Perspectives on Structuring a Research Presentation
or
"One motto, three rules of thumb"

Claire Le Goues

SSSG, September 28, 2015


SEE BACK
for other great
Morton® products
MORTON®
SALT
THIS SALT DOES NOT SUPPLY IODIDE, A NECESSARY NUTRIENT.

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

Repairi... ...e Search

...yn T. Stolee ...Computer Science ...State University ...e, kstolee}@iastate.edu

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

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

...ted program repair can potentially reduce ... prove software quality but recent studies have ... ortcomings in the quality of automatically ge... opose a new kind of repair that uses the large ... urce code to find potential fixes. The key chal... tly finding code semantically similar (but not ... e code and then appropriately integrating th... y program. We present SearchRepair, a repa... esses these challenges by (1) encoding a large d... tten code fragments as SMT constraints on in... (2) localizing a given defect to likely buggy p... d deriving the desired input-output behavior fo... fragments, (3) using state-of-the-art constraint ... database for fragments that satisfy that desired ... g the likely buggy code with these potential pa... ing that the patches repair the bug against pr... find that SearchRepair repairs 150 (19%) of ... defects written by novice students, 20 of which ... by GenProg, TrpAutoRepair, and AE. We compare ... the patches generated by the four techniques by mea... many independent, not-used-during-repair tests they p... find that SearchRepair-repaired programs pass 97.3% of the ... on average, whereas GenProg-, TrpAutoRepair-, and AE-repai... 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 ...riods of time [32].

...t the same time, the expansion of the open-source ...ent has led to many large, publicly accessible source ...bases, such as GitHub, BitBucket, and SourceForge. ... programs include routines, data structures, and ...een previously implemented in other software ..., we posit that, if a method or component ...s a defect, with high probability, there ... version of that component in some ... The research challenge ... implementations ...

Our key idea is to...

existing open-source code to find correct implementa... of buggy components and methods, and use the results ... *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 ... to determine if it indeed repairs the defect in question.

... SearchRepair possible, we first extend our previous ...tic code search [68] to C program fragments. ...t spectrum-based fault localization [36] to ...egions of faulty code and construct input-ou... ...s input to semantic search. Third, we build ... perform semantic code search over the SMT-en... adapt the returned code fragment to the defe... riable renaming, and validate against provi...

Our goal w... to produce high quality patches while stil... range of defects. A key feature of a high... her human- or tool-generated, is that it ge... desired, often unwritten specification of ... ior. This is a challenge for automatic repa... [7], [10], [11], [15], [16], [18], [19], [21]... [39], [42], [48], [49], [50], [51], [52], [54], [56]... [7... [74], [76]), many of which use test su... patching efforts. Modern test-suite guide... particularly those following a *generate-and-*... (i.e., heuristically constructing and then testing ... candidate repairs), although typically general a... produce poor-quality patches that overfit ... test suites used to guide patch gener...

By definition, test suites onl... of correct behavior. A pat... given test suite may th... ated with ... ication. This is ... ning phenomenon ... on, 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
{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.

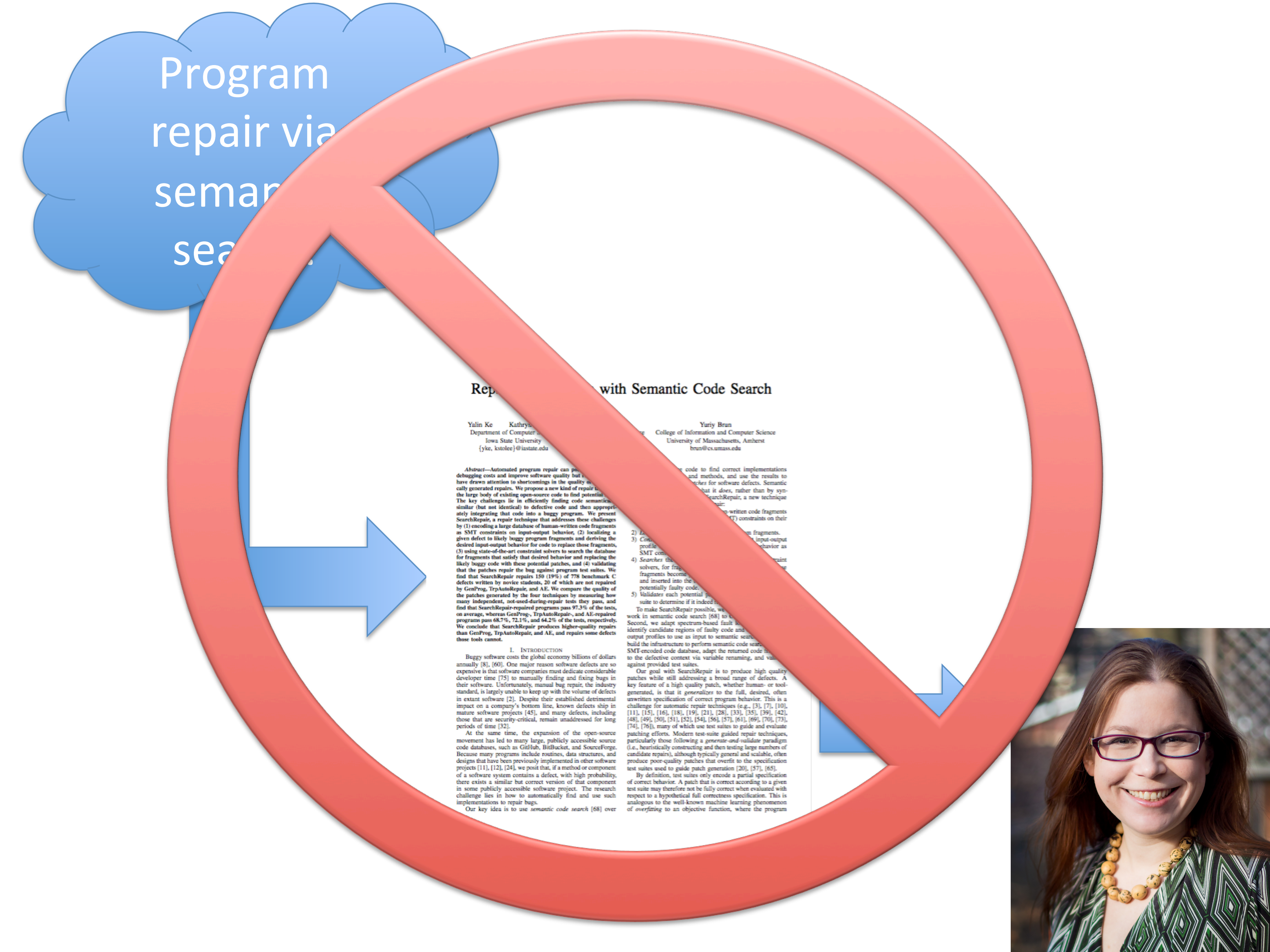
Our key idea is to use *semantic code search* [68] over existing open-source code to find correct implementations of buggy components and methods, and use the results to *automatically generate patches* for software defects. Semantic search identifies code by what it *does*, rather than by syntactic 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


Program
repair via
seman
sea
Rep with Semantic Code Search
Yalin Ke
Yuriy Brun
College of Information and Computer Science
University of Massachusetts, Amherst
I. Introduction

Program repair via semantic search.

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.

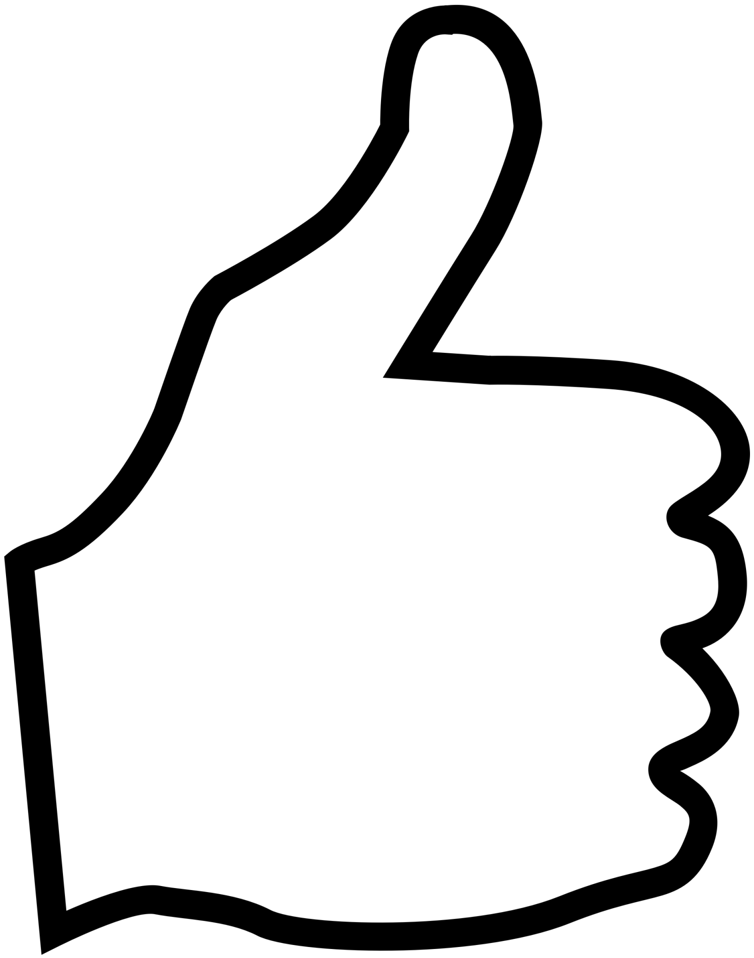
To make SearchRepair possible, we first extend our previous work in semantic code search [68] to C program fragments. Second, we adapt spectrum-based fault localization [36] to identify candidate regions of faulty code and construct input-output profiles to use as input to semantic search. Third, we 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

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

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:

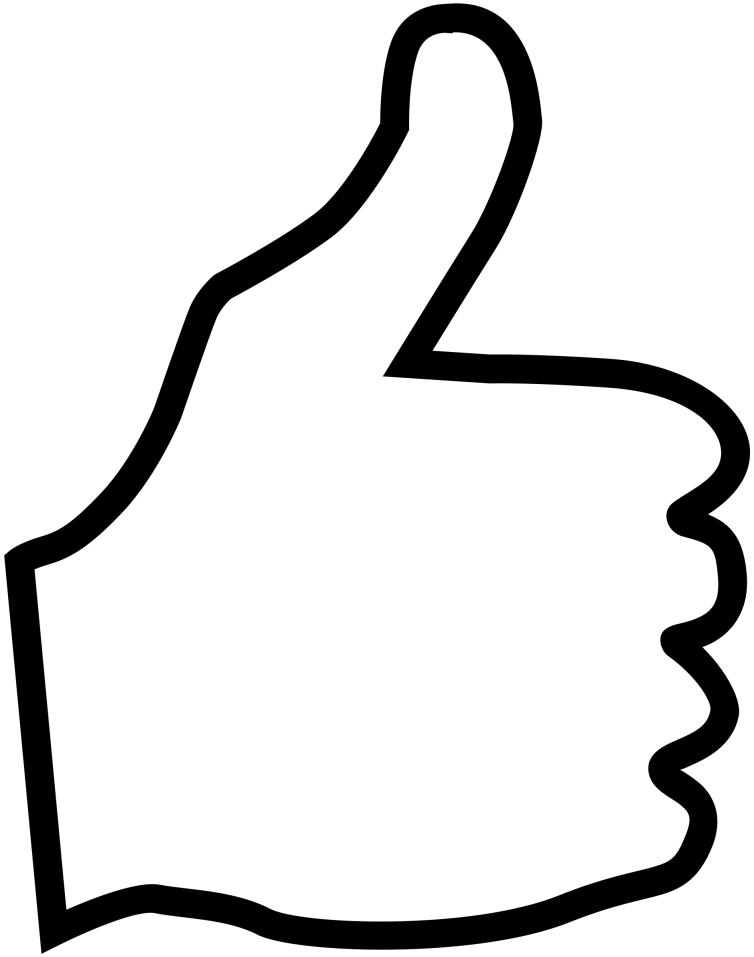
1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
2) *Localizes* a defect to likely buggy program fragments.
3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired 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. **"Look, there she is, hey, let's go talk to her!"**



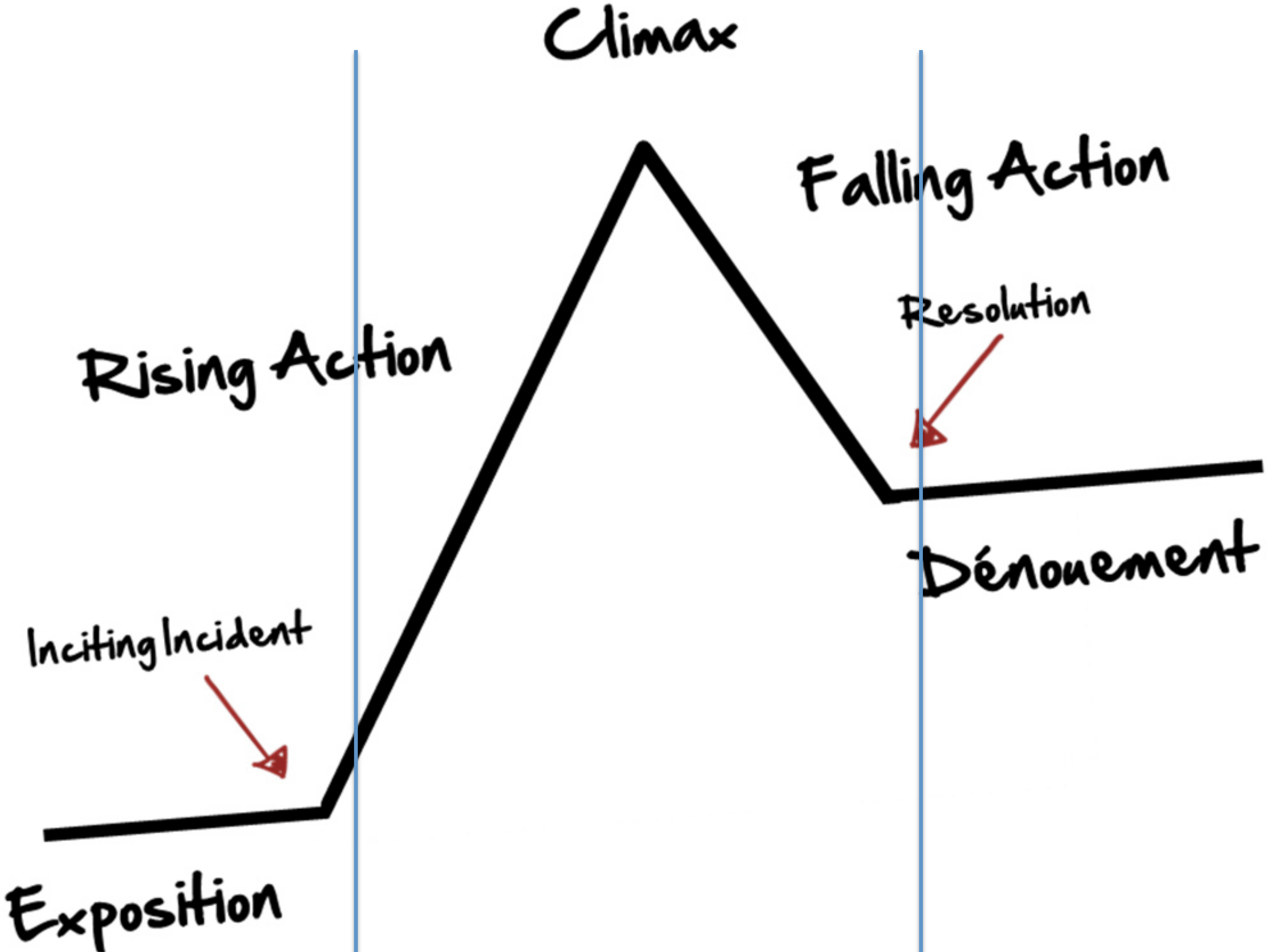
- Your audience will only remember 3 things.
- **Tell a story.**
- Never confuse your listeners.

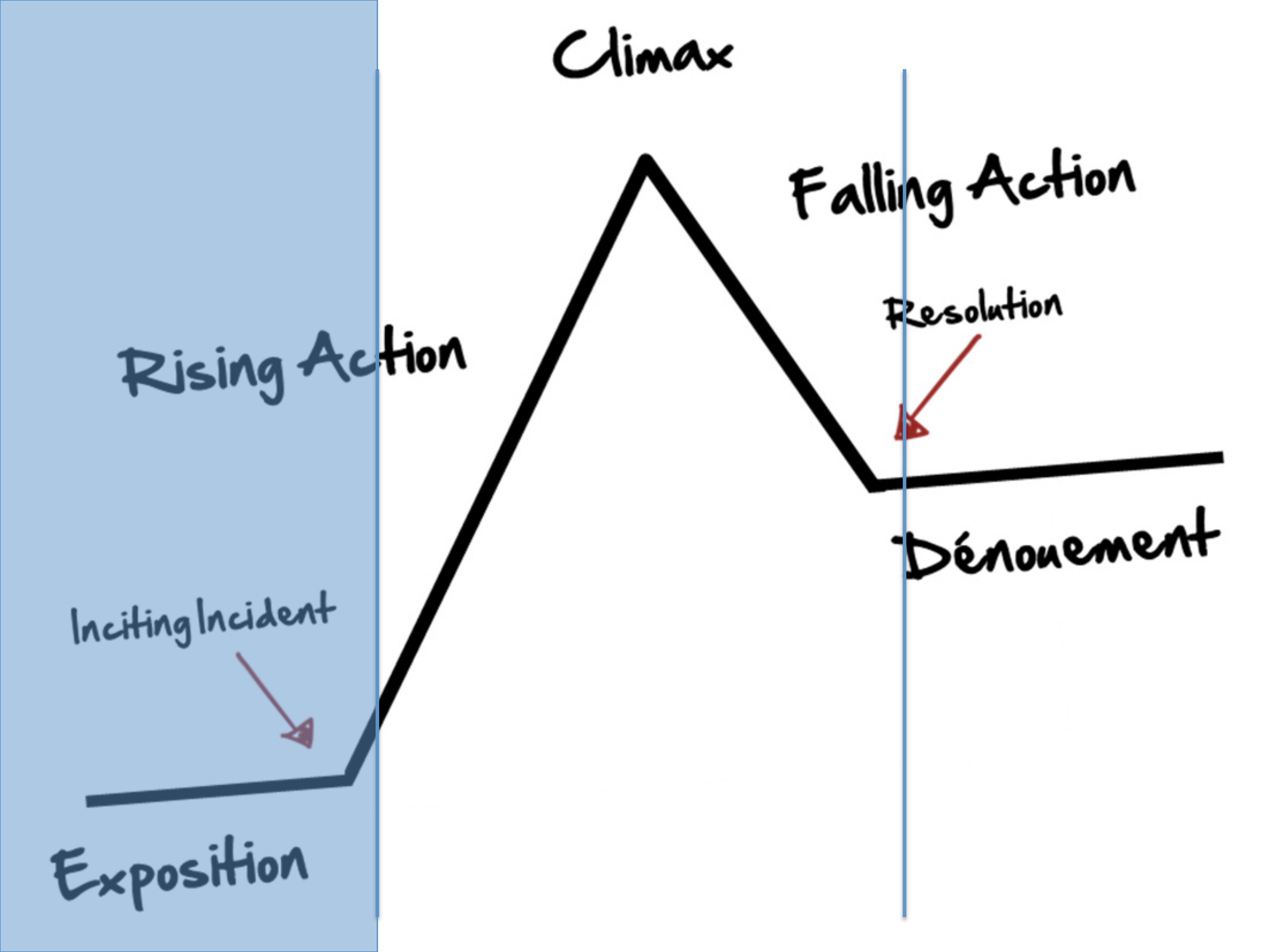
- *Interesting.*

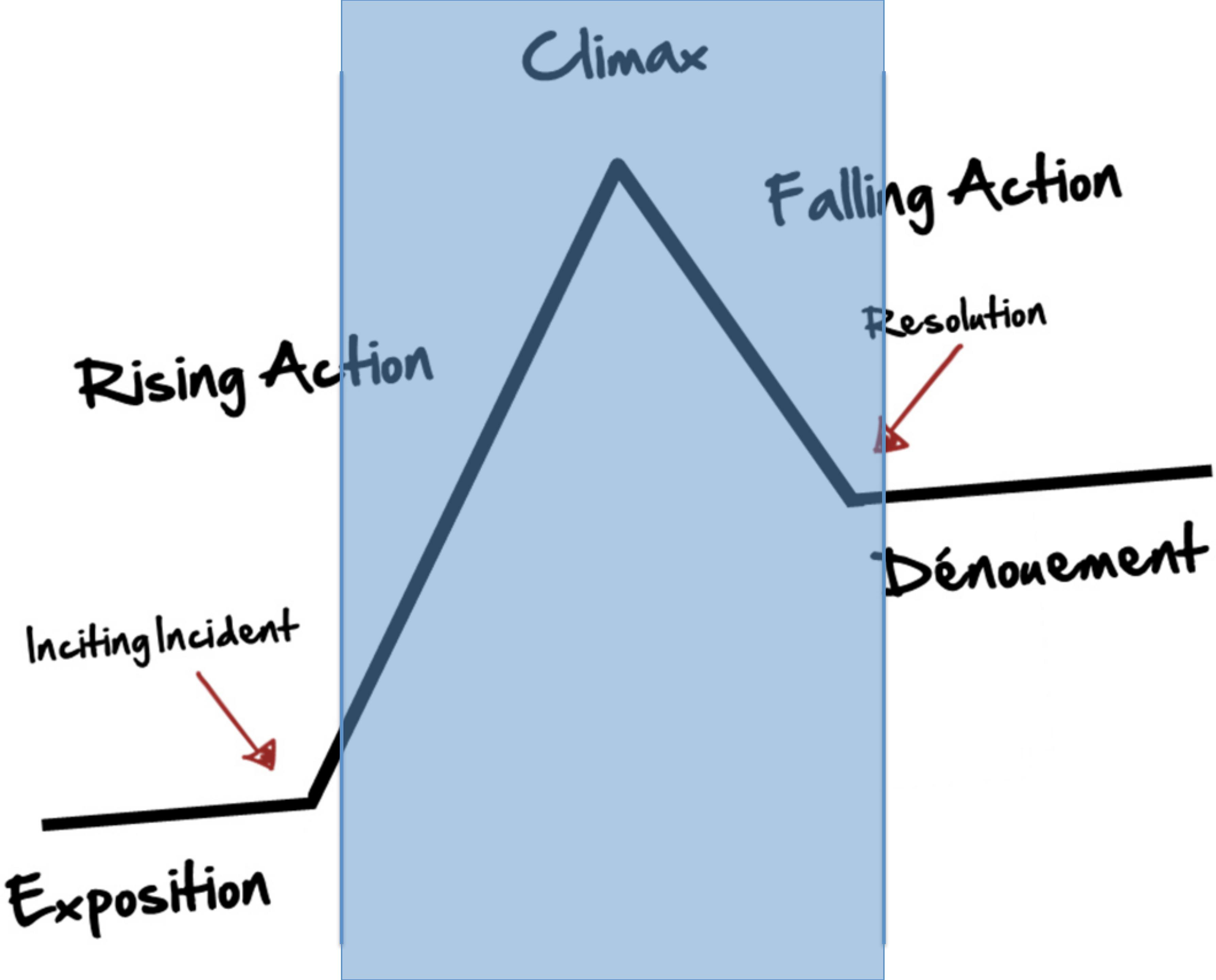
- Simple, and not overly detailed.

- Selectively repetitive.

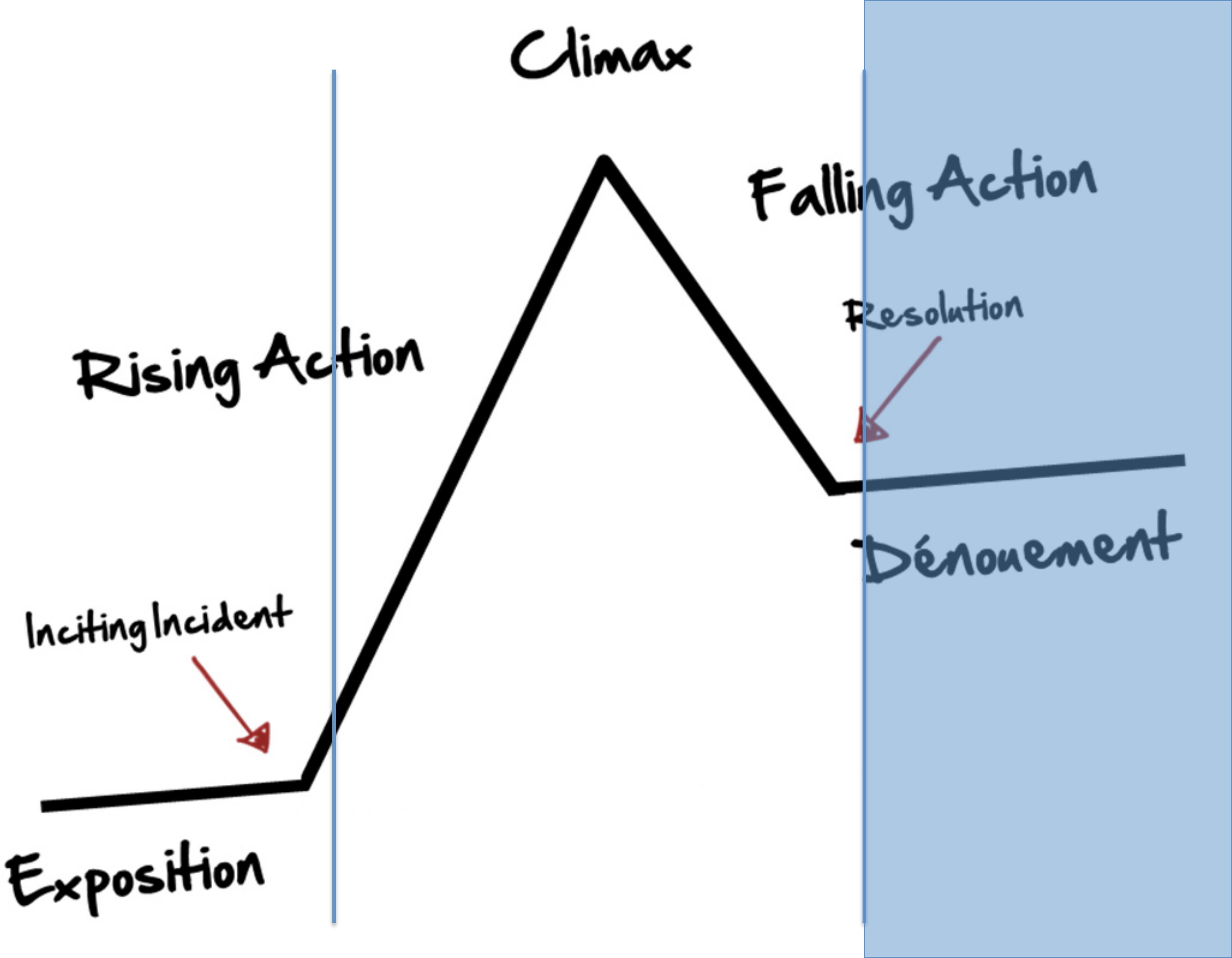
- Coherent narrative arc.

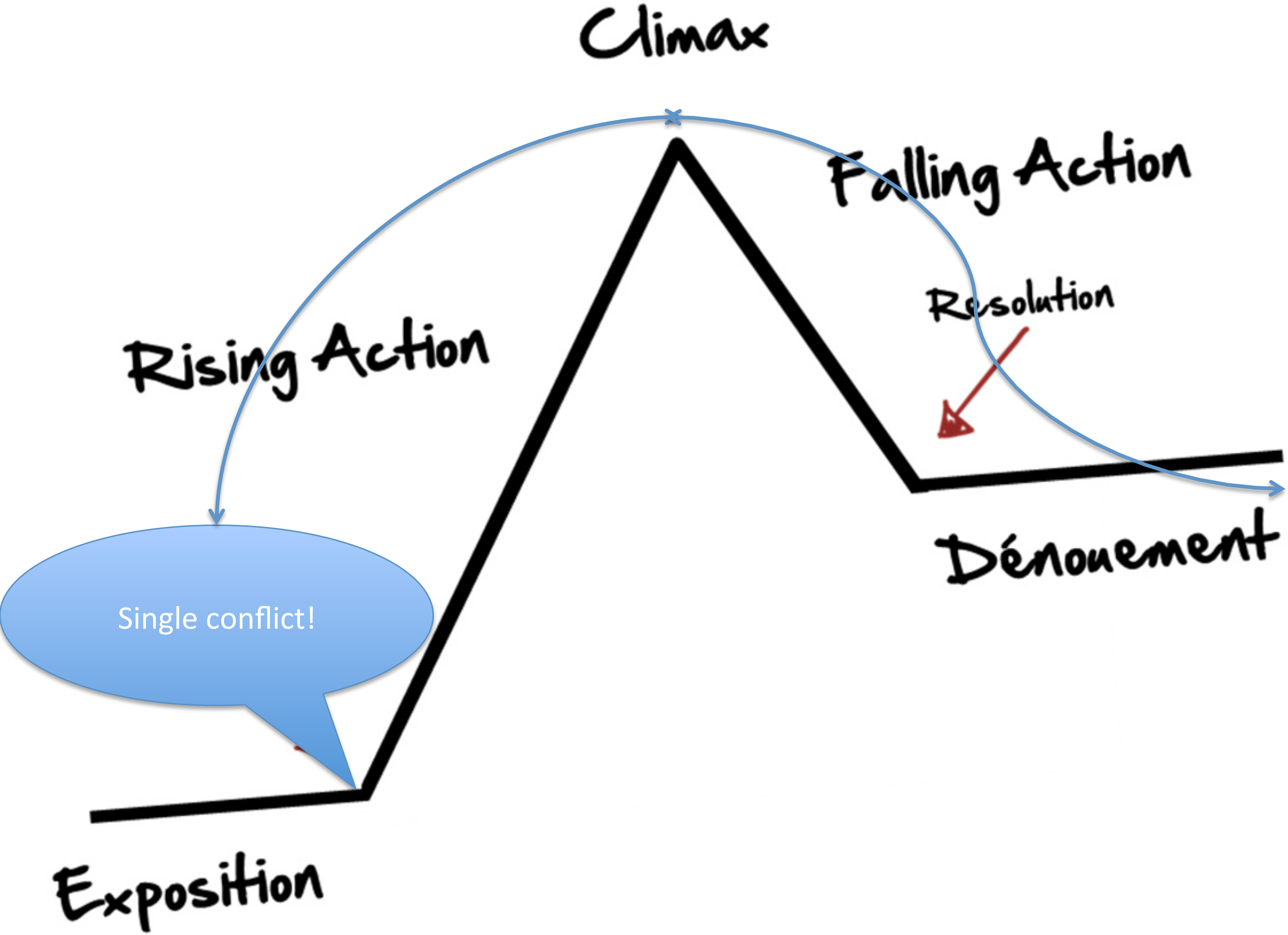

Climax
Falling Action
Rising Action
Resolution
Dénouement
Inciting Incident
Exposition


Climax
Falling Action
Resolution
Rising Action
Dénouement
Inciting Incident
Exposition

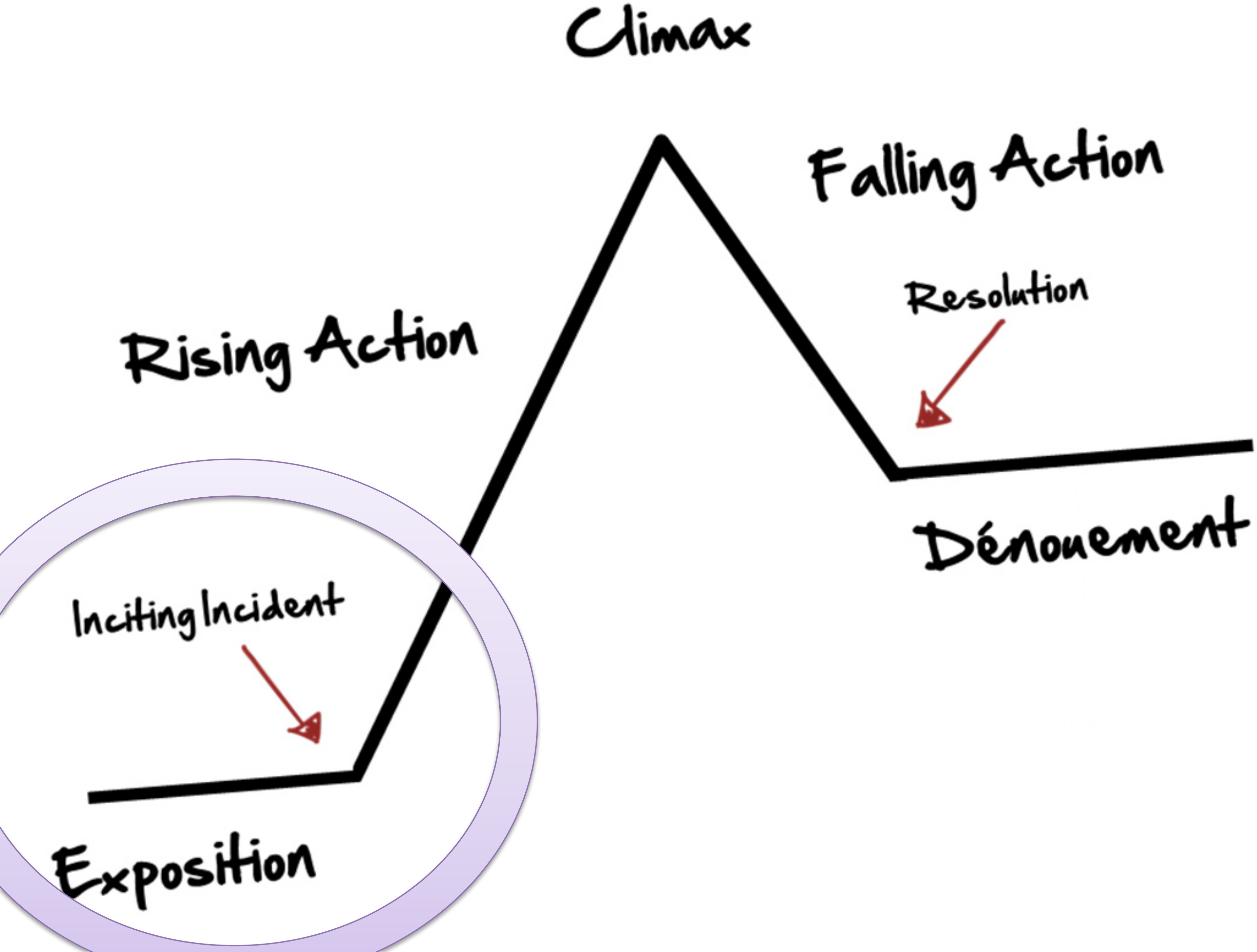


Climax
Falling Action
Resolution
Rising Action
Dénouement
Inciting Incident
Exposition


Climax
Falling Action
Resolution
Rising Action
Dénouement
Inciting Incident
Exposition



Climax
Falling Action
Resolution
Rising Action
Dénonement
Single conflict!
Exposition


Climax
Falling Action
Resolution
Rising Action
Dénouement
Inciting Incident
Exposition

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

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:

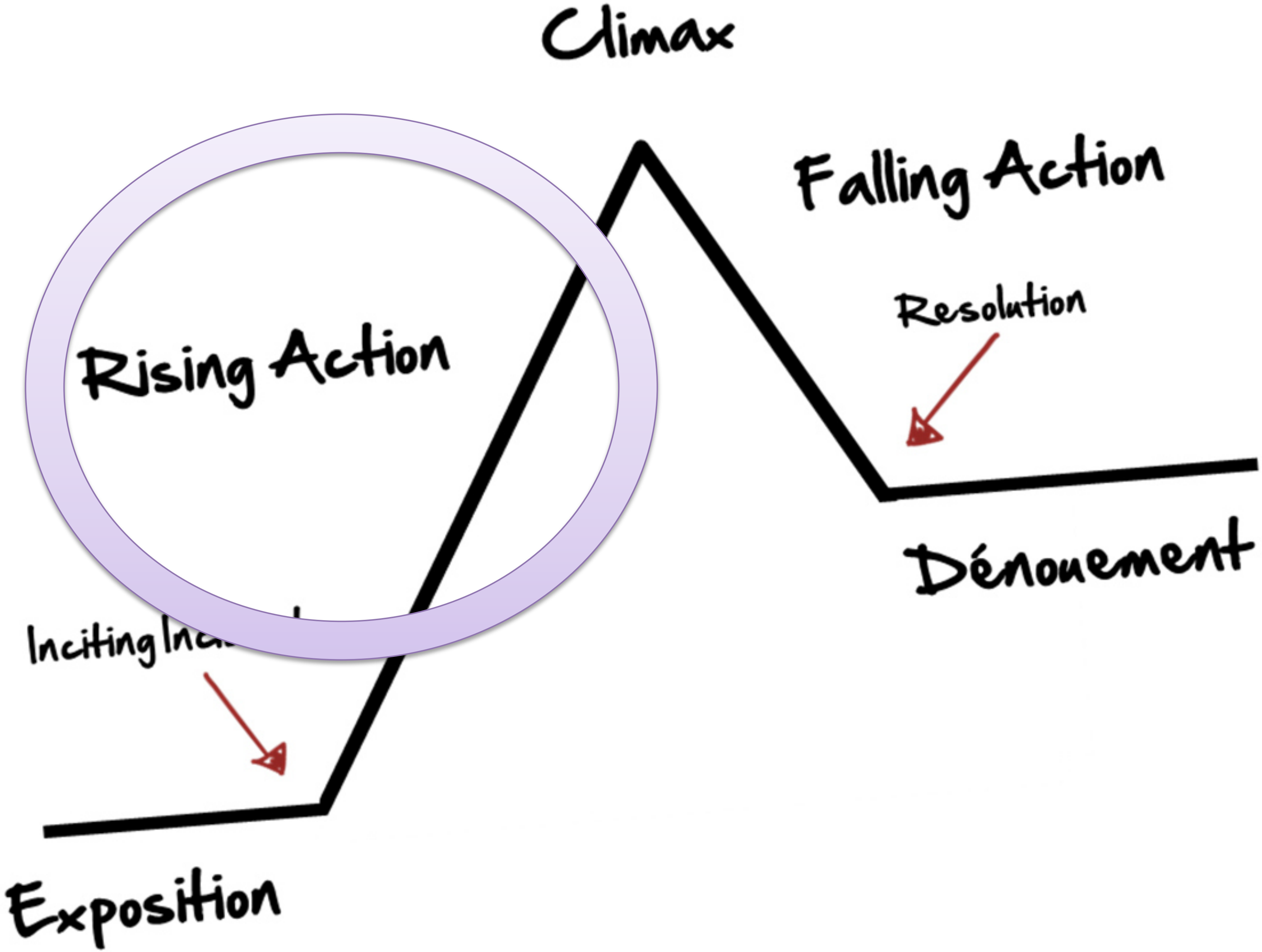
1) *Encodes* a large database of human-written code fragments as satisfiability modulo theories (SMT) constraints on their input-output behavior.
2) *Localizes* a defect to likely buggy program fragments.
3) *Constructs*, for each fragment, a lightweight input-output profile that characterizes desired 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].

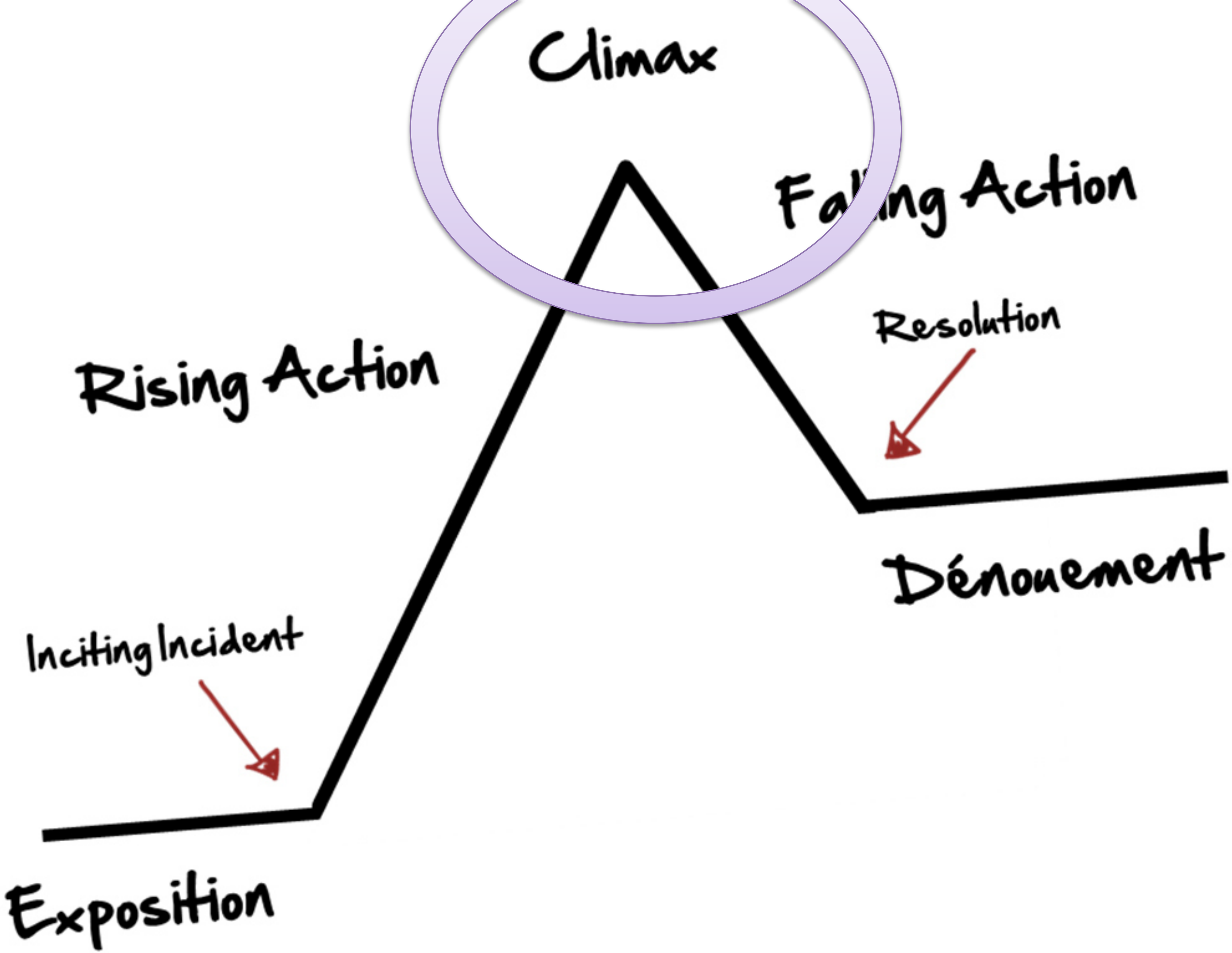
By definition, test suites only encode a partial specification of correct behavior. A patch that is correct according to a given test suite may therefore not be fully correct when evaluated with respect to a hypothetical full correctness specification. This is analogous to the well-known machine learning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

- 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.*


Climax
Falling Action
Resolution
Rising Action
Dénonement
Inciting Inci
Exposition

Inciting incident + rising action

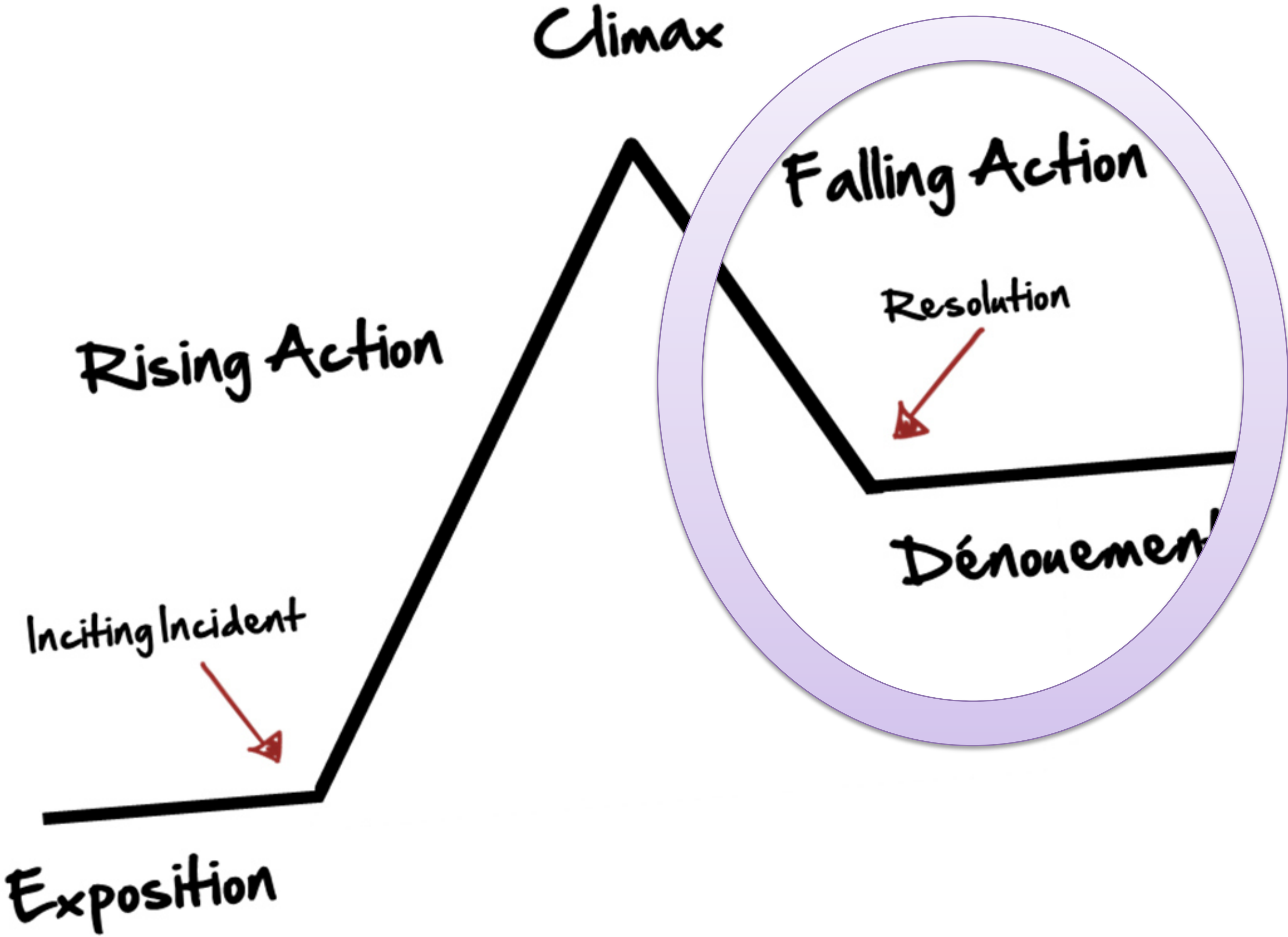
- Middle of story: Key technical insight.
- High-level outline of the approach.
- If you're not sure if a detail is high-level enough, it's probably not.
- (More on how to approach "the middle" in 10—11 slides)



Climax
Falling Action
Resolution
Rising Action
Dénouement
Inciting Incident
Exposition

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.

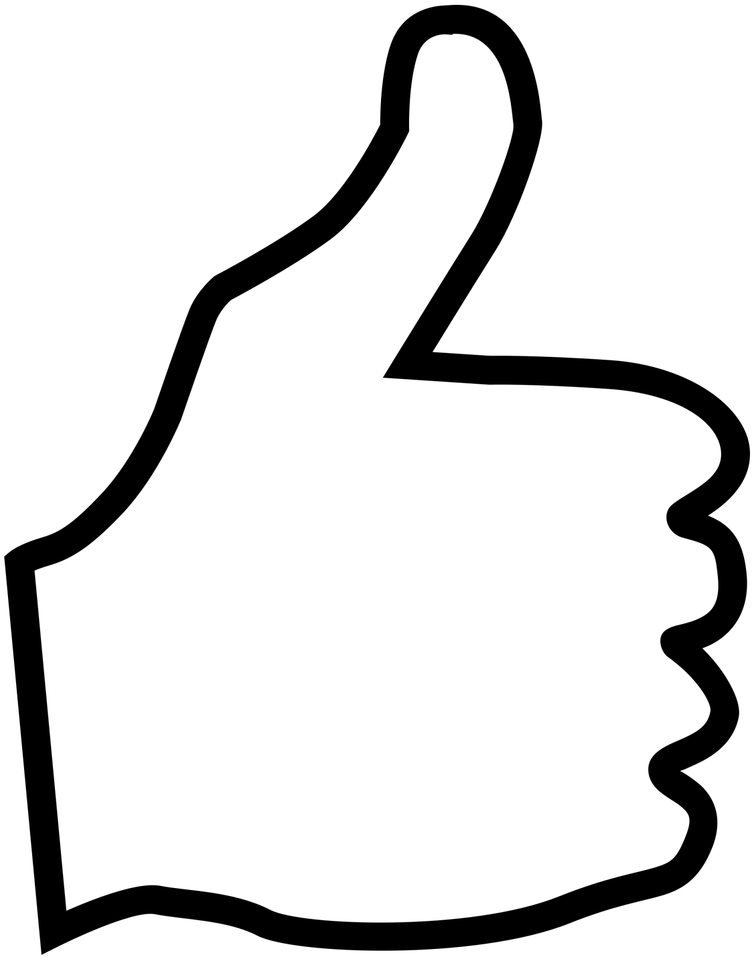

Climax
Falling Action
Resolution
Rising Action
Dénonement
Inciting Incident
Exposition

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.**

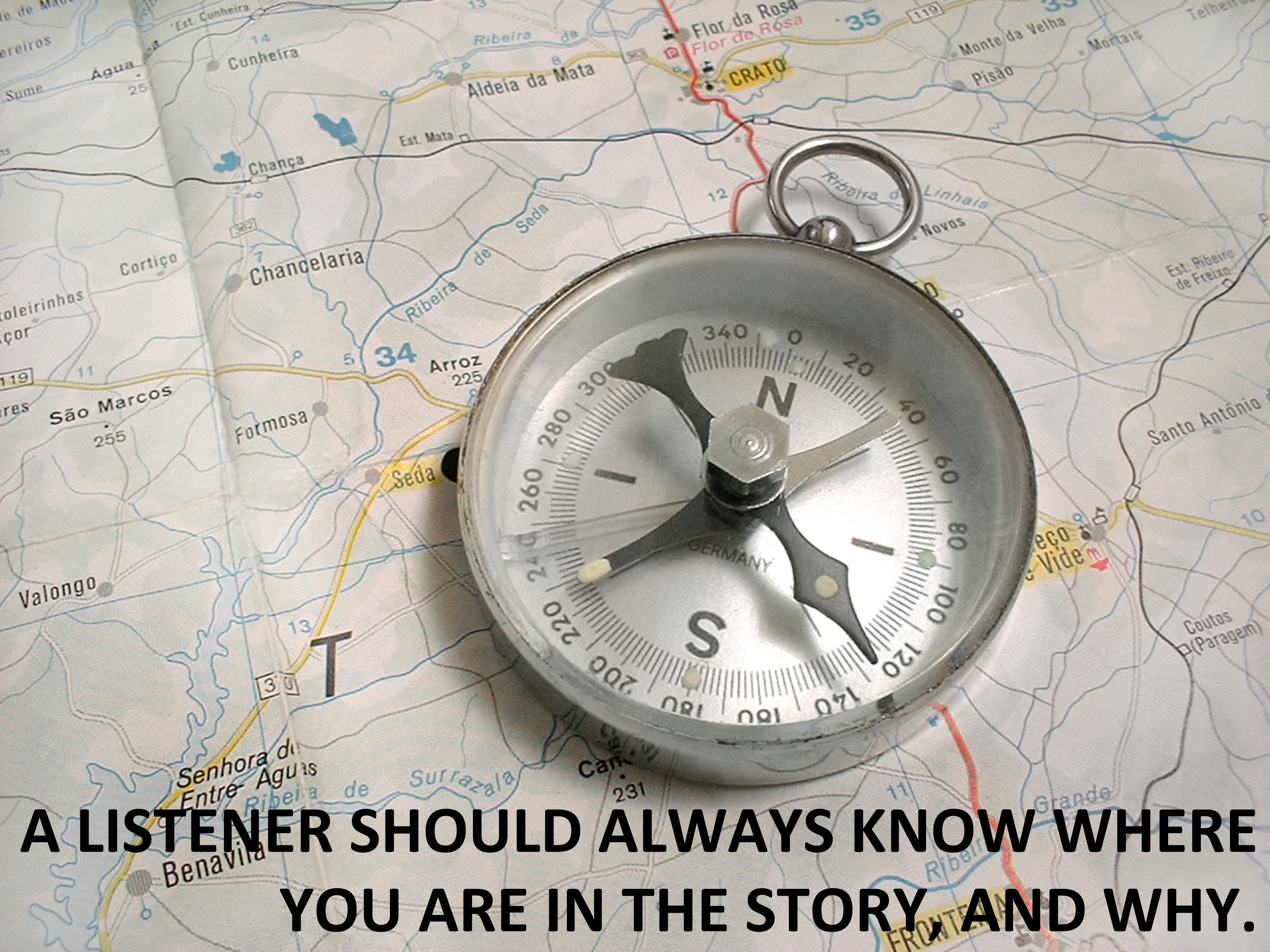
? ? ?

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!

Implic

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-d

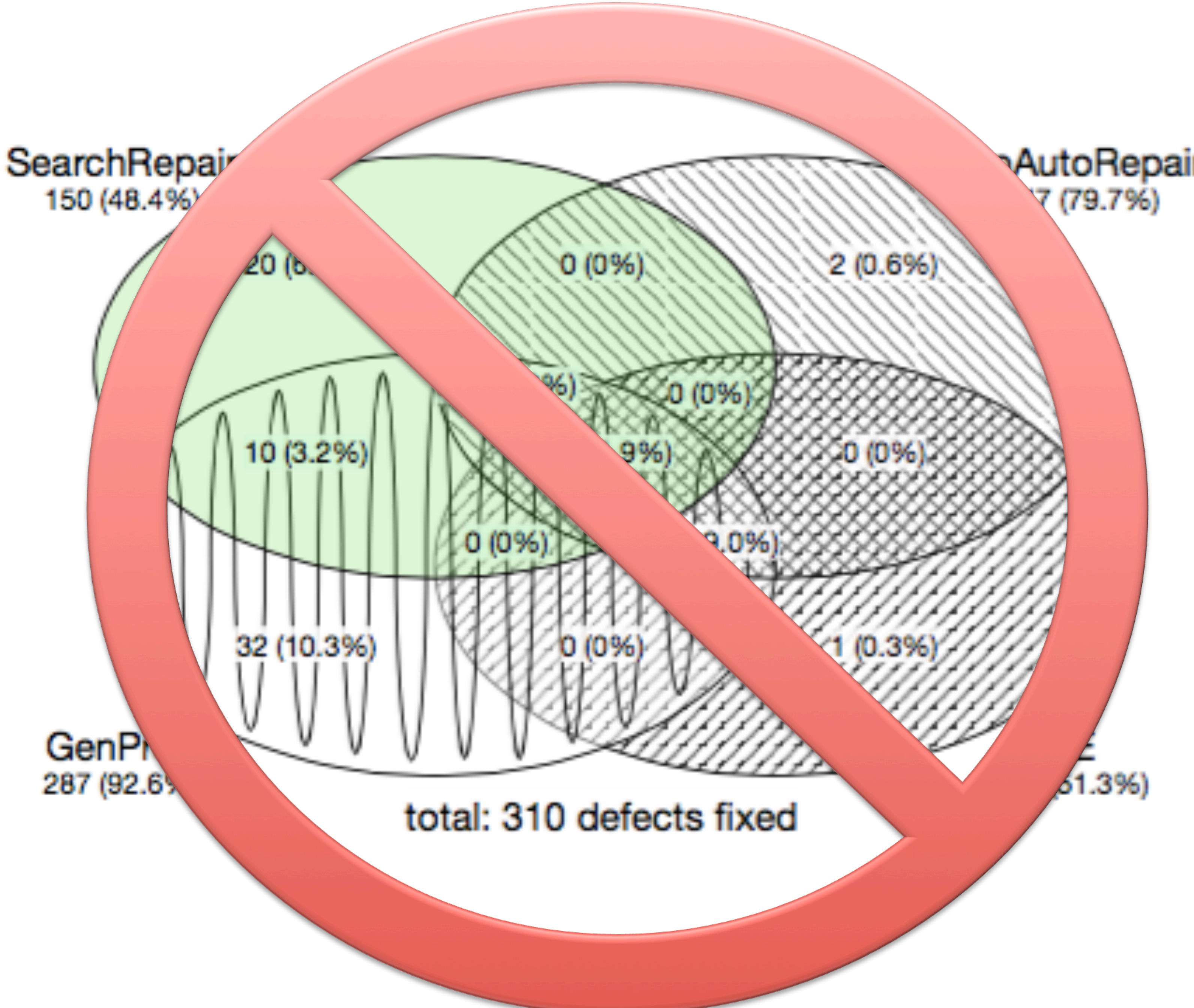
I violated this rule 11 slides ago.

DO NOT VISUALLY OVERWHELM YOUR LISTENERS.

DO NOT VISUALLY OVERWHELM YOUR LISTENERS.

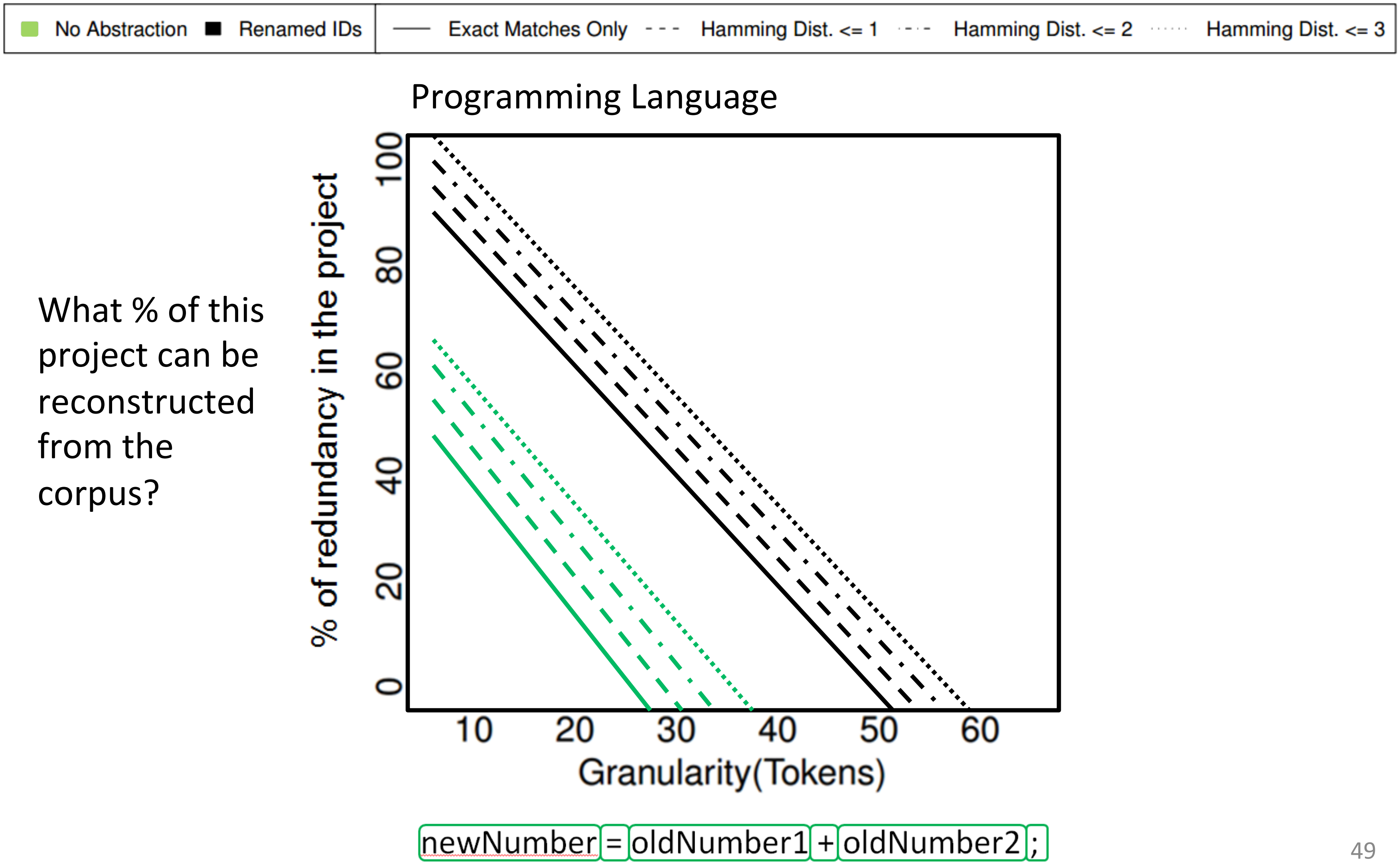
DO NOT VISUALLY OVERWHELM YOUR LISTENERS.

BEWARE THE RESULTS PRESENTATION.


SearchRepai
150 (48.4%)
AutoRepair
7 (79.7%)
20 (6
0 (0%)
2 (0.6%)
%)
0 (0%)
10 (3.2%)
9%)
0 (0%)
0 (0%)
.0%)
32 (10.3%)
0 (0%)
1 (0.3%)
GenP
287 (92.6
51.3%)
total: 310 defects fixed

(Slide borrowed from my student Mauricio.)

How to read the graph?



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

program	[illegible]pair	AE	GenProg	TrpAuto[illegible]	[illegible]	total
checksu[illegible]	0	0	8	0	[illegible]	29
digits	0	[illegible]	30	19	[illegible]	[illegible]91
grade	5	[illegible]	2	2	[illegible]	[illegible]26
media[illegible]	68	58	[illegible]08	93	[illegible]	[illegible]68
smalle[illegible]	73	71	[illegible]	119	[illegible]	[illegible]55
syllabl[illegible]	4	11	[illegible]	14	[illegible]	[illegible]09
total rep[illegible]d	150	159	287	247	[illegible]	[illegible]778

Program	Description	LOC	Bug Type	Time (s)
gcd	example	[illegible]	loop	153
nullhttpd	webserve	5575	hea erflow (code)	578
zune	exam	28	infinite lo	42
uniq	te g	1146	segmentation	34
look-u	ary	1169	segmentation fau	45
look-s	ionary loo	1363	infinite loop	55
units	etric conversion	1504	segmentation fault	109
deroff	ocument processing	2236	segmentation fault	131
indent	ode processing	9906	infinite loop	546
flex	exical analyzer generator	74	segmentation fault	230
openldap	irectory protocol	29	on-overflow denial of ce	665
ccrypt	ncryption utility	751	entation fault	330
lighttpd	bserver	51895	h ffer overflow)	394
atris	ical game	21553	local uffer e	80
php	sc language	764489	integer o	56
wu-ftpd	FTP	67029	format stri rability	2256
leukocyte	computa logy	6718	segm ault	360
tiff	image process	[illegible]	ation fault	108
imagemagick	image processing	[illegible]	wrong output	2160

Program	Description	LOC	Bug Type	Time (s)

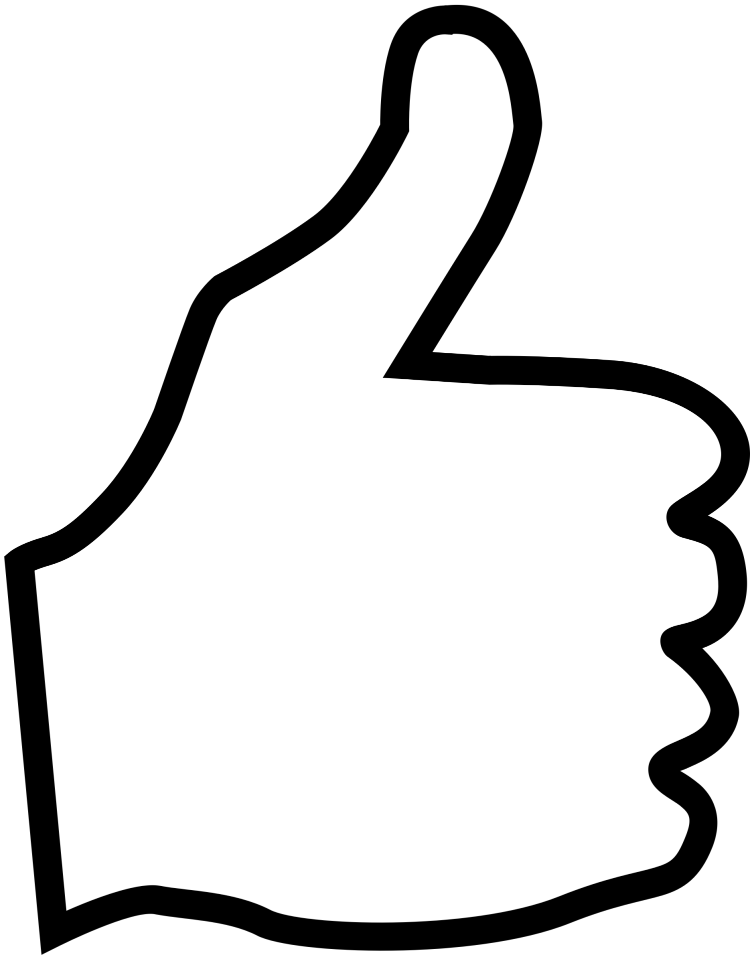
Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153

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

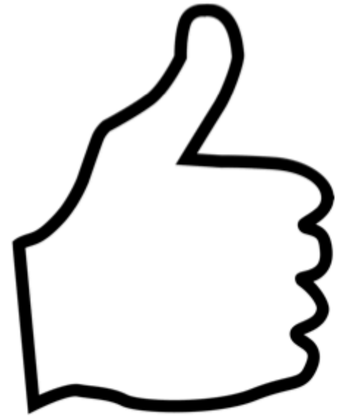
Program	Description	LOC	Bug Type	Time (s)
gcd	example	22	infinite loop	153
nullhttpd	webserver	5575	heap buffer overflow (code)	578
zune	example	28	infinite loop	42
uniq	text processing	1146	segmentation fault	34
look-u	dictionary lookup	1169	segmentation fault	45
look-s	dictionary lookup	1363	infinite 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
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	84067	segmentation fault	108
imagemagick	image processing	450516	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.

(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!"