

# Perspectives on Structuring a Research Presentation

*or*

“One motto, three rules of thumb”

Claire Le Goues

SSSG, September 28, 2015



# Repairing Programs with Semantic Code Search

Yalin Ke      Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
[{yke, kstolee}@iastate.edu](mailto:{yke, kstolee}@iastate.edu)

Claire Le Goues      Yuriy Brun  
School of Computer Science      College of Information and Computer Science  
Carnegie Mellon University      University of Massachusetts, Amherst  
[clegoues@cs.cmu.edu](mailto:clegoues@cs.cmu.edu)      [brun@cs.umass.edu](mailto:brun@cs.umass.edu)

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenge lies in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea. SearchRepair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Localizes* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it *generalizes* to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

Maryn T. Stolee  
School of Computer Science  
Iowa State University  
mstolee{kstolee}@iastate.edu

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
clegoues@cs.cmu.edu

Yuriy Brun  
College of Information and  
University of Massachusetts,  
brun@cs.umass.edu

Automated program repair can potentially reduce developer time and improve software quality but recent studies have shown significant shortcomings in the quality of automatically generated patches. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenge is efficiently finding code semantically similar (but not identical) to the faulty code and then appropriately integrating it into the faulty program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior; (2) localizing a given defect to likely buggy program regions; and (3) deriving the desired input-output behavior for these regions from code fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior; and (4) replacing the likely buggy code with these potential patches. We show that the patches repair the bug against programs from which they were derived. We also show that SearchRepair repairs 150 (19%) of the bugs in a dataset of 780 bugs written by novice students, 20 of which were previously unaddressed by GenProg, TrpAutoRepair, and AE. We compare the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass. We find that SearchRepair-repaired programs pass 97.3% of the tests on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

### I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source ecosystem has led to many large, publicly accessible source code repositories, such as GitHub, BitBucket, and SourceForge. These repositories contain programs include routines, data structures, and libraries that have been previously implemented in other software projects [1]. In fact, we posit that, if a method or component does not already exist to address a defect, with high probability, there is a high-quality version of that component in some public repository of open-source code. The research challenge lies in efficiently finding and integrating these implementations into a faulty program.

Our key idea is to

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea. SearchRepair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
  - 2) *Localizes* a defect to likely buggy program fragments.
  - 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
  - 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- Validates each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work on semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify regions of faulty code and construct input-output profiles for semantic search. Third, we build a database of fragments and perform semantic code search over the SMT-encoded fragments. Finally, we adapt the returned code fragment to the defect, perform variable renaming, and validate against provided test cases.

Our goal with SearchRepair is to produce high quality patches while still addressing a wide range of defects. A key feature of a high-quality patch is that, either human- or tool-generated, is that it generalizes to the desired, often unwritten specification of the faulty program behavior. This is a challenge for automatic repair systems [1, 2, 3, 4, 5, 6, 7, 10, 11, 15, 16, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 42, 43, 48, 49, 50, 51, 52, 54, 56, 57, 58, 59, 60, 71, 72, 73, 74, 76], many of which use test suites to guide patching efforts. Modern test-suite guided repair, particularly those following a *generate-and-validate* approach (i.e., heuristically constructing and then testing likely candidate repairs), although typically general and efficient, produce poor quality patches that often fail to pass the test suites used to guide patch generation.

By definition, test suites can only validate a small fraction of correct behavior. A patch that passes a given test suite may thus exhibit faulty behavior in other execution. This is a well-known phenomenon in the field of program mutation, where the program

Motto:

**YOU ARE NOT PRESENTING THE PAPER.  
YOU ARE PRESENTING THE WORK.**

# Program repair via semantic search.

## Repairing Programs with Semantic Code Search

Yalin Ke      Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
[{yalin, kstolee}@iastate.edu](mailto:{yalin, kstolee}@iastate.edu)

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
[clegoue@cs.cmu.edu](mailto:clegoue@cs.cmu.edu)

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
[tous@cs.umass.edu](mailto:tous@cs.umass.edu)

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the semantics of code to find patches that fix bugs. We propose that the key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately applying that code to the buggy regions. We propose SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints, (2) generating a set of potential patches for a given defect by likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using a search-based approach to find the best patch by searching for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the replaced code satisfies the original test cases. We find that SearchRepair repairs 159 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GePProg, TriPathRepair, and A2, and repairs some of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass and fail to repair. SearchRepair repairs 97.3% of the test cases, on average, whereas GePProg, TriPathRepair, and A2 repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GePProg, TriPathRepair, and A2, and repairs some defects that these tools cannot.

### I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software developers spend a significant amount of developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is slow and keeps growing as the number of defects in extant software [2]. Despite the estimated detrimental impact on a company's bottom line, known defects ship in mature software projects [45], and many defects, including those that are particularly critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code repositories [11], [12], [24], [31], [42], [51], [57], [66], [70], [73], [74], [76]. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that if a method or component of one program is a defect with a high probability that there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique based on our idea of Semantic Repair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Generate* a set of likely buggy program fragments.
- 3) *Construct*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Search* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original possibly faulty code.
- 5) *Validate* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we extend our previous work on semantic search [68] to C and Java code fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build a large database of human-written code fragments from the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

One key benefit of SearchRepair is to produce high-quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the broader desired environment and is able to prove correctness of behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [50], [52], [53], [56], [57], [60], [66], [70], [73], [74], [76]), many of which are limited to specific patches and locate patching efforts. Modern test suite guided repair techniques, particularly those following a *generate-and-validate* paradigm ([20], [57], [65], [66], [68], [70], [73], [76]), are capable of generating a large number of candidate repairs, although typically general and scalable, often produce poor-quality patches that overflow to the specification test suites used to guide patch generation ([20], [57], [65]).

By contrast, SearchRepair generates patches that are a close approximation of the original specification behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomena of *overfitting* to an objective function, where the program



# Program repair via semantic search

## Repairing programs with Semantic Code Search

Yalin Ke Kathryn Morris  
Department of Computer Science College of Information and Computer Science  
Iowa State University University of Massachusetts, Amherst  
{yalin, kathrym}@iastate.edu teus@cs.umass.edu

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
teus@cs.umass.edu

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but has not drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses semantic code search to find correct implementations of the faulty code. The key challenges lie in efficiently finding code semantically similar but not identical to defective code and then appropriately combining it with the defective code. We introduce *SearchRepair*, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as a set of constraints, (2) generating many potential patches for a given defect by likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using semantic code search to find correct implementations of fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the generated patch is correct according to the test suite. We find that *SearchRepair* repairs 159 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by *GenProg*, *TrapAutoRepair*, and *AE*. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that *SearchRepair* passes 97.3% of the tests, 9.7% more, on average, whereas *GenProg*, *TrapAutoRepair*, and *AE*-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that *SearchRepair* produces higher-quality repairs than *GenProg*, *TrapAutoRepair*, and *AE*, and repairs some defects that *GenProg*, *TrapAutoRepair*, and *AE* cannot.

### I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software development is a slow and error-prone process [11]. To manually find and fix bugs in their software, unfortunately, manual bug repair, the industry standard, is still the go-to method for fixing the most critical defects in extant software [21]. Despite the estimated detrimental impact on a company's bottom line, known defects ship in mature software projects [45], and many defects, including those that are particularly critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open source movement has led to many large, publicly accessible source code repositories containing many thousands of programs. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that if a method or component of one program is a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

code to find correct implementations of the faulty code for software defects. Semantic code search is different from what it does, rather than by syntactic search, it finds code that is semantically similar to the faulty code.

- 2) *Locate*: Find all likely buggy program fragments.
- 3) *Combine*: Derive the desired input-output profiles for the likely buggy code.
- 4) *Semantic search*: Use semantic code search, i.e., constraint solvers, for finding correct implementations of the fragments become semantically similar to the desired behavior and inserting them into the faulty code.
- 5) *Validate*: Evaluate each potential patch against the test suite to determine if it indeed fixes the bug.

To make *SearchRepair* possible, we first build a large database of human-written code fragments [68] to serve as the search space. Second, we adapt spectrum-based fault localization [10] to identify candidate regions of faulty code and output profiles to use as input to semantic search.

Third, we build a semantic search system that takes an SMT-encoded code database, adapt the returned code to the defective context via variable renaming, and validate provided test suites.

One key challenge of *SearchRepair* is to produce high-quality patches while still addressing a broad range of defects. A key feature of a high-quality patch, whether human- or tool-generated, is that it generalizes to the broader desired behavior, and that it generates a correct implementation. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [50], [51], [52], [53], [56], [57], [58], [66], [69], [70], [73], [74], [76]), many of which are too specific and generate patching efforts. Modern test suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (e.g., [20], [21], [24], [25], [26], [27], [28], [29], [30] of candidate repairs), although typically general and scalable, often produce poor quality patches that overflow to the specification test suites used to guide patch generation [20], [57], [65].

By contrast, *SearchRepair* produces a patch that is a generalization of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomena of *overfitting* to an objective function, where the program



# Program repair via semantic search.

## Repairing Programs with Semantic Code Search

Yulin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
{yke, kstolee}@iastate.edu

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
cldeg@cs.cmu.edu

Yuriy Brun

College of Information and Computer Science  
University of Massachusetts, Amherst  
brun@cs.umass.edu

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated patches. We propose a new form of automated repair that leverages the body of existing open-source code to find potential patches. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately applying it to repair the defect. We introduce *SearchRepair*, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as Satisfiability Modulo Theories (SMT) constraints; (2) given a defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments; (3) performing a semantic search over the SMT database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches; and (4) validating that the patch actually repairs the defect. We evaluate *SearchRepair* on 159 C programs from the Geotools-C benchmark suite. We find that *SearchRepair* repairs 159 (19%) of 778 benchmark C defects written by novice students, 20 of which are never repaired by humans. We also compare *SearchRepair* to three baselines of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that *SearchRepair* passes 97.0% of the tests, on average, whereas GeProg, TryAutoRepair, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that *SearchRepair* produces higher-quality repairs than GeProg, TryAutoRepair, and AE, and faster.

### I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that they require significant developer time [75] to manually find and fix bugs in their software. Unfortunately, manual bug repair, the industry standard, largely goes unnoticed to keep up with the volume of defects in extant software [2]. Despite the established negative impact on a company's bottom line, known defects ship in mature software projects [45], and many defects, including those that are severely critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Bug databases, which include bug databases for designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a patch for it in one of the bug databases in some publicly accessible software project. The research challenge lies in how to automatically find and use such improvements to fix bugs.

Our key idea is to use semantic code search [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop *SearchRepair*, a new technique predicated on our use of Semantic Repair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Derives* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original problematic code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make *SearchRepair* possible, we first extend our previous work on semantic search [68] to C programs. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as inputs to semantic search. Third, we build a large database of open-source code and use the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with *SearchRepair* is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often narrow context of a specific program [3]. This is a challenge for automatic repair programs (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [67], [70], [73], [76]), most of which focus on a single defect and generate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically instrumenting and then testing large numbers of candidate patches), although typically generalizable, often produce poor quality patches that overflow to the specification test suites used to guide patch generation [20], [57], [65].

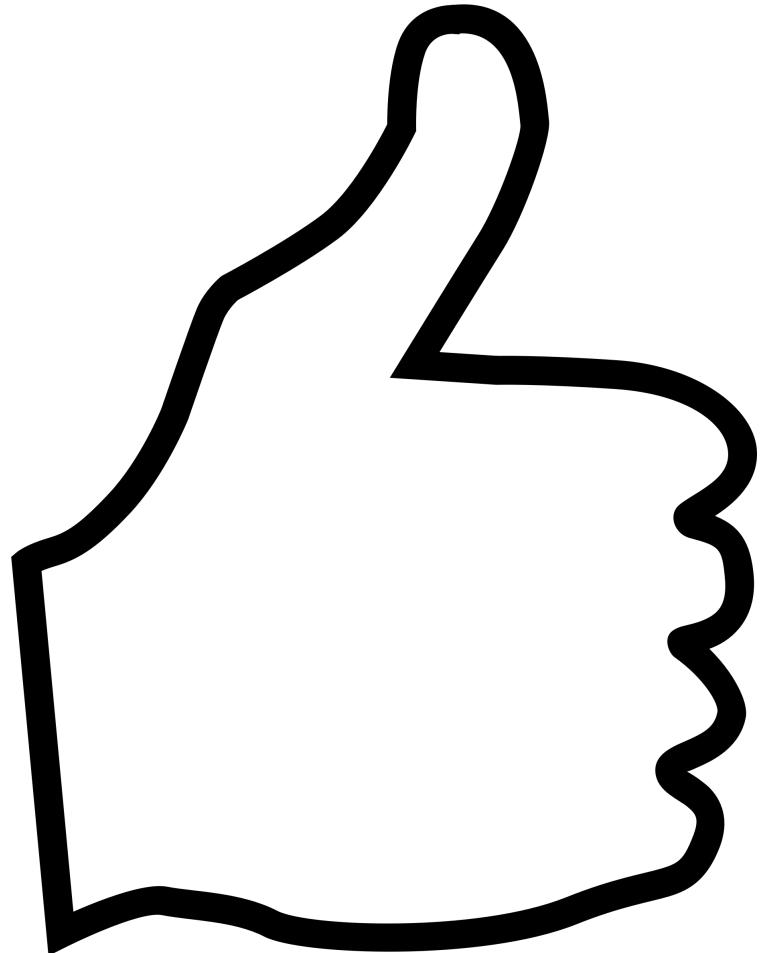
By definition, test suites only encode a partial specification of a program. A patch that is correct with respect to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program





Motto:

**YOU ARE NOT PRESENTING THE PAPER.  
YOU ARE PRESENTING THE WORK.**



- The audience will only remember 3 things.
- Tell a story.
- Never confuse your listeners.



Average conference attendee.

The \$\$\$\$\$\$ question:

**WHAT SHOULD THOSE THREE  
THINGS BE?**



(Average audience member.)

## CLG's Goal

1. The *exciting and important* problem I am solving.
2. The *key nugget of awesomeness* underlying the approach.
3. 1–2 major result(s).
4. “That paper/person seems cool, I want to read it/talk to her!”

# Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
`{yke, kstolee}@iastate.edu`

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
`clegoues@cs.cmu.edu`

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
`brun@cs.umass.edu`

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea: SearchRepair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Localizes* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

1. *The exciting and important problem I am solving.*
2. *The key nugget of awesomeness underlying the approach.*
3. *1–2 major result(s).*
4. *“That paper/person seems cool, I want to read it/talk to her!”*

# Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
`{yke, kstolee}@iastate.edu`

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
`clegoues@cs.cmu.edu`

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
`brun@cs.umass.edu`

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea. SearchRepair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Localizes* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

1. **Automatic defect repair that produces high-quality patches.**
2. *The key nugget of awesomeness underlying the approach.*
3. **1–2 major result(s).**
4. **“That paper/person seems cool, I want to read it/talk to her!”**

# Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
`{yke, kstolee}@iastate.edu`

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
`clegoues@cs.cmu.edu`

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
`brun@cs.umass.edu`

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg, TrpAutoRepair, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea: SearchRepair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Localizes* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

1. Automatic defect repair that produces high-quality patches.
2. SMT-based semantic search, which looks for code based on what it should do.
3. 1–2 major result(s).
4. “That paper/person seems cool, I want to read it/talk to her!”

# Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
`{yke, kstolee}@iastate.edu`

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
`clegoues@cs.cmu.edu`

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
`brun@cs.umass.edu`

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea: SearchRepair:

- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Localizes* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

1. Automatic defect repair that produces high-quality patches.
2. SMT-based semantic search, which looks for code based on what it should do.
3. Empirically wins.
4. “That paper/person seems cool, I want to read it/talk to her!”

# Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
[{yke, kstolee}@iastate.edu](mailto:{yke, kstolee}@iastate.edu)

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
[clegoues@cs.cmu.edu](mailto:clegoues@cs.cmu.edu)

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
[brun@cs.umass.edu](mailto:brun@cs.umass.edu)

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea: SearchRepair:

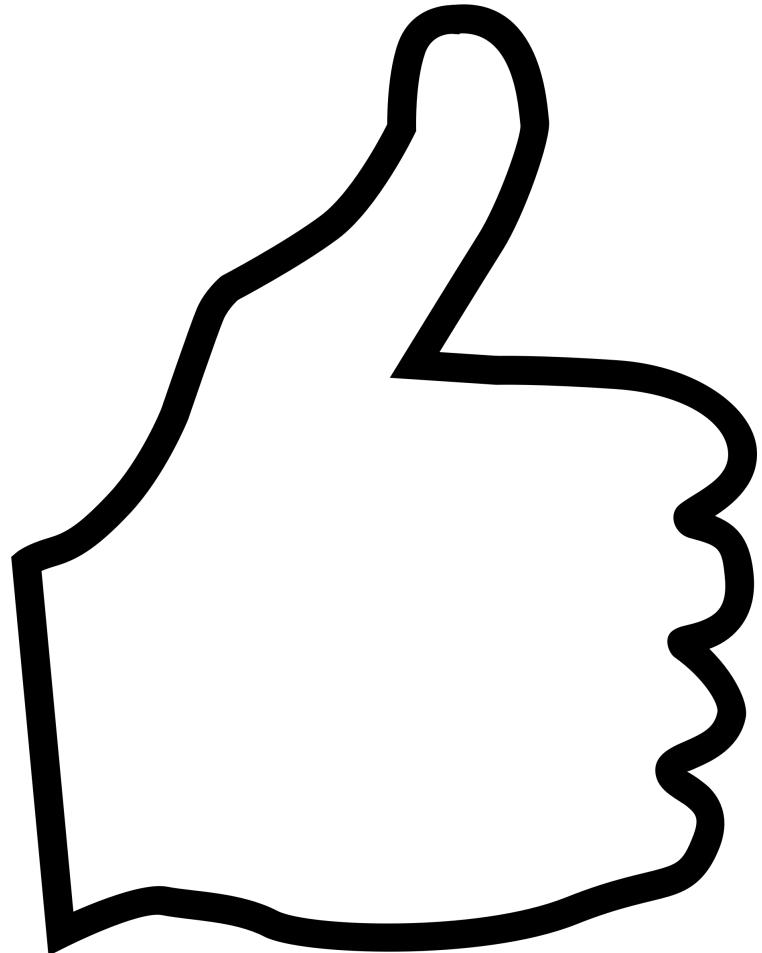
- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Localizes* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

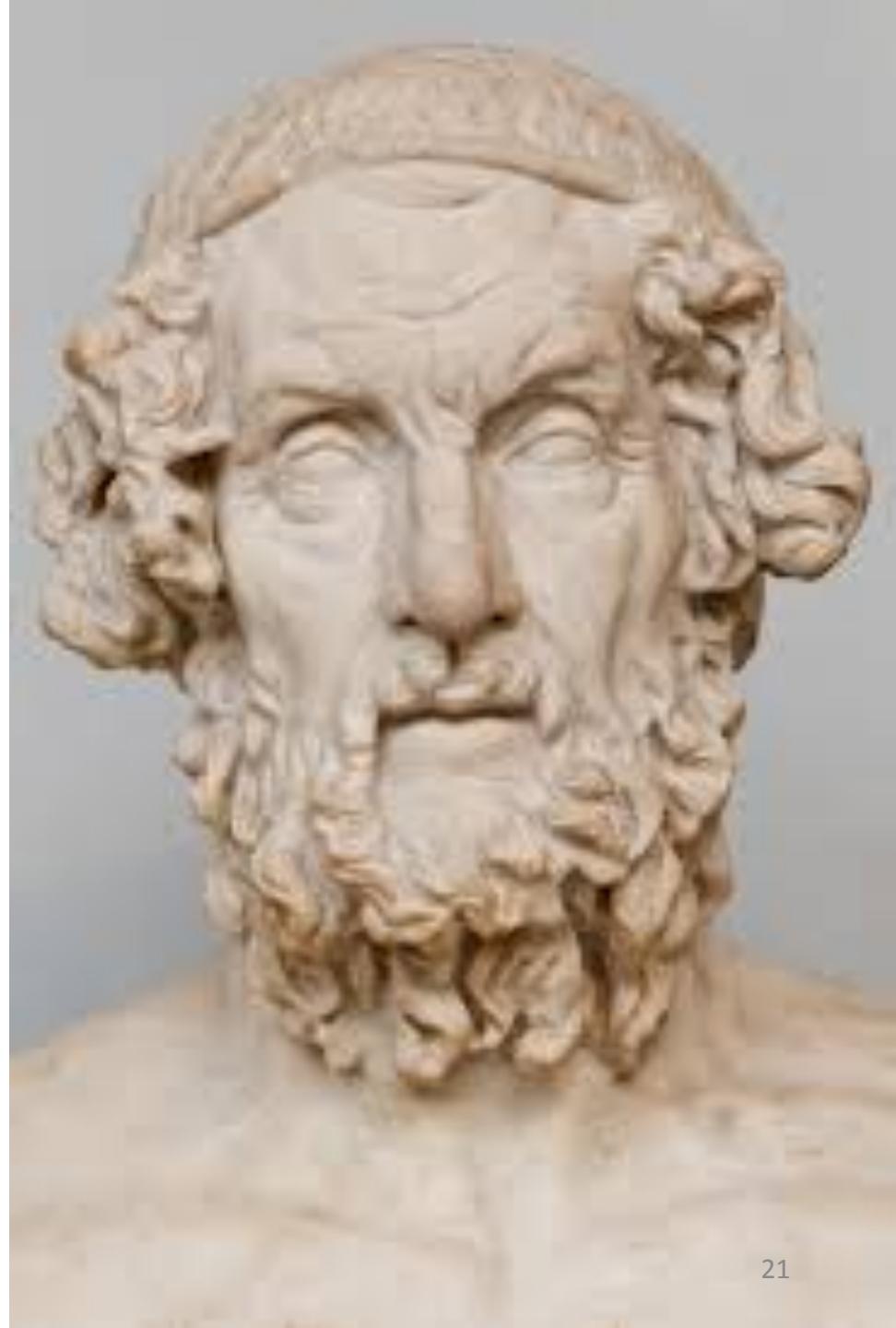
By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

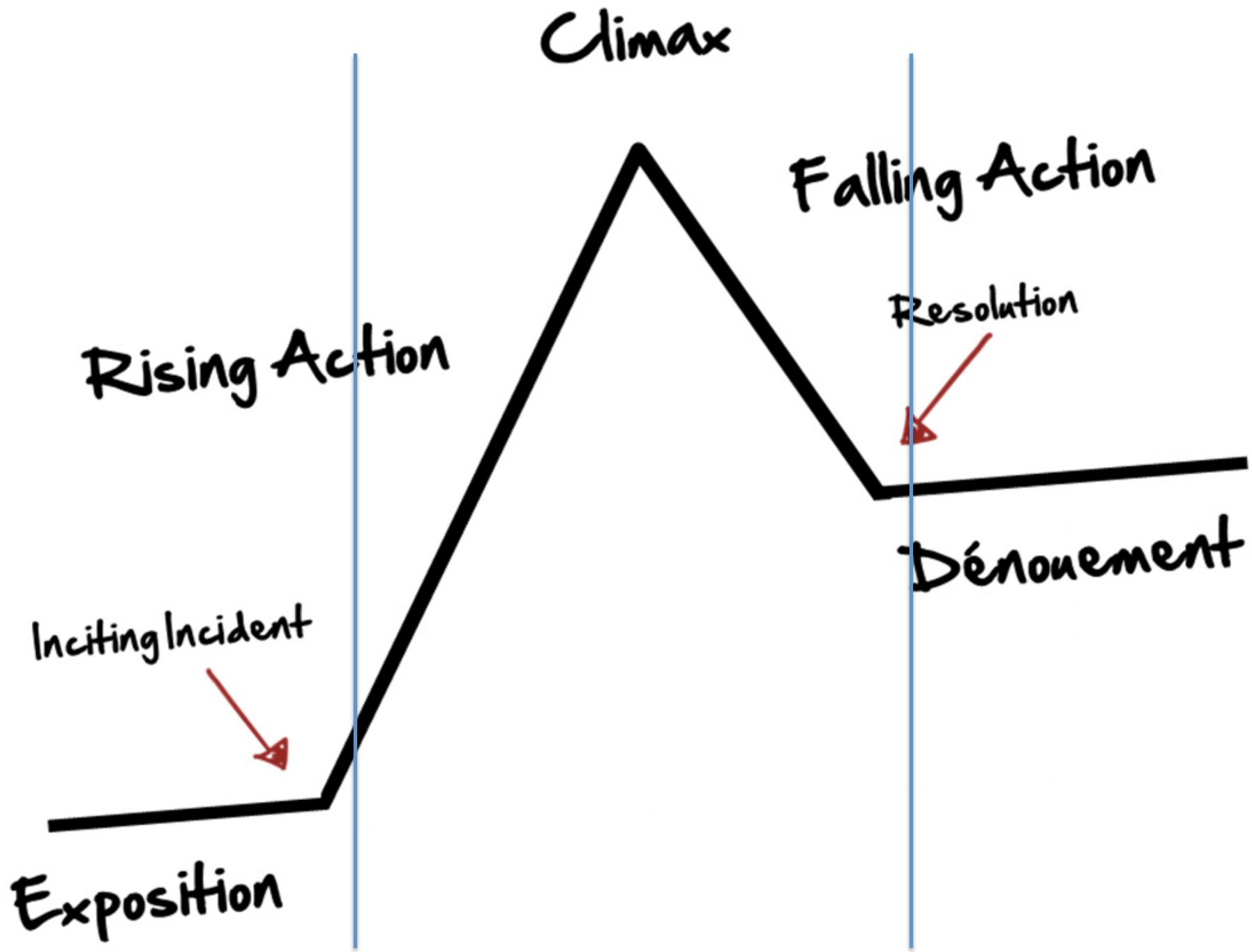
1. Automatic defect repair that produces high-quality patches.
2. SMT-based semantic search, which looks for code based on what it should do.
3. Empirically wins.
4. “Look, there she is, hey, let’s go talk to her!”

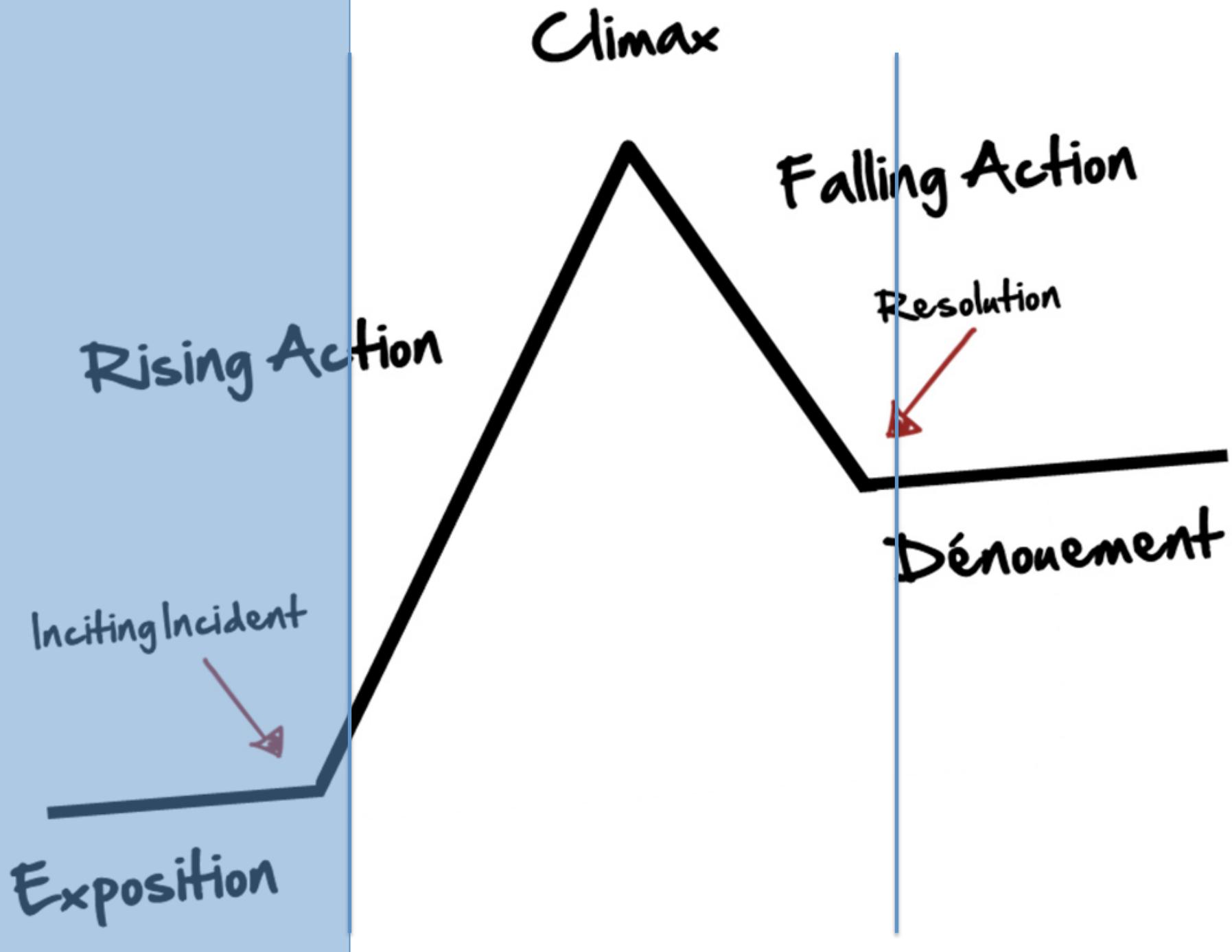


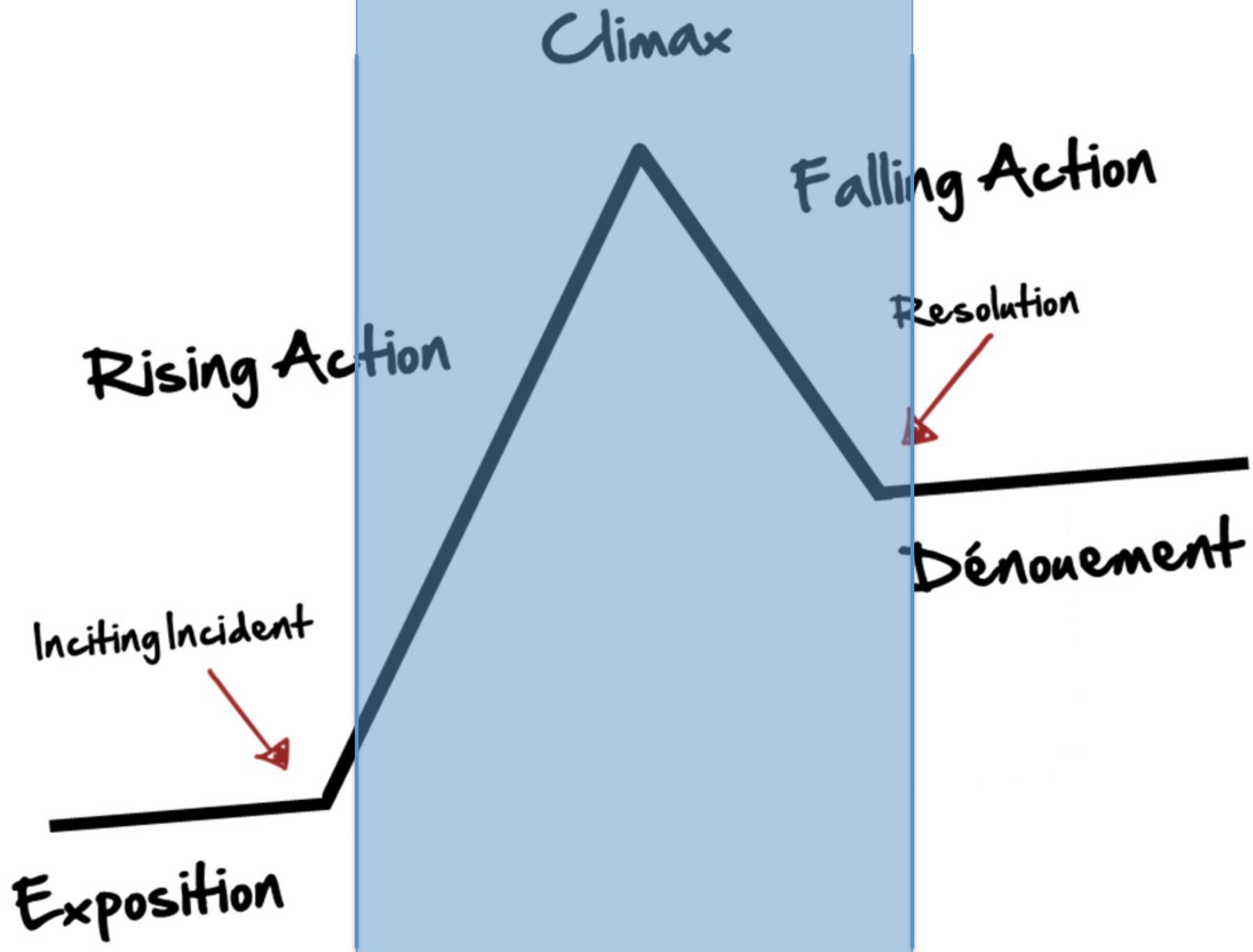
- Your audience will only remember 3 things.
- Tell a story.
- Never confuse your listeners.

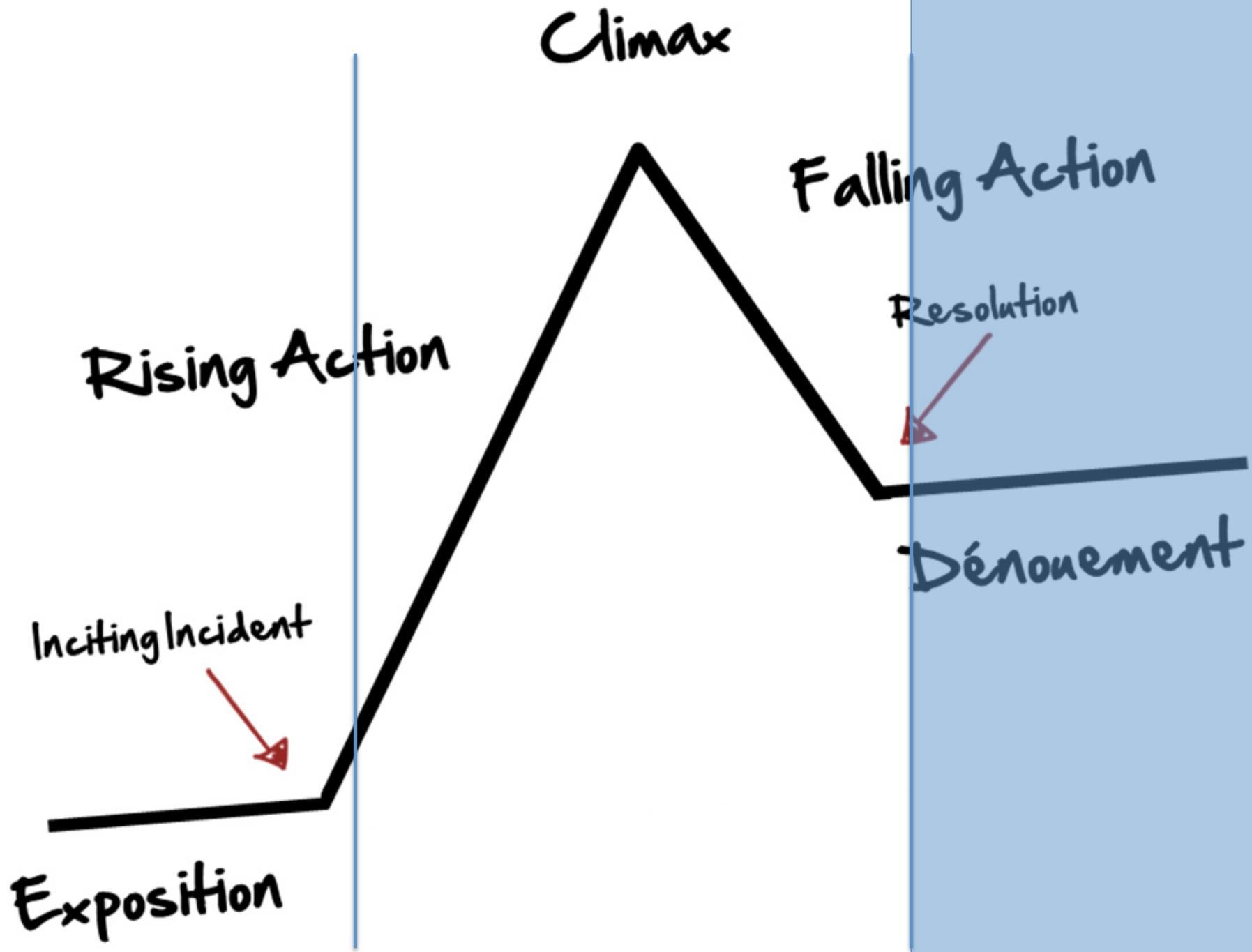
- *Interesting.*
- Simple, and not overly detailed.
- Selectively repetitive.
- Coherent narrative arc.

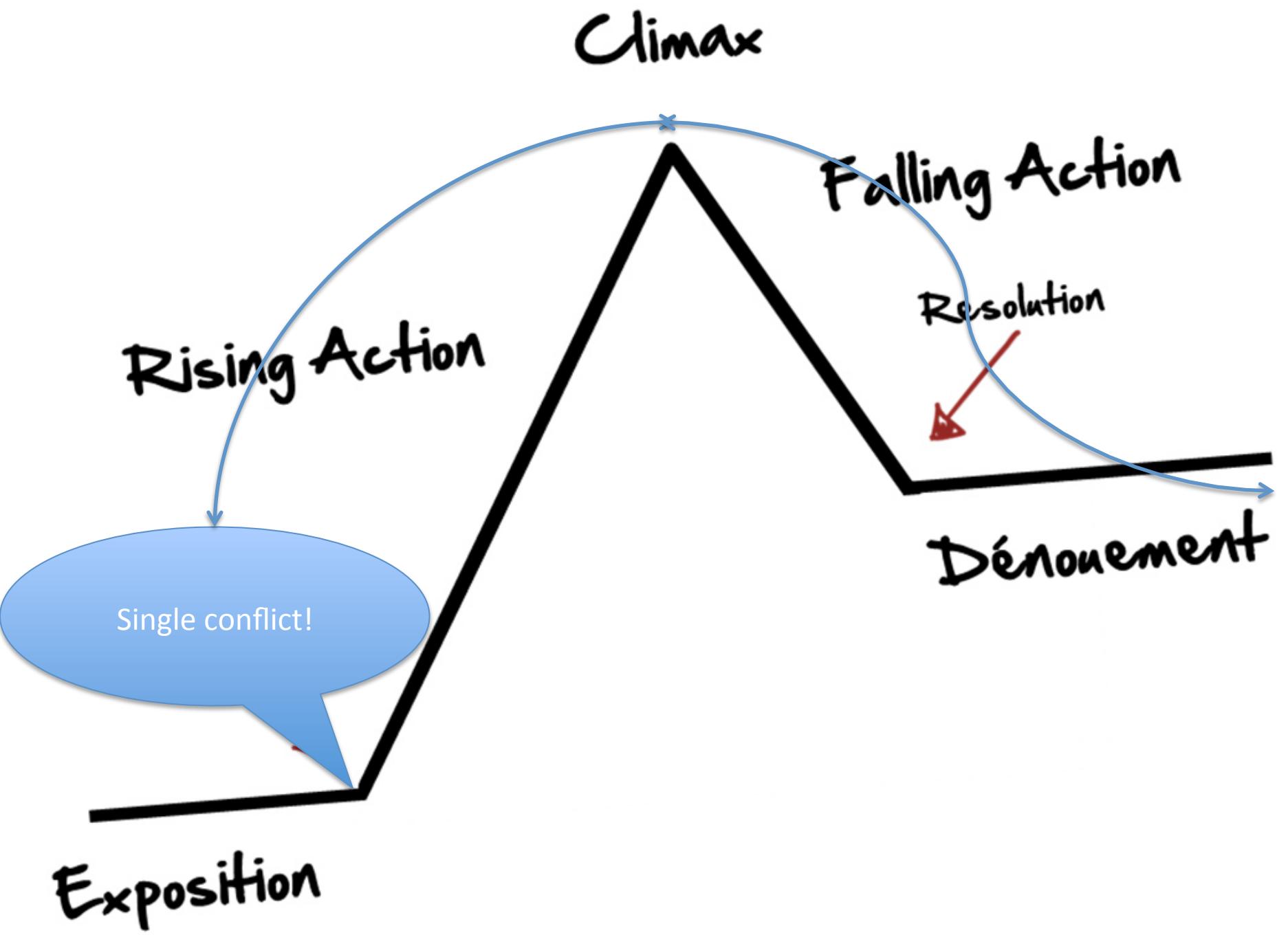


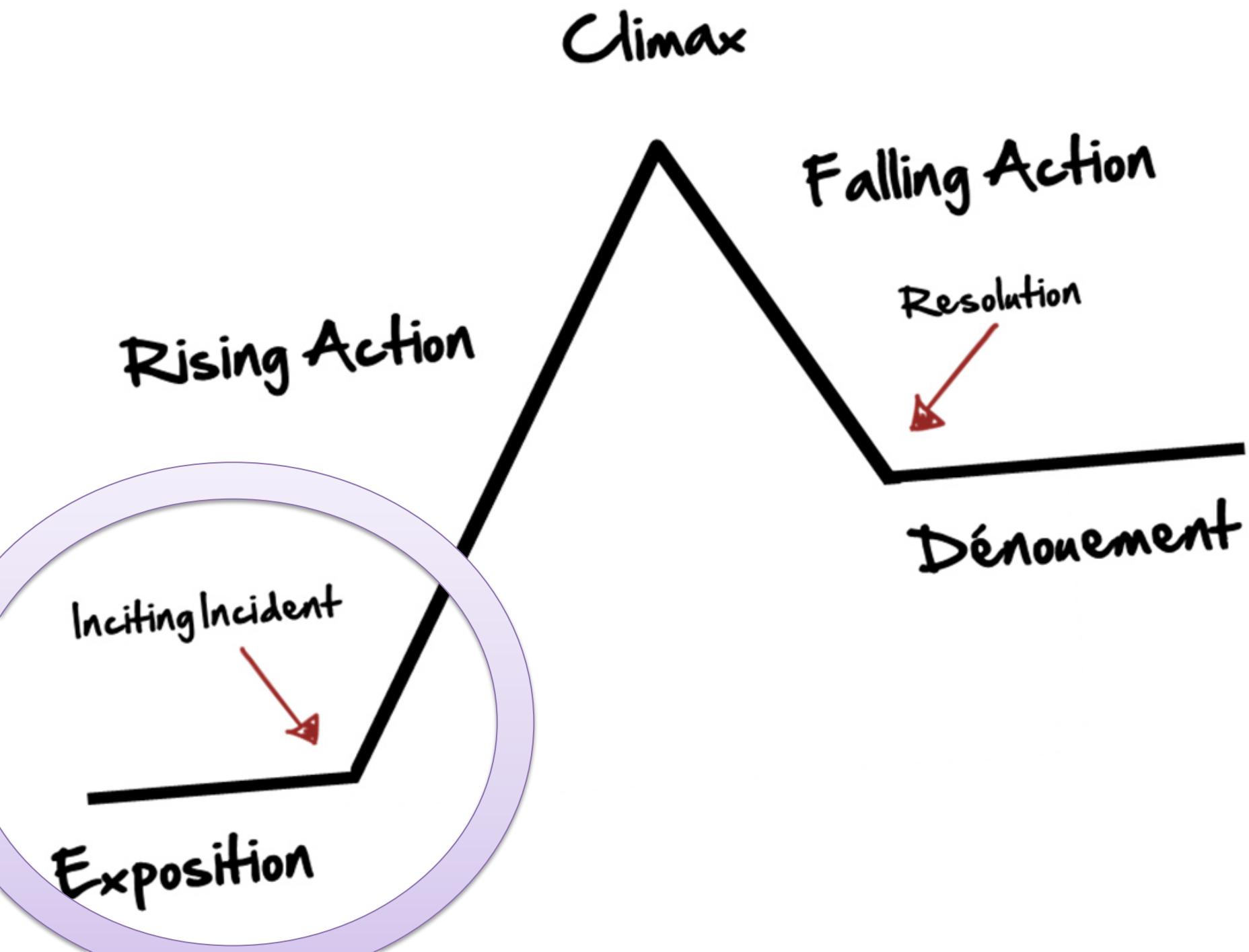












# Exposition, conflict

- *Important, interesting* problem that I am solving.
- Show, don't tell: motivating example, story, easy-to-grasp soundbites.
  - Sometimes a reasonable place to *delicately* mention related or previous work.
- Will guide/motivate the subsequent events of the story; focus on *one type* of motivation.

# Repairing Programs with Semantic Code Search

Yalin Ke Kathryn T. Stolee  
Department of Computer Science  
Iowa State University  
[{yke, kstolee}@iastate.edu](mailto:{yke, kstolee}@iastate.edu)

Claire Le Goues  
School of Computer Science  
Carnegie Mellon University  
[clegoues@cs.cmu.edu](mailto:clegoues@cs.cmu.edu)

Yuriy Brun  
College of Information and Computer Science  
University of Massachusetts, Amherst  
[brun@cs.umass.edu](mailto:brun@cs.umass.edu)

**Abstract**—Automated program repair can potentially reduce debugging costs and improve software quality but recent studies have drawn attention to shortcomings in the quality of automatically generated repairs. We propose a new kind of repair that uses the large body of existing open-source code to find potential fixes. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. We present SearchRepair, a repair technique that addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites. We find that SearchRepair repairs 150 (19%) of 778 benchmark C defects written by novice students, 20 of which are not repaired by GenProg, TrpAutoRepair, and AE. We compare the quality of the patches generated by the four techniques by measuring how many independent, not-used-during-repair tests they pass, and find that SearchRepair-repaired programs pass 97.3% of the tests, on average, whereas GenProg-, TrpAutoRepair-, and AE-repaired programs pass 68.7%, 72.1%, and 64.2% of the tests, respectively. We conclude that SearchRepair produces higher-quality repairs than GenProg, TrpAutoRepair, and AE, and repairs some defects those tools cannot.

## I. INTRODUCTION

Buggy software costs the global economy billions of dollars annually [8], [60]. One major reason software defects are so expensive is that software companies must dedicate considerable developer time [75] to manually finding and fixing bugs in their software. Unfortunately, manual bug repair, the industry standard, is largely unable to keep up with the volume of defects in extant software [2]. Despite their established detrimental impact on a company’s bottom line, known defects ship in mature software projects [45], and many defects, including those that are security-critical, remain unaddressed for long periods of time [32].

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as GitHub, BitBucket, and SourceForge. Because many programs include routines, data structures, and designs that have been previously implemented in other software projects [11], [12], [24], we posit that, if a method or component of a software system contains a defect, with high probability, there exists a similar but correct version of that component in some publicly accessible software project. The research challenge lies in how to automatically find and use such implementations to repair bugs.

Our key idea is to use *semantic code search* [68] over

existing open-source code to find correct implementations of buggy components and methods, and use the results to automatically generate patches for software defects. Semantic search identifies code by what it does, rather than by syntactic keywords. We develop SearchRepair, a new technique predicated on our idea: SearchRepair:

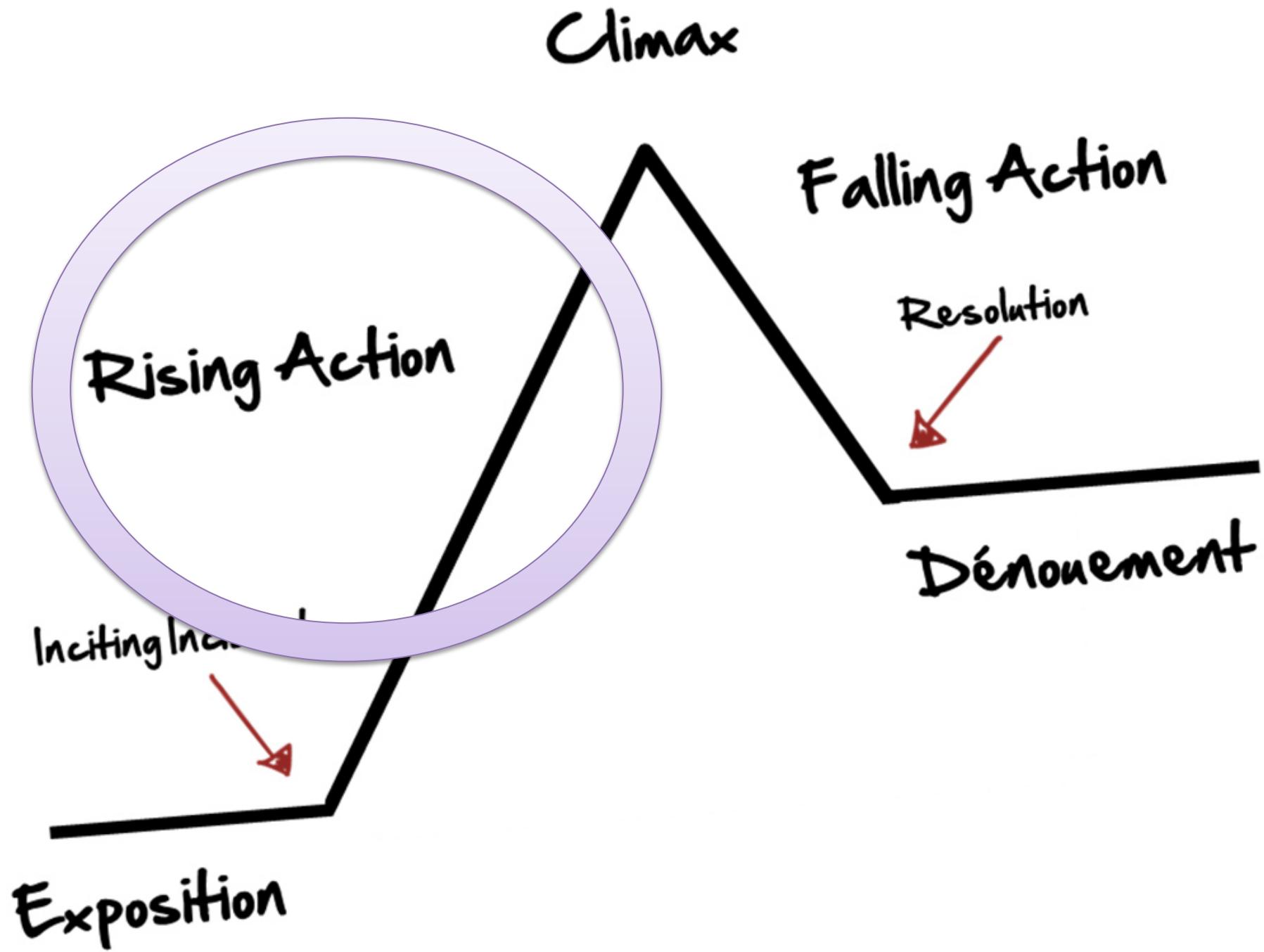
- 1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
- 2) *Locates* a defect to likely buggy program fragments.
- 3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired functional behavior as SMT constraints.
- 4) *Searches* the database, using state-of-the-art constraint solvers, for fragments that satisfy such a profile. These fragments become potential patches when contextualized and inserted into the buggy regions, replacing the original potentially faulty code.
- 5) *Validates* each potential patch against the program test suite to determine if it indeed repairs the defect in question.

To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

Our goal with SearchRepair is to produce high quality patches while still addressing a broad range of defects. A key feature of a high quality patch, whether human- or tool-generated, is that it generalizes to the full, desired, often unwritten specification of correct program behavior. This is a challenge for automatic repair techniques (e.g., [3], [7], [10], [11], [15], [16], [18], [19], [21], [28], [33], [35], [39], [42], [48], [49], [50], [51], [52], [54], [56], [57], [61], [69], [70], [73], [74], [76]), many of which use test suites to guide and evaluate patching efforts. Modern test-suite guided repair techniques, particularly those following a *generate-and-validate* paradigm (i.e., heuristically constructing and then testing large numbers of candidate repairs), although typically general and scalable, often produce poor-quality patches that overfit to the specification test suites used to guide patch generation [20], [57], [65].

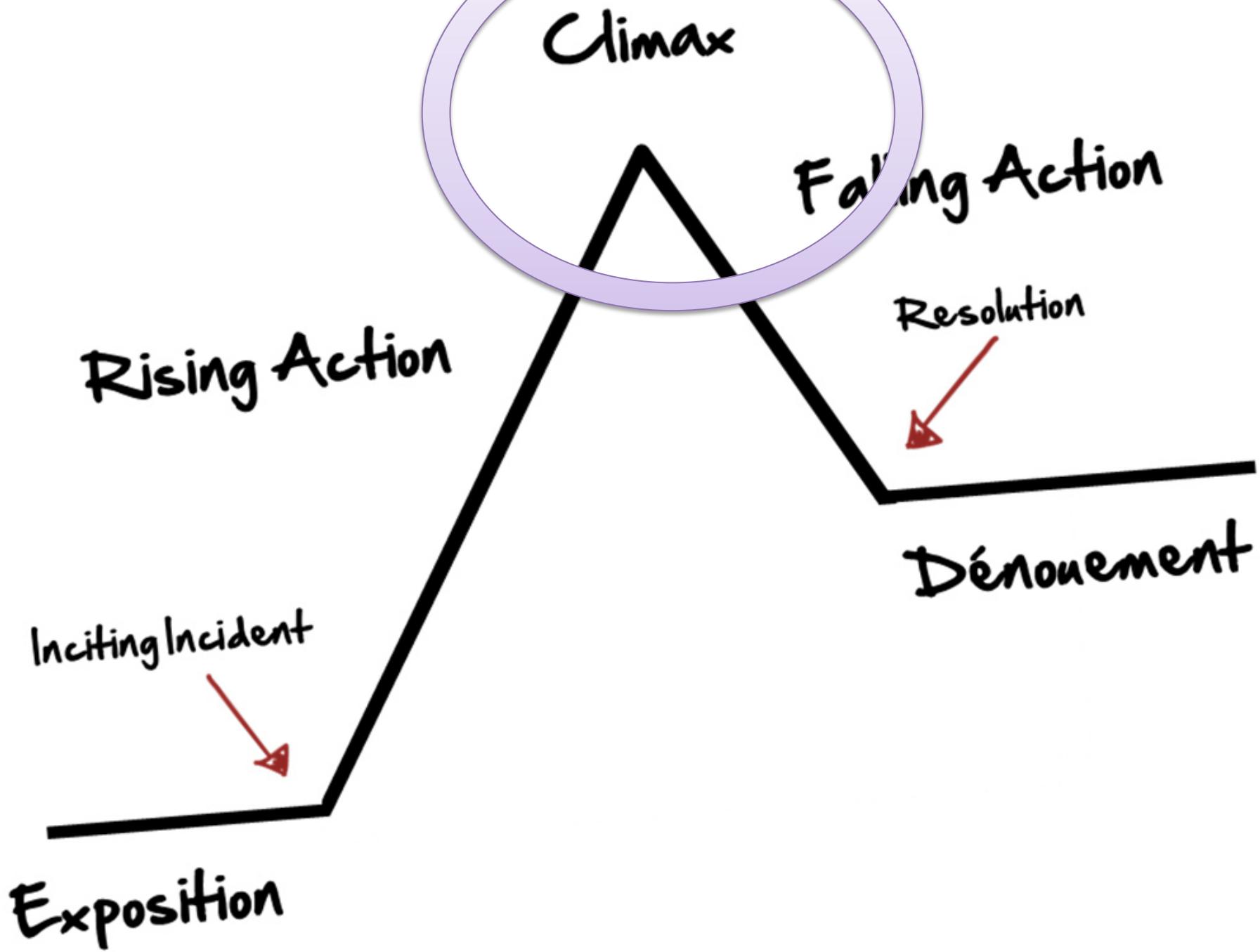
By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning phenomenon of *overfitting* to an objective function, where the program

- Options:
  - Example showing how semantic search works.
  - Example walking through hypothetic program repair/semantic search combo use case.
  - Compelling results highlighting “quality problem” in previous results.
- *But I will only choose one of them.*



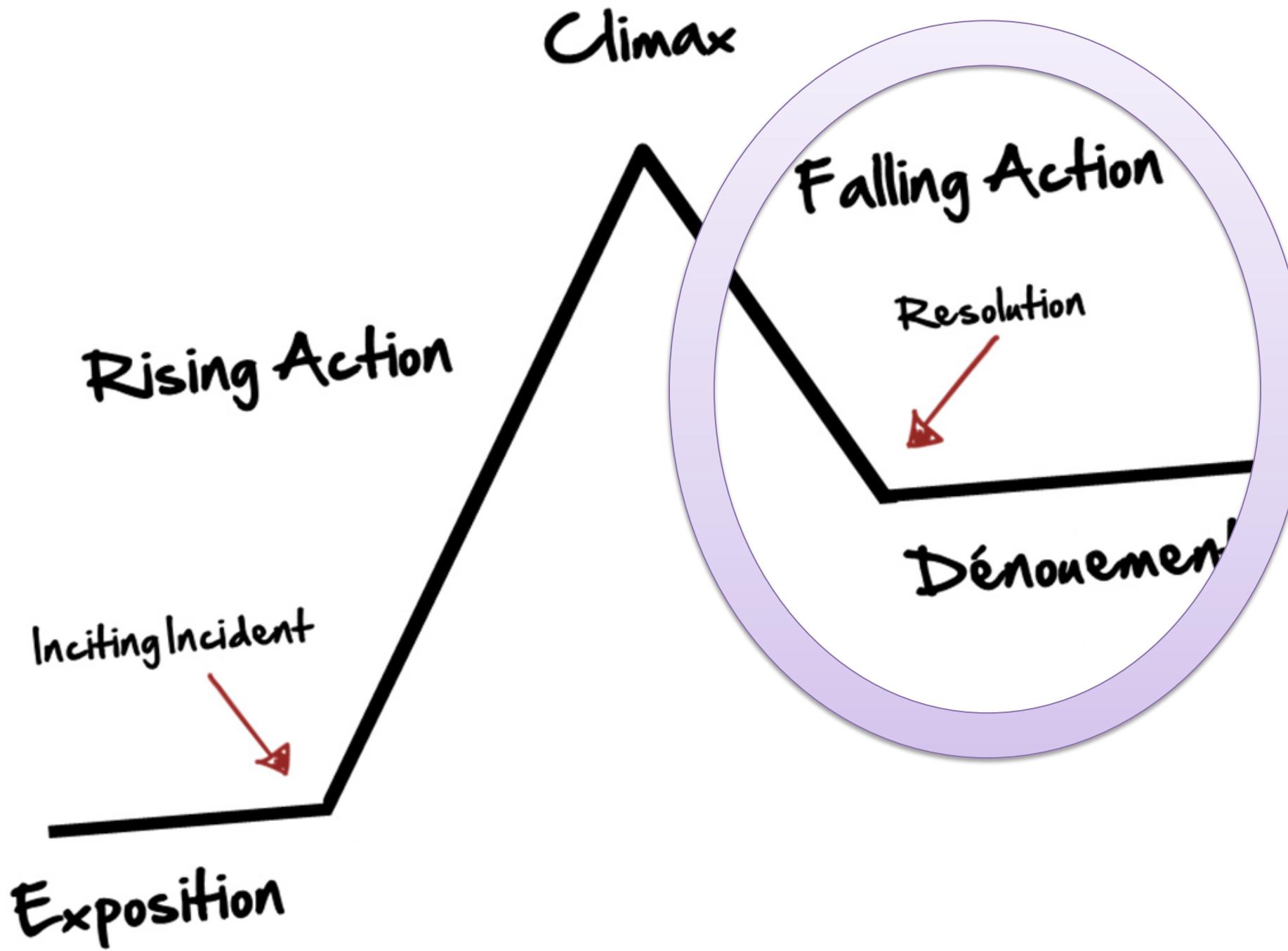
# Inciting incident + rising action

- Middle of story: Key technical insight.
- High-level outline of the approach.
- If you're not sure if a detail is high-level enough, it's probably not.
- (More on how to approach “the middle” in 10 –11 slides)



# Climax

- Results presentation: *selected* key results.
- Emphasis on the type of experimental methodology used, experimental question(s)
  - Calling back to your exposition!
  - Remember: oral communication is often cyclic/repetitive.

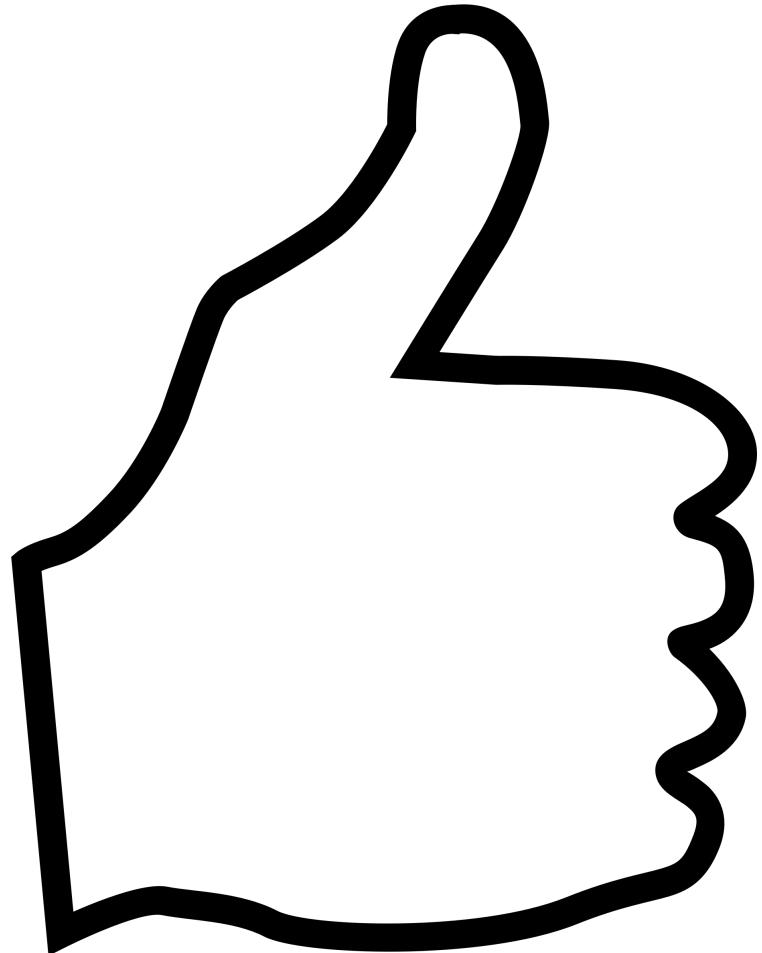


# Falling action

- Observations/implications.
- Another place related work *might* make an appearance.
- Possibly, future work.
  - (CLG thinks this is pointless, but acknowledges the existence/validity of opposing viewpoints.)

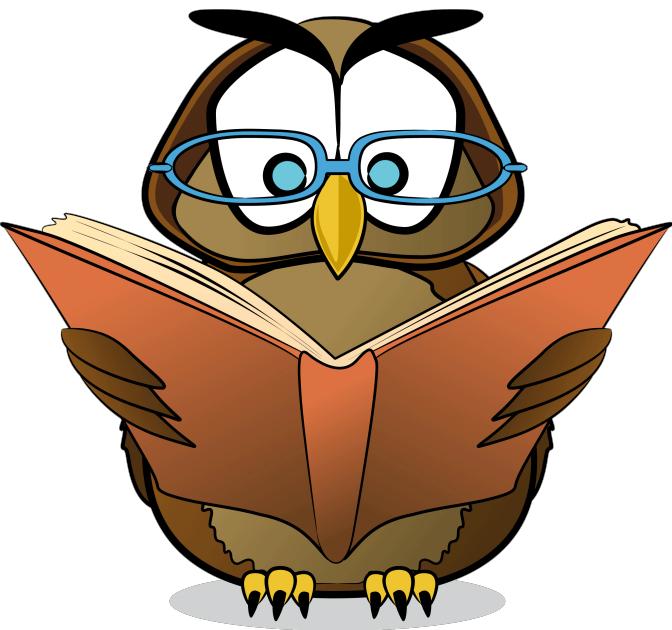
# Conclusion/Denouement

- “Say what you’re going to say, then say it, then **say what you said.**”
- Wrap up pithily.
- Remind me of the three things you want me to remember.



- Your audience will only remember 3 things.
- Tell a story.
- **Never confuse your listeners.**





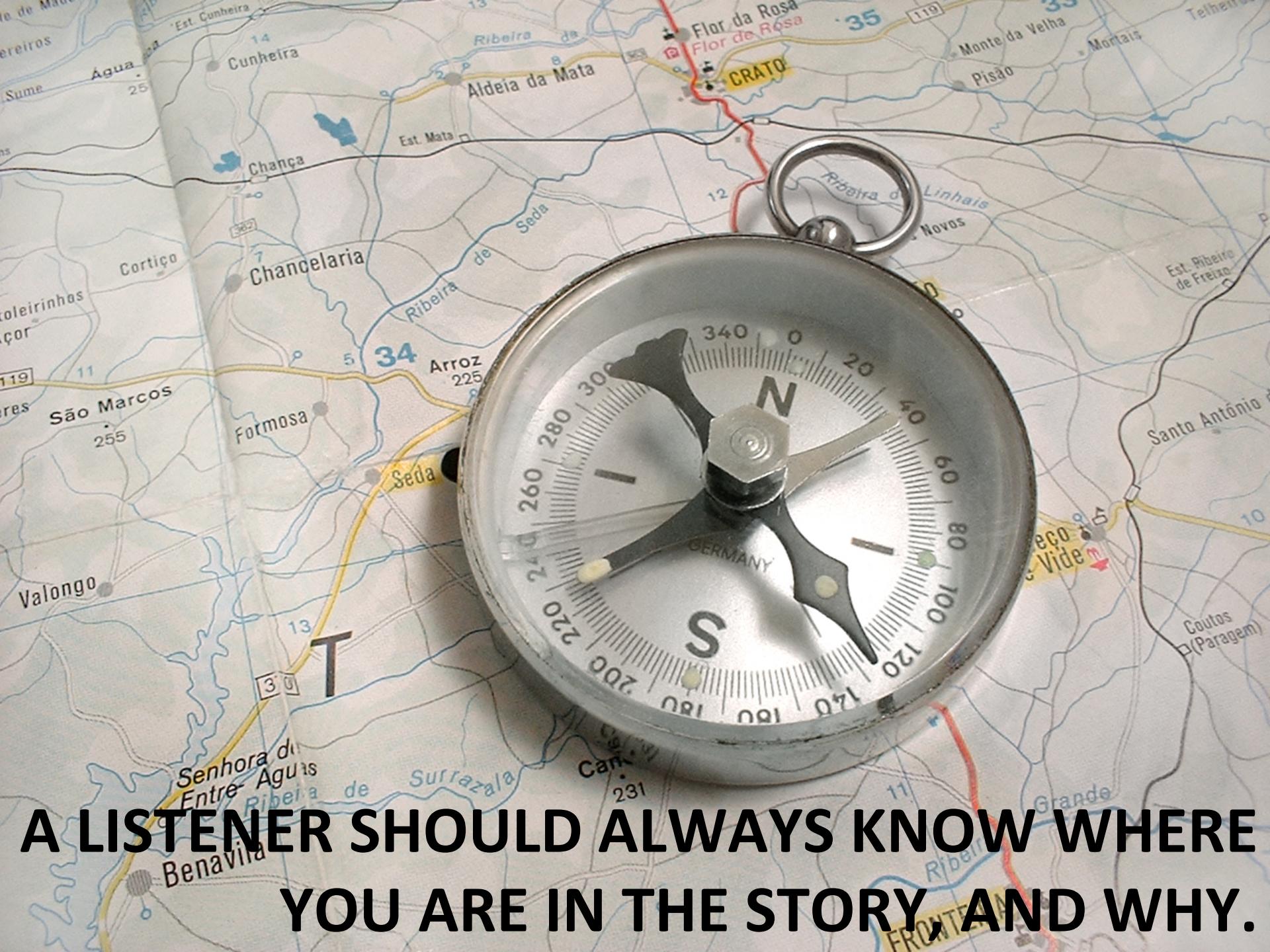
When confused, readers can:

- Pause, reflect.
- Reread confusing passage.
- Go back to review a previous section.
- Look something up.

When confused, listeners can:

- Possibly interrupt to ask a question.
- Furtively look it up.
- Give up and start reading their email.





A LISTENER SHOULD ALWAYS KNOW WHERE  
YOU ARE IN THE STORY, AND WHY.

# Why?

- Listener has to synthesize what you're saying into the story.
- If she doesn't know why you're telling her something, she won't know where to "put" a piece of information in the overall picture.
- Result: listener is anxious, and likely to forget key pieces of information before they're needed!

# Implicit

I've done this several times already.

- Signpost as you go.
  - “This is important, because...”
  - Return to your outline slide, if you’re using it.
- Only introduce necessary information, and only when it is necessary.
- *Strongly* avoid forward references.
- *Strictly* avoid use-before-def.

I violated this rule 11 slides ago.

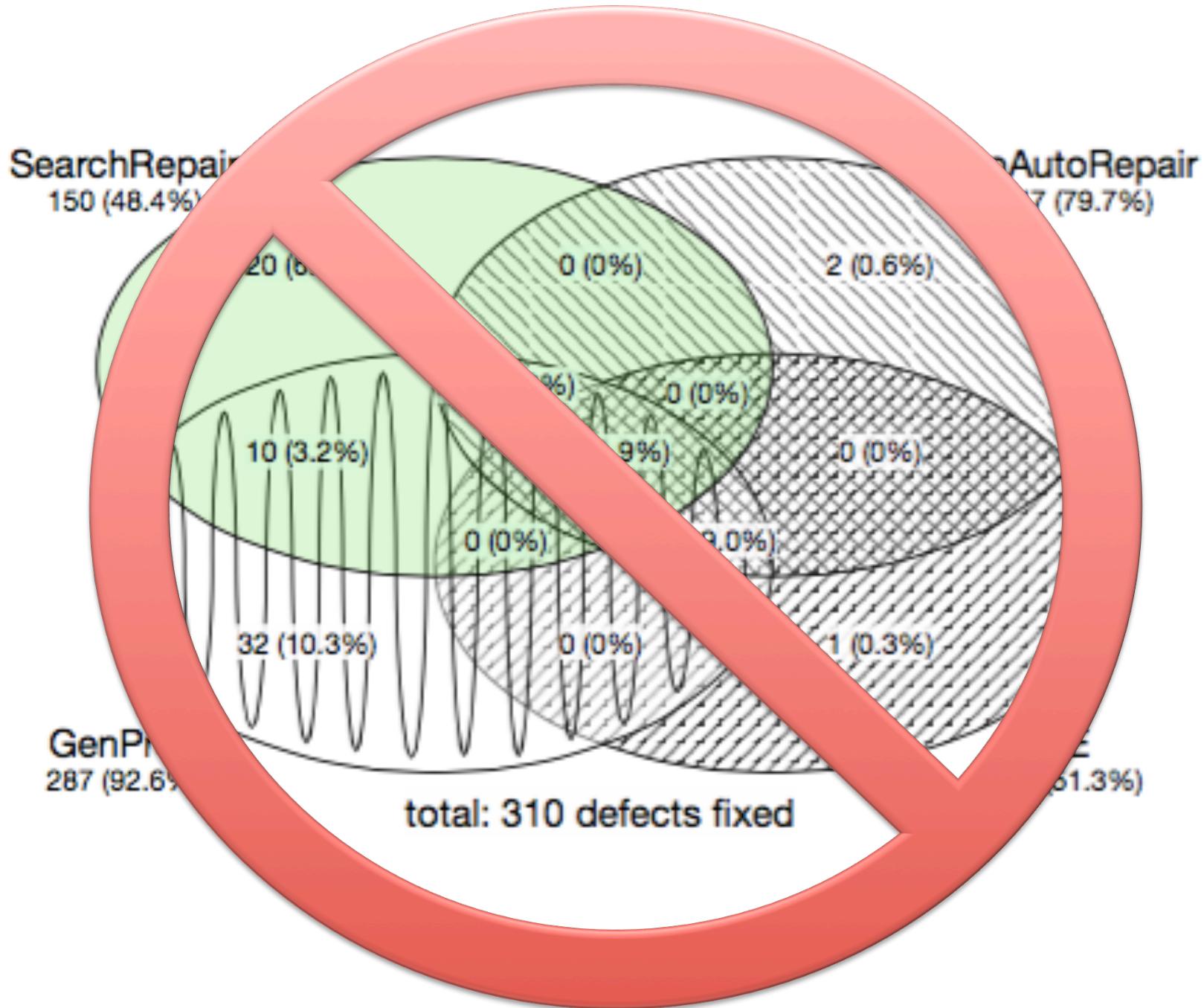
**DO NOT VISUALLY OVERWHELM  
YOUR LISTENERS.**



**DO NOT VISUALLY OVERWHELM  
YOUR LISTENERS.**

**DO NOT VISUALLY OVERWHELM  
YOUR LISTENERS.**

**BEWARE THE RESULTS  
PRESENTATION.**

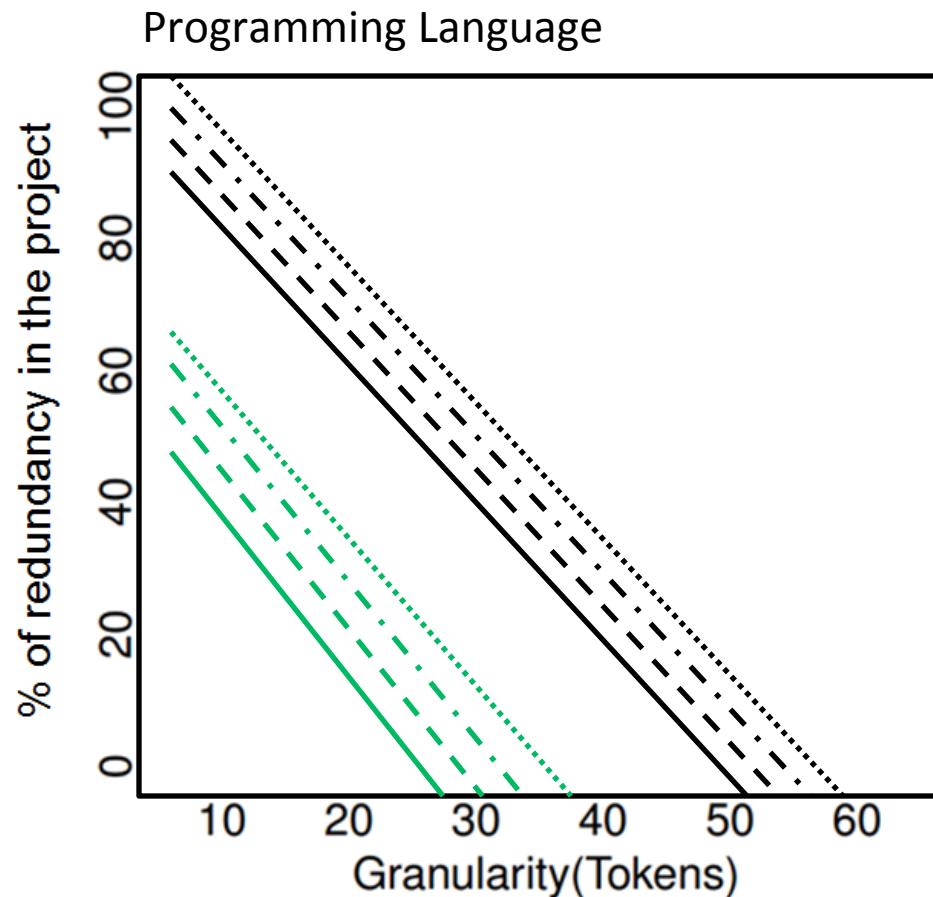


(Slide borrowed from my student Mauricio.)

# How to read the graph?



What % of this project can be reconstructed from the corpus?



```
newNumber = oldNumber1 + oldNumber2;
```

---

program	repair	AE	GenProg	TrpAutom	total
checksum	0	0	8	0	29
digits	0	7	30	19	91
grade	5	2	2	2	26
media	68	58	108	93	368
smaller	73	71		119	255
syllable	4	11		14	309
total repaired	150	159	287	247	778

---

Program	Description	LOC	Bug Type	Time (s)
gcd	example		loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text filtering	1146	segmentation fault	34
look-u	dictionary	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite loop	55
units	metric conversion	1504	segmentation fault	109
deroff	document processing	2236	segmentation fault	131
indent	code processing	9906	infinite loop	546
flex	lexical analyzer generator	74	segmentation fault	230
openldap	directory protocol	29	non-overflow denial of service	665
ccrypt	encryption utility	751	segmentation fault	330
lighttpd	web server	51895	heap buffer overflow (code)	394
atris	arcade game	21553	local stack buffer overflow	80
php	scripting language	764489	integer overflow	56
wu-ftpd	FTP server	67029	format string vulnerability	2256
leukocyte	computational biology	6718	segmentation fault	360
tiff	image processing		segmentation fault	108
imagemagick	image processing	126	wrong output	2160

Program	Description	LOC	Bug Type	Time (s)

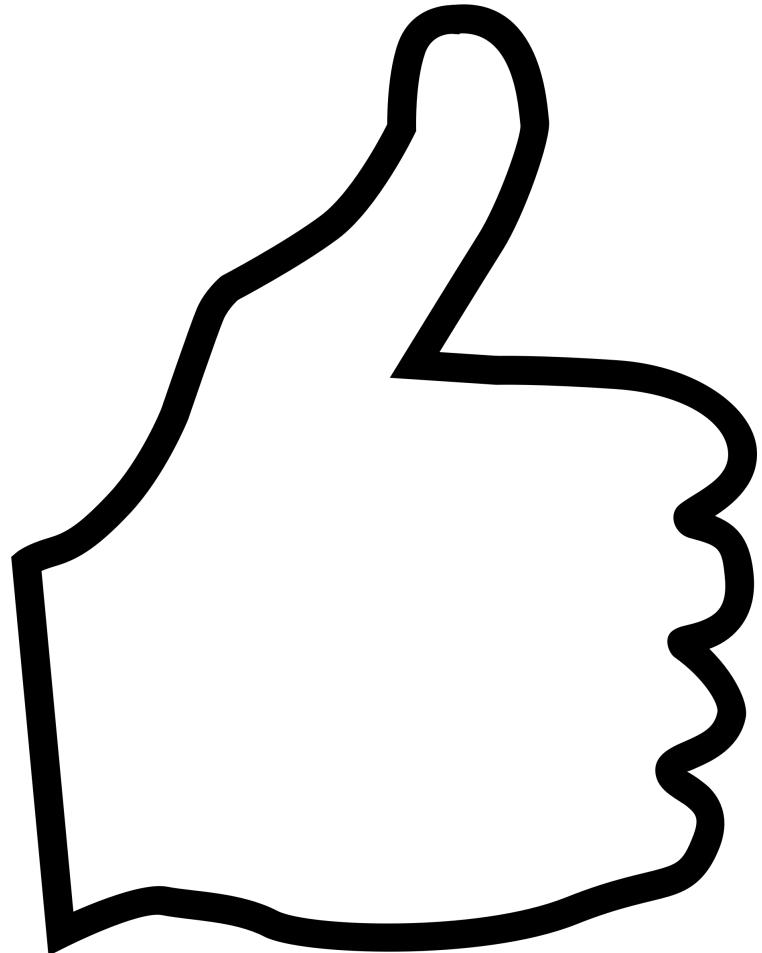
Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text processing	1146	segmentation fault	34
look-u	dictionary lookup	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite loop	55
units	metric conversion	1504	segmentation fault	109
deroff	document processing	2236	segmentation fault	131
indent	code processing	9906	infinite loop	546
flex	lexical analyzer generator	18774	segmentation fault	230

Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text processing	1140	segmentation fault	34
look-u	dictionary lookup	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite loop	55
units	metric conversion	1504	segmentation fault	109
deroff	document processing	2230	segmentation fault	131
indent	code processing	9900	infinite loop	546
flex	lexical analyzer generator	18774	segmentation fault	230
openldap	directory protocol	292598	non-overflow denial of service	665
ccrypt	encryption utility	7515	segmentation fault	330
lighttpd	webserver	51895	heap buffer overflow (vars)	394
atris	graphical game	21553	local stack buffer exploit	80
php	scripting language	764489	integer overflow	56
wu-ftpd	FTP server	67029	format string vulnerability	2256
leukocyte	computational biology	6718	segmentation fault	360
tiff	image processing	84061	segmentation fault	108
imagemagick	image processing	450510	wrong output	2160

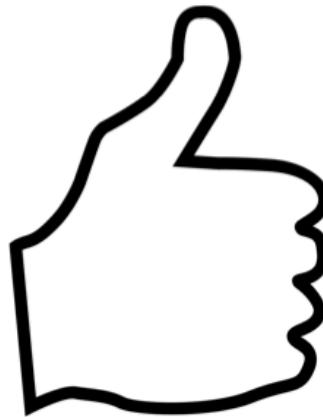


- Your audience will only remember 3 things.
- Tell a story.
- Never confuse your listeners.



Motto:

**YOU ARE NOT PRESENTING THE PAPER.  
YOU ARE PRESENTING THE WORK.**



- The audience will only remember 3 things.
- Tell a story.
- Never confuse your listeners.

8

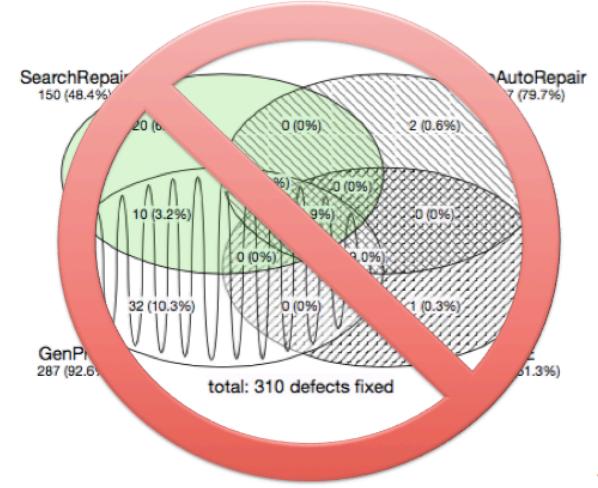
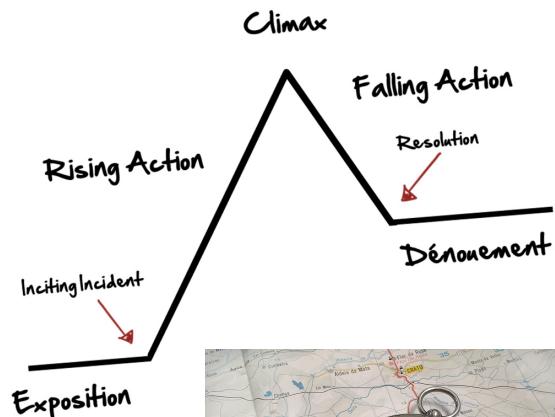


(Average audience member.)



## CLG's Goal

1. The *exciting and important* problem I am solving.
2. The *key nugget of awesomeness* underlying the approach.
3. 1–2 major result(s).
4. “That paper/person seems cool, I want to read it/talk to her!”



43