

Символьное исполнение в .NET

Автоматическое тестирование, верификация и синтез программ

Dmitry Ivanov, *Huawei Saint Petersburg Research Center*, korifey@gmail.com,
@korifey_ad

Dmitry Mordvinov, *JetBrains Research, SPbU*, mordvinov.dmitry@gmail.com

Что мы сегодня узнаем?

- [Часть первая, **продуктовая**] – Зачем нужно генерировать тесты?
- [Часть вторая, **теоретическая**] – Как работает символьное исполнение?
- [Часть третья, **практическая**] – Как использовать SMT-солвер?
- [Часть четвёртая, **демонстрационная**] – Символьная виртуальная машина V#
- [Часть пятая, **визионерская**] – Поговорим о будущем и о синтезе программ

PART I : TEST GENERATION

LIVE DEMO

IntelliTest

Pain point

Programmers **don't like** to write tests

Unit tests:

- Best defense against **regression** (quality increases)
- Kind of living **specification** (understanding increases)
- Errors found by unit tests **easier to correct** (costs reduces)

Solution

Generate unit test automatically



Unit test generation

Goal: to fixate code behavior

Regression suite

Criteria: generate **minimum** number of unit tests that will cover **maximum** lines of code



Safety verification

Goal: to find bugs and vulnerabilities

Error suite

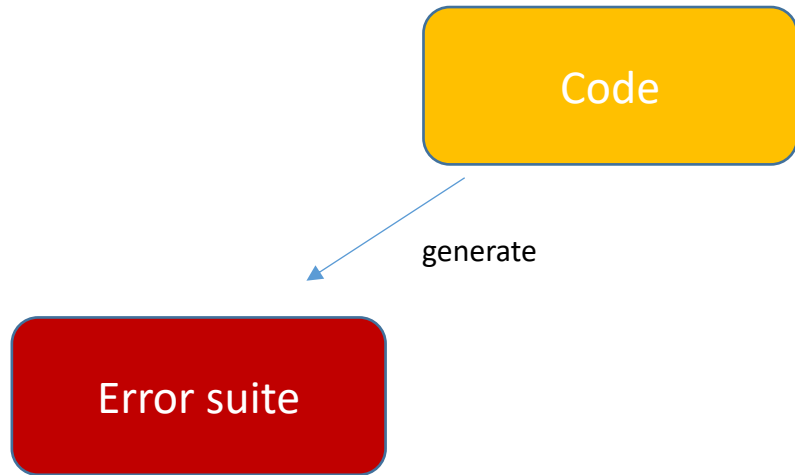
Criteria: Find **maximum** number bugs and express them in form of tests

No tests

Code

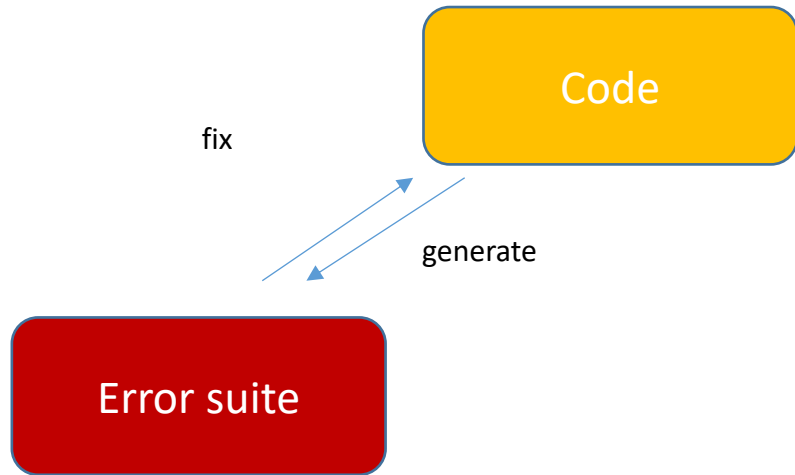
Code contains NPE,
StackOverflows and so on

Error suite



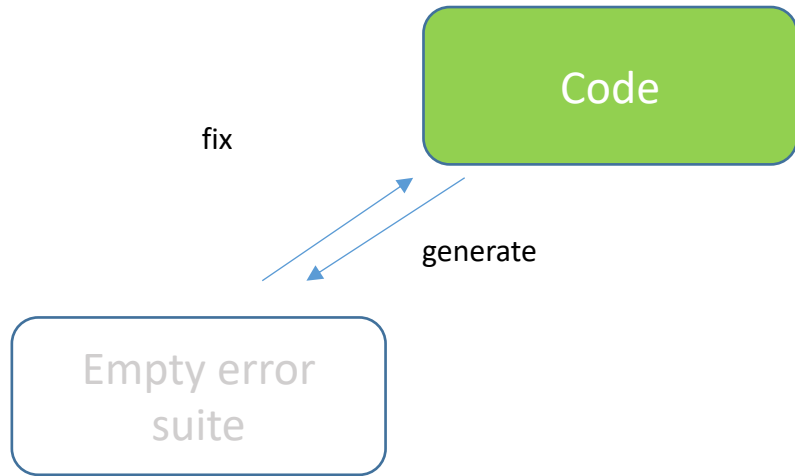
Code contains NPE,
StackOverflows and so on

Error suite



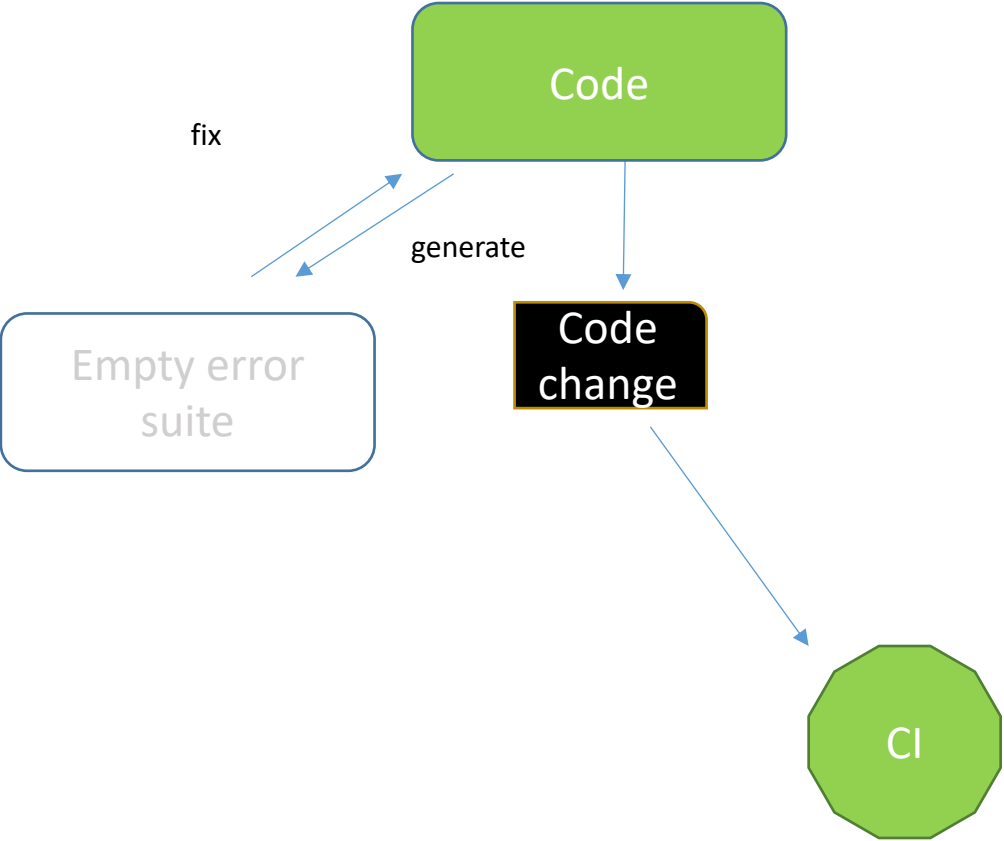
Code contains NPE,
StackOverflows and so on

Error suite



Code hasn't NPE,
StackOverflows and so on

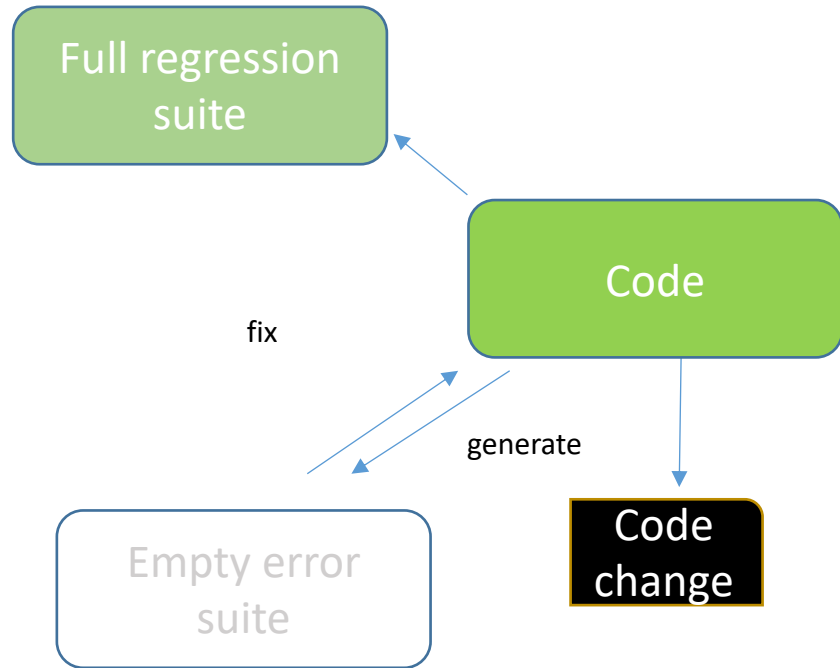
Regression



Code hasn't NPE,
StackOverflows and so on

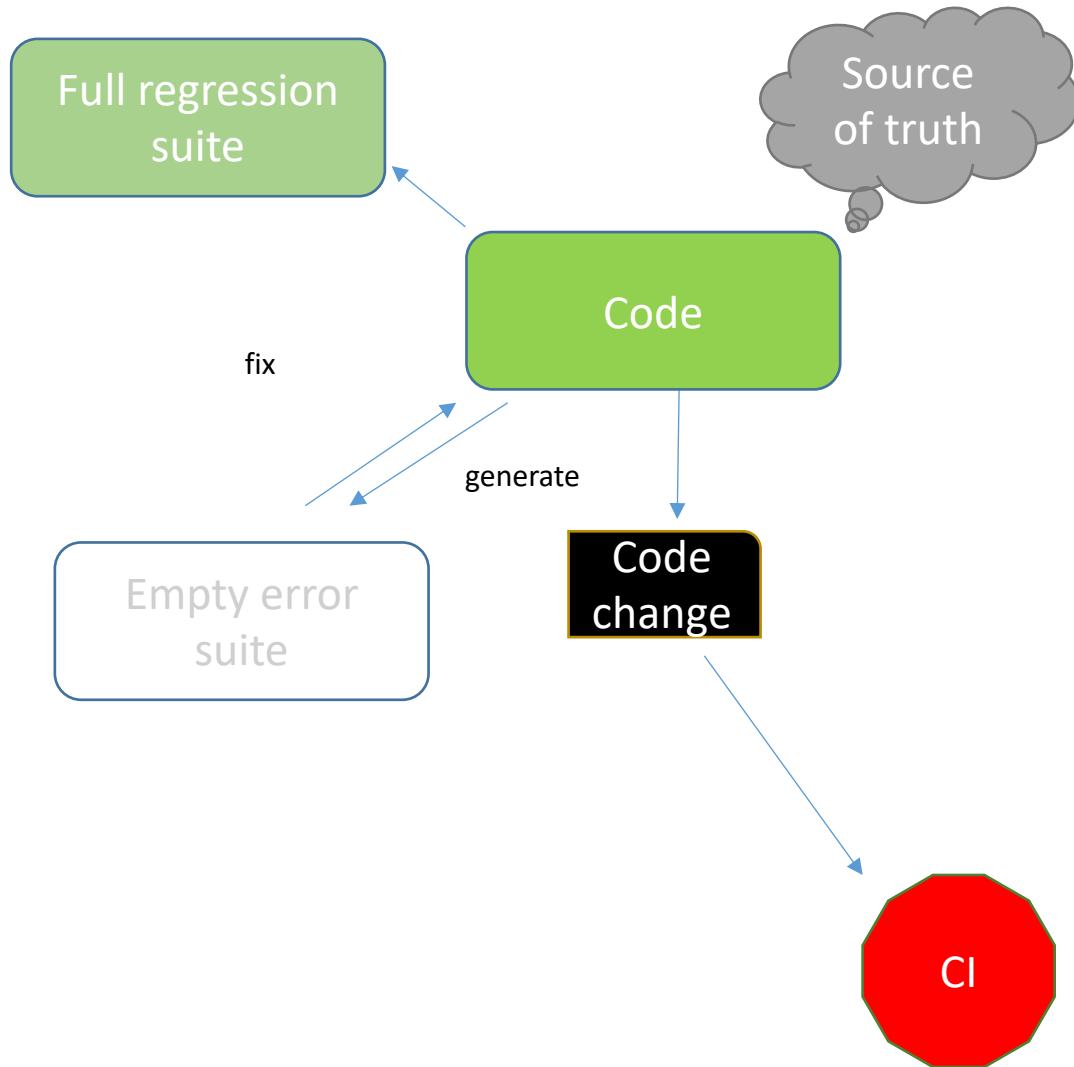
If developer commit change
nobody will notice bug until
it happens on production

Regression suite



Code hasn't NPE,
StackOverflows and so on

Regression suite



Code hasn't NPE,
StackOverflows and so on

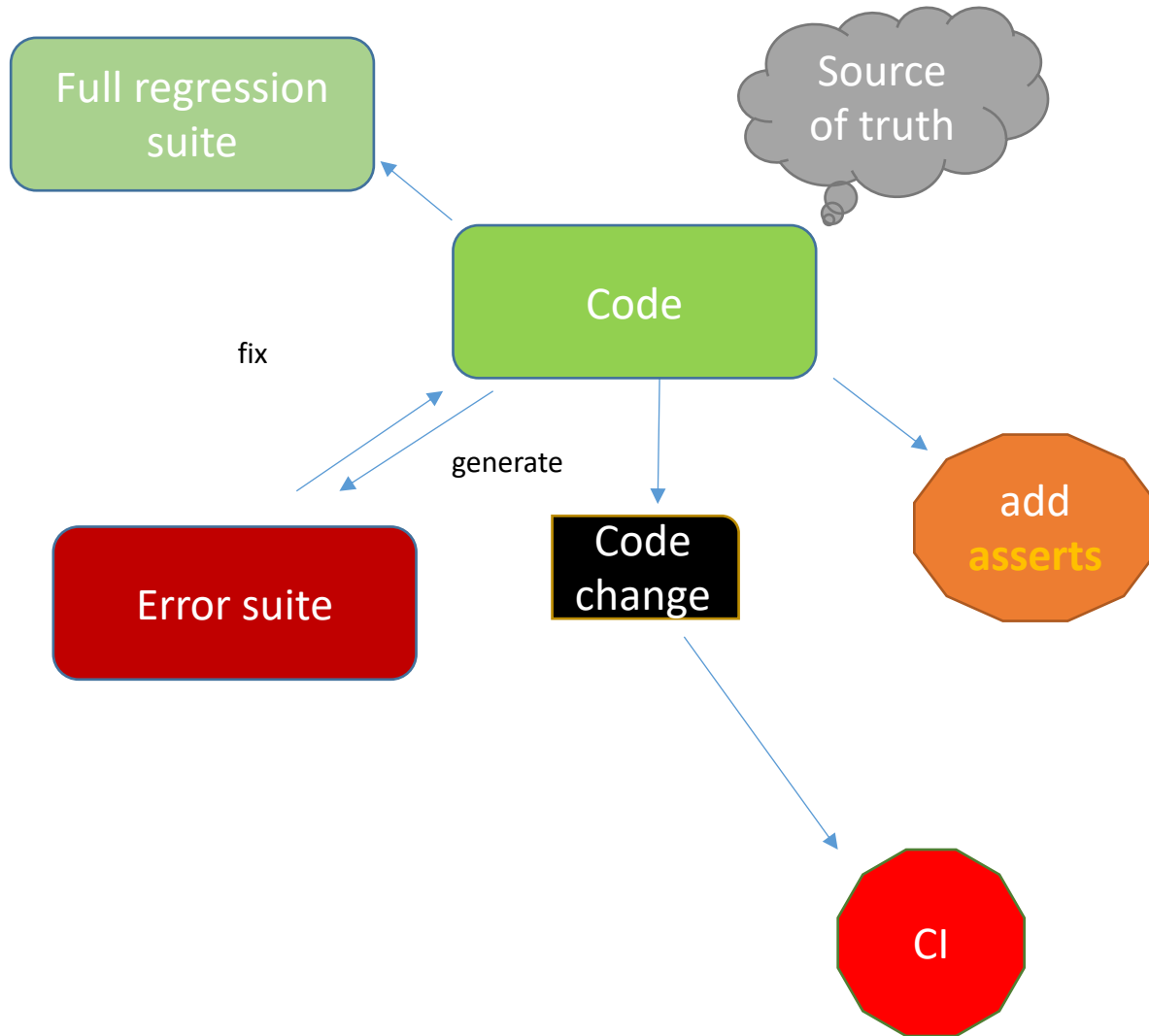
Now behavior is fixated.

Red CI status means one of two things:

- Code change breaks correct behavior
- Initial code behavior wasn't correct

Anyway it's *easy to localize* problem

Specification



Code hasn't NPE,
StackOverflows and so on

Now behavior is fixated.
Red CI status means one of two things:

- Code change breaks correct behavior
- Initial code behavior wasn't correct

Anyway it's easy to localize problem

Code is tested against
specification formalized by
asserts

PART II : TECHNIQUES

How to verify program is correct?

```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
        // else  
    ...  
}
```



ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

FASE 2021

3rd Competition on Software Testing (Test-Comp 2021)

FASE '21
Tue, March ??, 2021
Luxembourg

3rd Intl. Competition on Software Testing held at FASE 2021 in Luxembourg.

About Test-Comp

Important Dates

Competition Jury

-  Competition Description
-  2020 Competition Report

Motivation

Tool competitions are a special form of comparative evaluation, where each tool has a team of

<https://test-comp.sosy-lab.org/2021/>



ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

TACAS 2021

10th Competition on Software Verification (SV-COMP 2021)


TACAS '21
Thur, March ??, 2021
Luxembourg

10th Intl. Competition on Software Verification held at TACAS 2021 in Luxembourg.

About SV-COMP

Important Dates

Competition Jury

-  2020 Competition Report (results of the competition and a lot of detailed information on SV-COMP 2020)

Motivation

Competition is a driving force for the invention of new methods, technologies, and tools. This web page describes the competition of software-verification tools, which will take place at TACAS.

<https://sv-comp.sosy-lab.org/2021/>

Fuzzers

Random testing

Random inputs generation until crash

BLACK-BOX
TECHNIQUE

Adaptive random testing (ART)

```
while (coverage is not enough) {  
  generate new test()  
  add to suite if coverage increases  
}
```

Instrument
program

GREY-BOX
TECHNIQUE

Evolutionary algorithm

- Generations (test suites)
- Cross-over existing tests
- Mutate existing tests

Tool	Score	Ranking
t3	145.27	2.30
evosuite	255.43	2.38
randoop	154.34	2.51
tardis	66.80	3.73
sushi	39.84	4.09

Symbolic execution

Symbolic Virtual Machine State

Instruction: **<Enter>**

Symbolic Memory (SM): $x = x_0, y = y_0$

Path Condition (PC): true



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
        // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (x > 0)`

SM: `x = x0, y = y0`

PC: `true && x0 > 0`



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `abs = x`

SM: `x = x0, y = y0, abs = x0`

PC: `x0 > 0`



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (y == 42)`

SM: $x = x_0, y = y_0, \text{abs} = x_0$

PC: $x_0 > 0 \ \&\& \ y_0 == 42$



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (abs + y < 0)`

SM: $x = x_0, y = y_0, \text{abs} = x_0$

PC: $x_0 > 0 \ \&\& \ y_0 == 42 \ \&\& \ x_0 + y_0 < 0$



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: **ERROR**

SM: $x = x_0, y = y_0, \text{abs} = x_0$

PC: $x_0 > 0 \ \&\& \ y_0 == 42 \ \&\& \ x_0 + y_0 < 0$

(Ask SMT Solver “Is PC satisfiable?”)



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (abs + y < 0) //else`

SM: $x = x_0, y = y_0, abs = x_0$

PC: $x_0 > 0 \ \&\& \ y_0 == 42 \ \&\& \ \underline{\neg(x_0+y_0<0)}$

(Go to **else** branch: negate last condition)



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```


SMT solver

SMT = Satisfiability modulo theories

PC:

$x_0 > 0$

&&

$y_0 == 42$

&&

$x_0 + y_0 < 0$

```
(declare-const x0 Int)
```

```
(declare-const y0 Int)
```

```
(assert (> x0 0))
```

```
(assert (= y0 42))
```

```
(assert (< (+ x0 y0) 0))
```

```
(check-sat)
```

SMT-LIB2 syntax

SMT solver: Z3

PC:

```
x0 > 0
&&
y0 == 42
&&
x0 + y0 < 0
```

```
(declare-const x0 Int)
(declare-const y0 Int)

(assert (> x0 0))
(assert (= y0 42))
(assert (< (+ x0 y0) 0))

(check-sat) ;unsatisfiable

sample.smt
```

Try it here online:

<https://rise4fun.com/z3/tutorial>

Or use command line tool:

```
#> apt-get install z3
```

```
#> z3 -smt2 sample.smt
```

```
#unsat
```

SMT solver: theories

Integer arithmetic theory

```
PC:  
x0 > 0  
&&  
y0 == 42  
&&  
x0 + y0 < 0
```

```
(declare-const x0 Int)  
(declare-const y0 Int)  
  
(assert (> x0 0))  
(assert (= y0 42))  
(assert (< (+ x0 y0) 0))  
  
(check-sat) ;unsatisfiable
```

Bitvector theory

```
(set-option :pp.bv-literals false)  
(declare-const x0 (_ BitVec 32))  
(declare-const y0 (_ BitVec 32))  
  
(assert (bvsgt x0 (_ bv0 32)))  
(assert (= y0 (_ bv42 32)))  
(assert (bvslt (bvadd x0 y0) (_ bv0 32)))  
  
(check-sat) ;satisfiable  
(get-model) ;x0 = MAXINT-41, y0 = 42
```

Symbolic execution: problems

1. How to deal with *path explosion* ?
2. How to handle loops / recursion?
3. How to present Heap in symbolic memory?
4. How to invoke native functions/syscalls?
5. What to do with concurrency?
6. if (sha256(x) == “try to solve this!”)

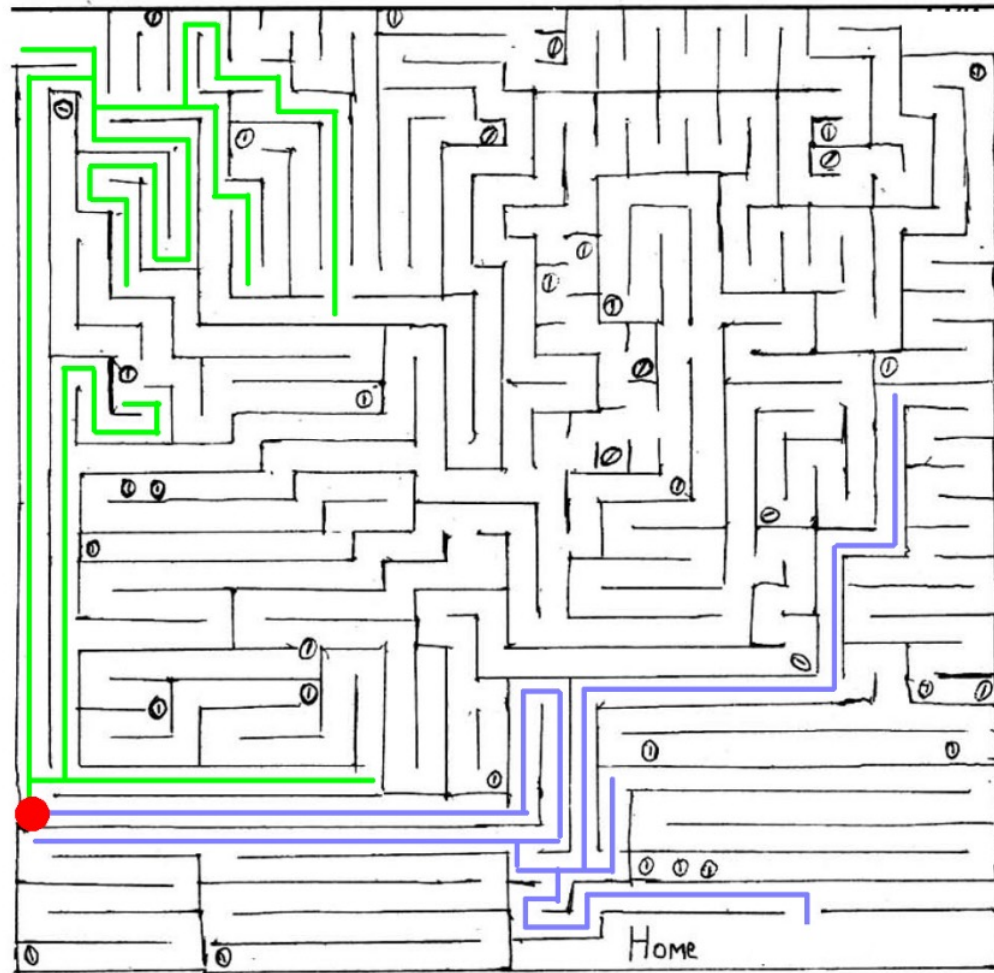
[A Survey of Symbolic Execution Techniques](#)

Path Explosion

```
for (int i = 0; i < n; ++i) {  
    if (cond(i))  
        // ...  
    else  
        // ...  
}
```

- Symbolic conditional statements fork the execution state
- N iterations of loop with conditional statement can fork 2^N times

Bidirectional Symbolic Execution



Weakest Preconditions

- Is throw reachable?

- Yes, if

`p != null`

`&&`

`ReferenceEquals(p, q)`

- Intuitively, weakest preconditions "roll back" the condition through the program

```
class A
{
    public int X { get; set; }
}

void F(A p, A q)
{
    p.X = 1;
    q.X = 2;
    if (p.X == q.X)
        throw new Exception();
}
```

Program Invariants

- Loops and recursion can lead to the unbounded amount of different program behaviours
- Solution: over-approximate them!

```
int max = 0;  
for (int i = 0; i < a.Length; ++i)  
    max = Math.Max(Math.Abs(a[i]), max);
```

```
if (max < 0)  
    throw new Exception();
```

Over-approximate the whole loop with

```
max >= 0
```


Heap and Symbolic Execution

Memory models in symbolic execution: key ideas and new thoughts

Luca Borzacchiello, Emilio Coppa^{*†}, Daniele Cono D’Elia and Camil Demetrescu

Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Rome, Italy

SUMMARY

Symbolic execution is a popular program analysis technique that allows seeking for bugs by reasoning over multiple alternative execution states at once. As the number of states to explore may grow exponentially, a symbolic executor may quickly run out of space. For instance, a memory access to a symbolic address may potentially reference the entire address space, leading to a combinatorial explosion of the possible resulting execution states. To cope with this issue, state-of-the-art executors either concretize symbolic addresses that span memory intervals larger than some threshold or rely on advanced capabilities of modern satisfiability modulo theories solvers. Unfortunately, concretization may result in missing interesting execution states, for example, where a bug arises, while offloading the entire problem to constraint solvers can lead to very large query times. In this article, we first contribute to systematizing knowledge about memory models for symbolic execution, discussing how four mainstream symbolic executors deal with symbolic addresses. We then introduce MEMSIGHT, a new approach to symbolic memory that reduces the need for concretization: rather than mapping address instances to data as previous approaches do, our technique maps symbolic address expressions to data, maintaining the possible alternative states resulting from the memory referenced by a symbolic address in a compact, implicit form. Experiments on prominent programs show that MEMSIGHT, which we implemented in both ANGR and KLEE, enables the exploration of states that are unreachable for memory models that perform concretization and provides a performance level comparable with memory models relying on advanced solver theories. © 2019 John Wiley & Sons, Ltd.

Received 11 April 2019; Revised 21 October 2019; Accepted 22 October 2019

KEY WORDS: symbolic execution; software testing; pointer reasoning

1. INTRODUCTION

Symbolic execution is a technique for program property verification largely employed in the software testing and security domains [1]. By taking on symbolic rather than concrete input values, multiple execution paths can be explored at once, with each path describing the program’s behaviour for a well-defined class of inputs. Nonetheless, the number of paths to explore can be prohibitively large, for example, in the presence of unbounded loops. In this article, we tackle one specific problem that may affect exploration in a symbolic executor: *symbolic pointers*.

As in prior works [2], we use the term *symbolic pointer* to refer to any symbolic expression used during the exploration carried by a symbolic engine to reason over the memory state of the program under analysis. In more detail, any expression with at least one non-concrete constituent and that is used by program to perform a memory read or write operation can be regarded under this term. Symbolic pointers may lead an executor to fork the execution, possibly generating an extremely large number of paths. When forks are avoided or at least limited, symbolic executors must resort

Heap Abstractions for Static Analysis

VINI KANVAR and UDAY P. KHEDKER, Indian Institute of Technology Bombay

Heap data is potentially unbounded and seemingly arbitrary. Hence, unlike stack and static data, heap data cannot be abstracted in terms of a fixed set of program variables. This makes it an interesting topic of study and there is an abundance of literature employing heap abstractions. Although most studies have addressed similar concerns, insights gained in one description of heap abstraction may not directly carry over to some other description.

In our search of a unified theme, we view *heap abstraction* as consisting of two steps: (a) *heap modelling*, which is the process of representing a heap memory (i.e., an unbounded set of concrete locations) as a heap model (i.e., an unbounded set of abstract locations), and (b) *summarization*, which is the process of bounding the heap model by merging multiple abstract locations into summary locations. We classify the heap models as storeless, store based, and hybrid. We describe various summarization techniques based on *k*-limiting, allocation sites, patterns, variables, other generic instrumentation predicates, and higher-order logics. This approach allows us to compare the insights of a large number of seemingly dissimilar heap abstractions and also paves the way for creating new abstractions by mix and match of models and summarization techniques.

Categories and Subject Descriptors: A. [General and Reference]: General Literature; F.3 [Theory of Computation]: Logic—*Logic and verification*; Design and Analysis of Algorithms; Semantics and Reasoning—*Program analysis*; Abstraction; D.2 [Software Engineering]: Software Organization and Properties—*Automated static analysis*; *Software verification*

General Terms: Design, Algorithms, Verification, Languages

Additional Key Words and Phrases: Abstraction, heap, pointers, shape analysis, static analysis, store based, storeless, summarization

ACM Reference Format:

Vini Kanvar and Uday P. Khedker. 2016. Heap abstractions for static analysis. ACM Comput. Surv. 49, 2, Article 29 (June 2016), 47 pages.
DOI: <http://dx.doi.org/10.1145/2931098>

1. HEAP ANALYSIS: MOTIVATION

Heap data is potentially unbounded and seemingly arbitrary. Although there is a plethora of literature on heap, the formulations and formalisms often seem dissimilar. This survey is a result of our quest for a unifying theme in the existing descriptions of heap.

1.1. Why Heap?

Unlike stack or static memory, heap memory allows on-demand memory allocation based on the statements in a program (and not just variable declarations). Thus, it

Symbolic execution: problems

1. How to deal with *path explosion* ?
 2. How to handle loops / recursion?
 3. How to present Heap in symbolic memory?
 4. How to invoke native functions/syscalls?
 5. What to do with concurrency?
 6. if (sha256(x) == “try to solve this!”)
1. Bidirectional symbolic execution
 2. Try calculate “Inductive invariant” – Horn clauses
 3. SMM: Theory of Arrays + Theory of Bitvectors
 4. Write mock / make value concrete and execute / forget
 5. Hard: $\#concurrent_states = \#states \wedge \#threads$
 6. Rare in real programs, bypass

[A Survey of Symbolic Execution Techniques](#)

PART III : SMT SOLVER

SAT/SMT by Example

$$\begin{aligned} \text{○} + \text{○} &= 10 \\ \text{○} \times \text{□} + \text{□} &= 12 \\ \text{○} \times \text{□} - \text{△} \times \text{○} &= \text{○} \\ \text{△} &= ? \end{aligned}$$

```
#!/usr/bin/python
from z3 import *

circle, square, triangle = Ints('circle square triangle')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square-triangle*circle==circle)
print s.check()
print s.model()
```

```
sat
[triangle = 1, square = 2, circle = 5]
```

SAVES WITH BY EXAMPLE

9		6		7		4		3
			4			2		
	7			2	3		1	
5						1		
	4		2		8		6	
		3						5
	3		7				5	
		7			5			
4		5		1		7		8



LIVE DEMO

Using Z3 in .NET


PART IV : V#

IntelliTest .NET Core issues

On Roadmap ⓘ

47 167

^ **167**
∨ Votes

 [Roberto Santana Perdomo](#) - Reported Oct 16, 2018

Most of our projects are being developed on ASP.NET Core and we develop our libraries on .NET Standard.
We really want to add IntelliTest to our set of testing technologies but it currently only supports projects targeting the .NET Framework.

As we believe .NET Core/Standard to be the future of .NET (web) development, I think great features like these should be prioritized for these ecosystems.

visual studio testing-tools enterprise-2017 dotnet-roadmap vs2022-roadmap

 [Kendra Havens \[MSFT\]](#) ...

We are working on re-costing this work. I'll update this ticket when we have something scheduled. To give background, adding .NET Core / .NET Standard / .NET 5 support may require complete rewrites of certain components of IntelliTest and we would need to spend significant time rebuilding the code base knowledge within the team. We are trying to invest in experiences that will have the highest impact while also keeping pace with rapid advancements in the platform and tooling (including .NET 6 Hot Reload, VS 2022 64bit to name a few). Thank you for bearing with us.

 8 

May 19, 2021

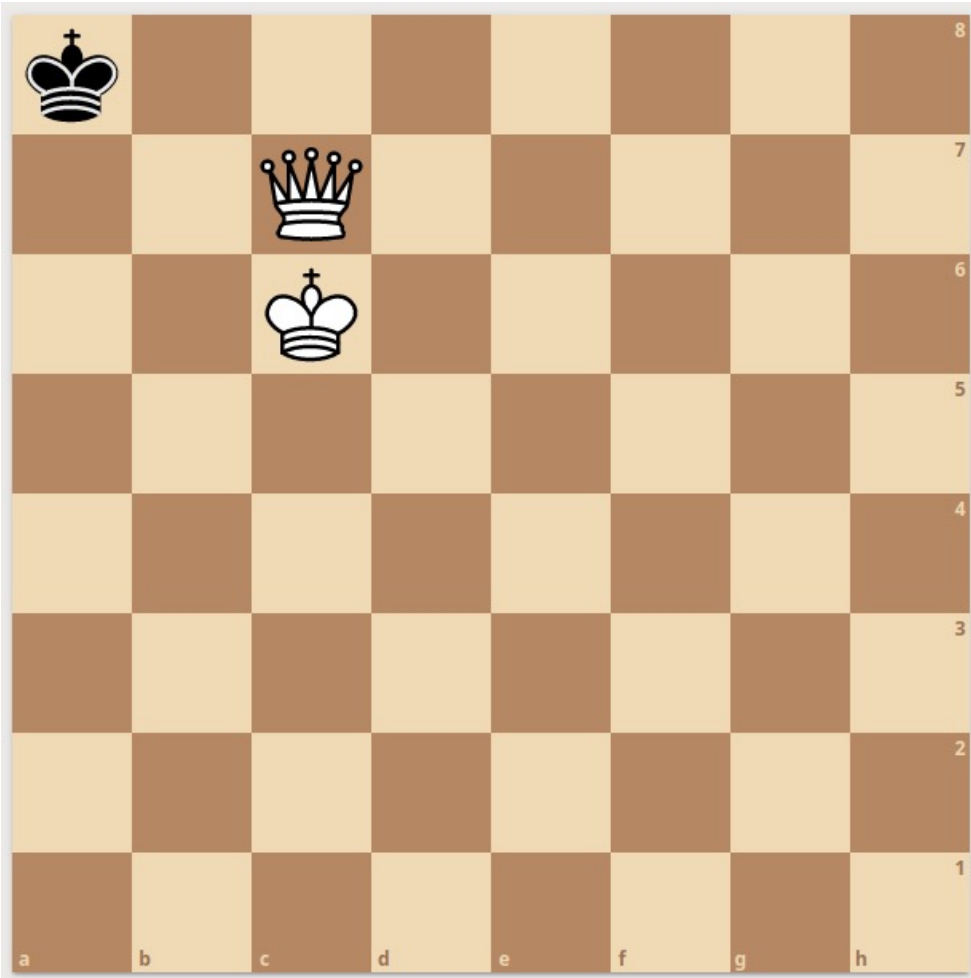
V#

- <https://github.com/vsharp-team/vsharp>
- Open source symbolic execution engine
- Supports .NET Core
- Written in F#
- Currently under development

LIVE DEMO

V#

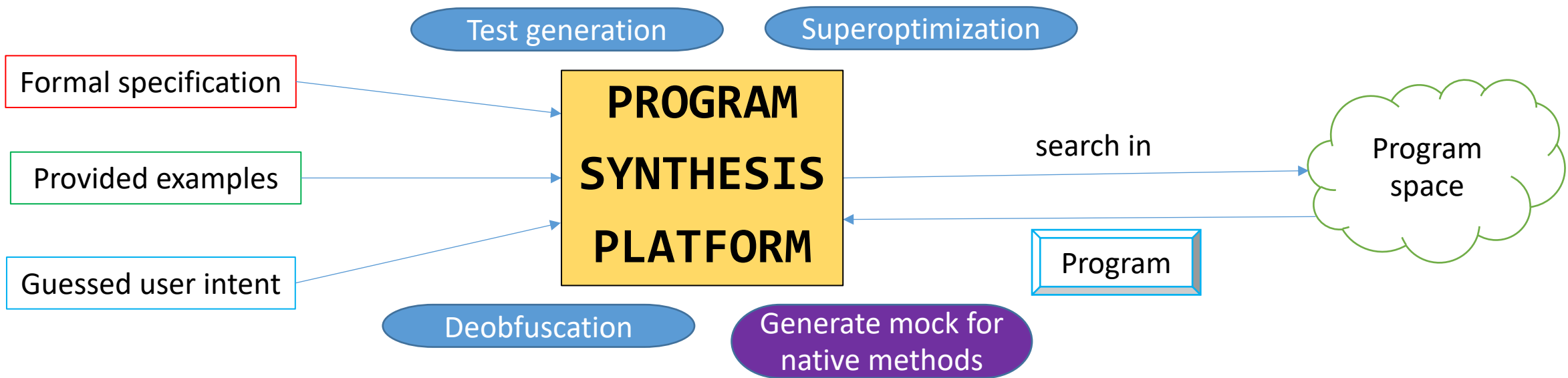
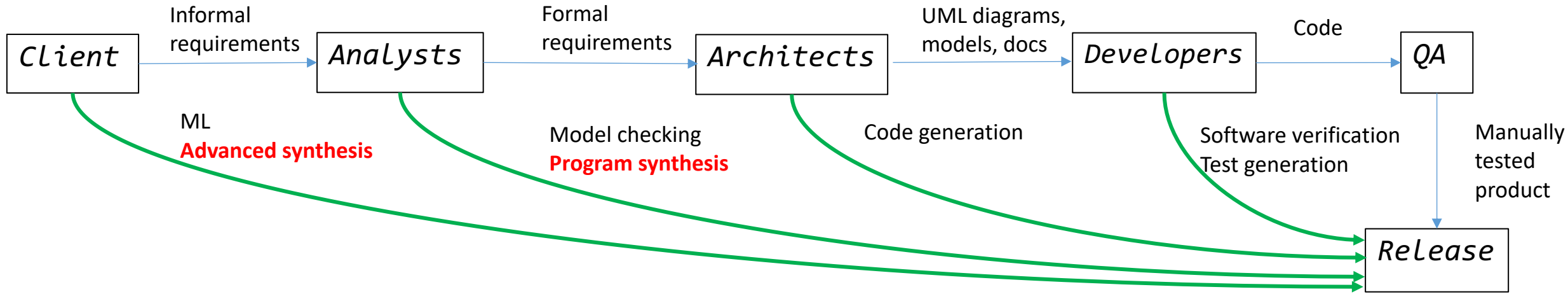
Chess.NET library test



```
public static bool CheckMate(string moveString)
{
    var whiteKing = (Piece) new King(owner: Player.White);
    var blackKing = (Piece) new King(owner: Player.Black);
    var whiteQueen = (Piece) new Queen(owner: Player.White);
    var board = new [] {
        new Piece[8] { blackKing, null, null, null, null, null, null, null },
        new Piece[8] { null, null, whiteQueen, null, null, null, null, null },
        new Piece[8] { null, null, whiteKing, null, null, null, null, null },
        new Piece[8] { null, null, null, null, null, null, null, null },
        new Piece[8] { null, null, null, null, null, null, null, null },
        new Piece[8] { null, null, null, null, null, null, null, null },
        new Piece[8] { null, null, null, null, null, null, null, null },
        new Piece[8] { null, null, null, null, null, null, null, null },
    };
    var data = new GameCreationData {Board = board, WhoseTurn = Player.White};
    var game = new ChessGame(data);
    Move move = new Move(originalPosition: "C7", newPosition: moveString, Player.White);
    game.ApplyMove(move, alreadyValidated: true);
    return game.IsCheckmated(Player.Black);
}
```

Epilogue: **FUTURE**

Next decade opportunities for Software Automation



Programming by Example: FlashFill in Excel

	A	B
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	nancy freehafer
3	Andrew.Cencici@northwindtraders.com	andrew cencici
4	Jan.Kotas@litwareinc.com	jan kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	mariya sergienko
6	Steven.Thorpe@northwindtraders.com	steven thorpe
7	Michael.Neipper@northwindtraders.com	michael neipper
8	Robert.Zare@northwindtraders.com	robert zare
9	Laura.Giussani@adventure-works.com	laura giussani
10	Anne.HL@northwindtraders.com	anne hl
11	Alexander.David@contoso.com	alexander david
12	Kim.Shane@northwindtraders.com	kim shane
13	Manish.Chopra@northwindtraders.com	manish chopra
14	Gerwald.Oberleitner@northwindtraders.com	gerwald oberleitner
15	Amr.Zaki@northwindtraders.com	amr zaki
16	Yvonne.McKay@northwindtraders.com	yvonne mckay
17	Amanda.Pinto@northwindtraders.com	amanda pinto

Excel automatically synthesizes the formula

```
Concatenate(ToLower(Substring(v, WordToken, 1)), " ", ToLower(Substring(v, WordToken, 2)))
```

Code Completion

Formal specification: code compiles

Guessed user intent:

- Probability of each expression in program

Provided examples: statistics of previous completions

```
object Main {  
  def main(args:Array[String]) = {  
    var body = "email.txt"  
    var sig = "signature.txt"
```

```
    var inStream:SequenceInputStream = |
```

```
    var eof:Boolean = false  
    var byteCount:Int = 0  
    while (!eof) {  
      var c:Int = inStream.read()  
      if (c == -1)
```

```
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(sig))  
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(body))  
new SequenceInputStream(new FileInputStream(body), new FileInputStream(sig))  
new SequenceInputStream(new FileInputStream(body), new FileInputStream(body))  
new SequenceInputStream(new FileInputStream(sig), System.in)
```


Superoptimization

- Optimization: try some predefined rewriting rules
- Superoptimization: search the optimal implementation in the **whole program space**

Formal specification: same semantics,
verified by symbolic execution

Average of x and y:

$(x + y) / 2$ can overflow

Guessed user intent:
Fastest possible

$(x / 2) + (y / 2) + (((a \% 2) + (b \% 2)) / 2)$ is too expensive

Superoptimizer synthesises $(x|y) - ((x \oplus y) \gg 1)$

Полезные ссылки

- [IntelliTest](#) - официальные доки от Microsoft
- [A Survey of Symbolic Execution Techniques](#) - хорошая вводная статья в символьное исполнение
- “SAT/SMT by Example” – много практических примеров использования SAT и SMT солверов
- <https://github.com/vsharp-team/vsharp>
- [Program Synthesis: Opportunities for the Next Decade](#)

Dmitry Ivanov, *Huawei Saint Petersburg Research Center*, korifey@gmail.com,
@korifey_ad

Dmitry Mordvinov, *JetBrains Research, SPbU*, mordvinov.dmitry@gmail.com