Why Source Code Analysis and Manipulation Will Always Be Important

Mark Harman
University College London, CREST Centre, Department of Computer Science, Malet Place, London, WC1E 6BT, UK.

***Abstract*—This paper[1] makes a case for Source Code Analysis and Manipulation. The paper argues that it will not only remain important, but that its importance will continue to grow. This argument is partly based on the 'law' of tendency to executability, which the paper introduces. The paper also makes a case for Source Code Analysis purely for the sake of analysis. Analysis for its own sake may not be merely indulgent introspection. The paper argues that it may ultimately prove to be hugely important as source code gradually gathers together all aspects of human socioeconomic and governmental processes and systems.**

I. Introduction

Source Code Analysis and Manipulation is important. It will always be important and its importance will increase. The argument for the longevity of Source Code Analysis and Manipulation is based on the observation that all notations tend to become executable. This means that every new notation, whether lower level (and therefore already source code) or higher level (and therefore seeking, initially to escape the code level) will ultimately become source code. This is the 'law of tendency towards execution'.

Because source code is, in the words of the SCAM conference's own call for papers 'the only precise description of the behaviour of the system' its analysis and manipulation will always be important and will always throw up new challenges. Perhaps this much is uncontroversial.

However, a little more controversially, I argue that Source Code Analysis and Manipulation will become ever more important, because source code will increasingly find its way into the socioeconomic, legal and governmental fabric of our interconnected world. This is not necessarily a universally *good thing*. As software drives the trend for rapid change, typified by ever shorter cycle time between conception, development and adoption we face new dangers.

There is a consensus that this trend has many positive effects. Nonetheless, it draws engineers and users into a spiral of increasing complexity and scale. Increasing automation creates a potentially pernicious trap, into which we are at risk of falling as humans and systems are become ever more fixed into a global Carbon-Software Skin.

This skin of interconnected systems, processes and people may soon come to resemble more the inflexible mesh of entrapment if we do not develop ways to make the systems more flexible, responsive and adaptive to the pace of change that they, themselves engender. I will argue that source code will come to be seen to be one of the most fundamental and pivotal materials with which humankind has ever worked. As a result, its analysis and manipulation are paramount concerns.

In the technical part of the paper (Section VII), I will consider the definition of source code, of analysis and of manipulation, adopted by the community over the ten years since the founding of this conference, bringing a sharper technical focus to the claims, in this introduction, that source code lies at the heart of socioeconomic interaction, government and even our very biology. The analysis and the manipulation of source code are interwoven; both can and should be used to yield insight into the properties, characteristics and behaviour of the code.

Section II, reviews the definition of source code and explains why all descriptions will tend towards executability, guaranteeing the study of source code analysis and manipulation and seat at the table of discourse in perpetuity. Section III presents a very brief history of computation, source code and source code analysis from Stonehenge to Ada Lovelace and the very first ever paper on Source Code Analysis. Section IV develops this historical perspective to look towards the future development of source code and, with it, the future of Source Code Analysis and Manipulation. The section argues that Source Code Analysis will be essential and may just rescue humankind from the jaws of a ghastly, yet banal source–code–controlled dystopia. Section V shifts back to a more down-to-earth and technical treatment of two of the conference's widely studied source code manipulation techniques: slicing and transformation. It shows that both are closely–related special cases of manipulation, formalizing this observation within the projection framework. Section VI argues that source code manipulation can be used for source code analysis. It also makes the case for source code analysis for its own sake, illustrating in Section VII with some personal experience of Source Code Analysis and Manipulation as a 'voyage of discovery'.

II. What is source code and why is it guaranteed to be always important?

Definition 1 gives the SCAM call for papers' definition of source code. The definition focuses on the way in which source code fully describes the execution of the system. This was intended to be a general definition. It would appear to have served the community well because there has been no

[1]The paper is a written to accompany the author's keynote presentation at the 10^{th} International Working Conference on Source Code Analysis and Manipulation (SCAM 2010). As such, the body of the paper adopts a polemic, first person singular prose style and takes the form of an extended position paper.

change to the definition over the decade since the conference's inception. Also, there have been no protracted discussions of the form 'what is source code?' during any of the previous nine instances of the conference over the years.

Definition 1 (Source Code): For the purpose of clarity 'source code' is taken to mean any fully executable description of a software system. It is therefore so construed as to include machine code, very high level languages and executable graphical representations of systems.

One obvious reason why source code analysis and manipulation will always be important derives from the prevalence of source code. As we try to escape its clutches we are drawn back into its embrace. For instance, early attempts to move away from source code to higher level languages led to a natural inclination towards specification languages such as Z [75] and VDM [53]. However, this was shortly followed by a retrenchment to what were termed 'executable specifications'. According to the SCAM definition of source code, since an 'executable specification' is executable, it is, *ipso facto*, source code.

More recently there has been a migration from lower level code towards models. Model Driven Development has a lot to offer. The use of UML, for instance, is undoubtedly an engineering advance. However, it is inevitable that we shall seek to compile as much as possible of the UML design of a system into executable code. As soon as we can compile it to an executable code it simply becomes the new 'high level language'. This is certainly progress, but we must not lose sight of the fact that it means that UML will have become 'source code' too. Already, there is work on analysis and manipulation of aspects of the UML using, for example, slicing [1], [56]. This should not be seen as a descent to some low level compromise by adherents of the UML. It is a natural evolution of the notation. I believe it might even be called a law:

"Descriptions tend towards executability".

That is, no matter how we try to abstract away from what we may uncharitably describe as 'low level' code to 'higher' levels, we shall soon find that we are writing some form of source code. This is a natural progressive characteristic, that can be thought of as a law in the same sense as Lehman's laws of software evolution [61]. We are compelled by the human desire to master and control our environment. This entails making descriptions executable: If our higher, more abstract, notations are any use to us, then why would we not want to automate the process of transforming them into some lower level executable code? Why would we want to undertake that effort ourselves, and thereby be forced to descend back down to the lower levels? As soon as we have fully automated the process of transformation from our higher levels of abstraction to executable code, we have invented a new form of source code. Perhaps it will be a better source code, but it will be source code, nevertheless.

III. The Beginnings of Source Code and its Analysis and Manipulation

There is much debate as to the origins of computation and claims and counter claims as to originality permeate the literature on the history of computing [63]. I shall adopt a parochial view and follow the story primarily from an English perspective, starting with what might be the world's first computer. Though now obsolete, the remains of this computer can still be seen on Salisbury Plain in Wiltshire in England. It is known as 'Stonehenge' [3]. This vast stone age computer was one of the world's first timepieces, arguably a forerunner of today's GPS–enabled, time–and–position aware applets. Though its origins and precise purpose remain partly mysterious, it is clear that it allowed primitive date computations to be made, including predicting the summer and winter solstices.

The first incarnation, built in 3,100 BC was constructed from earth, augmented in 3,000BC by a structure in wood. This proved to be a rather poor material from which to build the computers of the day and so, around 2,600BC, a re-implementation was constructed in stone. This much more costly undertaking was accompanied by what might be termed one of the world's first known 'bug fixes'. It was found that the position of the northeastern entrance was not quite accurate and this was corrected by the stone re-implementation. The bug fix consisted of widening this northeastern entrance [3], [69].

Of course, this early computer was built without the one key ingredient that makes computation so astonishingly powerful and all-embracing: *source code*. This could be contrasted with the other early computational device, dating to 2,700BC, the abacus [51]. With the abacus, one can perform many *different* computations, based on a prescription for how to move the beads of the device. If these prescriptions for how to move the beads are written down then they become a form of source code, albeit one that denotes a sub-Turing language of computation. Without source code, the computation performed cannot change and so the implementation remains, perhaps reliable, but inherently inflexible.

The most significant early landmark in the history of source code was reached with the work of Charles Babbage and Ada Augusta Lovelace in their work on the Analytical Engine[2]. This was an entirely different machine from the earlier attempt by Babbage, with the Difference Engine, to replace 'human computers'; clerks who were trained in arithmetic and who labouriously performed mundane computations, entirely without the aid of automation. The analytical engine was, to all intents and purposes, conceived entirely like a present-day computer; its punched–card source code had versions of assignments, conditionals and loops. The analytical engine that was to execute this source code was to be built of brass, rather than earth, wood or stone. Crucially it was to be *automated*, replacing potentially fault-prone human computers with automated computation. This automation was to be powered by steam rather than electricity, which was only available from primitive zinc–copper batteries at the time.

[2]As other authors have found, it is hard to provide a reliable and precise citation for this crucially important piece of work. Lovelace's comments appeared in her English translation of an article, originally written in Italian, by Menabrae: 'Sketch of Analytical Engine Invented by Charles Babbage'. Lovelace added a commentary of seven entries, labelled 'Note A' to 'Note G' to the translation and initialed each her entries 'A.A.L'. The full translation and commentary are available on the web at the time of writing: `http://www.fourmilab.ch/babbage/sketch.html`.

It is truly astonishing for any researcher in Source Code Analysis and Manipulation to read Lovelace's account of the the coding of the engine, which is by far the larger part of the article at approximately 75% of the overall article length. In her seven prescient notes she on the one hand, recognises the practical importance of loops, optimization and debugging, while on the other she speculates on theoretical aspects such as artificial intelligence. All this, written in 1842: over a century before *any* form of source code would ever be executed by a computer.

At the heart of the Analytical Engine approach was *automation*. Automation is impossible without a source code that defines the process to be automated. Ada Lovelace, the daughter of a mathematician and a poet, wrote eloquently about the manner in which the source code inscribed on punched cards was to be used to program the Analytical Engine:

> "*The distinctive characteristic of the Analytical Engine, and that which has rendered it possible to endow mechanism with such extensive faculties as bid fair to make this engine the executive right-hand of abstract algebra, is the introduction into it of the principle which Jacquard devised for regulating, by means of punched cards, the most complicated patterns in the fabrication of brocaded stuffs. It is in this that the distinction between the two engines lies. Nothing of the sort exists in the Difference Engine. We may say most aptly, that the Analytical Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.*" Extract from Ada Lovelace's 'Note A' to her translation of Menabrae's manuscript.

This was more than an analogy: the inspiration for the 'punched cards' of the analytical engine came from the cards used in Joseph Jacquard's looms, the automation of which revolutionised the clothing industry. Jacquard's punched cards were the equivalent of straight line code; configurable, but ultimately producing the same garment on each rendition. In the same way, a musical box is constrained to identically reperform the piece inscribed by its metal teeth. In computing nomenclature, Jacquard's loom could be thought of as a Visual Display Unit, fashioned out of fabric, with the punch cards describing a kind of vector image, translated into a 'bit map' on cloth by execution on the loom. As source code, the language was clearly not Turing complete. What Babbage and Lovelace had in mind for the analytical engine was a flexible code capable of capturing arbitrary arithmetic computation and, thereby, essentially a Turing complete programming language in the sense that we would now understand it.

According to the SCAM definition of source code (Definition 1) the punched cards of the Analytical Engine do indeed inscribe source code. Even Jaquard's punch cards are source code according to the SCAM definition, with the loom as computer in exactly the same way that Lovelace describes it. Lovelace clearly recognised the profound significance of the advent of source code and made the first remarks about its analysis and manipulation in her note on the Analytical Engine. For example, she realised the need to analyze source code to find the most efficient expression of computational intent from those available:

> "*The order in which the operations shall be performed in every particular case is a very interesting and curious question, on which our space does not permit us fully to enter. In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.*" Extract from Ada Lovelace's 'Note D' to her translation of Menabrae's manuscript.

This is probably the first statement ever made in print regarding Source Code Analysis and Manipulation.

Looking back over 176 years since she wrote these words we can see the astonishing manner in which source code touches fundamental aspects of intellectual creativity, with profound practical ramifications. The ambit of source code has continued to widen: through computability source code encompasses mathematics, logic and philosophy. Work on evolutionary metaphors of computation bring it into contact with concepts of natural selection and evolution. In time, Sociology, Economics and even Political Science will all submit to source code, the analysis of which will yield insights into socioeconomic and political systems and processes.

IV. The Developing Importance of Source Code and Another Reason Why it will Continue to Grow in Importance

In Section II, I argued that source code analysis and manipulation will always be important because source code will always be important; as long as there is computation, there will always be new notations that will be used as source code. This argument is, I believe, uncontroversial: A cursory review of the development of programming languages will reveal a continuous cycle of invention, innovation and development. With each new concept in the expression of computational abstraction comes a concomitant need for research in analysis and manipulation. Each change in platform and implementation brings associated issues for the practice of source code analysis and manipulation. In this section, I would like to advance a potentially more controversial argument regarding what I believe will be a dramatic growth in the future importance of source code analysis and manipulation.

In 1948, Alan Turing wrote a paper (sometimes overlooked by comparison to his other groundbreaking work) entitled 'Checking a large routine' [78]. So what was this 'large routine', for which Turing sought a checking method and, in so doing, produced possibly the earliest work on software verification and testing? It is interesting to speculate on what might have constituted a large routine in 1948. In his short paper, Turning does not elaborate on the size he had in mind. The example Turing used to illustrate his ideas was

multiplication by repeated addition. Turing makes it clear that this is merely an illustrative example. This simple loop was sufficient for him to be able to tease out issues such as the separation of tester and developer, the use of embedded assertions and the distinction between partial correctness and total correctness.

In the same year, Eric Blair, under the pen name George Orwell, was speculating about how government might evolve. In order to compute a date for his book's title, it is widely believed that he took the simple expedient of transposing the two last digits of the date the manuscript was completed (December 1948), yielding the now famous title 'Nineteen Eighty–Four' [71]. In the software development community we have had some problems of our own with two digit date fields [72]. Transposing the last two digits of the present date will not move us forward. Instead, let us allow ourselves to speculate about the development of source code over a similar time frame. Let us move our focus, first from 1948 to 2010 and then, from 2010 to 2072.

In 1948, the structure of DNA was still, as yet, unknown. Social, financial and governmental processes were entirely mediated by humans. A 'data base' was still the term most likely used to refer to a collection of papers. By this time there were, of course, electrical calculating machines and even primitive computers. However, the idea of a 'computer' might, in the public mind, still refer equally well to the human operator as it might to the machine. Looking forward by the same amount of time, what will we think of as a 'large routine' in 2072?

Having unlocked the source code that underlies the characteristics of our own biological systems, is it likely, perhaps inevitable, that we might move from analysis of this new source code to manipulation. Humans will become the first species on earth to perform self-modifying code. As a research and practitioner community working on Source Code Analysis and Manipulation we are only too well aware of the possibilities this creates, for good and ill. I believe that the idea of our programming our own source code will not be science fiction in 2072. We had better start to prepare for this: who knows what the consequences of implementation bugs and misunderstood requirements will be in this new source code? We shall also have to contend with the feature interaction problems and other unintended consequences.

It is a tired analogy that describes DNA as 'the source code of life'. Nonetheless, worn though this observation may seem, it continues to inspire and confound. I am simply taking the analogy to its logical conclusion. It is not inconceivable that this new source code may become a 'programming language' in its own right. This year, there have been credible claims that the first steps towards the programming language 'DNA++' have already been taken [36].

Currently, the term 'bioinformatics' is used to refer to the use of computers and the algorithms they implement to analyze the vast databases of DNA and other biological sequences available to us. It is a subject in its infancy. What will another 62 years of the development of bioinformatics research produce? Perhaps, by 2072, the notion of computers as analyzers of DNA will seem quaint, essentially mundane and faintly ridiculous. This is our present day impression of the Victorian vision of a bank of 'computers', busily consulting their logarithmic tables and inscribing the fruits of their labours in indian ink.

Governmental treaties such as the Maastricht Treaty [79] are famous, perhaps infamous, for their complexity, with the result that such treaties are themselves the subject of analysis in the literature on 'political science' [86]. Debates rage among member states of the European Union about the degree to which the rules and regulations of such treaties are followed. Might we start to use computers to check these 'large routines'? After all, they are rule based systems. It would be a reckless optimist who would predict future simplification of the rules that govern our social, economic and intergovernmental interactions. It seems far more realistic to presume that the complexity of these systems will become ever higher. Indeed, I would argue that is is precisely because we have large digital databases and automated computation and networked connectivity that we are able to countenance the existing complexity. As such, technology is not a passive witness, recording and adapting to complexity increase, but is an active driver of the very complexity it records.

Consider again, the SCAM definition of source code (Definition 1). Depending on how one chooses to define 'execution' it is possible to see how this definition could equally well apply for the code that defines the behaviour (execution) of a cell or an entire organism for which the source code is located in the DNA. It also captures, as source code, the rules and regulations that delimit governmental interactions, social networking, bureaucratic rules and economic transaction protocols. Humans and systems constructed of part human, part machines are the executors of this source code.

With this view of source code in mind, it would appear that the concept of source code is gradually spreading out from its foundation within computers, to almost all spheres of human activity. Biological organisms, financial systems and governments have all been compared to systems or machines, and so it should not be surprising that the rules that govern their behaviour could (and perhaps should) be thought of as source code.

I believe that the law of 'tendency to executability' described in Section II also applies to other rules and processes that humans create for themselves and for which we create systematic descriptions. We seek to 'execute' these procedures and processes in all aspects of our lives; in companies, in voluntary oganisations, in government, in laws and in treaties. There are two ways in which this law of execution manifests itself. The first, and longer–established, is the implementation of processes and procedures by the humans we sometimes disparagingly refer to as bureaucrats. Long before the advent of digital technology, humans served the role of computer, implementing the execution of these descriptions.

Franz Kafka noticed this apparently inherent human impulse towards executability and its potential to enmesh the human spirit in a bureaucratic cage [55]. He takes the unfortunate consequences of inflexible 'governmental source code' to a gruesome conclusion. In his novel The Trial, bureaucratic

execution of procedures leads to wrongful execution of an innocent victim.

The second and more recent example of the law of execution comes from our ability to replace the human bureaucrat with a cheaper, faster and more reliable software–centric equivalent. The idea that laws and treaties might be coded as source code is not new. Indeed, 25 years ago, there was already work on executable descriptions of laws. One notable example was the source codification of the British Nationality Act as a program in Prolog[73]. Partly because of the doctrine of binding precedent and partly as a result of humans' natural propensity for augmentation laws have increasingly become more numerous and more complex. The same applies to the procedures and processes that exist within and between organisations. As a result we can expect more automation of these processes through source codification.

Authors have begun to consider the implications of so-called 'e-government' and the potentially pernicious aspects of automation that accompany this trend [54]. At present, systems are restricted to merely electronic data capture; the bureaucrats' forms are now implemented in javascript rather than folded paper. However this opens a door to greater automation.

Automation speeds up processes and, it is often claimed, reduces costs. As engineers, we seek ever faster cheaper solutions. However, a side effect of this speed up is the gradual extinction of human decision making and the surrender of human sovereignty over aspects of the increasingly automated process.

We might speak figuratively of our world increasingly being 'governed by source code'. We should be careful we do not sleepwalk into an Orwellian world that really is *governed* by source code. In his novel 'nineteen eighty–four', Orwell envisioned the 'party' controlling every aspect of human existence, including language and through it, thought itself. I am surely neither the first nor the last author to raise the dystopian spectre of automation and its dehumanising potential. However, previous concerns have focussed on the legacy of Henry Ford and the mundane deadening of repetitive manual tasks [50]. There are also potentially pernicious effects of control of the intellectual space by a monster entirely of our own making. It may not be an 'Orwellian party' but 'the source code' that evermore controls our thought processes and mediates what can and cannot be done. Even without our intending it, perhaps this banal source code monster may come to *control that which can be expressed*.

In this future world of automated socioeconomic and governmental processes, it seems clear that the full and proper understanding of the source code that captures the processes will be a paramount concern. At present our aims to manage the complexity of the source code of software have presented great challenges. In future, we may have to raise our game to meet the challenge of world in which everything that matters to the human enterprise is, in one way or another, captured by some form of 'source code'. In this emergent paradigm of automated socioeconomic and governmental interaction, 'understanding the source code' will take on a new significance and urgency.

As source code comes to define the actions in which people and organisations may engage we will need new kinds of source code analysis. Source code will increasingly capture, delimit, prescribe and proscribe the permissible forms of communication between organisations and states and between states and their peoples. We currently rightly, but primarily, think of type theory [65] and abstract interpretation [27], [28] as routes to program correctness. When we reach the point where source code defines the parameters that may be exchanged between governmental bodies and their citizens, how much more important will be the lattice of types that describe these exchanges? Organisations and even states may have properties in their interactions that can only be understood in terms of source code, because of the (possibly unintended) high level effects of lower level automaton in code. In such a world, how much more important will it be to have precise yet efficient means of abstracting out these properties from the code?

For this reason, if for no other, I believe that source code analysis and manipulation will continue to grow in importance, but the goals of our analysis will move up the abstraction chain, as source code itself does the same. For our part, as a primarily engineering and scientific community, we cannot alone grapple with the social, ethical, legal and moral issues raised by the growing significance of source code. It is not for us to dictate the response required to the fundamental questions this raises. However, through source code analysis and manipulation we may hope to better understand the effects and influences of source code.

I hope that the community will extend outwards from analysis and manipulation of purely software source code, to embrace a wider conception of source code that includes 'socioeconomic and governmental source code' and even 'bio source code'. We have already seen that source code analysis can help us to understand the business rules of organisations [74]. This work draws on the central 'SCAM observation' that source code 'contains the only precise description of the behaviour of the system'. If an organisation relies heavily on source code, then it may not matter what the organisational documents prescribe nor what the managers believe about organisational rules; the source code knows better. It is the source code that implements these rules and, therefore, the source code that 'decides' precisely what are the rules.

Our response as a source code analysis and manipulation community has been to analyse the source code to extract business rules [74]. We have already seen that this is necessary for organisations that have lost sight of their business process. Such organisations are bound to turn to software engineers to extract the business logic from source code logic. My argument is that it is inevitable that we will increasingly have to adopt this form of post-hoc extraction and analysis in order to discover exactly what is going on in our source code governed world.

However, having allowed myself the luxury of considering the nature of what I believe will prove to be one of the great challenges of the 21st century, I must now turn to the relatively meagre technical contributions of this present position paper.

V. Slicing and Transformation: Two Identical Forms of Source Code Manipulations

In considering the relationship between analysis and manipulation it is helpful to review the SCAM conference's own definition of 'analysis' and 'manipulation' as captured in Definition 2 below. Like the definition of source code (Definition 1 above), it has remained unchanged in the call for papers for 10 years.

Definition 2 (Analysis and Manipulation): The term 'analysis' is taken to mean any automated or semi automated procedure which takes source code and yields insight into its meaning. The term 'manipulation' is taken to mean any automated or semi-automated procedure which takes and returns source code.

Consider the widely studied source code manipulation technique of program slicing [85]. Harman et al. [42] introduced a theoretical framework called the 'projection framework', for formalizing and comparing definitions of slicing as a pair of relations, containing an equivalence and an ordering relation. The idea is very simple: The equivalence relation captures the property that slicing seeks to preserve, while the ordering relation captures that property that slicing seeks to improve. If s is to be a slice of a program p then s and p should be equivalent according to the equivalence relation and s should be no worse than p according to the ordering relation. More formally, this s captured in the three definitions below (taken from the 2003 JSS paper [42]).

Definition 3 (Ordering):
A syntactic ordering, denoted by $\lesssim$, is any computable transitive reflexive relation on programs.

Definition 4 (Equivalence):
A semantic equivalence, denoted by $\approx$, is an equivalence relation on a projection of program semantics.

Definition 5 ($(\lesssim, \approx)$ Projection):
Given syntactic ordering $\lesssim$ and semantic equivalence $\approx$,

Program p is a $(\lesssim, \approx)$ *projection* of program $q \Leftrightarrow p \lesssim q \wedge p \approx q$

Traditionally [42], this framework was instantiated with Weiser's trajectory semantics for the equivalence relation and syntactic statement inclusion for the ordering so as to capture Weiser's 1984 definition of static program slicing [85]. Several authors have studied the way this framework can be instantiated to capture other forms of slicing, such as dynamic, conditioned and amorphous forms of slicing [8], [42]. Other authors have also addressed semantic questions about the meaning of slicing [35], [82].

One such work by Ward and Zedan [83] pointed out that the original 2003 interpretation of equivalence and ordering was flawed because it requires that semantic properties are captured by the equivalence relation, while syntactic properties are captured by the ordering relation. This interpretation fails to cater adequately for the manner in which a slice could be more defined than the original from which it was constructed; a slice may introduce termination though it may not remove it. Termination is clearly a 'semantic' property. However, because a traditional slice must be more defined than the program from which it is constructed, semantic termination has to form part of the ordering relation.

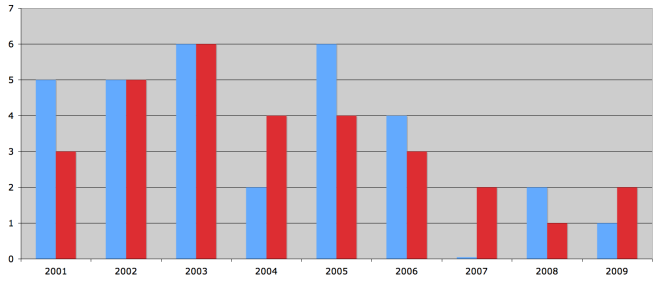


Fig. 1. Numbers of papers on Slicing and Transformation at SCAM over the years. The left hand bar of each pair shows the number of papers purely on transformation, while the right hand bar shows the number of papers purely on slicing.

In order to cater for this observation, I will, in the remainder of this section, drop the requirement that the equivalence relation should capture semantic properties, while the ordering should capture syntactic properties. Rather, I will allow either relation to capture either form of property. In this way, the projection framework simply says that slicing must hold some properties of the program constant and faithful to the original while changing some other properties (hopefully improving upon them in some way).

Over the 10 years of its history, the SCAM symposium has published 30 papers (about one sixth of all papers) on slicing and 31 (a further one sixth of all papers) on transformation. Figure 1 shows the numbers of papers on slicing and transformation over the years. In arriving at the figures, I made a (somewhat imprecise) assessment of whether a paper was primarily about slicing, primarily about transformation or primarily about neither.

The figures are not intended to be rigorous in any way and should be treated with a high degree of caution. In arriving at these figures I adopted the well-known source code analysis and manipulation principle of 'conservative approximation': I excluded papers that were not *clearly* about one of these two topics. For instance papers about the SDG [49] are clearly *related* to dependence analysis and, thereby, slicing. However, if a paper concerned with the SDG did not have a significant degree of work on *slicing* from the SDG it was not counted as a slicing paper. Similarly, there are many source code manipulations, such as refactoring, that could be thought of as forms of transformation. However, if the paper did not explicitly present itself as a transformation paper, then it was not counted as such.

My conservative analysis of the last nine years is that roughly one third of all papers at the conference concern one or other of these two topics (in roughly equal measures). In the discussions at the symposium there have been several debates about the relative merits of the two techniques.

However, using the projection framework, we can see that it is possible to consider traditional transformation as a special case of slicing. Those working on program transformation may be pleased to hear that it is also possible to consider slicing as a special case of transformation. As such I believe that

these are both examples of the same thing: *manipulation*. I think discussions about relative merits are a distraction and we should consider all forms of manipulation of source code to be equally valuable. Indeed, I believe that we should include refactoring and any other techniques that take source code and return source code to be equivalently part of the same category of 'source code manipulation'.

Let us consider the two relations in the projection framework in turn. A transformation is a slice in which the equivalence relation is functional equivalence. That is, traditionally speaking, since its inception in the 1970s [29] a transformation has been viewed as a function from programs to programs that must be 'meaning preserving'. For slicing, on the other hand, the equivalence relation is more restricted, so that only a projection of the meaning of the original program need be preserved during slicing. In this sense, a transformation is a specific case of slicing, in which the equivalence relation is the smallest possible and therefore the most restrictive. Slicing merely relaxes this to consider programs equivalent 'with respect to a slicing criterion'.

More formally, let $E_{(x,n)}$ denote equivalence with respect to some set of variables x at some set of program points n and let V denote the set of all variables and N the set of all possible program points. For transformation, the equivalence relation is $E_{(V,N)}$, whereas for slicing, it is $E_{(i,\{p\})}$ for some set of variables i and some program point n. The pair (i, n) is merely a *parameter* called the slicing criterion. The slicing criterion relaxed the traditional transformation view of equivalence, allowing it to capture many forms of projected meaning, while retaining the ability to capture traditional transformation as a special case. In this way, transformation is a special case of slicing.

Now let us turn our attention to the ordering relation. Traditionally [85], slices are constructed by statement deletion, so that a slice is a syntactic subset of the program from which it was constructed. This is more restrictive than transformation, which allows any change of syntax that preserves the meaning of the original. In program transformation the goal of transformation is left unspecified [26]. It may be, as Lovelace originally envisioned, that transformation is performed to improve the execution speed of the source code. However, other authors have used transformation, in its broadest sense, for many other purposes, including refactoring [33], [9], [46], [66], restructuring [39], [60], [80], reuse [24], testability [40], [44], [67], and migration [81]. In this sense a slice has a more restrictive ordering relation than transformation, which can use many different orderings depending upon the application.

More formally, using the projection framework, a slice is a $(\lesssim, \approx)$ Projection in which $\lesssim$ is restricted to program size and $\approx$ is arbitrary. A transformation is a $(\lesssim, \approx)$ Projection in which $\lesssim$ is arbitrary and $\approx$ is restricted to functional equivalence. Each is a special case of a more general $(\lesssim, \approx)$ Projection for which neither $\lesssim$ nor $\approx$ is restricted. Such a more general projection is, in SCAM terms, a source code manipulation. That is, in its most general form, we want all source code manipulations to hold some properties invariant according to a chosen equivalence relation while improving on others according to some chosen ordering relation.

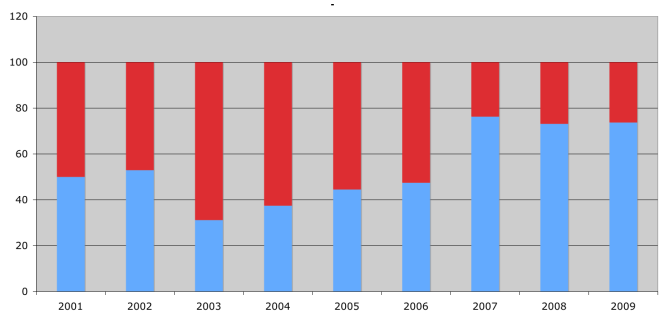


Fig. 2. Trend in papers on Analysis and Manipulation at SCAM. Upper bar shows percentage of papers on manipulation; lower bar shows percentage of papers on analysis.

At the time of writing, the developed and developing world are collectively entering into a period of austerity unprecedented in living memory, brought about by the global financial crisis of 2008. Many governments are asking themselves and their citizens the question: 'on what areas of government activity should we cut back our spending?'. The application of the 'cuts' is crude and imprecise. It has uncertain consequences and potentially unexpected side effects. However, suppose, after the next economic meltdown, source code has continued its present trend to the point where it has permeated every aspect of the government and economy of the world. Might we not be able to apply dependence analysis in order to discover potential effects (and otherwise unforeseen side effects) of governmental actions? Instead of 'cutting back' in a crude and imprecise manner, might our governments be able to use source code analysis and manipulation to 'slice back'?

VI. Analysis Purely for the Sake of Analysis

In recent years, the SCAM conference has shown an increasing trend for papers on Analysis over Manipulation (see Figure 2). In this section I want to make a case for source code analysis purely for its own sake. I also seek to argue that we can use Source Code Manipulation for Source Code Analysis. Those authors working on Source Code Manipulation who are alarmed by the trend towards analysis, therefore need not be concerned; manipulation is another way to do analysis.

The need to always seek an application before embarking on research in source code analysis is worthy. Should it become mandatory then it would become a tyranny. I wish to argue:

> "it is valuable to use Source Code Analysis and Manipulation purely and simply in order to discover that which may be there to be discovered".

In the longer term, this unconstrained analysis for its own sake may have profound and important applications. However, it is not always possible to *predict* the application at the outset of analysis and so this should not be *de rigueur*, nor should it be a pre–requisite for publication.

Of course, we often do use source code analysis and manipulation techniques with a purpose in mind. These applications lead to a prevalence of analysis and manipulation papers in conferences such as those concerned with Program Comprehension [19], [37], [57], [64], [70], Maintenance [2],

[7], [23], [25], [31], [59] and Reverse Engineering [32], [43], [68].

However, there is value in what might be termed ‘analysis for the sake of analysis’. That is, the pursuit of analysis as a voyage of discovery with an uncertain destination. With this approach, we treat source code more like a ‘naturally occurring phenomenon’; one that we seek to understand by experimental investigation. We seek to understand and expose structures and relationships in source code that occur almost as the economist might seek to explain the structures and relationships that occur in economic systems. Both are concerned with systems that are entirely created by humans but which exhibit behaviour for which those involved cannot have planned. Since we aim to identify unplanned outcomes, we cannot know that which we seek, we can only hope to recognise anything interesting and potentially important when we stumble across it. I believe that this trend in the evolution of source code will force us to consider it as having properties with an emergent character. This will drive us towards analysis that treats source code more like a naturally occurring phenomenon and less like a human-constructed artefact.

At the time of writing the economist has no ‘source code’ that can form a basis for economic analysis and must therefore rely upon simulations instead. How long will it be before source code is a viable mechanism for understanding and investigating world economic structures? How much more complex will computer software become before there is a need for not just models, but *simulations* that abstract from the detail in order to examine and predict the bahaviour of software systems?

The two worlds of economics and software are becoming increasingly interwoven. Therefore, we may as well try to understand software in a more exploratory manner that accepts that it will possess unintended, but nonetheless interesting, structures and relationships.

For small programs, this does not make sense; we can be sure that the purpose, effects and characteristics are largely those the designer intended and so there is no need to treat the source code as some sort of ‘undiscovered continent’. However, above a certain ‘critical mass’ (or perhaps critical complexity of interactions between code elements) it becomes impossible to completely characterize the code.

I am not referring merely to undiscovered bugs, but aspects of behaviour which are simply unanticipated (and will often not necessarily be localised). It has been known for some time that software systems develop their own dynamics as they grow and evolve [61], [62]. I am taking this view to its logical conclusion. That is, we should accept that some of the properties that source code will exhibit are not those intentionally placed there by the designer. They will only be discovered by exploratory experimental investigation.

The SCAM conference satisfies a need for a conference in which those interested in source code analysis and manipulation can meet, unfettered by the constraints imposed by the constraints imposed by specific applications. This is not to say that these applications are unimportant; they are very important. However, there is a need for a forum for discussion of tools, techniques, theory and algorithms, for analysis and manipulation that *does not* target any particular application. In the words of the SCAM call for papers:

> “*While much attention in the wider software engineering community is properly directed towards other aspects of systems development and evolution, such as specification, design and requirements engineering, it is the source code that contains the only precise description of the behaviour of the system. The analysis and manipulation of source code thus remains a pressing concern.*”

This is not to say that the work that has appeared in the conference has not been applied; it has. Space does not permit a comprehensive list of all the applications that have been addressed in the conference proceedings. As an indication of the breadth covered, a chronological list of applications from the past ten years of the Working Conference on Source Code Analysis and Manipulation includes Security [84], Documentation [5], Re-engineering [58], Prediction [21], Change Management [34], Inspections [20], Design Patterns [4], Testing [52], Clone Detection [77] and Evolution [76].

I am not making a case for application–free research. What I propose is that we should continue to allow space for analysis and manipulation that yields interesting results, but for which we cannot, at present, discern a definite application. Source code analysis is an *end in itself* and we should not be afraid to say so.

In the future, we may hope that a software engineer would be provided with a suite of tools for analysis and manipulation of source code that encourages and facilitates this kind of exploratory investigation. This can have several benefits. Initially, such an approach can be used to support computer science education. In the laboratory and the classroom, students would be supported by an exploratory investigative approach to source code. Lectures that teach programming from first principles through necessarily small toy examples can be supported by investigative, ‘browse and dig’ classes, in which the student browses a large code base, occasionally using more in-depth analysis to dig deeper when an interesting feature is noticed. Exploration can help students to become fluent with the most common of software engineering tasks; reading and evolving other engineers’ code. In this regard, the agenda that speaks of ‘analysis for the sake of analysis’ is not entirely without practical application.

However, it is not merely in the classroom that ‘analysis for its own sake’ is potentially valuable. Researchers can and should be involved in the process of exploratory analysis of source code. If we are honest with ourselves, many of our techniques originate in such speculative and exploratory analysis: we notice a trend or pattern in the code, formulate a hypothesis, develop tools to investigate and then, occasionally, are pleased to confirm our hypothesis.

The strictures of academic discourse then render our discoveries in reverse chronological order in a publication. It is usually inefficient for other scientists to read our work ‘bottom up’. The reader of a scientific paper does not always want to retrace the author’s steps along their voyage of discovery. However, our approach to scientific writing creates an unrealistic and unrepresentative view of the process of

scientific discovery and can adversely influence the manner in which future students of a discipline attempt to practice their subject.

VII. Some Results from My Own Voyage in Source Code Analysis

I should like to take the opportunity, afforded to me by this keynote invitation, to illustrate 'source code analysis as an exploratory voyage of discovery' by presenting the results of some of my own collaborative work with colleagues. I will take the liberty of presenting this work in the 'wrong academic writing style': informal and bottom up. I am sure my experience is similar to that of many in our community and within the wider scientific and engineering research community and so perhaps it is useful, once in a while, to admit this and to uncover just a little of this personal 'voyage of discovery'.

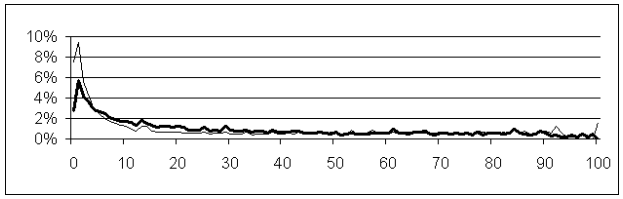
The rest of this section reviews the way we arrived at some of the results we have previously published on source code analysis manipulation by just such a process of experimentation and discovery. This work has been conducted in collaboration with Dave Binkley, Keith Gallagher, Nicolas Gold and Jens Krinke, all of whom I gratefully acknowledge for allowing me to present our work in the following manner.

In 2002, Dave Binkley and I started to use the emerging robust industrial strength slicing tool CodeSurfer from Grammatech [38] for slicing what we thought of at the time as a set of large real world programs. Our goal was simply to assess the size of program slices [13], [15]. In order to construct the large number of slices required, Dave needed to develop several novel slice efficiency techniques, which led to some potentially valuable developments in efficient slice computation that were published in this conference series [16], and subsequently, in extended form, in TOPLAS [18].

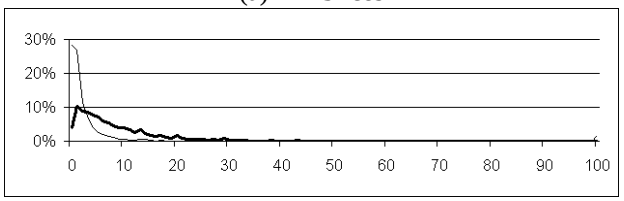
At the same time I was developing a different line of research in Search based Software Engineering (SBSE) [45] and applying this to software testing [6] and so Dave and I started to consider the way dependence might influence testability; the more parameters a predicate depended upon, the larger the search space for the test data generator and so the harder, so we thought, would be the search for branch adequate test data.

Having obtained the results of an analysis of predicate dependence, we noticed a statistically significant trend for functions with larger numbers of formal parameters to depend upon smaller proportions of these parameters. This was not the result we set out to find, but it was the most interesting finding to come from the work, which was subsequently published with only a glancing mention of the originally intended application to testing [14], [17]. The visualisations we used to explain our results proved useful in analysing programs for DaimlerChrysler in some (sadly rather rare) paid consultancy work. This work for DaimlerChrysler concerned the application of source code analysis to find problematic dependence features [10].

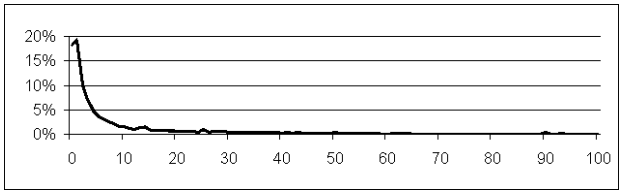
As we analysed more programs, we noticed two phenomena that we could not initially explain. These phenomena related to the distribution of forward slice sizes (compared to backward slice sizes) and the strange jumps in the monotonic order of slice sizes. I would like to (re)present these results in a little more technical detail since I believe that they illustrate my idea of source code analysis and manipulation as an uncertain voyage of experimental discovery, rather than a deterministic, pre-determined top down analysis from hypotheses to conclusion.



(a) All Slices



(b) Control Dependence Only Slices



(c) Data Dependence Only Slices

Fig. 3. Forward (faint) and Backward (bold) slice size distributions over all criteria [11]

In theory, forward and backward slicing are simply duals of one-another and so there appears to be little point in studying both. Results for forward slicing should be merely a mirror–image of results for backward slicing. After all, we are simply looking 'the opposite way' along the dependence arrows.

It is true that the *average* size of a set of forward slices of procedure or program will be identical to the average size of its backward slices. However the distributions of these slices are very different [11]. Forward slices are typically much smaller than backward slices. That is, there is typically a large set of very small forward slices and a few extremely large forward slices. By contrast, there are few exceptionally large backward slices, but also fewer very small backward slices.

It took us quite a while to realize that this phenomenon is a product of the structured nature of most of the code studied and the way in which control dependence influences slice size. After several failed attempts to get the work published, Dave realised that the reason for the difference was entirely down to the difference between control dependence and data dependence and the fact that most programs are relatively structured. We then set about investigating this by considering 'data only' and 'control only' slices of programs to explore the difference.

Figure 3a shows the size distributions over all slices (normalized as a percentage of the procedure from which they are

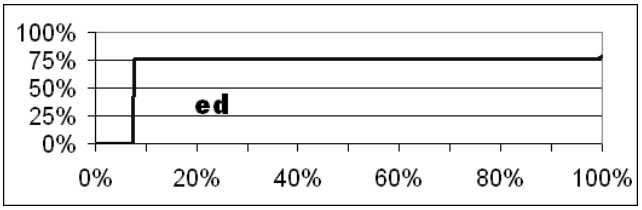


Fig. 4. Monotonic Slice size Graph of the open source program ed

taken), while Figures 3b and Figure 3c show the results for all possible slices using, respectively, only control dependence and only data dependence. Backward slice distributions are shown in bold; forward slice distributions unboldened. A point at (x, y) on one of these graphs indicates that y% of the distribution of slices consist of slices which include x% of the procedures from which they are constructed. Notice that the forward and backward slice size distributions for data-only slices are almost identical, but the forward and backward distributions for control only slices are markedly different In the paper [11] we provide a more detailed analysis.

As we analysed the results we were getting from the set of all slices of the programs we were collecting, we noticed a strange pattern in many of the graphs. When we plotted the size of slices in a monotonically increasing order, there were 'shelves'; the plot of the size of the slices would suddenly 'fall off a cliff drop'. An example can be seen in Figure 4, which shows a graph of the set of all slices of the program ed ordered by size. We called this graph a 'Monotonic Slice size Graph' (MSG). The MSG for ed shows a very dramatic example of the 'cliff drop' phenomena. The slices of ed fall into two categories: a large number of enormous slices all of the same size and a few very small slices. It seemed very curious that a program would (or even could) have so many large slices and counter-intuitive that they would all turn out to have the same size. We noticed that more and more of the programs we studied possessed MSGs with this characteristic 'shelf'. We started to try to explain it.

We realised that the shelves on the MSGs indicated the presence of dependence clusters; sets of statements all of which depend upon one another. We realised that if slices were of identical size, then they were likely to have identical content. We experimented with this and found that it was, indeed, the case. It was only then that we formed the hypothesis that programs may contain large dependence clusters and used the MSG as a mechanism for identifying them quickly by eye. In our first paper on the topic we demonstrated, empirically, that MSGs are a good approximation with which to find large dependence clusters by eye (two non–trivial slices of the same size are likely to be the same slice) and we looked at ways to re-factor programs that had large dependence clusters, since such clusters seemed to be a generally 'bad' phenomenon [12].

When Dave and I discussed these results with Jens Krinke and Keith Gallagher, we found that they had made similar observations and so we set about a combined effort to analyse the results for more programs. Nicolas Gold attacked the problem with Formal Concept Analysis and we had many exciting and interesting discussions about the nature of dependence clusters, ultimately leading to a joint paper on the topic in TOPLAS [41]. We were surprised that, once we had noticed this phenomenon, it seemed that almost *all* the programs we studied possessed large dependence clusters.

To give the reader some data to back this claim, Figure 5 (from the TOPLAS paper) shows a count of programs with large dependence clusters for various *largeness thresholds*. It is immediately clear from looking at this figure just how prevalent these large clusters are. For instance, if we set the threshold for 'largeness' at 10% all but 5 of the 45 programs we studied have large dependence clusters.

I think that if 10% of my program were to lie in one cluster of dependence then that is something I ought to know about and would have to take into account in almost any other analysis of the program. If the reader thinks that a threshold of 10% is not large enough to be a cause for concern, then a different threshold can be chosen and the resulting number of programs with large dependence clusters according to this more stringent criterion for 'largeness' can be read off from Figure5.

Since this discovery, we have found that all our empirical work that uses any kind of dependence analysis must first take into account whether a program has large dependence clusters or not. Programs with very large clusters tend to produce very different results for all the other kinds of analysis we have performed. Other researchers who find themselves experimentally exploring the impact of some form of dependence–related analysis might want to check their experimental subjects for the presence of very large clusters. If one simply applies a dependence–based technique to a set of programs, a potentially promising technique might be abandoned because its performance or behaviour is adversely affected by very large dependence clusters. Fortunately, a simple check can be performed by plotting the MSG and checking for the tell–tale 'shelves' that Dave Binkley initially noticed.

I believe that the discovery of large dependence clusters in source code, if replicated, may turn out to be important because of the importance of dependence analysis and the far–reaching implications of such large tight knots of dependence. They will clearly impact in comprehension, testing and maintenance activities and research work on these topics that rests upon dependence analysis. Of course, I cannot claim that such findings will have any further bearing beyond the source code analysis and manipulation community itself. Nonetheless, I hope that this section has illustrated how source code analysis can be something of an uncertain journey of discovery and that there are interesting structures and relationships to be discovered 'out there' in the growing corpus of source code available. I believe that as source code reaches further into the fabric of our global financial and governmental organisations and processes, the discoveries yet to be made may have far greater significance than those I have outlined in this section.

VIII. Conclusion

In the 1970s, the prevailing view in computer science research literature regarded software as a purely mathematical object that was to be entirely the subject of mathematical

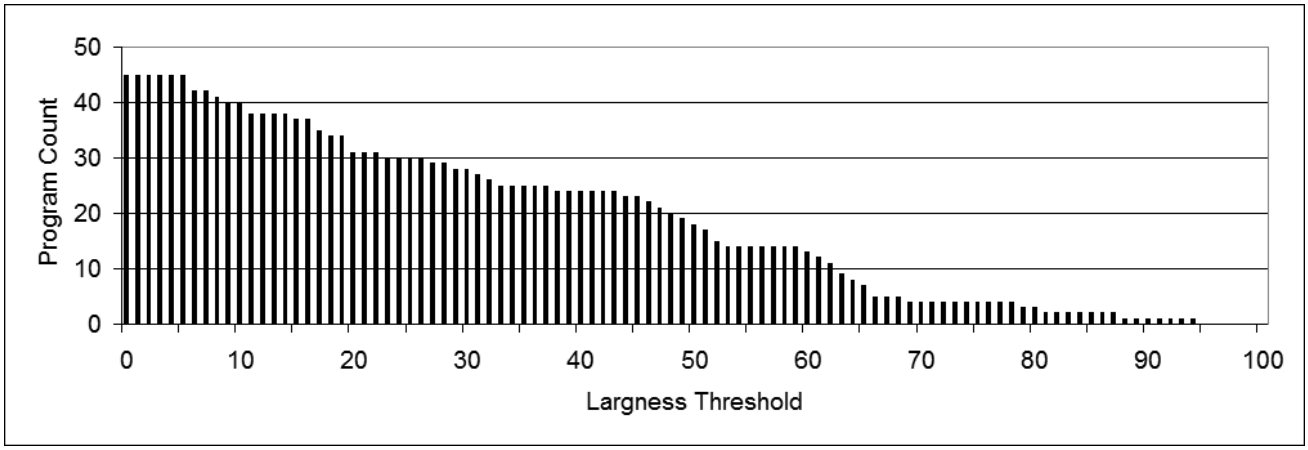


Fig. 5. The number of the 45 programs having large clusters for various largeness thresholds [41].

reasoning and understanding [30], [48]. This view point has been important, not least because it has led to practical technologies for verification; at first merely for small programs, but more recently non-trivial subsystems and components. For example, it has been demonstrated that device drivers of approximately 10,000 lines of code can be proved free from pointer violations through automated analysis [87], while other more 'partial' approaches may scale significantly [22]. This is work of exceptional importance.

However, by the 1990s it was already becoming clear that any hope of proving correct all real world software was in serious doubt [47]. The size of the body of source code with which we have to deal is simply growing at a much faster rate than any ability we have to completely master it as a formal mathematical object. I see no evidence to suggest that this trend will change; engineering practice will always outstrip formal scientific ability. Experimental and exploratory techniques will always be needed, as a complement to more rigorous formal techniques.

Increasing inter-connectedness, concurrent execution, and faster cycles of development, innovation and re-development will increase, dramatically, the size of the global software project with which we shall need to grapple. I have argued that it is inevitable that within current lifetimes, source code will have drawn in biological, social, economic and governmental processes. To some extent, it already has done so. I believe we shall soon reach a point at which we can 'understand these processes' through source code analysis. Indeed, it may be our *only hope* of understanding them, since the source code will have become, in the words of the SCAM call for papers 'the only precise description of the behaviour of the system'. As such, this promises an exceptionally interesting future for those working on the analysis and manipulation of source code.

However, as I have also argued, I believe that we are increasingly governed by source code, some of which takes forms we do not readily recognise as such. If we do not recognise that our treaties, rules, processes and procedures and possibly, even our own biology are gradually becoming source code, then we risk a technological tyranny. Source code analysis and manipulation has a crucial role to play in helping us escape this fate. If we fail to understand source code we are destined to be controlled by it. The understanding we shall need will go far beyond the business and engineering concerns of our present interest in 'program comprehension'.

IX. Acknowledgements

I am grateful to many colleagues for discussions that contributed to the ideas developed and reviewed in this paper. Space does not permit a full list and I apologise to those I may have neglected to mention. The ideas presented here have been shaped by discussions with Giulio Antoniol, Dave Binkley, Mark Bishop, Sue Black, Edmund Burke, Gerardo Canfora, John Clark, Jim Cordy, Sebastian Danicic, Andrea De Lucia, Max Di Penta, Chris Fox, Keith Gallagher, Nicolas Gold, Tibor Gyimòthy, Kathy Harman, Robert Hierons, Mike Holcombe, John Howroyd, Akos Kiss, Bogdan Korel, Jens Krinke, Arun Lakhotia, Bill Langdon, Phil McMinn, Malcolm Munro, Jeff Offutt, Peter O'Hearn, Mark Priestley, Marc Roper, Harry Sneed, Paolo Tonella, Xin Yao and Martin Ward.

References

[1] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *Fundamental Approaches to Software Engineering (FASE '09)*, vol. 5503. York, UK: Springer LNCS, Mar. 2009, pp. 216–230.

[2] D. C. Atkinson and W. G. Griswold, "Implementation techniques for efficient data-flow analysis of large programs," in *IEEE International Conference on Software Maintenance (ICSM'01)*. Los Alamitos, California, USA: IEEE Computer Society Press, Nov. 2001, pp. 52–61.

[3] R. J. C. Atkinson, *Stonehenge*. Penguin Books, 1956.

[4] L. Aversano, L. Cerulo, and M. Di Penta, "Relating the evolution of design patterns and crosscutting concerns," in *Proceedings of the* 7^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation*. Paris, France: IEEE Computer Society, 2007, pp. 180–192.

[5] F. Balmas, "Using dependence graphs as a support to document programs," in *Proceedings of the* 2^{nd} *IEEE International Workshop on Source Code Analysis and Manipulation*. Montreal, Canada: IEEE Computer Society, 2002, pp. 145–154.

[6] A. Baresel, D. W. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach," in *International Symposium on Software Testing and Analysis (ISSTA 2004)*, Jul. 2004, pp. 108–118.

[7] A. Beszédes and T. Gyimóthy, "Union slices for the approximation of the precise slice," in *IEEE International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, Oct. 2002, pp. 12–20.

[8] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and L. Ouarbya, "Formalizing executable dynamic and forward slicing," in 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*. Los Alamitos, California, USA: IEEE Computer Society Press, Sep. 2004, pp. 43–52.

[9] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Tool-supported refactoring of existing object-oriented code into aspects," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 698–717, 2006.

[10] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener, "Dependence anti patterns," in 4^{th} *International ERCIM Workshop on Software Evolution and Evolvability (Evol'08)*, L'Aquila, Italy, September 2008, pp. 25–34.

[11] D. Binkley and M. Harman, "Forward slices are smaller than backward slices," in 5^{th} *IEEE International Workshop on Source Code Analysis and Manipulation*. Los Alamitos, California, USA: IEEE Computer Society Press, 2005, pp. 15–24.

[12] ——, "Locating dependence clusters and dependence pollution," in 21^{st} *IEEE International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, 2005, pp. 177–186.

[13] D. W. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, pp. 1–32, 2007.

[14] D. W. Binkley and M. Harman, "An empirical study of predicate dependence levels and trends," in 25^{th} *IEEE International Conference and Software Engineering (ICSE 2003)*. Los Alamitos, California, USA: IEEE Computer Society Press, May 2003, pp. 330–339.

[15] ——, "A large-scale empirical study of forward and backward static slice size and context sensitivity," in *IEEE International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, Sep. 2003, pp. 44–53.

[16] ——, "Results from a large–scale study of performance optimization techniques for source code analyses based on graph reachability algorithms," in *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*. Los Alamitos, California, USA: IEEE Computer Society Press, Sep. 2003, pp. 203–212.

[17] ——, "Analysis and visualization of predicate dependence on formal parameters and global variables," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 715–735, 2004.

[18] D. W. Binkley, M. Harman, and J. Krinke, "Empirical study of optimization techniques for massive slicing," *ACM Transactions on Programming Languages and Systems*, vol. 30, pp. 3:1–3:33, 2007.

[19] D. W. Binkley, M. Harman, L. R. Raszewski, and C. Smith, "An empirical study of amorphous slicing as a program comprehension support tool," in 8^{th} *IEEE International Workshop on Program Comprehension*. Los Alamitos, California, USA: IEEE Computer Society Press, Jun. 2000, pp. 161–170.

[20] C. Boogerd and L. Moonen, "Prioritizing software inspection results using static profiling," in *Proceedings of the* 6^{th} *IEEE International Workshop on Source Code Analysis and Manipulation*. Philadelphia, USA: IEEE Computer Society Press, Sep. 2006, pp. 149–158.

[21] M. Bruntink and A. Deursen, "Predicting class testability using object-oriented metrics," in *Proceedings of the* 4^{th} *IEEE International Workshop on Source Code Analysis and Manipulation*. Chicago, IL, USA: IEEE Computer Society Press, Sep. 2004, pp. 36 – 145.

[22] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the* 36^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, Z. Shao and B. C. Pierce, Eds. Savannah, GA, USA: ACM, 2009, pp. 289–300.

[23] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca, "Software salvaging based on conditions," in *International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, Sep. 1994, pp. 424–433.

[24] W. C. Chu, P. Luker, and H. Yang, "Code understanding through program transformation for reusable component identification," in 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*. Los Alamitos, California, USA: IEEE Computer Society Press, May 1997.

[25] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *IEEE International Conference on Software Maintenance (ICSM'97)*. Los Alamitos, California, USA: IEEE Computer Society Press, 1997, pp. 188–195.

[26] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[27] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in 4^{th} *Principles Of Programming Languages (POPL 1977)*, Los Angeles, CA, Jan. 1977, pp. 238–252.

[28] ——, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, Aug. 1992.

[29] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica*, vol. 6, pp. 41–60, 1976.

[30] E. W. Dijkstra, *A discipline of programming*. Prentice Hall, 1972.

[31] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, 2003, special issue on ICSM 2001.

[32] T. Eisenbarth, R. Koschke, and G. Vogel, "Static trace extraction," in *IEEE Working Conference on Reverse Engineering*. Los Alamitos, California, USA: IEEE Computer Society Press, Oct. 2002, pp. 128–137.

[33] R. Ettinger and M. Verbaere, "Untangling: a slice extraction refactoring," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2004, pp. 93–101.

[34] B. Fluri, H. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," in *Proceedings of the* 5^{th} *IEEE International Workshop on Source Code Analysis and Manipulation*. Budapest, hungary: IEEE Computer Society, 2005, pp. 66–74.

[35] R. Giacobazzi and I. Mastroeni, "Non–standard semantics for program slicing," *Higher-Order and Symbolic Computation*, vol. 16, no. 4, pp. 297–339, 2003, special issue on Partial Evalution and Semantics-Based Program Manipulation.

[36] D. G. Gibson, J. I. Glass, C. Lartigue, V. N. Noskov, R.-Y. Chuang, M. A. Algire, G. A. Benders, M. G. Montague, L. Ma, M. M. Moodie, C. Merryman, S. Vashee, R. Krishnakumar, N. Assad-Garcia, C. Andrews-Pfannkoch, E. A. Denisova, L. Young, Z.-Q. Qi, T. H. Segall-Shapiro, C. H. Calvey, P. P. Parmar, C. A. H. III, H. O. Smith, and J. C. Venter, "Creation of a bacterial cell controlled by a chemically synthesized genome," *Science*, 2010.

[37] N. E. Gold and K. H. Bennett, "A flexible method for segmentation in concept assignment," in 9^{th} *IEEE International Workshop on Program Comprehension*. Los Alamitos, California, USA: IEEE Computer Society Press, May 2001, pp. 135–144.

[38] Grammatech Inc., "The codesurfer slicing system," 2002. [Online]. Available: www.grammatech.com

[39] W. G. Griswold and D. Notkin, "Automated assistence for program restructuring," Department of Computer Science and Engineering, University of California, Sand Diego, Technical Report CS92–221, Jan. 1993.

[40] M. Harman, "Open problems in testability transformation (keynote)," in *1st International Workshop on Search Based Testing (SBT 2008)*, Lillehammer, Norway, 2008.

[41] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 1, Oct. 2009, article 1.

[42] M. Harman, D. W. Binkley, and S. Danicic, "Amorphous program slicing," *Journal of Systems and Software*, vol. 68, no. 1, pp. 45–64, Oct. 2003.

[43] M. Harman, N. Gold, R. M. Hierons, and D. W. Binkley, "Code extraction algorithms which unify slicing and concept assignment," in *IEEE Working Conference on Reverse Engineering (WCRE 2002)*. Los Alamitos, California, USA: IEEE Computer Society Press, Oct. 2002, pp. 11 – 21.

[44] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, Jan. 2004.

[45] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[46] M. Harman and L. Tratt, "Pareto optimal search-based refactoring at the design level," in *GECCO 2007: Proceedings of the* 9^{th} *annual conference on Genetic and evolutionary computation*. London, UK: ACM Press, Jul. 2007, pp. 1106 – 1113.

[47] C. A. R. Hoare, "How did software get so reliable without proof?" in *IEEE International Conference on Software Engineering (ICSE'96)*. Los Alamitos, California, USA: IEEE Computer Society Press, 1996, keynote talk.

[48] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language Pascal," *Acta Informatica*, vol. 2, no. 4, pp. 335–355, Dec. 1973.

[49] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[50] A. Huxley, *Brave New World*. London, United Kingdom: Chatto and Windus, 1932.

[51] G. Ifrah, *The Universal History of Computing: From the Abacus to the Quantum Computer*. New York: John Wiley and Sons, 2001.

[52] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*. Beijing, China: IEEE Computer Society, 2008, pp. 249–258.

[53] C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed. Prentice Hall, 1990.

[54] S. Jones and B. Crowe, *Transformation not automation: The e-government challenge*. London: Demos, 2001.

[55] F. Kafka, *The Trial*. New York, NY, USA: Schloken Books, 1925.

[56] H. H. Kagdi, J. I. Maletic, and A. Sutton, "Context-free slicing of UML class models," in 21^{st} *IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 2005, pp. 635–638.

[57] B. Korel and J. Rilling, "Dynamic program slicing in understanding of program execution," in 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*. Los Alamitos, California, USA: IEEE Computer Society Press, May 1997, pp. 80–89.

[58] J. Kort and R. Lämmel, "Parse-tree annotations meet re-engineering concerns," in *Proceedings of the* 3^{rd} *IEEE International Workshop on Source Code Analysis and Manipulation*. Amsterdam, Netherlands: IEEE Computer Society, 2003, pp. 161–170.

[59] J. Krinke, "Evaluating context-sensitive slicing and chopping," in *IEEE International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, Oct. 2002, pp. 22–31.

[60] A. Lakhotia and J.-C. Deprez, "Restructuring programs by tucking statements into functions," *Information and Software Technology Special Issue on Program Slicing*, vol. 40, no. 11 and 12, pp. 677–689, 1998.

[61] M. M. Lehman, "On understanding laws, evolution and conservation in the large program life cycle," *Journal of Systems and Software*, vol. 1(3), pp. 213–221, 1980.

[62] ——, "Software's future: Managing evolution," *IEEE Software*, vol. 15, no. 1, pp. 40–44, Jan. / Feb. 1998.

[63] M. Mahoney, "The history of computing in the history of technology," *Annals of the History of Computing*, vol. 10, no. 2, pp. 113–125, 1988.

[64] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," in 13^{th} *IEEE International Workshop on Program Comprehension (IWPC'05)*. IEEE Computer Society Press, 2005, pp. 33–42.

[65] P. Martin Löf, "Constructive mathematics and computer programming," in *Mathematical Logic and Computer Programming*, C. A. R. Hoare and J. C. Shepherdson, Eds. Prentice-Hall, 1984, pp. 167–184.

[66] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in *SSR '01: Proceedings of the 2001 symposium on Software reusability*. New York, NY, USA: ACM Press, 2001, pp. 31–40.

[67] P. McMinn, "Search-based failure discovery using testability transformations to generate pseudo-oracles," in *Genetic and Evolutionary Computation Conference (GECCO 2009)*, F. Rothlauf, Ed. Montreal, Québec, Canada: ACM, 2009, pp. 1689–1696.

[68] T. Meyers and D. W. Binkley, "Slice-based cohesion metrics and software intervention," in 11^{th} *IEEE Working Conference on Reverse Engineering*. Los Alamitos, California, USA: IEEE Computer Society Press, Nov. 2004, pp. 256–266.

[69] R. S. Newall, "Stonehenge: A review," *Antiquity*, vol. 30, no. 119, 1956.

[70] A. Orso, S. Sinha, and M. J. Harrold, "Effects of pointers on data dependences," in 9^{th} *IEEE International Workshop on Program Comprehension*. Los Alamitos, California, USA: IEEE Computer Society Press, May 2001, pp. 39–49.

[71] G. Orwell, *Nineteen Eighty-four*. Penguin, 1949.

[72] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the Year 2000 problem," in 6^{th} *European Software Engineering Conference and* 5^{Pth} *ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, Zurich, Switzerland, 1997, pp. 432–449, lecture Notes in Computer Science, Volume 1301.

[73] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory, "The British Nationality Act as a logic program," *Communications of the ACM*, vol. 29, no. 5, pp. 370–386, 1986.

[74] H. M. Sneed and K. Erdös, "Extracting business rules from source code," in 4^{th} *IEEE Workshop on Program Comprehension (WPC '96)*. Berlin, Germany: IEEE Computer Society, 1996, p. 240==247.

[75] J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[76] S. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *Proceedings of the* 10^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation*. Timisoara, Romania: IEEE Computer Society, 2010, p. To appear (in this volume).

[77] R. Tiarks, R. Koschke, and R. Falke, "An assessment of type-3 clones as detected by state-of-the-art tools," in *Proceedings of the* 9^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation*. Edmonton, Canada: IEEE Computer Society, 2009, pp. 67–76.

[78] A. M. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*. Cambridge, England: University Mathematical Laboratory, Jun. 1949, pp. 67–69.

[79] E. Union, "The Maastrict Treaty," 1992, available on line at http://eur-lex.europa.eu/en/treaties/.

[80] A. van Deursen and T. Kuipers, "Identifying objects using cluster and concept analysis," Centrum voor Wiskunde en Informatica (CWI), Tech. Rep. SEN-R9814, Sep. 1998. [Online]. Available: ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R9814.ps.Z

[81] M. Ward, "Assembler to C migration using the FermaT transformation system," in *IEEE International Conference on Software Maintenance (ICSM'99)*. Los Alamitos, California, USA: IEEE Computer Society Press, Aug. 1999.

[82] ——, "The formal approach to source code analysis and manipulation," in 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*. Los Alamitos, California, USA: IEEE Computer Society Press, 2001, pp. 185–193.

[83] M. P. Ward and H. Zedan, "Deriving a slicing algorithm via FermaT transformations," *IEEE Transactions on Software Engineering*, p. to appear, 2010.

[84] M. Weber, V. Shah, and C. Ren, "A case study in detecting software security vulnerabilities using constraint optimization," in *Proceedings of the* 1^{st} *IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2001)*. Florence, Italy: IEEE Computer Society, 2001, pp. 3–13.

[85] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[86] D. Wincott, "Federalism and the european union: The scope and limits of the treaty of Maastricht," *international Political Science Review*, vol. 17, no. 4, pp. 403–415, 1996.

[87] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn, "Scalable shape analysis for systems code," in 20^{th} *International Conference on Computer Aided Verification (CAV 2008)*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds., vol. 5123. Princeton, NJ, USA: Springer, 2008, pp. 385–398.