MENTOR: Automated Feedback for Introductory Programming Exercises

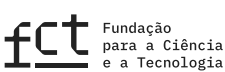
Pedro Orvalho [1]

[1]Department of Computer Science, University of Oxford, Oxford, UK

Previously at:
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
CIIRC, Czech Technical University in Prague, Czechia

DI Seminars, FCT, Universidade Nova de Lisboa

Lisbon, 22 April 2025



Motivation

- The increasing demand for programming education has given rise to all kinds of online evaluations focused on introductory programming assignments (IPAs):

Motivation

- The increasing demand for programming education has given rise to all kinds of online evaluations focused on introductory programming assignments (IPAs):
  - MIT's MOOC, Introduction to CS, **reached 1.2 M enrollments** in 2018;

Motivation

- The increasing demand for programming education has given rise to all kinds of online evaluations focused on introductory programming assignments (IPAs):
  - MIT's MOOC, Introduction to CS, **reached 1.2 M enrollments** in 2018;
  - In 2020, Stanford's CS MOOC had **more than 10K students**.

Motivation

- In these courses it is a challenge to **provide personalized feedback to students**.

Motivation

- In these courses it is a challenge to **provide personalized feedback to students**.
- Providing feedback in IPAs **requires substantial time and effort by faculty**.

Automated Program Repair

Example (A program that finds the maximum number among three numbers.)

```
1  int max_three(int n1, int n2, int n3){
2      int max = 0;
3      if(n1 > max){
4          max = n1;
5      }
6      if (n2 > max){
7          max = n2;
8      }
9      if (n3 > max){
10         max = n3;
11     }
12     return max;
13 }
```

Table 1: Test-suite with three tests (t1, t2, and t3).

	Input			Output
	num1	**num2**	**num3**	
t1	1	2	3	3
t2	-1	-2	-3	-1
t3	1	2	1	2

Automated Program Repair

Example (A program that finds the maximum number among three numbers.)

```
1  int max_three(int n1, int n2, int n3){
2      int max = 0;
3      if(n1 > max){
4          max = n1;
5      }
6      if (n2 > max){
7          max = n2;
8      }
9      if (n3 > max){
10         max = n3;
11     }
12     return max;
13 }
```

Table 1: Test-suite with three tests (t1, t2, and t3).

	Input			Output
	num1	num2	num3	
t1	1	2	3	3
t2	-1	-2	-3	-1
t3	1	2	1	2

Automated Program Repair (APR)

Given a buggy program P_o and a set of input-output examples T (test suite).

Automated Program Repair (APR)

Given a buggy program P_o and a set of input-output examples T (test suite).

The goal of *Automated Program Repair* is to find a program P_f by **semantically change a subset S_1 of P_o's statements** ($S_1 \subseteq P_o$) for another set of statements S_2, s.t.,

$$P_f = ((P_o \setminus S_1) \cup S_2)$$

and

$$\forall (t^i_{in}, t^i_{out}) \in T \ : \ P_f(t^i_{in}) = t^i_{out}$$

Problem Description

Limitations of Past Approaches

Semantic APR techniques (e.g. CLARA, VERIFIX), require:

Problem Description

Limitations of Past Approaches

Semantic APR techniques (e.g. CLARA, VERIFIX), require:

1. **Perfect match between the control flow graphs** of two programs;

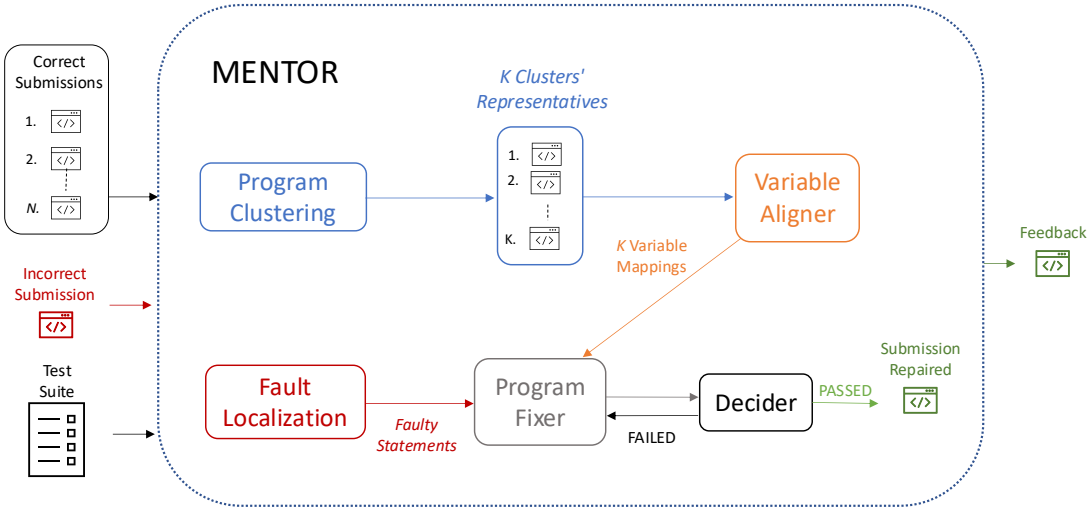
Problem Description

Limitations of Past Approaches

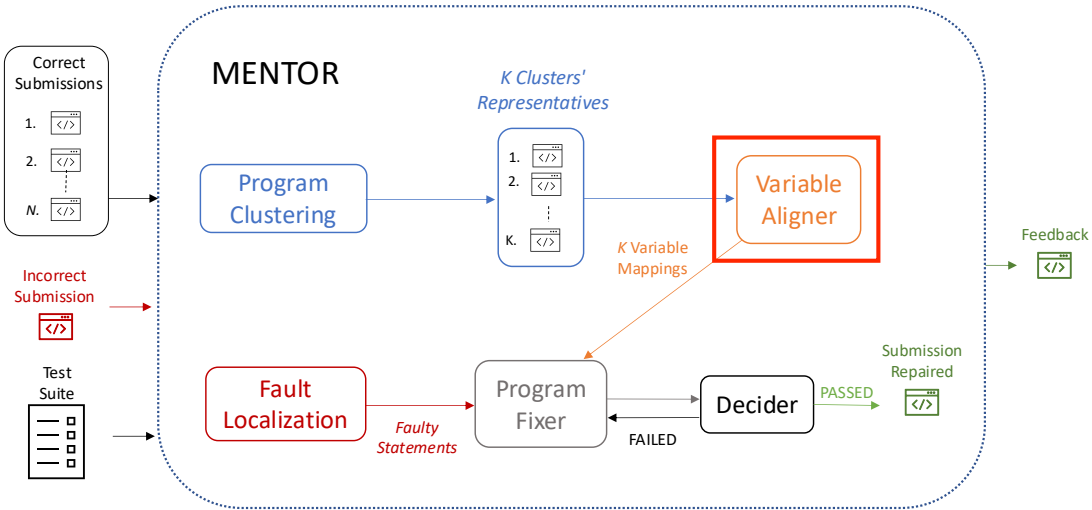
Semantic APR techniques (e.g. CLARA, VERIFIX), require:

1. **Perfect match between the control flow graphs** of two programs;
2. **Bijective relation between the sets of variables** of both programs.

MENTOR


Correct Submissions
1.
2.
N.
Incorrect Submission
Test Suite
MENTOR
K Clusters' Representatives
Program Clustering
1.
2.
K.
Variable Aligner
K Variable Mappings
Feedback
Fault Localization
Faulty Statements
Program Fixer
Decider
PASSED
FAILED
Submission Repaired

MENTOR


Correct Submissions
1.
2.
N.
Incorrect Submission
Test Suite
MENTOR
Program Clustering
K Clusters' Representatives
1.
2.
K.
Variable Aligner
K Variable Mappings
Feedback
Fault Localization
Faulty Statements
Program Fixer
Decider
PASSED
FAILED
Submission Repaired

Variable Mapping

- **ECAI 23** - Graph Neural Networks For Mapping Variables Between Programs;
- **ESEC/FSE 2022** - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation;
- **AITP 22** - Project Proposal: Learning Variable Mappings to Repair Programs.

Variable Mapping - Motivation

- Comparing two programs is **highly challenging**;

Variable Mapping - Motivation

- Comparing two programs is **highly challenging**;
- **A relation between** two programs' sets of **variables is required**;

Variable Mapping - Motivation

- Comparing two programs is **highly challenging**;
- **A relation between** two programs' sets of **variables is required**;
- Mapping variables between two programs is **useful for a variety of program related tasks**, such as, program equivalence, program repair, etc.

Variable Mapping - Motivation

1: Function that finds and returns the maximum number among `n1`, `n2` and `n3`.

```
1  int max(int n1, int n2, int n3)
2  {
3    int m = n1 > n2 ? n1 : n2;
4    return n3 > m ? n3 : m;
5  }
```

2: Function that finds and returns the maximum number among `x`, `y` and `z`.

```
1  int max(int x, int y, int z){
2    int m = 0;
3    m = x > m ? x : m;
4    m = y > m ? y : m;
5    return z > m ? z : m;
6  }
```

Variable Mapping - Motivation

3: Function that finds and returns the maximum number among `n1`, `n2` and `n3`.

```
1  int max(int n1, int n2, int n3)
2  {
3    int m = n1 > n2 ? n1 : n2;
4    return n3 > m ? n3 : m;
5  }
```

4: Function that finds and returns the maximum number among `x`, `y` and `z`.

```
1  int max(int x, int y, int z){
2    int m = 0;
3    m = x > m ? x : m;
4    m = y > m ? y : m;
5    return z > m ? z : m;
6  }
```

Variable Mapping: $\{\texttt{m} : \texttt{m}; \texttt{n1} : \texttt{x}; \texttt{n2} : \texttt{y}; \texttt{n3} : \texttt{z}\}$.

Motivation

5: Function that finds and returns the maximum number among `n1`, `n2` and `n3`.

```
1  int max(int n1, int n2, int n3)
2  {
3    int m = n1 > n2 ? n1 : n2;
4    return n3 > m ? n3 : m;
5  }
```

6: Function that finds and returns the maximum number among `x`, `y` and `z`.

```
1  int max(int x, int y, int z){
2    int m = 0;
3    m = x > m ? x : m;
4    m = y > m ? y : m;
5    return z > m ? z : m;
6  }
```

Variable Mapping: $\{\texttt{m} : \texttt{m};\ \texttt{n1} : \texttt{x};\ \texttt{n2} : \texttt{y};\ \texttt{n3} : \texttt{z}\}$.

Contribution

- A graph program representation that takes **advantage of the structural information of the *abstract syntax trees (ASTs)*** of programs;

Contribution

- A graph program representation that takes **advantage of the structural information of the *abstract syntax trees (ASTs)*** of programs;
- Our program representation is **agnostic to the names of the variables**;

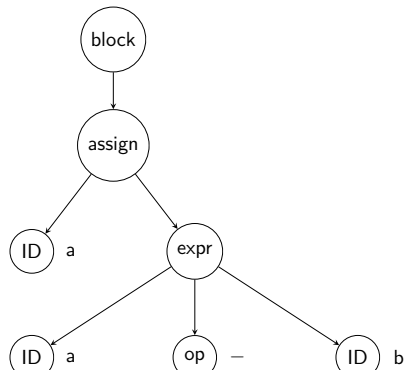
Contribution

- A graph program representation that takes **advantage of the structural information of the *abstract syntax trees (ASTs)*** of programs;
- Our program representation is **agnostic to the names of the variables**;
- **Map the variables between a correct program and a faulty one** using *Graph Neural Networks (GNNs)*.

Program Representation

7: An expression that uses int variables a and b, previously declared in the program.

```
1  {
2    // a and b are ints
3    a = a - b;
4  }
```



(a) Part of the AST representation.

Program Representation

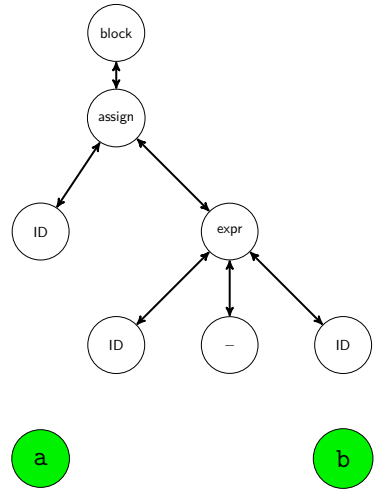
8: An expression that uses int variables `a` and `b`, previously declared in the program.

```
1  {
2    // a and b are ints
3    a = a - b;
4  }
```

Types of edges:

AST ⟷

Variable Node



(b) Our program representation.

Program Representation

9: An expression that uses int variables `a` and `b`, previously declared in the program.

```
1  {
2    // a and b are ints
3    a = a - b;
4  }
```

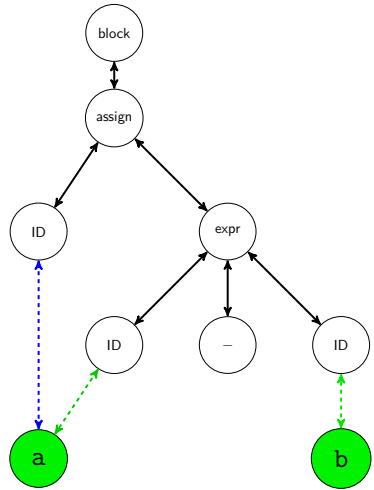
Types of edges:

AST ⟷

Read ⟷

Write ⟷

Variable Node



(c) Our program representation.

Program Representation

10: An expression that uses int variables a and b, previously declared in the program.

```
1  {
2    // a and b are ints
3    a = a - b;
4  }
```

Types of edges:

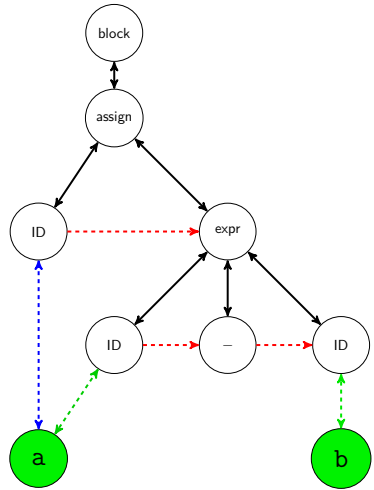
AST

Read

Write

Sibling

Variable Node



(d) Our program representation.

Program Representation

11: An expression that uses int variables `a` and `b`, previously declared in the program.

```
1  {
2    // a and b are ints
3    a = a - b;
4  }
```

Types of edges:

AST

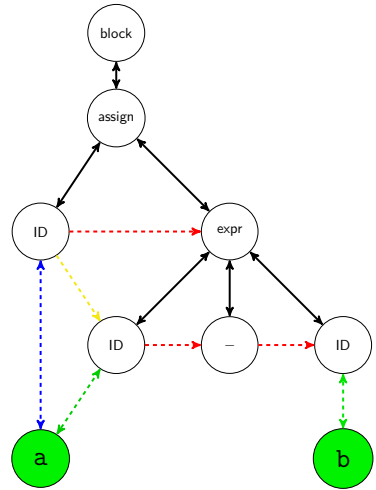
Read

Write

Sibling

Chronological

Variable Node



(e) Our program representation.

Graph Neural Networks (GNNs)

- We perform ***message passing*** **between the nodes of our representations**;

Graph Neural Networks (GNNs)

- We perform ***message passing*** **between the nodes of our representations**;
- We obtain **vectors corresponding to each variable node** in each program;

Graph Neural Networks (GNNs)

- We perform ***message passing*** **between the nodes of our representations**;
- We obtain **vectors corresponding to each variable node** in each program;
- We compute **scalar products between each possible combination of variable nodes** in the two programs, followed by a softmax function.

Data Augmentation

- We use C-Pack-IPAs [Orvalho et al., 2024], a set of 10 **introductory programming assignments**, comprising **486 faulty programs**;

Data Augmentation

- We use C-Pack-IPAs [Orvalho et al., 2024], a set of 10 **introductory programming assignments**, comprising **486 faulty programs**;
- Since we need to know the real variable mappings between programs to evaluate our representation, we used **MultIPAs [Orvalho et al., 2022] to generate a dataset of pairs of correct/incorrect programs**:

Data Augmentation

- We use C-Pack-IPAs [Orvalho et al., 2024], a set of 10 **introductory programming assignments**, comprising **486 faulty programs**;
- Since we need to know the real variable mappings between programs to evaluate our representation, we used **MultIPAs [Orvalho et al., 2022] to generate a dataset of pairs of correct/incorrect programs**:
  - MultIPAs can perform six syntactic program mutations;

Data Augmentation

- We use C-Pack-IPAs [Orvalho et al., 2024], a set of 10 **introductory programming assignments**, comprising **486 faulty programs**;
- Since we need to know the real variable mappings between programs to evaluate our representation, we used **MultIPAs [Orvalho et al., 2022] to generate a dataset of pairs of correct/incorrect programs**:
  - MultIPAs can perform six syntactic program mutations;
  - MultIPAs can introduce three kinds of bugs: wrong comparison operator (WCO), variable misuse (VM), and missing expression (ME).

Variable Mapping - Results

	Buggy Programs (Total = 186366)
Correct Mappings	179470 (96.49%)

Table 2: Validation Performance after 20 training epochs.

Variable Mapping - Results

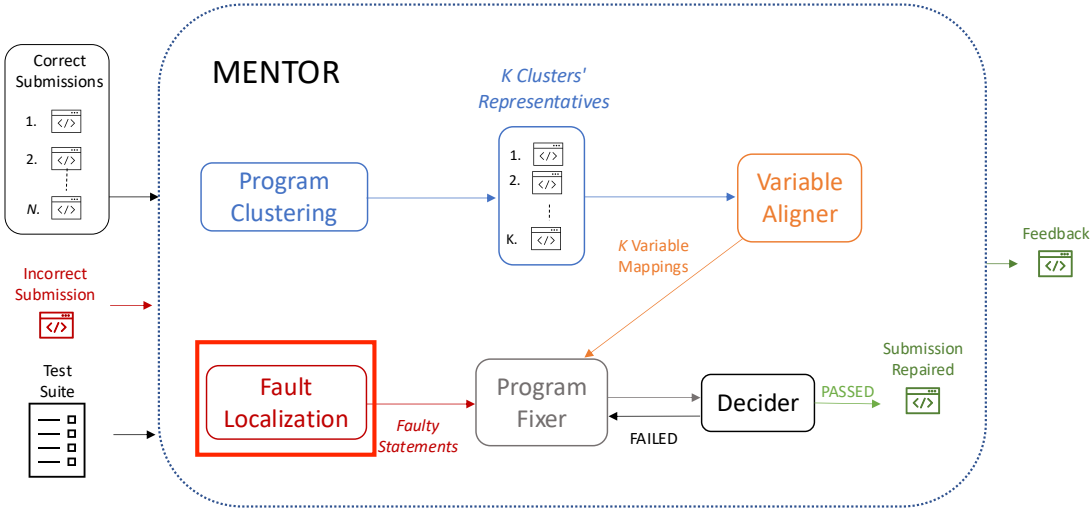
	Buggy Programs (Total = 186366)
Correct Mappings	179470 (96.49%)

Table 2: Validation Performance after 20 training epochs.

Evaluation Metric	**Buggy Programs**
# Correct Mappings	82.77%
Avg Overlap Coefficient	95.05%

Table 3: Test Performance.

MENTOR


Correct Submissions
1.
2.
N.
Incorrect Submission
Test Suite
MENTOR
Program Clustering
K Clusters' Representatives
1.
2.
K.
Variable Aligner
K Variable Mappings
Feedback
Fault Localization
Faulty Statements
Program Fixer
Decider
PASSED
FAILED
Submission Repaired

Fault Localization

- **FM24** - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases.

Fault Localization - Motivation

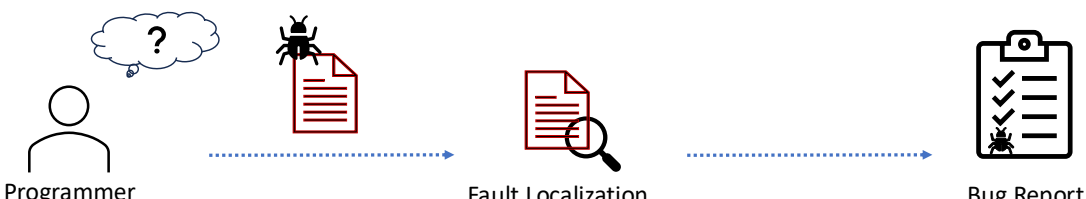
- Debugging is one of the most time-consuming and expensive tasks in software development.

Fault Localization - Motivation

- Debugging is one of the most time-consuming and expensive tasks in software development.
- In 2024, the estimated global cost of Crowdstrike's error that hit Microsoft systems, is 5.4 Billion US$ [The Guardian UK, 2024].

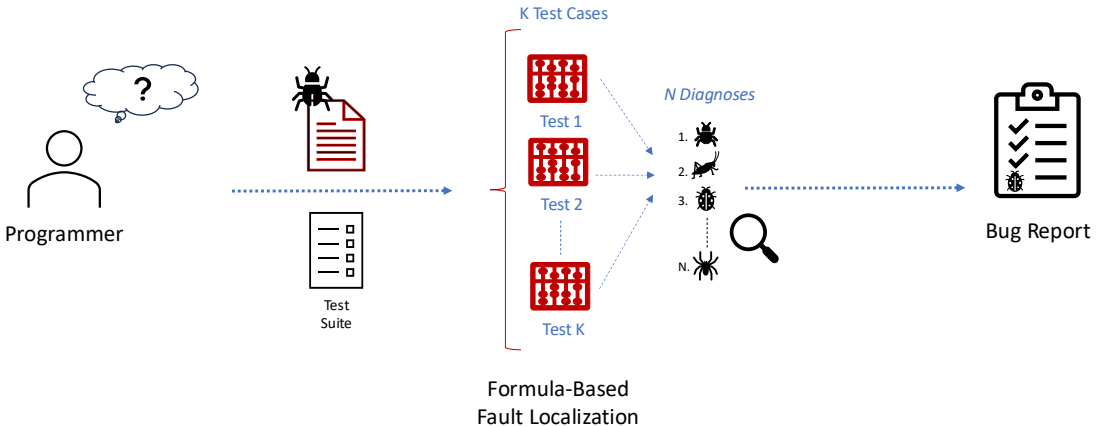
Fault Localization

- Given a buggy program, *fault localization (FL)* involves identifying locations in the program that could cause a faulty behaviour (bug).



Formula-Based Fault Localization (FBFL)

- FBFL methods encode the localization problem into **several optimization problems** to identify a minimal set of bugs (diagnoses).



Formula-Based Fault Localization (FBFL)

Limitations of Past Approaches

FBFL tools especially for programs with multiple faults:

Formula-Based Fault Localization (FBFL)

Limitations of Past Approaches

FBFL tools especially for programs with multiple faults:

- **do not ensure a minimal diagnosis** across all failing tests (e.g., BugAssist);

Formula-Based Fault Localization (FBFL)

Limitations of Past Approaches

FBFL tools especially for programs with multiple faults:

- **do not ensure a minimal diagnosis** across all failing tests (e.g., BugAssist);
- may produce an overwhelming number of **redundant diagnoses** (e.g., SNIPER).

Contribution

- We formulate the FL problem as a **single optimization problem**;

Contribution

- We formulate the FL problem as a **single optimization problem**;
- We leverage MaxSAT and the theory of *Model-Based Diagnosis (MBD)* [Reiter et al., 1987], **integrating all failing test cases simultaneously**;

Contribution

- We formulate the FL problem as a **single optimization problem**;
- We leverage MaxSAT and the theory of *Model-Based Diagnosis (MBD)* [Reiter et al., 1987], **integrating all failing test cases simultaneously**;
- We implement this MBD approach in a publicly available tool called CFaults.

Model-Based Diagnosis

- A system description $\mathcal{P}$ **is composed of a set of components** $\mathcal{C} = \{c_1, \ldots, c_n\}$.

Model-Based Diagnosis

- A system description $\mathcal{P}$ **is composed of a set of components** $\mathcal{C} = \{c_1, \ldots, c_n\}$.
- Each component in $\mathcal{C}$ can be declared **healthy** or **unhealthy**.

Model-Based Diagnosis

- A system description $\mathcal{P}$ **is composed of a set of components** $\mathcal{C} = \{c_1, \ldots, c_n\}$.
- Each component in $\mathcal{C}$ can be declared **healthy** or **unhealthy**.
- For each component $c \in \mathcal{C}$, $h(c) = 0$ **if** c **is unhealthy, otherwise,** $h(c) = 1$.

Model-Based Diagnosis

- A system description $\mathcal{P}$ **is composed of a set of components** $\mathcal{C} = \{c_1, \ldots, c_n\}$.
- Each component in $\mathcal{C}$ can be declared **healthy** or **unhealthy**.
- For each component $c \in \mathcal{C}$, $h(c) = 0$ **if** c **is unhealthy, otherwise,** $h(c) = 1$.
- $\mathcal{P}$ is described by a CNF formula, where $\mathcal{F}_c$ denotes the encoding of component c:

$$\mathcal{P} \triangleq \bigwedge\nolimits_{c \in \mathcal{C}} (h(c) \implies \mathcal{F}_c)$$

Model-Based Diagnosis

- **Observations represent deviations** from the expected system behaviour.

Model-Based Diagnosis

- **Observations represent deviations** from the expected system behaviour.
- An observation, denoted as o, can be encoded in CNF as a set of unit clauses.

Model-Based Diagnosis

- **Observations represent deviations** from the expected system behaviour.
- An observation, denoted as o, can be encoded in CNF as a set of unit clauses.
- In our work, **the failing test cases represent the set of observations**.

Model-Based Diagnosis

- **Observations represent deviations** from the expected system behaviour.
- An observation, denoted as o, can be encoded in CNF as a set of unit clauses.
- In our work, **the failing test cases represent the set of observations**.
- A system $\mathcal{P}$ is considered **faulty if there exists an inconsistency with a given observation o when all components are declared healthy**:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C}} h(c) \vDash \bot$$

Model-Based Diagnosis

- The problem of model-based diagnosis (MBD) aims to **identify a set of components which, if declared unhealthy, restore consistency**;

Model-Based Diagnosis

- The problem of model-based diagnosis (MBD) aims to **identify a set of components which, if declared unhealthy, restore consistency**;
- For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot$$

Model-Based Diagnosis

- The problem of model-based diagnosis (MBD) aims to **identify a set of components which, if declared unhealthy, restore consistency**;
- For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot$$

- A diagnosis Δ is:

Model-Based Diagnosis

- The problem of model-based diagnosis (MBD) aims to **identify a set of components which, if declared unhealthy, restore consistency**;
- For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot$$

- A diagnosis Δ is:
  - **minimal** iff no subset of Δ, $\Delta' \subsetneq \Delta$, is a diagnosis;

Model-Based Diagnosis

- The problem of model-based diagnosis (MBD) aims to **identify a set of components which, if declared unhealthy, restore consistency**;
- For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot$$

- A diagnosis Δ is:
  - **minimal** iff no subset of Δ, $\Delta' \subsetneq \Delta$, is a diagnosis;
  - Δ is of **minimal cardinality** if there is no other diagnosis $\Delta'' \subseteq \mathcal{C}$ with $|\Delta''| < |\Delta|$;

Model-Based Diagnosis

- The problem of model-based diagnosis (MBD) aims to **identify a set of components which, if declared unhealthy, restore consistency**;
- For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot$$

- A diagnosis Δ is:
  - **minimal** iff no subset of Δ, $\Delta' \subsetneq \Delta$, is a diagnosis;
  - Δ is of **minimal cardinality** if there is no other diagnosis $\Delta'' \subseteq \mathcal{C}$ with $|\Delta''| < |\Delta|$;
  - is **redundant** if it is not subset-minimal [Ignatiev et al., 2019].

Model-Based Diagnosis

To encode the MBD problem with one observation with partial MaxSAT:

- The set of **clauses that encode $\mathcal{P}$ represents the set of hard clauses**;

Model-Based Diagnosis

To encode the MBD problem with one observation with partial MaxSAT:

- The set of **clauses that encode $\mathcal{P}$ represents the set of hard clauses**;
- The soft clauses consists of unit clauses that aim to **maximize the set of healthy components**, i.e.,:

$$\bigwedge_{c \in \mathcal{C}} h(c);$$

Model-Based Diagnosis

To encode the MBD problem with one observation with partial MaxSAT:

- The set of **clauses that encode $\mathcal{P}$ represents the set of hard clauses**;
- The soft clauses consists of unit clauses that aim to **maximize the set of healthy components**, i.e.,:

$$\bigwedge_{c \in \mathcal{C}} h(c);$$

- This encoding enables enumerating subset **minimal diagnoses, considering a single observation**;

Model-Based Diagnosis with Multiple Test Cases

We **integrate all failing test cases** in a single MaxSAT formula.

Model-Based Diagnosis with Multiple Test Cases

We **integrate all failing test cases** in a single MaxSAT formula.

- We **generate only minimal diagnoses** capable of identifying all faulty components within the system, in our case, a C program;

Model-Based Diagnosis with Multiple Test Cases

We **integrate all failing test cases** in a single MaxSAT formula.

- We **generate only minimal diagnoses** capable of identifying all faulty components within the system, in our case, a C program;
- Given m observations, $\mathcal{O} = \{o_1, \ldots, o_m\}$, a distinct replica of the system, denoted as $\mathcal{P}_i$, is required for each observation o_i;

Model-Based Diagnosis with Multiple Test Cases

We **integrate all failing test cases** in a single MaxSAT formula.

- We **generate only minimal diagnoses** capable of identifying all faulty components within the system, in our case, a C program;
- Given m observations, $\mathcal{O} = \{o_1, \ldots, o_m\}$, a distinct replica of the system, denoted as $\mathcal{P}_i$, is required for each observation o_i;
- The hard clauses, ϕ_h, in our MaxSAT formulation correspond to:

$$\phi_h = \bigwedge_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i);$$

Model-Based Diagnosis with Multiple Test Cases

We **integrate all failing test cases** in a single MaxSAT formula.

- We **generate only minimal diagnoses** capable of identifying all faulty components within the system, in our case, a C program;
- Given m observations, $\mathcal{O} = \{o_1, \ldots, o_m\}$, a distinct replica of the system, denoted as $\mathcal{P}_i$, is required for each observation o_i;
- The hard clauses, ϕ_h, in our MaxSAT formulation correspond to:

$$\phi_h = \bigwedge_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i);$$

- The soft clauses are formulated as:

$$\phi_s = \bigwedge_{c \in \mathcal{C}} h(c).$$

Model-Based Diagnosis with Multiple Test Cases

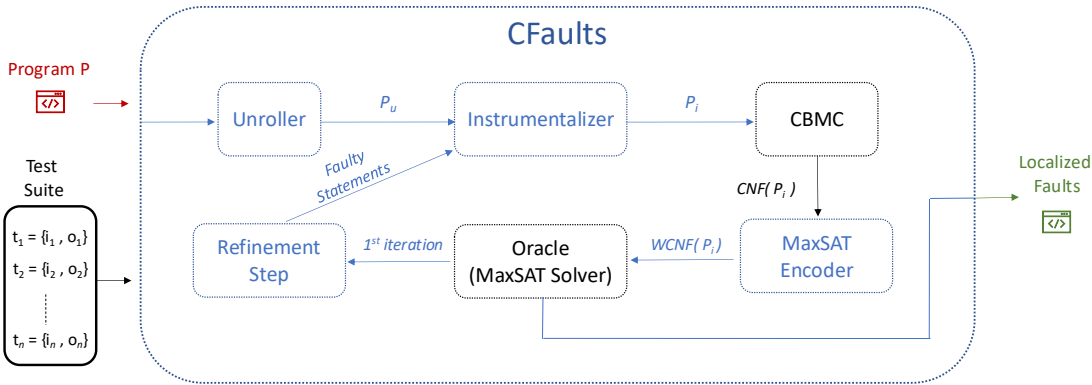
- Given a MaxSAT solution, **the set of unhealthy components ($h(c) = 0$), corresponds to a subset-minimal aggregated diagnosis**.

Model-Based Diagnosis with Multiple Test Cases

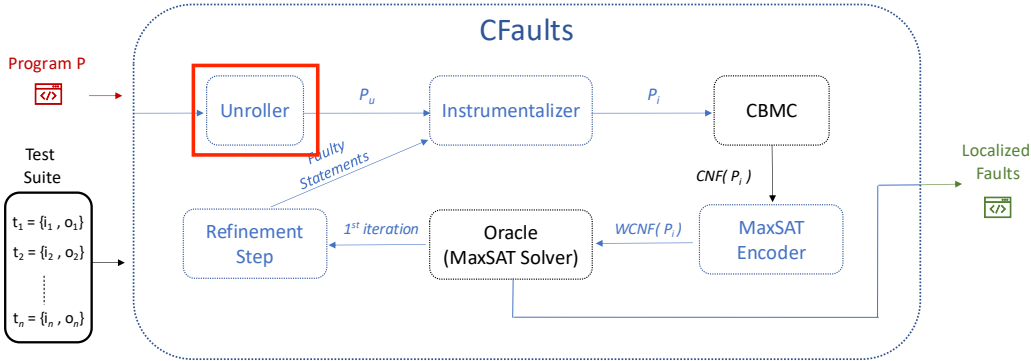
- Given a MaxSAT solution, **the set of unhealthy components ($h(c) = 0$), corresponds to a subset-minimal aggregated diagnosis**.
- This diagnosis makes the system **consistent with all observations**, as follows:

$$\bigwedge\nolimits_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i) \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot$$

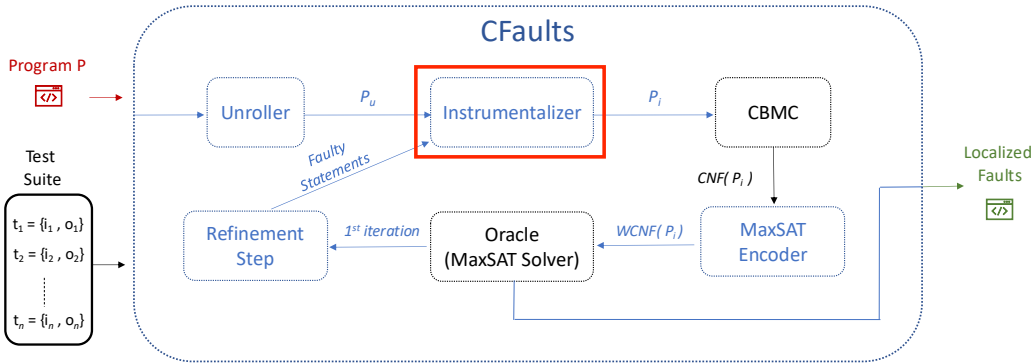
CFaults


CFaults
Program P
Test Suite
$t_1 = \{i_1 , o_1\}$
$t_2 = \{i_2 , o_2\}$
$t_n = \{i_n , o_n\}$
Unroller
P_u
Instrumentalizer
P_i
CBMC
Faulty Statements
CNF(P_i)
Refinement Step
1st iteration
Oracle (MaxSAT Solver)
WCNF(P_i)
MaxSAT Encoder
Localized Faults

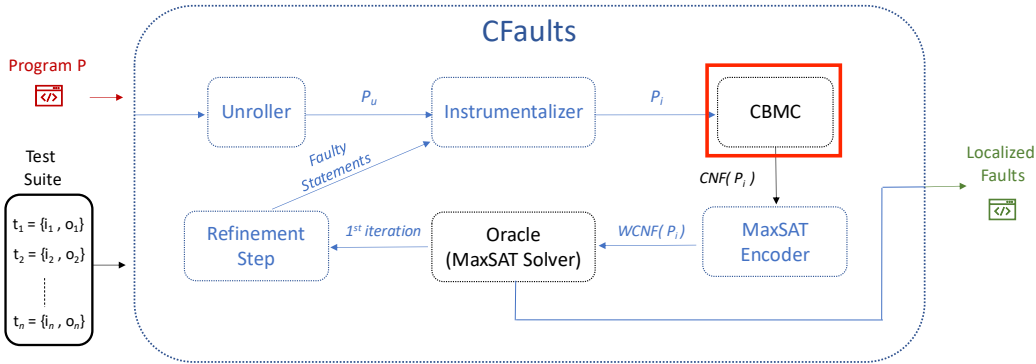
CFaults


CFaults
Program P
Test Suite
$t_1 = \{i_1 , o_1\}$
$t_2 = \{i_2 , o_2\}$
$t_n = \{i_n , o_n\}$
Unroller
P_u
Instrumentalizer
P_i
CBMC
CNF(P_i)
MaxSAT Encoder
WCNF(P_i)
Oracle (MaxSAT Solver)
1st iteration
Refinement Step
Faulty Statements
Localized Faults

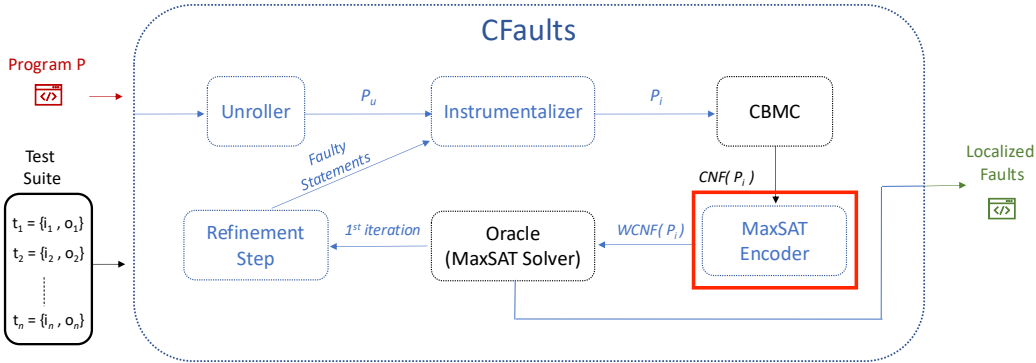
CFaults


CFaults
Program P
Unroller
P_u
Instrumentalizer
P_i
CBMC
Test Suite
$t_1 = \{i_1 , o_1\}$
$t_2 = \{i_2 , o_2\}$
$t_n = \{i_n , o_n\}$
Faulty Statements
$CNF(P_i)$
Localized Faults
Refinement Step
1st iteration
Oracle (MaxSAT Solver)
$WCNF(P_i)$
MaxSAT Encoder

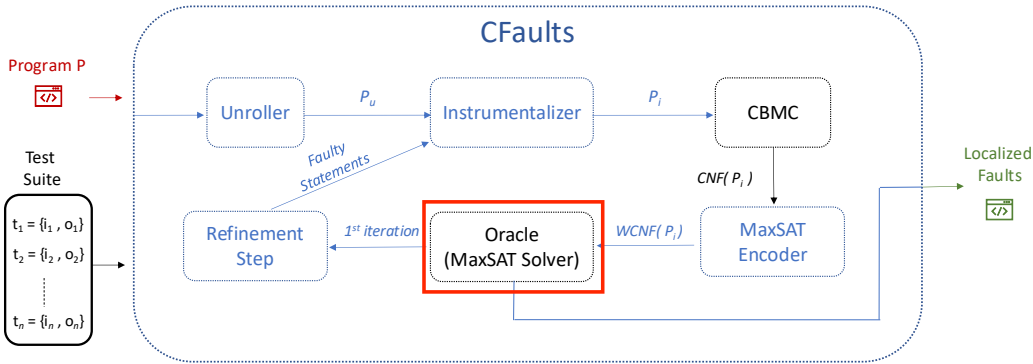
CFaults


CFaults
Program P
Test Suite
$t_1 = \{i_1 , o_1\}$
$t_2 = \{i_2 , o_2\}$
$t_n = \{i_n , o_n\}$
Unroller
P_u
Instrumentalizer
P_i
CBMC
CNF(P_i)
MaxSAT Encoder
WCNF(P_i)
Oracle (MaxSAT Solver)
1st iteration
Refinement Step
Faulty Statements
Localized Faults

CFaults


CFaults
Program P
Test Suite
$t_1 = \{i_1 , o_1\}$
$t_2 = \{i_2 , o_2\}$
$t_n = \{i_n , o_n\}$
Unroller
P_u
Instrumentalizer
P_i
CBMC
CNF(P_i)
MaxSAT Encoder
WCNF(P_i)
Oracle (MaxSAT Solver)
1st iteration
Refinement Step
Faulty Statements
Localized Faults

CFaults


CFaults
Program P
Test Suite
t1 = {i1 , o1}
t2 = {i2 , o2}
tn = {in , on}
Unroller
Pu
Instrumentalizer
Pi
CBMC
Faulty Statements
CNF(Pi)
Refinement Step
1st iteration
Oracle (MaxSAT Solver)
WCNF(Pi)
MaxSAT Encoder
Localized Faults

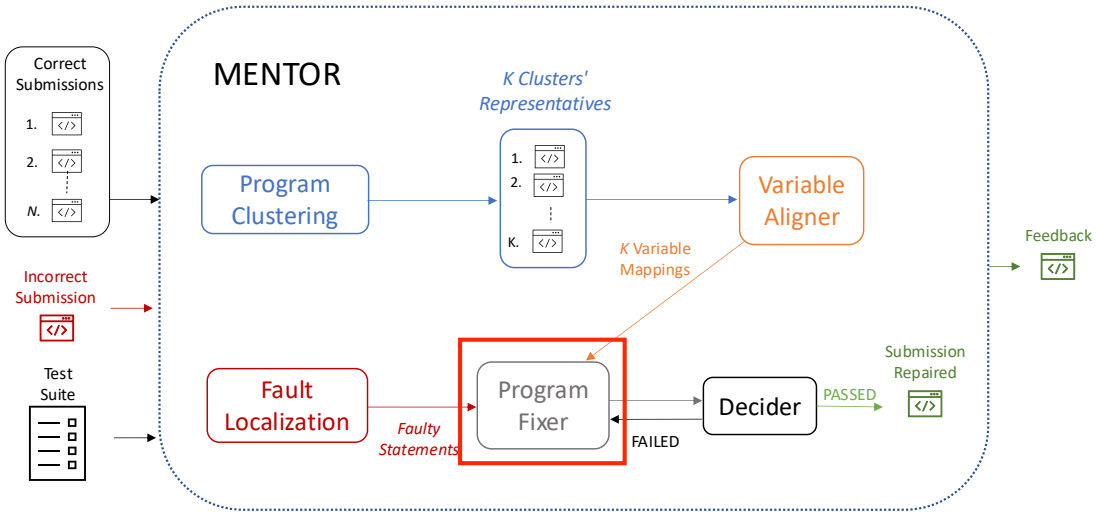
CFaults - Results

Benchmark: **C-Pack-IPAs**

	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	454 (93.42%)	0 (0.0%)	32 (6.58%)
SNIPER	446 (91.77%)	4 (0.82%)	36 (7.41%)
CFaults	**483 (99.38%)**	1 (0.21%)	2 (0.41%)

Table 4: BugAssist, SNIPER and CFaults fault localization results on C-Pack-IPAs.

MENTOR


Correct Submissions
1.
2.
N.
Incorrect Submission
Test Suite
MENTOR
K Clusters' Representatives
1.
2.
K.
Program Clustering
Variable Aligner
K Variable Mappings
Feedback
Fault Localization
Faulty Statements
Program Fixer
Decider
PASSED
FAILED
Submission Repaired

Program Repair

- **AAAI 2025** - Counterexample Guided APR Using MaxSAT-based Fault Localization.

Motivation

12: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

Motivation

13: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

Motivation

14: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

LLMs for code (LLMCs)

- GRANITE and CODEGEMMA **cannot fix** the buggy program within 90 secs;

Motivation

15: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

LLMs for code (LLMCs)

- GRANITE and CODEGEMMA **cannot fix** the buggy program within 90 secs;
- Even if we provide the assignment's **description and IO tests**.

Program Sketches

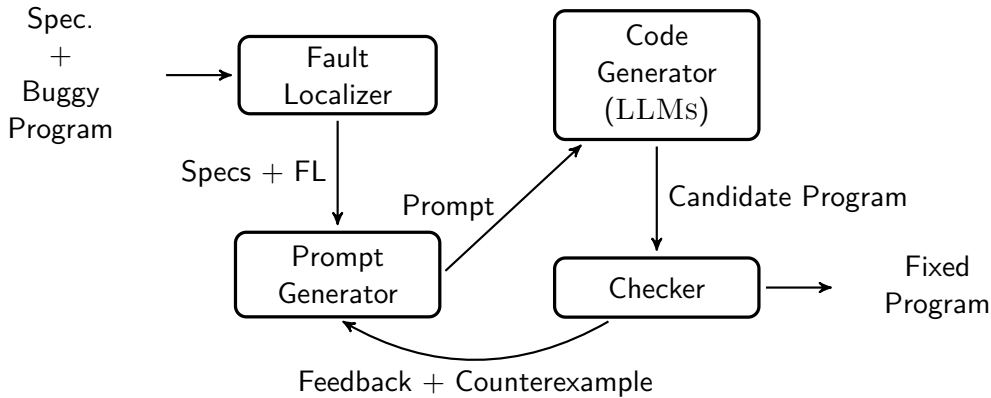
16: Semantically incorrect program. Faults :{4,8}.

```
1  int main(){ //finds max of 3 nums
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9        printf("%d",t);
10
11    return 0;
12 }
```

17: Program sketch with holes.

```
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     @ HOLE 1 @
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     @ HOLE 2 @
9        printf("%d",t);
10
11    return 0;
12 }
```

Counterexample Guided Automated Repair



Prompt Example without Fault Localization

````
Fix all semantic bugs in the buggy program
below. Modify the code as little as possible.
Do not provide any explanation.

### Problem Description ###
Write a program that determines and
prints the largest of three integers
given by the user.

### Test Suite
#input:
6 2 1
#output:
6
// The other input-output tests

# Reference Implementation
(Do not copy this program) <c> #
```c
int main(){
  // Reference Implementation
}
```

### Buggy Program <c> ###
```c
int main(){
  // Buggy program from Listing 1
}
```

### Fixed Program <c> ###
```c
````
````

Prompt with Fault Localization (Sketches)

Complete all the '@ HOLES N @' in the incomplete program below.
Modify the code as little as possible.
Do not provide any explanation.

Problem Description
Write a program that determines and prints the largest of three integers given by the user.

Test Suite
#input:
6 2 1
#output:
6
// The other input-output tests

Reference Implementation
(Do not copy this program) <c> #
```c
int main(){
  // Reference Implementation
}
```

Incomplete Program <c>

```c
int main(){
  // Buggy program from Listing 1
}
```

Complete Program <c>
```c
```

Experimental Setup

- **Evaluation Benchmark:** C-PACK-IPAS, a set of twenty-five IPAS, comprising **1431 faulty programs**;

Experimental Setup

- **Evaluation Benchmark:** C-PACK-IPAS, a set of twenty-five IPAS, comprising **1431 faulty programs**;
- **Large Language Models (LLMs):** We evaluated **six different LLMs**.

Experimental Setup

- **Evaluation Benchmark:** C-Pack-IPAs, a set of twenty-five IPAs, comprising **1431 faulty programs**;
- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Three of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's Granite;
    - Google's CodeGemma;
    - Meta's CodeLlama.

Experimental Setup

- **Evaluation Benchmark:** C-Pack-IPAs, a set of twenty-five IPAs, comprising **1431 faulty programs**;
- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Three of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's Granite;
    - Google's CodeGemma;
    - Meta's CodeLlama.
  - **The other three models are general-purpose LLMs**:
    - Google's Gemma;
    - Meta's Llama3;
    - Microsoft's Phi3.

Experimental Setup

- **Evaluation Benchmark:** C-Pack-IPAs, a set of twenty-five IPAs, comprising **1431 faulty programs**;
- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Three of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's Granite;
    - Google's CodeGemma;
    - Meta's CodeLlama.
  - **The other three models are general-purpose LLMs**:
    - Google's Gemma;
    - Meta's Llama3;
    - Microsoft's Phi3.
- Experiments were conducted using a memory limit of **10GB**, and a timeout of **90s**.

LLM-Driven APR with CFaults

	Prompt Configurations			
LLMs	**De-TS**	**De-TS-CE**	**Sk_De-TS**	**Sk_De-TS-CE**
CodeGemma	597 (41.7%)	606 (42.3%)	682 (47.7%)	**688 (48.1%)**
CodeLlama	492 (34.4%)	500 (34.9%)	**573 (40.0%)**	561 (39.2%)
Gemma	496 (34.7%)	492 (34.4%)	532 (37.2%)	**534 (37.3%)**
Granite	626 (43.7%)	624 (43.6%)	**691 (48.3%)**	681 (47.6%)
Llama3	564 (39.4%)	590 (41.2%)	578 (40.4%)	**591 (41.3%)**
Phi3	494 (34.5%)	489 (34.2%)	**547 (38.2%)**	535 (37.4%)
Verifix	90 (6.3%)			
Clara	495 (34.6%)			

Table 5: The number of programs fixed by each LLM under various configurations. Mapping abbreviations to configuration names: **De** - IPA *Description*, **TS** - *Test Suite*, **CE** - *Counterexample*, **SK** - *Sketches*.

LLM-Driven APR with CFaults + VMs

	Prompt configurations with access to Reference Implementations and Variable Mappings			
LLMs	**Sk_De-TS**	**Sk_De-TS-CE**	**Sk_De-TS-CE-CPA-VM**	**Sk_De-TS-CE-RI-VM**
CodeGemma	682 (47.7%)	688 (48.1%)	**782 (54.6%)**	780 (54.5%)
CodeLlama	573 (40.0%)	561 (39.2%)	**681 (47.6%)**	677 (47.3%)
Gemma	532 (37.2%)	534 (37.3%)	756 (52.8%)	**766 (53.5%)**
Granite	691 (48.3%)	681 (47.6%)	901 (63.0%)	**921 (64.4%)**
Llama3	578 (40.4%)	591 (41.3%)	**792 (55.3%)**	720 (50.3%)
Phi3	547 (38.2%)	535 (37.4%)	**691 (48.3%)**	**691 (48.3%)**

Table 6: The number of programs fixed by each LLM under various configurations. Mapping abbreviations to configuration names: **CPA** - *Closest Program using* AASTs, **De** - IPA *Description*, **RI** - *Reference Implementation*, **SK** - *Sketches*, **TS** - *Test Suite*, **VM** - *Variable Mapping*.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

Research Overview

- Automated Program Repair
  - MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation. **ESEC/FSE 2022**;
  - Graph Neural Networks For Mapping Variables Between Programs. **ECAI 2023**;
  - C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. **APR 2024**;
  - GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. **SIGCSE Virtual 2024**;
  - CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases. **FM 2024**;
  - Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. **AAAI 2025**;
  - On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. **[Under Review]**;
  - MENTOR: Providing Feedback for Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair. **[Under Review]**.

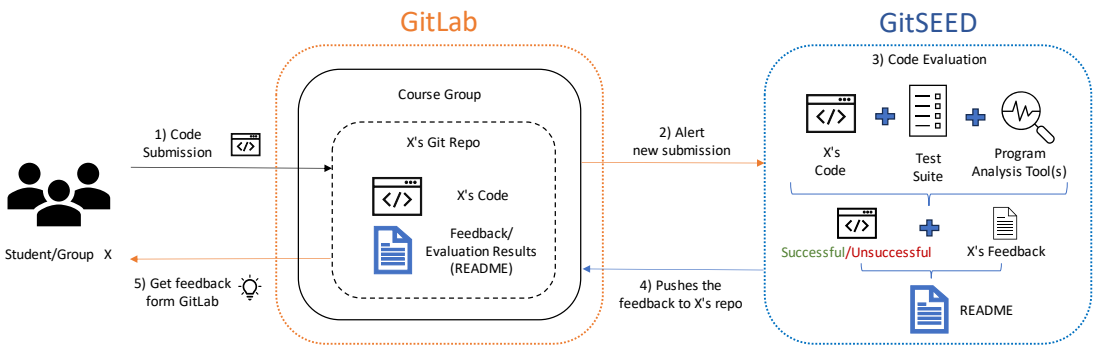
Research Overview

- Program Synthesis:
  - Encodings for Enumeration-Based Program Synthesis. **CP 2019**;
  - SQUARES: A SQL Synthesizer Using Query Reverse Engineering. **VLDB 2020**;

Research Overview

- Program Synthesis:
  - Encodings for Enumeration-Based Program Synthesis. **CP 2019**;
  - SQUARES: A SQL Synthesizer Using Query Reverse Engineering. **VLDB 2020**;
- Maximum Satisfiability (MaxSAT):
  - UpMax: User partitioning for MaxSAT. **SAT 2023**;
  - AlloyMax: Bringing maximum satisfaction to relational specifications. **ESEC/FSE 2021**. [ACM SIGSOFT Distinguished Paper Award];

GitSEED: Git-backed AAT for Software Engineering and Programming Education



Demo

Obrigado!
Thank you!

Pedro Orvalho

Thank you!

https://pmorvalho.github.io

References


Reiter, Raymond (1987)
A Theory of Diagnosis from First Principles.
Artif. Intell. 1987.


Do, Hyunsook and Elbaum, Sebastian G. and Rothermel, Gregg (2005)
Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.
Empir. Softw. Eng. 2005.


Jose, Manu and Majumdar, Rupak (2011)
Cause clue clauses: error localization using maximum satisfiability.
PLDI 2011.


Lamraoui, Si-Mohamed and Nakajima, Shin (2016)
A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs.
J. Inf. Process. 24(1), 88 – 98.

References


Ignatiev, Alexey and Morgado, António and Weissenbacher, Georg and Marques-Silva, João (2019)
Model-Based Diagnosis with Multiple Observations.
IJCAI 2019.


Orvalho, Pedro and Janota, Mikolas and Manquinho, Vasco (2024)
C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments.
Automated Program Repair (APR) 2024.


The Guardian - Year 2000 Problem
www.theguardian.com/commentisfree/2019/dec/31/millennium-bug-face-fears-y2k-it-systems
The Guardian 2019.


The Guardian UK - Crowdstrike Meltdown
https://www.theguardian.com/technology/article/2024/jul/24/crowdstrike-outage-companies-cost.
The Guardian UK.


Ahmed, Umair Z and Fan, Zhiyu and Yi, Jooyong and Al-Bataineh, Omar I and Roychoudhury, Abhik (2022)
Verifix: Verified repair of programming assignments.
TOSEM 22 12(3), 45 – 678.

References

Orvalho, Pedro and Janota, Mikoláš and Manquinho, Vasco (2022)

MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation.

ESEC/FSE 2022.

Gulwani, Sumit and Radiček, Ivan and Zuleger, Florian (2018)

Automated clustering and program repair for introductory programming assignments.

PLDI 18 52(4), 465 – 480.

Orvalho, Pedro and Janota, Mikolas and Manquinho, Vasco (2024)

CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases.

Formal Methods (FM) 2024.

Orvalho, Pedro and Janota, Mikolas and Manquinho, Vasco (2025)

Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization.

AAAI 2025.