

Search-based Synthesis of Equivalent Method Sequences

Alberto Goffi, **Andrea Mattavelli**, Mauro Pezzè
University of Lugano - Switzerland

Alessandra Gorla
Saarland University - Germany

Paolo Tonella
Fondazione Bruno Kessler - Italy





Software is Redundant

“ A system is redundant when it is able to perform **equivalent functionalities** by executing **different code**.

Software is Redundant

“

A system is redundant when it is able to perform **equivalent functionalities** by executing **different code**.

- 
- compute **identical results**
 - lead to **identical states**

Equivalence in Software

Java Stack

```
Stack s = new Stack();  
//...  
//remove the element on top of the stack  
s.pop();
```

Equivalence in Software

Java Stack

```
Stack s = new Stack();
//...
//remove the element on top of the stack
s.pop();
s.remove(size() - 1);
s.removeElementAt(size() - 1);
```

Equivalence in Software

Java Stack

```
Stack s = new Stack();
//...
//remove the element on top of the stack
s.pop();
s.remove(size() - 1);
s.removeElementAt(size() - 1);
```

Joda-Time

```
DateTime t = new DateTime();
//...
//get the beginning of the day for time t
DateTime beginDay = t.millisOfDay().withMinimumValue();
```

Equivalence in Software

Java Stack

```
Stack s = new Stack();
//...
//remove the element on top of the stack
s.pop();
s.remove(size() - 1);
s.removeElementAt(size() - 1);
```

Joda-Time

```
DateTime t = new DateTime();
//...
//get the beginning of the day for time t
DateTime beginDay = t.millisOfDay().withMinimumValue();
                     = t.toDateMidnight().toDateTime();
                     = t.withTimeAtStartOfDay();
```

Exploiting Equivalence

Automatic Recovery

Antonio Carzaniga*

*University
Faculty of
Lugano, Sv

Abstract—We present a technique to make applications fault-tolerant. This technique is intended to make faulty application functional in the field while it works on permanent and radical fixes. We target applications built on reusable components. In this technique exploits the intrinsic redundancy of those by identifying workarounds consisting of alternates of faulty components that avoid the failure. The currently implemented application is Java application, but no assumption about the nature of the application is made. The technique can be applied to any application without interrupting the execution flow of the application or restarting its components. We demonstrate this technique on four mid-size applications and libraries of reusable components affected by real faults. In these cases the technique is effective: the application fully functional with between 15% and 30% of the failure-causing faults, depending on the application. Our experiments also show that the technique incurs runtime overhead in all cases.

1. INTRODUCTION

Software systems are sometimes released and deployed with faults, and those faults may cause field failures despite the best effort and the rigorous testing by developers and testers. Furthermore, even when reported to developers, field failures may take a long time to be diagnosed and eliminated. As a perhaps extreme but unique example, consider fault n. 3655 in the FitFitter system, which was reported first in March 1999 and still happens despite the best effort and the rigorous testing by developers and testers. Our experiments show that workarounds can be applied within Web applications, through a simple plug-in, and that program-rewriting rules derive complementary properties of a common library can be found valid and previously unknown workarounds.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging; Error handling and recovery

General Terms

Reliability, Design

Keywords

Automatic Workarounds, Web Applications, Web Services

*Mauro Pezzè is also with the University of Milan.

Permission to make digital or hard copies of all or part of this material for personal or classroom use is granted without fee provided that (1) the full bibliographic details are given, (2) this notice and the full citation on the first page, (3) the full text of the article is not changed in any way, and (4) permission is not granted for other kinds of copying such as copying for general distribution for advertising or promotional purposes, for creating new collective works, or for resale.

FSE '14, November 7–11, 2014, Santa Fe, New Mexico, USA
Copyright 2014 ACM 978-1-4503-2756-3/14/05 ...\$15.00.

https://bugzilla.mozilla.org/show_bug.cgi?id=3655

Automatic Workarounds

Antonio Carzaniga, Alessandra Gorla

{antonio.carzaniga,alessandra.gorla}@saarland.de

ABSTRACT

We present a technique that finds and executes workarounds for faulty Web applications automatically and transparently. Automatic workarounds exploit the inherent redundancy of Web applications, whereby a functionality of the application can be obtained through different sequences of calls to Web APIs. In general, runtime workarounds are triggered by a failure, and require that the application remains in a consistent state before and after the execution of a workaround. Therefore, they are ideally suited for active Web applications, since those allow the developer to detect a failure with minimal effort, and use read-only state or manage their state through a transactional data store. In this paper we focus on enhancing BPEL engines with such facilities. Unfortunately, the de facto standard for modeling Web services "heals" themselves in case of failure, current BPEL engines lack appropriate mechanisms to permit satisfying self-healing requirements. We propose a transparent way to functional failures of composite Web services during runtime. This paper presents a technique for automatically inserting oracles into BPEL engines to enhance BPEL engines with such facilities. The enhancement happens through the following steps: i) identify categories of failures, develop solutions to recover from these failures, and suggest a prototype that illustrates our ideas. ii) we can then apply these solutions to BPEL engines (with focus on composition). iii) we can then apply these solutions to BPEL engines (with focus on composition).

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging; Error handling and recovery

General Terms

Reliability, Design

Keywords

Automatic Workarounds, Web Applications, Web Services

*Mauro Pezzè is also with the University of Milan.

Permission to make digital or hard copies of all or part of this material for personal or classroom use is granted without fee provided that (1) the full bibliographic details are given, (2) this notice and the full citation on the first page, (3) the full text of the article is not changed in any way, and (4) permission is not granted for other kinds of copying such as copying for general distribution for advertising or promotional purposes, for creating new collective works, or for resale.

FSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-3/14/05 ...\$15.00.

https://bugzilla.mozilla.org/show_bug.cgi?id=3655

International Conference on Software Technology and Engineering Practice (STEP'04)

On the Enhancement of BPEL Engines

Sattanathan Subramanian,
Ghita Kouadri-Saadouni

¹INRIA Saclay-Île-de-France, France

²Precise, University of Saarland, Germany

³IBM India Research Lab, Bangalore, India

⁴Comlab, Cambridge, UK

⁵College of Information Technology, University of Saarland, Germany

Abstract

The dynamic nature of the Internet poses various challenges to the successful execution of composite Web services. One of the main challenges is how to deal with faults in composite Web services. Faults in composite Web services are samples of these challenges. It is well known that the success of a composite Web service depends on the correct execution of its constituent Web services. Unfortunately, the de facto standard for modeling Web services "heals" themselves in case of failure, current BPEL engines lack appropriate mechanisms to permit satisfying self-healing requirements. We propose a transparent way to functional failures of composite Web services during runtime. This paper presents a technique for automatically inserting oracles into BPEL engines to enhance BPEL engines with such facilities. The enhancement happens through the following steps: i) identify categories of failures, develop solutions to recover from these failures, and suggest a prototype that illustrates our ideas. ii) we can then apply these solutions to BPEL engines (with focus on composition). iii) we can then apply these solutions to BPEL engines (with focus on composition).

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software Composition; D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Reliability, Design

Keywords

Automatic Workarounds, Web Applications, Web Services

Permission to make digital or hard copies of all or part of this material for personal or classroom use is granted without fee provided that (1) the full bibliographic details are given, (2) this notice and the full citation on the first page, (3) the full text of the article is not changed in any way, and (4) permission is not granted for other kinds of copying such as copying for general distribution for advertising or promotional purposes, for creating new collective works, or for resale.

FSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-3/14/05 ...\$15.00.

https://bugzilla.mozilla.org/show_bug.cgi?id=3655

Cross-Validation from Intrinsic Properties

Antonio Carzaniga,
University of Saarland,
antonio.carzaniga@cs.uni-saarland.de

Alessandra Gorla,
Saarland University,
Germany
gorla@st.cs.uni-saarland.de

¹INRIA Saclay-Île-de-France, France

²Precise, University of Saarland, Germany

³IBM India Research Lab, Bangalore, India

⁴Comlab, Cambridge, UK

⁵College of Information Technology, University of Saarland, Germany

Abstract

Despite the recent advances in automatic test generation, testers must still write test oracles manually. If no test oracle is available, it might be possible to generate test cases based on the specifications. A technique that is based on a form of specification mining leverages more information from the system under test than is usually available. In this paper we assume that the system under test is somewhat reliable in the sense that some operations are designed to do what they are expected to do. Otherwise, if two operations have different behaviors, then they are likely to be different. Our experiments show that this technique exists and is easily documented. We then generate cross-checking test cases with the help of oracles, which we replace some operations with redundant ones. We then develop this notion of cross-checking oracles in a technique to automatically insert oracles into BPEL engines. An experimental evaluation shows that cross-checking used in combination with automatic test generation techniques can be very effective in revealing faults. Finally, we can then apply these oracles to BPEL engines to automatically insert oracles into BPEL engines.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software Composition; D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Reliability, Design

Keywords

Automatic Workarounds, Web Applications, Web Services

Permission to make digital or hard copies of all or part of this material for personal or classroom use is granted without fee provided that (1) the full bibliographic details are given, (2) this notice and the full citation on the first page, (3) the full text of the article is not changed in any way, and (4) permission is not granted for other kinds of copying such as copying for general distribution for advertising or promotional purposes, for creating new collective works, or for resale.

FSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-3/14/05 ...\$15.00.

https://bugzilla.mozilla.org/show_bug.cgi?id=3655

Metamorphic Testing and Beyond *

T. Y. Chen[†], F.-C. Kuo[†], T. H. Tse^{‡§}, Zhi Quan Zhou[†]

[†]School of Information Technology
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia

Email: {tchen, dkuo, zhzhou}@it.swin.edu.au

[‡]Department of Computer Science and Information Systems

The University of Hong Kong
Pokfulam Road, Hong Kong

Email: tse@csis.hku.hk

Abstract

When testing a program, correctly executed test cases are seldom explored further, even though they may carry useful information. Metamorphic testing proposes to generate follow-up test cases to check important properties of the target function. It does not need a human oracle for output prediction and comparison. In this paper, we highlight the basic concepts of metamorphic testing and some interesting extensions on the areas of program testing, proving, and debugging. Future research directions are also discussed.

Keywords: Follow-up test cases, metamorphic testing, semi-proving, successful test case, test case selection strategy, testing oracle

1. Introduction

It is impractical, if not impossible, to test a program with all conceivable inputs [1]. Instead, we should aim at selecting test cases with higher probabilities of revealing program failures. Hence, a lot of research has been done on developing test case selection strategies.

A *successful test case* is one on which the program computes correctly. Since successful test cases do not reveal any failure, they are conventionally considered useless [22] and thus discarded by testers or merely retained

*This research is supported in part by a discovery grant of the Australian Research Council (Project No. DP045147), a grant of the Research Grants Council of Hong Kong, and a grant of the University of Hong Kong.
†Contact author.

‡A *metamorphic testing* (MT) method has been proposed [4] with a view to making use of the valuable information in successful test cases. It does not depend on the availability of an oracle. It proposes to generate follow-up test cases based on metamorphic relations, or properties among inputs and outputs of the target function. In this

Exploiting Equivalence

Automatic Recovery

Antonio Carzaniga^{*} Alessandra Orsi[†]

^{*}University
Faculty of
Lugano, Sv

Abstract—We present a technique to make application fail-safe. This technique is intended to faulty application functional in the field while it works on permanent and radical fixes. We target applications built on reusable components. In a technique exploits the intrinsic redundancy of those by identifying workarounds consisting of altering the faulty components that avoid the failure. The currently implemented for Java applications but no assumptions about the nature of the application without interrupting the flow of the application's execution of its components. We demonstrate this technique on four mid-size applications and libraries of reusable components affected by real faults. In these cases the technique is effective: the application fully functional with between 1% of the failure-causing faults, depending on the application experiments also show that the technique incurs runtime overhead in all cases.

1. INTRODUCTION

Software systems are sometimes released and found with faults, and those faults may cause field fail-happens despite the best effort and the rigorous developers and testers. Furthermore, even when reported to developers, field failures may take a long time to diagnose and eliminate. As a perhaps extreme but unique example, consider fault n. 3655 in the Firefox which was reported first in March 1999 and only the following ten years, and is yet to be corrected at the time of writing of this paper (summer 2012). The longevity of faults in deployed applications is due to the difficulty of reproducing failures in the environment or more generally to the difficulty of eliminating faults at a cost and with a schedule that meet the objectives of developers and users.

At any rate, dealing with faults that escape environment seems to be a necessity for modern, in fact, several lines of research have been devoted to at least mitigating the effects of faults in deployed software. A primary example is software fault tolerance hardware fault-tolerance techniques such as RAID. Fault tolerance is based on the idea of performing different versions of an application (or as to obtain a correct behavior from the majority, even just one) of the versions [2], [3].

^{*}https://bugzilla.mozilla.org/show_bug.cgi?id=3655

Automatic Workarounds

International

Antonio Carzaniga, Alessandra Orsi

^{*}University
Faculty of
Lugano, Sv

On the Enhancement of BPEL

Antonio Carzaniga¹, Alessandra Orsi², Santanath Bhattacharya³, Sankar Ghosh⁴, Subrata Mukherjee⁵, Gopala Venkateswaran⁶, Gargi Chakrabarti⁷, and Rakesh Goradia⁸
¹INRIA Saclay-Île-de-France, France
²Pescara Univ., Italy
³IBM India R&D, Bangalore, India
⁴ComLab, University of Cambridge, UK
⁵College of Information Technology, University of Alberta, Canada
⁶Singapore Management University, Singapore
⁷INRIA, Paris, France
⁸University of Saarland, Saarbrücken, Germany

ABSTRACT

We present a technique that finds and executes workarounds for faulty Web applications automatically and transparently. Workarounds exploit the inherent redundancy of Web applications, whereby functionality of the system can be obtained through different sequences of calls to Web APIs. In general, runtime workarounds are in response to a failure, and require that the application remain in a consistent state before and after the workaround. Therefore, they are ideally suitable for active Web applications, since those allow the user to act as a failure detector with minimal effort, and to use read-only state or manage their state through actional data stores. In this paper we focus on it in the access libraries of widely used Web APIs as Google Maps. We start by classifying a number of faults in the Google Maps and YouTube services known workarounds. From those we derive a general and API-specific program-rewriting rule that applies to other faults, and which no workaround is available. Our approach is based on the notion of composition, placed within Web applications, through a simple plugin-in, and that program-rewriting rules derive symmetry properties of a common library can be finding valid and previously unknown workarounds.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging; Handling and recovery

General Terms

Reliability, Design

Keywords

Automatic Workarounds, Web Applications, Web Services

^{*}Mano Manzini is also with the University of Milan, Italy

Permission to make digital or hard copies of all or part of this material 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. To copy otherwise, to post on servers or to redistribute to lists, requires permission and/or a fee.

ICSE '14, November 7–11, 2014, Santa Fe, New Mexico, USA

Copyright 2014 ACM 978-1-4503-2756-5/14/05 ...\$15.00.

Cross-Validation from Intrinsic Properties

Metamorphic Testing and Beyond *

T. Chen[†], F.-C. Kuo[†], T. H. Tse^{‡§}, Zhi Quan Zhou[†]
[†]Sydney University of Technology
Sydney, Victoria 3122, Australia
Email: {tchen, zkuo, zhzhou}@it.swin.edu.au

[‡]Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
Email: tse@csis.hku.hk

Manual identification of equivalence

ABSTRACT

Despite the success of many automated tools for cross-validation, there is still a need for manual validation. We argue that a oracle can be used to validate test cases. We propose a technique that leverages more information from the system under test to assume that the system under test is somehow in the sense that some operations are designed like others but their executions are different. One of this and previous work shows that this costs are significantly decreased. We then propose cross-checking the execution of a test with the one which we replace some operations with redundant development this notion of cross-checking oracles is a technique to automatically insert oracles into experimental evaluation shows that cross-checking used in combination with automatic test generation can be very effective in revealing faults they can even improve good hand-written test

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program testing; D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Verification

Keywords

Redundancy, test oracles, oracle generation

Permission to make digital or hard copies of all or part of this material 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. To copy otherwise, to post on servers or to redistribute to lists, requires permission and/or a fee.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

Copyright 2014 ACM 978-1-4503-2756-5/14/05 ...\$15.00.

Abstract

When testing a program, correctly executed test cases are seldom explored further, even though they may carry useful information. Metamorphic testing proposes to generate follow-up test cases to check important properties of the target function. It does not need a human oracle for output prediction and comparison. In this paper, we highlight the basic concepts of metamorphic testing and some interesting extensions in the areas of program testing, proving, and debugging. Future research directions are also proposed.

Keywords: Follow-up test cases, metamorphic testing, semi-proving, successful test case, test case selection strategy, testing oracle

1. Introduction

It is impractical, if not impossible, to test a program with all conceivable inputs [1]. Instead, we should aim at selecting test cases with higher probabilities of revealing program failures. Hence, a lot of research has been done on developing test case selection strategies.

A *successful test case* is one on which the program computes correctly. Since successful test cases do not reveal any failure, they are conventionally considered useless [2,2] and thus discarded by testers or merely retained

This research is supported in part by a discovery grant of the Australian Research Council (Project No. DP0345147), a grant of the Research Grants Council of Hong Kong, and a grant of the University of Hong Kong.

*Contact author.

Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'04)
0-7695-2218-1/04 \$20.00 © 2004 IEEE



Exploiting Equivalence

Automatic Recovery

Antonio Carzaniga^{*} Alessandro Orsi[†]

^{*}University
Faculty of
Lugano, Sv

Abstract—We present a technique to make application fault-tolerant. This technique is intended to faulty application functional in the field while it works on permanent and radical fixes. We target applications built on reusable components. In this technique exploits the intrinsic redundancy of those by identifying *workarounds* consisting of alternates the faulty components that avoid the failure. The currently implemented for Java applications but it makes no assumptions about the nature of the application without interrupting its normal flow of the application running on its components. We demonstrate this technique on four mid-size applications and libraries of reusable components affected by real faults. In these cases the technique is effective: the application fully functional with between 1% of the failure-causing faults, depending on the application experiments also show that the technique incurs runtime overhead in all cases.

1. INTRODUCTION

Software systems are sometimes released and found with faults, and those faults may cause field fail-happens despite the best effort and the rigorous developers and testers. Furthermore, even when reported to developers, field failures may take a long time to diagnose and eliminate. As a perhaps extreme but unique example, consider fault n. 3655 in the Firefox which was reported first in March 1999 and only the following ten years, and is yet to be corrected at the time of writing of this paper (summer 2012). The longevity of faults in deployed applications is due to the difficulty of reproducing failures in the environment or more generally to the difficulty of eliminating faults at a cost and with a schedule that matches the objectives of developers and users.

At any rate, dealing with faults that escape environment seems to be a necessity for modern software systems. In fact, several lines of research have been devoted to at least mitigating the effects of faults in deployed systems. A primary example is software fault tolerance: hardware fault-tolerance techniques such as RAID are fault tolerance is based on the idea of parallelly executing different versions of an application (or as to obtain a correct behavior from the majority, even just one) of the versions [2], [3].

^{*}https://bugzilla.mozilla.org/show_bug.cgi?id=3655

Automatic Workarounds

Antonio Carzaniga, Alessandro Orsi

^{*}University
Faculty of
Lugano, Sv

On the Enhancement of BPEL

Santanath Bhattacharya[‡], Antonio Carzaniga[§], Alessandro Orsi[†]

[‡]INRIA Saclay-Orsay
[§]Pescara, Univ.
[†]IBM India R&D
[§]ComLab,
[†]College of Information Technol-

Cross-Testing from Intrinsic Equivalence

Anton A. Chen[†], F.-C. Kuo[†], T. H. Tse^{‡§}, Zhi Quan Zhou[†]

[†]Singapore University of Technology
[‡]Hong Kong, Victoria 3122, Australia
Email: {achen, akuo, zhzhou}@it.swin.edu.au

[§]Department of Computer Science and Information Systems

The University of Hong Kong

Poole Road, Hong Kong

Email: tse@csis.hku.hk

ABSTRACT

We present a technique that finds and executes workarounds for faulty Web applications automatically and transparently. Our technique exploits the intrinsic redundancy of Web applications, whereby functionality of the system can be obtained through different sequences of calls to Web APIs. In general, runtime workarounds are triggered in response to a failure, and require that the application remain in a consistent state before and after the workaround. Therefore, they are ideally suitable for active Web applications, since those allow the user to act as a failure detector with minimal effort, and to use read-only state or manage their state through actioned data stores. In this paper we focus on it in the access libraries of widely used Web applications as Google Maps. We start by classifying a number of faults of the Google Maps and YouTube known workarounds. From those we derive a general and API-specific program-rewriting rule that applies to other faults, and which no workaround. Our technique generates a new workaround that is placed within Web applications, through a simple plugin, and that program-rewriting rules derive memory properties of a common library can be finding valid and previously unknown workarounds.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Handling and review

General Terms

Reliability, Design, Languages, Performance, Tools, Web

*Mauro Pezzè is also with the University of Milan

Manual identification of equivalence

ABSTRACT

Despite the success of automated test case generation, there is still a need for manual intervention in the process of generating test cases. One reason is that automated tools often generate redundant test cases, which degrades the efficiency of the test generation process. Another reason is that automated tools often generate test cases that are not useful for testing the system under test. In this paper, we propose a technique for cross-checking the execution of a test with the source code. We replace some operations with redundant ones and then use them to cross-check the execution of a test with the source code. We then use the source code to develop this notion of cross-checking oracles in order to automatically insert oracles into the test cases. Experimental evaluation shows that cross-checking used in combination with automatic test generation can be very effective in revealing faults in the system. They can even improve good hand-written test cases.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Testing and Debugging—D.2.2 [Software Engineering]: Testing and Debugging—D.2.1 [Software Engineering]: Testing and Debugging—D.2.3 [Software Engineering]: Testing and Debugging—D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Reliability, Design, Languages, Performance, Tools, Web

Keywords

Redundancy, test oracles, oracle generation

... is the main cost!

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that: (1) copies are not made or distributed for profit or commercial advantage; (2) copies are not sold in whole or in part; (3) copies bear this notice and the full citation on the first page; (4) copies are not made or distributed outside the participant's organization; (5) copies are not posted on servers or to redistribute to lists, require permission and/or a fee.

ICSE '14, November 7–11, 2014, Santa Fe, New Mexico, USA

Copyright 2014 ACM 978-1-4503-2756-5/14/05 ...\$15.00.

for reuse in regression testing later. We note, however, that successful test cases do carry useful information, albeit seldom explored. Fault-based testing [21], for example, is a significant attempt to make use of such information. In fault-based testing, if a program has successfully passed all the test cases, then it can be guaranteed to be free from certain types of faults. Unfortunately, most testing methods are not fault-based, and most test cases are executed successfully. Thus, some valuable information that results from program testing will remain buried and unused.

Another limitation of software testing is the *oracle problem* [23]. An oracle is a mechanism against which people can decide whether the outcome of the program on test cases is correct. In some situations, the oracle is not available or is too expensive to be applied [23]. In cryptography systems, for example, large number arithmetic is usually involved. It is very expensive to verify the correctness of a computed result. Other examples include establishing the equivalence between the source and target code when testing a compiler, and deciding the correctness of an output when testing a program that performs numerical integration. Furthermore, even when perfect, the selection and comparison of testing results are time-consuming. They are often time consuming and error-prone [18, 20]. The oracle problem is “one of the most difficult tasks in software testing” [20] but is often ignored in the testing theory [18].

A *metamorphic testing* (MT) method has been proposed [4] with a view to making use of the valuable information in successful test cases. It does not depend on the availability of an oracle. It proposes to generate follow-up test cases based on metamorphic relations, or properties among inputs and outputs of the target function. In this

Automatic Synthesis of Equivalent Method Sequences



Automatic Synthesis of Equivalences

Java™
Stack
pop()



```
int el = s.peek();  
int index = s.size();  
index = index - 1;  
s.remove(index);  
return el;
```

Automatic Synthesis of Equivalences

Java™
Stack
`pop()`



```
int el = s.peek();  
int index = s.size();  
index = index - 1;  
s.remove(index);  
return el;
```

Execution scenarios

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
Object ret = s.pop();
```

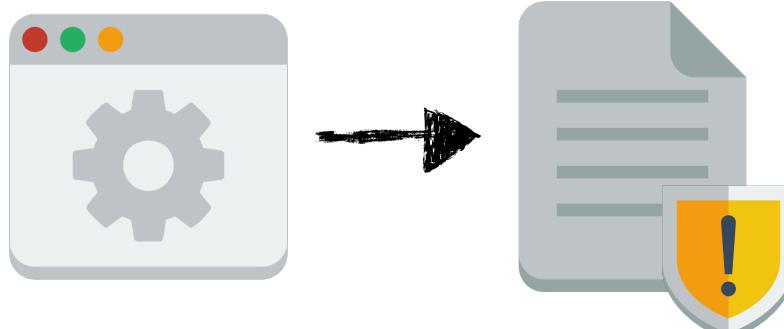
```
Stack s = new Stack();  
s.push(-4);  
Object ret = s.pop();
```

Automatic Synthesis of Equivalences



Stack
pop()

Synthesis



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

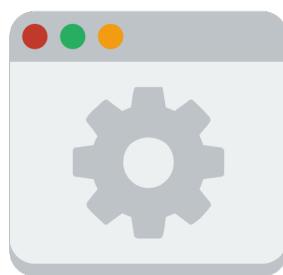
```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Automatic Synthesis of Equivalences

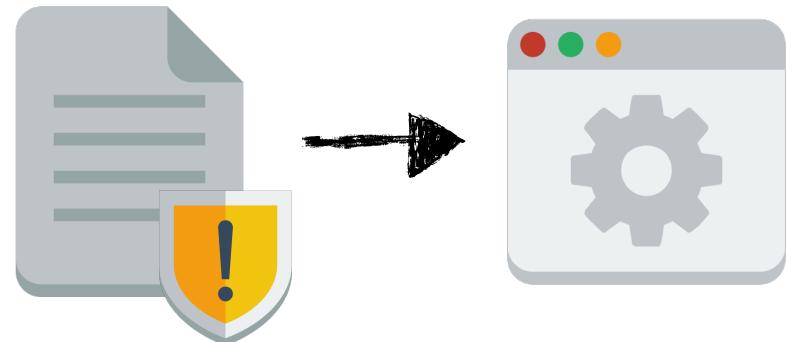


Stack
pop()

Synthesis



Counterexample



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Automatic Synthesis of Equivalences

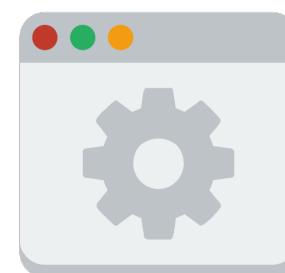


Stack
pop()

Synthesis



Counterexample



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Automatic Synthesis of Equivalences

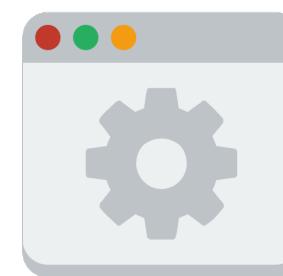


Stack
pop()

Synthesis



Counterexample



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

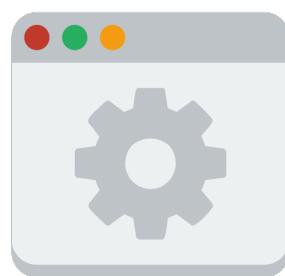
```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Automatic Synthesis of Equivalences

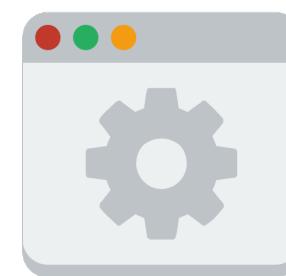


Stack
pop()

Synthesis



Counterexample



Timeout!



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Automatic Synthesis of Equivalences

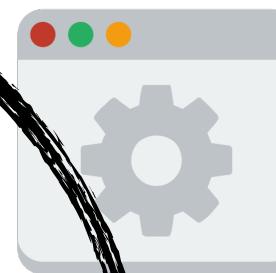


Stack
pop()

Synthesis



Counterexample



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Timeout!

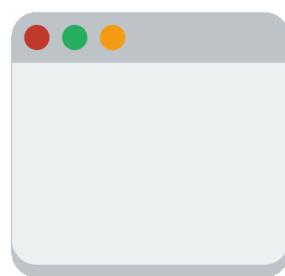


Search-based Synthesis of Equivalences

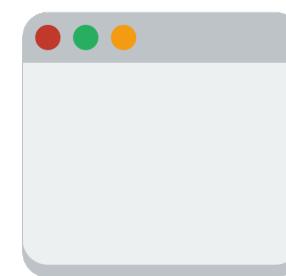


Stack
pop()

Synthesis



Counterexample



Timeout!



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

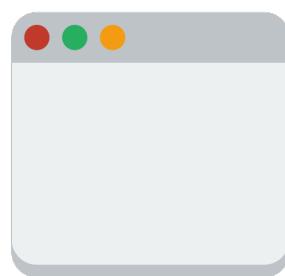
```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Search-based Synthesis of Equivalences

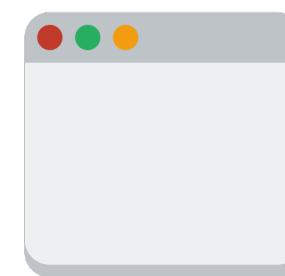


Stack
pop()

EVASUITE



EVASUITE



Timeout!



Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```



Equivalence Synthesis as TCG Problem



```
public void method_under_test() {  
    if ( condition ) {  
        // equivalent!  
    }  
}
```

Equivalence Synthesis as TCG Problem



```
public void method_under_test() {  
    if (condition) {  
        // equivalent!  
    }  
}
```

On **all** execution scenarios:

- compute **identical** results
- lead to **identical** object states

Equivalence Synthesis as TCG Problem



↑
pop()

Java™
Stack

```
public void methodUnderTest() {  
    if ( condition ) {  
        // equivalent!  
    }  
}
```

Execution scenarios

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

Equivalence Synthesis as TCG Problem



↑
pop()



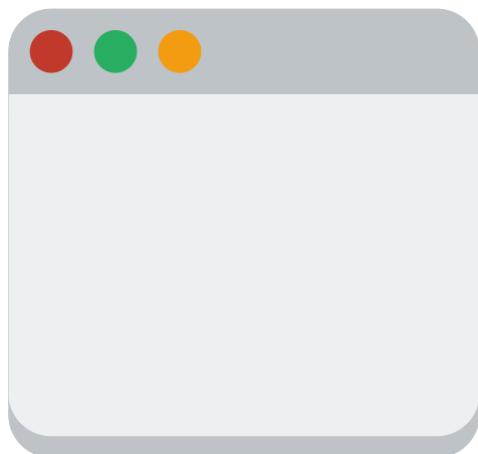
```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = ? ? ? ? ? ;
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = ? ? ? ? ? ;
```

Equivalence Synthesis as TCG Problem



↑
pop()


Java™
Stack

```
public void method_under_test() {  
    if ( [green box] == [orange box] && [blue box] == [purple box] && [yellow box] == [pink box] ) {  
        [green box]  
    }  
}
```

// equivalent!

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = ? ? ? ? ? ;
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = ? ? ? ? ? ;
```

Equivalence Synthesis as TCG Problem



↑
pop()

 Java™
Stack

```
public void method_under_test() {  
    if ( [green] == [orange] && [blue] == [purple] &&
```

```
        [green] == [orange] && [blue] == [purple] ) {  
            // equivalent!
```

```
}
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = ? ? ? ? ? ;
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = ? ? ? ? ? ;
```

Equivalence Synthesis as TCG Problem



```
public void method_under_test() {  
    if ( [green] == [orange] && [blue] == [purple] &&  
        [green] == [orange] && [blue] == [purple] ) {  
        // equivalent!  
    }  
}
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = ? ? ? ? ? ;
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = ? ? ? ? ? ;
```

Equivalence Synthesis as TCG Problem



remove(0)



```
public void method_under_test() {  
    if ( [green] == [orange] && [blue] == [purple] &&  
        [green] == [orange] && [blue] == [purple] ) {  
        // equivalent!  
    }  
}
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = ? ? ? ? ? ;
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = ? ? ? ? ? ;
```

Equivalence Synthesis as TCG Problem



↓
remove(0)



```
public void method_under_test() {  
    if ( [green] == [orange] && [blue] == [purple] &&  
        [green] == [orange] && [blue] == [purple] ) {  
        // equivalent!  
    }  
}
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.remove(0);
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.remove(0);
```

Equivalence Synthesis as TCG Problem



↓
remove(0)



```
public void method_under_test() {  
    if ([1] == [1] && [] == [] &&  
        1 == 1 && -4 == -4 ){  
        // equivalent!  
    }  
}
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(-4);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(1);  
s.push(1);  
int ret = s.remove(0);
```

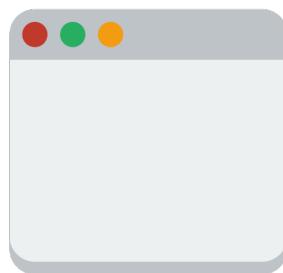
```
Stack s = new Stack();  
s.push(-4);  
int ret = s.remove(0);
```

Search-based Synthesis of Equivalences

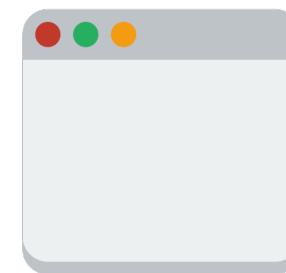


Stack
pop()

Synthesis



Counterexample



remove(0)

Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```

Counterexample as TCG Problem



```
public void methodUnderTest() {  
    if ( condition ) {  
        // counterexample  
    }  
}
```

Counterexample as TCG Problem



On **one** execution scenario:

- compute **different** results, or
- lead to **different** object states

Counterexample as TCG Problem



↑
remove(0)



```
public void methodUnderTest() {  
    if ( condition ) {  
        // counterexample  
    }  
}
```

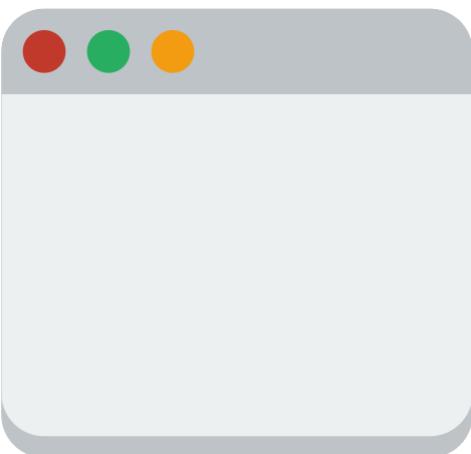
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.pop();
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.remove(0);
```

Counterexample as TCG Problem



↑
remove(**0**)



```
public void methodUnderTest() {  
    if ( green box != orange box ||  
        != ) {  
    }  
}  
// counterexample
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.pop();
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.remove(0);
```

Counterexample as TCG Problem



remove(0)



Stack

```
public void methodUnderTest() {  
    if ( [green box] != [orange box] ||  
        [green box] != [red box] ) {  
    }  
}
```

// counterexample

```
? ? ? ? ? ? ? ? ? ?  
? ? ? ? ? ? ? ? ? ?  
? ? ? ? ? ? ? ? ? ?  
  
int ret = s.pop();
```

```
? ? ? ? ? ? ? ? ? ?  
? ? ? ? ? ? ? ? ? ?  
? ? ? ? ? ? ? ? ? ?  
  
int ret = s.remove(0);
```

Counterexample as TCG Problem



```
public void method_under_test() {  
    if ( [green] != [orange] ||  
        [green] != [red] ) {  
        // counterexample  
    }  
}
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.pop();
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.remove(0);
```

Counterexample as TCG Problem



```
public void methodUnderTest() {  
    if ( [green] != [orange] ||  
        [green] != [red] ) {  
        // counterexample  
    }  
}
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.pop();
```

?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?

```
int ret = s.remove(0);
```

```
Stack s = new Stack();  
s.push(0);  
s.push(1);
```

Counterexample as TCG Problem



```
public void method_under_test() {  
    if ( [green] != [orange] ||  
        [green] != [red] ) {  
        // counterexample  
    }  
}
```

```
Stack s = new Stack();  
s.push(0);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(0);  
s.push(1);  
int ret = s.remove(0);
```

```
Stack s = new Stack();  
s.push(0);  
s.push(1);
```

Counterexample as TCG Problem



```
public void method_under_test() {  
    if ([0] != [1] ||  
        1 != 0) {  
        // counterexample  
    }  
}
```

```
Stack s = new Stack();  
s.push(0);  
s.push(1);  
int ret = s.pop();
```

```
Stack s = new Stack();  
s.push(0);  
s.push(1);  
int ret = s.remove(0);
```

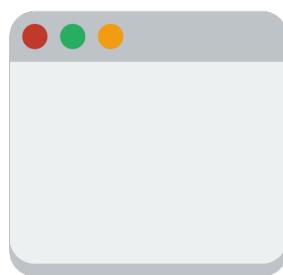
```
Stack s = new Stack();  
s.push(0);  
s.push(1);
```

Search-based Synthesis of Equivalences

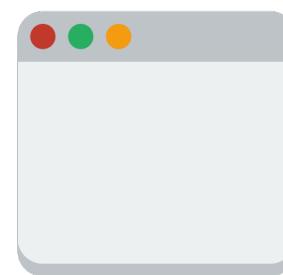


Stack
pop()

Synthesis



Counterexample



remove(0)

Execution scenarios

```
Stack s = new Stack();
s.push(1);
s.push(1);
Object ret = s.pop();
```

```
Stack s = new Stack();
s.push(-4);
Object ret = s.pop();
```



```
Stack s = new Stack();
s.push(0); s.push(1);
```

Search-based Synthesis of Equivalences

SBES



Search-based Synthesis of Equivalences



Stack

`clear()`

Execution scenario

SBES



Search-based Synthesis of Equivalences

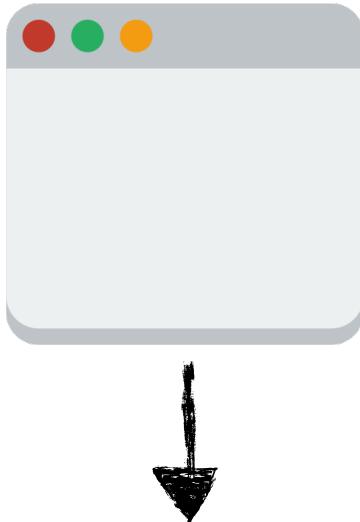


Stack

`clear()`



SBES



- `removeAllElements()`

Search-based Synthesis of Equivalences

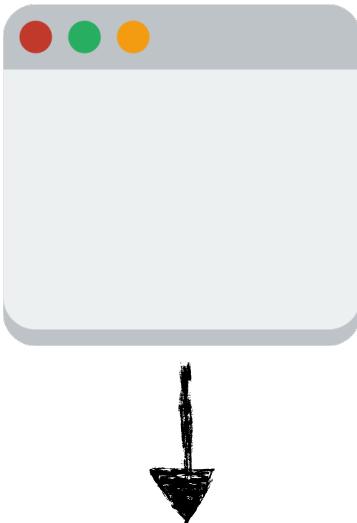


Stack

`clear()`

Execution scenario

SBES



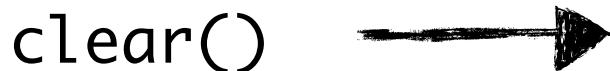
- `removeAllElements()`
- `setSize(0)`

Search-based Synthesis of Equivalences

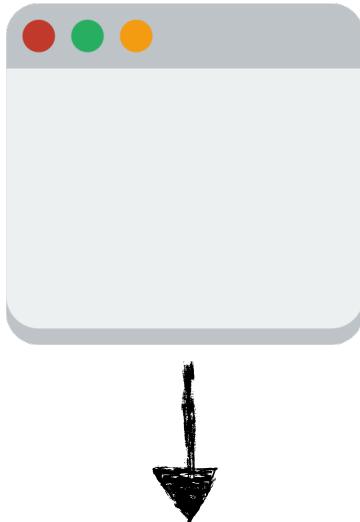


Stack

`clear()`



SBES



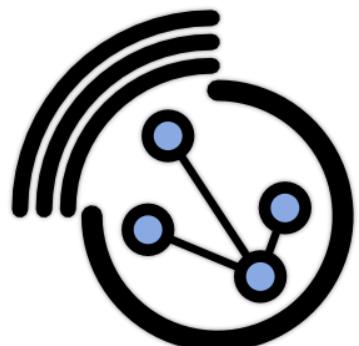
- `removeAllElements()`
- `setSize(0)`
- `Collection c = new Collection();
retainAll(c)`

Evaluation

Evaluation



java.util.Stack



GraphStream

graphstream.Path
graphstream.Edge
graphstream.Node
graphstream.MultiNode
graphstream.Vector2
graphstream.Vector3

Evaluation



java.util.Stack

pop()

remove(size() - 1)

Evaluation



java.util.Stack

pop()

remove(size() - 1)

push(0); pop();
remove(size() - 1)

Evaluation



java.util.Stack

pop()

remove(size() - 1)

push(0); pop();

remove(size() - 1)

push(0); pop(); push(0); pop();

remove(size() - 1)

...

Evaluation



java.util.Stack

pop()

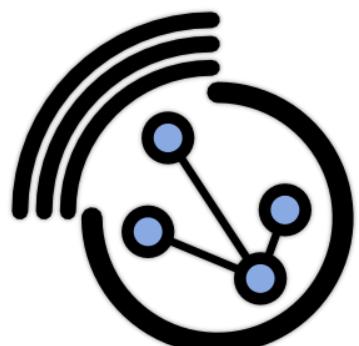
remove(size() - 1)

push(\emptyset); pop();
remove(size() - 1)

push(\emptyset); pop(); push(\emptyset); pop();
remove(size() - 1)

...

Evaluation



GraphStream

	# methods	# eqs
java.util.Stack	15	45
graphstream.Path	2	5
graphstream.Edge	9	20
graphstream.Node	5	12
graphstream.MultiNode	5	12
graphstream.Vector2	5	21
graphstream.Vector3	6	22
	<hr/>	
	47	137

Evaluation

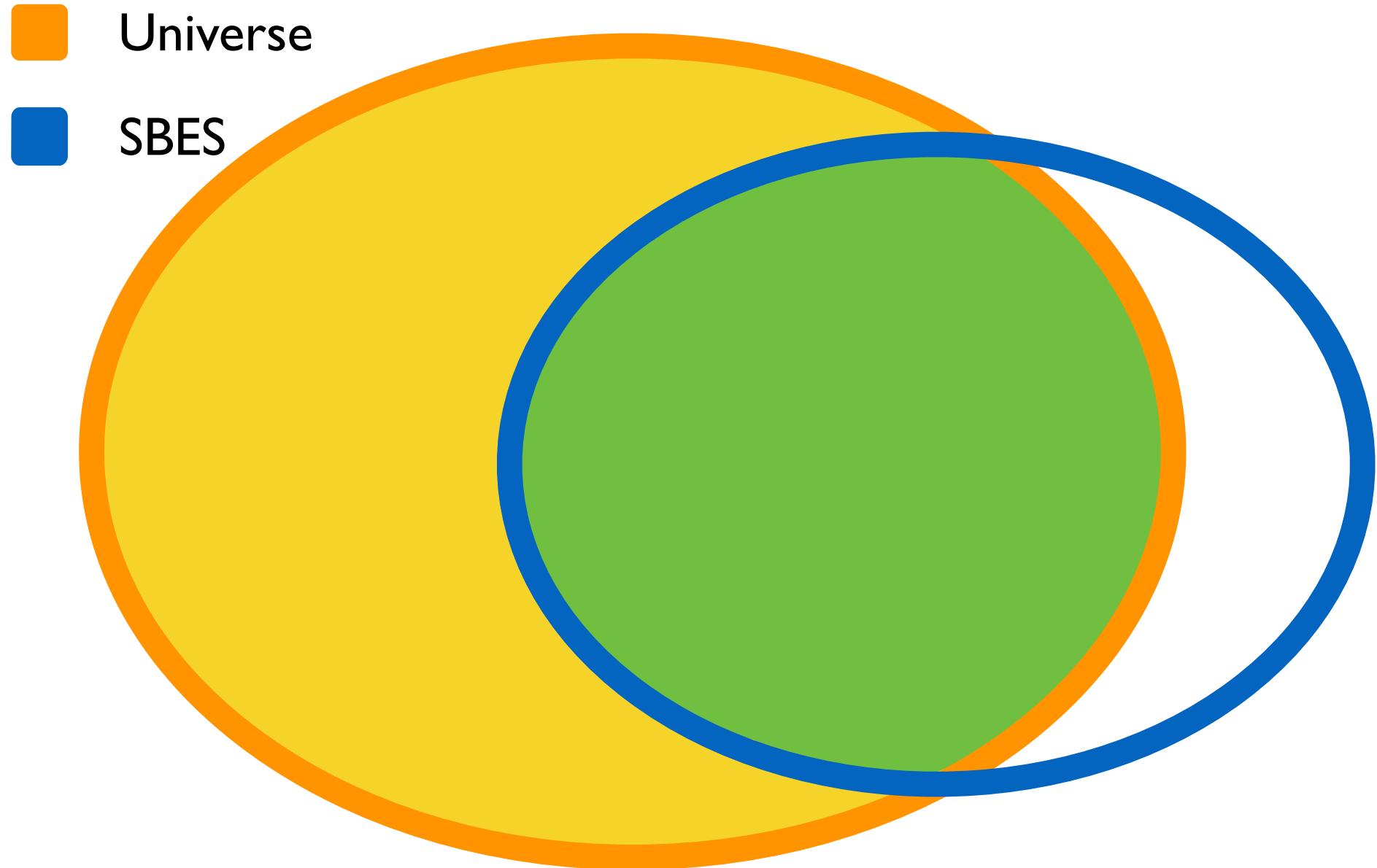
How **effective** is the search-based synthesis?

How **efficient** is the search-based synthesis?

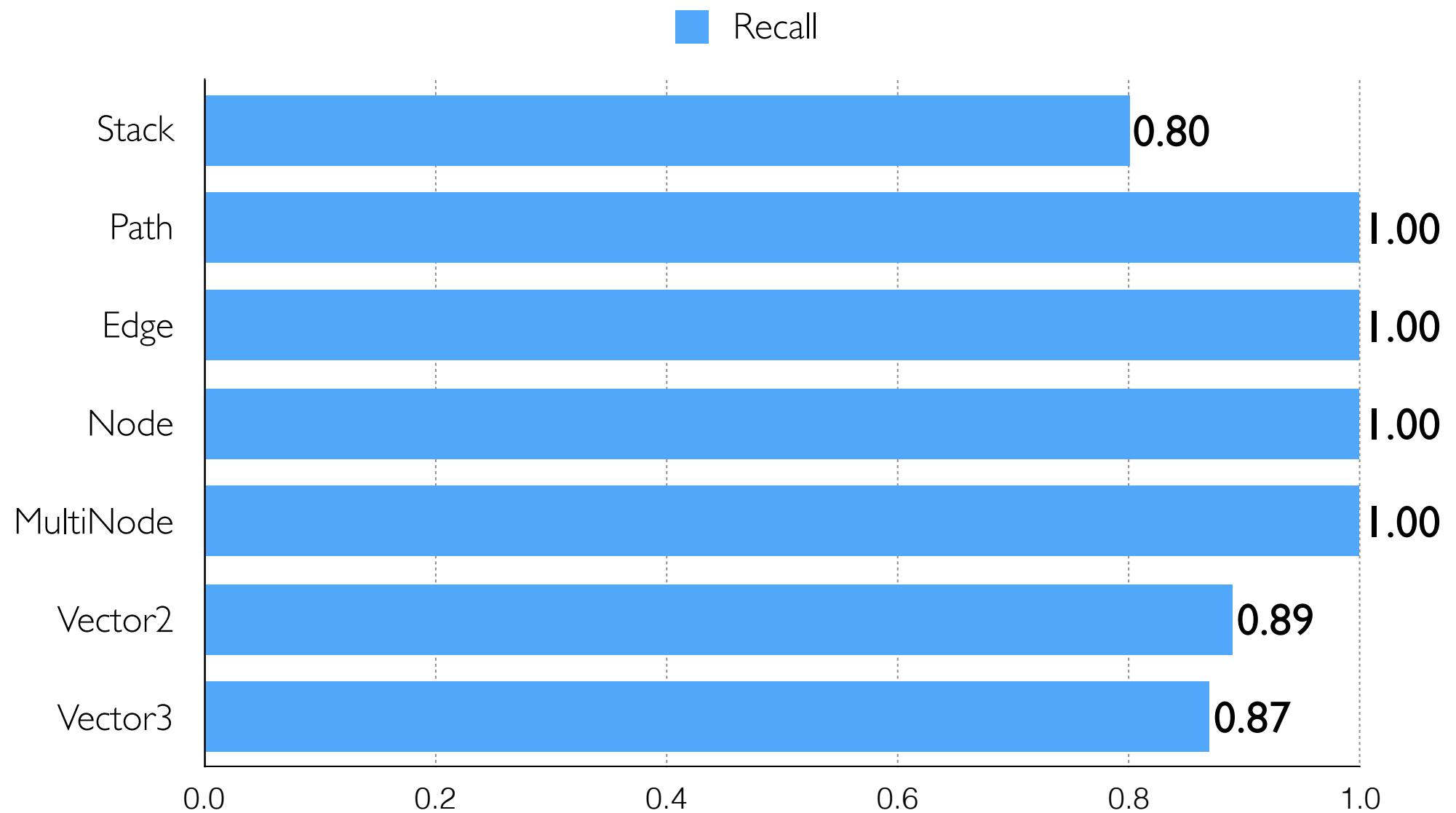
Evaluation

How **effective** is the search-based synthesis?

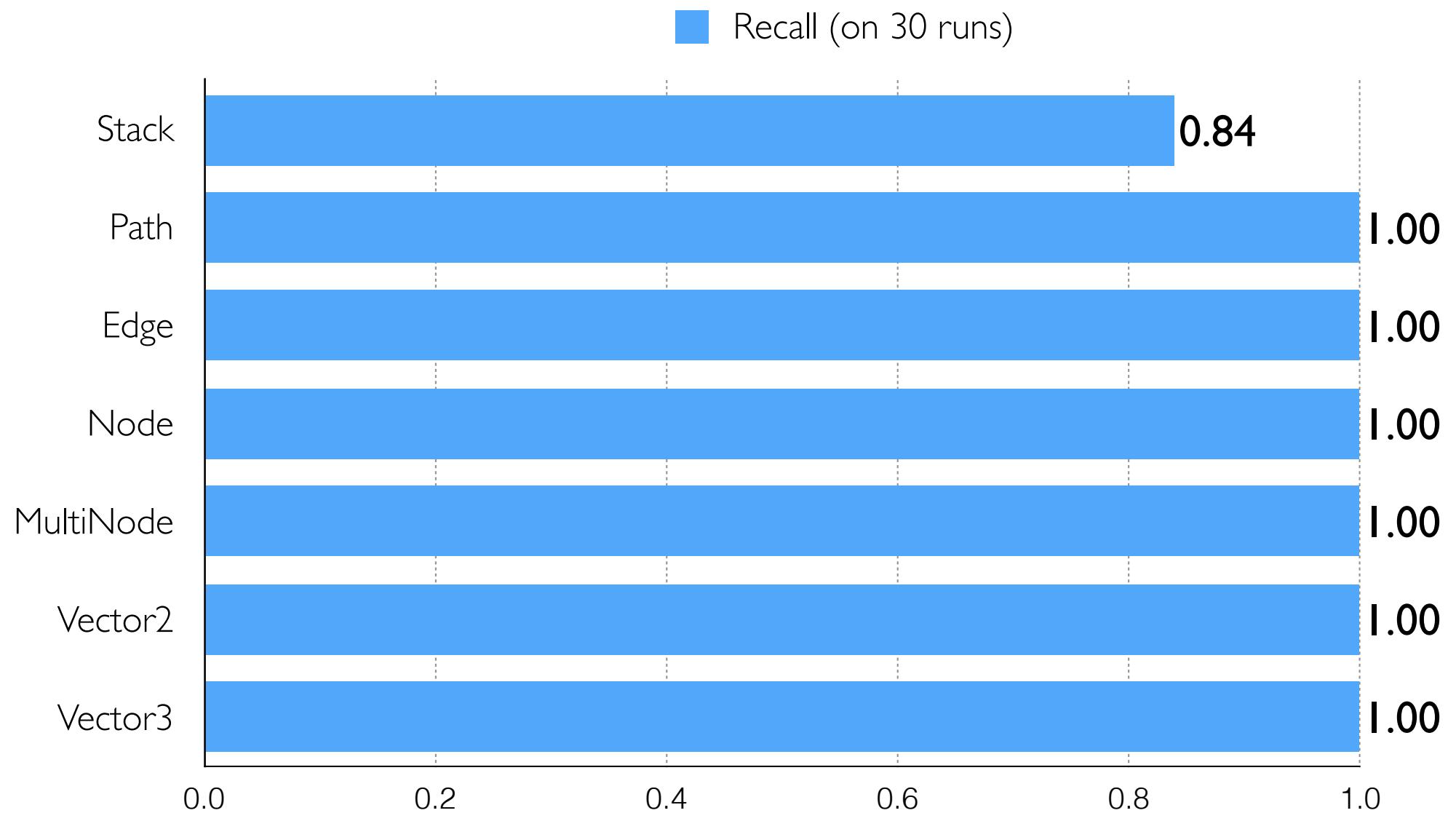
Recall



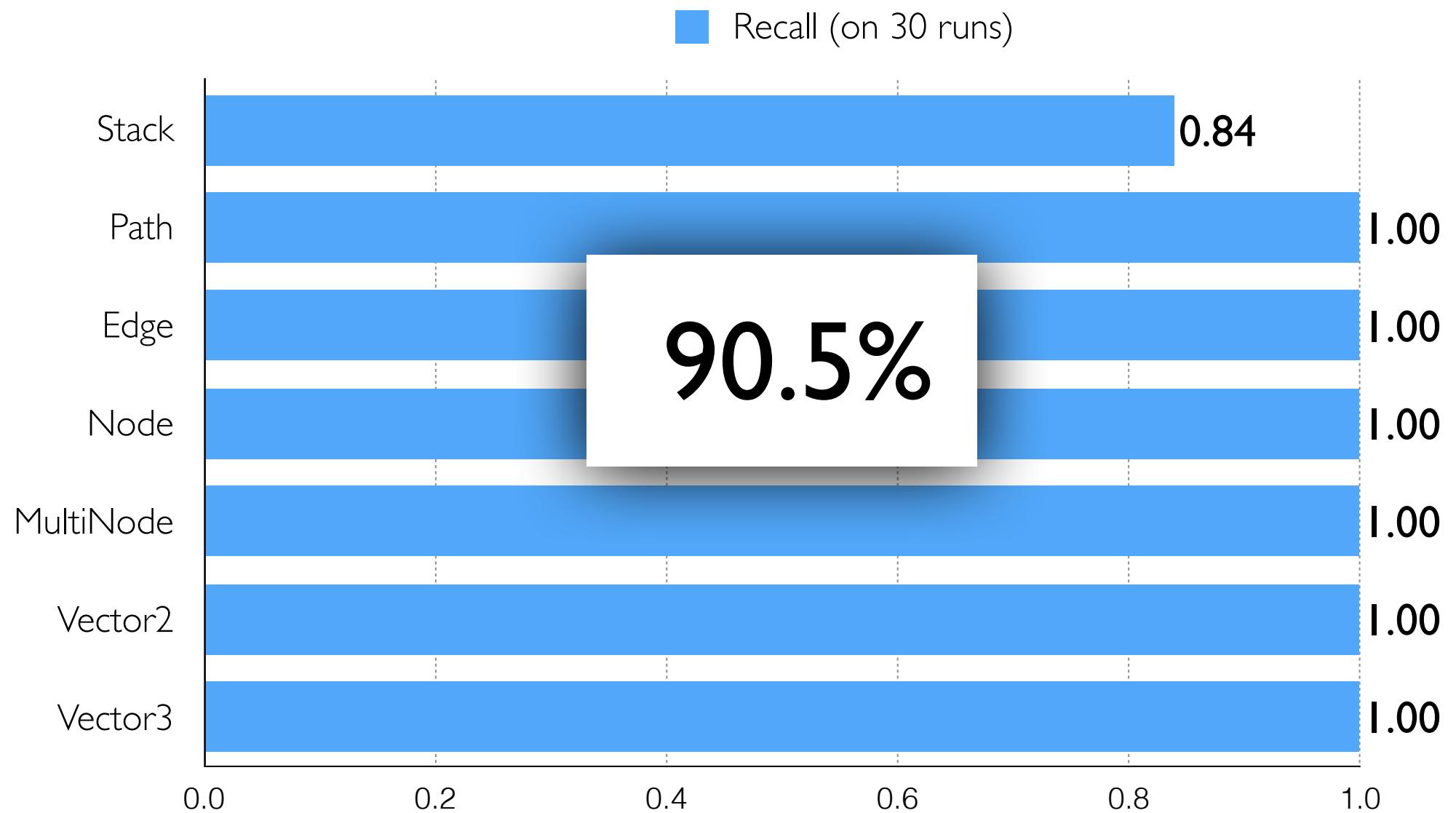
Effectiveness of Search-based Synthesis



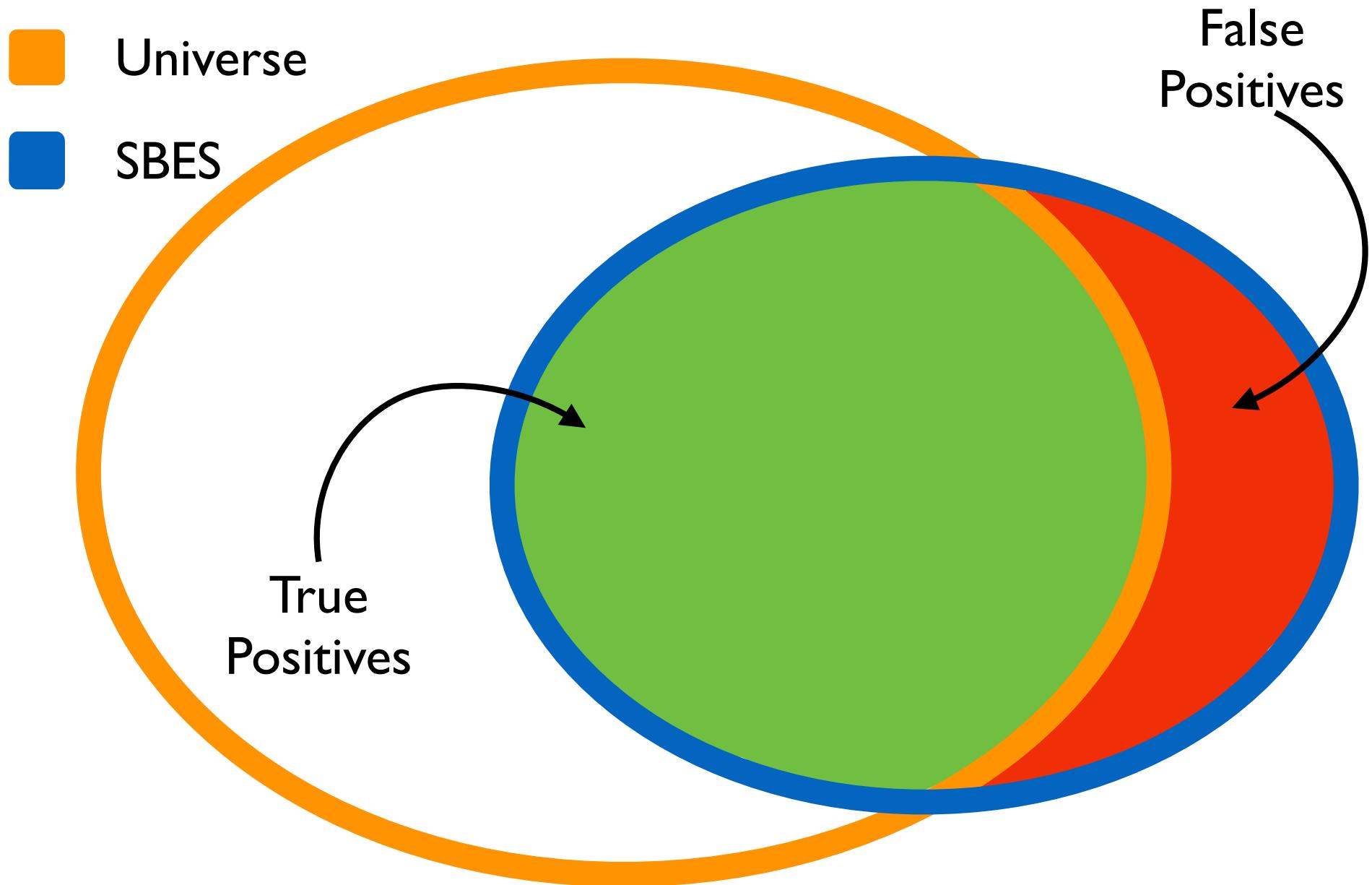
Effectiveness of Search-based Synthesis



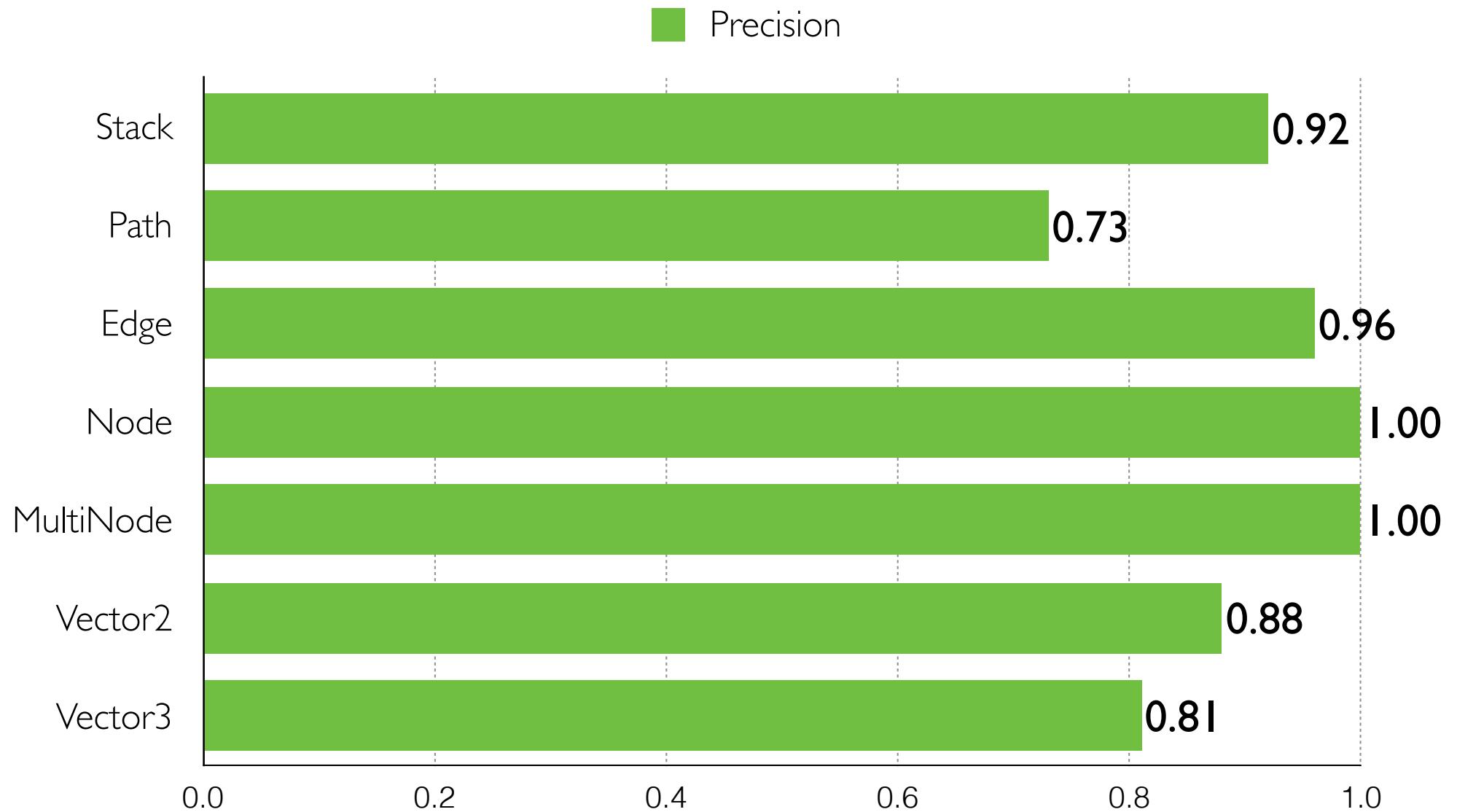
Effectiveness of Search-based Synthesis



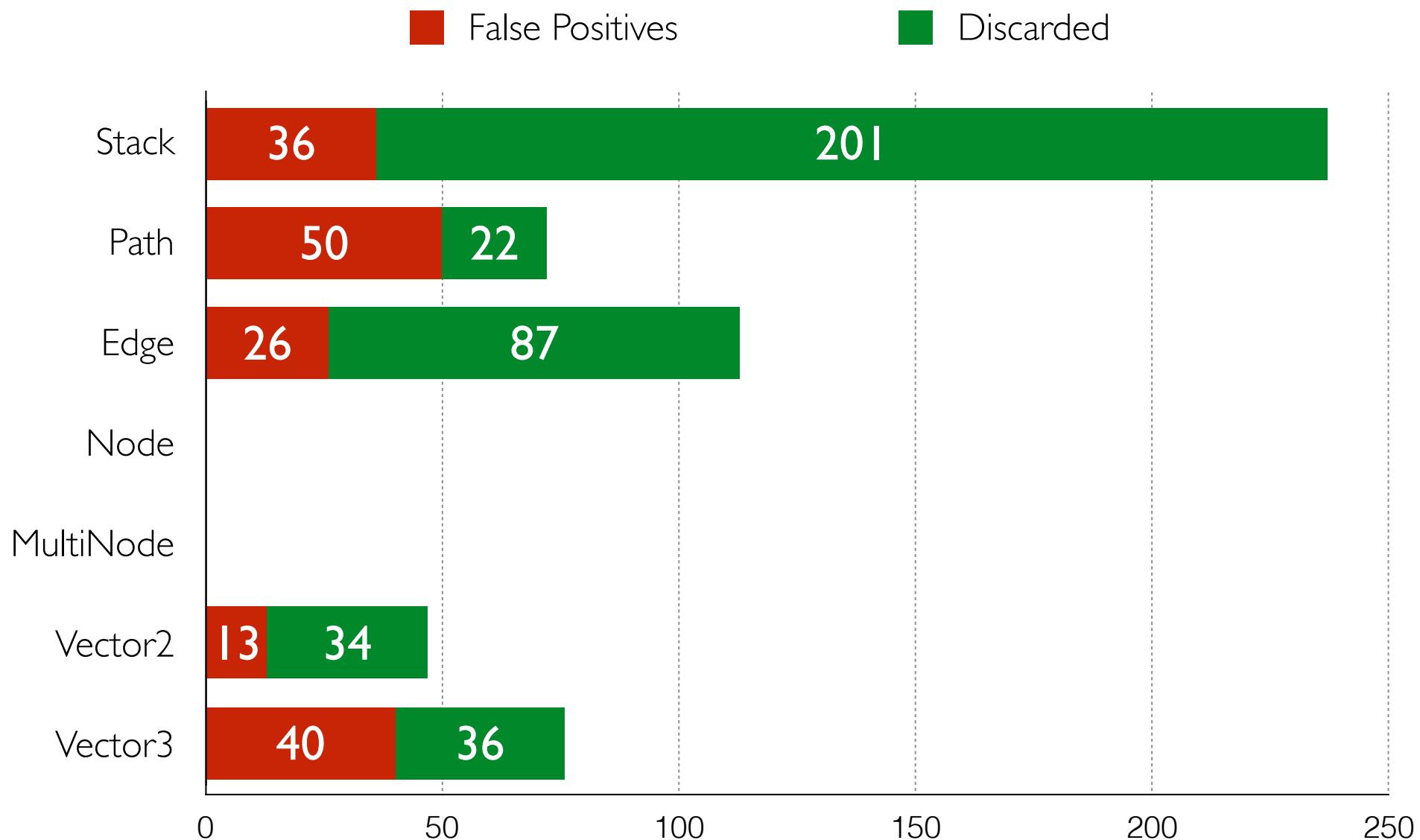
Precision



Effectiveness of Search-based Synthesis



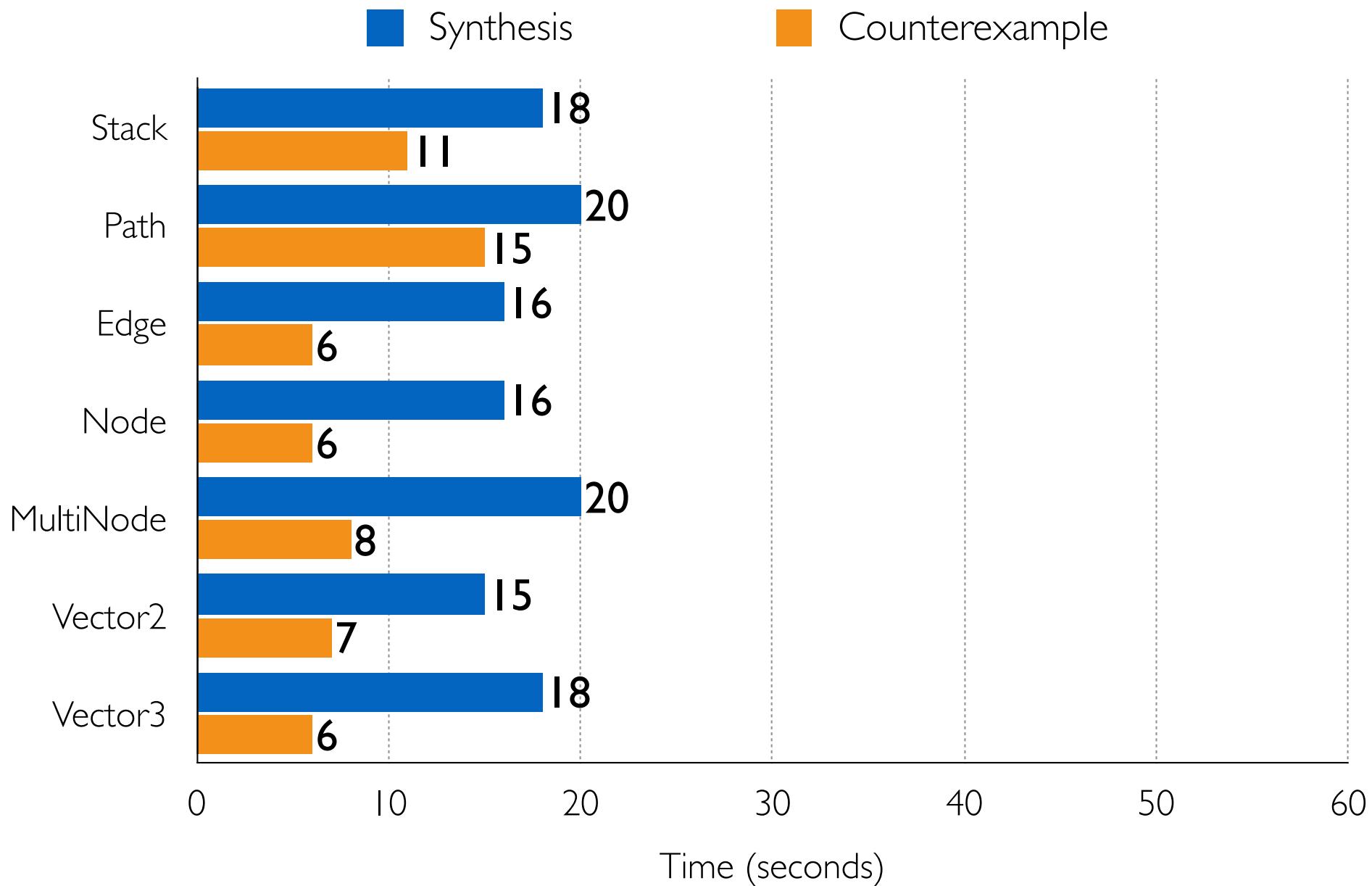
Effectiveness of Counterexamples



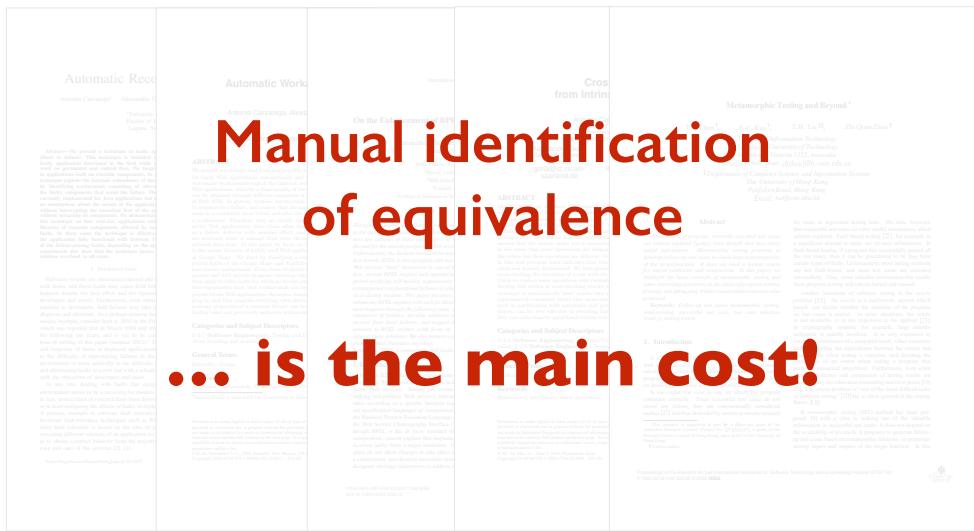
Evaluation

How **efficient** is the search-based synthesis?

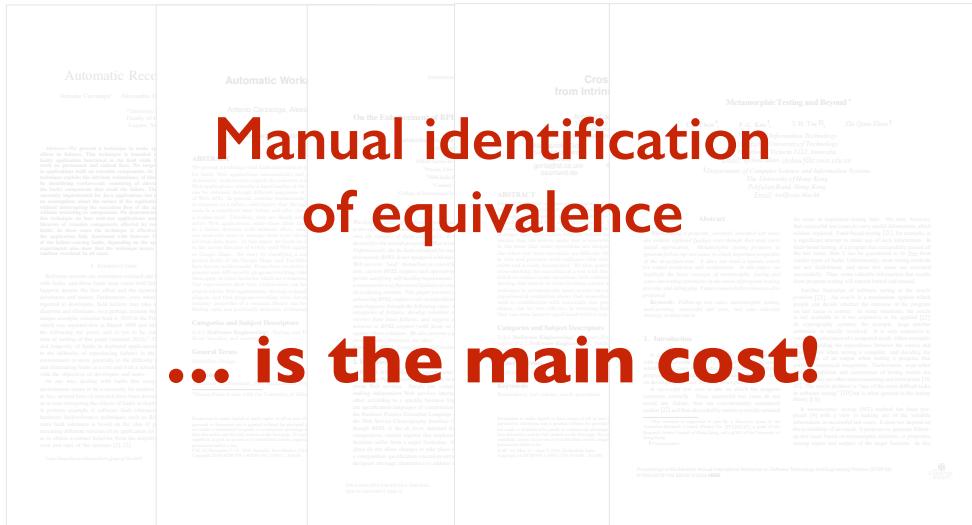
Efficiency of Search-based Synthesis



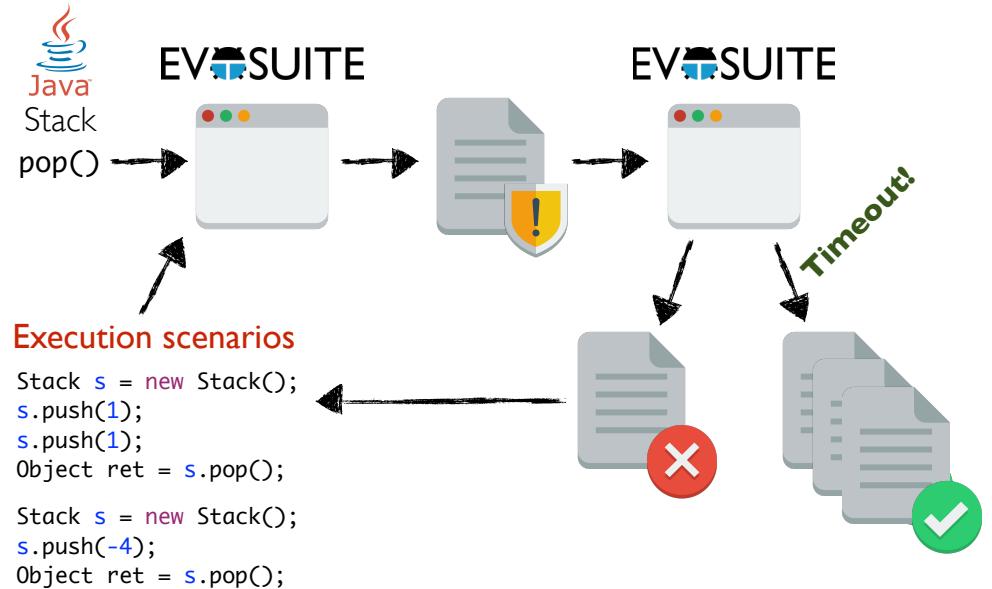
Exploiting Equivalence



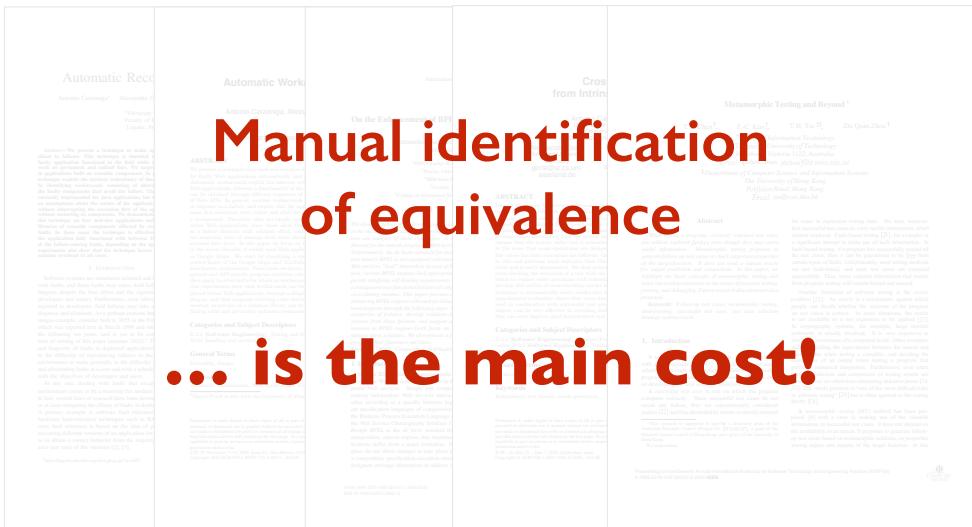
Exploiting Equivalence



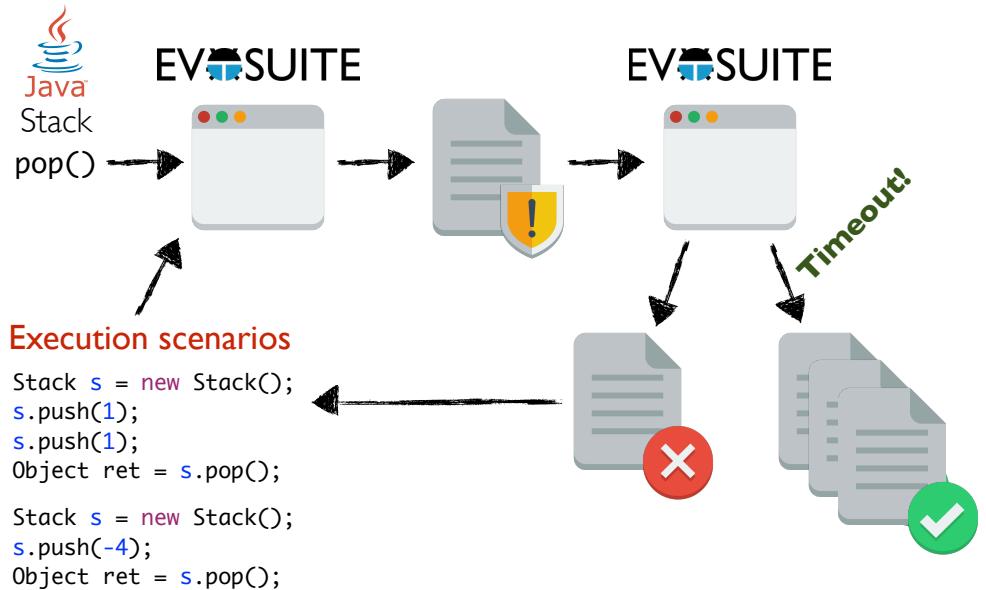
Search-based Synthesis of Equivalences



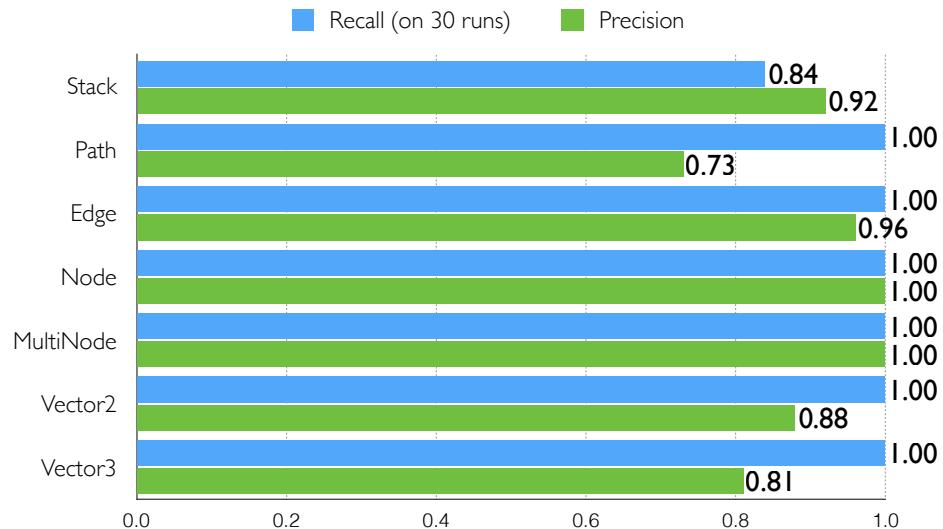
Exploiting Equivalence



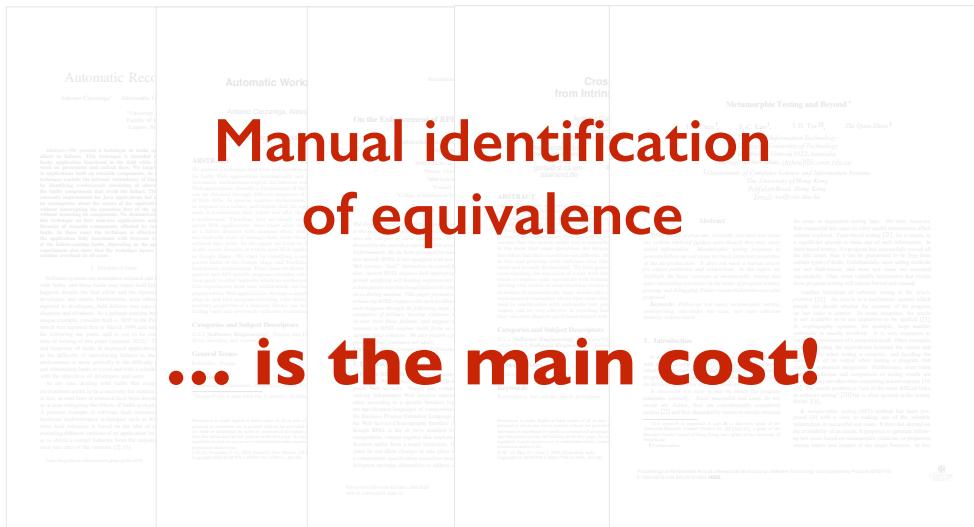
Search-based Synthesis of Equivalences



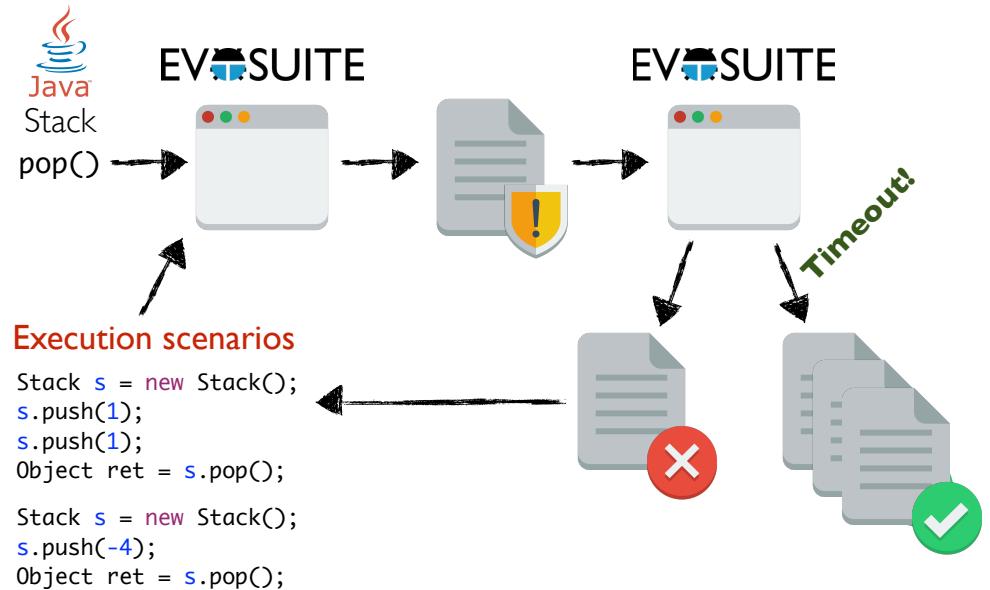
Effectiveness of Search-based Synthesis



Exploiting Equivalence



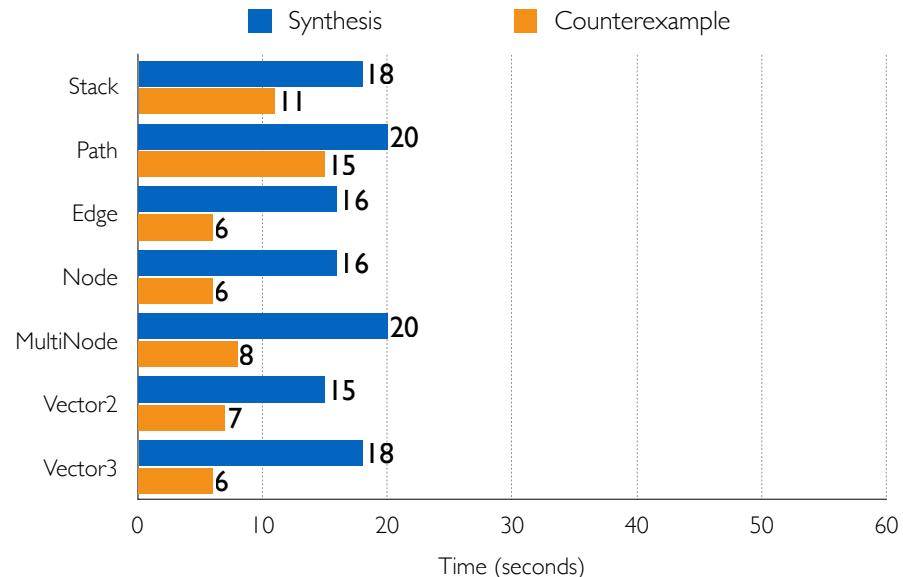
Search-based Synthesis of Equivalences



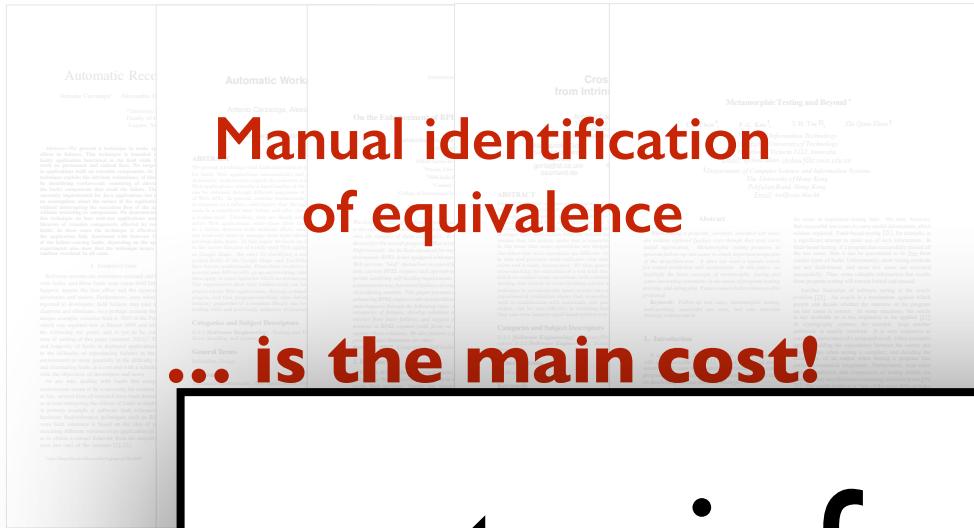
Effectiveness of Search-based Synthesis



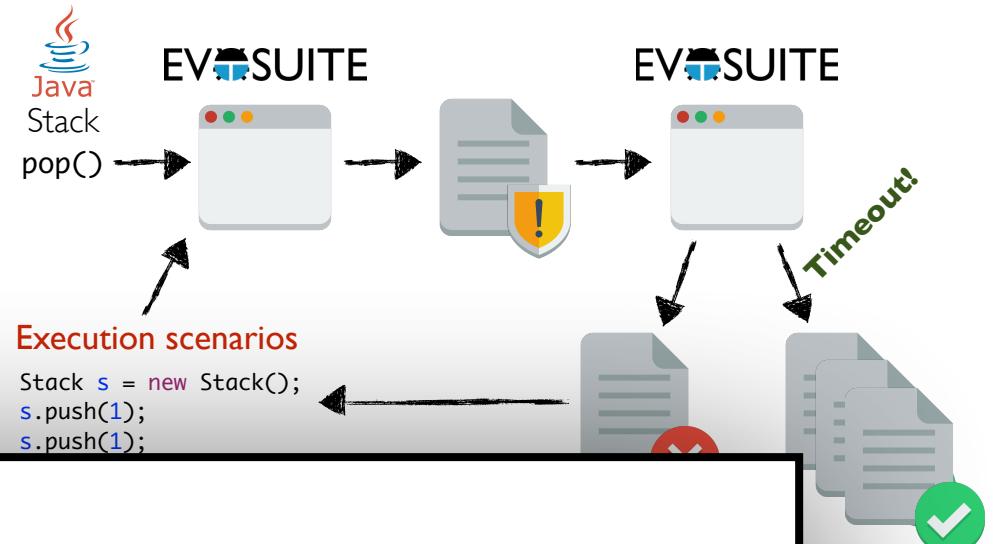
Efficiency of Search-based Synthesis



Exploiting Equivalence



Search-based Synthesis of Equivalences



star.inf.usi.ch/sbes

Effectiv

thesis

