# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Program Analysis</td>
<td>2</td>
</tr>
<tr>
<td>Definition</td>
<td>2</td>
</tr>
<tr>
<td>Examples of Dynamic Analysis Programs</td>
<td>2</td>
</tr>
<tr>
<td>Static Program Analysis</td>
<td>2</td>
</tr>
<tr>
<td>Definition</td>
<td>2</td>
</tr>
<tr>
<td>Examples of static analysis programs</td>
<td>3</td>
</tr>
<tr>
<td>Code Invariance</td>
<td>6</td>
</tr>
<tr>
<td>Iterative Approximation</td>
<td>7</td>
</tr>
<tr>
<td>Live Variables</td>
<td>8</td>
</tr>
<tr>
<td>More examples of static Analysis</td>
<td>10</td>
</tr>
<tr>
<td>The Termination Problem</td>
<td>10</td>
</tr>
<tr>
<td>Challenges of Static Program Analysis</td>
<td>11</td>
</tr>
<tr>
<td>Abstract Domain Analysis</td>
<td>11</td>
</tr>
<tr>
<td>The basics of Static Analysis – Interpreting a program over an “Abstract Domain”</td>
<td>12</td>
</tr>
<tr>
<td>Examples of abstract interpretation</td>
<td>12</td>
</tr>
<tr>
<td>Abstraction – Graphic Interpretation</td>
<td>13</td>
</tr>
<tr>
<td>Abstraction Example I</td>
<td>15</td>
</tr>
<tr>
<td>Abstraction Example II</td>
<td>18</td>
</tr>
<tr>
<td>Abstraction Example III</td>
<td>19</td>
</tr>
<tr>
<td>Fundamental Issues of Undecidability</td>
<td>21</td>
</tr>
</tbody>
</table>
Dynamic Program Analysis

**Definition** – Analysis of computer software that is done while executing the program. The results, speed and depth of the analysis depend on the program's input.

**Examples of dynamic analysis programs:**

1. Programs that make sure that memory access is in bounds and valid, and that no leakage occurs:
   a. BoundsChecker – A memory access tool for Windows based programs.
   b. Valgrind – Runs the program on a “Valgrind” virtual machine that allows Valgrind to control memory access.
   c. Purify – Links the program source code with Purify code. The added code is than able to report information about where and how the error occurred.

2. Detecting variable that remain invariant at specific locations:
   a. Daikon – Runs a program over a number of different variables and reports what variables don’t change at certain locations.

3. Program optimization:
   a. Pareon – A program that suggests code optimization for multicore systems based on parallelization. In other words, watches the execution and sees what can be run in parallel.
   b. Intel thread profiler & Vtune – A program that is able to track cpu usage, stack state, and hardware events throughout the program. Data about the program can then be accessed and used to optimize thread performance.

Static Program Analysis

**Definition** – Analysis of computer software that is performed on the program code without actually running the code. The analysis can be performed on the source code, or the object code, but will usually be done on the “higher-level”. Static analysis returns insights that are relevant invariantly.
Examples of static analysis programs

1. Compilers usually make use of static program analysis in order to warn about errors in advance of the code running. This is usually done by compile-time static analysis which deals with:
   a. Stronger type checking (for example in C).
   b. Array bound violations.
   c. Identify dangling pointers.
   d. Generate test cases.
   e. Generate certification proofs.

Examples of static program analysis compilers:

   f. CODAN – A static analysis tool made use of by the Eclipse compiler.
   g. Clang – Another C++ compiler tool.
   h. **Problematic code example:**
      i. The following code leads memory leakage, regardless of the variables that are given:
         1. List reverse(Element *head){
         2.     List rev, n;
         3.     rev = NULL;
         4.     while (head != NULL) {
         5.         n = head -> next;
         6.         head -> next = rev
         7.         head = n;
         8.         rev = head;
         9.     }
        10. return rev;
        11. }

        The above program will change the first node’s pointers correctly. However, the result of row 8 is that “rev” and “head” will be equal going into the loop for the second time. If we continue into the loop for the second time, we can see in row 6 that “head” will in essence be pointing at itself.

      ii. The following is the corrected code:
         1. Element* reverse(Element *head) {
         2.     Element *rev, *n;
         3.     rev = NULL;
         4.     while (head != NULL) {
         5.         n = head -> next;
         6.         head -> next = rev;
         7.         rev = head;
         8.         head = n;
         9.     }
        10. return rev;
        11. }
The change in the above code results in “rev” being equal to the address of the node that was one node before the new “head”. Note that the next iteration of the loop will have “head” connect to “rev”. Hence, the above change has fixed the problem seen in the previous code.

2. Interface validity tools that make sure that a programs API is used as specified:
   a. SLAM (Software, Languages, Analysis, and Modeling) – A Microsoft program that analysis the interface of new components to make sure that they “connect” with Microsoft drivers in a valid way. The SLAM algorithm works as follows:
      i. Merge the driver source code, codified as SLIC rules for correct driver interface, with the components source code. The combining is done by C2BP (C too Boolean Program). The combined code is a Boolean program that is in essence an abstract version of the original component’s code and the driver’s code (more on abstraction later). The simplified program will be known as Bebop.
      ii. Run the Bebop.
      iii. If Bebop returns true (there is an error), than:
           1. “Newton” program is run to determine the feasibility of the error. If it is feasible, return the bug and close.
           2. If the error is infeasible, create predicates that rule out the error (assertions), and rerun Bebop (return to ii.)
      iv. Run C2BP again and create a less abstract version of the code (return to i.).

As can be seen, the above code does not necessarily terminate, but continues creating more and more specific versions of the code until bugs are found.

b. CPPLint – Google’s software compliance checker. Verifies that the program complies with Google’s code styling guidelines.

3. Software Security analysis:
   a. HP Fortify Software – Reviews the program for known security vulnerabilities.
   b. IBM Rational AppScan – Reviews early source code for security vulnerabilities.
   c. **Problematic code example:**
      i. The following code contains the potential for a buffer overrun attack:
         1. void foo(char *s ) {
         2. while ( *s != ’ ’ ){
         3.     s++;;
         4. } *s = 0;
         5. }
         6. }
Note, the above code never checks that the above pointer is valid (non-null), nor does it ever assert that the char pointer points a string. The above code could end up continuing indefinitely and overwriting large portions of the memory.

ii. The following is the fixed code:
   1. void foo(char *s) @require string(s) {
   2.     while ( *s != ' ' && *s != 0) {
   3.         s++;
   4.     } *s = 0;
   5. }

   The above function on the other hand only accepts strings, and therefore, by definition, the requirement of the while() will eventually be fulfilled. Furthermore, all of the values that will be changed before the loop terminates will have been part of the string.

iii. The following code could be taken advantage of using buffer overrun in order to access a systems passwords:
   1. int check_authentication(char *password) {
   2.     int auth_flag = 0;
   3.     char password_buffer[16];
   4.     strcpy(password_buffer, password);
   5.     if(strcmp(password_buffer, "brillig") == 0)
   6.         auth_flag = 1;
   7.     if(strcmp(password_buffer, "outgrabe") == 0)
   8.         auth_flag = 1;
   9.     return auth_flag;
   10. }

   The above code could be exploited as follows:
   1. The program sends check_authentication () a char array that is longer than 16 chars.
   2. strcpy() will overwrite the size of password_buffer and continue writing on the heap.
3. If important values exist directly after the memory location of password_buffer, for example auth_flag, the program might change their value in a malicious way.

4. Static program analysis can also be done by the programmer/designer in order to deepen his/her understanding of the code.

**Code Invariance:**
Program invariants occur when a program always acts in the same way at a specific location. Recognizing program invariants can allow a program to be simplified, conceptually and/or practically in order to save run time.

**Examples:**

1. Arithmetic invariants – The following statements always return the same arithmetic value, and hence never need to be calculated:
   a. $(a * 2)\%2$
   b. $((a * 2)\%2 == 0)$
   c. $\text{long } a/2^{32}$. This will always be zero because $a$ is bounded.

2. Route invariants – Code segments that will return the same value regardless of the route they take:
   a. Example 1:
      i.  
      ii.  
      iii. $return 0;$
   b. Example 2:
      i.  
      ii.  
      iii. $a++;$
      iv. $return a\%2;$

**Spot the code invariances in the following code sections:**

1. Code section I:
   a. int p(int x){
   b. return x *x ;
   c. void main()}
   d. {
   e. int z;
   f. if (getc()) $z = p(6) + 8;$
   g. else $z = p(-7) -5;$
   h. printf (z);
}

2. Code Section II:
   a. int x
   b. void p(a) {
c. read (c);
d. if c > 0 {
  e. a = a -2;
  f. p(a);
  g. a = a + 2;
  h. }
  i. x = -2 * a + 5;
  j. print (x);
  k. }
l. void main {
  m. p(7);
  n. print(x);
  o. }

**Answers:**

1. Line 9 will always print 44.
2. Regardless of what is printed inside the recursion, line 14 will always print -9.

**Iterative Approximation:**

One method of performing static analysis is the iterative approximation method. Iterative approximation works by “running the program” over a set of possible values instead of over a single value. For example, for the following of code:

\[
\text{int } f(\text{int } z)\{
\text{z = z/20;}
\text{return z;}\}
\]

If we assume that only the numbers 1..10 will be given than:

\{1,2,3,4,5,6,7,8,9,10\} → \( f \rightarrow \{0\}.\)

In other words, we know that if we “run” the function over the numbers 1..10, the function will always return 0.

Iterative approximation means running sections of code over all the possible variable values, and noting where sets of possible values condense to one value.

It is often useful to perform iterative approximation over simplified versions of the program.

*Side note:* In certain senses, interactive approximation is similar to calculating the unfolding of a physical event based on the events wave function. In both cases, the current “state” is represented by a mix of all possible variables; the state is then operated on by the “world function.”
Example 1:

The following program contains variables `int x, y, z`. We will use the iterative approximation method in order to discover coding invariance.

We will represent the value at every state by `{< possible values of X >, < possible values of Y >, < possible values of Z >}

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Location in the code</th>
<th>Value (x,y,z)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>{?, ??}</td>
<td>This is true because at the first position we have no information.</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>{?, ?,3}</td>
<td>This state is never reached because z = 3 at state B.</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>{?, ??}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>{?, ?,3}</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>{1, ?,3}</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>{?, ?,3}</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>{?,7,3}</td>
<td>This is true because it doesn’t matter which side we arrive at the state from, Y receives the same value.</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>{?,7,3}</td>
<td>If D is arrived at from G, than it receives this value.</td>
</tr>
</tbody>
</table>

**Live Variables:**

*Definition:* Variable `x` will be considered alive at point `A`, if at any state after point `A` the value of `x` can possible be used.

Determining what variables are alive at certain locations can be used by the compiler in order to determine register allocation. Static program analysis can be used by the compiler in order to determine the “liveness” of variables at different points in the code.
Algorithm for determining “liveness”:

At the end of the program all of the variables are dead.

For each state $N$, the liveness of the variables is determined by by:

$$(Variables\ that\ are\ alive\ at\ N + 1)\ \cup (Variables\ that\ are\ used\ to\ calculate\ live\ variable\ N + 1) - (Values\ that\ are\ overwritten\ at\ N)$$

Using reverse propagation, we continue until the current state does not change.

**Example:**

Each state will be represented by a set of all the live variables at that state.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Location in the code</th>
<th>Live Variables</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>${c}$</td>
<td>Because $c$ is used when going from $E$ to $F$.</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>${c}$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>${c, b}$</td>
<td>Because $c$ is calculated by using $b$.</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>${c, a}$</td>
<td>Because $a$ is used to calculate $b$, and because $b$ is overwritten.</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>${c, a}$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>${c, b}$</td>
<td>Because $b$ is used to calculate $a$, and because $a$ is overwritten.</td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td>${c, b}$</td>
<td></td>
</tr>
</tbody>
</table>
The algorithm terminates here because no change was made. In this case, the compiler “knows” that no more than 2 variables are ever “alive”; hence only 2 registers are needed.

**More examples of Static program Analysis:**

1. Reaching definitions – Determining what variables “reach” (or “survive”) unchanged from location a to location b.
2. Expressions that are ``available'' – Determining which values are “available”. A value is said to be available at a given point if the value does not need to be recomputed, i.e. if it won’t change before reaching the point.
3. Dead code – Determining if a portion of the source code does not change the outcome of the program.
4. Pointer variables never point into the same location.
5. Points in the program in which it is safe to free an object.
6. An invocation of virtual method whose address is unique – Determining if virtual methods can be overwritten.
7. Statements that can be executed in parallel.
8. An access to a variable which must be in cache.
9. Integer intervals.
10. The termination problem.

**The Termination Problem:**

The termination problem is an attempt to determine from the source code if a given program will terminate for all possible inputs.

The termination problem is theoretically very difficult (Note: recall that to determine if a function terminates for a given set of inputs is a Recursively Enumerable problem. Determining termination for all possible inputs is not even Recursively Enumerable) because it involves determining invariances throughout the program. However, the termination of most practical programs can be determined fairly easily. There are currently a number of programs that solve termination for most “normal” programs, for example, MSR Terminator and ARMC.

**Examples:**

Determine if the following programs terminate:

1. Program I:
   a. \( z := 3; \)
   b. \( \text{while } z > 0 \text{ do } \{
   \)   c. \( \text{if } (x == 1) z := z +3; \)
   d. \( \text{else } z := z + 1; \)
2. Program II:
   a. \( \text{while } z > 0 \text{ do } \{
   \)   b. \( \text{if } (x == 1) z := z -1; \)
   c. \( \text{else } z := z -2; \)
3. Program III:
a. while (x !=1) do {
  b. if (x %2) == 0 {
      c. x := x / 2;
    } 
  d. else {
      e. x := x * 3 + 1;
    }
  f. }

**Answers:**

1. Never terminates.
2. Always terminates.
3. Program III’s termination is an open problem in computer science. It’s termination is part of the “Collage Theorem” which deals with convergence and fractal space.

**Challenges of Static Program Analysis**

1. Non-trivial – Being able to discover relevant issues.
2. Correctness – Guaranteeing that all errors are caught.
3. Precision – Minimizing the amount of false warnings.
4. Efficiency of the analysis – Performing the analysis quickly and with minimal resources.
5. Scaling – Being able to analyze small and large programs.

**Example – The C Compiler:**

The C compiler was designed to minimize the need for static analysis. However, the very flexibility that was built into C leads to a need for static analysis in order to avoid errors. These days C compilers spend most of their compilation time on static analysis.

An example of a potential error caused by the flexibility in memory allocation allowed in C:

a = malloc();
b = a;
cfree (a);
c = malloc();
if (b == c)
printf(“unexpected equality”);

**Abstract Domain Analysis**
The basics of Static Analysis – Interpreting a program over an “Abstract Domain”:
Determining a function over an “Abstract Domain” is a way to check the validity of a function which guarantees the soundness of the analysis. This is done by “rounding up”. In a sense, a simplified version of the program is applied to a “worst case”, all inclusive set of variables.

Lessons from this “worst case” application of the program are then analyzed for possible problems. The above definition leads to a system of interpretation which yields:

1. All constants found by this analysis are guaranteed.
2. Not all constants are found.

Abstract Interpretation

The α is an abstraction function, γ is a concretization function, and γ(α( )) is known as a Galois function. For our use, abstraction is the process of finding a common denominator for a set of possible values. Concretization is taking an abstraction and returning it to the set that includes all values of the abstractions. The Galois function never decreases the site of the concrete set.

Examples of abstract interpretation:

1. Casting out nines:
   a. A system for checking the validity of arithmetic equations by checking simpler equations.
   b. Algorithm:
i. Whenever an intermediate result exceeds 8, replace by the sum of its digits (recursively).
ii. Report an error if the values do not match.

c. Theory:
   i. \((10a + b) \mod 9 = (a + b) \mod 9\)
   ii. \((a+b) \mod 9 = (a \mod 9) + (b \mod 9)\)
   iii. \((a\times b) \mod 9 = (a \mod 9) \times (b \mod 9)\)

d. Example:
   i. “123 * 457 + 76543 = 132654” – is the equation valid?
   ii. Left side: 123*457 + 76543 = 6 * 7 + 7 = 6 + 7 = 4
   iii. Right side: 3
   iv. Report an error

2. Verbal example (from Wikipedia):

   Consider the people in a conference room. Suppose you had a list of unique identifiers for each person in the room, like a social security number in the United States. To prove that someone is not present, all one needs to do is see if their social security number is not on the list. Since two different people cannot have the same number, it is possible to prove or disprove the presence of a participant simply by looking up his or her number. However it is possible that only the names of attendees were registered. If the name of a person is not found in the list, we may safely conclude that that person was not present; but if it is, we cannot conclude definitely without further inquiries, due to the possibility of homonyms (for example, two people named John Smith). Note that this imprecise information will still be adequate for most purposes, because homonyms are rare in practice. However, in all rigor, we cannot say for sure that somebody was present in the room; all we can say is that he or she was possibly here. If the person we are looking up is a criminal, we will issue an alarm; but there is of course the possibility of issuing a false alarm. Similar phenomena will occur in the analysis of programs.

   If we are only interested in some specific information, say, “was there a person of age \(n\) in the room?”, keeping a list of all names and dates of births is unnecessary. We may safely and without loss of precision restrict ourselves to keeping a list of the participants' ages. If this is already too much to handle, we might keep only the age of the youngest, \(m\) and oldest person, \(M\). If the question is about an age strictly lower than \(m\) or strictly higher than \(M\), then we may safely respond that no such participant was present. Otherwise, we may only be able to say that we do not know.

**Abstraction – Graphic interpretation:**
In other words, the process here can be seen as taking the concrete set of variables, abstracting it, and performing operations on the abstract set. This set is then concretized and analysed.
**Abstraction Example I:**

In this example, we will use abstraction to the evenness or oddness of $x$ throughout the function.

1. while $(x != 1)$ do {
2.     if $(x \% 2) == 0$
3.         { $x := x / 2; \}$
4.     else
5.         { $x := x * 3 + 1;$
6.     }

We shall attempt to assert that at row 6, $x$ is always even. The abstraction function can be seen as:
We can see from the above abstraction that at row 5, \( x \) is always odd, and therefore we can assert that at row 6, \( x \) is always even.

**Example:**

An abstraction of addition and multiplication as it effects odds and evens can be seen here:
The above tables show the connection between abstract semantics and concrete semantics.

**Abstraction Example II:**

Determine for each variable if it is positive or negative at each location in the program.

We shall define the abstraction: \{P, N, ?\}. The abstraction of multiplication is given by:

\[
\begin{array}{c|ccc}
+ & P & N & ? \\
\hline
P & P & N & ? \\
N & N & P & ? \\
\end{array}
\]

\[
\begin{array}{c|c}
\times & P & N \\
\hline
P & P & N \\
N & N & P \\
? & ? & ? \\
\end{array}
\]
Abstraction Example III:
In the following example, we will attempt to determine, through abstraction, where the value of the variables is constant. This form of abstraction is known as constant propagation. In essence, we go through the program from the beginning, and perform the function on the set of possible values.

The abstract semantics for addition are given by:

<table>
<thead>
<tr>
<th></th>
<th>?</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+#</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>?</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>?</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The abstraction of multiplication is:
Example program 1:

1. \( x = 5; \)
2. \( y = 7; \)
3. if (getc())
4. \( y = x + 2; \)
5. \( z = x + y; \)
6. 

Propagation:

1. \( x = \{5\}, y = \{?\} \)
2. \( x = \{5\}, y = \{7\} \)
3. \( x = \{5\}, y = \{7\} \)
4. \( x = \{5\}, y = \{7\} \)
5. \( x = \{5\}, y = \{7\} \), here we merged the possibility of row 4 with the possibility of not row 4, since they are the same we receive the above results.
6. \( x = \{5\}, y = \{7\} \)

Example program 2:

1. if (getc())
2. \( x = 3; y = 2; \)
3. else
4. \( x = 2; y = 3; \)
5. \[ z = x + y; \]

6. 

**Propagation:**

1. \[ x = \{?\}, y = \{?\}, z = \{?\} \]
2. \[ x = \{3\}, y = \{2\}, z = \{?\} \]
3. \[ x = \{?\}, y = \{?\}, z = \{?\} \]
4. \[ x = \{2\}, y = \{3\}, z = \{?\} \]
5. \[ x = \{?\}, y = \{?\}, z = \{?\} \], here we merged the possibility of row 4 with the possibility of row 2, hence, we lost our constants.
6. \[ x = \{?\}, y = \{?\}, z = \{?\} \]

As we can see from the above example, some programs contain undecidability issues. In other words, questions that we cannot determine, even for ideal program conditions.

**Fundamental Issues of Undecidability:**

Some static analysis problems are undecidable even if the program conditions are ignored.

**Coping with undecidability:**

In order to get around the above problem, we can limit the type of “valid” programs. By limiting the type of programs that we deal with, we can force to program to make our analysis job easier. Examples of self-imposed limitations:

1. Programs with no loops.
2. Allowing the analysis program to ask questions.
3. Non-optimal analysis:
   a. Warnings for non-erroneous states.
   b. Non optimal solutions (rounding errors).

**More on program abstraction:**

**History of abstraction:**

1. [Naur1965] The Gier Algol compiler “A process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not their value”.
2. [Reynolds 1969] Interesting analysis which includes infinite domains (context free grammars)
3. [Syntzoff 1972] Well foudedness of programs and termination
4. [Cousot and Cousot 1976,77,79] The general theory
5. [Kamm and Ullman, Kildall 1977] Algorithmic foundations
7. [Sharir and Pnueli 1981] Foundation of the interprocedural case
8. [Allen, Kennedy, Cock, Jones, Muchnick and Schwartz]

**Example of modern abstraction:**

TVLA - Tal Lev-Ami, Alexey Loginov, Roman Manevich.

An abstraction program for checking programs for errors based on shape analyzing the algorithm. The program works by “merging” multiple states into “equivalency states”. In the bellow example, working with linked lists, when adding a new value to the linked list, all the different values for the rest of the list are equivalent (because adding a new value only moves the head of a linked list):
Summary of Basic Program Analysis:

Runtime vs. Static Testing

<table>
<thead>
<tr>
<th></th>
<th>Runtime</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effectiveness</td>
<td>Missed Errors</td>
<td>False alarms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Locate rare errors</td>
</tr>
<tr>
<td>Cost</td>
<td>Proportional to program’s execution</td>
<td>Proportional to program’s size</td>
</tr>
<tr>
<td></td>
<td>No need to efficiently handle rare cases</td>
<td>Can handle limited classes of programs and still be useful</td>
</tr>
</tbody>
</table>

Note: In the examples section of static analysis, two programs (HP Fortify Software & IBM Rational AppScan) were mentioned. Although these programs are “static analysis programs”, these tools analyse programs semantically instead of looking for invariances. In this course, we will not deal with tools like this, which are for example
unsafe and incomplete (as opposed to invariance analysis programs), and in general, use different assumptions for their work.

**Relation to Program Verification**

<table>
<thead>
<tr>
<th>Program Analysis</th>
<th>Program Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>♦ Fully automatic</td>
<td></td>
</tr>
<tr>
<td>♦ Applicable to a programming language</td>
<td></td>
</tr>
<tr>
<td>♦ Can be very imprecise</td>
<td></td>
</tr>
<tr>
<td>♦ May yield false alarms</td>
<td></td>
</tr>
<tr>
<td>♦ Requires specification and loop invariants</td>
<td></td>
</tr>
<tr>
<td>♦ Program specific</td>
<td></td>
</tr>
<tr>
<td>♦ Relative complete</td>
<td></td>
</tr>
<tr>
<td>♦ Provide counter examples</td>
<td></td>
</tr>
<tr>
<td>♦ Provide useful documentation</td>
<td></td>
</tr>
<tr>
<td>♦ Can be mechanized using theorem provers</td>
<td></td>
</tr>
</tbody>
</table>

The imprecision of Program analysis can be seen in the following example of non-homomorphic abstractions:

We will be dealing with program analysis, which is broader and more general. We shall define criteria for precision. The following are the levels of precision that we will use (from most precise to least):
1. Precise (with respect to a subset of the programs).
2. Precise under the assumption that all paths are executable (statically exact).
3. Relatively optimal with respect to the chosen abstract domain.
4. Good enough.