CREST 10th Anniversary

Automated Software Transplantation

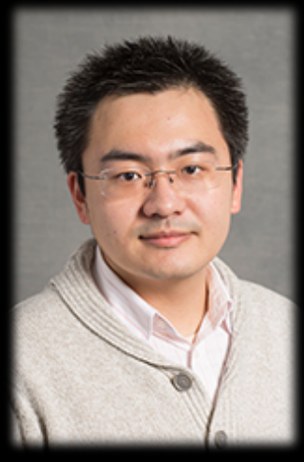
Alexandru Marginean

CREST, University College London

People from CREST

Earl T. Barr

Mark Harman

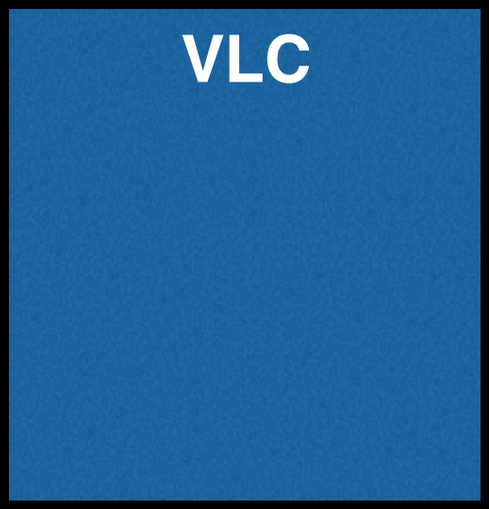


Yue Jia

William B. Langdon

Justyna Petke

Why Autotransplantation?

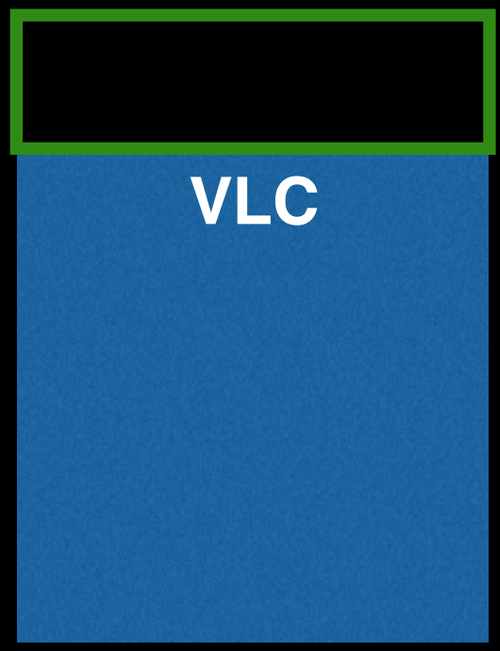


Why Autotransplantation?

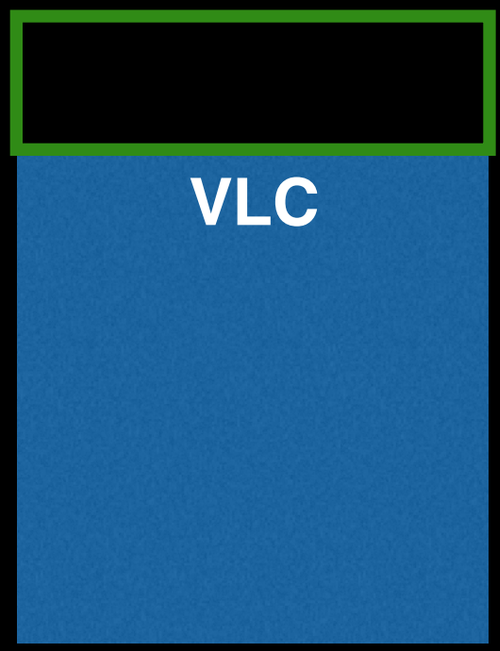
Why Autotransplantation?

VLC

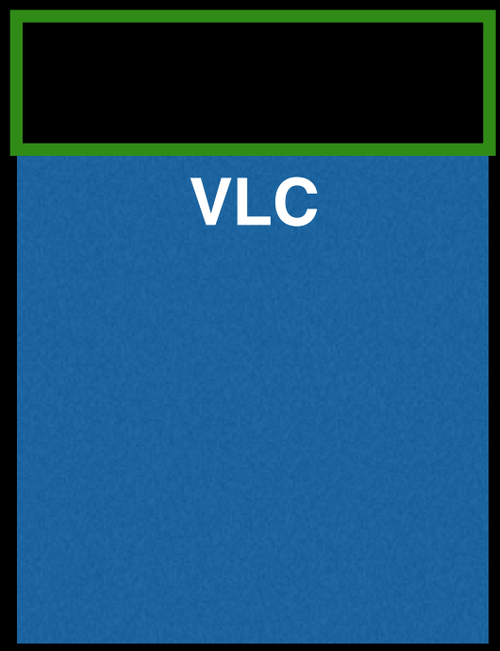
Why Autotransplantation?



Why Autotransplantation?

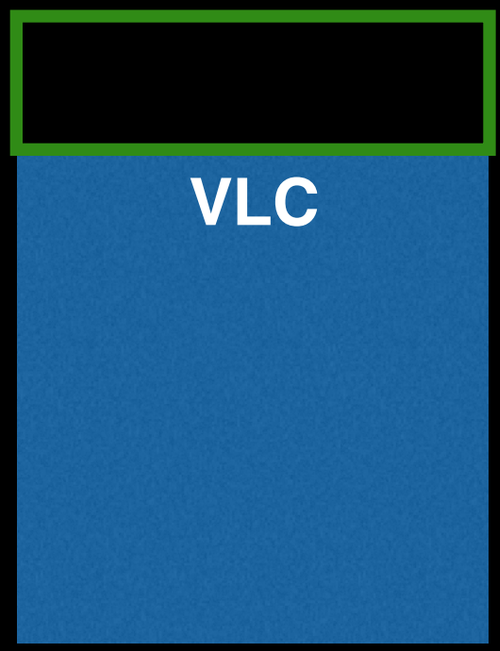


Why Autotransplantation?

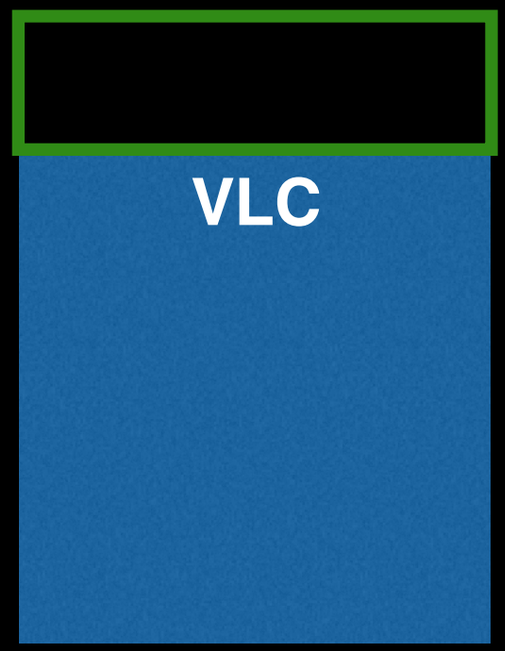


Start from scratch

Why Autotransplantation?



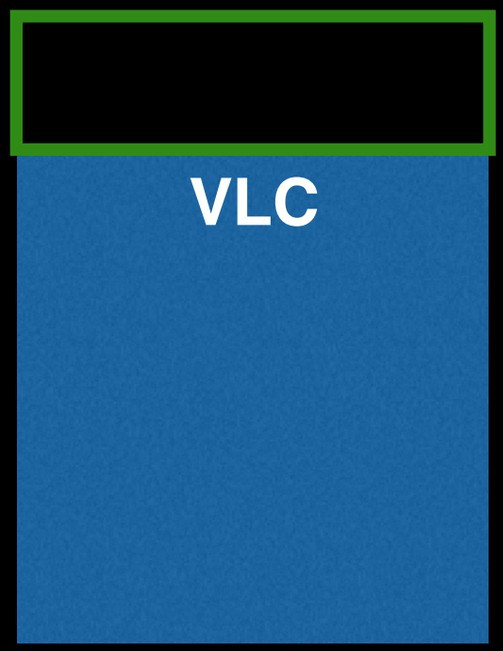
Why Autotransplantation?



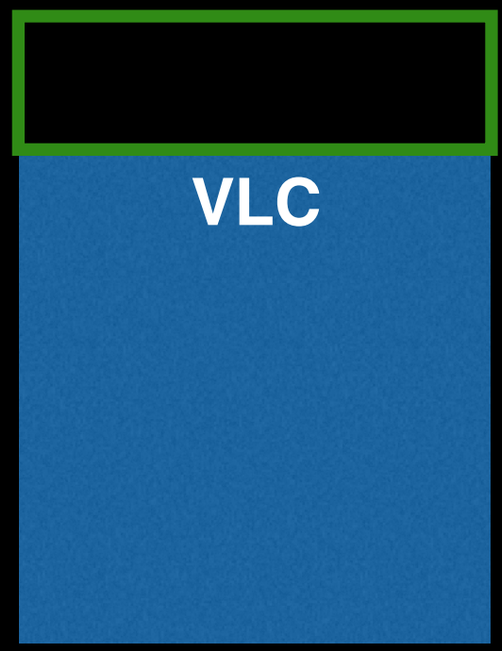
Check open source repositories

Start from scratch

Why Autotransplantation?



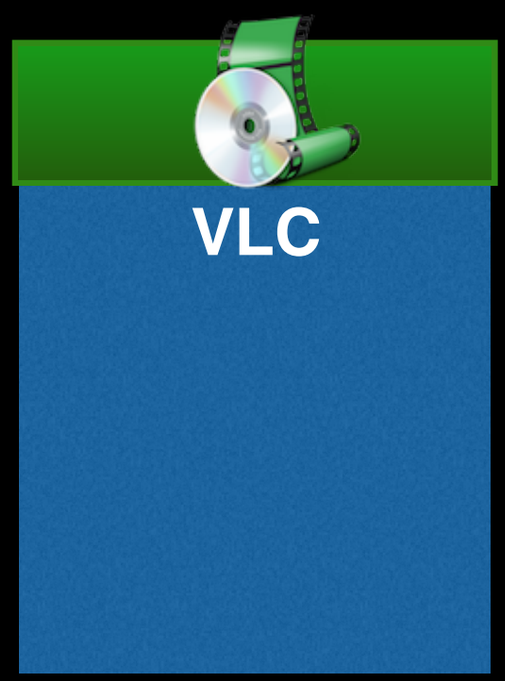
Why Autotransplantation?



Check open source repositories

Start from scratch

Why Autotransplantation?

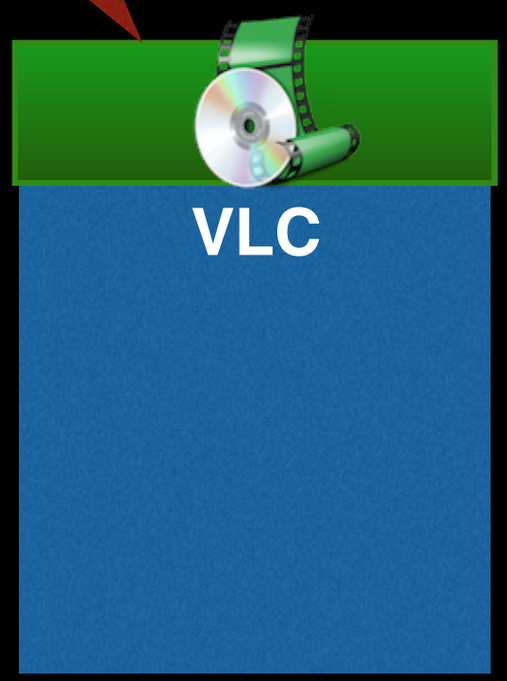


Check open source repositories

Start from scratch

Why A ansplantation?

~100 players

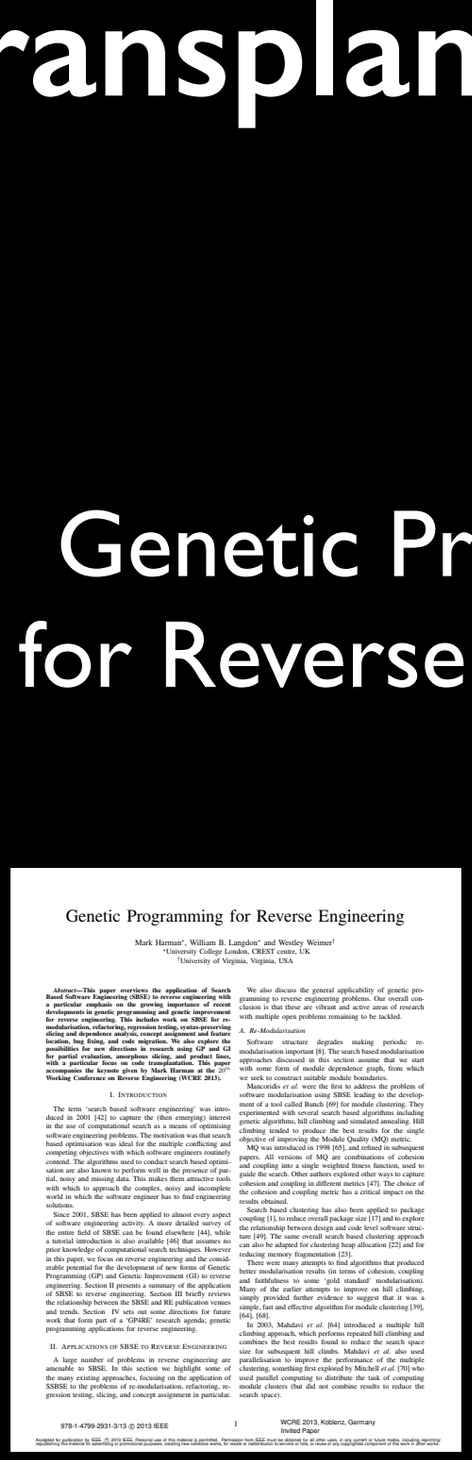


Check open source repositories

Start from scratch

Code Transplants

Genetic Programming
for Reverse Engineering



Genetic Programming for Reverse Engineering

Mark Harman*, William B. Langdon* and Westley Weimer†
*University College London, CREST centre, UK
†University of Virginia, Virginia, USA

Abstract—This paper overviews the application of Search Based Software Engineering (SBSE) to reverse engineering with a particular emphasis on the growing importance of recent developments in genetic programming and genetic improvement for reverse engineering. This includes work on SBSE for re-modularisation, refactoring, regression testing, syntax-preserving slicing and dependence analysis, concept assignment and feature location, bug fixing, and code migration. We also explore the possibilities for new directions in research using GP and GI for partial evaluation, amorphous slicing, and product lines, with a particular focus on code transplantation. This paper accompanies the keynote given by Mark Harman at the 20th Working Conference on Reverse Engineering (WCRE 2013).

I. Introduction

The term 'search based software engineering' was introduced in 2001 [42] to capture the (then emerging) interest in the use of computational search as a means of optimising software engineering problems. The motivation was that search based optimisation was ideal for the multiple conflicting and competing objectives with which software engineers routinely contend. The algorithms used to conduct search based optimisation are also known to perform well in the presence of partial, noisy and missing data. This makes them attractive tools with which to approach the complex, noisy and incomplete world in which the software engineer has to find engineering solutions.

Since 2001, SBSE has been applied to almost every aspect of software engineering activity. A more detailed survey of the entire field of SBSE can be found elsewhere [44], while a tutorial introduction is also available [46] that assumes no prior knowledge of computational search techniques. However in this paper, we focus on reverse engineering and the considerable potential for the development of new forms of Genetic Programming (GP) and Genetic Improvement (GI) to reverse engineering. Section II presents a summary of the application of SBSE to reverse engineering. Section III briefly reviews the relationship between the SBSE and RE publication venues and trends. Section IV sets out some directions for future work that form part of a 'GPiRE' research agenda; genetic programming applications for reverse engineering.

II. Applications of SBSE to Reverse Engineering

A large number of problems in reverse engineering are amenable to SBSE. In this section we highlight some of the many existing approaches, focusing on the application of SSBSE to the problems of re-modularisation, refactoring, regression testing, slicing, and concept assignment in particular. We also discuss the general applicability of genetic programming to reverse engineering problems. Our overall conclusion is that these are vibrant and active areas of research with multiple open problems remaining to be tackled.

A. Re-Modularization

Software structure degrades making periodic re-modularisation important [8]. The search based modularisation approaches discussed in this section assume that we start with some form of module dependence graph, from which we seek to construct suitable module boundaries.

Mancoridis *et al.* were the first to address the problem of software modularisation using SBSE leading to the development of a tool called Bunch [69] for module clustering. They experimented with several search based algorithms including genetic algorithms, hill climbing and simulated annealing. Hill climbing tended to produce the best results for the single objective of improving the Module Quality (MQ) metric.

MQ was introduced in 1998 [65], and refined in subsequent papers. All versions of MQ are combinations of cohesion and coupling into a single weighted fitness function, used to guide the search. Other authors explored other ways to capture cohesion and coupling in different metrics [47]. The choice of the cohesion and coupling metric has a critical impact on the results obtained.

Search based clustering has also been applied to package coupling [1], to reduce overall package size [17] and to explore the relationship between design and code level software structure [49]. The same overall search based clustering approach can also be adapted for clustering heap allocation [22] and for reducing memory fragmentation [23].

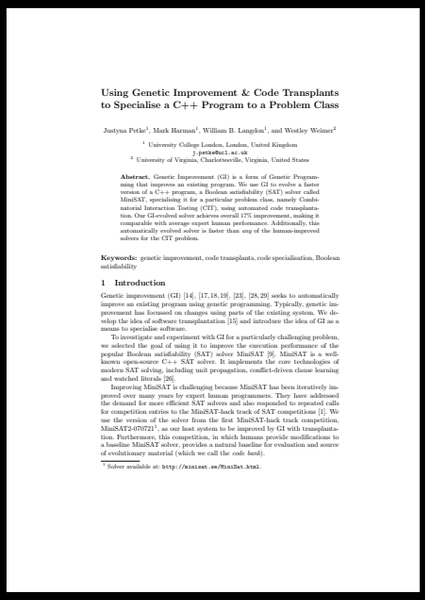
There were many attempts to find algorithms that produced better modularisation results (in terms of cohesion, coupling and faithfulness to some 'gold standard' modularisation). Many of the earlier attempts to improve on hill climbing, simply provided further evidence to suggest that it was a simple, fast and effective algorithm for module clustering [39], [64], [68].

In 2003, Mahdavi *et al.* [64] introduced a multiple hill climbing approach, which performs repeated hill climbing and combines the best results found to reduce the search space size for subsequent hill climbs. Mahdavi *et al.* also used parallelisation to improve the performance of the multiple clustering, something first explored by Mitchell *et al.* [70] who used parallel computing to distribute the task of computing module clusters (but did not combine results to reduce the search space).

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany
Invited Paper

Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class



Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class

Justyna Petke[1], Mark Harman[1], William B. Langdon[1], and Westley Weimer[2]
[1] University College London, London, United Kingdom
j.petke@ucl.ac.uk
[2] University of Virginia, Charlottesville, Virginia, United States

Abstract. Genetic Improvement (GI) is a form of Genetic Programming that improves an existing program. We use GI to evolve a faster version of a C++ program, a Boolean satisfiability (SAT) solver called MiniSAT, specialising it for a particular problem class, namely Combinatorial Interaction Testing (CIT), using automated code transplantation. Our GI-evolved solver achieves overall 17% improvement, making it comparable with average expert human performance. Additionally, this automatically evolved solver is faster than any of the human-improved solvers for the CIT problem.

Keywords: genetic improvement, code transplants, code specialisation, Boolean satisfiability

1 Introduction

Genetic improvement (GI) [14], [17, 18, 19], [23], [28, 29] seeks to automatically improve an existing program using genetic programming. Typically, genetic improvement has focussed on changes using parts of the existing system. We develop the idea of software transplantation [15] and introduce the idea of GI as a means to specialise software.

To investigate and experiment with GI for a particularly challenging problem, we selected the goal of using it to improve the execution performance of the popular Boolean satisfiability (SAT) solver MiniSAT [9]. MiniSAT is a well-known open-source C++ SAT solver. It implements the core technologies of modern SAT solving, including unit propagation, conflict-driven clause learning and watched literals [26].

Improving MiniSAT is challenging because MiniSAT has been iteratively improved over many years by expert human programmers. They have addressed the demand for more efficient SAT solvers and also responded to repeated calls for competition entries to the MiniSAT-hack track of SAT competitions [1]. We use the version of the solver from the first MiniSAT-hack track competition, MiniSAT2-070721[4], as our host system to be improved by GI with transplantation. Furthermore, this competition, in which humans provide modifications to a baseline MiniSAT solver, provides a natural baseline for evaluation and source of evolutionary material (which we call the code bank).

[4] Solver available at: http://minisat.se/MiniSat.html.

2013 — *Harman et al.*

2014 — *Petke et al*

Code Transplants

Genetic Programming
for Reverse Engineering

Genetic Programming for Reverse Engineering

Mark Harman*, William B. Langdon* and Westley Weimer†
*University College London, CREST centre, UK
†University of Virginia, Virginia, USA

Abstract—This paper overviews the application of Search Based Software Engineering (SBSE) to reverse engineering with a particular emphasis on the growing importance of recent developments in genetic programming and genetic improvement for reverse engineering. This includes work on SBSE for remodularisation, refactoring, regression testing, syntax-preserving slicing and dependence analysis, concept assignment and feature location, bug fixing, and code migration. We also explore the possibilities for new directions in research using GP and GI for partial evaluation, amorphous slicing, and product lines, with a particular focus on code transplantation. This paper accompanies the keynote given by Mark Harman at the 20th Working Conference on Reverse Engineering (WCRE 2013).

I. Introduction

The term 'search based software engineering' was introduced in 2001 [42] to capture the (then emerging) interest in the use of computational search as a means of optimising software engineering problems. The motivation was that search based optimisation was ideal for the multiple conflicting and competing objectives with which software engineers routinely contend. The algorithms used to conduct search based optimisation are also known to perform well in the presence of partial, noisy and missing data. This makes them attractive tools with which to approach the complex, noisy and incomplete world in which the software engineer has to find engineering solutions.

Since 2001, SBSE has been applied to almost every aspect of software engineering activity. A more detailed survey of the entire field of SBSE can be found elsewhere [44], while a tutorial introduction is also available [46] that assumes no prior knowledge of computational search techniques. However in this paper, we focus on reverse engineering and the considerable potential for the development of new forms of Genetic Programming (GP) and Genetic Improvement (GI) to reverse engineering. Section II presents a summary of the application of SBSE to reverse engineering. Section III briefly reviews the relationship between the SBSE and RE publication venues and trends. Section IV sets out some directions for future work that form part of a 'GPiRE' research agenda; genetic programming applications for reverse engineering.

II. Applications of SBSE to Reverse Engineering

A large number of problems in reverse engineering are amenable to SBSE. In this section we highlight some of the many existing approaches, focusing on the application of SSBSE to the problems of re-modularisation, refactoring, regression testing, slicing, and concept assignment in particular.

We also discuss the general applicability of genetic programming to reverse engineering problems. Our overall conclusion is that these are vibrant and active areas of research with multiple open problems remaining to be tackled.

A. Re-Modularization

Software structure degrades making periodic re-modularisation important [8]. The search based modularisation approaches discussed in this section assume that we start with some form of module dependence graph, from which we seek to construct suitable module boundaries.

Mancoridis *et al.* were the first to address the problem of software modularisation using SBSE leading to the development of a tool called Bunch [69] for module clustering. They experimented with several search based algorithms including genetic algorithms, hill climbing and simulated annealing. Hill climbing tended to produce the best results for the single objective of improving the Module Quality (MQ) metric.

MQ was introduced in 1998 [65], and refined in subsequent papers. All versions of MQ are combinations of cohesion and coupling into a single weighted fitness function, used to guide the search. Other authors explored other ways to capture cohesion and coupling in different metrics [47]. The choice of the cohesion and coupling metric has a critical impact on the results obtained.

Search based clustering has also been applied to package coupling [1], to reduce overall package size [17] and to explore the relationship between design and code level software structure [49]. The same overall search based clustering approach can also be adapted for clustering heap allocation [22] and for reducing memory fragmentation [23].

There were many attempts to find algorithms that produced better modularisation results (in terms of cohesion, coupling and faithfulness to some 'gold standard' modularisation). Many of the earlier attempts to improve on hill climbing, simply provided further evidence to suggest that it was a simple, fast and effective algorithm for module clustering [39], [64], [68].

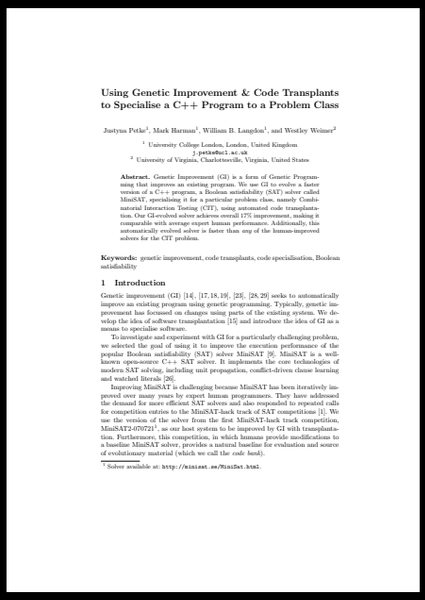
In 2003, Mahdavi *et al.* [64] introduced a multiple hill climbing approach, which performs repeated hill climbing and combines the best results found to reduce the search space size for subsequent hill climbs. Mahdavi *et al.* also used parallelisation to improve the performance of the multiple clustering, something first explored by Mitchell *et al.* [70] who used parallel computing to distribute the task of computing module clusters (but did not combine results to reduce the search space).

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany
Invited Paper

Introduced the idea of using GP for autotransplantation

Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class



Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class

Justyna Petke[1], Mark Harman[1], William B. Langdon[1], and Westley Weimer[2]
[1] University College London, London, United Kingdom
j.petke@ucl.ac.uk
[2] University of Virginia, Charlottesville, Virginia, United States

Abstract. Genetic Improvement (GI) is a form of Genetic Programming that improves an existing program. We use GI to evolve a faster version of a C++ program, a Boolean satisfiability (SAT) solver called MiniSAT, specialising it for a particular problem class, namely Combinatorial Interaction Testing (CIT), using automated code transplantation. Our GI-evolved solver achieves overall 17% improvement, making it comparable with average expert human performance. Additionally, this automatically evolved solver is faster than any of the human-improved solvers for the CIT problem.

Keywords: genetic improvement, code transplants, code specialisation, Boolean satisfiability

1 Introduction

Genetic improvement (GI) [14], [17, 18, 19], [23], [28, 29] seeks to automatically improve an existing program using genetic programming. Typically, genetic improvement has focussed on changes using parts of the existing system. We develop the idea of software transplantation [15] and introduce the idea of GI as a means to specialise software.

To investigate and experiment with GI for a particularly challenging problem, we selected the goal of using it to improve the execution performance of the popular Boolean satisfiability (SAT) solver MiniSAT [9]. MiniSAT is a well-known open-source C++ SAT solver. It implements the core technologies of modern SAT solving, including unit propagation, conflict-driven clause learning and watched literals [26].

Improving MiniSAT is challenging because MiniSAT has been iteratively improved over many years by expert human programmers. They have addressed the demand for more efficient SAT solvers and also responded to repeated calls for competition entries to the MiniSAT-hack track of SAT competitions [1]. We use the version of the solver from the first MiniSAT-hack track competition, MiniSAT2-070721[4], as our host system to be improved by GI with transplantation. Furthermore, this competition, in which humans provide modifications to a baseline MiniSAT solver, provides a natural baseline for evaluation and source of evolutionary material (which we call the code bank).

[4] Solver available at: http://minisat.se/MiniSat.html.

2013 — *Harman et al.*

2014 — *Petke et al*

Code Transplants

mming
neering

Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class

Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class

Justyna Petke[1], Mark Harman[1], William B. Langdon[1], and Westley Weimer[2]

[1] University College London, London, United Kingdom
j.petke@ucl.ac.uk
[2] University of Virginia, Charlottesville, Virginia, United States

Abstract. Genetic Improvement (GI) is a form of Genetic Programming that improves an existing program. We use GI to evolve a faster version of a C++ program, a Boolean satisfiability (SAT) solver called MiniSAT, specialising it for a particular problem class, namely Combinatorial Interaction Testing (CIT), using automated code transplantation. Our GI-evolved solver achieves overall 17% improvement, making it comparable with average expert human performance. Additionally, this automatically evolved solver is faster than any of the human-improved solvers for the CIT problem.

Keywords: genetic improvement, code transplants, code specialisation, Boolean satisfiability

1 Introduction

Genetic improvement (GI) [14], [17, 18, 19], [23], [28,29] seeks to automatically improve an existing program using genetic programming. Typically, genetic improvement has focussed on changes using parts of the existing system. We develop the idea of software transplantation [15] and introduce the idea of GI as a means to specialise software.

To investigate and experiment with GI for a particularly challenging problem, we selected the goal of using it to improve the execution performance of the popular Boolean satisfiability (SAT) solver MiniSAT [9]. MiniSAT is a well-known open-source C++ SAT solver. It implements the core technologies of modern SAT solving, including unit propagation, conflict-driven clause learning and watched literals [26].

Improving MiniSAT is challenging because MiniSAT has been iteratively improved over many years by expert human programmers. They have addressed the demand for more efficient SAT solvers and also responded to repeated calls for competition entries to the MiniSAT-hack track of SAT competitions [1]. We use the version of the solver from the first MiniSAT-hack track competition, MiniSAT2-070721[4], as our host system to be improved by GI with transplantation. Furthermore, this competition, in which humans provide modifications to a baseline MiniSAT solver, provides a natural baseline for evaluation and source of evolutionary material (which we call the *code bank*).

[4] Solver available at: `http://minisat.se/MiniSat.html`

ntroduced the a of using GP for otransplantation

Ba
G

t al.

2014 — *Petke et al.*

2014

Code Transplants

mming
neering

ntroduced the
a of using GP for
otransplantation

Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class

Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class

Justyna Petke[1], Mark Harman[1], William B. Langdon[1], and Westley Weimer[2]

[1] University College London, London, United Kingdom
j.petke@ucl.ac.uk
[2] University of Virginia, Charlottesville, Virginia, United States

Abstract. Genetic Improvement (GI) is a form of Genetic Programming that improves an existing program. We use GI to evolve a faster version of a C++ program, a Boolean satisfiability (SAT) solver called MiniSAT, specialising it for a particular problem class, namely Combinatorial Interaction Testing (CIT), using automated code transplantation. Our GI-evolved solver achieves overall 17% improvement, making it comparable with average expert human performance. Additionally, this automatically evolved solver is faster than any of the human-improved solvers for the CIT problem.

Keywords: genetic improvement, code transplants, code specialisation, Boolean satisfiability

1 Introduction

Genetic improvement (GI) [14], [17, 18, 19], [23], [28,29] seeks to automatically improve an existing program using genetic programming. Typically, genetic improvement has focussed on changes using parts of the existing system. We develop the idea of software transplantation [15] and introduce the idea of GI as a means to specialise software.

To investigate and experiment with GI for a particularly challenging problem, we selected the goal of using it to improve the execution performance of the popular Boolean satisfiability (SAT) solver MiniSAT [9]. MiniSAT is a well-known open-source C++ SAT solver. It implements the core technologies of modern SAT solving, including unit propagation, conflict-driven clause learning and watched literals [26].

Improving MiniSAT is challenging because MiniSAT has been iteratively improved over many years by expert human programmers. They have addressed the demand for more efficient SAT solvers and also responded to repeated calls for competition entries to the MiniSAT-hack track of SAT competitions [1]. We use the version of the solver from the first MiniSAT-hack track competition, MiniSAT2-070721[4], as our host system to be improved by GI with transplantation. Furthermore, this competition, in which humans provide modifications to a baseline MiniSAT solver, provides a natural baseline for evaluation and source of evolutionary material (which we call the code bank).

[4] Solver available at: http://minisat.se/MiniSat.html

Most cited paper

t al.

2014 — *Petke et al.*

2014

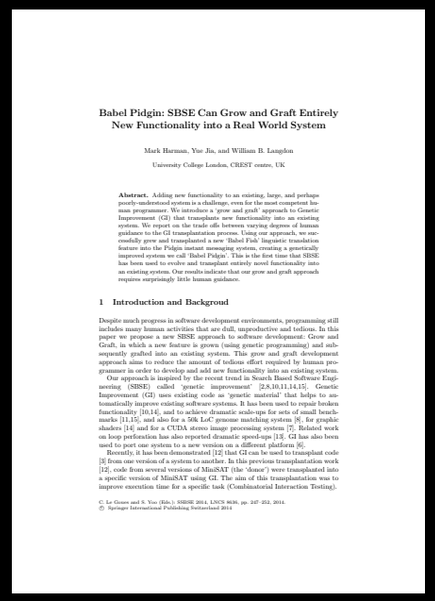
Code Transplants

ement and
pecialise a
blem Class

ies 2014

st cited
paper

Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System



Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System

Mark Harman, Yue Jia, and William B. Langdon

University College London, CREST centre, UK

Abstract. Adding new functionality to an existing, large, and perhaps poorly-understood system is a challenge, even for the most competent human programmer. We introduce a 'grow and graft' approach to Genetic Improvement (GI) that transplants new functionality into an existing system. We report on the trade offs between varying degrees of human guidance to the GI transplantation process. Using our approach, we successfully grow and transplanted a new 'Babel Fish' linguistic translation feature into the Pidgin instant messaging system, creating a genetically improved system we call 'Babel Pidgin'. This is the first time that SBSE has been used to evolve and transplant entirely novel functionality into an existing system. Our results indicate that our grow and graft approach requires surprisingly little human guidance.

1 Introduction and Background

Despite much progress in software development environments, programming still includes many human activities that are dull, unproductive and tedious. In this paper we propose a new SBSE approach to software development: Grow and Graft, in which a new feature is grown (using genetic programming) and subsequently grafted into an existing system. This grow and graft development approach aims to reduce the amount of tedious effort required by human programmer in order to develop and add new functionality into an existing system.

Our approach is inspired by the recent trend in Search Based Software Engineering (SBSE) called 'genetic improvement' [2,8,10,11,14,15]. Genetic Improvement (GI) uses existing code as 'genetic material' that helps to automatically improve existing software systems. It has been used to repair broken functionality [10,14], and to achieve dramatic scale-ups for sets of small benchmarks [11,15], and also for a 50k LoC genome matching system [8], for graphic shaders [14] and for a CUDA stereo image processing system [7]. Related work on loop perforation has also reported dramatic speed-ups [13]. GI has also been used to port one system to a new version on a different platform [6].

Recently, it has been demonstrated [12] that GI can be used to transplant code [3] from one version of a system to another. In this previous transplantation work [12], code from several versions of MiniSAT (the 'donor') were transplanted into a specific version of MiniSAT using GI. The aim of this transplantation was to improve execution time for a specific task (Combinatorial Interaction Testing).

C. Le Goues and S. Yoo (Eds.): SSBSE 2014, LNCS 8636, pp. 247–252, 2014.

Auto
Tr

2014 — *Harman et al.*

2015

Code Transplants

ement and
pecialise a
blem Class

ies 2014

st cited
paper

Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System

Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System

Mark Harman, Yue Jin, and William B. Langdon

University College London, CREST centre, UK

Abstract. Adding new functionality to an existing, large, and perhaps poorly-understood system is a challenge, even for the most competent human programmer. We introduce a 'grow and graft' approach to Genetic Improvement (GI) that transplants new functionality into an existing system. We report on the trade offs between varying degrees of human guidance to the GI transplantation process. Using our approach, we successfully grow and transplanted a new 'Babel Fish' linguistic translation feature into the Pidgin instant messaging system, creating a genetically improved system we call 'Babel Pidgin'. This is the first time that SBSE has been used to evolve and transplant entirely novel functionality into an existing system. Our results indicate that our grow and graft approach requires surprisingly little human guidance.

1 Introduction and Background

Despite much progress in software development environments, programming still includes many human activities that are dull, unproductive and tedious. In this paper we propose a new SBSE approach to software development: Grow and Graft, in which a new feature is grown (using genetic programming) and subsequently grafted into an existing system. This grow and graft development approach aims to reduce the amount of tedious effort required by human programmer in order to develop and add new functionality into an existing system.

Our approach is inspired by the recent trend in Search Based Software Engineering (SBSE) called 'genetic improvement' [2,8,10,11,14,15]. Genetic Improvement (GI) uses existing code as 'genetic material' that helps to automatically improve existing software systems. It has been used to repair broken functionality [10,14], and to achieve dramatic scale-ups for sets of small benchmarks [11,15], and also for a 50k LoC genome matching system [8], for graphic shaders [14] and for a CUDA stereo image processing system [7]. Related work on loop perforation has also reported dramatic speed-ups [13]. GI has also been used to port one system to a new version on a different platform [6].

Recently, it has been demonstrated [12] that GI can be used to transplant code [3] from one version of a system to another. In this previous transplantation work [12], code from several versions of MiniSAT (the 'donor') were transplanted into a specific version of MiniSAT using GI. The aim of this transplantation was to improve execution time for a specific task (Combinatorial Interaction Testing).

C. Le Goues and S. Yoo (Eds.): SSBSE 2014, LNCS 8636, pp. 247–252, 2014.
© Springer International Publishing Switzerland 2014

ssbse2014 BRAZIL

International Symposium on Search Based Software Engineering

August 26 – 29, 2014

Fortaleza - Brazil

Certificate of Award

BEST SBSE CHALLENGE TRACK PAPER

The paper

"Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system"

by

Mark Harman, Yue Jin and William B. Langdon

has been awarded the Best SBSE Challenge Track paper of the 6th International Symposium on Search Based Software Engineering (SSBSE'2014), held in Fortaleza, Brazil, on August 26-29, 2014.

Jerffeson Teixeira de Souza, Ph.D.

SSBSE'2014 Conference Chair

Auto
Tr

2014 — *Harman et al.*

2015

Code Transplants

el Pidgin: SBSE Can Grow and ft Entirely New Functionality into a Real World System


gin: SBSE Can Grow and Graft Entirely
nctionality into a Real World System

ark Harman, Yue Jia, and William B. Langdon

University College London, CREST centre, UK

Adding new functionality to an existing, large, and perhaps
rstood system is a challenge, even for the most competent hu-
mmer. We introduce a 'grow and graft' approach to Genetic
t (GI) that transplants new functionality into an existing
report on the trade offs between varying degrees of human
the GI transplantation process. Using our approach, we suc-
ew and transplanted a new 'Babel Fish' linguistic translation
the Pidgin instant messaging system, creating a genetically
stem we call 'Babel Pidgin'. This is the first time that SBSE
ed to evolve and transplant entirely novel functionality into
ystem. Our results indicate that our grow and graft approach
prisingly little human guidance.

ion and Background

gress in software development environments, programming still
man activities that are dull, unproductive and tedious. In this
e a new SBSE approach to software development: Grow and
a new feature is grown (using genetic programming) and sub-
into an existing system. This grow and graft development
reduce the amount of tedious effort required by human pro-
to develop and add new functionality into an existing system.
is inspired by the recent trend in Search Based Software Engi-
called 'genetic improvement' [2,8,10,11,14,15]. Genetic
) uses existing code as 'genetic material' that helps to au-
ve existing software systems. It has been used to repair broken
14], and to achieve dramatic scale-ups for sets of small bench-
l also for a 50k LoC genome matching system [8], for graphic
or a CUDA stereo image processing system [7]. Related work
n has also reported dramatic speed-ups [13]. GI has also been
system to a new version on a different platform [6].
been demonstrated [12] that GI can be used to transplant code
n of a system to another. In this previous transplantation work
veral versions of MiniSAT ('the 'donor') were transplanted into
of MiniSAT using GI. The aim of this transplantation was to
time for a specific task (Combinatorial Interaction Testing).

ce (Eds.): SSBSE 2014, LNCS 8636, pp. 247–252, 2014.
mal Publishing Switzerland 2014


ssbse2014
BRAZIL

International Symposium on Search Based Software Engineering

August 26 – 29, 2014

Fortaleza - Brazil

Certificate of Award

BEST SBSE CHALLENGE TRACK PAPER

The paper

"Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system"

by

Mark Harman, Yue Jin and William B. Langdon

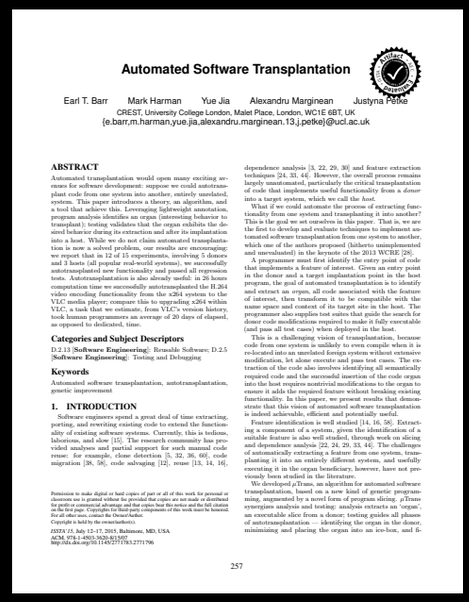
has been awarded the Best SBSE Challenge Track paper of the 6th International Symposium on Search Based Software Engineering (SSBSE'2014), held in Fortaleza, Brazil, on August 26-29, 2014.

Jerffeson Teixeira de Souza, Ph.D.

SSBSE'2014 Conference Chair

2014 — *Harman et al.*

Automated Software Transplantation


Automated Software Transplantation

Earl T. Barr Mark Harman Yue Jia Alexandru Marginean Justyna Petke

CREST, University College London, Malet Place, London, WC1E 6BT, UK
{e.barr,m.harman,yue.jia,alexandru.marginean.13,j.petke}@ucl.ac.uk

ABSTRACT

Automated transplantation would open many exciting avenues for software development: suppose we could autotransplant code from one system into another, entirely unrelated, system. This paper introduces a theory, an algorithm, and a tool that achieve this. Leveraging lightweight annotation, program analysis identifies an organ (interesting behavior to transplant); testing validates that the organ exhibits the desired behavior during its extraction and after its implantation into a host. While we do not claim automated transplantation is now a solved problem, our results are encouraging: we report that in 12 of 15 experiments, involving 5 donors and 3 hosts (all popular real-world systems), we successfully autotransplanted new functionality and passed all regression tests. Autotransplantation is also already useful: in 26 hours computation time we successfully autotransplanted the H.264 video encoding functionality from the x264 system to the VLC media player; compare this to upgrading x264 within VLC, a task that we estimate, from VLC's version history, took human programmers an average of 20 days of elapsed, as opposed to dedicated, time.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Automated software transplantation, autotransplantation, genetic improvement

1. INTRODUCTION

Software engineers spend a great deal of time extracting, porting, and rewriting existing code to extend the functionality of existing software systems. Currently, this is tedious, laborious, and slow [15]. The research community has provided analyses and partial support for such manual code reuse: for example, clone detection [5, 32, 36, 66], code migration [38, 58], code salvaging [12], reuse [13, 14, 16], dependence analysis [3, 22, 29, 30] and feature extraction techniques [24, 33, 44]. However, the overall process remains largely unautomated, particularly the critical transplantation of code that implements useful functionality from a *donor* into a target system, which we call the *host*.

What if we could automate the process of extracting functionality from one system and transplanting it into another? This is the goal we set ourselves in this paper. That is, we are the first to develop and evaluate techniques to implement automated software transplantation from one system to another, which one of the authors proposed (hitherto unimplemented and unevaluated) in the keynote of the 2013 WCRE [28].

A programmer must first identify the entry point of code that implements a feature of interest. Given an entry point in the donor and a target implantation point in the host program, the goal of automated transplantation is to identify and extract an *organ*, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. The programmer also supplies test suites that guide the search for donor code modifications required to make it fully executable (and pass all test cases) when deployed in the host.

This is a challenging vision of transplantation, because code from one system is unlikely to even compile when it is re-located into an unrelated foreign system without extensive modification, let alone execute and pass test cases. The extraction of the code also involves identifying all semantically required code and the successful insertion of the code organ into the host requires nontrivial modifications to the organ to ensure it adds the required feature without breaking existing functionality. In this paper, we present results that demonstrate that this vision of automated software transplantation is indeed achievable, efficient and potentially useful.

Feature identification is well studied [14, 16, 58]. Extracting a component of a system, given the identification of a suitable feature is also well studied, through work on slicing and dependence analysis [22, 24, 29, 33, 44]. The challenges of automatically extracting a feature from one system, transplanting it into an entirely different system, and usefully executing it in the organ beneficiary, however, have not previously been studied in the literature.

We developed µTrans, an algorithm for automated software transplantation, based on a new kind of genetic programming, augmented by a novel form of program slicing. µTrans synergizes analysis and testing: analysis extracts an 'organ', an executable slice from a donor; testing guides all phases of autotransplantation — identifying the organ in the donor, minimizing and placing the organ into an ice-box, and fi-

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771796

257

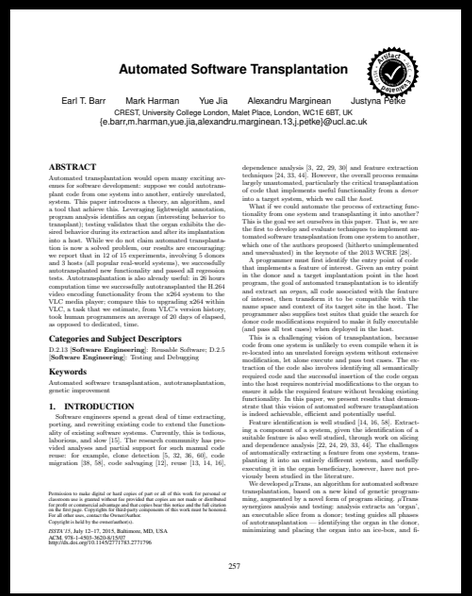
2015 — *Barr et al.*

Auto by H Acro

2015

Code Transplants

Automated Software Transplantation



Automated Software Transplantation

Earl T. Barr Mark Harman Yue Jia Alexandru Marginean Justyna Petke
CREST, University College London, Malet Place, London, WC1E 6BT, UK
{e.barr,m.harman,yue.jia,alexandru.marginean.13,j.petke}@ucl.ac.uk

ABSTRACT

Automated transplantation would open many exciting avenues for software development: suppose we could autotransplant code from one system into another, entirely unrelated, system. This paper introduces a theory, an algorithm, and a tool that achieve this. Leveraging lightweight annotation, program analysis identifies an organ (interesting behavior to transplant); testing validates that the organ exhibits the desired behavior during its extraction and after its implantation into a host. While we do not claim automated transplantation is now a solved problem, our results are encouraging: we report that in 12 of 15 experiments, involving 5 donors and 3 hosts (all popular real-world systems), we successfully autotransplanted new functionality and passed all regression tests. Autotransplantation is also already useful: in 26 hours computation time we successfully autotransplant the H.264 video encoding functionality from the x264 system to the VLC media player; compare this to upgrading x264 within VLC, a task that we estimate, from VLC's version history, took human programmers an average of 20 days of elapsed, as opposed to dedicated, time.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Automated software transplantation, autotransplantation, genetic improvement

1. INTRODUCTION

Software engineers spend a great deal of time extracting, porting, and rewriting existing code to extend the functionality of existing software systems. Currently, this is tedious, laborious, and slow [15]. The research community has provided analyses and partial support for such manual code reuse: for example, clone detection [5, 32, 36, 60], code migration [38, 58], code salvaging [12], reuse [13, 14, 16], dependence analysis [3, 22, 29, 30] and feature extraction techniques [24, 33, 44]. However, the overall process remains largely unautomated, particularly the critical transplantation of code that implements useful functionality from a *donor* into a target system, which we call the *host*.

What if we could automate the process of extracting functionality from one system and transplanting it into another? This is the goal we set ourselves in this paper. That is, we are the first to develop and evaluate techniques to implement automated software transplantation from one system to another, which one of the authors proposed (hitherto unimplemented and unevaluated) in the keynote of the 2013 WCRE [28].

A programmer must first identify the entry point of code that implements a feature of interest. Given an entry point in the donor and a target implantation point in the host program, the goal of automated transplantation is to identify and extract an *organ*, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. The programmer also supplies test suites that guide the search for donor code modifications required to make it fully executable (and pass all test cases) when deployed in the host.

This is a challenging vision of transplantation, because code from one system is unlikely to even compile when it is re-located into an unrelated foreign system without extensive modification, let alone execute and pass test cases. The extraction of the code also involves identifying all semantically required code and the successful insertion of the code organ into the host requires nontrivial modifications to the organ to ensure it adds the required feature without breaking existing functionality. In this paper, we present results that demonstrate that this vision of automated software transplantation is indeed achievable, efficient and potentially useful.

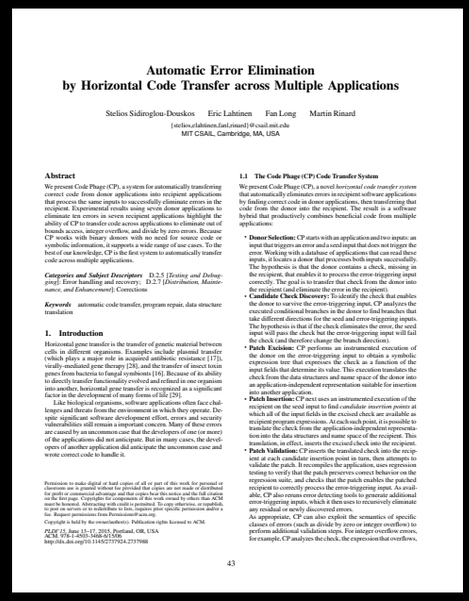
Feature identification is well studied [14, 16, 58]. Extracting a component of a system, given the identification of a suitable feature is also well studied, through work on slicing and dependence analysis [22, 24, 29, 33, 44]. The challenges of automatically extracting a feature from one system, transplanting it into an entirely different system, and usefully executing it in the organ beneficiary, however, have not previously been studied in the literature.

We developed *µ*Trans, an algorithm for automated software transplantation, based on a new kind of genetic programming, augmented by a novel form of program slicing. *µ*Trans synergizes analysis and testing: analysis extracts an 'organ', an executable slice from a donor; testing guides all phases of autotransplantation — identifying the organ in the donor, minimizing and placing the organ into an ice-box, and fi-

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771796

2015 — *Barr et al.*

Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications



Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications

Stelios Sidiroglou-Douskos Eric Lahtinen Fan Long Martin Rinard
{stelios,elahtinen,fanl,rinard}@csail.mit.edu
MIT CSAIL, Cambridge, MA, USA

Abstract

We present Code Phage (CP), a system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Experimental results using seven donor applications to eliminate ten errors in seven recipient applications highlight the ability of CP to transfer code across applications to eliminate out of bounds access, integer overflow, and divide by zero errors. Because CP works with binary donors with no need for source code or symbolic information, it supports a wide range of use cases. To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Corrections

Keywords automatic code transfer, program repair, data structure translation

1. Introduction

Horizontal gene transfer is the transfer of genetic material between cells in individual organisms. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [17]), virally-mediated gene therapy [28], and the transfer of insect toxin genes from bacteria to fungal symbionts [16]. Because of its ability to directly transfer functionality evolved and refined in one organism into another, horizontal gene transfer is recognized as a significant factor in the development of many forms of life [29].

Like biological organisms, software applications often face challenges and threats from the environment in which they operate. Despite significant software development effort, errors and security vulnerabilities still remain a important concern. Many of these errors are caused by an uncommon case that the developers of one (or more) of the applications did not anticipate. But in many cases, the developers of another application did anticipate the uncommon case and wrote correct code to handle it.

PLDI'15, June 13–17, 2015, Portland, OR, USA
ACM 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2737988

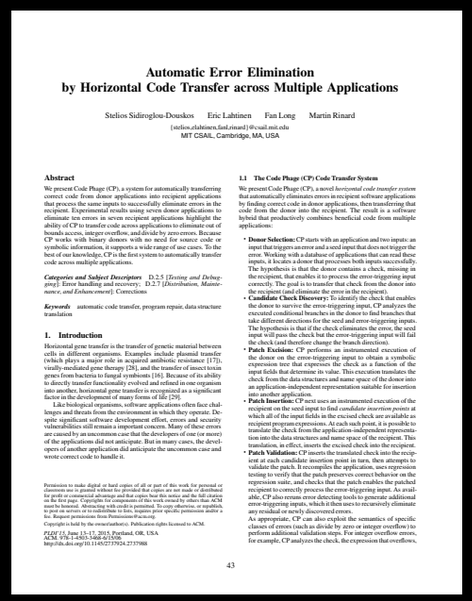
2015 — *Sidiroglou-Douskos et al.*

A

20

Code Transplants

omatic Error Elimination
lorizontal Code Transfer
oss Multiple Applications



Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications

Stelios Sidiroglou-Douskos Eric Lahtinen Fan Long Martin Rinard
{stelios,elahtinen,fanl,rinard}@csail.mit.edu
MIT CSAIL, Cambridge, MA, USA

Abstract

We present Code Phage (CP), a system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Experimental results using seven donor applications to eliminate ten errors in seven recipient applications highlight the ability of CP to transfer code across applications to eliminate out of bounds access, integer overflow, and divide by zero errors. Because CP works with binary donors with no need for source code or symbolic information, it supports a wide range of use cases. To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Corrections

Keywords automatic code transfer, program repair, data structure translation

1. Introduction

Horizontal gene transfer is the transfer of genetic material between cells in different organisms. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [17]), virally-mediated gene therapy [28], and the transfer of insect toxin genes from bacteria to fungal symbionts [16]. Because of its ability to directly transfer functionality evolved and refined in one organism into another, horizontal gene transfer is recognized as a significant factor in the development of many forms of life [29].

Like biological organisms, software applications often face challenges and threats from the environment in which they operate. Despite significant software development effort, errors and security vulnerabilities still remain a important concern. Many of these errors are caused by an uncommon case that the developers of one (or more) of the applications did not anticipate. But in many cases, the developers of another application did anticipate the uncommon case and wrote correct code to handle it.

PLDI'15, June 13-17, 2015, Portland, OR, USA
ACM. 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2737988

1.1 The Code Phage (CP) Code Transfer System

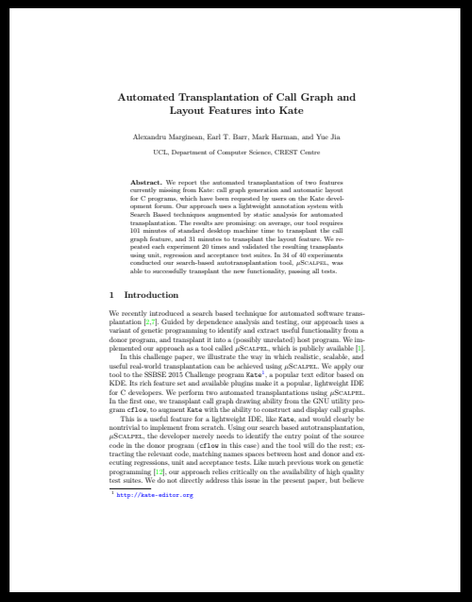
We present Code Phage (CP), a novel *horizontal code transfer system* that automatically eliminates errors in recipient software applications by finding correct code in donor applications, then transferring that code from the donor into the recipient. The result is a software hybrid that productively combines beneficial code from multiple applications:

- **Donor Selection:** CP starts with an application and two inputs: an input that triggers an error and a seed input that does not trigger the error. Working with a database of applications that can read these inputs, it locates a donor that processes both inputs successfully. The hypothesis is that the donor contains a check, missing in the recipient, that enables it to process the error-triggering input correctly. The goal is to transfer that check from the donor into the recipient (and eliminate the error in the recipient).
- **Candidate Check Discovery:** To identify the check that enables the donor to survive the error-triggering input, CP analyzes the executed conditional branches in the donor to find branches that take different directions for the seed and error-triggering inputs. The hypothesis is that if the check eliminates the error, the seed input will pass the check but the error-triggering input will fail the check (and therefore change the branch direction).
- **Patch Excision:** CP performs an instrumented execution of the donor on the error-triggering input to obtain a symbolic expression tree that expresses the check as a function of the input fields that determine its value. This execution translates the check from the data structures and name space of the donor into an application-independent representation suitable for insertion into another application.
- **Patch Insertion:** CP next uses an instrumented execution of the recipient on the seed input to find *candidate insertion points* at which all of the input fields in the excised check are available as recipient program expressions. At each such point, it is possible to translate the check from the application-independent representation into the data structures and name space of the recipient. This translation, in effect, inserts the excised check into the recipient.
- **Patch Validation:** CP inserts the translated check into the recipient at each candidate insertion point in turn, then attempts to validate the patch. It recompiles the application, uses regression testing to verify that the patch preserves correct behavior on the regression suite, and checks that the patch enables the patched recipient to correctly process the error-triggering input. As available, CP also reruns error detecting tools to generate additional error-triggering inputs, which it then uses to recursively eliminate any residual or newly discovered errors.

As appropriate, CP can also exploit the semantics of specific classes of errors (such as divide by zero or integer overflow) to perform additional validation steps. For integer overflow errors, for example, CP analyzes the check, the expression that overflows,

5 — *Sidiroglou-Douskos et al.*

Automated transplantation of call graph and layout features into Kate



Automated Transplantation of Call Graph and Layout Features into Kate

Alexandru Marginean, Earl T. Barr, Mark Harman, and Yue Jia
UCL, Department of Computer Science, CREST Centre

Abstract. We report the automated transplantation of two features currently missing from Kate: call graph generation and automatic layout for C programs, which have been requested by users on the Kate development forum. Our approach uses a lightweight annotation system with Search Based techniques augmented by static analysis for automated transplantation. The results are promising: on average, our tool requires 101 minutes of standard desktop machine time to transplant the call graph feature, and 31 minutes to transplant the layout feature. We repeated each experiment 20 times and validated the resulting transplants using unit, regression and acceptance test suites. In 34 of 40 experiments conducted our search-based autotransplantation tool, μSCALPEL, was able to successfully transplant the new functionality, passing all tests.

1 Introduction

We recently introduced a search based technique for automated software transplantation [2,7]. Guided by dependence analysis and testing, our approach uses a variant of genetic programming to identify and extract useful functionality from a donor program, and transplant it into a (possibly unrelated) host program. We implemented our approach as a tool called μSCALPEL, which is publicly available [1].

In this challenge paper, we illustrate the way in which realistic, scalable, and useful real-world transplantation can be achieved using μSCALPEL. We apply our tool to the SSBSE 2015 Challenge program `Kate`[*], a popular text editor based on KDE. Its rich feature set and available plugins make it a popular, lightweight IDE for C developers. We perform two automated transplantations using μSCALPEL. In the first one, we transplant call graph drawing ability from the GNU utility program `cflow`, to augment `Kate` with the ability to construct and display call graphs.

This is a useful feature for a lightweight IDE, like `Kate`, and would clearly be nontrivial to implement from scratch. Using our search based autotransplantation, μSCALPEL, the developer merely needs to identify the entry point of the source code in the donor program (`cflow` in this case) and the tool will do the rest; extracting the relevant code, matching names spaces between host and donor and executing regressions, unit and acceptance tests. Like much previous work on genetic programming [12], our approach relies critically on the availability of high quality test suites. We do not directly address this issue in the present paper, but believe

[*] `http://kate-editor.org`

2015 — *Marginean et al.*

Code Transplants

omatic Error Elimination
lorizontal Code Transfer
oss Multiple Applications

Automated transplantation
of call graph and layout
features into Kate

Automatic Error Elimination
by Horizontal Code Transfer across Multiple Applications

Stelios Sidiroglou-Douskos Eric Lahtinen Fan Long Martin Rinard
{stelios,elahtinen,fanl,rinard}@csail.mit.edu
MIT CSAIL, Cambridge, MA, USA

Abstract

We present Code Phage (CP), a system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Experimental results using seven donor applications to eliminate ten errors in seven recipient applications highlight the ability of CP to transfer code across applications to eliminate out of bounds access, integer overflow, and divide by zero errors. Because CP works with binary donors with no need for source code or symbolic information, it supports a wide range of use cases. To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Corrections

Keywords automatic code transfer, program repair, data structure translation

1. Introduction

Horizontal gene transfer is the transfer of genetic material between cells in different organisms. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [17]), virally-mediated gene therapy [28], and the transfer of insect toxin genes from bacteria to fungal symbionts [16]. Because of its ability to directly transfer functionality evolved and refined in one organism into another, horizontal gene transfer is recognized as a significant factor in the development of many forms of life [29].

Like biological organisms, software applications often face challenges and threats from the environment in which they operate. Despite significant software development effort, errors and security vulnerabilities still remain a important concern. Many of these errors are caused by an uncommon case that the developers of one (or more) of the applications did not anticipate. But in many cases, the developers of another application did anticipate the uncommon case and wrote correct code to handle it.

PLDI'15, June 13-17, 2015, Portland, OR, USA
ACM. 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2737988

1.1 The Code Phage (CP) Code Transfer System

We present Code Phage (CP), a novel *horizontal code transfer system* that automatically eliminates errors in recipient software applications by finding correct code in donor applications, then transferring that code from the donor into the recipient. The result is a software hybrid that productively combines beneficial code from multiple applications:

- **Donor Selection:** CP starts with an application and two inputs: an input that triggers an error and a seed input that does not trigger the error. Working with a database of applications that can read these inputs, it locates a donor that processes both inputs successfully. The hypothesis is that the donor contains a check, missing in the recipient, that enables it to process the error-triggering input correctly. The goal is to transfer that check from the donor into the recipient (and eliminate the error in the recipient).
- **Candidate Check Discovery:** To identify the check that enables the donor to survive the error-triggering input, CP analyzes the executed conditional branches in the donor to find branches that take different directions for the seed and error-triggering inputs. The hypothesis is that if the check eliminates the error, the seed input will pass the check but the error-triggering input will fail the check (and therefore change the branch direction).
- **Patch Excision:** CP performs an instrumented execution of the donor on the error-triggering input to obtain a symbolic expression tree that expresses the check as a function of the input fields that determine its value. This execution translates the check from the data structures and name space of the donor into an application-independent representation suitable for insertion into another application.
- **Patch Insertion:** CP next uses an instrumented execution of the recipient on the seed input to find *candidate insertion points* at which all of the input fields in the excised check are available as recipient program expressions. At each such point, it is possible to translate the check from the application-independent representation into the data structures and name space of the recipient. This translation, in effect, inserts the excised check into the recipient.
- **Patch Validation:** CP inserts the translated check into the recipient at each candidate insertion point in turn, then attempts to validate the patch. It recompiles the application, uses regression testing to verify that the patch preserves correct behavior on the regression suite, and checks that the patch enables the patched recipient to correctly process the error-triggering input. As available, CP also reruns error detecting tools to generate additional error-triggering inputs, which it then uses to recursively eliminate any residual or newly discovered errors.

As appropriate, CP can also exploit the semantics of specific classes of errors (such as divide by zero or integer overflow) to perform additional validation steps. For integer overflow errors, for example, CP analyzes the check, the expression that overflows,

CREST published the bulk of transplantation papers

Interest from other groups

In the first one, we transplant call graph drawing ability from the GNU utility program `cflow`, to augment `Kate` with the ability to construct and display call graphs.

This is a useful feature for a lightweight IDE, like `Kate`, and would clearly be nontrivial to implement from scratch. Using our search based autotransplantation, JSCALPEL, the developer merely needs to identify the entry point of the source code in the donor program (`cflow` in this case) and the tool will do the rest; extracting the relevant code, matching names spaces between host and donor and executing regressions, unit and acceptance tests. Like much previous work on genetic programming [1-3], our approach relies critically on the availability of high quality test suites. We do not directly address this issue in the present paper, but believe

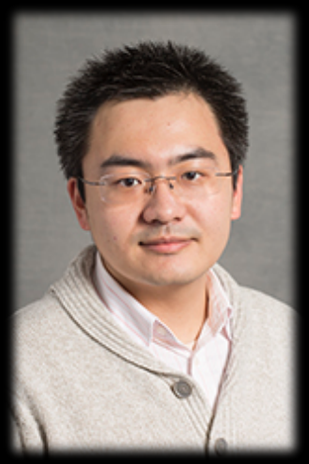
[1] http://kate-editor.org

5 — *Sidiroglou-Douskos et al.*

2015 — *Marginean et al.*

Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system

Mark Harman, Yue Jia, William B. Langdon



Thank you!

Yue Jia

Babel Pidgin

Can we "grow" a software component and "graft" it to existing system automatically?

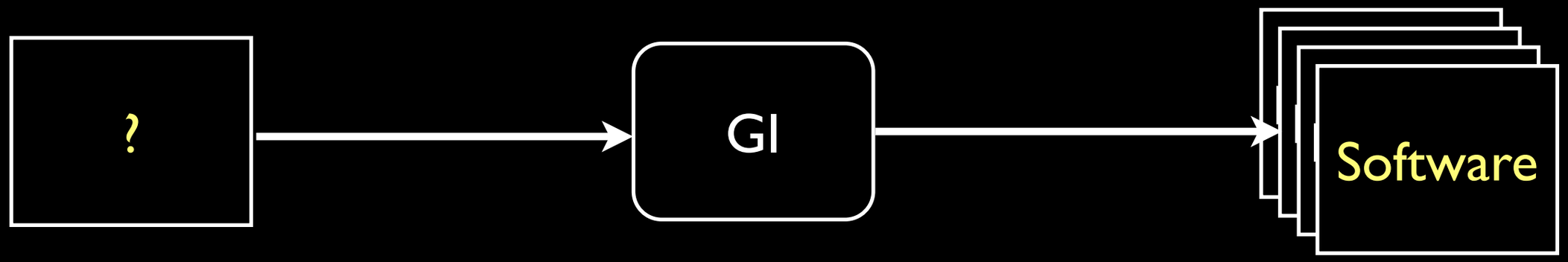
Babel Pidgin

Can we "grow" a software component and "graft" it to existing system automatically?

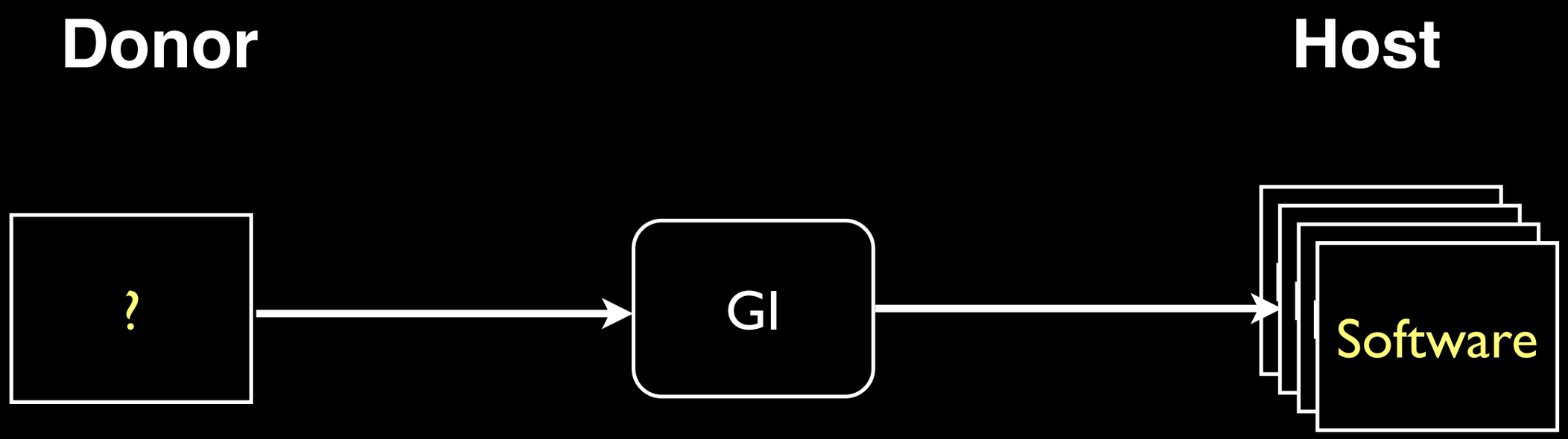
Reduce the amount of tedious effort required by human programmer in order to develop and add new functionality into an existing system.

Code Transplants

Donor **Host**

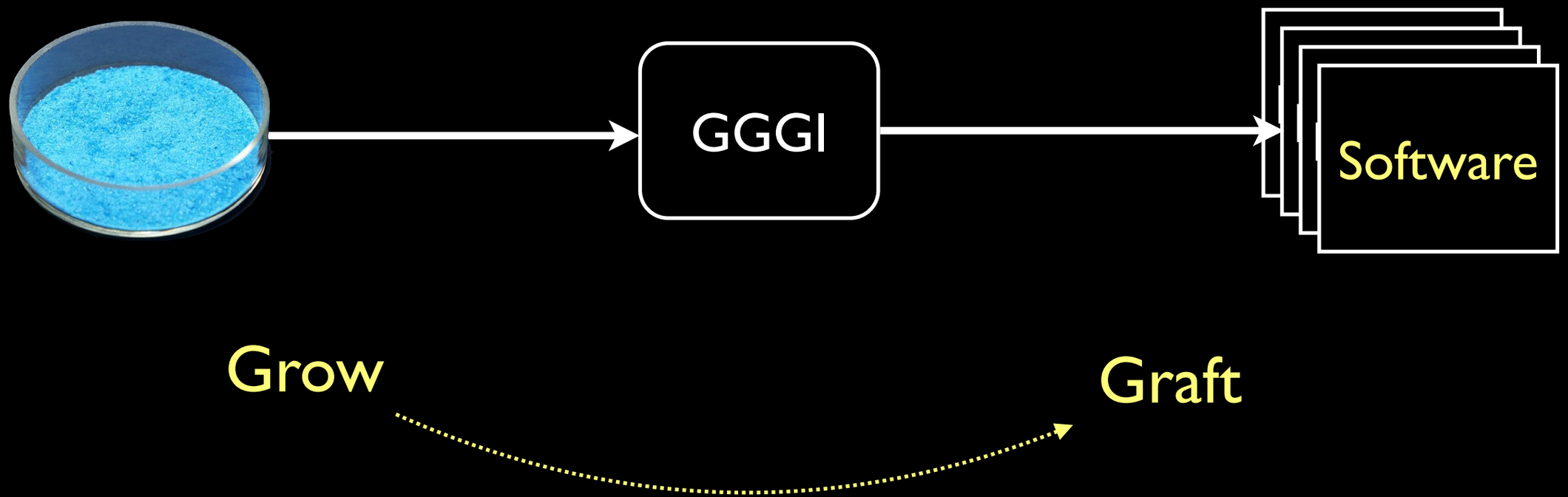


Code Transplants



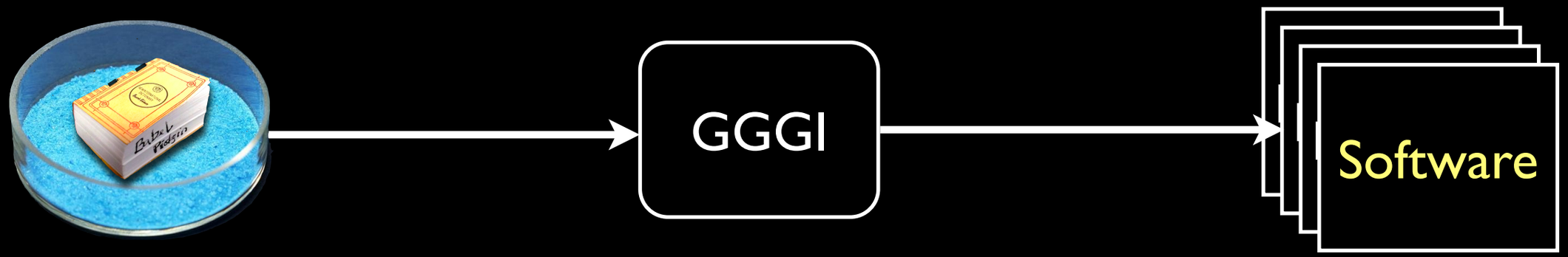
What if we want to transplant a new functionality, which we could not find it in any existing donors?

GGGI Grow and Graft for Genetic Improvements



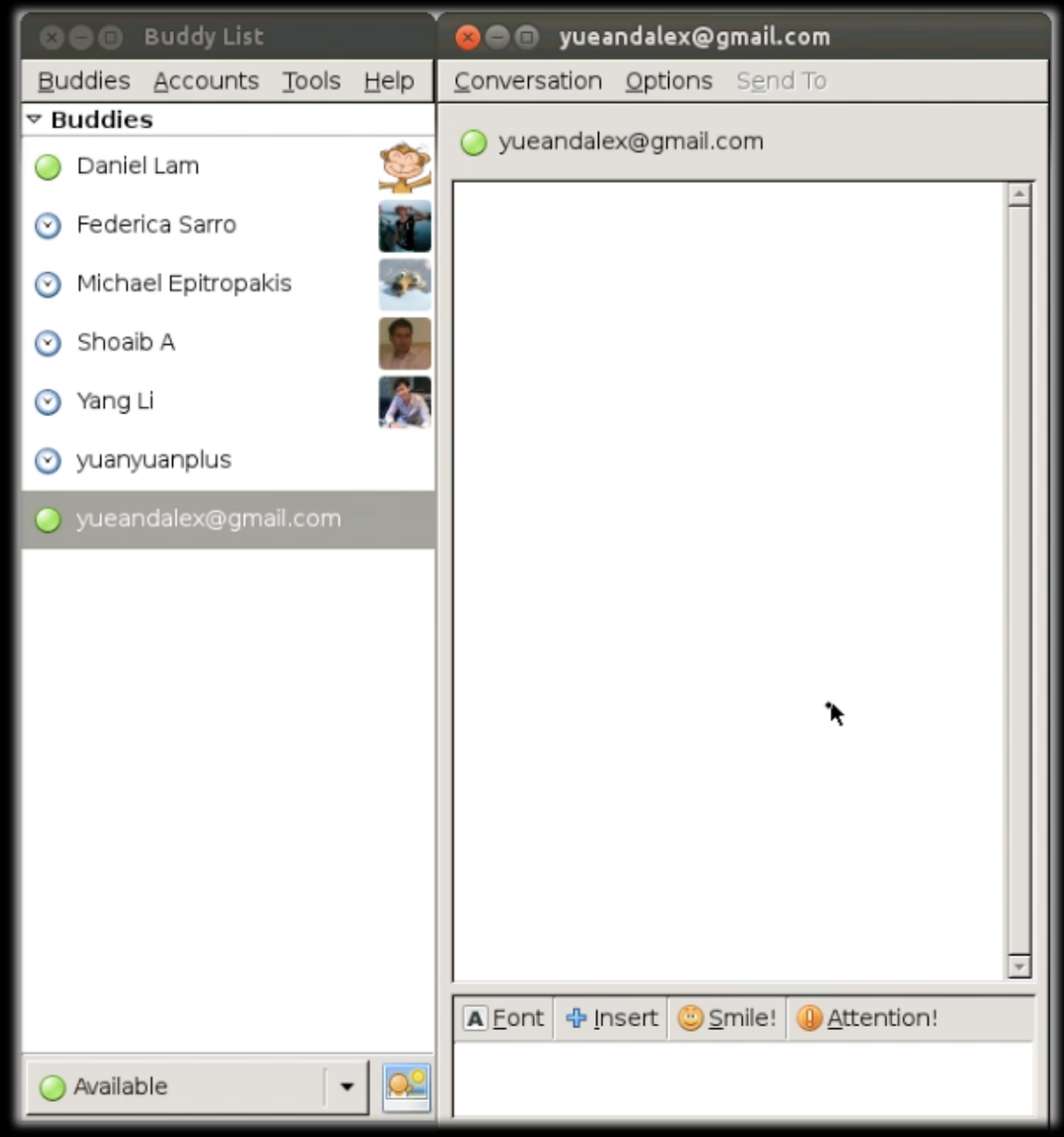
Grow code for new functionality, rather than to improve existing non-functional properties of the system.

GGGI Grow and Graft for Genetic Improvements



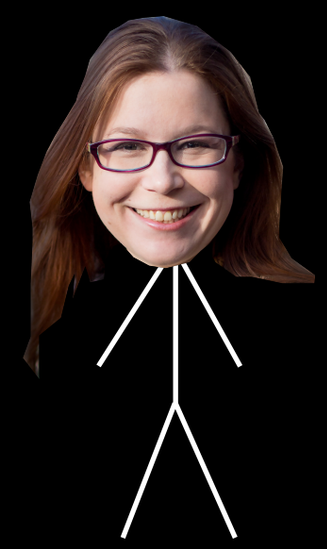
Grow code for new functionality, rather than to improve existing non-functional properties of the system.

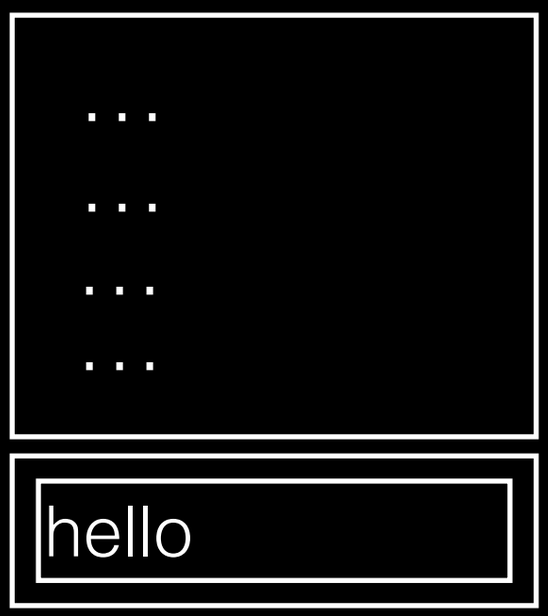
Pidgin

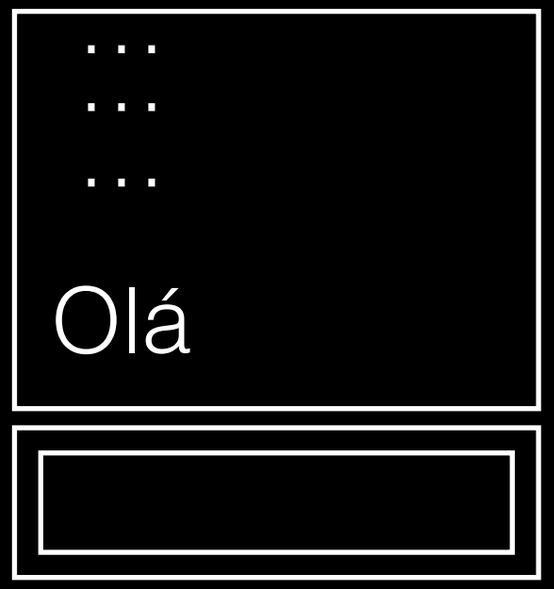


Version 2.10.9
C/C++ Instant message client used by million users worldwide

Babel Pidgin







Awards

Awards

ssbse2014
BRAZIL

International Symposium on Search Based Software Engineering
August 26 – 29, 2014
Fortaleza - Brazil

Certificate of Award

BEST SBSE CHALLENGE TRACK PAPER

The paper

"Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system"

by

Mark Harman, Yue Jin and William B. Langdon

has been awarded the Best SBSE Challenge Track paper of the 6th International Symposium on Search Based Software Engineering (SSBSE 2014), held in Fortaleza, Brazil, on August 26-29, 2014.

Jerffeson Teixeira de Souza, Ph.D.
SSBSE 2014 Conference Chair

Awards


ssbse2014
BRAZIL

International Symposium on Search Based Software Engineering
August 26 – 29, 2014
Fortaleza - Brazil

Certificate of Award

BEST SBSE CHALLENGE TRACK PAPER

The paper

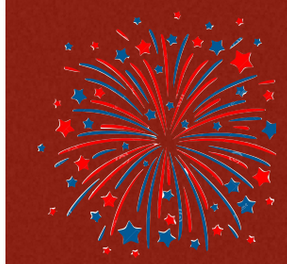
"Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system"

by

Mark Harman, Yue Jin and William B. Langdon

has been awarded the Best SBSE Challenge Track paper of the 6th International Symposium on Search Based Software Engineering (SSBSE 2014), held in Fortaleza, Brazil, on August 26-29, 2014.

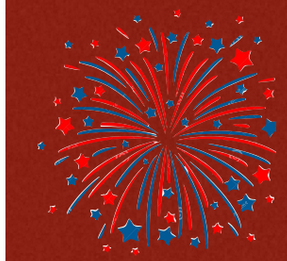
Jerffeson Teixeira de Souza, Ph.D.
SSBSE 2014 Conference Chair



Best SBSE Challenge Paper

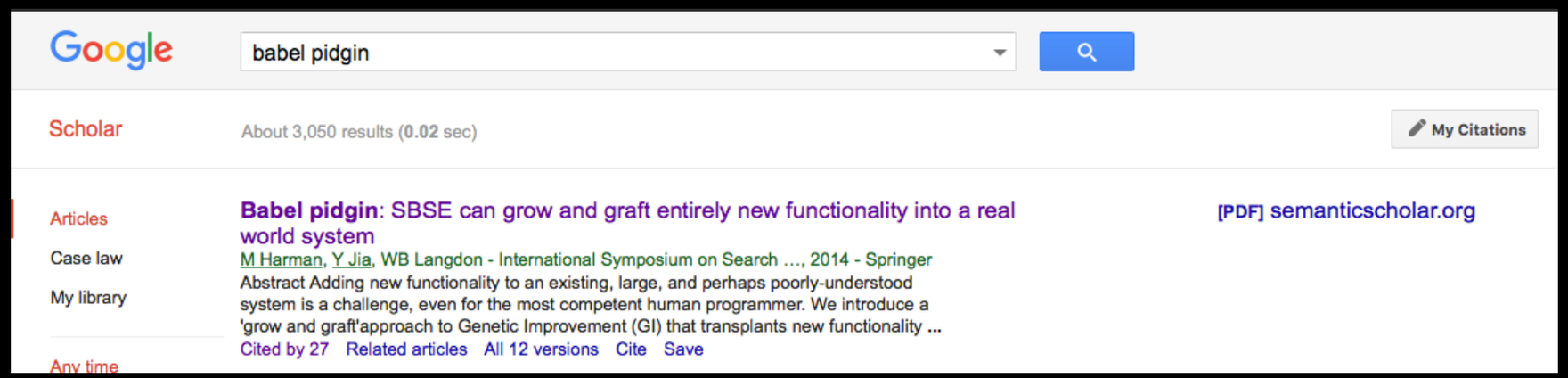
400 USD Amazon gift card plus 1,000 USD cash prize

Awards

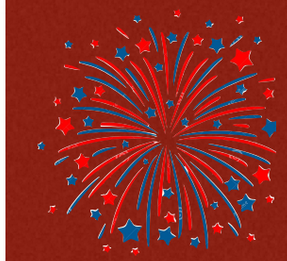


Best SBSE Challenge Paper

400 USD Amazon gift card plus 1,000 USD cash prize

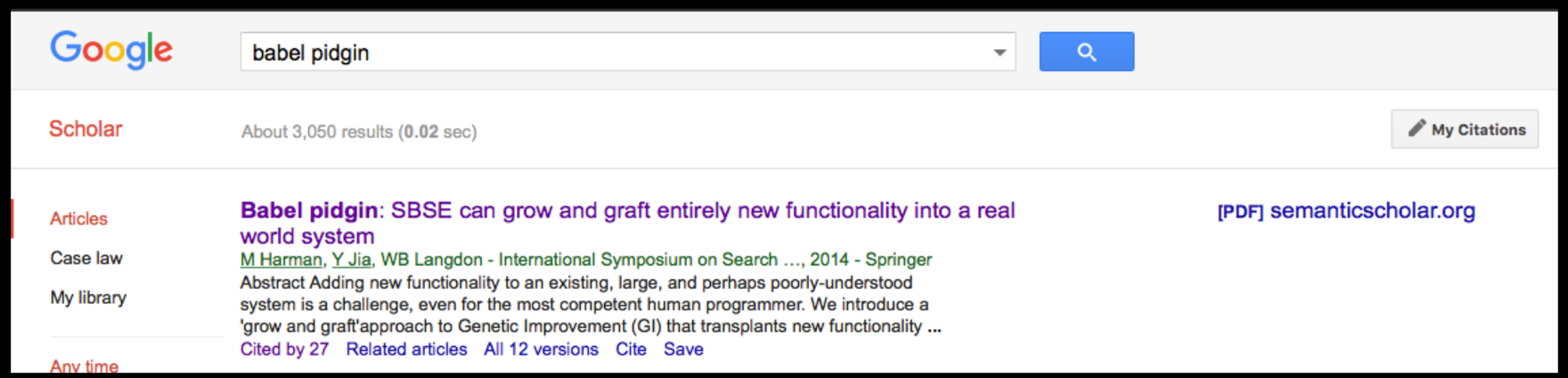


Awards



Best SBSE Challenge Paper

400 USD Amazon gift card plus 1,000 USD cash prize



EPSRC
Pioneering research and skills

£581,560 Grant
28 October 2015 — 27 October 2019
Microsoft, Visa Europe

Automated Software Transplantation

Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, Justyna Petke

Automated Transplantation of Call Graph and Layout Features into Kate

Alexandru Marginean, Earl T. Barr, Mark Harman, Yue Jia

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

```
Stream *ds = decodeFile(vF);
encodeStream(ds, out);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
Stream *ds = decodeFile(vF);
encodeStream(ds, out);
```

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
Stream *ds = decodeFile(vF);
encodeStream(ds, out);
```

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
Stream *ds = decodeFile(vF);
encodeStream(ds, out);
```

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

```
char *vF;
vF = getFile();
initCodec(vF);
```

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
char *vF;
vF = getFile();
initCodec(vF);
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
iF = getFile();
initCodec(iF);
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
iF = getFile();
initCodec(iF);
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
iF = getFile();
initCodec(iF);
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
~~iF = getFile();~~
initCodec(iF);
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Transplantation

Donor

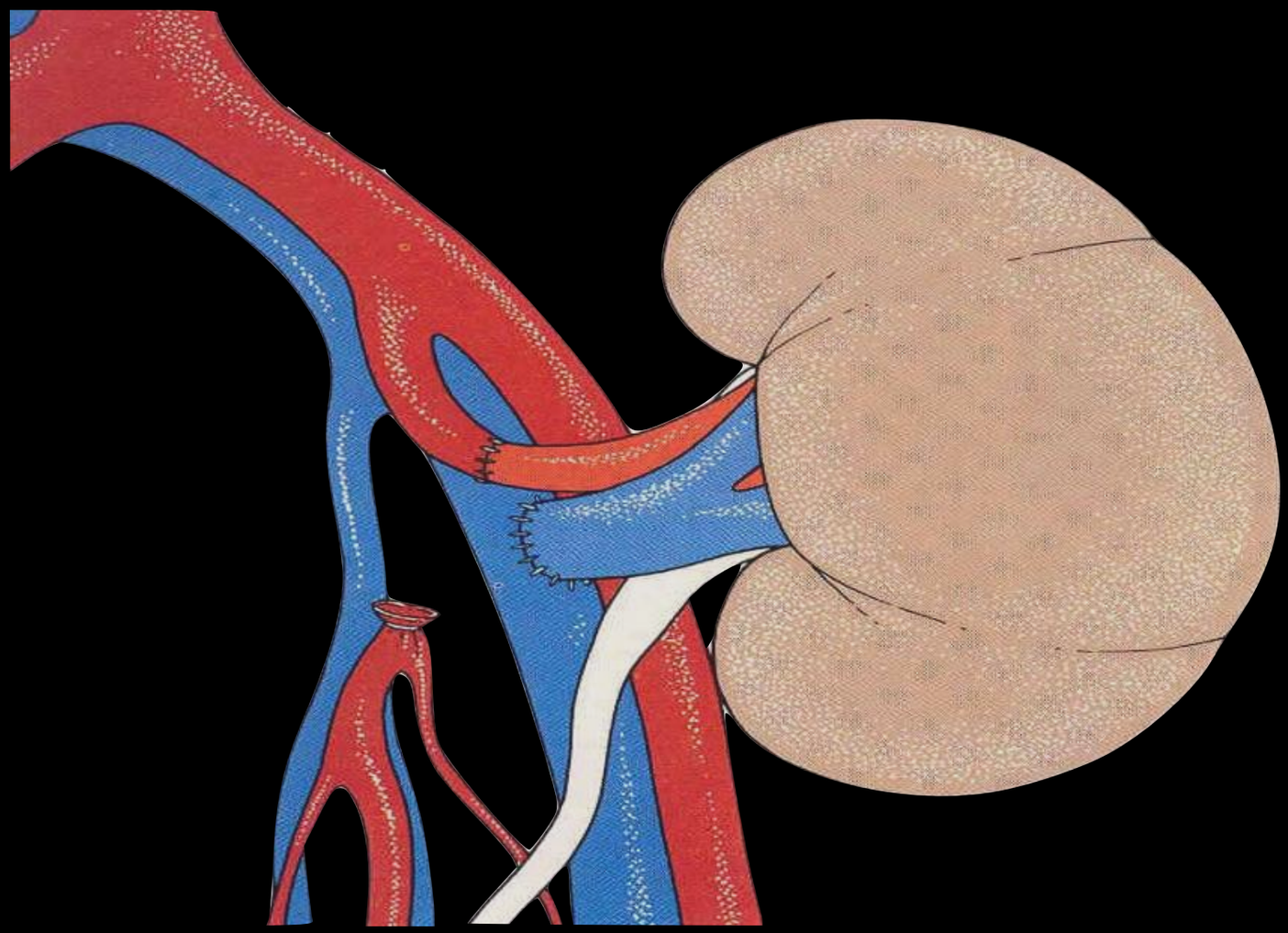
Host

```
char * iF = getInputFile();
char * oF = getOutputFile();
```

```
if = getFile();
initCodec(iF);
```

```
Stream *ds = decodeFile(iF);
encodeStream(ds, oF);
```

Human Organ Transplantation



Autotransplantation

Donor

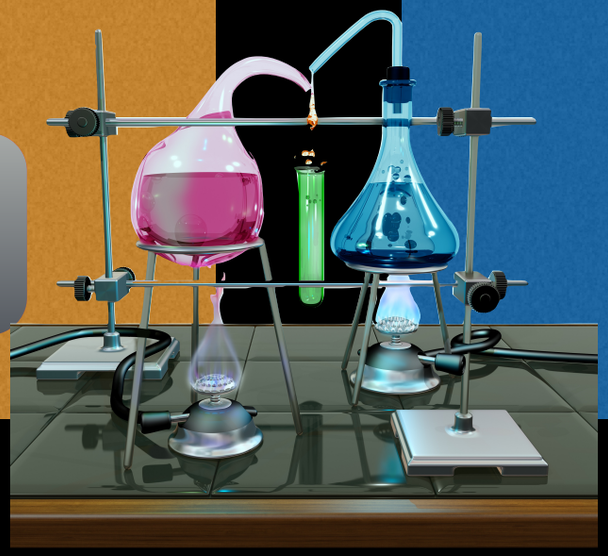
ENTRY

Veins

Organ

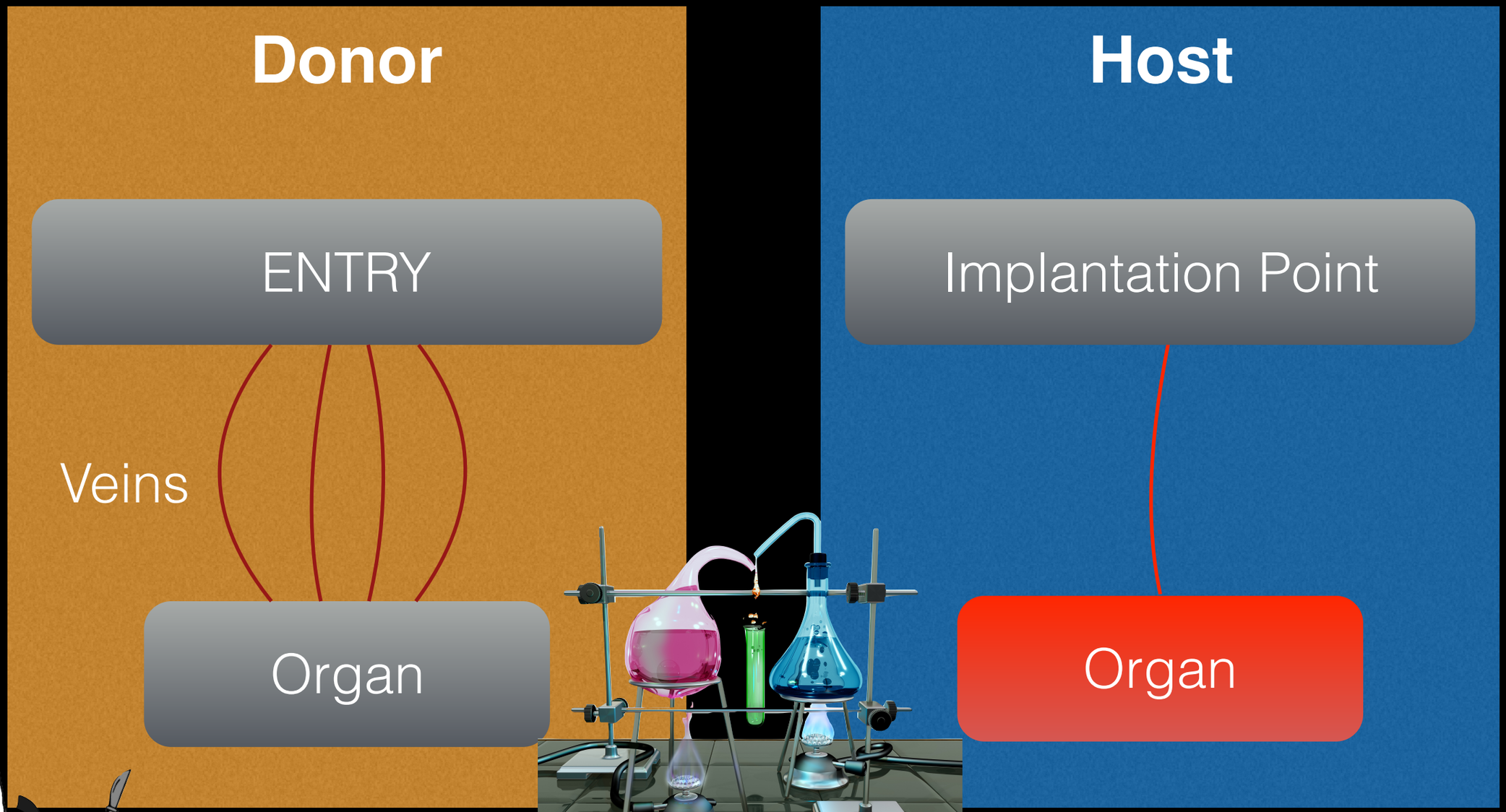
Host

Implantation Point



Organ Test Suite

Autotransplantation



Organ Test Suite

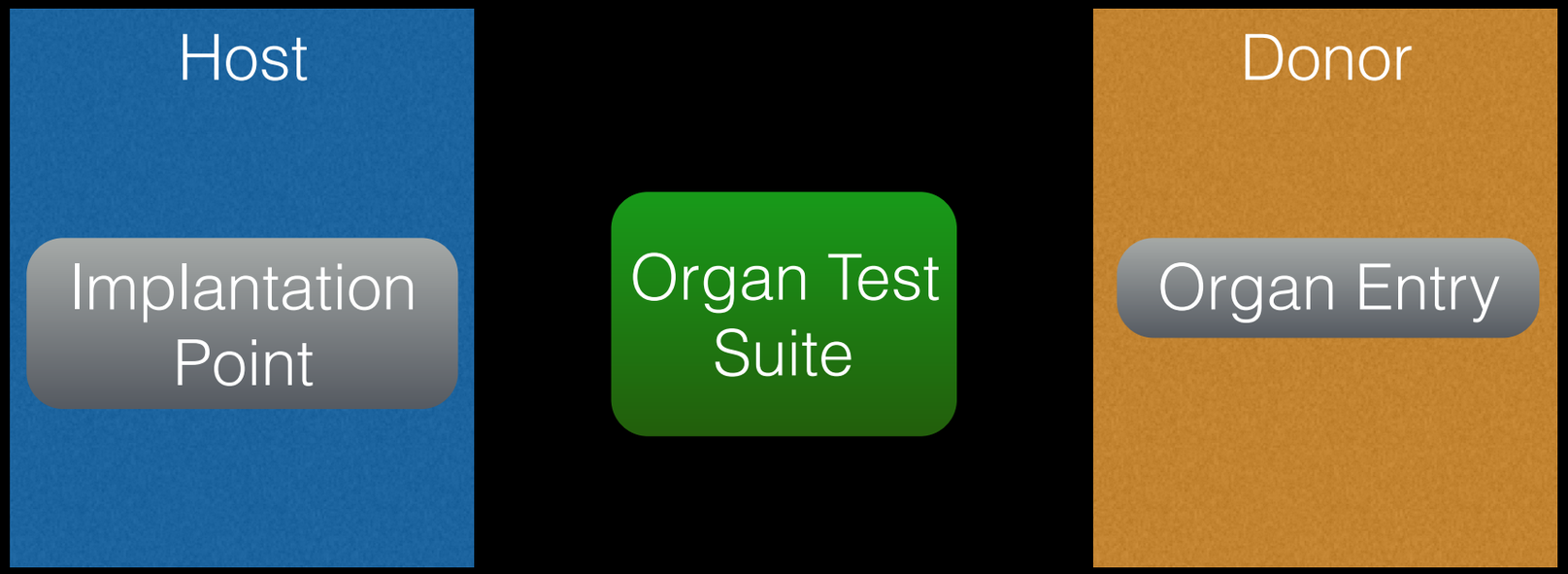
Experimental Corpus

Subjects	Type	Size KLOC	Reg. Tests
Empirical Study			
Idct	Donor	2.3	-
Mytar	Donor	0.4	-
Cflow	Donor	25	-
Webserver	Donor	1.7	-
TuxCrypt	Donor	2.7	-
Pidgin	Host	363	88
Cflow	Host	25	21
SoX	Host	43	157
Case Studies			
VLC	Host	422	27
Kate	Host	50	238
x264	Donor	63	-
Cflow	Donor	22	-
Indent	Donor	26	-

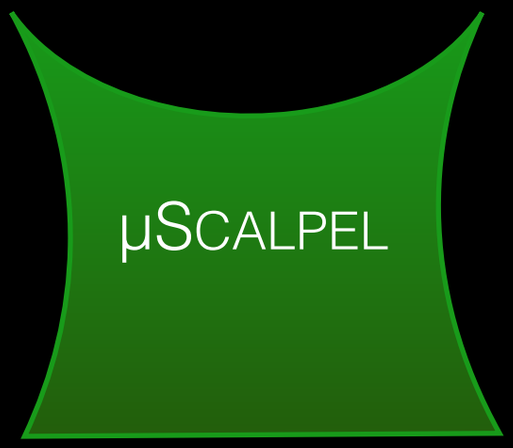
Aggregate Statistics (Size)

Minimum	**0.4k**
Maximum	**422k**
Donor (Average)	**16k**
Host (Average)	**213k**

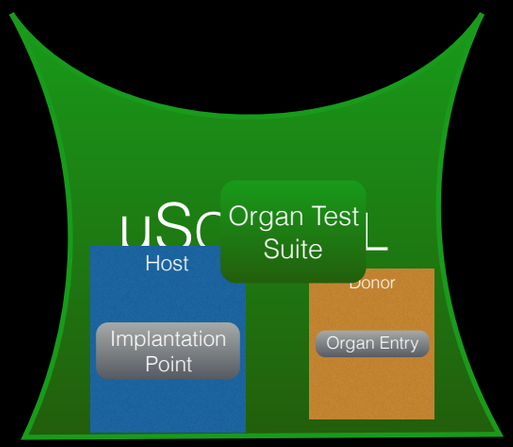
Experimental Methodology and Setup



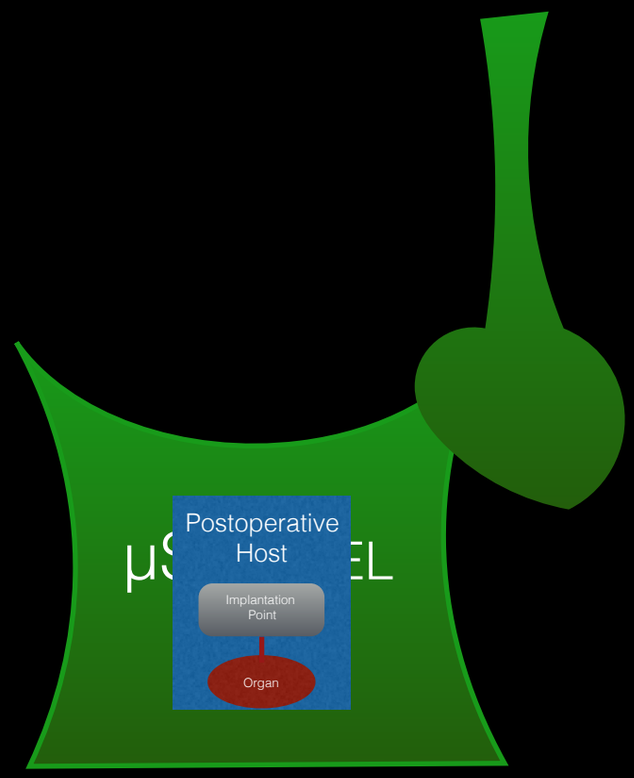
64 bit Ubuntu 14.10
16 GB RAM
8 threads



Experimental Methodology and Setup



Experimental Methodology and Setup



Experimental Methodology

a

x 20

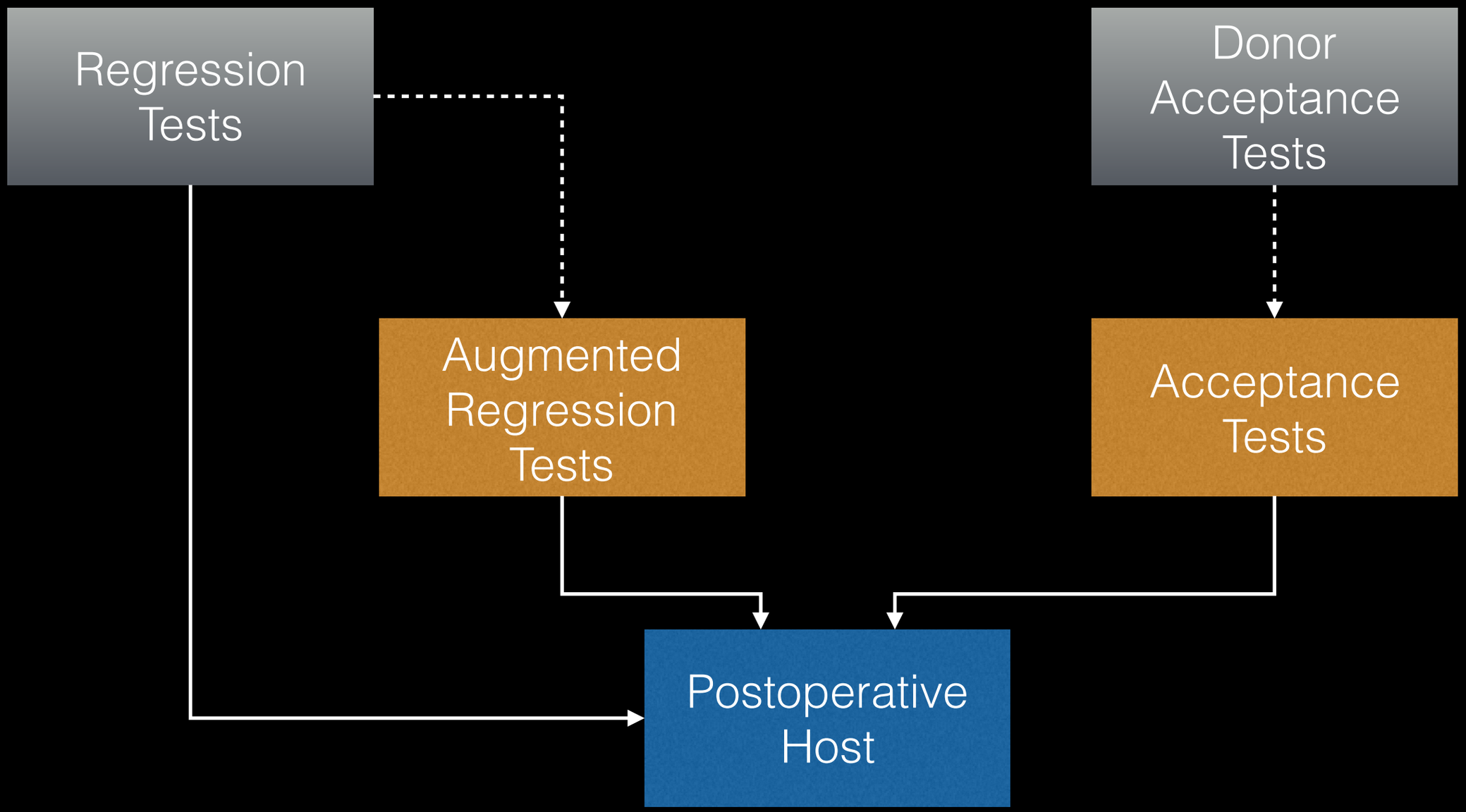
Postoperative Host

Implantation Point

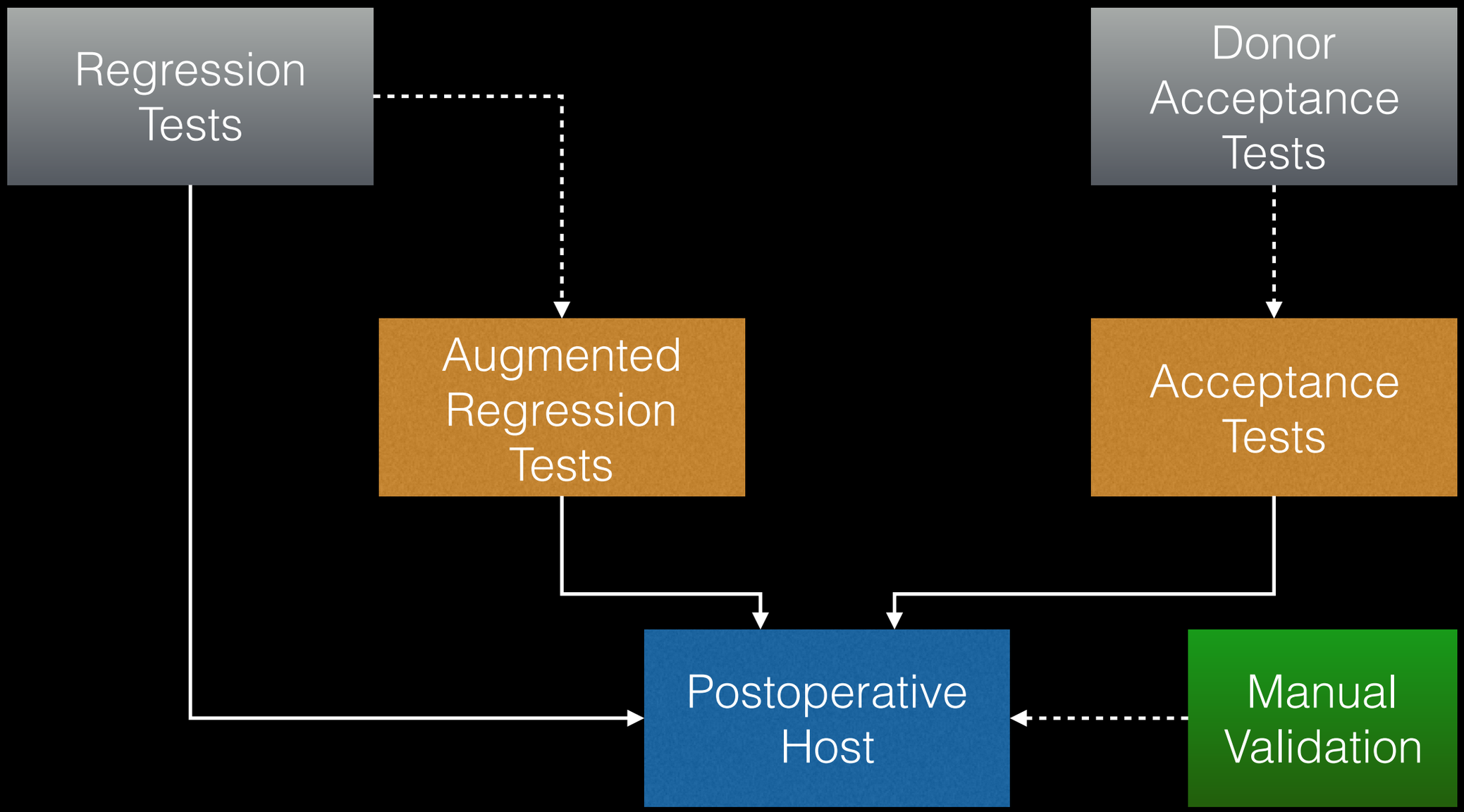
Organ

μSCALPEL

Transplant Validation



Transplant Validation



Transplant Validation

Regression Tests

Donor Acceptance Tests

Augmented Regression Tests

Acceptance Tests

Postoperative Host

Manual Validation

Empirical Study

		Execution Time (minutes)		
Donor	Host	Successful	Average	Total (hours)
Idct	Pidgin	**16**	**5**	**97**
Mytar	Pidgin	**16**	**3**	**65**
Web	Pidgin	**0**	**8**	**160**
Cflow	Pidgin	**15**	**58**	**1151**
Tux	Pidgin	**15**	**29**	**574**
Idct	Cflow	**16**	**3**	**59**
Mytar	Cflow	**17**	**3**	**53**
Web	Cflow	**0**	**5**	**102**
Cflow	Cflow	**20**	**44**	**872**
Tux	Cflow	**14**	**31**	**623**
Idct	SoX	**15**	**12**	**233**
Mytar	SoX	**17**	**3**	**60**
Web	SoX	**0**	**7**	**132**
Cflow	SoX	**14**	**89**	**74**
Tux	SoX	**13**	**34**	**94**
TOTAL		188/300	334	72

Empirical Study

Donor	Host	Execution Time (minutes)		
			Average	Total (hours)
Idct	Pidgin	16	5	97
Mytar	Pidgin	16	3	65
Web	Pidgin	0	8	160
Cflow	Pidgin	15	58	1151
Tux	Pidgin	15	29	574
Idct	Cflow	16	3	59
Mytar	Cflow	17	3	53
Web	Cflow	0	5	102
Cflow	Cflow	20	44	872
Tux	Cflow	14	31	623
Idct	SoX	15	12	233
Mytar	SoX	17	3	60
Web	SoX	0	7	132
Cflow	SoX	14	89	74
Tux	SoX	[illegible]	34	94
TOTAL			334	72

Successful

188/300

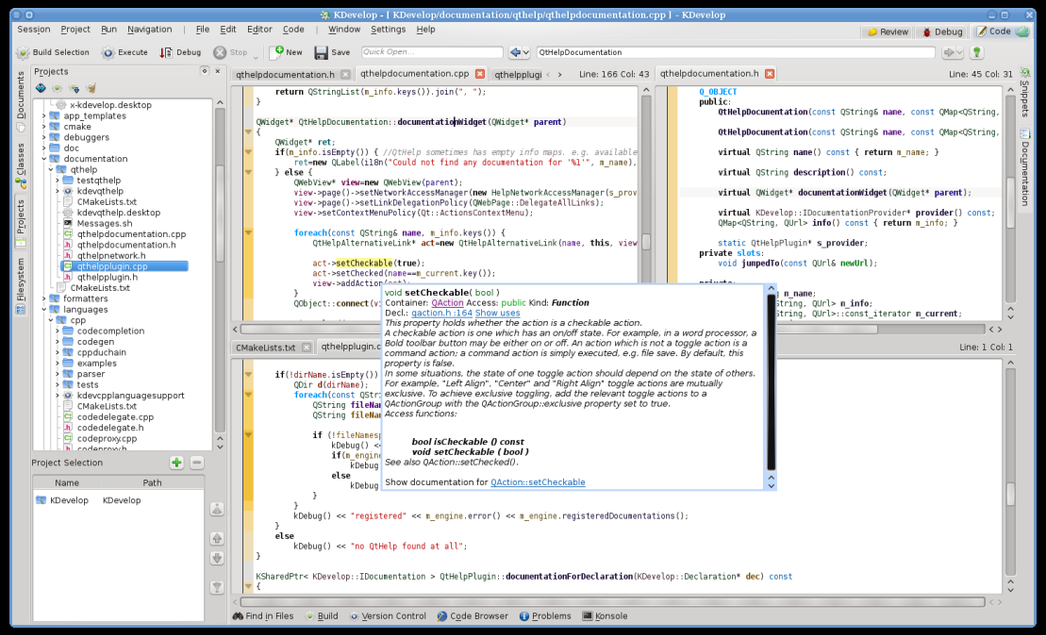
Empirical Study

			Execution Time (minutes)	
Donor	Host	Success	Average	Total (hours)
Idct	Pidgin	**16**	**5**	**97**
Mytar	Pidgin	**16**	**3**	**65**
Web	Pidgin	**0**	**8**	**160**
Cflow	Pidgin	**15**	**58**	**1151**
Tux	Pidgin	**15**	**29**	**574**
Idct	Cflow	**16**	**3**	**59**
Mytar	Cflow	**17**	**3**	**53**
Web	Cflow	**0**	**5**	**102**
Cflow	Cflow	**20**	**44**	**872**
Tux	Cflow	**14**	**31**	**623**
Idct	SoX	**15**	**12**	**233**
Mytar	SoX	**17**	**3**	**60**
Web	SoX	**0**	**7**	**132**
Cflow	SoX	**14**	**89**	**74**
Tux	SoX	**13**	**34**	**94**
TOTAL		188/300	334	72

Empirical Study

		Execution Time (minutes)		
Donor	Host	Successful	Average	Total (hours)
Idct	Pidgin	**16**	**5**	**97**
Mytar	Pidgin	**16**	**3**	**65**
Web	Pidgin	**0**	**8**	**160**
Cflow	Pidgin	**15**	**58**	**1151**
Tux	Pidgin	**15**	**29**	**574**
Idct	Cflow	**16**	**3**	**59**
Mytar	Cflow	**17**	**3**	**53**
Web	Cflow	**0**	**5**	**102**
Cflow	Cflow	**20**	**44**	**872**
Tux	Cflow	**14**	**31**	**623**
Idct	SoX	**15**	**12**	**233**
Mytar	SoX	**17**	**3**	**60**
Web	SoX	**0**	**7**	**132**
Cflow	SoX	**14**	**89**	**74**
Tux	SoX	**13**	**34**	**94**
TOTAL		188/300	334	72

Kate

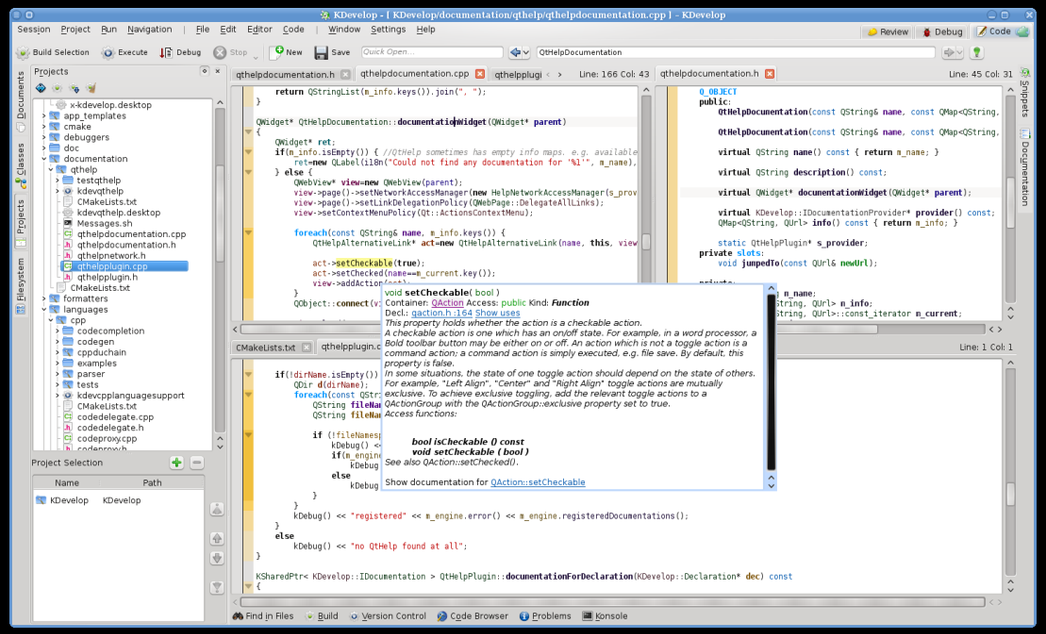


μSCALPEL

Kate

C Layout Feature?

C Call Graphs?

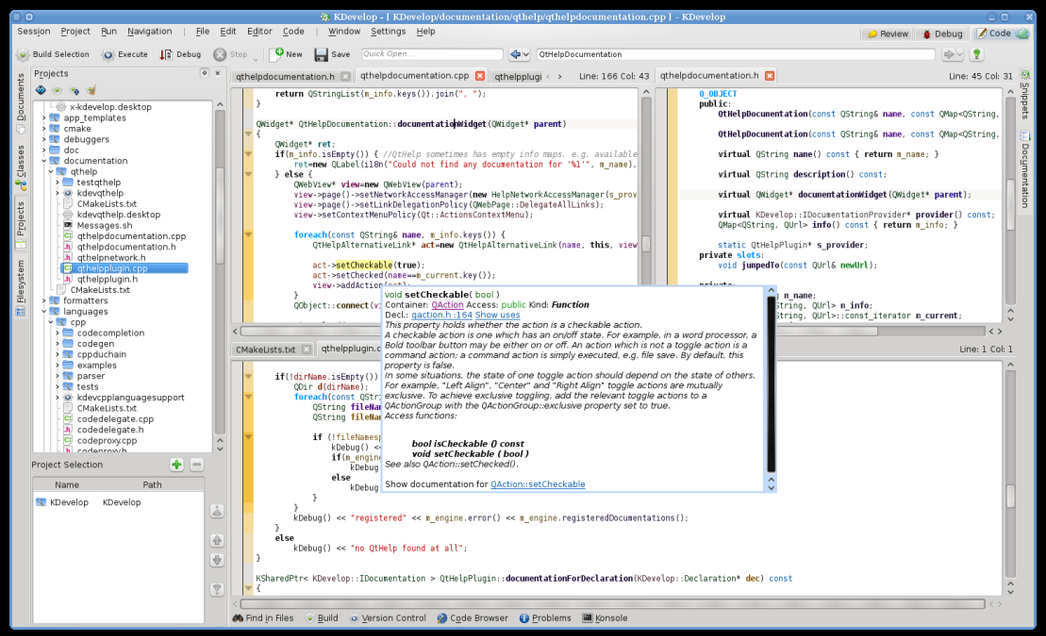


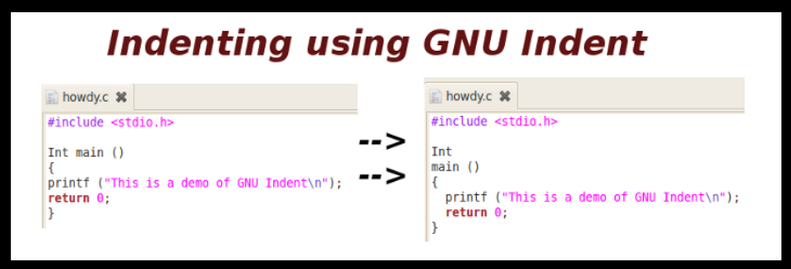
μSCALPEL

Kate

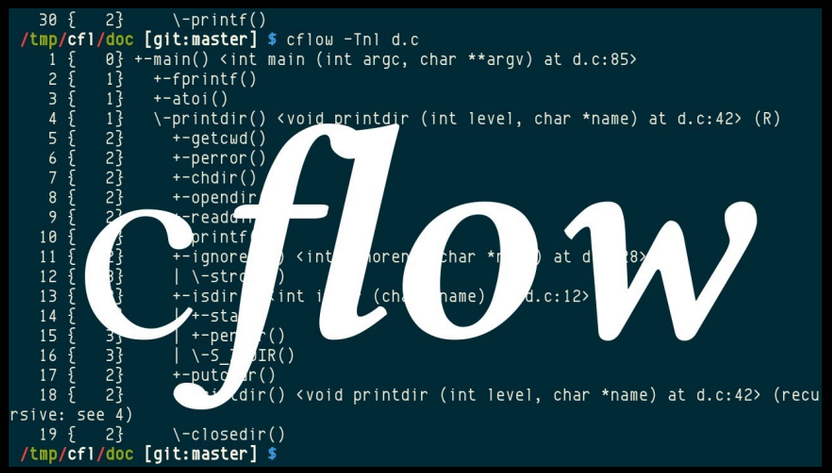
C Layout Feature?

C Call Graphs?

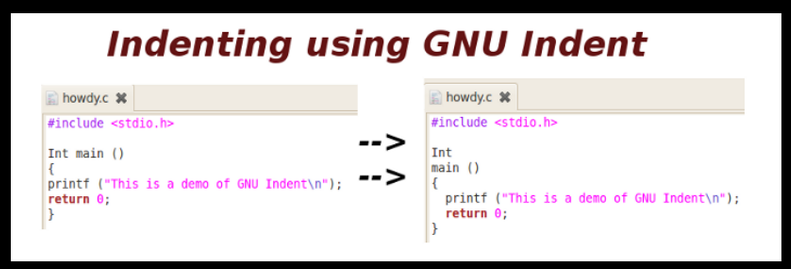


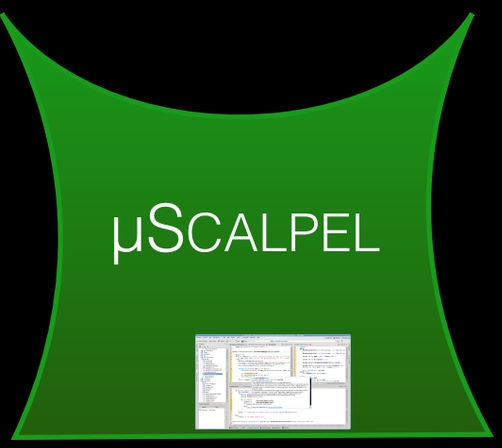


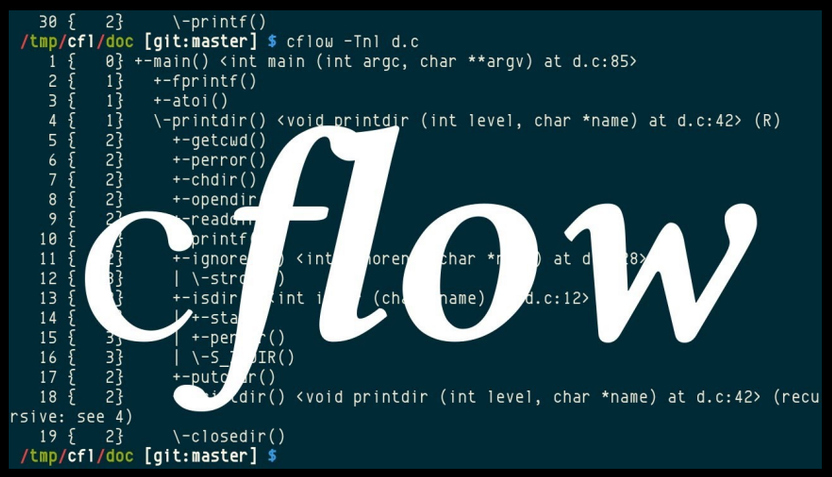
μSCALPEL



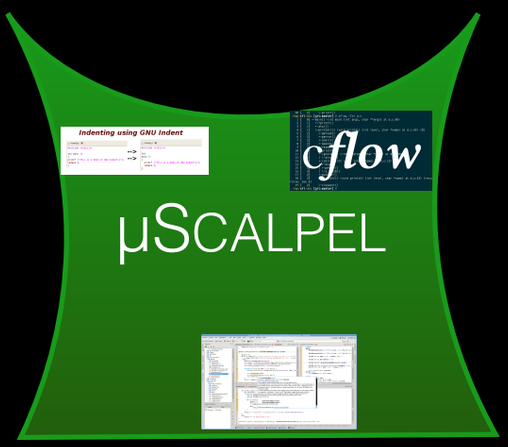
Kate


Indenting using GNU Indent
howdy.c
#include <stdio.h>
Int main ()
{
printf ("This is a demo of GNU Indent\n");
return 0;
}
-->
-->
howdy.c
#include <stdio.h>
Int
main ()
{
printf ("This is a demo of GNU Indent\n");
return 0;
}

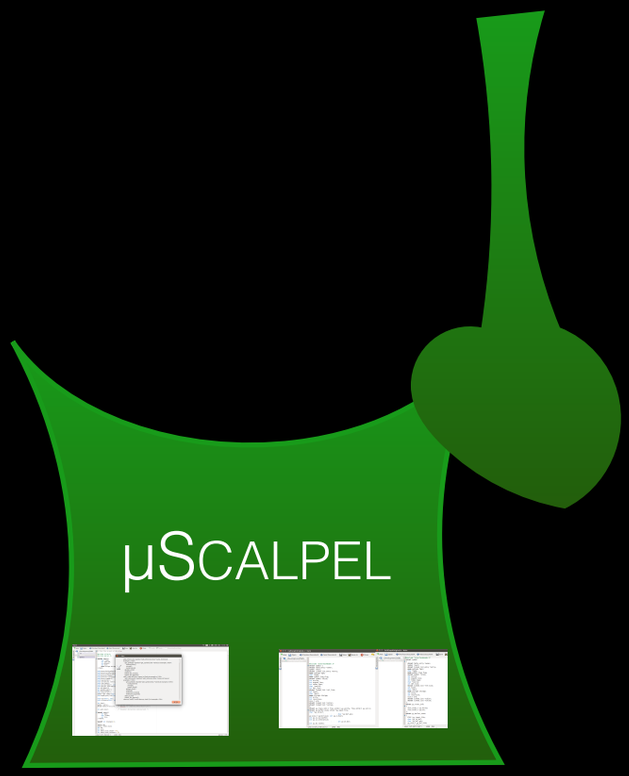

µSCALPEL


30 { 2} \-printf()
/tmp/cfl/doc [git:master] $ cflow -Tnl d.c
1 { 0} +-main() <int main (int argc, char **argv) at d.c:85>
2 { 1} +-fprintf()
3 { 1} +-atoi()
4 { 1} \-printdir() <void printdir (int level, char *name) at d.c:42> (R)
5 { 2} +-getcwd()
6 { 2} +-perror()
7 { 2} +-chdir()
8 { 2} +-opendir()
9 { 2} +-readdir()
10 { 2} +-printf()
11 { 2} +-ignore_dot_or_dotdot() <int ignore_dot_or_dotdot (char *name) at d.c:28>
12 { 3} | \-strcmp()
13 { 2} +-isdir() <int isdir (char *name) at d.c:12>
14 { 3} | +-stat()
15 { 3} | +-perror()
16 { 3} | \-S_ISDIR()
17 { 2} +-putchar()
18 { 2} +-printdir() <void printdir (int level, char *name) at d.c:42> (recursive: see 4)
19 { 2} \-closedir()
/tmp/cfl/doc [git:master] $
cflow

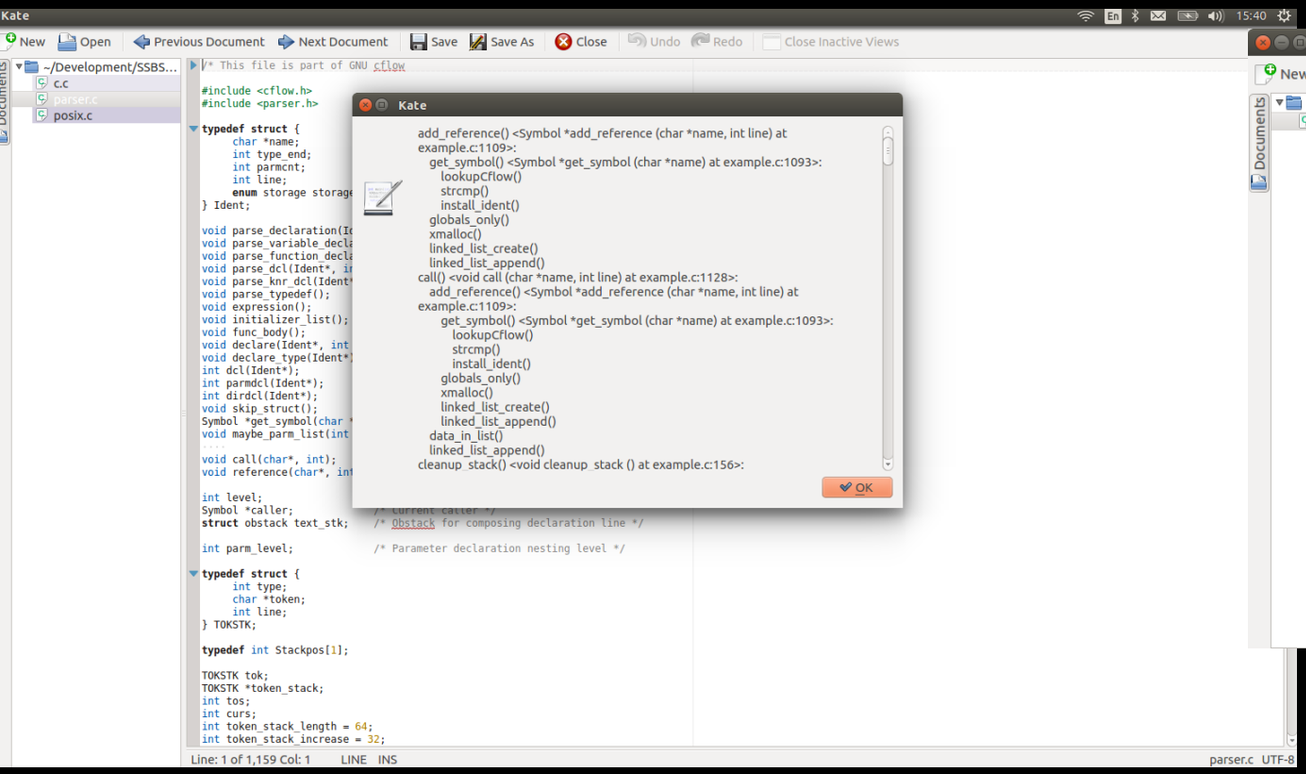
Kate

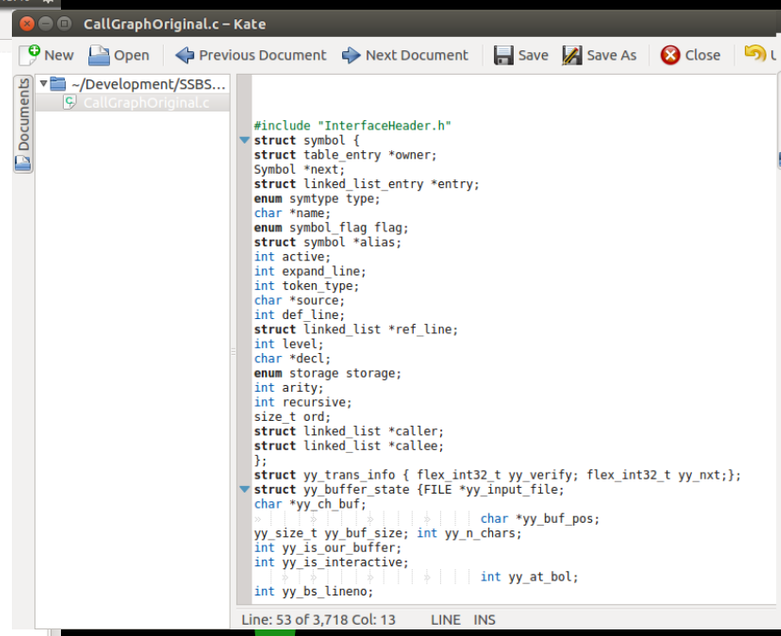


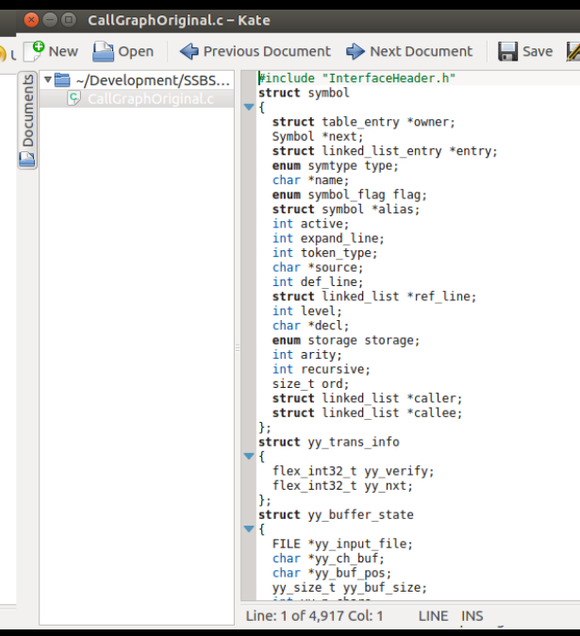
Kate

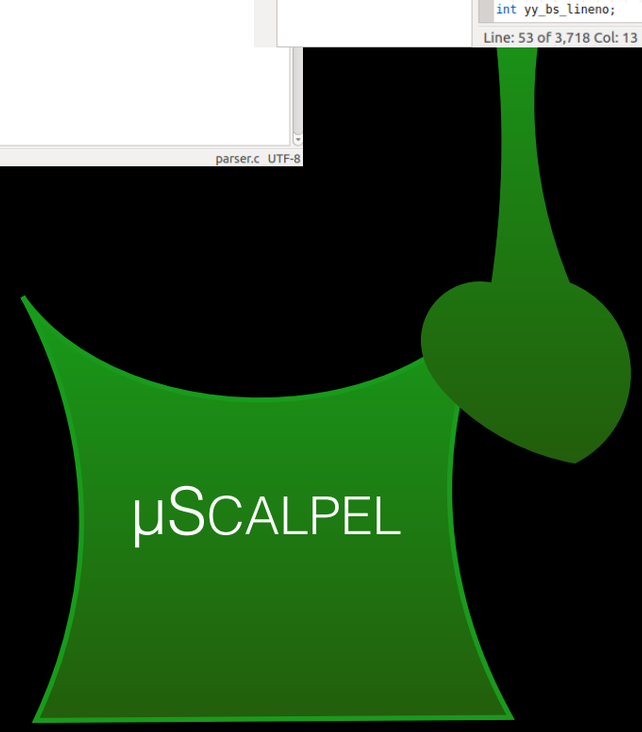


Kate


Kate
New
Open
Previous Document
Next Document
Save
Save As
Close
Undo
Redo
Close Inactive Views
~/Development/SSBS...
c.c
parser.c
posix.c
/* This file is part of GNU cflow
#include <cflow.h>
#include <parser.h>
typedef struct {
char *name;
int type_end;
int parmcnt;
int line;
enum storage storage
} Ident;
void parse_declaration(Id
void parse_variable_decla
void parse_function_decla
void parse_dcl(Ident*, in
void parse_knr_dcl(Ident*
void parse_typedef();
void expression();
void initializer_list();
void func_body();
void declare(Ident*, int
void declare_type(Ident*)
int dcl(Ident*);
int parmdcl(Ident*);
int dirdcl(Ident*);
void skip_struct();
Symbol *get_symbol(char *
void maybe_parm_list(int
void call(char*, int);
void reference(char*, int
int level;
Symbol *caller;
struct obstack text_stk; /* Obstack for composing declaration line */
int parm_level; /* Parameter declaration nesting level */
typedef struct {
int type;
char *token;
int line;
} TOKSTK;
typedef int Stackpos[1];
TOKSTK tok;
TOKSTK *token_stack;
int tos;
int curs;
int token_stack_length = 64;
int token_stack_increase = 32;
Kate
add_reference() <Symbol *add_reference (char *name, int line) at example.c:1109>:
get_symbol() <Symbol *get_symbol (char *name) at example.c:1093>:
lookupCflow()
strcmp()
install_ident()
globals_only()
xmalloc()
linked_list_create()
linked_list_append()
call() <void call (char *name, int line) at example.c:1128>:
add_reference() <Symbol *add_reference (char *name, int line) at example.c:1109>:
get_symbol() <Symbol *get_symbol (char *name) at example.c:1093>:
lookupCflow()
strcmp()
install_ident()
globals_only()
xmalloc()
linked_list_create()
linked_list_append()
data_in_list()
linked_list_append()
cleanup_stack() <void cleanup_stack () at example.c:156>:
OK
Line: 1 of 1,159 Col: 1
LINE
INS
parser.c
UTF-8


CallGraphOriginal.c – Kate
New
Open
Previous Document
Next Document
Save
Save As
Close
Documents
~/Development/SSBS...
CallGraphOriginal.c
#include "InterfaceHeader.h"
struct symbol {
struct table_entry *owner;
Symbol *next;
struct linked_list_entry *entry;
enum symtype type;
char *name;
enum symbol_flag flag;
struct symbol *alias;
int active;
int expand_line;
int token_type;
char *source;
int def_line;
struct linked_list *ref_line;
int level;
char *decl;
enum storage storage;
int arity;
int recursive;
size_t ord;
struct linked_list *caller;
struct linked_list *callee;
};
struct yy_trans_info { flex_int32_t yy_verify; flex_int32_t yy_nxt;};
struct yy_buffer_state {FILE *yy_input_file;
char *yy_ch_buf;
char *yy_buf_pos;
yy_size_t yy_buf_size; int yy_n_chars;
int yy_is_our_buffer;
int yy_is_interactive;
int yy_at_bol;
int yy_bs_lineno;
Line: 53 of 3,718 Col: 13
LINE
INS


CallGraphOriginal.c – Kate
New
Open
Previous Document
Next Document
Save
Documents
~/Development/SSBS...
CallGraphOriginal.c
#include "InterfaceHeader.h"
struct symbol
{
struct table_entry *owner;
Symbol *next;
struct linked_list_entry *entry;
enum symtype type;
char *name;
enum symbol_flag flag;
struct symbol *alias;
int active;
int expand_line;
int token_type;
char *source;
int def_line;
struct linked_list *ref_line;
int level;
char *decl;
enum storage storage;
int arity;
int recursive;
size_t ord;
struct linked_list *caller;
struct linked_list *callee;
};
struct yy_trans_info
{
flex_int32_t yy_verify;
flex_int32_t yy_nxt;
};
struct yy_buffer_state
{
FILE *yy_input_file;
char *yy_ch_buf;
char *yy_buf_pos;
yy_size_t yy_buf_size;
Line: 1 of 4,917 Col: 1
LINE
INS


μSCALPEL

Case Study - Kate

		Execution Time (minutes)		
Donor	Host	Successful	Average	Total (hours)
Cflow	Kate	**16**	**101**	**33**
Indent	Kate	**18**	**31**	**11**
TOTAL		34/40	132	44

Case Study - Kate

		Successful	Execution Time (minutes)	
Donor	Hos		Average	Total (hours)
Cflow	Kate	**16**	**101**	**33**
Indent	Kate	**18**	**31**	**11**
TOTAL		34/40	132	44

Case Study - Kate

			Execution Time (minutes)	
Donor	Host	Successful	Average	Total (hours)
Cflow	Kate	**16**	**101**	**33**
Indent	Kate	**18**	**31**	**11**
TOTAL		34/40	132	44

Case Study - Kate

		Execution Time (minutes)			Total (hours)
Donor	Host	Successful	Average		
Cflow	Kate	**16**	**101**	**33**	
Indent	Kate	**18**	**31**	**11**	
TOTAL		34/40	132		44

Case Study: x264 & VLC

Donor

Host

Case Study: x264 & VLC

Award winning tool for H.264 encoding [2,3,4]

Donor

Host

Case Study: x264 & VLC

Award winning tool for H.264 encoding [2,3,4]

Donor

“Most popular desktop video player” [1]

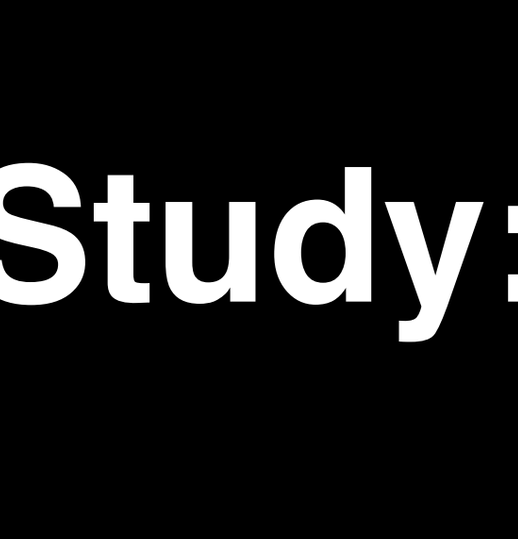
Host

Case Study: x264 & VLC

Award winning tool for H.264 encoding [2,3,4]

"Most popular desktop video player" [1]

Donor



Host

Case Study: x264 & VLC

Award winning tool for H.264 encoding [2,3,4]

"Most popular desktop video player" [1]

Organ: H264

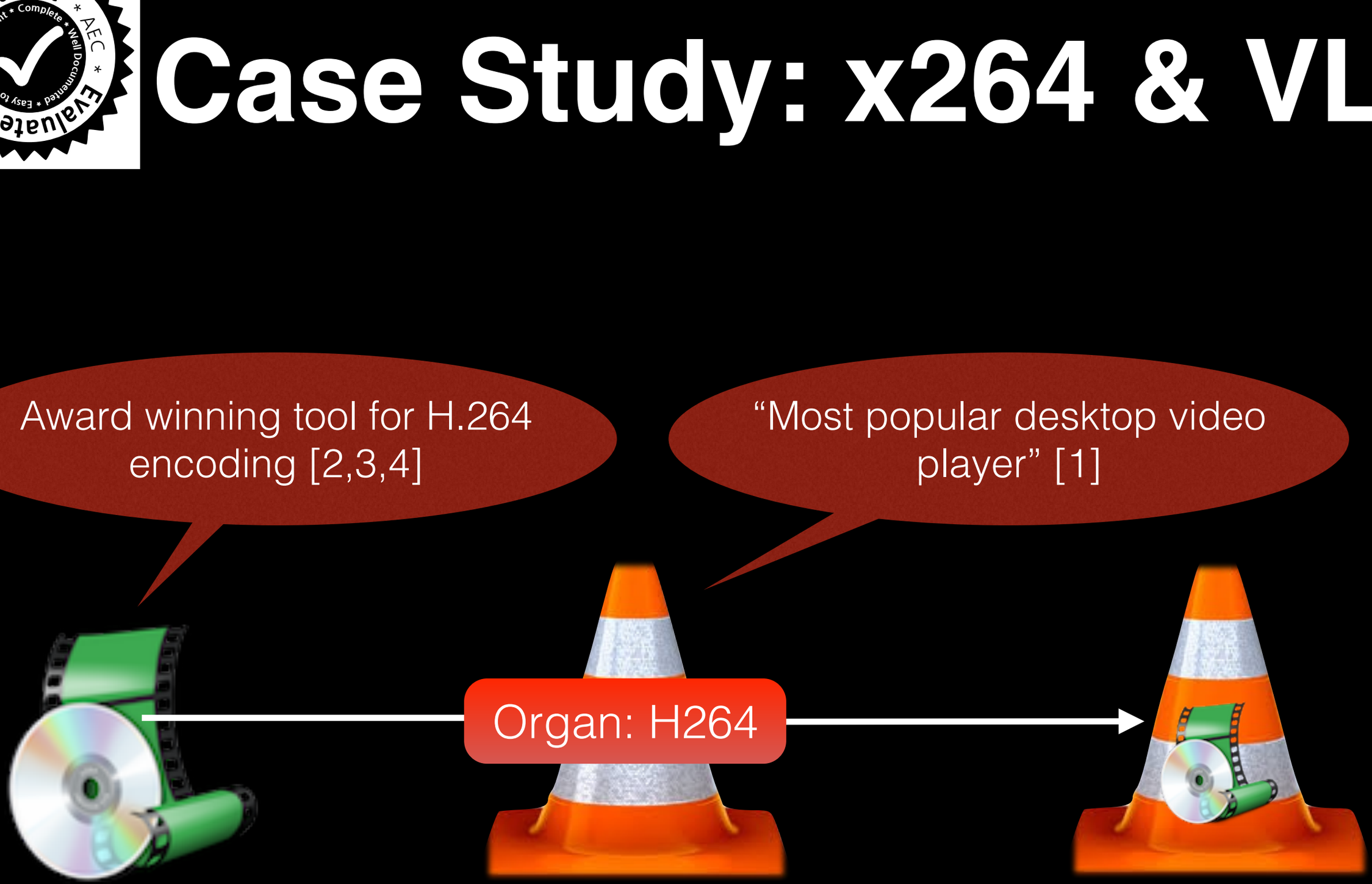
Donor

Host

Case Study: x264 & VLC

Award winning tool for H.264 encoding [2,3,4]

"Most popular desktop video player" [1]



Case Study: x264 & VLC

Automatic Transplantation of H264 Encoder

	Time (hours)	Regression Tests	Manual Tests	Acceptance Tests
μSCALPEL	26	100%	100%	100%

Autotransplantation is Human Competitive!

Autotransplantation vs Human Transplantation

Autotransplantation vs Human Transplantation

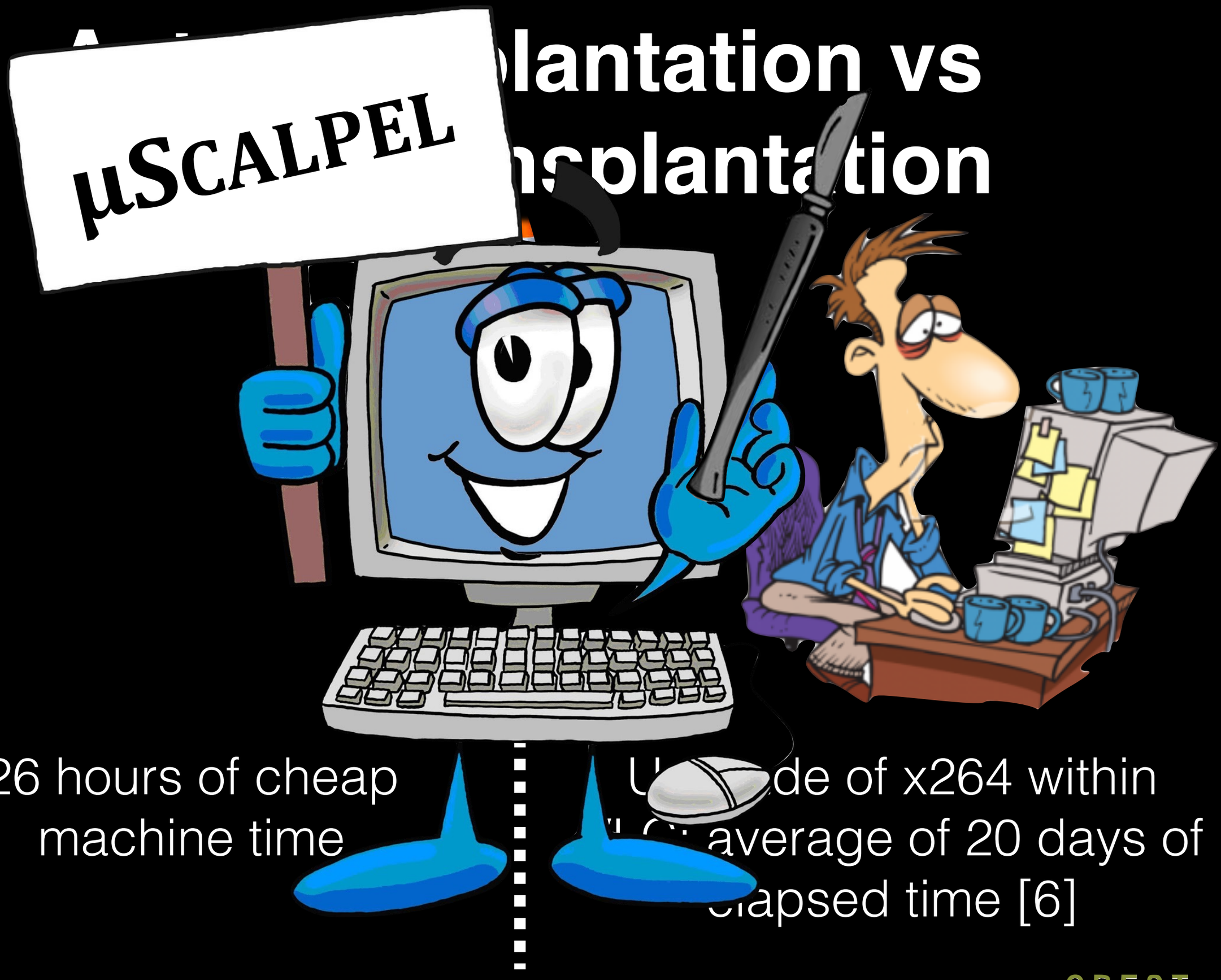
Autotransplantation vs Human Transplantation

Autotransplantation vs Human Transplantation

26 hours of cheap machine time

Upgrade of x264 within VLC: average of 20 days of elapsed time [6]

Automated Software Transplantation vs Manual Transplantation



26 hours of cheap machine time

Upgrade of x264 within average of 20 days of elapsed time [6]

Best Human Implementation

Best Human Implementation

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

Best Human Implementation

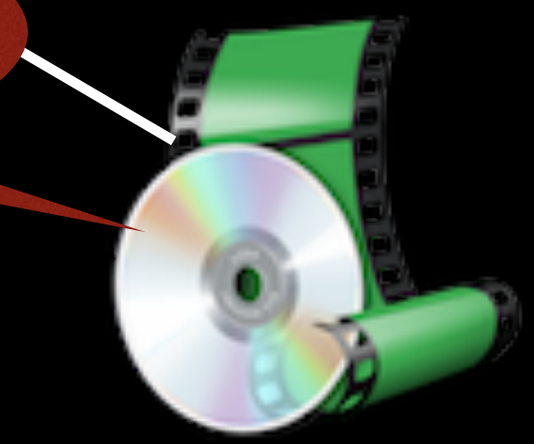
MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

Best Human Implementation

GRAPHICS & MEDIA LAB
VIDEO GROUP

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

x264 won with ~24% better encoding
than second place



Best Human Implementation

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

Best Human Implementation

GRAPHICS & MEDIA LAB
VIDEO GROUP

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

Best Human Implementation

GRAPHICS & MEDIA LAB
VIDEO GROUP

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

2.4% faster

Best Human Implementation

GRAPHICS & MEDIA LAB
VIDEO GROUP

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

2.4% faster

We automatically transplanted
new functionality!

Best Human Implementation

GRAPHICS & MEDIA LAB
VIDEO GROUP

MSU Sixth MPEG-4 AVC/H.264
Video Codecs Comparison [4]

2.4% faster

Social Media Reactions: “Am I Obsolete?”

Social Media Reactions: “Am I Obsolete?”

UK Contractor Forum [7]

5th August 2015 10:48 #1

DimPrawn

Richer than sasguru

DimPrawn - scorchio!

Join Date: Jul 2005
Location: In a state of dysphoria
Posts: 30,793
Thanks (Given): 206
Thanks (Received): 516
Likes (Given): 2695
Likes (Received): 2612

I am obsolete

Code 'transplant' could revolutionise programming (Wired UK)

Code has been automatically "transplanted" from one piece of software to another for the first time, with researchers claiming the breakthrough could radically change how computer programs are created.

The process, demonstrated by researchers at University College London, has been likened to organ transplantation in humans. Known as MuScalpel, it works by isolating the code of a useful feature in a 'donor' program and transplanting this "organ" to the right "vein" in software lacking the feature.

Bugger, no one is going to hire me now.

5th August 2015 10:53

BrilloPad

TripleIronDad

BrilloPad is a fount of knowledge

Couple that to a 3D printer and the ruling class will not need the plebs soon.

Social Media Reactions: “Am I Obsolete?”

UK Contractor Forum [7]

5th August 2015 10:48 #1

DimPrawn

Richer than sasguru

DimPrawn - scorchio!

Join Date: Jul 2005
Location: In a state of dysphoria
Posts: 30,793
Thanks (Given): 206
Thanks (Received): 516
Likes (Given): 2695
Likes (Received): 2612

I am obsolete

Code 'transplant' could revolutionise programming (Wired UK)

Code has been automatically "transplanted" from one piece of software to another for the first time, with researchers claiming the breakthrough could radically change how computer programs are created.

The process, demonstrated by researchers at University College London, has been likened to organ transplantation in humans. Known as MuScalpel, it works by isolating the code of a useful feature in a 'donor' program and transplanting this "organ" to the right "vein" in software lacking the feature.

Bugger, no one is going to hire me now.

5th August 2015 10:53

BrilloPad

TripleIronDad

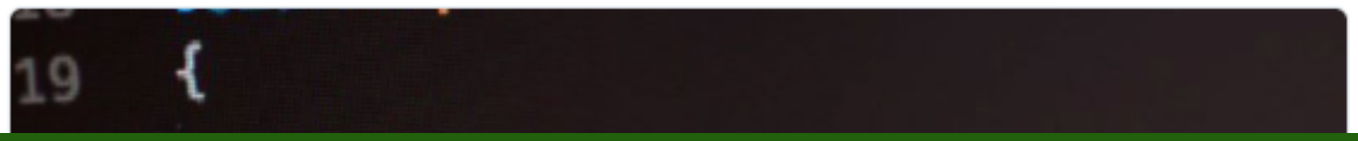
BrilloPad is a fount of knowledge

Couple that to a 3D printer and the ruling class will not need the plebs soon.

Anirban
@anirbanbasu

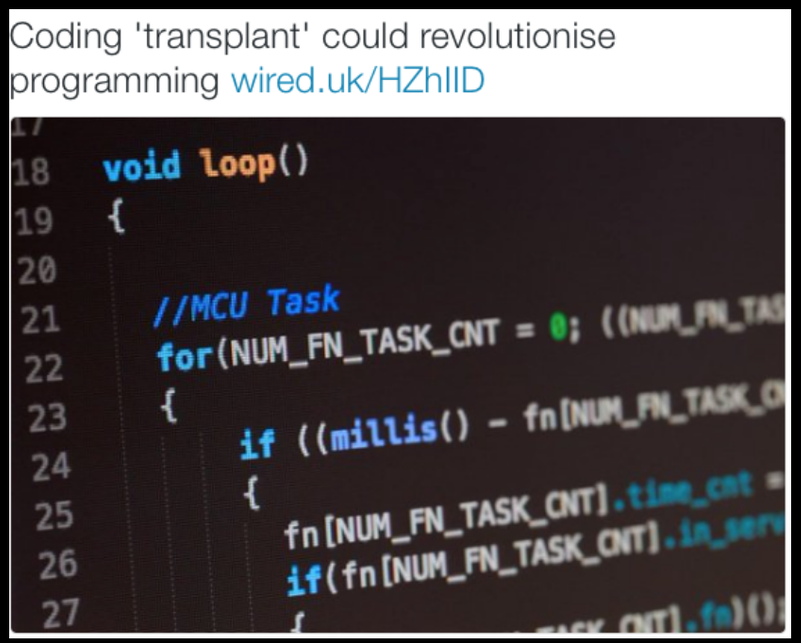
Follow

MuScalpel - code transplantation is here. So in the next 10 years there would be no jobs for computer programmers...



Public Recognition

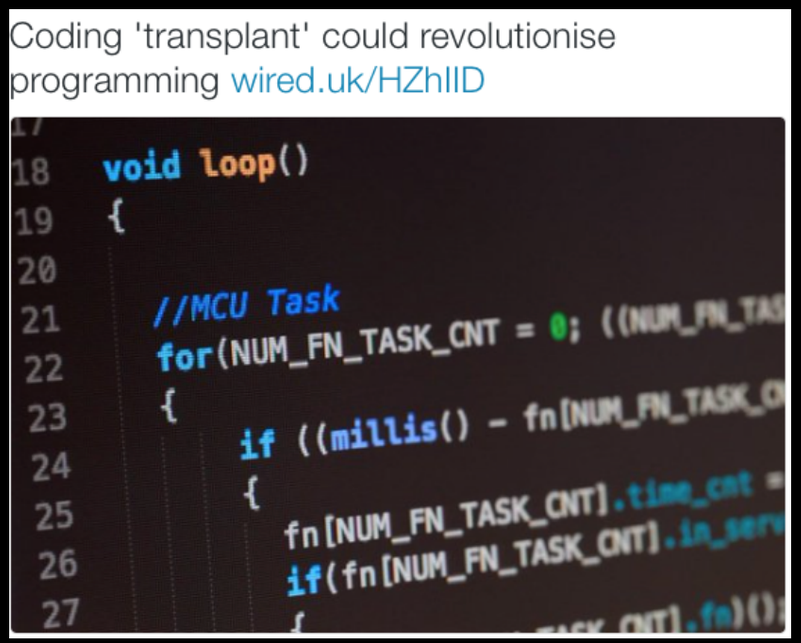
Public Recognition



WIRED

article, with more than 2000 shares

Public Recognition



WIRED

article, with more than 2000 shares

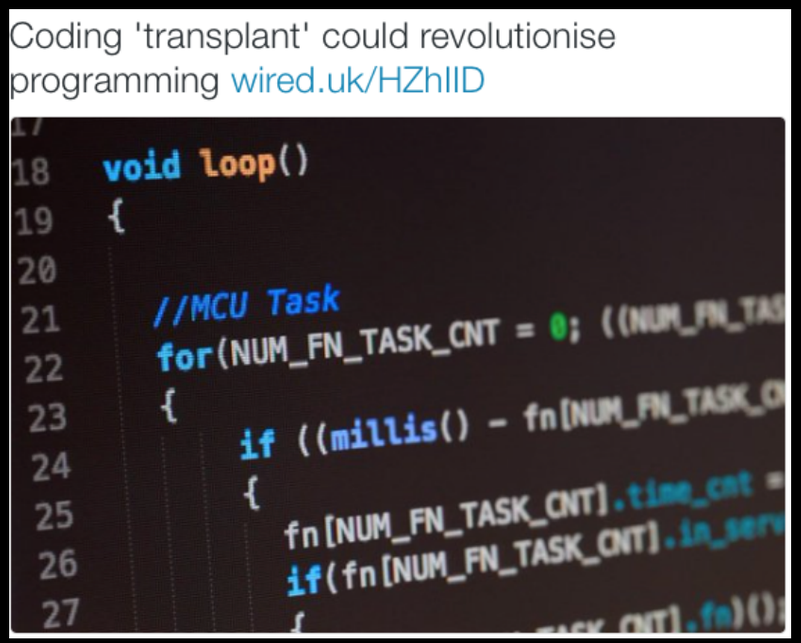
Public Recognition

Coding 'transplant' could revolutionise programming wired.uk/HZhIID

"the BBC's biggest global brand with sales of the TV show, DVDs, books, live shows and other merchandise worth more than £50m a year" [5]

than 2000 shares

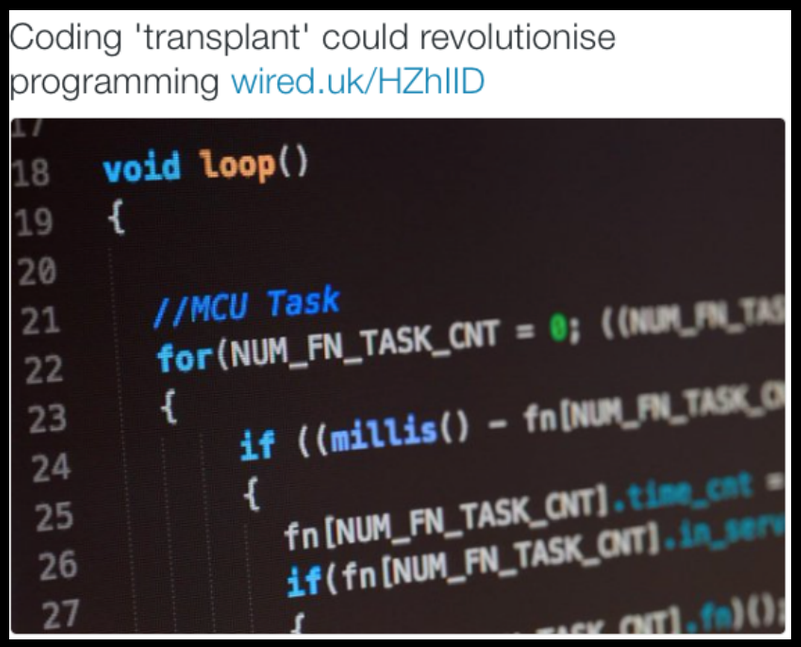
Public Recognition



WIRED

article, with more than 2000 shares

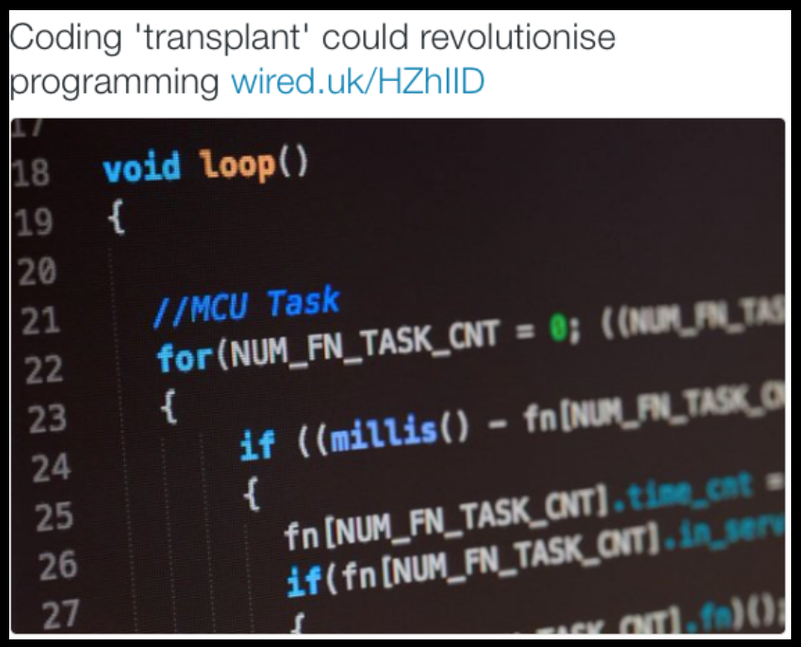
Public Recognition



WIRED

article, with more than 2000 shares

Public Recognition



WIRED

article, with more than 2000 shares

More shares for Autotransplantation!

Public Recognition

Public Recognition

ACM Distinguished Paper Award at ISSTA ‘15

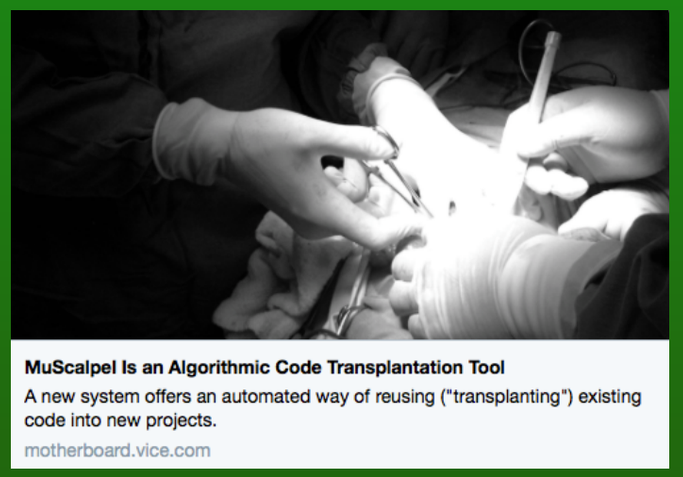
Public Recognition

ACM Distinguished Paper Award at ISSTA ‘15

Featured on:

BBC WORLD SERVICE Click

MOTHERBOARD



Humies 2016

Humies 2016

Competition

cash awards for human-competitive results that were produced by any form of genetic and evolutionary computation

Humies 2016

Competition

cash awards for human-competitive results that were produced by any form of genetic and evolutionary computation

$10.000

Humies 2016 Gold Medal

"it has been won for the past five years by games - except one where the gold medal was shared but still with a game" [8]

Humies 2016 Gold Medal

"it has been won for the past five years by games - except one where the gold medal was shared but still with a game" [8]

Humies 2016 Gold Medal

"it has been won for the past five years by games - except one where the gold medal was shared but still with a game" [8]

$5000

CREST Transplantation Awards

CREST Transplantation Awards

Silver Medal at Humies 2014

CREST Transplantation Awards

Silver Medal at Humies 2014

SSBSE 2014 Challenge Track Winner

CREST Transplantation Awards

Silver Medal at Humies 2014

SSBSE 2014 Challenge Track Winner

ISSTA 2015 Distinguish Paper Award

CREST Transplantation Awards

Silver Medal at Humies 2014

SSBSE 2014 Challenge Track Winner

ISSTA 2015 Distinguish Paper Award

Gold Medal at Humies 2016

CREST Transplantation Awards

Silver Medal at Humies 2014

SSBSE 2014 Challenge Track Winner

CREST Research is World Leading in Code Transplants!

ISSTA 2015 Distinguish Paper Award

Gold Medal at Humies 2016

Code Transplants

mming
neering

ntroduced the
a of using GP for
otransplantation

Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class

Humies 2014

Most cited paper

t al. 2014 — *Petke et al.*

Autotransplantation

Silver Medal at Humies 2014

SSBSE 2014 Challenge Track Winner

CREST Research is World Leading in Code Transplants!

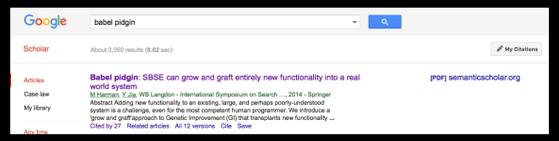
ISSTA 2015 Distinguish Paper Award

Gold Medal at Humies 2016

Awards

Best SBSE Challenge Paper

400 USD Amazon gift card plus 1,000 USD cash prize



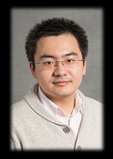
EPSRC Pioneering research and skills

£581,560 Grant
28 October 2015 — 27 October 2019
Microsoft, Visa Europe

People from CREST

Earl T. Barr

Mark Harman



Yue Jia

William B. Langdon

Justyna Petke

CREST

Public Recognition

ACM Distinguished Paper Award at ISSTA '15

Featured on:

BBC WORLD SERVICE Click

MOTHERBOARD

Humies 2016 Gold Medal

"it has been won for the past five years by games - except one where the gold medal was shared but still with a game"

$5000

References

[1] http://lifehacker.com/five-best-desktop-video-players-1503859883/1506086048

[2] http://www.compression.ru/video/codec_comparison/h264_2012/

[3] http://www.streamingmedia.com/articles/editorial/featured-articles/first-look-h.264-and-vp8-compared-67266.aspx

[4] http://www.compression.ru/video/codec_comparison/h264_2010/

[5] http://www.theguardian.com/media/2015/mar/11/top-gear-bbc-jeremy-clarkson

[6] Barr et al. Automated Software Transplantation

[7] UK Contractor Forum link

[8] https://www.facebook.com/mark.harman.794/posts/10155034911439838?comment_id=10155035033384838&reply_comment_id=10155035041274838&comment_tracking=%7B%22tn%22%3A%22R%22%7D