C. Michael and Gary McGraw. Automated Software Test Data Generation for Complex Programs. In *Proceedings 13th IEEE Conference in Automated Software Engineering*, pages 136–146. IEEE Computer Society, October 1998. URL *citeseer.nj.nec.com/66954.html*.

[27] F. Mueller and J. Wegener. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. In *IEEE Real Time Technology and Applications Symposium*, 1998.

[28] Michael Neumann. Comparing and Introducing Ruby. Tech. Rep. (February 2002). URL *http://www.s-direktnet.de/homepages/neumann/rb/download_ruby.html*.

[29] Michael Neumann, Robert Feldt, Lyle Johnson, Jonothon Ortiz. *Ruby Developer's Guide.* Syngress, 2002.

[30] Roy P. Pargas and Mary Jean Harrold and Robert Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability* **9** (4), 263–282 (July 1999). URL *citeseer.nj.nec.com/pargas99testdata.html*.

[31] Ian Parmee. *Evolutionary and Adaptive Computing in Engineering Design: The Integration of Adaptive Search Exploration and Optimization with Engineering Design Processes.* Springer Verlag UK, 2000.

[32] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Program Design Scenario. Tech. Rep. AI memo 933A (1987), MIT's Artificial Intelligence Laboratory.

[33] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman and A. Howe. Generating Test Cases from an OO Model with an AI Planning System. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE '99)*, 1999.

[34] Krzysztof Socha and Marek Kisiel-Dorohinicki. Agent-based Evolutionary Multiobjective Optimisation. In *Proceedings of Congress on Evolutionary Computation (CEC'02), Honolulu, HI, USA*, pages 109-114, May 12-17 2002.

[35] Peter Testa, Una-May O'Reilly and Simon Greenwold. AGENCY GP: Agent-Based Genetic Programming for Spatial Exploration. In *Proceedings of the ACSA*, 2002. URL *http://www.ai.mit.edu/projects/emergentDesign/agency-gp/ACSA.html*.

[36] Dave Thomas and Andy Hunt. *Programming Ruby: A Pragmatic Programmer's Guide.* Addison-Wesley, 2000.

[37] N J Tracey and J A Clark and K C Mander and J A McDermid. An Automated Framework for Structural Test-Data Generation. In *Proceedings 13th IEEE Conference in Automated Software Engineering.* IEEE Computer Society, October 1998. URL *http://www.cs.ukc.ac.uk/pubs/1998/974*.

[38] Nigel Tracey and John Clark and Keith Mander. Automated Program Flaw Finding using Simulated Annealing. In *Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–81. ACM SIGSOFT, March 1998. URL *http://www.cs.york.ac.uk/testsig/publications/njt-mar98b.html*.

[39] P. Wyckoff. T Spaces. *IBM Systems Journal* **37** (3) (1998). URL *http://www.research.ibm.com/journal/sj/373/wyckoff.html*.

Part II.

3. Robert Feldt. *An Experiment on Using Genetic Programming to Develop Multiple Diverse Software Variants*, Technical Report no. 98-13, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, September 1998.

4. Robert Feldt. *Genetic Programming as an Explorative Tool in Early Software Development Phases*, Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering, pp. 11-21, Limerick, Ireland, 12th-14th April, 1999.

5. Robert Feldt. *Forcing Software Diversity by Making Diverse Design Decisions - an Experimental Investigation*, Technical Report no. 98-46, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, December 1998.

Paper 3.

Robert Feldt. *An Experiment on Using Genetic Programming to Develop Multiple Diverse Software Variants*, Technical Report no. 98-13, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, September 1998.

This report includes the two previously published papers:

Robert Feldt. *Generating Multiple Diverse Software Versions with Genetic Programming - an Experimental Study*, IEE Proceedings - Software, vol. 145, issue 6, pp. 228-236, December 1998.

Robert Feldt. *Generating Multiple Diverse Software Versions with Genetic Programming,* Proceedings of the 24th EUROMICRO Conference, Workshop on Dependable Computing Systems, pp. 387-396, Västerås, Sweden, August 1998.

An Experiment on Using Genetic Programming to Generate Multiple Software Variants

Robert Feldt

Technical Report 98-13, Department of Computer Engineering, Chalmers University of Technology, Sweden, October 1998.

Abstract

Software fault tolerance schemes often employ multiple software variants developed to meet the same specification. If the variants fail independently of each other, they can be combined to give high levels of reliability. While design diversity is a means to develop these variants, it has been questioned because it increases development costs and because reliability gains are limited by common-mode failures. We propose the use of genetic programming to generate multiple software variants by varying parameters to the genetic programming algorithm. We have developed an environment to generate programs for a controller in an aircraft arrestment system. Eighty programs have been developed and tested on 10000 test cases. The experimental data shows that failure diversity is achieved but for the top performing programs its levels are limited.

1. Introduction

One approach to software fault tolerance employs multiple variants of the same software to mask the effect of faults when a minority of variants fails [Avizienis77]. Design diversity, *i.e.*, several diverse development efforts, has been proposed as a technique for generating these redundant variants. The difference in the programs, which is generated by the different design methods, is called software diversity. The hope is that the diversity in the programs will make them exhibit different failure behavior; they should not fail for the same input and, if they do, they should not fail in the same manner.

There are two main drawbacks with the approach of design diversity: (1) it is not obvious if and how we can guarantee that the programs fail independently and (2) the life cycle cost of the software will likely increase. The original idea of N-variant programming (NVP) opted for the specification of the software to be given to different development teams [Avizienis77]. The teams should independently develop a solution, and this independence between the teams should manifest itself in independent failure behavior. However, software development personnel have similar education and training and use similar thinking, methods and tools. This may lead to common-mode failures, *i.e.*, several variants failing for the same input, and limit the diversity that can be achieved. Experimental research has shown that there are systems for which the independence assumption is not valid [Knight86]. The strength of using design diversity has thus been questioned.

In [Lyu94], the term *random diversity* was proposed to denote the above scenario; generation of diversity is left to chance and arises from differences in background and capabilities of the personnel in the development teams. In contrast to this, they introduced the notion of *enforced diversity*. By listing the known possible sources of diversity and varying them between the different development teams, the software variants can be forced to differ. In [Littlewood89], Littlewood and Miller showed that the probability that two variants developed with different methodologies would fail on the same input is determined by the correlation between the methodologies. The correlation is a theoretical measure of diversity defined over all possible programs and all possible inputs. Littlewood and Millers calculations set the goal for studies into achieving software diversity: find methodologies with small or negative correlation.

A problem in using design diversity is that life cycle costs can increase. Obviously, the development cost will increase; we have to develop N variants instead of one. In addition to this, maintenance costs increase. Each change or extension to the specifications of the software must be implemented, and possibly even redesigned, in each of the diverse variants. The actual cost increases have been estimated to be near N-fold [Hatton97].

This paper introduces a novel approach for developing multiple diverse software variants to the same specification that addresses both the development cost and non-independence problems of design diversity. The proposed approach uses genetic programming (GP) which, according to [Koza92], is a technique for searching spaces of computer programs for individual programs that are highly "fit" in solving (or approximately solving) a problem. GP evolves programs from specified atomic parts and adhering to a basic specified structure. Genetic algorithms model evolutionary processes in nature and are studied under the subject of Evolutionary Computation (see for example [Bäck97]). By varying a number of parameters affecting the development of programs, we can force them to differ.

Section 2 introduces genetic programming and section 3 discusses how it can be used to develop diverse software variants. The experiment that have been performed is described in section 4 followed by the experimental results in section 5. Section 6 evaluates the results. Finally, we conclude and indicate future work.

2. Genetic programming

Genetic algorithms mimic the evolutionary process in nature to find solutions to problems. Genetic programming is a special form of genetic algorithm in which the solution is expressed as a computer program. It is essentially a search algorithm that has shown to be general and effective for a large number of problems.

In the classical view of natural evolution, individuals in a population compete for resources. The most "fit" individuals survive, *i.e.*, they have a higher probability of having offspring in the next generation. This process is modeled in genetic algorithms in which the individuals are objects expressing a certain, often partial or imperfect, solution to the investigated problem. In each generation, each individual is evaluated as to how good a solution it constitutes. Individuals that are good are chosen for the next generation with a higher probability than low-fit individuals. By combining parts of the chosen individuals into new individuals, the algorithm constructs the population of the next generation. Mutation also plays an important part. At random, some parts of an individual are randomly altered. This is a source of new variations in the population.

While a genetic algorithm generally works on data or data structures tailored to the problem at hand, genetic programming works with individuals that are computer programs. This technique was introduced by Koza in [Koza92] and has recently spurred a large body of research [Koza97]. Kozas programs are trees that are interpreted in software but a number of other approaches exist. For example, in [Nordin95] Nordin evolved machine language programs that control a miniature robot.

A number of GP systems are available. To use one of them to solve a particular problem, we must tailor it to the problem. This involves choosing the basic building blocks (called terminals), such as variables and constants, and functions that are to be components of the programs evolved, expressing what are good and bad characteristics of the programs, choosing values for the control parameters of the system and a condition for when to terminate the evolution of programs [Koza92]. The control parameters prescribe, for example, how many individuals are to be in the population, the probability that a program should be mutated and how the initial population of programs should be created.

The major part of tailoring a GP system to a specific problem is to determine a fitness function that evaluates good and bad characteristics of the programs and to develop an environment in which these characteristics can be evaluated. There is no reason to use GP if it is harder to implement an evaluation environment than it is to implement a program solution. However, GP can be used for problems that we can state but for which no solution is known. The fitness function is often implemented via test cases with known good answers. However, the fitness evaluation process is much more general and constitutes any activity taken to evaluate the performance of a program. For example, in [Nordin95], the programs are evaluated in a real robot; the ability of the program to avoid obstacles while keeping the robot moving is evaluated and used as a fitness rank.

2.1 Diversity in genetic programming

The term diversity is used with a special meaning in the Evolutionary Computation (EC) community. If the population contains programs that are different, it is said to be diverse. When there is no diversity left in the population, i.e. all programs look and behave the same, the GP run is said to have *converged* to a solution. This can happen before good solutions to the problem have been found and thus different ways to maintain and enhance the diversity are studied (see for example [Ryan96]). Measuring the diversity in the population is fundamental to this aim.

Several different measures of diversity have been proposed in the EC community and are classified into two different classes: genotypic and phenotypic measures [Banzhaf98]. These classes directly correspond to two of the four characteristics of software diversity listed in [Lyu94]. Genotypic diversity is called *structural diversity* by Lyu et al. and measures structural differences between the programs. Phenotypic diversity is called *failure diversity* by Lyu et al. and measures differences in the failure behavior of the programs.

The phenotypic diversity remaining in the population when the GP run is terminated can be used to enhance the effectiveness of GP. In [Zhang97], Zhang and Joung proposed that a pool of programs, instead of a single one, should be retained from a GP run. The output for a certain input is established by applying the programs in the pool to the input and taking a vote between them to decide the master output, similar to an N-variant system.

Our approach is distinct from the approach of Zhang and Joung, since we propose that diversity from several runs of a GP system should be exploited and that the parameters to the system should be systematically varied to promote diversity.

Our goals are also markedly different from the research on measuring diversity in GP populations. The main goal of such research is to decide whether the run should be stopped because the population has converged [Banzhaf98].

2.2 Parameters to a GP system

In the remainder of this paper, we take a pragmatic view of genetic programming. We consider it a technique for searching a space of programs and view it as a "black box" with three sets of parameters: parameters defining the program space to be searched (program space parameters, PSP), parameters defining details about the search (search parameters, SP) and parameters to the evaluation environment (evaluation parameters, EP).

The program space parameters include parameters defining the terminal and function sets and the structure of the programs. These parameters define a space of all possible programs adhering to the specified structure and applying the specified functions to the specified terminals.

The search parameters affect only the result, *i.e.*, effectiveness, of the searches in the space of programs defined by the program space parameters. Examples of search parameters are the number of programs in the population and the probability that a program should be mutated.

The evaluation parameters define, for example, the number and nature of test cases to be used in evaluation. The strategy for evaluation is also viewed as a parameter. An example of a strategy would be to let the test cases change during evolution to test the programs on difficult input values.

It is worth noting that this black-box view frees us from considering only genetic programming. We can consider other algorithms searching a user-definable program space or other algorithms that generate programs. Possible substitutions for GP could be program induction methods or other machine learning algorithms studied in the area of artificial intelligence. Diversity could be found by varying the algorithm used.

3. Software diversity with genetic programming

The output from a run of a GP system is a population of programs that are solutions to the problem stated in the fitness function implemented in the evaluation environment. The solutions are of differing quality; some programs may solve the problem perfectly, others might not even be near solving a single instance of the problem and in between are programs with differing rates of success. The diversity in this population can be exploited [Zhang97]. However, the amount of diversity available in the population after a GP run will be limited since populations tend to converge to a solution. One way to overcome this might be to rerun the system with the same parameter settings. GP is a stochastic search process, and two runs with the same parameters can produce different results. Diversity might also be achieved by altering parameter values between different runs of the GP system. If we change the search parameters to a GP system, the search might end in different areas of the search space of programs. Furthermore, if we change the program space to be searched by altering the program space parameters, we will get programs using different functions and terminals and adhering to a different structure. Diversity might also be achieved by changing parameters to the evaluation environment. Thus, we propose that diverse software variants are developed by running, re-running and varying parameters to a genetic programming system tailored to the specification for the variant(s).

3.1 Procedure for developing diverse programs with genetic programming

Table 1 outlines the seven different phases in the procedure we propose. We start by developing an environment to evaluate the quality of programs (phase I), *i.e.*, how well they adhere to the requirements stated in the specification. Thus, upon entering phase I, we need to have a specification at hand. Next, we need to choose which parameters to vary, which values to vary them between and which combinations of parameter values to run with the GP system. This is done in phases II, III and IV, respectively. Research is needed to evaluate which parameters most affect the diversity. The principle for choice of values should be to include building blocks, *i.e.*, functions and terminals, which are thought to be needed to develop a solution. Careful consideration must be made so that the diversity is not limited. There are large numbers of parameters to a GP system, and most of them can take multiple values, so the number of combinations of parameter values is vast. We propose that a systematic exploration of these different combinations should be tried. Statistical methods for the design and analysis of experiments, such as for example fractional factorials described in [Box78], is needed to this end.

Phase	Description
I. Evaluation environment	Design a fitness function from the software specification. Implement the fitness function in an evaluation environment.
II. Parameters to vary	Choose which parameters of the GP system and evaluation environment shall be varied.
III. Parameter values	Choose parameter values to vary between.
IV. Parameter combinations	Choose the combinations of parameter values to use in the different runs.
V. Generate programs	Run the GP system for each combination of parameter values.
VI. Test programs	Test the program variants that have been generated. Calculate measures of diversity.
VII. Choose programs	Choose the combination of programs that give the lowest total failure probability for the software fault tolerance structure to be used.

Table 1. Phases of procedure for developing diverse programs by varying parameters to a genetic programming system

In the next phase (phase V), the chosen combinations of parameter values are supplied to the GP system which is run to produce the programs. From each run, the best, several or all of the developed program variants can be kept for later testing. If the program generation is not successful, iteration back to phases II, III and IV may be necessary. Upon leaving phase V, we have a pool of programs. Running a GP system is an automatic process and does not need any human intervention, so the number of programs developed can be large. If we are to use the programs in a specific software fault tolerance scheme, such as an N-variant system, we need to choose which programs in the pool to use (phase VII). Calculating measures of diversity such as the correlation measures in [Littlewood89] or the failure diversity measure in [Lyu94] might be useful in this task. The measures can be calculated from the test data generated in phase VI.

In [Littlewood89], systematic approaches to making design choices when employing design diversity were introduced. If we hypothesize that our choices of parameter values are analogous to these design choices, the findings in [Littlewood89] might be used to choose among the combinations of parameter values. A particular set of design choices is called a design methodology in [Littlewood89] and, if we take our analogy even further, our GP approach would enable us to try a large number of design methodologies in the same setting. However, it is unclear whether the use of GP or a common evaluation environment limits the diversity to be explored such that the variations in design methodologies are only minor. An experiment to evaluate this is described in section 4 below.

In the following, we list sources of diversity when an approach such as NVP is used and identifies which GP parameters relate to these sources. Thereafter, the cost issue of using the proposed GP approach is briefly discussed. Central to the result of applying the described method is that GP can evolve good solutions in the first place. It is not probable that the variants can be used if they fail on a large number of input cases. This issue is further discussed below.

3.2 Comparison of diversity sources

To qualitatively assess the value of the proposed approach, we would like to compare the sources of design diversity with the parameters we can affect in the GP system and what effect on the generated program they might have. Table 2 shows a taxonomy of sources of design diversity and GP parameters that correspond to these sources. The taxonomy is not intended to be complete but covers the most important aspects mentioned in the literature (see for example [Saglietti90] and [Lyu94]). The taxonomy has been carried over from the Software Metrics area [Fenton91]. Our motivation for this is that what can be measured can be varied and what can be varied, and applies to software and its development, is a potential source of diversity. Fenton arrives at this taxonomy by viewing a piece of software as a set of activities (processes) using resources to produce artifacts (products) [Fenton91]. In table 2, a diversity source with leading number 1 is a process, with leading number 2 a product and with leading number 3 a resource.

We stress that making a comparison like this is not easy; it is not clear-cut how an approach such as GP can be compared with more traditional software development techniques.

Source of design diversity	GP counterpart
1.1 Specification process	Same source
1.2 Design process	Choice of allowed structure, functions and terminals
1.3 Implementation process	Program representation, type of GP system
1.4 Testing process	Evaluation strategy
2.1 Specification products	Same source
2.2 Algorithms	Program space parameters
2.3 Data structures	Functions, Terminals
2.4 Implementation language	Program representation

2.5 Test data	Test cases, Testing scheme
3.1 Personnel / Team	No direct counterpart
3.2 Tools	GP System, Diversity in compiler/linker/loader can be similarly exploited

Table 2. Correspondence between sources of diversity in traditional design diversity approaches (based on [Fenton91], [Saglietti90] and [Lyu94]) and our proposed GP approach

Processes. The potential diversity arising from different specification processes and/or types also can be used with the GP approach. The difference is that each specification must be implemented in an evaluation environment. The design and implementation processes have no direct counterpart in GP. With GP, we do not explicitly design the programs; they evolve to meet our specification. However, the task of choosing parameters, their values and combinations to be used in the different runs resembles a high-level design activity. We decide not exactly how the program is to be designed but which major concepts can be used.

The potential diversity from using different implementation processes resembles using different types of GP system with, for example, different program representations. An example would be using function trees to represent the programs in one run and using linear representations in another.

The diversity to be found by different testing schemes has no direct counterpart in GP. However, choosing the number and values for the test cases to use in evaluating the programs relates to testing as well as to test data (point 2.5). For different runs, we might choose to concentrate the test cases in a special region of the input data space. Another parameter that resembles alternating the test process would be to allow the test cases to change dynamically.

Products. We cannot directly specify what algorithms and data structures the GP programs should use. If we were to give two development teams different functions and terminals to use in their program, however, it might affect what algorithm they used to solve their problem. If the same reasoning applies to our GP system, we would expect the algorithm used in the developed programs to differ for runs with different functions and terminals. The same argument applies for the parameter that determines the permitted structure of the programs. If we dictate that a development team cannot use any subroutines or cannot use recursion, that team might not implement a certain algorithm, forcing them to consider other solutions. In GP, we can introduce functions and terminals that give access to certain types of data structures, such as indexed memory, lists or stacks.

Some studies have shown that using different implementation language can give rise to diversity [Lyu94]. The counterpart in GP is the representation language. This could be one of the earlier mentioned function trees or machine instructions. Other examples are programs implemented with directed acyclic graphs, functional languages or stack-based microinstructions.

Resources. The representation languages in GP are often only intermediary. After the GP run, this intermediate language can be translated into some target language. This makes it possible to leverage diversity available from using different compilation tools, such as compilers, linkers and loaders. The personnel and team sources of design diversity have no direct counterpart in GP. There are many parameters to be set when using GP that have no direct counterparts in ordinary development methodologies. These should not be viewed as purely new ways of adding diversity sources since it is probable that a variation in many of them will have to be restricted considerably for the GP process to find a satisfactory solution.

Summary. There are a large number of parameters in a GP system, and they correspond to some of the sources of diversity in traditional design diversity approaches. Research is needed to evaluate which of the parameters, if any, can be used to force the development of diverse software variants.

We believe that a change in the program space parameters has the greatest potential for generating diversity since it alters the space of programs that are searched. Furthermore, changing these parameters is not difficult and does not incur a large cost and thus should be the focus of a pilot experiment. Changing the parameters of the evaluation environment also shows potential for diversity. However, the cost of doing so is greater and may involve developing alternative evaluation environments. Finally, changing the search specific parameters should primarily change the rate of success for the GP system. Thus these parameters must be altered to find suitable solutions and may not be available to use for diversity purposes.

3.3 Cost of using genetic programming

Developing one program variant in GP is an automatic process, once the GP system has been set-up. It needs a great deal of processing power but can be speeded up by using parallel computers. The evaluation of individuals in a GP population can be done in parallel, and different runs can be made in parallel. Compared to a traditional approach to design diversity, such as NVP, the cost of development will likely be low; NVP uses human software developers while GP uses processors. This would imply that using GP would decrease the cost of developing an N-variant system. The initial cost for the GP approach may be higher, however; we may need to try parameter combinations we have not pre-specified, and it is unclear how the verification and maintenance costs compare with a traditional approach.

When using GP, we design and implement an evaluation environment from the specification, choose which GP parameters to vary and which values to vary between. With the NVP process, this preparation phase includes administrative tasks such as choosing the design teams, distributing information to them and managing their work. An additional cost in the GP approach is converting the developed variants to a format suitable for execution. The internal representation in the GP system must be converted to binaries for the target machine. However, this cost can be expected to be low since it can be automated.

The cost issue is further complicated if we take verification and maintenance into account. It is unclear how the verification costs of the two approaches compare. The programs developed with GP are generally difficult for humans to read. And cannot be debugged in the ordinary sense. The programs may need to be reinserted into the GP system and further developed. Another approach might be to re-run development but emphasizing requirements on the program differently. Similar approaches may be used when maintenance is performed on the N-variant system owing to, for example, changing requirements.

3.4 Applicability of genetic programming

We stress that there are serious deficiencies in the theoretical knowledge about genetic programming. The research field is only a few years old, and the technique has been applied mostly to toy problems. There is a feeling in the evolutionary computation community that it is time to "step up" and attack real problems, but there is a risk that GP will not scale up to more complex tasks. The applicability of our proposed approach is directly tied to the applicability of GP. If GP cannot be scaled up to larger problems, neither can our proposed approach.

At its current level of maturity, GP is probably best suited for small and isolated program components, such as simple controllers, even though this somewhat contradicts the reason for using software diversity in the first place. The success criteria for control algorithms can be more easily described than, for example, desktop applications since their effects are apparent in the physical world (or in a simulation). Furthermore, GP can be applied even if the underlying control algorithms are poorly understood or not even theoretically known. If we can implement our requirements in an evaluation environment, GP can be applied.

When using the proposed approach, it is crucial that the evaluation environment is free from errors. Since the environment is used to evaluate all programs developed, it is a single point of failure in our development process. This is analogous to the role of the specification in NVP.

4. Description of experiment

We have used a genetic programming system to conduct two rounds of experiments. In the first round 80 program variants were developed from the same specification and in the latter round 320 additional programs were developed. In the following we discuss the results from the 80-variant experiment. Detailed results from the second round of experiments can be found in appendix III and the analyses are briefly described in section 6.5.

All programs were developed automatically by a custom developed system running on a SUN Enterprise 10000 with the Sun Solaris OS 2.5 and Java Development Kit 1.2. The GP system was run five times for sixteen different settings of parameters. The resulting eighty programs were subjected to the same 10000 test cases and their failure behavior analyzed to assess the failure diversity of the programs. Figure 1 gives a sketch of the experiment environment. Below we describe the target system, the GP system, the design of the experiment and the testing procedure. A more detailed description is given in appendix I.

4.1 Target system

The target system is designed to arrest aircraft on a runway. Incoming aircraft attach to a cable and the system applies pressure on two drums of tape attached to the cable. A computer that determines the break pressure to be applied controls the system. By dynamically adapting the pressure to the energy of the incoming airplane the program should make the aircraft come to a smooth stop. The requirements on a system like this can be found in [US Air Force86]. The system has been used in other research at our department and a simulator simulating aircraft with different mass and velocity is available. The system is more fully described in [Christmansson98].

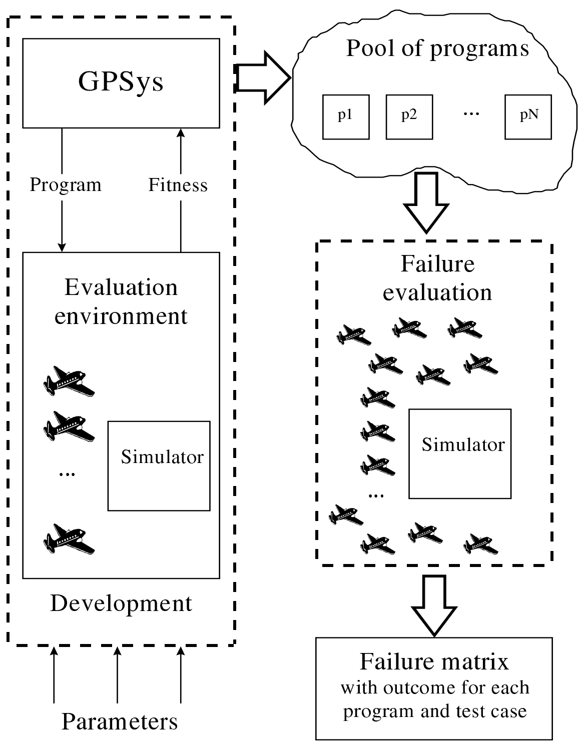


Figure 1. Experiment environment for developing and evaluating airplane arrestment controllers

The main function of the system is to brake aircraft smoothly without exceeding the limits of the braking system, the structural integrity of the aircraft or the pilot in the aircraft. The system should cope with aircraft having maximum energy of $8.81*10^7$ J and mass and velocity in the range 4000 to 25000 kg and 30 to 100 m/s, respectively. More formally the program should[1] (name of corresponding failure class in parentheses)

- stop aircraft at or as close as possible to a target distance (275 m)
- stop the aircraft before the critical length of the tape (335 m) in the system (OVERRUN)
- not impose a force in the cable or tape of more than 360 kN (CABLE)
- not impose a retarding force on the pilot corresponding to more than 2.8g (RETARDATION)
- not impose a retarding force exceeding the structural limit of the aircraft, given for a number of different masses and velocities in [USAF86] (HOOKFORCE)

[1] Our system adopts the requirements of [USAF 86] with the addition of the allowed ranges for mass and velocity and a critical length of 335 m (950 feet in [USAF 86]).

The programs are allowed to use floating point numbers in its calculations. They are invoked for each 10 meters of cable and calculate the break pressure, for the following 10 meters, given the current amount of rolled out cable and angular velocity of the tape drum.

An existing simulator of the system has been ported from C to Java. It implements a simple mechanical model of the airplane and braking system and calculates the position, retardation, forces and velocities in the system. It does not model the inertia in the hydraulic system or oscillatory movement of the aircraft due to elasticity in the tape. The simulator has been set to simulate braking with a time step of 62.5 milliseconds.

4.2 Genetic programming system

Our development system is built on top of the GPSys genetic programming system written in Java by Adhil Quereshi at the University College in London. The programs in this system are function trees, which are interpreted when used in braking the aircraft. During evolution GPsys invokes the simulator to evaluate the fitness of programs. Values from the simulation are used to assign penalty values on the four fitness criteria. The penalties are assigned in a non-linear fashion with high values when the program fails on the criteria. For the OVERRUN criteria:

- If the stop position of the aircraft is larger than the critical length of the system a basic penalty is assigned. The basic penalty was chosen as 80% of the maximum penalty for the criteria.
- A guiding penalty is assigned if the velocity of the aircraft is larger than zero on the critical length. This is to distinguish programs that almost succeeded in braking the aircraft from programs that haven't even tried and "guides" the programs in the direction of good performance. The basic penalty was chosen as 20% of the maximum penalty for the criteria.
- If the aircraft comes to a halt, a linear penalty is assigned. It diminishes from its maximum value at position 0 up to the target distance and then increases up to its maximum again at the ciritcal length. This is to ensure that a halt position close to the target distance will give the program a low penalty. The maximum amount of linear penalty is a parameter to the system but is typically much smaller than 80%.

The penalties for the other criteria are assigned in a similar manner. For more details consult appendix I. The penalty values on the four criteria are summed to give the total fitness for the test case. The total fitness of the program is the sum of the finesses on all the test cases. A perfect program would get a fitness value of zero.

4.3 Testing procedure

After each run of the GP system the best program is evaluated on 10000 test cases evenly spread on the range of valid masses and velocities. Dividing the range of allowed mass into 100 location 212.12 kg apart generates these test cases. For each mass a maximum velocity is calculated so that the resulting energy does not exceed the $8.81*10^7$ J specified in [USAF86]. The range [30, max velocity for this mass] is divided into 100 velocities and a total of 100*100=10000 test cases result.

4.4 Experimental design

The discussion in section 3.2 argued that the program space defining parameters (PSP) should have the largest effect on the diversity of the resulting programs. The parameters to the evaluation environment (EP) should also have an effect while the search parameters (SP) may primarily affect the effectiveness of the GP system. In accordance with this we have chosen to vary four program space parameters, three evaluation environment parameters and one search parameter. Many of these parameters can take multiple values giving rise to an enormous number of combinations. To make a study feasible we have confined the parameters to two levels, represented by '-' and '+'. The parameters and their levels are listed in table 2. More details on the choice of parameters and levels can be found in Appendix I. All other parameters to the system were held constant during the experiment. Each run used 1000 programs in the population and ran for 200 generations.

The result of a GP run is not deterministic and we need replicated runs for each setting of the parameters. The number of unique settings of eight 2-valued parameters is 256 but we used a $2^{(8-4)}$ fractional factorial of resolution IV to reduce this to 16 [Box78] [USAF86]. The settings of the parameters are shown in table 3, where a '-' indicate the 'low' level of the parameter and a '+' indicate the 'high' level. Once the order in which to run the 80 experiments had been randomized the experiment was started. The system ran the 80 runs over a course of five days without any human intervention.

Factor	Level	Description
A	-	No effect.
	+	The statement IF, and operators LE, AND and NOT can be used in the programs.
B	-	No effect.
	+	The functions SIN and EXP can be used in the programs.
C	-	The average velocity, average retardation, and the index to the current checkpoint can be used in the programs.
	+	The angular velocity, current time since start of the braking, the previous angular velocity and the time of the previous checkpoint can be used in the programs.
D	-	Programs cannot use any subroutines.
	+	Two subroutines (automatically defined functions) can be used in the program. They are evolved in the same manner as the rest of the program.
E	-	Maximum penalty on the RETARDATION failure criteria is 1000.0.
	+	Maximum penalty on the RETARDATION failure criteria is 2000.0.
F	-	Linear penalties are not used.
	+	Linear penalties are used and a maximum penalty of 30.0 is assigned on each failure criteria.
G	-	25 test cases uniformly spread on the range of possible values for mass and velocity are used to evaluate fitness during evolution.
	+	25 test cases chosen randomly for each run of the GP system are used to evaluate fitness during evolution.
H	-	Probability of mutation is 0.05.
	+	Probability of mutation is 0.6.

Table 2. Factors that are varied in the experiment

Setting	A	B	C	D	E	F	G	H
1	-	-	-	-	-	-	-	-
2	+	-	-	-	-	+	+	+
3	-	+	-	-	+	-	+	+
4	+	+	-	-	+	+	-	-
5	-	-	+	-	+	+	+	-
6	+	-	+	-	+	-	-	+
7	-	+	+	-	-	+	-	+
8	+	+	+	-	-	-	+	-
9	-	-	-	+	+	+	-	+
10	+	-	-	+	+	-	+	-
11	-	+	-	+	-	+	+	-
12	+	+	-	+	-	-	-	+
13	-	-	+	+	-	-	+	+
14	+	-	+	+	-	+	-	-
15	-	+	+	+	+	-	-	-
16	+	+	+	+	+	+	+	+

Table 3. Fractional factorial design of experiment with levels for the parameters at each setting

5. Experimental results

For each test case executed, a trace of the braking of the airplane is returned from the simulator. Four values are extracted from this trace to classify the behavior of the program: halt distance of the aircraft, maximum force in the cable, maximum retardation force on the hook and maximum retardation during the braking. These values correspond to the four fitness criteria above. We record a failure for a particular variant on a particular test case if *any* value exceeds its limits. Failure is indicated by one (1) and success by zero (0) and these binary values are collected into a failure behavior vector giving the failure behavior on a particular test case.

Setting	Run 1	Run 2	Run 3	Run 4	Run 5	Average	P_{succ}
1	1083	708	813	1327	1475	1081.2	89.19%
2	**591**	2100	648	1746	831	1183.2	88.17%
3	893	1275	888	1016	1150	1044.4	89.56%
4	2203	2694	1644	2639	1240	2084	79.16%
5	**588**	670	1657	**559**	1159	926.6	90.73%
6	801	**559**	965	753	2968	1209.2	87.91%
7	**499**	697	**575**	1054	985	762	92.38%
8	998	**586**	1479	767	713	908.6	90.91%
9	3146	2429	3609	2374	2408	2793.2	72.07%
10	1200	1433	1212	1063	2112	1404	85.96%
11	809	1432	1140	870	1027	1055.6	89.44%
12	1726	755	1782	2255	1789	1661.4	83.39%
13	811	996	852	754	1578	998.2	90.02%
14	**392**	1177	2240	1026	942	1155.4	88.45%
15	1108	1053	**630**	2388	**560**	1147.8	88.52%
16	2946	1111	1005	827	954	1368.6	86.31%

Table 4. The number of failures for each of the 80 versions, the average and the average success probability for each setting of the parameters

The quality of the eighty programs varies highly. Table 4 shows the observed failure rates of the variants. The average number of failures is 1298.96 (Probability of success, P_{succ} = 87.01%) with a standard deviation of 712.89 failures. The best program failed on 392 test cases (P_{succ} = 96.08%) while the worst failed on 3609 (P_{succ} = 63.91%). The top ten performing programs are shown in bold face in table 4. The average number of failures among them is 553.90 (P_{succ} = 94.46%) with a standard deviation of 65.93 failures.

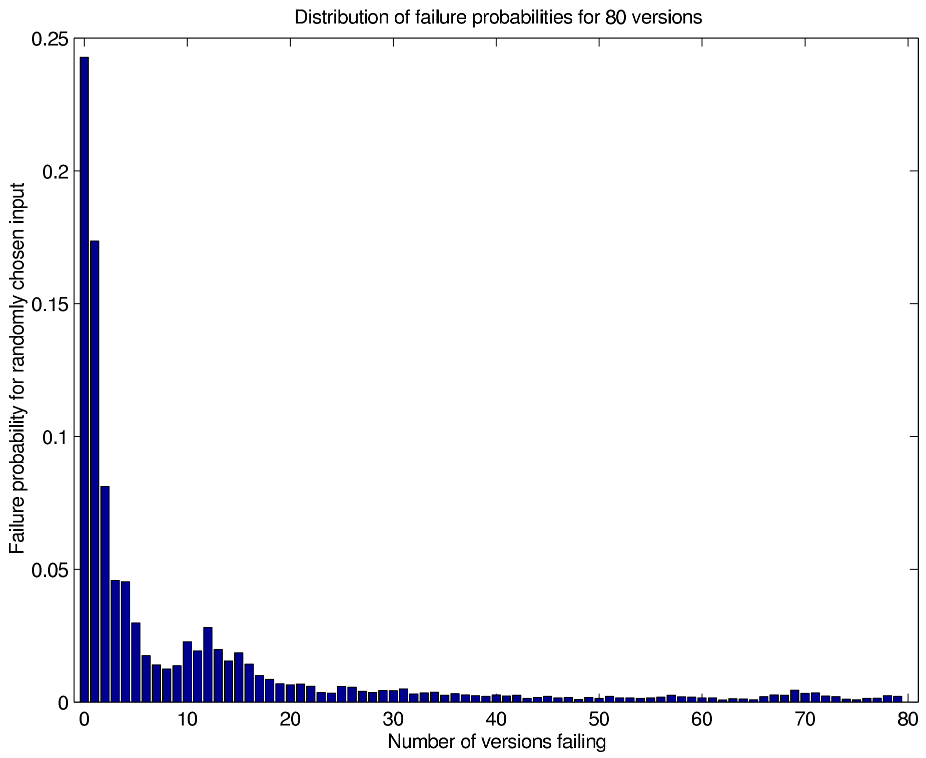


Figure 2. Distribution of failure probabilities for a randomly chosen input

Many programs failed on the same test case. Figure 2 shows the probability that n of the eighty variants fail on a randomly chosen test case among the 10000 test cases. There are no test cases for which all programs fail but many test cases seem to be troublesome for the programs. For example, there are 22 test cases on which 79 of the programs fail and 24 test cases on which 78 fail. This indicates that some test cases are more difficult than others. The variability in difficulty is shown in a contour plot in figure 3. Darker areas show regions where more programs fail.

The structural diversity of the programs varies. A simple measure of this diversity was recorded: the size of the program trees. The average size is 100.20 nodes in the tree with a standard deviation of 82.87. The maximum size is 459 and the minimum size is 17. No correlation was found between the size of the programs and the number of failures they exhibited (correlation coefficient 0.05). The average size of the top ten programs is 84.80 with a standard deviation of 46.07. The maximum size is 185 and the minimum is 38.

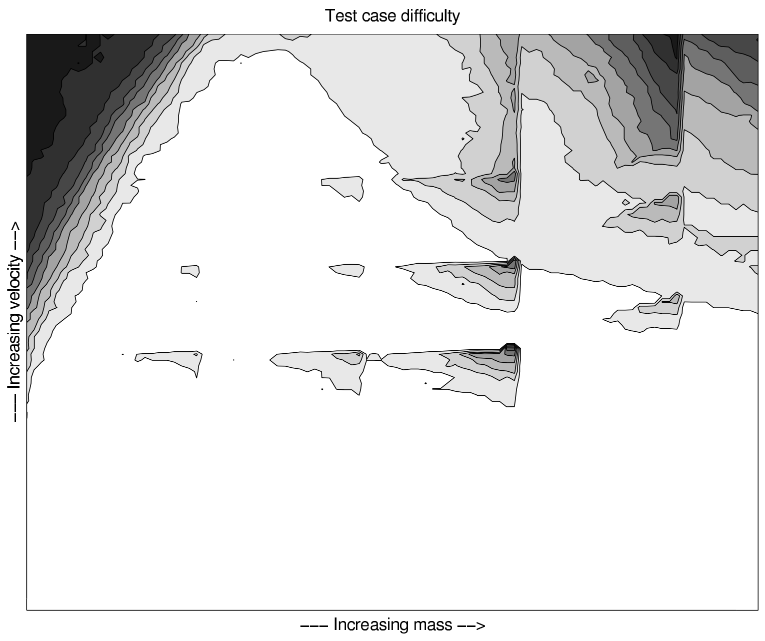


Figure 3. Contour plot of test case difficulty

6. Evaluation of results

Below we evaluate the failure diversity, test case difficulty variability and performance of 3-, 5- and 7-variant systems constructed from the programs and briefly describe the analysis of the second round of experiments with a total of 400 variants. The failure diversity is evaluated between the individual programs and between the different methods defined by our 16 different settings of parameters. A statistical test is performed to evaluate if varying the parameters to the system generates diversity.

6.1 Failure diversity

Different measures of diversity have been proposed in the literature. In [Littlewood89], Litttlewood and Miller propose that the amount of diversity between two design methods should be measured using the correlation coefficient of the joint distribution of their failures. Their measure is theoretical since it should be applied for all input cases and programs that can be developed with the methods. We have used it in the same way that Littlewood and Miller use it in their examples; by disregarding difficult issues of statistical sampling [Littlewood89]. Another failure diversity measure was used in [Lyu94]. It is defined as the number of distinct failures divided by the total number of failures, and below we denote it LFD.

Between programs. The diversity measures were calculated pairwise for all eighty programs. The minimum correlation[2] was –0.2131 and of the 3160 correlations 193 (6.11%) were below zero. The maximum LFD was 0.9894. This is encouraging since low correlations and high failure diversity indicates that taking a vote among variants can mask effects of failures. However if we consider only the top ten programs the picture is different. The lowest pairwise correlation found is 0.5495 and the highest pairwise LFD is 0.5965.

Between methods. We have calculated the 120 inter-method correlations where each setting of the parameters to the GP system is considered a unique method. The majority of the correlations are high but 8 are below 0.20 and two are negative. This was surprising and indicates that the variability of difficulty of the test cases may be overcome and the program variants can show better than independent failure behavior. However, the majority of methods involved in the lowest inter-method correlations are the ones having highest average failure rate. Thus, even if we pick programs for N-variant systems from methods showing low correlation, the failure rate of the system will probably not level that of the top performing programs.

Inter-method vs. intra-method diversity. To evaluate if diversity can be obtained by altering the parameters to the GP system we wanted to assess whether there is more diversity between variants in different methods than within the same method. To this end we used the following procedure:

- Randomly choose one method (A) and two distinct programs (A1 and A2) from it
- Randomly choose a program not in A and call it B1
- Compare the diversity between A1 and A2 with the diversity between A1 and B1. If the latter is larger than the former the outcome of the test is called positive.

Under the null hypothesis that there is no difference in diversity between the variants due to the different methods used the number of positive outcomes when the above procedure is repeated should be binomially distributed with n = the number of repetitions of the test and p = 0.5.

For each of the two diversity measures we performed 10000 test procedures. For the correlation measure 6370 positive outcomes were recorded and for the failure diversity measure 6365 positive outcomes. The null hypothesis could be rejected at the 0.01 level for both of the diversity measures (both with p-value $< 10^{-10}$) and we favor the hypothesis that *the failure diversity is larger between variants developed with different settings of the GP parameters than between variants with the same settings.*

The top ten programs do not make up a sufficient data record on which to perform this hypothesis testing. Instead, the procedure was applied on the 11 methods with an average failure rate below the total average[3]. In 10000 repetitions of the procedure 5613 (5551 with the failure diversity measure of Lyu et al) positive outcomes were recorded. Thus, the null hypothesis still could be rejected at the 0.01 level (both with p-value $< 10^{-10}$).

[2] Calculated as the correlation between the failure behavior vectors. This is a special case of the Littlewood and Miller correlation measure when there is only one variant in each method.

[3] Hence, methods number 4, 9, 10, 12 and 16 were excluded

6.2 Test case difficulty variability

Detailed study of the test case difficulty variability pictured in figure 3 reveals that there are three main areas of difficulty. Visually these areas is located in the upper left corner, in equidistant clusters in the center and in the upper right corner, respectively.

For the upper left corner, where aircraft have high velocity and low mass, the programs generally fail on the RETARDATION criteria. It seems plausible to assume that these failures arise because the programs do not properly measure and/or use a notion of the mass of the incoming aircraft in their control algorithm.

The failures in the center area are mainly due to failure on the HOOKFORCE criteria. The requirements in [USAF86] stated the maximum allowed hook force for certain "points" with specified mass and velocity. The clusters of failing programs seen in the center of figure 3 are located below (lower velocity) and to the left (lower mass) of these points. These are the areas where the energy of the aircraft is at a maximum for the requirement of maximum hook force. In this light the clusters can almost be expected to appear.

The failures in the upper right corner are made up of failures on the HOOKFORCE and HALTDISTANCE criteria. The former can be explained by the same reasoning as above and the latter arises because the energies of the aircraft take on their largest values this area. If the programs do not exert a high enough brake pressure in the start of the braking they will not have time to brake the aircraft before the critical length.

6.3 3-variant systems constructed from the programs

We constructed 3-variant systems from our programs. The majority vote between the failure behaviors of the programs was taken as the outcome if voting had been applied during the brakings. We believe that this is a worst-case scenario, but have not investigated it further. If the voting is applied in the checkpoints during the braking failures that occur at different points in time might be masked. For example, this would happen if program 1 exceeds the maximum allowed retardation early in the braking but after that performs well and program 2 have the opposite behavior (good performance early, failing in the end). With our post-run voting the behavior of the system would be deemed a failure regardless of the fact that actual voting at the checkpoints would mask the failures.

We considered all the 120 possible N-variant systems consisting of 3 programs taken from the top ten programs. In 41 (34.17%) of them the failure rate of the system was lower than the minimum failure rate of the individual programs. The best improvement found, compared to the minimum failure rate of the individual programs in the system, was a decrease from 559 to 444 failures (20.57%).

We also compared the performance of the best 3-variant systems to the performance of the best individual programs. Table 5 shows the top 25 programs or systems. In the table, the column marked 'Type' shows the type of system with 'Ind' indicating an individual program and '3VS' a 3-variant system. The column 'Failures' shows the number of failures. If the system is a '3VS' the column 'Improvement' shows the percent improvement in failure rate of the system compared to the best of the individual programs in the system.

As can be seen in the table no 3-variant system performed better than the best individual program. However, the 3-variant systems dominate the top list and only two individual programs perform good enough to qualify.

Rank	Type	Failures	Improvement
1	Ind	392	NA
2	3VS	444	11.02%
3	3VS	444	20.57%
4	3VS	444	20.57%
5	3VS	449	19.68%
6	3VS	455	18.60%
7	3VS	463	7.21%
8	3VS	464	16.99%
9	3VS	465	16.82%
10	3VS	467	16.46%
11	3VS	469	16.10%
12	3VS	470	5.81%
13	3VS	474	15.21%
14	3VS	484	3.01%
15	3VS	485	13.24%
16	3VS	490	12.34%
17	3VS	493	11.81%
18	Ind	499	NA
19	3VS	504	10.00%
20	3VS	515	7.87%
21	3VS	520	7.14%
22	3VS	524	6.26%
23	3VS	525	6.08%
24	3VS	525	6.25%
25	3VS	526	5.90%

Table 5. Top 25 performing individual programs and 3-variant systems

6.4 5- and 7-variant systems constructed from the programs

Five and seven variant systems was also constructed from the top 10 programs. No systems were found that improved upon the failure rate of the individual program in the system with the lowest failure rate.

6.5 Analysis of 400 variants

An additional 320 programs were developed in a second round of experiments. In this round, twenty programs were developed for each of the sixteen settings. The analyses above were carried out on the total of 400 programs and the detailed results can be found in Appendix III.

The results from these analyses are much the same as the ones given above with one notable exception. The negative and small inter-method correlations are not as frequent; only one inter-method is below 0.20 (0.0785 between settings 9 and 14). This indicates that the smallest correlations for the 80 variant analysis may be sampling errors.

7. Discussion and conclusions

We have proposed a procedure for developing diverse software variants and shown that the variants can be forced to be diverse by varying parameters to the genetic programming algorithm used to develop the programs. The low levels of inter-method diversity found between some settings of the parameters were surprising. It indicates that voting in an N-variants system could mask individual program failures. However, the methods giving the lowest correlations also are the ones with the highest failure rates, and the diversity cannot be exploited to give failure rates lower than the top performing programs. The diversity levels found in the top performing programs was much lower. Further analysis will be conducted to find out if the poor performance can be said to cause the high diversity.

The observed behavior may be explained by the special nature of the target system. It shows high level of input case difficulty variability, which is known to limit the amount of exploitable diversity [Littlewood89]. The difficulty arises from the fact that higher energies put more stress on the system. If this amount of input case variability is typical is not known. Further experiments with other target systems can shed light on this issue.

In our experiments we have varied eight parameters to the GP system. It is possible that different choices of parameters and their values would give different results. For example, the apparent problem of the programs to brake light aircraft with high velocities may be overcome by letting them use an indexed memory, making comparisons between values at different checkpoints possible. Further analysis of the experimental data should focus on revealing the effects of different parameters on the failure rate and diversity of the generated programs. The fractional factorial experimental design we have employed is well suited to this end. The diversity may also be limited by choices we made for the basic system design. For example, better results may be achieved if the programs get to calculate the break pressure more frequently during a braking. This would probably require a smaller time step in the simulation and lead to higher performance demands during program development. We will investigate tools for compiling the experimental environment to native machine code. The increase in performance will allow larger populations and longer runs of the GP system, possibly resulting in better program performance.

Our classification of a failure can be considered worst-case. When comparing the failure behavior of programs we do not compare each failure criteria individually. Thus, diversity in the way the programs fail is not accounted for even though it might be exploited in a system employing fault masking. Our failure classification does not take the time aspect of the program behavior into account. A situation can easily be envisioned where two programs both exceed the maximum allowed hook force but at different times. This faulty behavior might be masked by an N-variant system. It would be interesting to actually construct N-variant systems from our programs and evaluate their failure behavior.

Further experimentation with the existing system will be conducted since it mainly amounts to initiating runs and collecting and analyzing data; the development of the programs requires no human activity. Having large numbers of software variants that adhere to the same specification may prove an important step in understanding software diversity and its limitations. The approach described in this paper is not limited to genetic programming. It can be used with other techniques for program generation or induction to obtain more sources of diversity. It would be interesting to extend our work and compare different techniques of this kind.

Investigating how new computational models, such as evolutionary computation, affect and can be used in the field of software reliability and fault tolerance is interesting and generates many ideas. We believe that a well of inspiration for building reliable computing systems can be found by studying nature and biological organisms as suggested in [Avizienis95].

8. References

[Avizienis77] AVIZIENIS, A. and CHEN, L.: 'On the implementation of N-variant programming for software fault-tolerance during program execution', Proc. of COMPSAC-77, 1977, pp. 149-155

[Knight86] KNIGHT, J. C. and LEVESON, N.: 'An experimental evaluation of the assumption of independence in multivariant programming,' *IEEE Trans. on Software Engineering*, **12** (1) pp. 96-109

[Lyu 94] LYU, M., CHEN, J-H. and AVIZIENIS, A.: 'Experience in metrics and measurements for N-variant programming,' *Int. Journal of Reliability, Quality and Safety Engineering*, **1** (1) pp. 41-62

[Littlewood89] LITTLEWOOD, B. and MILLER, D. R.: 'Conceptual modelling of coincident failures in multivariant software,' *IEEE Trans. on Software Eng.*, **15** (12) pp. 1596-1614

[Hatton97] HATTON, L.: 'N-Variant design versus one good variant', *IEEE Software*, **14** (6) pp. 71-76

[Koza92] KOZA, J. R.: 'Genetic programming - on the programming of computers by means of natural selection' (MIT Press, Cambridge, Massachusetts, 1992)

[Bäck97] BÄCK, T., HAMMEL, U. and SCHWEFEL, H-P.: 'Evolutionary computation: comments on the history and current state,' *IEEE Trans. on Evolutionary Computation*, **1** (1) pp. 3-17

[Koza97] KOZA, J. R. (ed): 'Proceedings of Second Annual Conf. on Genetic Programming July 13-16, 1997' (Morgan Kaufmann, San Fransisco, California, 1997)

[Nordin95] NORDIN, P. and BANZHAF, W.: 'Real time evolution of behavior and a world model for a miniature robot using genetic programming', Technical Report 5/95, Department of Computer Science, University of Dortmund, 1995

[Ryan96] RYAN, C. O.: 'Reducing premature convergence in evolutionary algorithms.' PhD Dissertation, Computer Science Department, University College, Cork, 1996

[Banzhaf98] BANZHAF, W., NORDIN, P., KELLER, R. and FRANCONCE, F.: 'Genetic programming - an introduction' (Morgan Kaufmann, San Fransisco, California, 1998)

[Zhang97] ZHANG, B-T. and Joung, J-G.: 'Enhancing robustness of genetic programming at the species level', *Proc. of Second Annual Conference on Genetic Programming*, July 1997, Stanford University, USA, pp. 336-342

[Box78] BOX, G. E., HUNTER, W.G., and HUNTER, J.S.: ‘Statistics for experimenters - an introduction to design, data analysis and model building’ (John Wiley & Sons, New York, 1978)

[USAF86] US Air Force – 99: ‘Military Specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction’, MIL-A-38202C, Notice 1, US Department of Defense, 1986

[Christmansson98] CHRISTMANSSON, J.: ‘An exploration of models for software faults and errors’, PhD Dissertation, Department of Computer Engineering, Chalmers University of Technology, 1998

[Avizienis95] AVIZIENIS, A.: ‘Building dependable systems: how to keep up with complexity’. *Special Issue from FTCS-25 Silver Jubilee*, June 1995, Pasadena, California, pp. 4-15

[Saglietti90] SAGLIETTI, F.: ‘Strategies for the acheivement and assessment of software fault-tolerance’. *IFAC 11th World Congress on Automatic Control*, Tallinn, USSR, 1990, pp. 303-308

[Fenton91] FENTON, N. E.: ‘Software Metrics - A Rigorous Approach’. (Chapman & Hall, 1991)

Appendix I. Detailed description of the experiment environment

Below we give additional information on different parts of the experiment environment: the genetic programming system, Java system, the custom developed Java code, parameter values for the GP system, parameters that are varied in the experiment, design of the factorial experiment, fitness evaluation, the runs of the system and the analysis performed.

Genetic programming system

We have used versions 1.1 of the GP system called 'GPSys' developed by Adil Quereshi at the University College in London. This version of the system was released on the 30th of June 1997 and can be downloaded for free from the following web site:

http://www.cs.ucl.ac.uk/staff/ucacaxq/gpsys_doc.html

(if this URL is no longer valid you can contact the author of this paper to get a copy of the GPSys system).

GPSys is a strongly typed steady-state genetic programming system written in Java. It requires Java version 1.1 or later. The system is structured into a base package and a package of primitives, i.e. the terminals and functions that can be used in the development of programs. Both packages are object oriented and can be easily extended by writing additional Java classes. The system have support for automatically defined functions, i.e. sub-routines.

The major drawback of the system is that it is not compiled into a native machine language; the performance of the system is relatively poor.

Java system

We have used SUN's Java development kit 1.2 with the sunwjit just-in-time compiler. The first time a Java class is loaded it is compiled to machine code for the Sun Sparc architecture and can be executed with greater performance on following invocations. The speed-up achieved with the jit compiler was between 2-3 times.

Custom developed Java code

A total of 28 Java classes were developed to extend the GPsys system with additional functionality needed in the chosen application. These classes contain a total of 1724 lines of source code (comments excluded). Fifteen of the classes implement the evaluation of the programs by interfacing to the simulator defining test cases, assembling information about the braking, evaluating the braking and assign a fitness score. Thirteen classes interface with the GPSys system and the user. They are implemented so that the levels for the factors determining the parameter values that are varied can be given from the command line interface when an experiment is started.

Parameter values for the GP system

Eight different parameters were allowed to vary in the experiments. Table 6 below shows the values of the parameters that did not vary between different runs. When ADF were used two ADF's were allowed: ADF1 and ADF2. The values of parameters for these ADF's are shown in table 7 and 8, respectively.

Parameter	Value
Generations	200
Population size	1000
Tournament size	5
Max depth of program trees at creation	7
Max depth of program trees	19
Max depth of mutation trees	3
Create Method	Ramped-half-and-half

Table 6. Values of parameters that were not varied during the experiments

Parameter	Value
Max depth of program trees at creation	5
Max depth of program trees	9
Max depth of mutation trees	3
Functions allowed	Add, Sub, Mul, Div, If, GE, LE, ADF2
Terminals	Arg1, Arg2, Arg3, Double constant
Create Method	Ramped-half-and-half

Table 7. Values of parameters in ADF1

Parameter	Value
Max depth of program trees at creation	5
Max depth of program trees	9
Max depth of mutation trees	3
Functions allowed	Add, Sub, Mul, Div
Terminals	Arg1, Arg2, Double constant
Create Method	Ramped-half-and-half

Table 8. Values of parameters in ADF2

Factors that are varied in experiments

Of the eight factors varied in the experiment four (A, B, C and D) are program space parameters (PSP). Factors A and B allows the use of additional functions. When factor A is on its high level (indicated with a '+' in table 2) the programs can use the 'function' IF and the three operators LE (Larger-than-or-equal), AND (logical And), and NOT (logical negation). When factor B is on its high level the functions SIN and EXP can be used in the programs. It was thought that these functions might give diverse algorithms since they are suited to model oscillatory and damping behavior, respectively. Factor C governs what terminals can be used by the programs and factor D governs if subroutines can be used. When it is on its high level two subroutines can be used in the programs. The subroutines are evolved together with the main program.

Three factors (E, F and G) are evaluation parameters (EP) affecting how the programs are evaluated. Factors E and F alters different aspects of the fitness evaluation while factor G affects the number and choice of test cases. When factor G is at its low level (indicated with '-' in table 2) the test cases are evenly distributed on the range of allowed values of mass and incoming velocity. When it is on its high level the values for mass and velocity are randomly chosen.

One factor (H) is a search parameter (SP) governing the probability of mutation in the GP system. Initial runs indicated that high values might be beneficial.

Design of the factorial experiment

The defining relation for the factorial experiment is [Box78]:

$$I = BCDE = ACDF = ABCG = ABDH$$

To generate the 16 different settings we listed all 16 combination of factor levels for the factors A, B, C and D. From the defining relations the values for the remaining factors was calculated according to

$E = B*C*D$
$F = A*C*D$
$G = A*B*C$
$H = A*B*H$

where a '+' is assigned the value '+1' and '-' the value '–1'. For more information on these issues see [Box78].

Fitness evaluation

When program evaluation takes place a program is tested on a number of test cases, i.e. air planes with a certain mass and velocity, coming in to land on a runway. An aircraft is characterized by its mass and incoming velocity. The braking is simulated in time steps and at each time step a number of data about the braking is logged. The data is the position, velocity and retardation of the aircraft, the force on the hook of the aircraft, the force in the cable, and the pressure applied on the tape drum by the braking system.

After each braking the logged data are analyzed to evaluate if any of the four success criteria have been violated. On each criteria a penalty value is assigned and these values are added to give the penalty value on the braking. Bu summing the penalty values for all the brakings an aggregated penalty value is obtained. The penalty value is used as an "inverse" fitness value so that high penalty indicates low fitness and low penalty indicates high fitness. A "perfect" individual has a penalty value of zero indicating that no success criteria were violated, i.e. the program adhered completely to the specification, on the test cases.

The halt distance criteria (OVERRUN) ensures that the aircraft were arrested before the critical length of 335 meters. If the program did not succeed, i.e. the airplane speed was larger than zero at the critical length of 335 meters, a basic penalty of 800 units is assigned.

An additional penalty of maximum 200 units is assigned linearly based on how much the velocity of the airplane was when it exceeded the critical length. The maximum value of 200 is added when the velocity at the critical length is the same as the incoming velocity, i.e. the system has not even tried to brake the system. When the system has been able to brake the aircraft before the critical length a small penalty is assigned based on how much the braking distance deviates from the target distance of 275 meter. This penalty increases from zero at the target distance to a maximum of 30 units at the critical length and at the engaging position. The penalty assignment on this criterion is shown in figure 4 below.

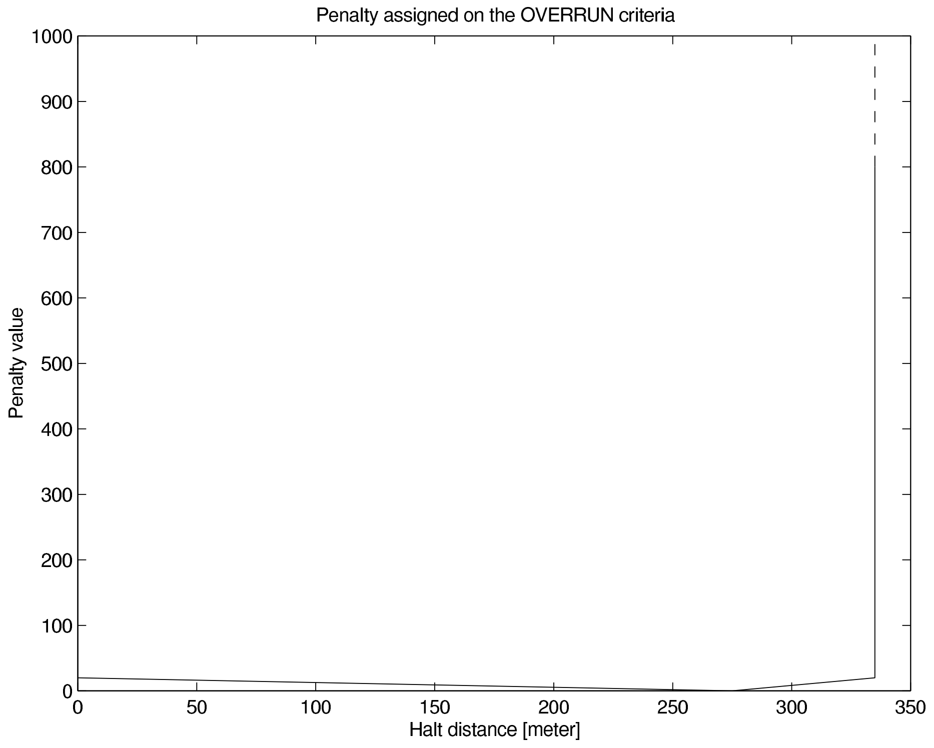


Fig 4. Penalty assignment on the OVERRUN criterion

The hook force criteria (HOOKFORCE) is parameterized on the maximum hook force that the aircraft can handle. These maximum hook forces are taken from the specification and can be found in a table in the document [USAF86]. The table gives the maximum allowed force in the hook on the aircraft with a specified mass and incoming velocity for a total of seventeen different mass and velocity "points". For masses and velocities not given in the table we have used the value of the closest given point that have the smallest higher mass and smallest higher velocity. This divided the mass and velocity plane into 17 boxes where the same maximum hook force is allowed. However, with the additional requirements we posed on the system that it should handle velocities up to 100 m/s and masses up to 25000 kg some parts is not covered by the 17 boxes. For these points we have used a formula obtained from the seventeen given values using linear regression.

If the maximum hook force exerted during a braking has been larger than the maximum hook force that the aircraft can handle, called the critical hook force, a basic penalty of 800 units is assigned. A linear extra penalty is assigned for maximum hook forces over the critical hook force taking the maximum value of 200 units at two times the critical hook force. A small linear penalty is assigned that has a value of zero at a maximum hook force of zero and the value 30 units at the critical hook force. An example of a penalty assignment on this criterion, for a critical hook force of 220.2 kN for aircraft with masses in the range [13608, 18144] kg and velocities in the range [61.8, 72.1] m/s, is shown in figure 5 below.

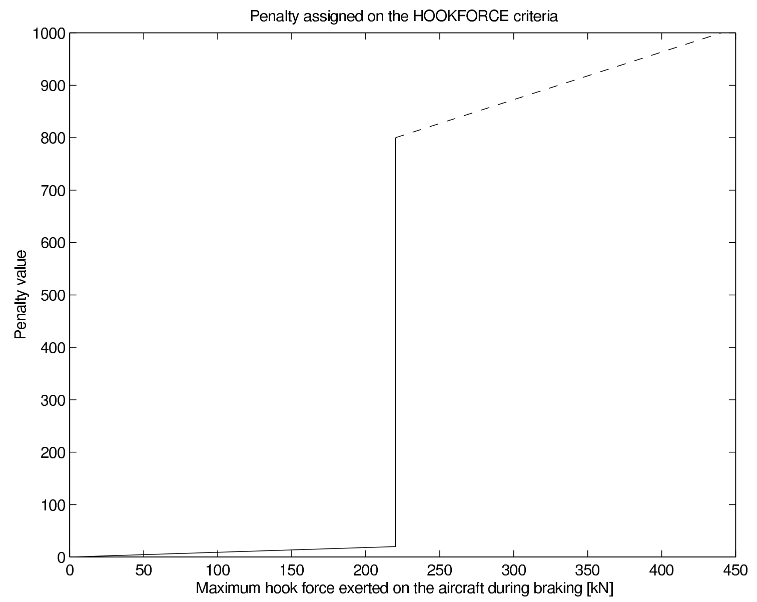


Fig 5. Penalty assignment on the HOOKFORCE criterion with a critical hook force of 220.2 kN

The penalty for the CABLEFORCE and RETARDATION criterion is assigned in the same way as the HOOKFORCE criterion above. The only differences is the critical value where the penalty value changes abruptly. For the CABLEFORCE criterion this critical value is the maximum allowed force in the cable that the braking system can handle. For the particular system used the value is 360.0 kN. For the Retardation criteria the critical value is 2.8*g, which is chosen to ensure that the pilot will not pass out during the braking.

Additional details about the runs

Each run of the development system has got a unique experiment number. The output from each run is four files: the experiment file, the failure data file, the log file and a file with the best of run individual.

The experiment file is a complete description of the experiment logging the experiment number, start time, duration of the run, the factor levels and the parameter values, fitness and complexity of the best-of-run individual, and a summary of the evaluation results.

The failure data file contains the result from the 10000 test cases. A number is output for each test case. The four low-order bits of the number corresponds to the four different failure classes and if a bit is ‘1’ the best-of-run individual failed on these criteria for this particular test case.

In the log file data about each generation of the run is logged during the run. For each generation the average fitness, average complexity, best individual, fitness of best individual and complexity of best individual is printed in the log. The log data can be used to analyze the evolutionary process.

The best-of-run individual is saved in binary form so that it can be loaded at a later time for additional tests.

Execution time of the runs

The execution time of the runs varied a lot. For the eighty programs the mean execution time was 19756 seconds, i.e. about 5 and a half hours. The standard variation was 9105 seconds, i.e. about 2 and a half hours. However, since each run can be run separately and the computer contained a total of fourteen CPU's the runs were parallellized and the experiments could be completed in a shorter time span.

The execution time is highly dependent on the use of the Java interpreting programming language. Using a GP system implemented in C och C++ or even a GP system using machine language for the program representation should decrease the execution times considerably. Nordin have reported speed-up factors of over 1000 times using a machine language representation [Banzhaf98]. Using an environment like that of Nordin would allow larger populations and more generations which should increase the likelihood of finding good solutions with small failure rates.

The process of starting experiment runs was automated so the process could be initiated and then left on its own.

Analysis of experiments

Analysis are performed off-line using MATLAB. Scripts have been written for the analyses and once all the experiment numbers have been listed and been read into MATLAB the scripts performs the analyses. The results are written to an output file and figures are generated from MATLAB and written on files. An example of such a file is shown in appendix II and III below.

Appendix II. Detailed results from the analysis of 80 variants

The analysis of the failure behaviors was carried out in MATLAB. A MATLAB-script runs the analyses and outputs the results in textual form to a file. Below the resulting file from the analyses of the 80-variant experiment described in section 4 and 5 are shown.

```
Tests and analyses for technical report 98-13 on the GPBRExperiments using 80 versions
-------------------------------------------------------------------------------------
Average number of failures: 1298.96 (Psucc=87.01%)
Standard deviation: 712.89
Best program, number of failures: 392 (Psucc=96.08%)
Worst program, number of failures: 3609 (Psucc=63.91%)
Top 10, Average number of failures: 553.90 (Psucc=94.46%)
Top 10, Standard deviation: 65.93

Graph showing probability that n versions out of 80 versions
fail on a randomly chosen test case among the 10000 test cases was written to file

   fig_probfailure.eps

Number of testcases that fail on 80 versions: 0
Number of testcases that fail on 79 versions: 22
Number of testcases that fail on 78 versions: 24
Number of testcases that fail on 77 versions: 15
Number of testcases that fail on 76 versions: 14
Number of testcases that fail on 75 versions: 9
Number of testcases that fail on 74 versions: 11
Number of testcases that fail on 73 versions: 21
Number of testcases that fail on 72 versions: 23
Number of testcases that fail on 71 versions: 35
Number of testcases that fail on 70 versions: 33

Contourplot of testcase difficulty variability written to file

    fig_testcase_difficulty.eps

Structural diversity
--------------------

Average size: 100.20
Standard deviation: 82.87
Max size: 459
Min size: 17

Correlation between size and failure rate: 0.05

Top 10 programs
Average size: 84.80
Standard deviation: 46.07
Max size: 185
Min size: 38
```

Failure diversity

1, Between programs

All programs:
Minimum correlation: -0.2131
Number of correlations: 3160
Negative correlations: 193.00 (6.11%)
Small correlations (<0.2): 574.00 (18.16%) Note that the negative correlations are included in the small correlations.
Maximum failure diversity: 0.9894

Top programs:
Minimum correlation: 0.5495
Maximum failure diversity: 0.5965

2, Between methods

Number of intermethod correlations: 120
Number of negative correlations: 2
Number of small correlations (<0.2): 8 Note that the negative correlations are included in the small correlations.

The 20 smallest inter-method correlations:

Method 1	Method 2	Correlation
9	14	-0.0454
9	11	-0.0180
9	12	0.0766
1	9	0.0978
7	9	0.0994
8	9	0.1322
4	14	0.1685
9	13	0.1869
4	11	0.2604
3	9	0.2758
9	10	0.2887
4	12	0.3080
2	9	0.3224
1	4	0.3703
4	7	0.3754
6	9	0.3832
5	9	0.4227
4	8	0.4367
9	16	0.4735
5	14	0.4963

3, Inter- vs. Intra

All programs:

The number of possible binomial tests: 24000
Number of tests performed: 10000

Correlation measure gave 6370 positive outcome.

```
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 6365 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than average:

Methods worse than average that are NOT included: 4   9  10  12  16
Methods better than average: 1   2   3   5   6   7   8  11  13  14  15
Number of methods used in the tests below: 11
The number of possible binomial tests: 11000
Number of tests performed: 10000

Correlation measure gave 5613 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 5551 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than second average (average of the ones that were better than average above):

Methods worse than second average that are NOT included: 1   2   3   4   6   9  10  11  12  14  15  16
Methods better than second average: 5   7   8  13
Number of methods used in the tests below: 4
The number of possible binomial tests: 1200
Number of tests performed: 1200

Correlation measure gave 718 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=5.3169e-12)
Diversity measure gave 695 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=2.2891e-08)

Construct 3VP systems from the top 10 programs
---------------------------------------------
Number of 3VS that can be constructed: 120
Number of 3VS that was constructed: 120
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 41 (34.17%)
Best improvement:
        minimum failures of individual program = 559
        failures of 3VS = 444
        percent improvement = 20.57%

Construct 5VP systems from the top 10 programs
---------------------------------------------
Number of 5VS that can be constructed: 252
Number of 5VS that was constructed: 252
Number of 5VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
        minimum failures of individual program = 559
        failures of 3VS = 620
        percent improvement = -10.91%

Construct 7VP systems from the top 10 programs
---------------------------------------------
Number of 7VS that can be constructed: 120
Number of 7VS that was constructed: 120
Number of 7VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
        minimum failures of individual program = 559
        failures of 3VS = 711
        percent improvement = -27.19%

Combined top list of individual program or 3VS among the top programs
--------------------------------------------------------------------

Rank  Type  Prg 1  Prg 2  Prg 3  Failures Improvement
 1    Ind   747                  392          NA
 2    3VS   744    758    827    444          11.02%
 3    3VS   758    827    803    444          20.57%
 4    3VS   758    827    770    444          20.57%
 5    3VS   801    758    827    449          19.68%
 6    3VS   758    827    743    455          18.60%
 7    3VS   744    801    758    463          7.21%
 8    3VS   758    827    776    464          16.99%
 9    3VS   758    827    764    465          16.82%
 10   3VS   801    758    803    467          16.46%
 11   3VS   801    758    764    469          16.10%
 12   3VS   744    801    827    470          5.81%
 13   3VS   801    827    803    474          15.21%
 14   3VS   744    758    743    484          3.01%
 15   3VS   758    803    743    485          13.24%
 16   3VS   758    764    743    490          12.34%
 17   3VS   801    827    764    493          11.81%
 18   Ind   744                  499          NA
 19   3VS   827    803    743    504          10.00%
 20   3VS   801    758    776    515          7.87%
 21   3VS   827    764    743    520          7.14%
 22   3VS   758    764    770    524          6.26%
 23   3VS   758    803    770    525          6.08%
 24   3VS   827    803    770    525          6.25%
 25   3VS   801    758    770    526          5.90%

Construct 3VP systems from ALL programs
---------------------------------------
Number of 3VS that can be constructed: 120
Number of 3VS that was constructed: 82160
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 14818
(18.04%)
Best improvement:
        minimum failures of individual program = 1746
        failures of 3VS = 1010
        percent improvement = 42.15%
Total time needed for analysis 83.7 (minutes)
```

Appendix III. Detailed results from the analysis of 400 variants

The original experiment was extended by developing an additional of 320 variants, 20 for each of the 16 settings of parameters. The result from the analysis carried out in MATLAB is shown below.

```
Tests and analyses for technical report 98-13 on the GPBRExperiments using 400 versions
------------------------------------------------------------------------------------
Average number of failures: 1308.96 (Psucc=86.91%)
Standard deviation: 680.65
Best program, number of failures: 392 (Psucc=96.08%)
Worst program, number of failures: 3701 (Psucc=62.99%)
Top 10, Average number of failures: 463.40 (Psucc=95.37%)
Top 10, Standard deviation: 35.60

Graph showing probability that n versions out of 400 versions
fail on a randomly chosen test case among the 10000 test cases was written to file

   fig_probfailure.eps

Number of testcases that fail on 400 versions: 0
Number of testcases that fail on 399 versions: 0
Number of testcases that fail on 398 versions: 0
Number of testcases that fail on 397 versions: 0
Number of testcases that fail on 396 versions: 0
Number of testcases that fail on 395 versions: 0
Number of testcases that fail on 394 versions: 1
Number of testcases that fail on 393 versions: 0
Number of testcases that fail on 392 versions: 1
Number of testcases that fail on 391 versions: 13
Number of testcases that fail on 390 versions: 9

Contourplot of testcase difficulty variability written to file

    fig_testcase_difficulty.eps

Structural diversity
--------------------

Average size: 108.01
Standard deviation: 97.06
Max size: 725
Min size: 5

Correlation between size and failure rate: 0.04

Top 10 programs
Average size: 107.20
Standard deviation: 49.44
Max size: 226
Min size: 59

Failure diversity
```

```
-----------------

1, Between programs

All programs:
Minimum correlation: -0.4438
Number of correlations: 79800
Negative correlations: 4518.00 (5.66%)
Small correlations (<0.2): 13708.00 (17.18%) Note that the negative correlations are included in the small
correlations.
Maximum failure diversity: 0.9995

Top programs:
Minimum correlation: 0.5371
Maximum failure diversity: 0.6150

2, Between methods

Number of intermethod correlations: 120
Number of negative correlations: 0
Number of small correlations (<0.2): 1 Note that the negative correlations are included in the small
correlations.

The 20 smallest inter-method correlations:

Method 1          Method 2          Correlation
--------  --------  -----------

9                 14                0.0785
9                 12                0.2050
1                 9                 0.3000
7                 9                 0.3182
4                 14                0.3184
9                 11                0.3258
3                 9                 0.4309
8                 9                 0.4354
4                 12                0.4510
5                 14                0.4938
14                16                0.4956
14                15                0.4969
9                 10                0.5011
9                 13                0.5013
12                16                0.5685
1                 4                 0.5713
4                 7                 0.5770
5                 12                0.5911
13                14                0.6076
2                 9                 0.6083

3, Inter- vs. Intra

All programs:

The number of possible binomial tests: 3600000
Number of tests performed: 10000

Correlation measure gave 6318 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
```

```
Diversity measure gave 6312 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than average:

Methods worse than average that are NOT included: 4   6   9  10  12  14  15  16
Methods better than average: 1   2   3   5   7   8  11  13
Number of methods used in the tests below: 8
The number of possible binomial tests: 840000
Number of tests performed: 10000

Correlation measure gave 5897 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 5949 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

For methods that are better than second average (average of the ones that were better than average above):

Methods worse than second average that are NOT included: 2   3   4   5   6   8   9  10  12  14  15  16
Methods better than second average: 1   7  11  13
Number of methods used in the tests below: 4
The number of possible binomial tests: 180000
Number of tests performed: 10000

Correlation measure gave 6313 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)
Diversity measure gave 6506 positive outcome.
Null hypothesis can be rejected at the 0.0100 level (p=0000)

Construct 3VP systems from the top 10 programs
---------------------------------------------
Number of 3VS that can be constructed: 120
Number of 3VS that was constructed: 120
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 32 (26.67%)
Best improvement:
        minimum failures of individual program = 488
        failures of 3VS = 405
        percent improvement = 17.01%

Construct 5VP systems from the top 10 programs
---------------------------------------------
Number of 5VS that can be constructed: 252
Number of 5VS that was constructed: 252
Number of 5VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
        minimum failures of individual program = 462
        failures of 3VS = 479
        percent improvement = -3.68%

Construct 7VP systems from the top 10 programs
---------------------------------------------
Number of 7VS that can be constructed: 120
Number of 7VS that was constructed: 120
Number of 7VS with lower failure rate than the lowest of the individual programs in the system: 0 (0.00%)
Best improvement:
        minimum failures of individual program = 462
        failures of 3VS = 516
        percent improvement = -11.69%
```

```
Combined top list of individual program or 3VS among the top programs
--------------------------------------------------------------------

Rank  Type   Prg 1   Prg 2   Prg 3   Failures  Improvement
 1    Ind    747                     392       NA
 2    3VS    1140    1070    1042    394       5.97%
 3    3VS    1140    1017    1042    395       5.73%
 4    3VS    1140    1029    1042    396       5.49%
 5    3VS    1140    1042    744     397       5.25%
 6    3VS    1140    1042    1189    397       5.25%
 7    3VS    1017    1042    1134    403       13.33%
 8    3VS    1070    1042    1134    404       12.93%
 9    3VS    1029    1042    1134    405       12.34%
 10   3VS    1042    1134    744     405       17.01%
 11   3VS    1042    1134    1189    405       17.01%
 12   Ind    1140                    419       NA
 13   3VS    880     1029    1042    422       6.43%
 14   3VS    880     1070    1042    423       6.21%
 15   3VS    880     1017    1042    424       5.99%
 16   3VS    880     1042    744     425       5.76%
 17   3VS    880     1042    1189    425       5.76%
 18   3VS    880     1042    1134    427       5.32%
 19   3VS    1070    1017    1134    443       4.53%
 20   3VS    880     1070    1017    445       1.33%
 21   3VS    1070    1017    1042    446       3.88%
 22   3VS    1029    1070    1134    451       2.38%
 23   3VS    1029    1017    1134    451       2.38%
 24   Ind    880                     451       NA
 25   3VS    1029    1070    1042    452       2.16%

Construct 3VP systems from the 100 best programs
------------------------------------------------
Number of 3VS that can be constructed: 161700
Number of 3VS that was constructed: 161700
Number of 3VS with lower failure rate than the lowest of the individual programs in the system: 51799
(32.03%)
Best improvement:
        minimum failures of individual program = 639
        failures of 3VS = 415
        percent improvement = 35.05%

Total time needed for analysis 238.5 (minutes)
```

Paper 4.

Robert Feldt. *Genetic Programming as an Explorative Tool in Early Software Development Phases*, Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering, pp. 11-21, Limerick, Ireland, 12th-14th April, 1999.

Genetic Programming as an Explorative Tool in Early Software Development Phases

Robert Feldt
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Tel: +46 31 772 5217, Fax: +46 31 772 3663
E-mail: feldt@ce.chalmers.se

Abstract

Early in a software development project the developers lack knowledge about the problem to be solved by the software. Any knowledge that can be gained at an early stage can reduce the risk of making erroneous decisions and injecting defects that can be expensive to eliminate in later phases. This paper presents the idea of using genetic programming to explore the difficulty of different input data in the input space, determine the effects of different requirements and identify design trade-offs inherent in the problem. Data from a pilot experiment is analysed and the knowledge gained is used to question and prioritize the requirements on the target system. Coping with high-dimensional input spaces and establishing the relationship between GP- and human-developed programs are identified as the major outstanding problems. An extended experimental environment is proposed based on techniques for visual database exploration.

1. Introduction

Software engineers transform the, often fuzzy, demands on a new piece of software into an executable system through a process involving requirements engineering, design, implementation and testing [Sommerville92]. Despite years of efforts on developing better software engineering methods, programming languages and development tools some software projects still fail to meet scheduled milestones and resource limitations. In part, this can be attributed to the fact that software systems are often 'one-off' products and experience from earlier projects can not be relied on in doing predictions and planning [Sommerville92]. In the early phases of a software project not much is known about the difficulty of the task, the inherent trade-offs in the problem or different possible strategies in devising a solution.

If problem-specific knowledge could be obtained early in a development project this could lead to fewer errors being made in the requirements and design phases that, in turn, could decrease development time and cost. Many authors have reported that the cost of finding defects rises sharply in later development phases: a relative time of up to 1000 to one for finding defects in field use compared to during requirements engineering and 80 to one for defects in testing compared to design reviews [Humphrey95]. Information about the problem to be solved that is obtained early in a development project can reduce the risk of making erroneous decisions and minimize the probability of injecting defects in the system that can be expensive to eliminate in later phases. Furthermore, this information could be used during project planning to predict the resources needed.

Soft computing techniques for machine learning can be used when very little is known about a problem and its possible solutions. The ultimate goal of the soft computing technique of genetic programming (GP) is to enable automatic programming, i.e. the computer programming itself [Banzhaf98]. When the requirements on the program have been coded in an executable form, a fitness function, and the parameters of the genetic programming system have been initialized, a large number of programs can be obtained from a GP system. In this paper we propose that problem-specific information be obtained early in a software development project by using genetic programming to explore the input data space of a program and the effect of different requirements on the difficulty of the problem. We propose the term *software problem exploration using genetic programming* (SPE-GP) for this idea and try to identify its implications.

A pilot experiment to test the basic idea of SPE-GP is described in section 2 and the results from the experiment are given in section 3. In section 4, we discuss what knowledge was gained in the experiment and what limits the conclusions that can be drawn from it. We also identify important questions for further research. A general framework for software problem exploration using GP is outlined in section 5. Finally, section 6 concludes the paper.

2. Pilot experiment

In a previous research project we have used a genetic programming system to develop 400 variants of an aircraft arrestment system software [Feldt98] [USAF86]. The programs were developed using the GPSys genetic programming system running on a SUN Enterprise 10000 with the Sun Solaris OS 2.1 and Java Development Kit 1.2 [Quereshi98]. The GP system was run 25 times for sixteen different settings of parameters and the best-of-run individuals were retained for further analysis. These 400 programs were subjected to the same 10000 test cases and their behavior compared to the requirements. Below we describe the target system, the GP system and the testing procedure. A more thorough description is given in [Feldt98].

2.1. Target system

The target system is designed to arrest aircraft on a runway. Incoming aircraft attach to a cable and the system applies pressure on two drums of tape attached to the cable. A computer that determines the break pressure to be applied controls the system. By dynamically adapting the pressure to the energy of the incoming airplane the program should make the aircraft come to a smooth stop. The requirements on a system like this can be found in [USAF86]. The system has been used in other research at our department and a simulator simulating aircraft with different mass and velocity is available [Christmansson98].

The main function of the system is to brake aircraft smoothly without exceeding the limits of the braking system, the structural integrity of the aircraft or the pilot in the aircraft. The system should cope with aircraft having maximum energy of $8.81*10^7$ J and mass and velocity in the range 4000 to 25000 kg and 30 to 100 m/s, respectively. More formally the program should[1] (name of corresponding failure class in parentheses)

- stop aircraft at or as close as possible to a target distance
- stop the aircraft before the critical length of the tape (335 m) in the system (OVERRUN)
- not impose a force in the cable or tape of more than 360 kN (CABLE)
- not impose a retarding force on the pilot corresponding to more than 2.8g (RETARDATION)
- not impose a retarding force exceeding the structural limit of the aircraft, given for a number of different masses and velocities in [USAF86] (HOOKFORCE)

[1] Our system adopts the requirements of [USAF86] with the addition of the allowed ranges for mass and velocity and a critical length of 335 m (950 feet in [USAF86]).

The programs are allowed to use floating point numbers in their calculations. They are invoked every 10 meters of cable and calculate the break pressure, for the following 10 meters, given the current amount of rolled out cable and angular velocity of the tape drum. An existing simulator of the system has been ported from C to Java. It implements a simple mechanical model of the airplane and braking system and calculates the position, retardation, forces and velocities in the system. It does not model the inertia in the hydraulic system or oscillatory movement of the aircraft due to elasticity in the tape. The simulator has been set to simulate braking with a time step of 62.5 milliseconds.

2.2. Genetic programming system

Our development system is built on top of the GPSys genetic programming system written in Java by Adhil Quereshi at the University College in London [Quereshi98]. During evolution GPsys invokes the simulator to evaluate the fitness of programs. Values from the simulation are used to assign penalty values on the four fitness criteria. The penalties are assigned in a non-linear fashion with high values when the program fails on the criteria. For the OVERRUN criteria:

- If the stop position of the aircraft is larger than the critical length of the system a basic penalty is assigned. The basic penalty was chosen as 80% of the maximum penalty for the criteria.
- A guiding penalty is assigned if the velocity of the aircraft is larger than zero on the critical length. The penalty is assigned proportional to the velocity of the aircraft on the critical length. This is to distinguish programs that almost succeeded in braking the aircraft from programs that haven't even tried and "guides" the programs in the direction of good performance. The guiding penalty was chosen as 20% of the maximum penalty for the criteria.
- If the aircraft comes to a halt a linear penalty is assigned. It diminishes from its maximum value at position 0 up to the target distance and then increases up to its maximum again at the critical length. This is to ensure that a halt position close to the target distance will give the program a low penalty. The maximum amount of linear penalty is a parameter to the system but should be much smaller than 80%.

The penalties for the other criteria are assigned in a similar manner. For more details consult [Feldt98]. The penalty values on the four criteria are summed to give the total fitness for the test case. The total fitness of the program is the sum of the finesses on all the test cases. A perfect program would get a fitness value of zero.

The values for the parameters to the GP system that was used in the experiments are shown in tables 1 and 2.

Parameter	Value
Generations	200
Population size	1000
Tournament size	5
Max depth of program trees at creation	7
Max depth of program trees	19
Max depth of mutation trees	3
Functions that were always used	Add, Sub, Mul, Div
Create Method	Ramped-half-and-half

Table 1. Values of parameters that were not varied during the experiments

Factor	Level	Description
A	-	No effect.
	+	The statement IF, and operators LE, AND and NOT can be used in the programs.
B	-	No effect.
	+	The functions SIN and EXP can be used in the programs.
C	-	The average velocity, average retardation, and the index to the current checkpoint can be used in the programs.
	+	The angular velocity, current time since start of the braking, the previous angular velocity and the time of the previous checkpoint can be used in the programs.
D	-	Programs cannot use any subroutines.
	+	Two subroutines (automatically defined functions) can be used in the program. They are evolved in the same manner as the rest of the program.
E	-	Maximum penalty on the RETARDATION failure criteria is 1000.0.
	+	Maximum penalty on the RETARDATION failure criteria is 2000.0.
F	-	Linear penalties are not used.
	+	Linear penalties are used and a maximum penalty of 30.0 is assigned on each failure criteria.
G	-	25 test cases uniformly spread on the range of possible values for mass and velocity are used to evaluate fitness during evolution.
	+	25 test cases chosen randomly for each run of the GP system are used to evaluate fitness during evolution.
H	-	Probability of mutation is 0.05.
	+	Probability of mutation is 0.6.

Table 2. Parameters that were varied between different runs of the GP system

2.3. Testing procedure

After each run of the GP system the best-of-run individual is evaluated on 10000 test cases evenly spread on the range of valid masses and velocities. Dividing the range of allowed mass into 100 locations 212.12 kg apart generates these test cases. For each mass a maximum velocity is calculated so that the resulting energy does not exceed the $8.81*10^7$ J specified in [USAF86]. The range [30, max velocity for this mass] is divided into 100 velocities and a total of 100*100=10000 test cases result.

3. Experimental results

For each test case executed, a trace of the braking of the airplane is returned from the simulator. Four values are extracted from this trace to classify the behavior of the program: halt distance of the aircraft, maximum force in the cable, maximum force on the hook of the aircraft and maximum retardation during the braking. These values correspond to the four fitness criteria above. We record a failure for a particular version on a particular test case if *any* value exceeds its limits. Failure is indicated by one (1) and success by zero (0) and these binary values are collected into a failure behavior vector giving the failure behavior for each test case.

The *difficulty* of a test case is defined as the proportion of GP-developed controllers that failed on the test case. Since two real-valued numbers characterize each test case the input data difficulty can be visualized in a 2D contour diagram. The diagram is shown in figure 1.

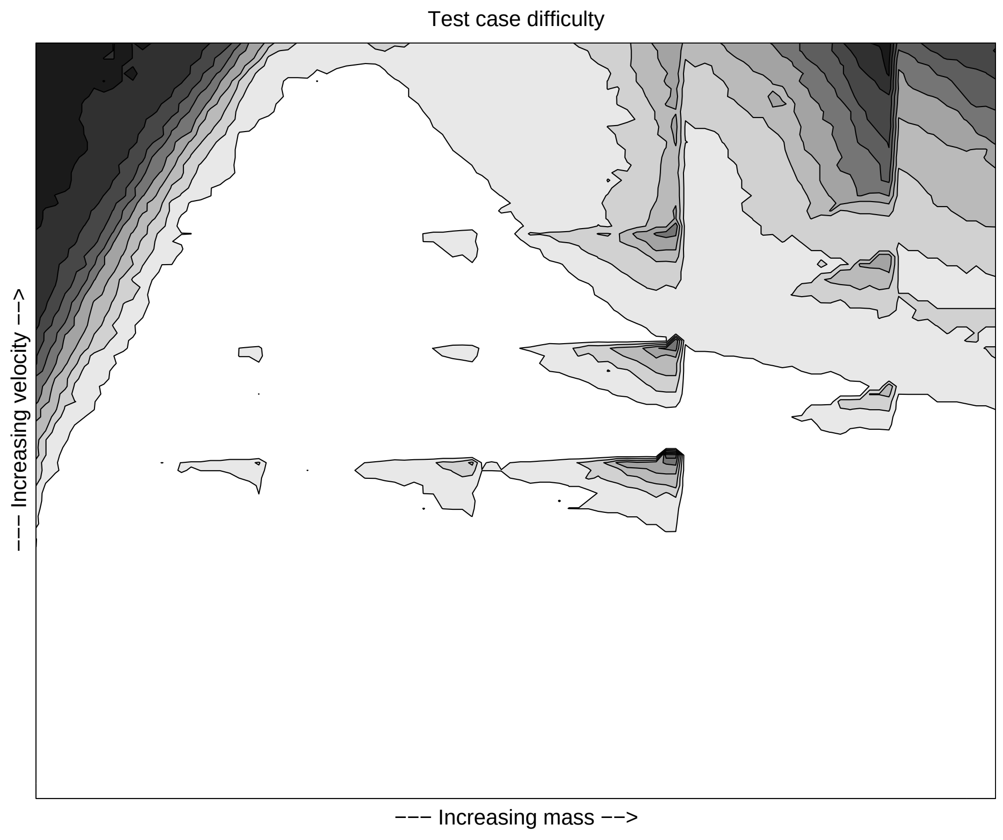


Figure 1. Contour diagram showing the difficulty of test cases with different mass and velocity

Darker areas indicate higher difficulty, i.e. indicating test cases where a larger proportion of the programs fail.

Detailed analysis of the diagram reveals that there are three main areas of difficulty. Visually these areas are located in the upper left corner, in equidistant clusters in the center and in the upper right corner, respectively.

For the upper left corner, where aircraft have high velocity and low mass, the programs generally fail because they violate the requirement on the maximum retardation on the pilot. It seems plausible that these failures arise because the programs do not properly measure and/or use a notion of the mass of the incoming aircraft in their control algorithm.

The failures in the center area are mainly due to excessive forces applied to the hook on the aircraft attaching to the cable of the braking system. A requirement in [USAF86] states the maximum allowed hook force for certain "points" with specified mass and velocity. The clusters of failing programs seen in the center of figure 1 are located below (lower velocity) and to the left (lower mass) of these points. These are the areas where the energy of the aircraft is at a maximum for the requirement of maximum hook force.

The failures in the upper right corner are made up of a combination of failing to restrict the force on the aircraft hook and failing to brake the aircraft before the end of the runway. The former can be explained by the same reasoning as above and the latter arises because the energies of the aircraft take on their largest values this area. If the programs do not exert a high enough brake pressure in the start of the braking they will not have time to brake the aircraft before the critical length.

4. Discussion

The experimental results imply that the central challenge in this problem is to cope with the high energies of heavy airplanes having large velocities. Failure to meet the OVERRUN criterion is fatal in that the aircraft has not come to a halt. Even though a large majority of the programs also have problems with the RETARDATION criterion for light airplanes with high velocities, detailed analysis of the braking behavior shows that the programs often only slightly exceed the threshold of 2.8g maximum retardation. The center clusters of difficulty show failures on the HOOKFORCE criterion but it is a smaller proportion of programs that fail here than on the previously mentioned criteria. The discontinuous property of these clusters can be attributed to the fact that the maximum hook force was specified at certain points in the requirements. The CABLE criterion was not violated by any of the programs. Detailed analysis of the experimental data also reveals that some programs make a trade-off between meeting different requirements. They trade better performance on the RETARDATION criterion for worse performance on the OVERRUN criterion. Other programs do the opposite. This indicates that there is a fundamental trade-off to be made between these criteria.

The knowledge gained by analyzing the behavior of the GP-developed programs could be used in a discussion with the client stating the requirements. It can be used to resolve questions, prioritize the requirements and show requirements that may be too easy to fulfill, indicating cost savings. An example of the former is to question the HOOKFORCE criterion that is only specified in certain points. Is this really what the client want or is it wiser to specify the requirement as a function of the incoming mass and velocity? The fact that the cable criterion is never violated might indicate that too strong a cable is used in the system. Based on the information gained from the experiment a weaker, but possibly less expensive, cable can be proposed. A suggestion to prioritize the requirements can be based on the trade-off between not retarding light airplanes too much and being able to brake high-energy airplanes at all. If the latter is preferred the developers can propose that a slight increase in the threshold for the RETARDATION criterion might change the difficulty, and therefore the cost, of designing a solution. Or they can question whether light airplanes with high velocities can be expected to appear.

It should be noted that the reported experiment is limited in several ways. Firstly, the target application is small, having few requirements and a low-dimensional input space. This latter feature makes it particularly well suited for visualization; with a high-dimensional input space visualization will be more difficult. Furthermore, an existing simulator could be used to evaluate the fitness and failure behavior of the programs. In general a simulator will not be available in early software development phases and it is unclear if the cost for developing one is motivated by the knowledge that can be gained. Further research is needed to clarify this.

Secondly, the presented experiment was not primarily designed to evaluate the SPE-GP idea put forward in this paper. For example, it would have been interesting to alter some of the requirements to assess their effect on the difficulty levels and their distribution.

In addition to these practical considerations there is a more fundamental question that needs to be addressed. If we are to gain any useful knowledge about the software problem at hand the behavior of the GP-developed programs must be representative for programs developed by human software developers. There is a risk that the behavioral data will be more GP-specific than problem-specific. An example of this is seen in our pilot experiment that is really showing the difficulty for *memory-less*[2] *controller programs with certain functions and terminals*. If the controllers were allowed to use, for example, indexed memory the difficulty landscape showed in figure 1 might change. A similar concern can be raised because of the limited applicability of present-day automatic programming systems, such as GP.

Provided that the SPE-GP technique can be used on more complex problems, the knowledge gained from using it can be compared to the outcome of the actual development project. For example, the average difficulty level obtained from using an SPE-GP scheme can be compared to the total development time. If, over several projects, correlations are found the average difficulty level can be used in project and resource planning.

Motivated by the deficiencies in the pilot experiment and the outstanding issues that need to be resolved in order to evaluate the power of SPE-GP we outline an extended environment for SPE-GP in the following section.

5. An extended environment for software problem exploration using genetic programming

In an extended environment for software problem exploration using genetic programming we need to address the problem of a high-dimensional input space. The two main problems with having many input parameters are the problem of selecting test cases for visualization and the problem of visualizing high-dimensional data. The former arises because there is a combinatorial explosion in the number of possible input cases; for most systems exhaustive testing of all possible combinations is not feasible. The latter problem arises because the high-dimensional data needs to be presented on our low-dimensional output devices.

One possible solution to the testcase-selection problem is to let the developer do the selection. Even though the developer might not be expected to have much knowledge about the problem they might have a hunch about what areas are more or less interesting. This scheme requires interaction between the developer and the SPE-GP environment. Another possible solution to the testcase-selection problem would be to use information from the GP development phase to guide the exploration and visualization system to the most difficult areas of the input space. One possible solution would be to let the testcases used during evolution, i.e. the fitness cases, be co-evolved with the programs.

There are a number of techniques for extracting knowledge from high-dimensional data. Often they involve some statistical technique for dimensional reduction, such as principal component analysis or multidimensional scaling, or use clever coding schemes to map the multiple dimensions to our 2D or 3D display devices [Keim97]. One promising approach is the Visor algorithm of König et al [König94]. Their system combines methods for feature extraction and class separation with a fast algorithm for visualizing high-dimensional data in 2D. These abilities can be useful in isolating clusters of similar test cases in the input space.

[2] The evolved programs can not access the memory used to calculate the terminals used in the programs.

Techniques for interactive exploration have also been studied in the area of visual database exploration [Keim97].

The environment we envisage is a developer's workbench for exploring the effects of different requirements and finding the difficult, and therefore crucial, areas of the input domain. With faster computers and more advanced display techniques one can imagine such a system performing the evolution of the programs in real-time in response to developer queries.

Figure 2 shows a conceptual diagram of an SPE-GP environment according to the discussion above. On the left side is the GP system developing a pool of programs that are tested on the right side to produce data that is displayed for the developer. The developer can interact with the visualization system to select areas of interest or display the data in different ways. The developer queries are sent to the test controller that uses existing data from a database or performs additional tests of the programs. The user can also alter the requirements and develop new programs that can give further information. Also note the evaluation controller supplying fitness cases to the evaluation of the programs in the development system. These fitness cases can be used as starting points for the tests in the exploration system.

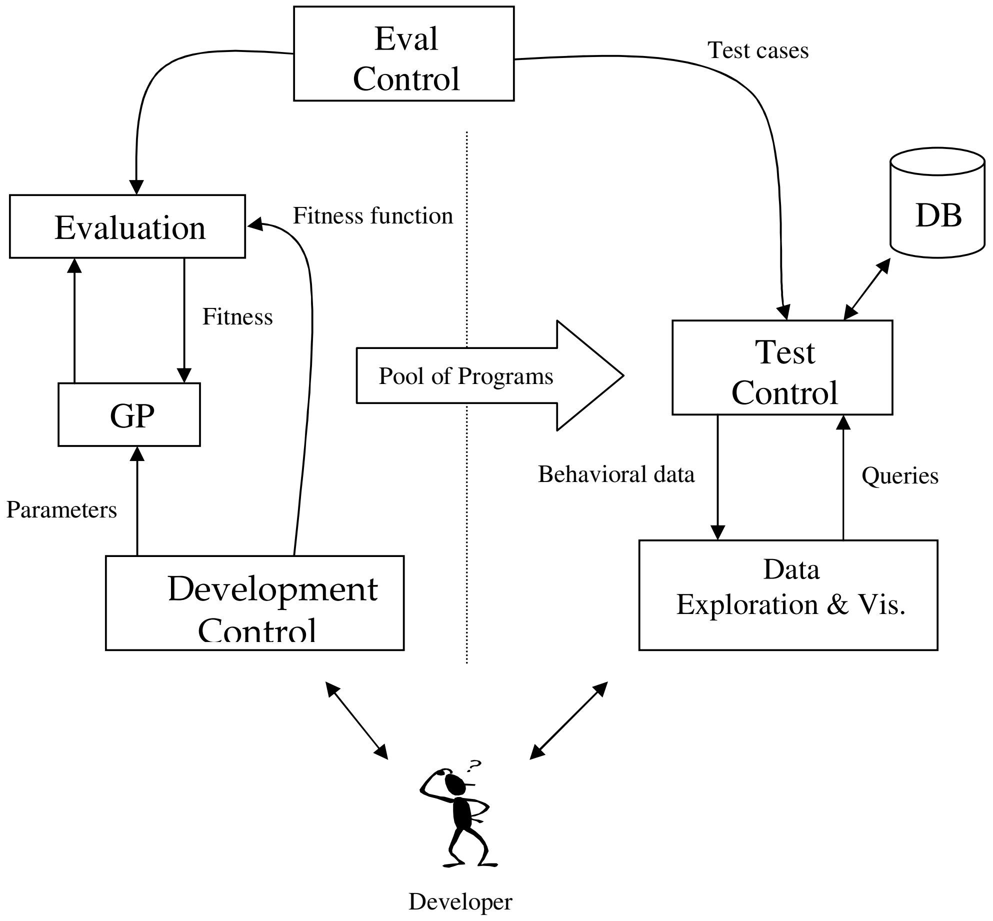


Figure 2. Conceptual diagram of a SPE-GP environment

6. Conclusions

Early in a software development project the software engineers lack knowledge about the problem to be solved by the software. Any knowledge that can be gained at an early stage can reduce the risk of making erroneous decisions and injecting defects that can be expensive to eliminate in later phases. Having such knowledge could thus lead to lower development costs and possibly a higher quality. Furthermore, it could be used in project planning to predict resource need. We propose that techniques for machine learning, especially genetic programming, be used to explore the software problem to be solved and to gain knowledge about difficult areas of the input space, the effects of different requirements and to identify design trade-offs that need to be addressed.

In an initial experiment, where genetic programming was used to develop 400 controller programs for an aircraft arrestment system, three main areas of difficulty in the input space were identified. These areas were related to the requirements, and the knowledge gained could be used to question and prioritize the requirements as well as indicating areas for cost savings. However, the target system was simple with few requirements and a low-dimensional input space well suited for visualization. For problems with high-dimensional input spaces it is unclear how to select the test cases for testing the programs and how to visualize the resulting data in a meaningful way for the human developers. We point out some possible solutions to these problems and outline a general environment for *software problem exploration using genetic programming* (SPE-GP).

The central outstanding research question is to establish what data on the behavior of programs developed using genetic programming tells us about the difficulties facing human developers. It is not likely that a general answer can be given to this question; the amount of knowledge gained will likely vary with the actual problem and the power of the genetic programming system used. We think that the importance of this software engineering problem motivates further study.

References

Banzhaf, Wolfgang et al. Genetic Programming – An Introduction. Morgan Kaufmann, San Fransisco, California, 1998.

Christmansson, Jörgen. An exploration of models for software faults and errors, PhD Dissertation, Department of Computer Engineering, Chalmers University of Technology, 1998.

Feldt, Robert. Generating Multiple Diverse Software Versions Using Genetic Programming - an Experimental Study. IEE Proceedings – Software, vol. 145, Issue 6, December 1998.

Humphrey, Watts. A discipline for software engineering, Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-54610-8.

Keim, D. A. Visual Techniques for Exploring Databases, Invited Tutorial, Int. Conference on Knowledge Discovery in Databases (KDD'97), Newport Beach, CA, 1997.

König, Andreas, Buhlman, Olaf, Glesner, Manfred. Systematic Methods for Multivariate Data Visualization and Numerical Assessment of Class Separability and Overlap in Automated Visual Industrial Quality Control, In *Proceedings of the 5th British Machine Vision Conference* (BMVC'94), pages 195-204, Sept. 1994.

Sommerville, Ian. Software Engineering, Addison-Wesley, Workingham, England, 1992. ISBN 0-201-56529-3.

Quereshi, Adil. GPSys 1.1 Homepage. Web Page. URL: http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html. 20 November 1998.

US Air Force. Military Specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction. 1986. MIL-A-38202C, Notice 1.

Paper 5.

Robert Feldt. *Forcing Software Diversity by Making Diverse Design Decisions - an Experimental Investigation*, Technical Report no. 98-46, Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, December 1998.

Forcing Software Diversity by Making Diverse Design Decisions – an Experimental Investigation

Robert Feldt
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Tel: +46 31 772 5217, Fax: +46 31 772 3663
E-mail: feldt@ce.chalmers.se

Abstract

When developing software versions for a multi-version system, the probability for coincident failures may be decreased by forcing the development efforts to be different by making diverse design decisions. There are theorems showing that the probability is minimized by making as diverse design decisions as possible but it is not known if the assumptions made in proving the theorems are valid in practice. To investigate this we have developed 435 versions of a software controller for an aircraft braking system. The versions were developed using genetic programming. Analyses of the failure behavior of these versions showed that the assumptions of failure independence among the decisions were valid, on average, for 74% of the test cases. The assumption of indifference between methodologies were not valid in a single case which seems to be the major cause invalidating the theorem. Thus, if we are not indifferent between design decisions, it is not guaranteed that increased diversity of design decisions will decrease the probability of coincident failures.

1. Introduction

N-version programming (NVP) has been proposed as a technique to develop multiple versions of the same software for fault-tolerant software systems [Avizienis95b]. In the case in which the versions fail independently of each other the reliability of a multi-version system could be significantly larger than the reliabilities of single versions. However, empirical and theoretical results have shown that independent failure behavior can not be expected; apart from the difficulties of making the development efforts independent varying difficulty of the input data guarantees that the versions will not fail independently [Knight86] [Bishop95]. In [Littlewood89], Littlewood and Miller proved theorems showing that the attainable reliability levels can be higher than if independence is assumed if we force the development efforts to be different. Thus, by actively making diverse design decisions in the different development efforts we can force software diversity, i.e. diversity in the structure and / or failure behavior of the resulting software versions.

The theorems of Littlewood and Miller further indicate how the diversity of the development efforts should be forced: the design decisions should be as diverse as possible to decrease the probability of coincident failure [Littlewood89]. In order to prove the theorems about the effectiveness of forced design diversity, Littlewood and Miller made a number of assumptions. They pointed out that these assumptions might not be valid in practice. In this paper we study if these assumptions are valid on a particular application and assess how the validity of a theorem of the effectiveness of forced diversity is affected when the assumptions are not fully valid.

To carry out these analyses in a statistically rigorous way we need failure data from a large number of versions developed with a number of different design decisions. The cost of conducting such an experiment would be prohibitively high. In a previous study we have introduced a procedure for forced design diversity using genetic programming to develop the software versions [Feldt98b]. Genetic programming (GP) is a technique for searching for computer programs with desirable properties. In the previous study we showed that failure diversity can be forced by varying parameters to a GP system and proposed that this technique can be used as a research tool in software fault-tolerance [Feldt98a]. In this paper we have used the technique as a tool to develop software controllers for an aircraft braking system.

In section 2 we recapitulate the results on forced diversity by Littlewood and Miller and state the research questions we investigate in this study. Section 3 describes the method we have used; it introduces genetic programming, the target application and the experiments we have carried out. The results from the experiments are given in section 4 followed by a discussion in section 5. Finally, we summarize the conclusions that can be drawn from this study and indicate future work.

2. Forced design diversity

In their seminal paper, Littlewood and Miller extended the model of multi-version program development introduced by Eckhardt and Lee, to account for diverse development methodologies [Littlewood89]. In the model, a development methodology is characterized by a mapping, S, giving the probability that a particular program, π, is chosen from the population of all possible programs, $\wp$, i.e.

$$P(\Pi = \pi) = S(\pi)$$

where Π is a random variable representing the random selection of a program. A key average performance measure is $\theta(x)$, giving the probability that a randomly chosen program fails for the input case x (the model can handle varying probabilities of input cases but we do not consider this in the present study; we assume that all input cases are equally probable). For a randomly chosen input X, $\theta(X)$ is a random variable Θ.

To model diversity between development methodologies Littlewood and Miller introduced the notion of a *design decision*. An example of a design decision would be the choice of testing strategy. In general, design decisions can have multiple levels but we focus on binary decision and associate ‘0’ and ‘1’ with the two possible outcomes of such decisions.

A development methodology is defined by the outcomes of a number of design decisions and can be described with a binary vector. For example, in the experiments in this paper there are seven binary design decisions and methodology number nine is defined by the descriptor

$$(0, 0, 0, 1, 0, 0, 1)$$

A natural measure of design diversity, i.e. the degree of diversity between development methodologies, when using these binary vectors is Hamming distance. We thus define

> the *developmental diversity*[1] between two methodologies M_1 and M_2 , denoted
> $\delta(M_1 , M_2)$, as the number of positions in which the descriptors of the methodologies differ, i.e. their Hamming distance.

The developmental diversity can be used to state an important theorem on forced diversity (theorem 3 in [Littlewood89]) in natural language as: "The greater the developmental diversity between two methodologies, the lesser chance for coincident failures of two versions developed with the methodologies". As Littlewood and Miller point out the theorem holds on average, when we consider all the possible programs that might be developed with a methodology and check the behavior on all possible input cases; for particular input or programs it might not hold.

The above theorem is important since it implies a rule for how to develop multi-version software. If we choose methodologies that have maximum developmental diversity, the probability of coincident failures is decreased and the reliability of the system should increase.

The proof of the theorem relies on the Cauchy-Schwarz inequality and the following three assumptions (for a pair of binary design decisions):

- A1, *Decision choice independence*: The choice taken in the different design decision should be independent of each other, so that
  $$P(\Pi \in (1,1)) = P(\Pi \in (1,*)) \rceil P(\Pi \in (*,1))$$
  where * is used to mark all possible outcomes of the decision.
- A2, *Decision failure independence*: There is no interaction between the design decisions in their effect on the failure behavior, so that
  $$P(\pi \in (1,1) \mid \pi \text{ fails on } x) = P(\pi \in (1,*) \mid \pi \text{ fails on } x) \rceil P(\pi \in (*,1) \mid \pi \text{ fails on } x)$$
  for all input cases x.

[1] We use this term to distinguish it from the more general notion assigned to 'design diversity'.

- A3, *Methodology indifference*: We are indifferent between methodological choices related by permutations of labels. Two specific instances are used in the proof:
  - A3a, Indifference for methodologies with developmental diversity 0:
    $E(\Theta_{11}, \Theta_{11}) = E(\Theta_{01}, \Theta_{01}) = E(\Theta_{10}, \Theta_{10}) = E(\Theta_{00}, \Theta_{00})$
    where $E(\Theta_{M1}, \Theta_{M2}) = P(\Pi_{M1}$ and Π_{M2} both fail on X) = probability that randomly chosen programs from two methodologies both fail on a randomly chosen input.
  - A3b, Indifference for methodologies with developmental diversity 1:
    $E(\Theta_{11}, \Theta_{01}) = E(\Theta_{11}, \Theta_{10}) = E(\Theta_{00}, \Theta_{10}) = E(\Theta_{00}, \Theta_{01})$

In their paper, Littlewood and Miller point out that the assumptions they make in proving their theorems might not be valid in practice. In this study we assess the assumptions empirically. Furthermore, we want to investigate how the validity of the theorem is affected when there are deviations from the assumptions above. The questions we want to answer in this study can thus be summarized as (for an application X):

1. Is the assumption of failure independence between factors (A2 above) valid for the application X?
2. Are the assumptions of indifference (A3a and A3b) valid for the application X?
3. If the assumptions (A2 or A3) are not fully valid, how does this affect the validity of the theorem? Is it still valid but in a restricted form? Which of the assumptions affect the validity of the theorem the most?

We do not investigate assumption A1 since we can make sure that it is fulfilled by the choice of design decisions.

3. Method

We have developed 435 versions of a software controller employed in an aircraft braking system. The versions were developed with a genetic programming system running on a SUN Enterprise 10000 with the Sun Solaris OS 2.5. The experiment environment, consisting of the genetic programming system GPSys, a simulator of the application and custom-developed software to interface between them was developed in Java and compiled with the java-to-C compiler Toba for increased performance [Quereshi98] [Proebsting96] [Feldt98a]. After development all software versions were subjected to the same 10000 test cases. The failure behavior on these test cases was analyzed to answer the research questions of this study.

Below we give a brief introduction to genetic programming and describe the application and the experiment we have conducted. Further details on the application can be found in [Feldt98a] and [Christmansson98]. Figure 1 shows an overview of the experiment environment

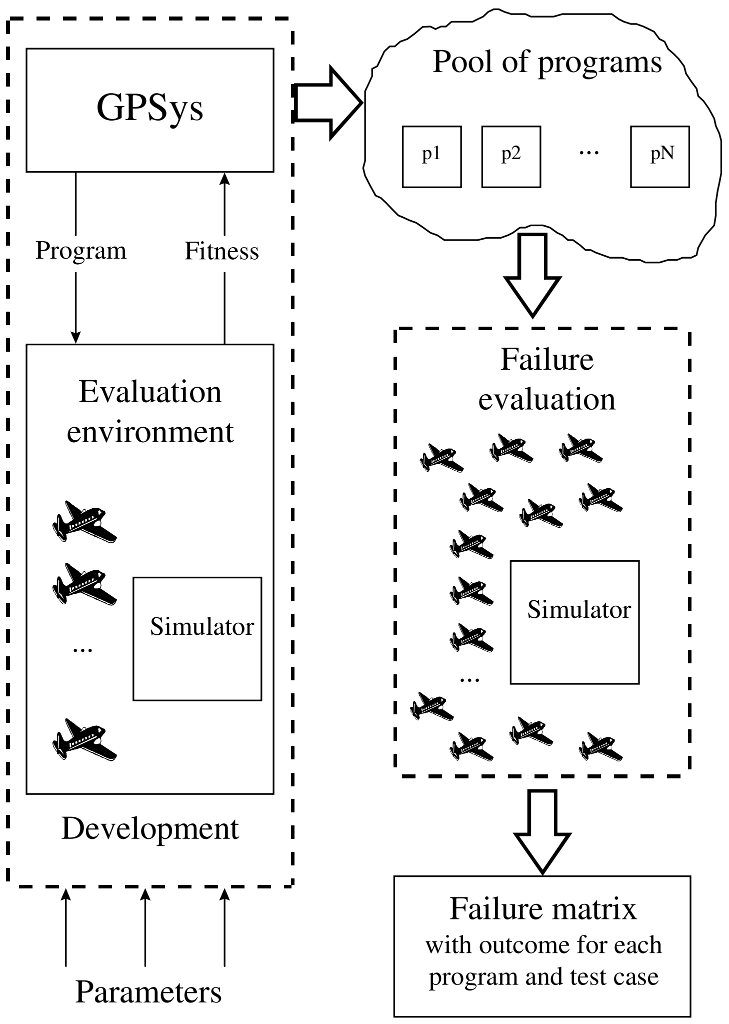


Figure 1. The experiment environment

3.1 Genetic programming

Genetic programming is an algorithm for searching for computer programs with desirable properties. The 'genetic' part of its name is used because the search process shows some resemblance with evolutionary processes in biological systems. The technique was introduced in 1992, is studied under the heading of Evolutionary Computation and have been successfully applied on a number of different problems [Koza92] [Banzhaf, *et al.*98] [Bäck97]. Using biologically inspired ideas in the research and design of fault-tolerant computers has been previously proposed in [Avizienis95a].

The search space searched by a genetic programming (GP) algorithm can be defined by the user. Parameters govern which basic building blocks, such as variables, constants and functions can be used in the programs. Other parameters govern how the search should proceed in the search space and how the programs should be tested during development. The tests carried out during development are called fitness evaluations; they assess how 'fit' each program is, i.e. how well it adheres to the specification.

Genetic programming is a parallel search procedure having multiple programs that are used as starting points for further explorations of the search space. In each iteration of the algorithm the fitness of all the programs is evaluated. Programs are selected based on how fit they are and combined into new programs by swapping parts between them. Old programs with low fitness are deleted. When this scenario is repeated there is a tendency that, after a large number of iterations, programs with increasingly higher fitness are found. The search procedure is stochastic by nature and running it repeatedly can give different results.

In a previous study we have proposed that diverse software versions could be developed by systematically varying parameters to a GP system [Feldt98a]. In this proposal, the choice of the values of a parameter is analogous to a design choice in Littlewood and Millers model. In an empirical investigation we showed that diverse failure behavior was obtained even though it was limited for the parameter settings giving the lowest failure rate. For more information on these results and on GP in general see [Feldt98a] [Banzhaf, *et al.*98].

3.2 Application

The target application is designed to arrest aircrafts on a runway. Incoming aircraft attach to a cable and the system applies pressure on two drums of tape attached to the cable. A computer that determines the brake pressure to be applied controls the system. By dynamically adapting the pressure to the energy of the incoming aircraft the program should make the aircraft come to a smooth stop. The requirements on a system like this can be found in [US Air Force86]. The system has been used in previous research at our department and a simulator simulating aircraft with different mass and velocity is available. The system is more fully described in [Christmansson98].

The main function of the system is to brake aircraft smoothly without exceeding the limits of the braking system, the structural integrity of the aircraft or the pilot in the aircraft. The system should cope with an aircraft having maximum energy of $8.81*10^7$ J and mass and velocity in the range 4000 to 25000 kg and 30 to 100 m/s, respectively. More formally the program should[2] (name of corresponding failure class in parentheses)

- stop aircraft at or as close as possible to a target distance (275 m)
- stop the aircraft before the critical length of the tape (335 m) in the system (OVERRUN)

[2] Our system adopts the requirements of [US Air Force86] with the addition of the allowed ranges for mass and velocity and a critical length of 335 m (950 feet in [US Air Force86]).

- not impose a force in the cable or tape of more than 360 kN (CABLE)
- not impose a retarding force on the pilot corresponding to more than 2.8g (RETARDATION)
- not impose a retarding force exceeding the structural limit of the aircraft, given for a number of different masses and velocities in [US Air Force86] (HOOKFORCE)

The programs are allowed to use floating point numbers in their calculations. They are invoked for each 10 meters of cable and calculate the brake pressure, for the following 10 meters, given the current amount of rolled out cable and angular velocity of the tape drum.

An existing simulator of the system has been ported from C to Java. It implements a simple mechanical model of the airplane and braking system and calculates the position, retardation, forces and velocities in the system. It does not model the inertia in the hydraulic system or oscillatory movement of the aircraft due to elasticity in the tape. The simulator has been set to simulate braking with a time step of 62.5 milliseconds.

During development of the programs GPSys invokes the simulator and tests the programs on a number of test cases, i.e. aircraft with different masses and velocities. Critical values from each braking are used to evaluate if a program has violated any of the requirements above. A penalty is assigned if there is a violation for any of the failure classes above.

3.3 Experiments

To answer the research questions stated in section 2 we need to choose two design decisions and develop programs with the resulting four methodologies. The outcome of such an experiment could be highly dependent on the particular choice of decisions. To lessen this sensitivity to design choices we have identified a group of seven design choices and studied all C(7,2) = 21 combinations of two choices from them. The seven design choices were chosen based on our previous experience with the experiment environment; we excluded choices with a negative effect on the failure rate of the resulting programs. This is similar to what we would do if we were to develop multiple versions using human software developers; we would not take design decisions that are known to give high failure rate just to increase the diversity of the software versions.

The design choices and their levels, i.e. the outcomes of the decisions, are shown in table 1. Decision number one concerns the testing performed by the GP system during evolution of the programs. When on level zero the test cases are equidistantly spread on the allowed range of mass and velocity of the incoming airplane. On level one the values are randomly assigned from a uniform distribution.

Design decision	Level	Description
1	0	36 uniformly spread test cases are used to evaluate fitness.
	1	25 randomly sampled test cases are used to evaluate fitness.

2	0	A guiding penalty of 2% of maximum penalty is used.
	1	A guiding penalty of 50% of maximum penalty is used.
3	0	Maximum penalty on the HALTDISTANCE failure criteria is 1000.0.
	1	Maximum penalty on the HATLDISTANCE failure criteria is 3000.0.
4	0	Programs can use the pressure at the previous checkpoint and the indices to the current and previous checkpoints.
	1	Programs can use the current time since start of the braking, the angular velocity and time at the previous checkpoint
5	0	Programs cannot use any subroutines.
	1	Programs can use a subroutine.
6	0	No effect.
	1	Programs can use the statement IF, and operators LE, AND and NOT.
7	0	Programs can use the function SIN and the terminal PI (3.1415).
	1	Programs can use the function EXP.

Table 1. The design decisions and their effect

Decisions two and three determine the fitness function used to rank the programs during development. For this particular application, the fitness function is calculated as a penalty assigned as to how much a certain requirement is violated. The fitness score is a sum of penalties on four criteria, with a penalty of 1000 units assigned when the requirement is violated, i.e. the program fail on the criteria. Decision 2 sets the level of the guiding penalty assigned to grade how far from fulfilling the requirement the programs are. This 'guides' the development in the direction of requirement fulfillment. Decision 3 defines the maximum penalty on the HALTDISTANCE criteria. When on its high level, it indicates that we consider the HALTDISTANCE criteria to be the most important criteria. The programs have something to gain from trying to fulfill this criterion with high priority.

Decision 4, 6 and 7 govern which variables (4) and functions (6 and 7) the programs can use. Decision 5 governs the structure of the programs. When at its high level the programs can use a subroutine.

Twenty-nine different methodologies were constructed from these seven design decisions. They are all the twenty-one methodologies with all pairs of two decisions chosen from the seven at their high level, the seven methodologies with each single decision at it's high level and the methodology with all decisions at its low level. Decisions that were not involved in deciding the methodology were held at their '0' level. This choice of methodologies allows us to test the validity of the theorem for the 21 different pairs of design decisions. Each methodology was used to develop 15 versions resulting in a total of 435 versions and each version were tested on the same 10000 test cases. The test cases were equidistantly spread on the allowed ranges for mass and velocity.

4. Results

In this section we describe the results from the analysis of the failure behavior of the 435 programs. We analyze, in turn, assumption A2, A3 and the validity of the theorem. This corresponds to research questions 1, 2 and 3 in section 2.

4.1 The assumption of decision failure independence (A2)

For each pair of decisions we can test assumption A2 four times: one for each combination of decisions 00, 01, 10 and 11. For each of these 21*4 = 84 tests we collected the failure data of the 4*15 = 60 versions relevant to the test. For each test case where at least one variant failed we calculated the dependency ratio

$$r_d = P(\Pi \in (1,1)) \,/\, (P(\Pi \in (1,*)) \rceil P(\Pi \in (*,1)))$$

which according to assumption A2 should be exactly one (we assigned the value one when both numerator and denominator were zero).

Table 3 below gives some summary statistics for the 84 tests. Note that the maximum percentage of ratios equal to 1 were 79.4%.

	Test cases with failure(s)	$r_d < 1$	$r_d = 1$	$r_d > 1$	Average r_d	Stdev of r_d
Max	6087	23.9%	79.4%	23.9%	1.0936	0.3889
Average	3955	12.9%	74.2%	12.9%	0.9993	0.2606
Min	2459	4.2%	69.6%	4.2%	0.9192	0.1764

Table 3. Summary statistics for the 84 tests of assumption A2

Only one pair of decisions had a balanced distribution of dependency ratios; the rest were either skewed to the right or to the left. However, when all ratios were considered together the distribution was balanced with equal number of ratios below and above independence at one.

4.2 The assumption of methodology indifference (A3)

To check assumption A3 we calculated the θ(x)-vectors for all 29 methodologies. For our experimental set-up assumption A3a implies that all the expectations

$$E(\Theta_M , \Theta_M)$$

where M is a methodology, should be equal. In our experiment, they are not. The average, minimum, maximum and standard deviation of these expectations are shown in table 4.

Max	0.0636
Average	0.0509
Min	0.0371

Standard deviation	0.0057

Table 4. Summary statistics for 29 expectations between methods with developmental diversity 0

Assumption A3b implies that all expectations

$$E(\Theta_{M1}, \Theta_{M2}) \text{ where } \delta(M_1, M_2) = 1$$

should be equal. In our experiment, they are not in a single case. There are 49 expectations of this kind and their summary statistics are shown in table 5.

Max	0.0573
Average	0.0482
Min	0.0379
Standard deviation	0.0042

Table 5. Summary statistics for 49 expectations between methods with developmental diversity 1

4.3 Validity of theorem

The theorem of Littlewood and Miller gives an ordering between groups of expectations having equal developmental diversity. For each pair of decisions there are three groups corresponding to the developmental diversities of 2, 1 and 0. The first group contains two elements and the latter groups four elements each. For each pair of decisions, we tested if the ordering of the theorem was valid. We also compared the averages of and the minimum expectation in the different groups. The results from these comparisons are shown in table 6.

	$\delta = 2$ vs. $\delta = 1$	$\delta = 1$ vs. $\delta = 0$
Theorem: Max < Min	0 (0.00%)	0 (0.0%)
Average < Average	17 (80.95%)	21 (100.00%)
Min < Min	5 (23.81%)	12 (57.14%)

Table 6. Number of cases where comparisons between groups of equal development diversity are valid

When we grouped the different expectations in the respective groups together and performed an analysis of variance there were statistically significant differences between the groups ($p < 0.01$). The average expectations for the three groups were 0.0476, 0.0482 and 0.0509 respectively. Thus, the ordering prescribed by the theorem is valid "on average". The ANOVA table is shown below.

Source	Sum of squares	Degrees of freedom	Mean square	Ratio	Significance

Groups with equal dev diversity	$1.98*10^{-4}$	2	$9.91*10^{-5}$	5.88	0.0037
Residuals	$1.97*10^{-3}$	117	$1.68*10^{-5}$		
Total	$2.17*10^{-3}$	119			

Table 7. Analysis of variance table

To assess the individual effects of the different assumptions we repeated the test on the validity once more, this time only including testcases where assumption A2 was valid. The test cases to include were chosen individually for each pair of design decisions. The ordering prescribed by the theorem was not valid in any of the 42 cases. The full result of these comparisons is shown in table 8.

	$\delta = 2$ vs. $\delta = 1$	$\delta = 1$ vs. $\delta = 0$
Theorem: Max < Min	0 (0.00%)	0 (0.00%)
Average < Average	18 (85.71%)	21 (100.00%)
Min < Min	6 (28.57%)	12 (57.14%)

Table 8. Result of comparisons when expectations are calculated on test cases where assumption A2 holds

5. Discussion

We have separated the discussion of the results in two parts. In the first we consider the results without acknowledging the fact that the programs have been developed with genetic programming. In section 5.2 we discuss how this fact might affect the validity of our results.

5.1 Implications of the results for forced design diversity

The results tell us that, for this particular application and with this particular choice of design decisions, the theorem on the effectiveness of forced diversity is not valid. However, this should not be expected since its assumptions are not fulfilled.

Assumption A2 was on average valid in about 74% of the test cases where at least one version failed. Furthermore, a majority of methodologies had skewed distributions with dependency ratios as high as 1.09 and as low as 0.91. This supports the intuitive notion pointed out by Littlewood and Miller that assumption A2 can not be expected to be valid in practice since there will be interactive effects between the design decisions. An example scenario of this would be if one design decision would be between programming languages where only one of them supports automatic garbage collection and memory handling (such as between C and Java) and a decision on testing governing if the tool `purify` is used to analyze the memory behavior during execution. An interactive effect between these design decisions seems plausible.

In our experiment, assumption A2 does not seem to be a major cause of theorem invalidity. When we tested the validity of the theorem and forced assumption A2 to be fulfilled this did not increase the level of theorem validity. This indicates that it might not be crucial for this assumption to be fulfilled. However, more experiments are needed to settle this.

If A2 is not the major cause of the invalidity of the theorem then this must be due to the unfulfillment of A3, the assumption of indifference. For all of the 21 pairs of design decisions and all their three groups of methodologies with equal developmental diversity there is some spread of the expected failure probabilities. Thus, the assumption of indifference is not valid in a single case. The expectations in each group are spread out so that there is some overlap between the groups. This causes the invalidity of the ordering that would be seen if the theorem would be valid.

It is interesting that the theorem is still valid in a majority of cases if we look at the averages of each group of developmental diversity. Analytical investigations of this might reveal extensions of the theorem that need not assume strict indifference. One possible way to go would be to express the theorem in terms of the amount of variation between expectations in the same groups. Such investigations would be important since it seems unlikely that we, in practice, will be strictly indifferent between methodologies. Interactive effects will likely invalidate the indifference assumption.

These results complicate the picture for practical development of multi-version systems. If indifference is typically not the case, larger developmental diversity will not guarantee smaller chance of coincident failure; variations among the combinations of methodologies with equal developmental diversity can alter the ordering. Further empirical and analytical investigations of these questions are needed to clarify the picture.

We would like to stress an important point stated by Littlewood and Miller: in the case of ignorance of the effect of different decisions the assumption of indifference will be plausible and, for this "state of knowledge", the theorem will be valid. However, over several projects developers might build up experience on the effects of different design decisions that can be used to take more informed design decisions. Extending the theorems of forced design decisions to guide these decisions seems worthy of future studies. One result in this direction were conjectured by Littlewood and Miller and a natural extension of this study would be to assess this conjecture and how it can be used when we have knowledge on the effect of different design decisions. Another important task for research on forced diversity will be to investigate the effect of different design decisions on diversity.

The model of multi-version development used by Littlewood and Miller uses the set of all possible programs that could be developed with a methodology and the set of all possible input cases to the programs. In our experiment we have sampled from these sets and all our results includes uncertainty as to these samples. We are confident that the sample from the input cases does not affect the results. In previous experiments with this application the qualitative results were not affected when the number of test cases was increased; only minor alterations in the quantitative results occurred. This could be due to the fact that our input space is simply a two-dimensional rectangle of the plane and we can cover it sufficiently good with a relatively small number of test cases. Furthermore, our application is a fairly simple controller with “continuos” characteristics. However, for applications with more complicated input spaces this would be an important issue.

To assess the effect of the sampling of the programs we repeated all the calculations and analyses reported above for different numbers of programs from each methodology in the range from 4 up to 15. This showed a convergence to the results reported above at the number of 7. This indicates that the results have converged and will not be seriously affected by increasing the size of the sample any more. However, a full analysis taking statistical inference into consideration at each step would be valuable.

5.2 Using genetic programming to investigate design diversity

Genetic programming is different from ordinary software development. It can be questioned if the results and analyses reported here are valid for software developed by humans. Our stance is that statistical theorems do not differentiate between programs because they have been developed using different techniques; the theorems should be valid in general.

There are a number of advantages of using an automated technique, such as genetic programming. A large number of versions can be developed at a relatively low development cost and parameters can be systematically varied to emulate different design decisions. These characteristics are important to get statistically significant results.

Of course, there are also a number of disadvantages of using genetic programming. The failure probabilities are typically higher than would be the case if we developed the software by hand. It is possible that this affects the results so that other behavior will be observed for lower failure probabilities. Furthermore, genetic programming has mostly been applied to smaller problems and the applicability of our approach is directly tied to the applicability of GP. If GP can not be used on larger and more complex problems then neither can our approach. These issues are discussed in more detail in [Feldt98a]. To settle them studies that compare the use of genetic programming with ordinary software development are needed. We are in the process of approaching researchers in the fault-tolerance area that have conducted studies with replicated-run replicated-variant experiments to try to make such studies possible [Knight86] [Kelly83].

In summary, we think the technique of developing multiple software versions using genetic programming can be used to investigate the theoretical limits of diversity and multi-version software. It can be used as a research tool to explore new possibilities and increase the knowledge on software and design diversity. This could help minimize the cost and increase the effectiveness of conducting research on multi-version software using human software developers.

6. Conclusions

We have developed 435 program versions of a software controller braking aircraft coming in to land on a runway. The versions were developed using an automated program searching technique called genetic programming. By varying parameters to the genetic programming algorithm we obtained 29 different development methodologies and developed 15 versions with each of these methodologies. By analyzing the failure behavior of the programs we tested the validity of a theorem indicating how to force diversity, stating that larger diversity between development methodologies, what we call developmental diversity, gives a smaller probability of coincident failures [Littlewood89].

We tested two assumptions that need to be fulfilled for the theorem to be valid: the assumption of independent failure behavior between design decisions (A2) and the assumption of indifference (A3). Assumption A2 was fulfilled in average on about 74% of the test cases while assumption A3 was never fulfilled. In accordance with this the theorem was only valid in part of one case (2.35% of the possible orderings). The major cause of theorem invalidity seems to be that assumption A3 is not valid. Thus, if we are not indifferent between design decisions, it is not guaranteed that increased diversity of design decisions will decrease the probability of coincident failures. An example of this situation would be if we suspect that two design decisions have an interactive effect; in this situation it is unclear if maximally forced diversity will give increased reliability.

When we considered averages of the groups of failure probabilities for methodologies with equal developmental diversity, the orderings predicted by the theorem was valid in a majority of cases. However, our results show that if we are not indifferent between combinations of design decisions there can be no theorem stating that the smallest probability of coincident failures will be in the group with highest developmental diversity. There can be variations within the groups that invalidate orderings of them. Analytical and / or empirical investigations into the possibility of extending the theorem based on the amount of variation in the groups would be valuable.

It may not be the case that these results affect the actual strategies that should be used when developing multi-version systems. As was pointed out by Littlewood and Miller the assumptions they make in proving their theorem are plausible given that we do not *know* the effect of different design decisions. Our results do not invalidate this and in a state of ignorance about the effects of different decisions the assumption of indifference will be plausible and the theorems valid.

Our results have been obtained for one application. We need to investigate more applications to evaluate the generality of our results. It would be especially interesting to make comparative studies with previous experiments on multi-version software. In addition, this would make it possible to assess how software developed using genetic programming differs from software developed by humans. There are large dissimilarities between the two that could question the validity of our results. However, genetic programming should be a valuable tool in investigating the theoretical limits and theorems of diversity; versions are not treated differently depending on how they have been developed.

Genetic programming is a computational technique inspired by biological processes. It is the author's opinion that many new and interesting ideas for building and conducting research on fault-tolerant computer systems can be found by studying nature and biological systems as was provoked in [Avizienis95a].

Acknowledgement

The author wishes to acknowledge Jörgen Christmansson, Marcus Rimén, Martin Hiller, Jan Torin and Håkan Edler whose comments increased the quality of this paper.

References

Avizienis, A. "Building Dependable Systems: How to Keep Up With Complexity." Proceedings of the Fault-Tolerant Computing Symposium 25th Silver Jubilee (FTCS-25): 1995a. 4-14.

---. "The Methodology of N-Version Programming." Software Fault Tolerance. editor M. Lyu. Chichester, England: John Wiley & Sons, 1995b. 23-46.

Bäck, T., U. Hammel, and H-P. Schwefel. "Evolutionary Computation: Comments on the History and Current State." IEEE Transactions on Evolutionary Computation 1.1 (1997): 3-17.

Banzhaf, W., et al. Genetic Programming - an Introduction. San Fransisco, California: Morgan Kaufmann, 1998.

Bishop, P. "Software Fault Tolerance by Design Diversity." Software Fault Tolerance. (ed.) M. Lyu. Chichester, England: John Wiley & Sons, 1995. 211-30.

Christmansson, J. "An Exploration of Models for Software Faults and Errors.". Chalmers University of Technology, 1998.

Feldt, Robert. Using Genetic Programming to Systematically Force Software Diversity. Gothenburg, Sweden: Chalmers University of Technology, 1998a. 296L Thesis for the degree of Licentiate of Engineering.

---. "Generating Multiple Diverse Software Versions Using Genetic Programming - an Experimental Study." IEE Proceedings - Software (1998b).

Kelly, J. P. J., and A. Avizienis. "A Specification-Oriented Multi-Version Software Experiment." Proceedings of the 13th Fault-Tolerant Computing Symp. (FTCS-13): 1983. 120-26.

Knight, J. C., and N. Leveson. "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming." IEEE Transactions on Software Engineering 12.1 (1986): 96-109.
Den klassiska empiriska stuiden med 27 varianter av Missile Launch Interceptor utvecklade av student-team på 2 personer. Mycket bra artikel med väl diskuterade forskningsresultat.

Koza, J. Genetic Programming - on the Programming of Computers by Means of Natural Selection. Cambridge, Massachuchetts: MIT Press, 1992.

Littlewood, Bev, and Douglas R. Miller. "Conceptual Modeling of Coincident Failures in Multiversion Software." IEEE Transactions on Software Engineering 15.12 (1989): 1596-614.

Proebsting, T., et al. "Toba: Java for Applications - A Way Ahead of Time (WAT) Compiler". Technical report, University of Arizona, 1996.

Quereshi, A. GPSys 1.1 Homepage. Web Page. URL: http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html. 20 November 1998.

US Air Force. Military Specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction. 1986. MIL-A-38202C, Notice 1.

Part III.

6. Robert Feldt and Peter Nordin. *Using Factorial Experiments to Evaluate the Effect of Genetic Programming Parameters,* In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, Genetic Programming, Proceedings of EuroGP'2000 , volume 1802 of LNCS , pages 271-282, Edinburgh, 15-16 April 2000. Springer-Verlag.

7. Robert Feldt, Michael O'Neill, Conor Ryan, Peter Nordin, and William B. Langdon. *GP-Beagle: A Benchmarking Problem Repository for the Genetic Programming Community*, In Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, pp. 90-97, Las Vegas, Nevada, USA, July 2000.

Paper 6.

Robert Feldt and Peter Nordin. *Using factorial experiments to evaluate the effect of genetic programming parameters*, In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, Genetic Programming, Proceedings of EuroGP'2000 , volume 1802 of LNCS , pages 271-282, Edinburgh, 15-16 April 2000. Springer-Verlag.

Using Factorial Experiments to Evaluate the Effect of Genetic Programming Parameters

Robert Feldt and Peter Nordin

Dept. of Computer Engineering & Dept. of Physical Resource theory
Chalmers University of Technology
Gothenburg, Sweden
feldt@ce.chalmers.se, nordin@phy.chalmers.se

Abstract. Statistical techniques for designing and analysing experiments are used to evaluate the individual and combined effects of genetic programming parameters. Three binary classification problems are investigated in a total of seven experiments consisting of 1108 runs of a machine code genetic programming system. The parameters having the largest effect in these experiments are the population size and the number of generations. A large number of parameters have negligible effects. The experiments indicate that the investigated genetic programming system is robust to parameter variations, with the exception of a few important parameters.

1 Introduction

The Genetic Programming (GP) method might be the first instance of real *automatic programming* (Koza et al 1999). In an even more general sense, GP could be the first technique to tell the computer *what* to do without having to specify *how* to do it. However, in order for that to be true the user must be able to run the GP system using only a minimal set of natural parameters. In an ideal case there should be no parameters or only parameters that make immediate sense to the user's requirements such as *maximal search time* etc. This is far from true with present genetic programming systems. A modern GP system with additions such as Automatically Defined Functions (ADFs), Demes, and Dynamic Subset Selection have a very large number of parameters and settings creating a combinatorial explosion for the complete parameter space. This enormous parameter search space makes the search for an optimal or near optimal parameter setting difficult for the user.

What is even more severe is the theoretical implication of numerous parameters and settings. Each time we set a parameter we supply information to the search algorithm. If we set too many specific parameters, we might *"point out"* the solution with the parameters and we will not get *more out of the system than we put in*. We are supplying more information than the system is giving us back or in other words we are spending more effort and intelligence on the search for the right combination of parameters than the system does for the right solution.

The standard defence against this argument is that GP is very robust and accepts a wide range setting with little degradation in performance. This is usually only a hunch from GP researchers since there has been no large, systematic investigation of parameter effect using genetic programming. Such an investigation would have the additional benefit of enhancing experiments by providing close to optimal parameter settings. The only broad directions in the literature are experience-based, *rule-of-thumb-type* parameter recommendations (Koza 1992), (Banzhaf et al 1998).

In this works we describe the first series of experiments that address parameter influence in a broad and systematic way.

The questions that we are addressing are:

- Is GP robust toward different parameter settings or do settings have an effect on performance?
- If there is an effect on fitness, which parameters have the largest effect?
- Is the parameter effect dependent on single parameter settings or are combinations of parameters important?
- Can some parameters be ignored and can general guidelines be devised for the most important ones?

This paper address these questions using statistically sound experimental methods for parameter screening based on *fractional factorial designs* (Box et al 1978). Thes methods reduce the number of runs needed and increases the amount of knowledge that can be gained.

2 Method

To overcome the combinatorial explosion in the number of parameter combinations that need to be considered we use experimental design methods studied in mathematical statistics.

2.1 Experimental design

Statistical Design of Experiments (DoE) provides a framework to design and analyze comparative experiments, ie. experiments with the purpose to determine the quantitative effects of inputs on some output (Kleijnen 1998) (Box et al 1978). In this context the inputs are called *factors* and the output is called the *response*. The major advantage of using DoE designs is that experimentation becomes more efficient: both the effects of individual factors and their interaction can be investigated with limited experimental effort. This is achieved by changing more than one factor at a time.

A basic DoE experimental design is the factorial design where each factor has a discrete number of levels. An example of a two-level factor in GP is whether a certain function should be included in the function set or not. Continuos factors, such as for example the population size, can be used in factorial experiments if two discrete levels are chosen from their valid range. In a full two-level factorial design all combination of factor levels are included, resulting in 2^k different parameter settings, where k is the number of factors. Even for relatively small k:s the number of combinations needed is impractical. To overcome this fractional factorials are used. They utilize the fact that higher-order interactions between factors, ie that two or more factors have a combined effect different from each one of them in isolation, often have negligible effects. By letting lower-order effects, such as the main effects of the parameters and their two-factor interactions, be confounded with each other only a fraction of the full factorial design needs to be run.

The amount of confounding between effects in a design is determined by the design resolution. Design resolution refers to the amount of detail, separate identification of factor effects and interactions, that a design supports. For example, in a design of resolution five the main effects are confounded with four-factor interactions while two-factor interactions are confounded with three-factor interactions. The confounding pattern can be calculated from the design generators that define how the design is to be constructed. For more information on factorial designs see (Box et al 1978).

A typical strategy for experimentation using DoE is to make sequential use of designs with increasing resolution (Box et al 1978). In the first experiment a large number of factors are included since we do not yet know which of them may have large effects on the response. A heavily fractionalized design with low resolution is often used to *screen out* a majority of the factors. The remaining factors are studied in more detail in later experiments. Later experiments typically have higher resolution to permit separation of main and two-factor effects.

Tradtional DoE have been developed for physical and medical sciences and its development has been biased by the typical applications in these fields. For example, when an experiment is conducted in the real world it is often impractical to control more than 15 factors. (Kleijnen 1998) points out that a number of things are different when conducting experiments on a computer simulation: there are often more factors to be studied, we can practically control many more factors, and we do not need to randomize the run order of the experiments to get results that are robust to uncontrolled, and possibly even unknown, factors. These issues apply, in a similar way to genetic programming experiments.

3 Experiments

A total of 1108 GP runs were performed in seven different experiments on three different problems. In these runs, a total of about 2.5 billion individuals have been evaluated. Below we describe the problems, GP system, factors and response variable used in the experiments. We also describe the design of the experiments.

3.1 Problems

We believe in the importance of evaluating machine learning algorithms over *several* problems. In this work we have used three different binary classification problems. However, we plan to expand the number and types of evaluated problems significantly in future work, see section 6. The problems used are all standard machine learning problems: Ionosphere, Gaussian, and Pima Indians Diabetes Database.

Ionosphere Problem

This real-world radar echo classification problem has been donated by Vincent Sigillito of the Space Physics Group at John Hopkins University in the US. It is taken from the UCI Machine Learning repository (UCI ML Repository 1999). There are 200 instances in the training set and 151 instances in the validation set. The problem has thirty-four attributes and a binary-valued response indicating whether the echoes have detected any structure in the ionosphere.

Gaussian Problem

The gaussian classification problem is an artificial problem for heavily overlapping distributions with non-linear separability. The class 0 is represented by a multivariate normal distribution with zero mean and standard deviation equal to 1 in all dimensions, and the class 1 by a normal distribution with zero mean and standard deviation equal to 2 in all dimensions.

There are 1000 patterns, 500 in each class. We have used a variant of the standard eight-dimensional version, where there are 16 additional false (random) inputs in addition to the eight true inputs. Theoretical maximal classification for the pure 8-D problem is 91%. The problem is probably *not* easier with the false inputs added.

Pima Indians Diabetes Problem

This real-world medical classification problem has been donated by National Institute of Diabetes, Digestive and Kidney Diseases in the US. It is taken from the UCI Machine Learning repository (UCI ML repository 1999). The diagnostic, binary-valued response variable indicates whether the patient shows signs of diabetes according to World Health Organisation criteria (i.e., if the 2 hour post-load plasma glucose was at least 200 mg/dl at any survey examination or if found during routine medical care). The population lives near Phoenix, Arizona, USA. There are 576 instances in the training set and 192 in the validation set. The problem has eight attributes and a binary-valued response value.

3.2 Genetic programming system, its parameters and their values

For our experiments we used the Discipulus^{TM} system, a commercial implementation of machine code GP (RML 1999). Discipulus^{TM} is based on the AIM-GP approach, a very efficient method for genetic programming formerly knows as CGPS (Nordin 1997). The system uses a linear representation of individuals and a substring-exchanging crossover. In this survey we have used most of the parameters in Discipulus^{TM}. These parameters are used as factors in the experiments described below. Their factor identifier (A to Q) and their value at the low and high level used for the experiments are given in table 1. The levels of the continous parameters were chosen to represent qualitatively distinct levels based on our previous experience with the GP system in use. The parameters are briefly described below:

A. PopSize: The number of individuals in the population. At the low level the population size is 50 and at the high level it is 2000.
B. Generations: The system uses steady-state tournament selection so the generation parameter is the number of *generation equivalents* computed from *number of tournaments*. At the low level 50 generations are used and at the high level 250 are used.
C. MutationsFreq: Mutation frequency is the probability that an offspring will be subject to mutation. At the low level the mutation frequency is 10% and at the high level it is 90%.
D. CrossoverFreq: Crossover frequency is the probability that an offspring will be subject to crossover. At the low level the crossover frequency is 10% and at the high level it is 90%.
E. Demes: Determines whether the population is subdivided into subpopulations. In each experiment with demes we used 5 subpopulations, a crossover rate between demes of 3% and a migration rate of 3%. At the low level demes are not used and at the high level they are used.
F. ErrorMeasurement: The error measurement determines whether fitness is the sum of *absolute values* of errors (parameter at low level) *or* the sum of *squared* errors (parameter at high level).
G. DynamicSubsetSelection: Dynamic Subset Selection (DSS) is a method that only uses a subset of all the fitness cases in each evaluation. The selection of fitness cases was based on their individual difficulty (40%), the time since they were last used in fitness calculation (40%) and randomly (20%) (Gathercole 1994). At the low level DSS is not used and at the high level it is used.

H. MissClassificationPenalty: The classification problems are mapped to symbolic regression problems; each class is given a unique number. The fitness value is either the absolute distance or the squared distance between the actual and desired value. This parameter governs the amount of extra penalty that is added to the fitness for incorrect (miss) classifications. At the low level it is 0.0 and at the high level it is 0.25.
I. FunDiv: Determines whether division instructions are in (high level) or not in (low level) the function set.
J. FunCondit: Determines whether conditional instructions such as comparison, conditional loads and jumps are in (high level) or not in (low level) the function set.
K. FunTrig: Determines whether trigonometric functions are in (high level) or not in (low level) the function set.
L. FunMisc: Determines whether other non- trigonometric, non-arithmetic and non-conditional functions are in (high level) or not in (low level) the function set.
M. InitSize: The maximal initial size of the individuals, measured in number of instructions. At the low level it is 50 and at the high level it is 100.
N. MaxSize: The maximal allowed size of an individual, in number of instructions. At the low level it is 128 and at the high level it is 1024.
O. Constants: Determines the number of constants used in each individual. At the low level it is 1 and at the high level it is 10.
P. MutationDistr: When an instruction block is mutated it can be done on several different levels; the block level, instruction level or sub-instruction level (RML 1998). At the low level the distribution between them is 80%, 10%, and 10% respectively, and at the high level it is 10%, 10% and 80%.
Q. HomologousCrossover: Determines the percentage of crossovers that are performed as homologous crossover (Nordin et al 1999). At the low level it is 5% and at the high level it is 95%.

3.3 Response variable

We have used the maximum validation hit rate as the response variable for all problems and runs. This value was obtained by extracting the best individual on *training data* and running it on the validation set. It is reported as the percentage of correctly classified instances in the validation set.

Our choice of response variable defines the unit for the effects from the analysis of the experimental data. If, for example, an effect is calculated to be 5 this means that the average effect that can be expected when changing the factor from its low to its high level will be 5 percentage units (*not* 5%). Thus if the average response is 65% we would expect 70% on average with the factor at its high level.

3.4 Experimental designs

We have used three different experimental designs in a sequential fashion, each one based on the results from the previous one. The first two designs have been used on all three problems with the settings of factor levels described above. The third design uses different levels for the factors and has only been used on the gaussian problem. The purpose of the first experiment is to screen the large number of factors down to a more manageable set. Later experiments study the effects of the remaining factors in more detail.

To reduce the number of runs in the screening experiment we have employed a saturated design first described by Ehlich (Ehlich 1964) (Statlib 1999). This design allows the estimation of the main effects of seventeen factors in eighteen runs. The confounding patterns for this design is very complicated; main effects are confounded with several two- and higher-order effects.

The factors that had the largest effect in the screening experiments are varied in the second round of experiments. The rest of the factors are held constant at intermediate levels (N = 256, P = (40, 40, 20), Q = 50) or at the level indicated by the sign of its effect from the screening experiment (I, K, L, M, O at their low level and J at its high). We employ a fractional factorial experiment of resolution four. In this design the main effects are confounded with three-factor interactions which are assumed to be negligible. This allows the estimation of all main effects. Two-factor effects can be estimated but are confounded with each other. The actual design used is a 2^{8-4} fractional factorial with 16 runs (Box et al 1978). The generators for this design are D=ABC, E=BCH, F=ACH and G=ABH where a low level is represented by –1 and a high level by 1.

In order to estimate all two-factor interactions individually we need a design of resolution five. This is illustrated for the gaussian problem in the third experiment, which uses a 2^{5-1} fractional factorial with 16 runs (Box et al 1978). The generator for this design is H = ABCD.

In this third experiment we study the five factors that had the largest effect in experiment 2 on the gaussian problem. We alter the levels of these factors to gain more knowledge of their effect. The population size and number of generations had a significant effect and by increasing them (A to (500, 5000) for low and high level respectively and B to (100, 500)) we want to investigate if this effect holds also for higher levels. By increasing the low level of the mutation and crossover probabilities to 50% and keeping the high level at 95%, we can investigate if the level of 95% was extreme. By altering the values of both the low (to 0.05) and high levels (to 0.5) of the miss-classification penalty we can investigate if it is only important to have this penalty regardless of level or if the level in itself is important.

4 Results

Below we document the results for the seven experiments conducted. All values reported for the effect of factors and for confidence intervals is in the same unit as the response variable, see section 3.3. We have conducted a sensitivity analysis to evaluate how sensitive our results are to the number of replicates used for each parameters setting. This analysis is briefly described below.

4.1 Results of the screening experiment on IONOSPHERE

For each of the eighteen factor settings ten (10) replicates were run on the ionosphere problem. The standard error calculated from these 180 runs was 2.04 giving a 95% confidence interval of 4.63. The effects that were statistically significant at this confidence level are (in order of decreasing effect): A, B, G, C, H, D, E, F. The effect of A was about 45% larger than the effect of F.

4.2 Results of the screening experiment on Gaussian

For each of the eighteen factor settings eight (8) replicates were run on the gaussian problem. The standard error calculated from these 144 runs was 0.74 giving a 95% confidence interval of 1.94. The effects that were statistically significant at this confidence level are (in order of decreasing effect): A, B, C, H, E, D, G, F, J*, O*, P*, L*. However, note that the four factors marked with an asterisk had much smaller effect than the previous eight. For example the effect of F is more than four times higher than the effect of J.

4.3 Results of the screening experiment on PIMA-diabetes

For each of the eighteen factor settings, eight (8) replicates were run on the pima-diabetes problem. The standard error calculated from these 144 runs was 0.33 giving a 95% confidence interval of 0.96. The effects that were statistically significant at this confidence level are (in order of decreasing effect): A, C, G, B, F, E, H, D, L*, N*, P*, M*. However, note that the four factors marked with an asterisk had much smaller effect than the previous eight. For example the effect of D is more than eight times the effect of L.

4.4 Result of Second experiment on ionosphere

For each of the sixteen factor settings ten (10) replicates were run on the ionosphere problem. The standard error calculated from these 160 runs was 0.66 giving a 95% confidence interval of 1.50. The effects that are statistically significant at this confidence level are shown in table 4.

Table 4: Factors and their levels for experiment 2 on the ionosphere problem

CONTRAST	**EFFECT**	**95% CONF. INTERVALL**
A	4.19	+/- 1.50
B	2.23	+/- 1.50
AD + BC + EH+ FG	2.16	+/- 1.50
AG + BH + CE + DF	1.89	+/- 1.50
AH + BG + CF + DE	1.75	+/- 1.50

The population size (A) has the largest effect while the number of generations (B) and three different two-factor-interaction combinations have similar effects. The values reported in the table should be interpreted in the following way: if we change the level of factor A from its low to its high level we can expect an average increase in the validation hit rate by 4.19 units with a 95% confidence interval from 2.69 to 5.69 units. The same type of interpretation can be made for all effects reported in this paper.

The average validation hit rate was 92.1%, with a maximum of 98.7% and a minimum of 66.9%. The maximum average for a particular setting of the factors was 98.2% and the minimum 85.8%. These results can be compared with the maximum reported result from the UCI database describing the `ionosphere` problem: an average of 96% obtained by a backprop NN and 96.7% obtained with the IB3 algorithm (UCI ML repository 1999). However, we measure generalisation in a slightly different way: In the GP community it is common to look for the best generalizer in the population at reporting intervals in contrast to noting generalization capabilities among the best performing solution candidate on the training set. This difference applies for all experiments in this paper.

4.5 Result of second experiment on gaussian

For each of the sixteen factor settings ten (10) replicates were run on the gaussian problem. The standard error calculated from these 160 runs was 0.84 giving a 95% confidence interval of 1.94. The effects that are statistically significant at this confidence level are shown in table 5.

Table 5: Factors and their levels for experiment 2 on the gaussian problem

CONTRAST	**EFFECT**	**95% CONF. INTERVAL**
A	11.51	+/- 1.94
C	5.21	+/- 1.94
B	5.14	+/- 1.94
AD + BC + EH+ FG	3.39	+/- 1.94
D	2.82	+/- 1.94
AH + BG + CF + DE	2.76	+/- 1.94
AF + BE + CH + DG	2.35	+/- 1.94

The population size (A) clearly has the largest effect with the mutation probability (C) and number of generations (B) having about half the effect of A. Three different two-factor-interaction combinations and the crossover probability (D) have smaller effects.

The average validation hit rate was 63.8%, with a maximum of 88.9% and a minimum of 48.6%. The maximum average for a particular factor setting was 83.7% and the minimum 52.3%. This can be compared to the theoretical limit for this problem with a dimensionality of eight: 91%. However, note that this limit does not take the eight false inputs into account.

4.6 Result of second experiment on pima-diabetes

For each of the sixteen factor settings ten (10) replicates were run on the pima-diabetes problem. The standard error calculated from these 160 runs was 0.72 giving a 95% confidence interval of 1.63. The effects that are statistically significant at this confidence level are shown in table 6.

Table 6: Factors and their levels for experiment 2 on the pima-diabetes problem

CONTRAST	**EFFECT**	**95% CONF. INTERVAL**

A	5.72	+/- 1.63
B	2.12	+/- 1.63
G	2.02	+/- 1.63

The population size (A) have the largest effect while the number of generations (B) and the dynamic subset selection (G) have smaller effects.

The average validation hit rate was 65.46%, with a maximum of 77.6% and a minimum of 61.5%. The maximum average for a particular setting of the factors was 72.8% and the minimum 61.5%. These results can be compared with the maximum reported result from the UCI database describing the pima-diabetes problem: 76% using the ADAP learning algorithm (UCI ML repository 1999).

4.7 Result of third experiment on gaussian

For each of the sixteen factor settings ten (10) replicates were run on the gaussian problem. The standard error calculated from these 160 runs was 0.89 giving a 95% confidence interval of 2.02. The effects that are statistically significant at this confidence level are shown in table 7. Note that the levels used for the factors in this experiment are not the same as for the previous experiments. Hence, the actual effects are not comparable between experiments 2a and 3.

Table 7: Factors and their levels for experiment 3

CONTRAST	**EFFECT**	**95% CONF. INTERVAL**
B	9.39	+/- 2.02
A	7.53	+/- 2.02
H	3.71	+/- 2.02
AD	-2.30	+/- 2.02

The number of generations (B) and the population size (A) have the largest effects. The positive effect of the increased miss-classification is smaller but still significant. The same is true for the interaction between the population size (A) and the crossover probability (D). Note that since this design has resolution five this two-factor interaction is not confounded with any other two-factor interaction, as was the case in previous experiments. The somewhat surprising negative effect of this interaction means that some caution is called for when using large population sizes; increasing the crossover probability might have a detrimental effect.

The average validation hit rate was 76.4%, with a maximum of 88.9% and a minimum of 61.6%. The maximum average for a particular factor setting was 85.8% and the minimum 65.1%.

5. Discussion

We have presented our first results in a larger project attempting to investigate the effect of GP parameters. Even though these results stem from a limited number of problems and experimental designs we believe that some interesting conclusions can be drawn. However, we are far from settling the questions raised in the introduction, but we can identify interesting patterns.

In all three screening experiments the same eight parameters had the largest effects with the remaining nine factors having small or statistically insignificant[1] effects. Among these nine factors that were consistently screened *out*, we can find the factors determining the function set, the initial and maximal size of the individuals, the number of constants, the distribution of different mutation operators and the amount of crossovers that are homologous. It will be interesting to see if this result is valid for other problems and in other ranges of the continuos parameters.

Consistently, on all three problems, the population size and the number of generations are the most significant parameters. The population size comes out on top in the second experiments on all three problems with the number of generations a close second or third. However, note that the effect of the population size is numerically much larger than the other effects; this indicates that having a large population is important to get good results with GP. *Effort* has not been individually targeted in this survey, but it is interesting to note that choosing a large population size sometimes is more important than a large number of generations. In other words: a large population size running for very small number of generations could be better than a small population size running for a “normal“ number of generations. More investigation is needed on this.

It is interesting to note that the mutation and crossover probabilities have rather large effects on the gaussian problem. This somewhat contradicts the notion that mutation probability should be low. However, these factors did not have a statistically significant effect on the two real-world problems.

Dynamic subset selection can have a positive effect on the performance (Gathercole 1994). The fact that it, in addition, decreases the execution time of a run considerably would further speak for a more widespread use.

On both the gaussian and the ionosphere problem there are significant two-factor interactions. Since the design for experiment number two had a resolution of four we cannot separate the effect of different two-factor interactions. If we would like to do so we could add further runs to the existing designs or use a design of resolution five. Note that it can often be wise to use a design with lower resolution first and then add runs to separate between two-factor interactions of interest. In general, this will reduce the total number of runs needed. For example, to separate the four two-factor interactions having a combined effect of 3.39 on the gaussian problem in table 5 would require 3 extra experiments. Using a design of resolution five would require 64 runs; 48 more runs than for the design used herein.

The third experiment on the gaussian problem was included to show an example of a design of resolution five. Furthermore, the levels of the factors studied were changed to see their effect in other ranges of values. It is notable that the population size and number of generations are still the dominant factors. Note, however that the population size is no longer dominating; this could indicate that there is a limit to what can be gained from increasing the population size. The positive effect of the miss-classification penalty factor indicates that not only is it good to have such a penalty, but a relatively large penalty is better than a smaller one.

Our results partly support the notion that GP systems are robust to different parameter settings, as long as we choose the right values for the most important ones: population size and number of generations. On some problems the crossover and mutation probability can give good results with large levels. However, the negative interaction between population size and crossover probability in experiment 3 indicates that some caution must be taken.

[1] If an effect is not statistically significant it can not be separated from the natural variation in the response, ie noise.

The methodology used in this work can be used to optimize the results from a GP system. For example, note that the average response on the third experiment on gaussian is higher than the average response on the second experiment on the same problem. This is because the levels used in the third experiment were chosen based on the results from the second experiment. Thus, in addition to giving researchers a way to map out the effect of different parameters, DoE techniques may be used to optimize the response on a particular problem.

A drawback with the kind of DoE techniques used in this work is that they assume that higher-order interactions between factors are negligible. The empirical evidence for making this assumption are abundant; experimental investigations frequently show that the effect can be explained by a few important factors (Kleijnen 1998). However, we can never be fully sure and it will probably be wise to conduct full factorial experiments on some problems to validate this assumption. We have also noted that the responses in our experiments are often not normally distributed but grouped into clusters. In theory this makes statistical analysis of effects difficult since it violates the assumption of normally distributed responses. In practice, most statistical techniques have shown to be robust against deviations from normality (Box et al 1978).

It is worth noting that the GP system consistently performed very well compared to the previously reported best results on the test problems but with the caveat that generalization is measured differently. In future work we plan to change generalisation measurements to comply with the methods used in the UCI-database.

7. Conclusions

The Design of Experiments (DoE) techniques, from mathematical statistics, have been introduced as a solid methodology for evaluating the effect of genetic programming parameters. These techniques can also be used to increase the performance of a GP system, by guiding the user in choosing ‘good’ parameter combinations.

Our experiments show that, on three binary classification problems, the most important parameter was the population size followed by the number of generations. On one problem, large mutation and crossover probabilities had a positive effect. Furthermore, on all three problems, the same and large number of factors could be screened out because their effect could not be distinguished from noise. The result supports the notion that GP systems are robust against parameter settings but highlights the fact that there are a few parameters that are crucial.

This work reports the first results from a larger project attempting to investigate the effect of GP parameters. Much more work, involving more detailed designs as well as more varied test problems, is needed before we can address the questions as to the role and effect of GP parameters. We believe that such findings can be of great importance to the applicability of genetic programming in both industry and academia.

Acknowledgements

The authors wish to acknowledge Martin Hiller and Bill Langdon, whose comments increased the quality of this paper. Peter Nordin gratefully acknowledges support from the Swedish Research Council for Engineering Sciences.

References

Banzhaf, W., Nordin, P. Keller, R. E., and Francone, F. D. (1998). *Genetic Programming — An Introduction. On the automatic evolution of computer programs and its applications*. Morgan Kaufmann, Germany

Box, G. E., Hunter, W. G., Hunter, J. S. (1978). *Statistics for Experimenters – an Introduction to Design, Data Analysis and Model Building*. Wiley & Sons, New York, USA.

Ehlich, H. (1964). Determinantenabschatzungen fur binare Matrizen. *Math. Z.* 83, 123-132.

Gathercole C. and Ross P. (1994) Dynamic Training Subset Selection for Supervised Learning in Genetic Programming, Chris. In proceedings of the 3rd conference on Parallel Problem Solving from Nature (PPSN III), Springer-Verlag, Berlin, Germany.

Kleijnen, J. P. C. (1998). Experimental Design for Sensitivity Analysis, Optimization, and Validation of Simulation Models. In *Handbook of Simulation*, (ed.) Banks, Wiley & Sons, New York, USA.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA., USA.

Koza J.R, Andre D., Bennett F.H., Keane M. A. (1999) *Genetic Programming III: Darwinian Invention and Problem Solving*. Academic Press/Morgan Kaufmann.

Nordin, J.P. (1997), *Evolutionary Program Induction of Binary Machine Code and its Application*. Krehl Verlag, Muenster, Germany.

Nordin J. P., Banzhaf W., and Francone F. (1999) Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. To appear in *Advances in Genetic Programming III*, (eds.) Langdon, O'Reilly, Angeline, Spector, MIT-Press, USA

RML (1999) Register Machine Learning Incorporated. http://www.aimlearning.com

StatLib (1999), Online Statistical resources library at the Department of Statistics, Carneige Mellon University, USA, http://lib.stat.cmu.edu/.

UCI ML repository (1999). Files for the Pima-diabetes and Ionosphere problems from the Machine Learning repository at University of California, Irvine describing the Ionosphere problem. http:// www.ics.uci.edu/~mlearn.

Paper 7.

Robert Feldt, Michael O'Neill, Conor Ryan, Peter Nordin, and William B. Langdon. *GP-Beagle: A Benchmarking Problem Repository for the Genetic Programming Community*, In Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, pp. 90-97, Las Vegas, Nevada, USA, July 2000.

GP-Beagle: A Benchmarking Problem Repository for the Genetic Programming Community

Robert Feldt
feldt@ce.chalmers.se
Computer Engineering
Chalmers University
SE-412 96, SWEDEN

Michael O'Neill, Conor Ryan
Michael.ONeill@ul.ie
Computer Science
University of Limerick
IRELAND

Peter Nordin
Complex Systems
Chalmers University
SE-412 96, SWEDEN

William B. Langdon
CWI, Kruislaan 413
1098 SJ Amsterdam
NETHERLANDS

Abstract

Experimental studies in genetic programming often only use a few, artifical problems. The results thus obtained may not be typical and may not reflect performance on problems met in the real world. To change this we propose the use of common suites of benchmark problems and introduce a benchmarking problem repository called GP-Beagle. The basic entities in the repository are problems, problem instances, problem suites and usage information. We give examples of problems and suites that can be found in the repository and identify its WWW site location.

1 INTRODUCTION

A large fraction of genetic programming (GP) research is empirical. New ideas are implemented and tested in experiments on a number of problems. Sometimes the performance of the new idea is compared to a baseline GP system. Even though this can show the relative merit of the new idea it does not easily extend to comparing the merits of different GP extensions. Furthermore, the problems used are often artificial so the results may not be representative of the performance on real-world problems. If real-world data are used the number of different problems is often limited. This can lead to happenstance results that is not typical of the performance on the majority of problems.

In this paper we introduce GP-Beagle, an infrastructure for establishing, maintaining and promoting a publically available repository of benchmarking problems for empirical investigation and performance evaluation of genetic programming systems. It includes both individual problems and benchmarking suites of problems. It defines a nomenclature and structure for different entities related to benchmarking, specifies attributes needed to describe each problem and suite and lists the publications in which they have been used. Inspired by the recent successes of the open source movement the repository is available under a GPL-like usage agreement where the use of the problems is free but published results must be reported back to the repository. This ensures that the repository can give an up-to-date view of the use of the problems and the knowledge gained. The GP-Beagle effort is supported by a WWW site, currently under development, at http://www.gp-beagle.org.

We believe that GP-Beagle will enable the GP community to make faster progress since it will promote the use of sound experimental methods, provide a common ground for comparisons, enable faster elimination of ideas that are not fruitful and evoke discussions about the problems we use in our research and their respective merits. However, this effort will be successfull only if we all, as a research community, make use of and extend the repository. We hope to convince you that taking part in this effort will be beneficial both to you and to the community as a whole.

There are a number of existing problem databases in areas related to GP and much can be gained by using them [1]. However, we think a new repository is needed for GP since GP can attack other types of problems and existing databases are mainly pools of problems and do not give an up-to-date view of the use of the problems. Furthermore, establishing a community-specific repository have the potential of raising the awareness and use of experimental studies far more than the act of pointing to existing databases.

Section 2 elaborates on experimental research and the pros and cons of using benchmarks. In section 3,

[1] One example is the UCI Machine Learning repository with about 100 classification and regression data sets [1].

the components and structure of GP-Beagle are introduced and in section 4 we detail the attributes used to describe the entities in the repository. Examples of problems and suites in the repository are described in section 5 and section 6 concludes the paper.

2 EXPERIMENTAL RESEARCH AND BENCHMARKS

Experimental research is an important part of the scientific method and it's importance in computer science has been recently stressed [7] [2]. Even though experiments can never prove a theory they allow us to test theoretic predictions in reality and to explore areas where theory can not (yet) reach. The main benefits of conducting experiments is that they help build a reliable knowledge base of adequate theories and methods, they give observations that can lead to unexpected insights and that they accelerate progress since they help to quickly eliminate unfruitful approaches and weed out erroneous claims. Experiments thus guides engineering practice and theory development in promising directions.

We know of no studies of the current level of experimental practice in GP research. Studies on the neural network community and computer science in general have revealed that the amount of experimental evaluation is low [5] [8]. A study of 190 neural networks articles published in 1993 and 1994 showed that only 8% presented results for more than one real-world problem, 29% did not employ even a single realistic learning problem and one third did not present any quantitative comparison with a previously known algorithm [5]. Even though some of the efforts in the ANN community to raise the level of experimental assessment have probably "spilled over" to the GP community we suspect that the situation in the GP community is not much different. A collective strive for better assessment practices thus seem called for.

Benchmarks are an effective and affordable way of conducting experiments and have been successfully used in many areas [3]. A benchmark is a collection of problems with well-defined performance measurements and a prescribed method how to evaluate performance. If they are chosen in a good way they allow repeatable and objective comparisons. The essential requirements on a benchmark are (based on [6]):

- *Volume*: the benchmark should include several and diverse problems,
- *Validity*: common errors that invalidate the results should be avoided,
- *Reproducibility*: problems and experiments should be documented well enough to be reproducible,
- *Comparability*: results should be comparable with the results in other studies.

[2]This section draws heavily on the two papers [7] and [6].

[3]A notable example are the "spec" problem suites used for benchmarking computer performance.

Conducting experiments with only a few problems makes it difficult to characterize a new algorithm or extension. If only problems of the same type are used the results may not show the typical performance. By including several problems of different types we can get a fuller picture of how the algorithm performs in general.

Methodological errors that threaten validity include the choice of a problem suited to the investigated algorithm, reporting the result on a data set that was used for training or using the test data set for tuning parameters. These errors can be avoided by using a well-defined evaluation procedure that separates between training, validation and testing.

If a paper does not describe the exact setup of an experiment the result can not be reproduced.

If results of different studies can not be compared it is difficult to choose between algorithms and ideas proposed in different studies. This slows down progress.

In addition to these four requirements, practitioners have the requirement of *representability*: benchmark problems should resemble the problems met in the real world. A risk with using artificial problems is that they have a limited information content [4] so that there is no room to discover and exploit different layers of complexity. For example, there might be no use in having a meta-learning ability, such as assembling information on search directions while searching, on the multiplexer problem. Comparing machine learning algorithms on simple problems with only one, central "idea" to "get" might evaluate problem solving ability in an unfair way.

The use of benchmarks has some disadvantages. One risk is that algorithms are specifically tailored to perform well on the benchmark problems. Another risk is that benchmarks focus too much on a single, numerical performance measure. This can hinder progress

[4]In an information theoretic sense. For example, a scalable artifical problem, such as Gaussian described in table 1, has the same information content (kolmogorov complexity) regardless of how the parameters are varied.

because researchers optimize a local optima instead of exploring new and innovative avenues of research. Another problem is that it is not clear how fair comparisons should be carried out. For example, it might be unfair to compare the accuracy of two GP algorithms without taking their execution time or the time needed to set them up or the time needed to tune their parameters into account. Finally, benchmarks have to evolve with the needs of the community and application areas; if they are static they will fail to reflect new knowledge and will thus become irrelevant.

At the current level of maturity of experimental practice in the GP community we think that the advantage of establishing and using common problems and benchmarks outweighs these potential drawbacks. By constantly remind ourselves of these pitfalls their negative effects can be avoided. Furthermore we have designed GP-Beagle to explicitly try to address them.

3 THE GP-BEAGLE PROBLEM REPOSITORY

GP-Beagle is designed to be a one-stop place for all information on GP problems and benchmarking suites of problems. The basic philosophy is that GP-Beagle should define an open framework that can be easily extended were suites of problems can evolve as knowledge is gained on them and the algorithms they are used to evaluate. Thus, GP-Beagle does not simply supply a number of problems, it also collects and presents information on their use. To guarantee that the usage information is up-to-date the problems are supplied under a usage agreement. The agreement states that the problems can be freely used but that information on their use should be reported back to the repository. It also encourages researchers to submit new problems to the repository. Any problems are accepted as long as they meet basic criteria (has been used in published work and several instances of the same type of problem are not already in the repository).

Since evolutionary algorithms are general search algorithms that can be applied to a large number of areas it would not be wise to specify one benchmark suite to be used in all research. GP-Beagle does not prespecify a number of suites but starts by recording the collections of problems that are actually used. Thus, the suites are de facto collections of problems. Over time it is anticipated that special suites will evolve for different sub-areas of GP research such as for example classification, regression or artificial problems. It is also anticipated that when a mass of problems and usage data have been assembled suites can be constructed in a rigorous way, using recent ideas on how to quantify the features of benchmarking suites [3].

GP-Beagle is implemented on a WWW server as a set of Perl-scripts accessing a MySQL database. The database consists of records for each of the basic entities: problems, problem instances, de facto and benchmarking suites and usage information. This implementation minimizes [5] the amount of human resources needed to maintain the repository. Statistics on the use of problems in the repository can be automatically collected. The structure of the repository and the GP-Beagle usage agreement is further described below. Section 4 gives a detailed view of the entities in the repository.

3.1 STRUCTURE OF THE REPOSITORY

The basic entity of the repository is a *problem*. A problem is either a data set, a data generator or a simulator. Both of the latter are programs that generate data to be used in fitness evaluation. The difference is that a data generator is used off-line, ie. by generating a data set prior to starting the GP run, while a simulator is used on-line in a dynamic evaluation of a GP individual. A problem can be either artificial or real-world.

Specifying which problem has been used in an experiment is not enough to allow full reproducibility and comparability of results [6] [2]. For instance it is not enough to specify which data set has been used; one must describe how the data set have been divided into training, validation and testing sets. For a simulator or data generator we need to know which parameters have been used, how many fitness cases have been generated and so forth. To encompass this level of detail GP-Beagle introduces the concept of a *problem instance*. This is a fully specified description of the problem and how it has been used [6]. Thus, each problem in the repository can have multiple instances but each instance can only stem from one problem. A problem defines a family of possible instances.

A collection of problem instances that have been used together in an experimental study is called a *problem suite*. A *homogeneous* suite consists of problem instances from the same problem, while a *heterogeneous* suite have instances from several problems.

[5] Human assistance will be needed to review that new submissions to the repository are complete, to create new benchmarks etc.

[6] An instance may contain multiple samples from the same problem data set.

A special kind of problem suites are the *benchmarking suites*. These suites are not de facto suites that have already been used in actual research. Instead they are explicitly added to the repository to promote new kinds of experiments or to define suites consisting of diverse problem instances.

In addition to these four basic entities the GP-Beagle repository contains usage information. The usage information details in which studies each problem instance and suite have been used and the results and knowledge obtained. This information can be easily accessed when browsing the repository. GP-Beagle also collects statistics on the use of problems so that hot-lists can be presented. This way a researcher can easily find the problems that are often used and that would thus give good opportunities for comparative analysis.

3.2 THE GP-BEAGLE USAGE AGREEMENT

The problems in the GP-Beagle repository are available free for any academic or commercial use as long as any published information generated by this use is reported back to the repository. Specifically the information that should be reported includes (general and suite-specific information):

1. Reference to paper where the experiment is described, and
2. The set of problem instances used, and
3. The goal of the experiment and a rationale for choosing this specific set of problems (if any), and
4. Any knowledge gained on the set of problems such as their suitability for achieving the goal.

and for each problem instance used (instance-specific information):

1. The result obtained on the performance measure defined for the problem instance, and optionally
2. The execution time.

A new problem instance can be generated or an existing instance can be altered as long as the new instance is supplied back to the repository together with the following information:

1. The reason for creating the new instance, and
2. A description of why the previously existing instances was not adequate.

4 ATTRIBUTES OF ENTITIES IN GP-BEAGLE

The following attributes are kept in a record on a problem in the repository:

- Name: A unique name for the problem. Once assigned the problem will always have this name and can thus be uniquely referred to in papers and discussions.
- Description: A textual description of the problem. Should ideally give some basic knowledge on the domain, describe the parameters in a DataGenerator or Simulator, if attribute values are missing in a DataSet etc.
- Version: A version number to reflect updates to the problem.
- Type: DataSet / DataGenerator / Simulator
- Sub-type: Regression / Binary Classification / 5-Classification etc.
- Origin: Artificial/Real-world. Artificial problems are further characterized as whether their difficulty can be varied.
- Source: Who submitted the problem.
- Status: Suggested / Reviewed. Indicates if the problem have been reviewed and thus “officially” entered the repository.
- Number and type of attributes: Total number of attributes, number of continous and discrete attributes.
- Number of instances: Number of instances in a DataSet.
- File: A gzip:ped tar file with all the files in the problem.

The unique attributes of a problem instance record:

- From problem: The problem that the instance is derived from.
- Description: Describes how the instance was derived from the “parent” problem, what components it consists of, why previously existing instances of this problem was not adequate etc.
- Reason created: Reason for creating the instance.

- Performance measure: Describes the "fitness" value used to evaluate algorithms on the instance.
- Number of instances: Number of instances that can be used in evolving a solution (ie. these instances can be divided in validation and training sets).
- Number of test instances: Instances in test set that cannot be used in any way to evolve a solution.
- GP result: Give an example of a good result obtained with a GP technique.
- GP paper: Pointer to a problem instance usage info record describing the paper in which the good result was obtained.
- Other result: Give an example of a good result obtained with a non-GP technique.
- Other paper: Briefly describe the technique used and give reference to paper where result can be found.
- Simple result: Give result achieved with a simple technique (for example plurality rule in classification task or a technique based on linear separation in regression).

The record for a suite contains the following unique attributes:

- Type: DeFacto / Benchmark.
- Problem instances: Instances in the suite.
- Sub-type: Heterogeneous / Homogeneous
- Performance measure: Performance measure for suite.

In addition to the above, basic entities the repository contains two types of usage information records: instance usage info and suite usage info. The unique attributes of the instance usage info are (the suite usage info record is similar):

- Paper: Paper where experiment with instance is described. Pointer to GP bibliography.
- Technique used: Algorithm or technique used.
- Performance obtained: Performance obtained.
- Time: Execution time to evolve a solution with the performance above.

We have contemplated using a standardized way to report the execution time but we do not think that one "right" way to do it is yet available. One possible way would be to report the actual execution time normalized with the spec benchmark result for the CPU used as in [4]. However, a number of objections can be raised to this scheme so we have chosen not to specify one way on how to measure the time needed.

5 EXAMPLES

Below we give examples of some entries in the repository. One is a problem, one is a problem instance, one is a de facto suite and one is a proposed benchmark. The descriptions are brief and primarily intended to give you a picture of the kind of information that can be found in the repository. More details can be found at the GP-Beagle web site.

5.1 PROBLEM: Gaussian(n,μ_1,σ_1,μ_2,σ_2,f,f_l,f_h)

The Gaussian problem is a DataGenerator problem. It's record in the GP-Beagle database is shown in table 1. The data file for the problem, gaussian.tar.gz, contains the following files:

- readme.txt - A description of the files included in this tar file, and
- gaussian.description - The data from the record shown in table 1, and
- gaussian.c - The DataGenerator implemented in ANSI-C, and
- usage.info - Description of how to compile and use the DataGenerator, and
- data.info - Description of the data file generated when the generator is run.

The files are typical of what should be included for a DataGenerator problem; they will differ for other types of problems.

5.2 PROBLEM INSTANCE: KddCup99-disctoint-1%

The KddCup99-disctoint-1% is a problem instance sampled from the KDD Cup 1999 data (a real-world 5-class classification DataSet problem). The problem instance record is shown in table 2. Note that the reference to the GP paper is given as the bibtex key in the

Table 1: Record for the Gaussian problem

Name: Gaussian(n,μ_1,σ_1,μ_2,σ_2,f,f_l,f_h)		
Type: DataGenerator	**SubType:** Binary Classification	**Version:** 1.0, 2000-05-22
VariableDifficulty: Yes	**Status:** Suggested	**Origin:** Artificial
# **Instances:** Varying	**Attributes:** Varying # of numerical	**File:** gaussian.tar.gz
Source: Carla Fredrica Gauss, cfgauss@math.rocks.org		
Description: Discriminate instances generated from either of two multivariate (n attributes) gaussian distributions with mean and stddev (μ_1, σ_1) and (μ_2, σ_2), respectively. The 'f' parameter governs how many false input attributes, uniformly sampled on [f_l,f_h], should be added to each instance. The difficulty of the problem (dimensionality, Bayes optimal classification rate and number of false attributes) can be varied by varying the parameters of the problem. The Bayes optimal classification rate (ultimate uncertainty in problem which no ML algorithm can do better than) can be calculated for parameter choices with f equal to 0. Generalization of a problem from Elena project.		

Table 2: Record for the KddCup99-disctoint-1% problem instance

Name: KddCup99-disctoint-1%		
FromProblem: KddCup99	**Status:** Suggested	**Version:** 1.0, 2000-05-18
# **Instances:** 48984	# **TestInstances:** 311029	**File:** kddcup99-disctoint-1.tar.gz
PerformanceMeasure: Average cost per test instance according to specified cost matrix		
Source: Catherine Darwin, cdarwin@evolution-rules.com		
Description: The data used in the KDD Cup 1999 competition had more than 4 million training instances and 311,029 testing instances. This problem instance contains a 1% sample of the training instances but all of the testing instances. The "disctoint" refers to the mapping from discrete input attributes to numerical integers. The task is relatively difficult since the class distribution in the test set is different from the class distribution in the training set.		
ReasonCreated: We wanted to test if a GP system can get competitive results even with the simplest possible mapping (mapping the values of an unordered discrete attribute to integers imposes an order that does not exist in the original data). We took a 1% sample because we wanted to get a more manageable data set that would give shorter execution times. The test set was kept intact since we wanted to be able to compare to the results of the algorithms in the KDD Cup.		
GPResult: 0.1985	**GPPaper:** gpbiblio:darwin:ieeetroec:2001	
OtherResult: 0.2331 with bagged and boosted decision trees (winner KDD Cup'99)		
OtherPaper: Elkan, C.: Results of the KDD'99 Classifier Learning Contest, http://www-cse.ucsd.edu/users/elkan/clresults.html, May 2000		
SimpleResult: 0.5220 with plurality rule and 0.2523 with a 1-nearest neighbor classifier.		

GP bibliography. We are planning to implement connections between GP-Beagle and the GP bibliography so that papers can easily be located and searched.

5.3 DE FACTO SUITE: Proben1-medical

A recent paper by Brameier and Banzhaf used six problems from the Proben1 benchmark suite to compare GP performance to that of neural nets [2]. Each problem used had three different samples of the same data set. We have put these three samples in the same instances and thus this de facto suite contains 6 different problem instances. Its record is shown in table 3 [7].

5.4 BENCHMARK SUITE: Classification-diverse18

To give an example of a benchmark suite we have created one by adding two large classification problems to the suite of 16 classification problems used in [4]. Note that the KddCup99-disctoint-1% problem instance described in table 2 is one of them. The record is shown in table 4. Also note that some of the problem instances used are from the same problems used in the Proben1-medical suite above. Since a different sampling and evaluation procedure (10-fold cross-validation vs. 3-fold cross-validation) was used in this suite the instances are distinct even though they stem from the same problems.

6 CONCLUSIONS

We have described GP-Beagle, an infrastructure for establishing, maintaing and promoting a publically available repository of benchmarking problems for empirical studies of genetic programming systems. By using benchmarks the genetic programming community can make faster progress since results from different studies can be more easily compared. Furthermore, benchmarks chosen in a good way promotes sound empirical studies since they include a broad and diverse set of problems and prescribe the evaluation procedure and performance measurements to be used.

To address some of the pitfalls of using benchmarks GP-Beagle is an open framework where benchmarks and problems can evolve; we have not pre-specified some benchmarks that must be used. We anticipate that over time the GP community, in a collective effort, can assemble benchmarks for different sub-areas of GP research in the framework supplied by GP-Beagle.

[7]In the paper, Brameier and Banzhaf does not report an aggregated performance measure as is indicated in table 3.

The basic entities in GP-Beagle are problems, problem instances and problem suites. Problem instances are concrete instances of a problem with a full description of how they should be used. They allow for full reproducibility of results. The repository also contains information on the use of the problems and suites. All problems are freely available as long as published results and problem extensions are reported back to the repository.

GP-Beagle is implemented as a set of records in a MySQL database. Perl scripts are used to extract information and update the data base. The interface to the repository is via a web site at http://www.gp-beagle.org. In order for this effort to really take off we encourage you to visit the site, start using the repository and submitting your problems and results.

References

[1] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[2] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, in press, 2000.

[3] Jozo J. Dujmovic. Universal benchmark suites. In *Proc. 7th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 197–205, 1999.

[4] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, Forthcoming, 2000.

[5] L. Prechelt. A quantitative study of experimental evaluations of neural network learning algorithms: Current research practice. *Neural Networks*, 9(3):457–462, 1996.

[6] Lutz Prechelt. Some notes on neural learning algorithm benchmarking. *Neurocomputing*, 9(3):343–347, 1995.

[7] W. Tichy. Should Computer Scientist Experiment More? *IEEE Computer*, 31(5):32–40, 1998.

[8] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *The Journal of Systems and Software*, 28(1):9–??, January 1995.

Table 3: Record for the Proben1-medical de facto suite

Name: Proben1-medical		
Type: DeFacto	**Status:** Suggested	**Version:** 1.0, 2000-05-24
# Instances: 6	**Id number:** 1	**File:** proben1-medical.tar.gz
Instances: Cancer-proben1, Diabetes-proben1, Gene-proben1, Heart-proben1, Horse-proben1, Thyroid-proben1		
PerformanceMeasure: Average classification error on the 3*6=18 test sets		
Source: Markus Brameier and Wolfgang Banzhaf (originally from the Proben1 benchmark), banzhaf@not.valid-email.de		
Description: A subset of six medical classification problems was extracted from the Proben1 neural network benchmark. Each instance consists of three different samples from one and the same problem.		

Table 4: Record for the Classification-diverse18 benchmark suite

Name: Classification-diverse18		
Type: Benchmark	**Status:** Suggested	**Version:** 1.0, 2000-05-24
# Instances: 18	**Id number:** 2	**File:** classification-diverse18.tar.gz
Instances: Cancer-lim, Cmc-lim, Dna-lim, Heart-lim, Boston-housing-lim, Led-lim, Liver-lim, Pima-indians-lim, Satimage-lim, Image-segmentation-lim, Smoking-lim, Thyroid-lim, Vehicle-lim, Voting-lim, Waveform-lim, Ta-evaluation-lim, KddCup99-disctoint-1%, KddCup98-disctoint-5%		
PerformanceMeasure: Average classification error rate		
Source: Robert Feldt, feldt@ce.chalmers.se		
Description: A broad and diverse suite of classification problems. Includes five binary, seven ternary, one 4-class, two 5-class, one 6-class, one 7-class and one 10-class classification problems. On “small” problems (less than 1000 instances in test set) 10-fold cross-validation is used to estimate the classification error rate. Sixteen of the problems have been used on 33 different ML techniques in a study by Tien-Sien Lim et al. This allows for comparisons to a large number of machine learning algorithms. Two additional data sets from the 1998 and 1999 KDD Cup competitions were added to the benchmark because many of the problems used in the Lim et al study was “small”.		