Università degli Studi di Trento

Facoltà di Scienze

Corso di Laurea in Informatica

Analisi e Testing del Software

dispense delle lezioni
`http://selab.fbk.eu/swat`

Docente: Ing. Paolo Tonella

11 dicembre 2007

Contents

Chapter 1

Introduction

Software maintenance has been defined as "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment." (ANSI/IEEE, 1983)

Maintenance activities can be classified into four categories:

- Perfective maintenance
- Adaptive maintenance
- Corrective maintenance
- Preventive maintenance

Software maintenance absorbs a relevant amount of resources. A rule of thumb often used by developers is the 80-20 rule: twenty percent of the effort is in development and eighty percent is in maintenance, during the entire life cycle of the system.

Maintenance activities are difficult, because of the effects of evolution (i.e., software changes over time) on the system. The laws of software evolution, coded by Lehman, summarize his findings in observing the typical patterns of evolution of several large software systems:

1. *Continuing change.* A program that is used undergoes continual change or becomes progressively less useful. The change or decay process

continues until it is more cost-effective to replace the system with a recreated version.

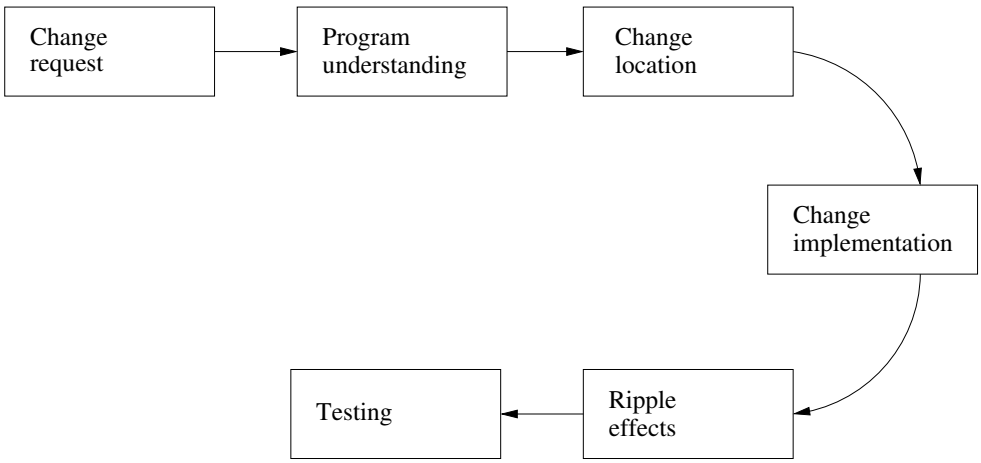
2. *Increasing complexity.* As an evolving program is continually changed, its structure deteriorates. Reflecting this, its complexity increases, unless work is done to maintain or reduce it.

3. *Fundamental law of program evolution.* Program evolution is subject to a dynamic that makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically-determinable trends and invariances.

4. *Conservation of organizational stability (invariant work rate).* During the active life of a program, the global activity rate in a programming project is statistically invariant.

5. *Conservation of familiarity (perceived complexity).* During the active life of a program, the release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

Software maintenance is characterized by the following fundamental activities:



The systems for which maintenance has become the main activity with respect to any other activity are called *legacy systems*. They typically exhibit the following features:

- They have been implemented several years ago.
- Their technology has become obsolete (programming languages, coding style, hardware).
- They have been maintained for a long time.
- Their structure deteriorated, thus making code comprehension very difficult.
- Their original authors are no longer available.

On the other side, their replacement with a brand new system is not feasible because:

- They contain business rules not recorded elsewhere.
- They represent a large investment of the company.
- The risks and costs of rewriting them are not affordable.

The main approaches to the maintenance of legacy systems are:

- Reverse engineering.
- Reengineering.
- Restructuring.

Code analysis support all the three options above. More generally, code analysis helps software engineers during maintenance by providing information that simplifies program understanding and allows assessing the impact of a change.

Flow analyses and slicing highlight the dependences that hold between single program instructions, while reverse engineering aims at extracting a high level view of the organization of the system. Restructuring can be supported by code analyses, which identify opportunities for restructuring and suggest possible changes, while highlighting the impact of the modification.

Software testing is a key activity, both in software development and in software maintenance. Its objectives are opposed to that of development: instead of making the system work, it aims at breaking it, thus revealing the presence of defects. This is the main reason why development and testing teams should be separate.

Code analyses are extremely helpful to testing. Specifically, *structural testing* (aka white-box testing), as opposed to *functional testing* (aka black-box testing), assumes the possibility to access the internal structure of the program, analyze it and derive testing criteria from such an analysis. The outcome supports:

- Test case production/automatic generation.
- Definition of stopping criteria.

Chapter 2

Static code analysis

2.1 Flow analysis

Control flow graph (CFG): program statements are the nodes of this graph. A particular node is marked as the initial node (n_1). A directed edge connects node n to node m if m is (syntactically) one of the direct successors of statement n in the program. Sometimes it is also useful to assume the existence of a unique final node (n_N), that can be fictitiously added if not present.

Flow analysis framework: general description of a procedure that can be used to determine properties that hold for a given control flow graph by propagating proper information flows inside such a graph. These information flows are altered during propagation according to the computation performed by each program statement.

A flow analysis framework consists of:

- V, the set of *values* that can be propagated in the graph. Elements of V will be assigned to IN[n] and OUT[n], for each node n of the control flow graph.
- F, the set of the transfer functions. Every node n is associated with a transfer function f_n: $V \to V$. Such a function represents the computation performed by node n: OUT[n] = f_n(IN[n]).
- The confluence (*meet*) operator, $\bigwedge$, used to join values coming from the OUT sets of the predecessors (or successors), into the IN set of the current node: IN[n] = $\bigwedge_{p \in pred(n)}$ OUT[p].

- The direction of propagation (*forward* or *backward*), which determines whether information is joined from predecessors or successors.

Flow analysis algorithm (forward propagation):

```
1   for each node n
2       IN[n] = ⊤
3       OUT[n] = f_n(IN[n])
4   end for
5   while any OUT[n] changes
6       for each node n
7           IN[n] = ⋀_{p∈pred(n)} OUT[p]
8           OUT[n] = f_n(IN[n])
9       end for
10  end while
```

Assumptions of the algorithm:

- F contains the identity: $\exists f \in F : f(x) = x$ for all $x \in V$.
- F is close with respect to composition: if f and g are in F, then $h(x) = g(f(x))$ is also in F.
- $\bigwedge$ is associative, commutative and idempotent:
  $x \bigwedge (y \bigwedge z) = (x \bigwedge y) \bigwedge z$
  $x \bigwedge y = y \bigwedge x$
  $x \bigwedge x = x$
- There exists a *top* element $\top$, such that:
  $\top \bigwedge x = x$
- The transfer functions are monotone:
  $x \leq y \Rightarrow f(x) \leq f(y)$ or, equivalently:
  $f(x \bigwedge y) \leq f(x) \bigwedge f(y)$
  where the (partial) order relationship is defined as:
  $x \leq y \Leftrightarrow x \bigwedge y = x$

Lemma: It is always true that $x \bigwedge y \leq x$ and that $x \bigwedge y \leq y$.

Theorem: The condition (1): $x \leq y \Rightarrow f(x) \leq f(y)$ is equivalent to the condition (2): $f(x \wedge y) \leq f(x) \wedge f(y)$.

Proof:
(1) $\Rightarrow$ (2):
$x \wedge y \leq x \Rightarrow_{(1)} f(x \wedge y) \leq f(x) \Leftrightarrow f(x \wedge y) \wedge f(x) = f(x \wedge y)$
$x \wedge y \leq y \Rightarrow_{(1)} f(x \wedge y) \leq f(y) \Leftrightarrow f(x \wedge y) \wedge f(y) = f(x \wedge y)$
Replacing the latter into the former:
$f(x \wedge y) \wedge f(y) \wedge f(x) = f(x \wedge y) \Leftrightarrow f(x \wedge y) \leq f(x) \wedge f(y)$

(2) $\Rightarrow$ (1):
$x \leq y \Leftrightarrow x \wedge y = x \Rightarrow$
$f(x) = f(x \wedge y) \leq_{(2)} f(x) \wedge f(y) \leq f(y)$ □

Often the transfer functions are *distributive* (and therefore also monotone):

$f(x \bigwedge y) = f(x) \bigwedge f(y)$

Meet-Over-Paths solution: let us indicate with P_n the (possibly infinite) set of all the paths from the initial node to n; the Meet-Over-Paths (MOP) solution to the flow problem is defined as:

$\text{MOP}[n] = \bigwedge_{p \in P_n} f_p(\top)$

where f_p is obtained by composition of the transfer functions f_k of the nodes k occurring along the path p.

A solution to a flow analysis problem is *exact* if it collects flow information which has been propagated only along *feasible* paths (i.e., paths for which there exists at least one input that allows traversing them).

$\text{EXACT}[n] = \bigwedge_{p \in feas(P_n)} f_p(\top)$

The MOP solution is conservative (i.e., it accounts for any possibe program execution) since it gathers the flow information along *all* paths, instead of collecting the flow information just along the *feasible* paths, which are in general unknown.

Theorem: The MOP solution is lower than or equal to the exact one: $\text{MOP}[n] \leq \text{EXACT}[n]$.

Proof:
$\bigwedge_{p \in P_n} f_p(\top) = (\bigwedge_{p \in\ feas(P_n)} f_p(\top)) \bigwedge (\bigwedge_{p \in\ infeas(P_n)} f_p(\top)) \leq \text{EXACT}[n]$

Conservativity of the solution (*safety*): any solution lower than or equal to (according to the relationship $\leq$) the exact solution is *conservative* (aka *safe*).

It can be shown that the solution produced by the flow algorithm is in turn lower than or equal to the MOP solution, and therefore it is also safe (conservative). If the framework is distributive, it can be shown that the solution of the flow algorithm is coincident with the MOP solution.

Convergence: convergence of the flow algorithm needs be proved for each particular instance of the framework. Convergence is trivially assured if V is finite. If V is infinite but the partial order relationship $\leq$ is associated to a lattice with a finite number of *layers*, then convergence is surely granted.

Inizialization: some instances of the general flow analysis framework may require a specific initialization of the sets $\text{IN}[n]$, different from the element $\top$ indicated in the generic algorithm.

2.2 Reaching definitions

Reaching definition: a reaching definition holds between nodes n and m on variable x if n defines x and a path exists in the control flow graph from n to m along which x is not redefined:

$(n, m, x) \in RD \Leftrightarrow$
$(n, x) \in DEF$,
$\exists p =< n, \ldots, k, \ldots, m > in\ CFG : k \neq n \wedge k \neq m \Rightarrow (k, x) \notin DEF$

Reachable use (aka live variable): a reachable use holds between nodes n and m on variable x if m uses x and a path exists in the control flow graph from n to m along which x is not redefined:

$(n, m, x) \in RU \Leftrightarrow$
$(m, x) \in USE$,
$\exists p =< n, \ldots, k, \ldots, m > in\ CFG : k \neq n \wedge k \neq m \Rightarrow (k, x) \notin DEF$

Data dependence (aka def-use chain): a data dependence holds between nodes n and m on variable x if n defines x, m uses x and a path exists in the control flow graph from n to m along which x is not redefined:

$(n, m, x) \in DU \Leftrightarrow$
$(n, x) \in DEF$,
$(m, x) \in USE$,
$\exists p =< n, \ldots, k, \ldots, m > in\ CFG : k \neq n \wedge k \neq m \Rightarrow (k, x) \notin DEF$

Reaching definitions computation:

- V contains sets of pairs (n, x), where n is a CFG node that defines variable x.
- Transfer functions f_n have the following structure:
  $f_n(x) = GEN_n \bigcup (x \setminus KILL_n)$, where
  $GEN_n = \{(n, x) : (n, x) \in DEF\}$
  $KILL_n = \{(k, x) : (n, x) \in DEF, k\ is\ a\ CFG\ node\}$
- The confluence operator is the set union.
- The direction of propagation is forward.

Reaching definitions algorithm:

1 **for each** node n
2 IN[n] = $\emptyset$
3 OUT[n] = GEN[n] $\bigcup$ (IN[n] $\setminus$ KILL[n]) = GEN[n]
4 **end for**
5 **while** any OUT[n] changes
6 **for each** node n
7 IN[n] = $\bigcup_{p \in pred(n)}$ OUT[p]
8 OUT[n] = GEN[n] $\bigcup$ (IN[n] $\setminus$ KILL[n])
9 **end for**
10 **end while**

Example of reaching definitions:

```
main()
{
1   scanf("%d", &a);        {a:1}
2   if (a == 3)             {a:1}
3      x = a;               {a:1; x:3}
4   else if (a == 4)        {a:1}
5           a++;            {a:5}
6        else while (x)     {a:1; x:7}
7                x--;       {a:1; x:7}
8   printf("%d %d", a, x);  {a:1,5; x:3,7}
}
```

Reachable uses computation:

- V contains sets of pairs (n, x), where n is a CFG node that uses variable x.
- Transfer functions f_n have the following structure:
  $f_n(x) = GEN_n \bigcup (x \setminus KILL_n)$, where
  $GEN_n = \{(n, x) : (n, x) \in USE\}$
  $KILL_n = \{(k, x) : (n, x) \in DEF, k \ is \ a \ CFG \ node\}$
- The confluence operator is the set union.

- The direction of propagation is backward.

Reachable uses algorithm:

1 **for each** node n
2 IN$[n] = \emptyset$
3 OUT$[n]$ = GEN$[n] \bigcup$ (IN$[n] \setminus$ KILL$[n]$) = GEN$[n]$
4 **end for**
5 **while** any OUT$[n]$ changes
6 **for each** node n
7 IN$[n] = \bigcup_{p \in succ(n)}$ OUT$[p]$
8 OUT$[n]$ = GEN$[n] \bigcup$ (IN$[n] \setminus$ KILL$[n]$)
9 **end for**
10 **end while**

Example of reachable uses:

```
main()
{
1  while ((a--)&&(x++))  IN ={x:1,3,4; a:2}
                         OUT={x:1; a:1}
2      while (a++)       IN ={x:1,3; a:1,2}
                         OUT={x:1,3; a:2}
3          x--;          IN ={x:1,3; a:2}
                         OUT={x:3; a:2}
4  a = x + 1;            IN ={}
                         OUT={x:4}
}
```

Data dependences can be obtained as the restriction of the reaching definitions to the triples $(n, m, x) \in RD$ for which the additional condition that m uses x holds, or, alternatively, as the restriction of the reachable uses to the triples $(n, m, x) \in RU$ for which the additional condition that n defines x holds. It can be shown that the two results obtained in this way are coincident.

Example of data dependences:

```
main()
{
1    while ((a--)&&(x++))    {x:1,3,4; a:2}
2        while (a++)         {a:1,2}
3            x--;            {x:1,3}
4    a = x + 1;              {}
}
```

2.3 Dominators

Dominator: node n dominates node m if n is contained in every path of the CFG going from the initial node n_1 to m.

$(n, m) \in DOM \Leftrightarrow$
$\forall p =< n_1, \ldots, k, \ldots, m > in\ CFG : n \in p$

Postdominator: node n postdominates node m if n is contained in every path of the CFG going from node m to the final node n_N. Moreover, every node does not postdominate itself.

$(n, m) \in PDOM \Leftrightarrow$
$n \neq m$,
$\forall p =< m, \ldots, k, \ldots, n_N > in\ CFG : n \in p$

It can be shown that every node (different from n_1) has a unique immediate dominator (i.e., such that it is dominated by all other dominators of the node). This allows representing the dominance relationship by means of a tree called the *dominator tree*, whose root is n_1.

Simmetrically, it can be shown that every node (different from n_N) has a unique immediate postdominator (i.e., such that it is postdominated by all other postdominators of the node). This allows representing the postdominance relationship by means of a tree called the *postdominator tree*, whose root is n_N.

Dominators computation:

- V contains sets of nodes of the CFG.
- Transfer functions f_n have the following structure:
  $f_n(x) = GEN_n \bigcup (x \setminus KILL_n)$, where
  $GEN_n = \{n\}$
  $KILL_n = \emptyset$
- The confluence operator is the set intersection.
- The direction of propagation is forward.
- Initial values are:
  $IN_n = N$ (all CFG nodes), if $n \neq n_1$
  $IN_{n_1} = \emptyset$

Dominators algorithm:

```
1   IN[n_1] = ∅
2   OUT[n_1] = GEN[n] ⋃ IN[n] = {n_1}
3   for each node n ≠ n_1
4       IN[n] = N
5       OUT[n] = GEN[n] ⋃ IN[n] = IN[n]
6   end for
7   while any OUT[n] changes
8       for each node n
9           IN[n] = ⋂_{p∈pred(n)} OUT[p]
10          OUT[n] = GEN[n] ⋃ IN[n] = {n} ⋃ IN[n]
11      end for
12  end while
```

Example of dominators:

```
main()
{
1   x = 3;                  {1}
2   if (x)                  {1, 2}
3      goto a;              {1, 2, 3}
4   y = x + 1;              {1, 2, 4}
5   while (y)               {1, 2, 4, 5}
6      y--;                 {1, 2, 4, 5, 6}
7   a:                      {1, 2, 7}
}
```

Postdominators computation:

- V contains sets of nodes of the CFG.
- Transfer functions f_n have the following structure:
  $f_n(x) = GEN_n \bigcup (x \setminus KILL_n)$, where
  $GEN_n = \{n\}$
  $KILL_n = \emptyset$
- The confluence operator is the set intersection.
- The direction of propagation is backward.
- Initial values are:
  $IN_n = N$ (all CFG nodes), if $n \neq n_N$
  $IN_{n_N} = \emptyset$

Postdominators algorithm:

1 $\text{IN}[n_N] = \emptyset$
2 $\text{OUT}[n_N] = \text{GEN}[n] \bigcup \text{IN}[n] = \{n_N\}$
3 **for each** node $n \neq n_N$
4 $\quad \text{IN}[n] = N$
5 $\quad \text{OUT}[n] = \text{GEN}[n] \bigcup \text{IN}[n] = \text{IN}[n]$
6 **end for**
7 **while** any $\text{OUT}[n]$ changes
8 $\quad$ **for each** node n
9 $\quad\quad \text{IN}[n] = \bigcap_{p \in succ(n)} \text{OUT}[p]$
10 $\quad\quad \text{OUT}[n] = \text{GEN}[n] \bigcup \text{IN}[n] = \{n\} \bigcup \text{IN}[n]$
11 $\quad$ **end for**
12 **end while**

Example of postdominators:

```
main()
{
1   x = 3;                  {2, 7}
2   if (x)                  {7}
3      goto a;              {7}
4   y = x + 1;              {5, 7}
5   while (y)               {7}
6      y--;                 {5, 7}
7   a:                      {}
}
```

2.4 Constant propagation

The purpose of this analysis is determining if the value of a variable at a given statement is undefined, constant or non constant.

Constant propagation:

- V contains sets of pairs (x, v), where x is a variable, while v is a constant, *undef* or *nonconst*. Such sets of pairs define a function, $val(x) = v$, associating each variable to the value it assumes.
- Transfer functions f_n have the following structure:
  $f_n(x) = GEN_n \bigcup (x \setminus KILL_n)$
  $KILL_n = \{(x, v) : v \text{ is any value, } x \text{ is in } GEN_n\}$
  The GEN_n set depends on the kind of statement n. For example:

  n : `x = 3;` $\Rightarrow$
  $GEN_n = \{(x, 3)\}$

  n : `x = y + 3;` $\Rightarrow$
  $GEN_n = \{(x, c + 3)\}$ if $val(y) = c$ ($\neq$ *undef/nonconst*)
  $GEN_n = \{(x, undef/nonconst)\}$ if $val(y)$ is *undef* or *nonconst*

  n : `scanf("%d", &x);` $\Rightarrow$
  $GEN_n = \{(x, nonconst)\}$
- The confluence operator is described by the following table:

	nonconst	c	$d \neq c$	*undef*
nonconst	*nonconst*	*nonconst*	*nonconst*	*nonconst*
c	*nonconst*	c	*nonconst*	c
undef	*nonconst*	c	d	*undef*

- The direction of propagation is forward.

2.5 Control dependences

Control dependence: node n holds a control dependence on node m if a path exists in the CFG from n to m, whose intermediate nodes are post-dominated by m, while n is not postdominated by m.

$(n, m) \in CD \Leftrightarrow$
$(m, n) \notin PDOM$,
$\exists p =< n, \ldots, k, \ldots, m > in\ CFG:$
$\forall k \in p, k \neq n, k \neq m \Rightarrow (m, k) \in PDOM$

Control dependences algorithm:

```
1   CD ← ∅
2   for each CFG edge (n, m)
3       if (m, n) ∉ PDOM
4           L ← parent(PDOM-TREE, n)
5           k ← m
6           repeat
7               CD ← CD ⋃{(n, k)}
8               k ← parent(PDOM-TREE, k)
9           until k = L
10      end if
11  end for
```

Example of control dependences:

```
main()
{
1   if (a == c) {           {2, 3, 5}
2      d--;                 {}
3      while (a < d)        {3, 4}
4         a++; }            {}
5   else while (a > d) {    {5, 6, 7}
6      a--;                 {}
7      c++; }               {}
8   printf("%d", a);        {}
}
```

2.6 Pointer analysis

2.6.1 Flow insensitive pointer analysis

This pointer analysis technique will be described with reference to a reduced imperative language capturing all relevant properties of languages, such as C, which allow for pointer manipulation.

Syntax of the language:

$$
\begin{array}{rcl}
S & ::= & \texttt{x = y;} \\
 & | & \texttt{x = \&y;} \\
 & | & \texttt{x = *y;} \\
 & | & \texttt{x = y + z;} \\
 & | & \texttt{x = alloc(y);} \\
 & | & \texttt{*x = y;} \\
 & | & \texttt{f(p1, ..., pn):r \{ } S^* \texttt{ \}} \\
 & | & \texttt{x = f(a1, ..., an);}
\end{array}
$$

Control flow statements are not present in that all flow insensitive analyses ignore them. Parameters are passed to functions by value.

Types: a non standard (i.e., different from `int, double` etc.) set of types is used to represent the relationship *points-to*:

$$\tau_i \quad ::= \quad \bot_i \mid \mathbf{ref}(\tau_j)$$

where the index of τ and $\bot$ univoquely identifies the type (when unnecessary, it will be omitted for $\bot$). Two types are equal if and only if they are both $\bot$ (τ) and they have the same index: $\tau_i = \tau_j (\bot_i = \bot_j) \Leftrightarrow i = j$. Consequently, $\tau_1 = \mathbf{ref}(\bot)$ and $\tau_2 = \mathbf{ref}(\bot)$ are considered different types, although they have the same structure.

Order relationship on types: given a statement assigning a value to a variable, the right hand side can either be a pointer to a location or it can contain a value. In the first case, the left hand side will point to the same location, and thus should reference the same type as the right hand side, while in the second case the equality of the referenced types is not required. This is expressed by the order relationship $\leq$ between types: the type τ_j,

referenced by the right hand side, will be required to be less than or equal to the type τ_i referenced by the left hand side, according to the following definition of the order relationship:

$$\tau_j \leq \tau_i \Leftrightarrow (\tau_j = \bot) \vee (\tau_j = \tau_i)$$

Typing rules:

$$\frac{\begin{array}{c} A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_i) \\ A \vdash \texttt{y}\text{: } \mathbf{ref}(\tau_j) \\ \tau_j \leq \tau_i \end{array}}{A \vdash \mathit{welltyped}(\texttt{x = y;})}$$

$$\frac{\begin{array}{c} A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_i) \\ A \vdash \texttt{y}\text{: } \tau_i \end{array}}{A \vdash \mathit{welltyped}(\texttt{x = \&y;})}$$

$$\frac{\begin{array}{c} A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_i) \\ A \vdash \texttt{y}\text{: } \mathbf{ref}(\tau_j) \\ A \vdash \tau_j = \mathbf{ref}(\tau_k) \\ \tau_k \leq \tau_i \end{array}}{A \vdash \mathit{welltyped}(\texttt{x = *y;})}$$

$$\frac{\begin{array}{c} A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_i) \\ A \vdash \texttt{y}\text{: } \mathbf{ref}(\tau_j) \\ A \vdash \texttt{z}\text{: } \mathbf{ref}(\tau_k) \\ \tau_j \leq \tau_i \\ \tau_k \leq \tau_i \end{array}}{A \vdash \mathit{welltyped}(\texttt{x = y + z;})}$$

$$\frac{\begin{array}{c} A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_i) \\ A \vdash \tau_i = \mathbf{ref}(\tau_j) \end{array}}{A \vdash \mathit{welltyped}(\texttt{x = alloc(y);})}$$

$$\frac{\begin{array}{c} A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_i) \\ A \vdash \tau_i = \mathbf{ref}(\tau_k) \\ A \vdash \texttt{y}\text{: } \mathbf{ref}(\tau_j) \\ \tau_j \leq \tau_k \end{array}}{A \vdash \mathit{welltyped}(\texttt{*x = y;})}$$

$$\frac{\begin{array}{c} \texttt{f(p1, ..., pn):r} \\ A \vdash \texttt{p1}\text{: } \mathbf{ref}(\tau_{p1}), \ldots, \texttt{pn}\text{: } \mathbf{ref}(\tau_{pn}) \\ A \vdash \texttt{r}\text{: } \mathbf{ref}(\tau_r) \\ A \vdash \texttt{a1}\text{: } \mathbf{ref}(\tau_{a1}), \ldots, \texttt{an}\text{: } \mathbf{ref}(\tau_{an}) \\ A \vdash \texttt{x}\text{: } \mathbf{ref}(\tau_x) \\ \tau_{a1} \leq \tau_{p1}, \ldots, \tau_{an} \leq \tau_{pn} \\ \tau_r \leq \tau_x \end{array}}{A \vdash \mathit{welltyped}(\texttt{x = f(a1, ..., an);})}$$

Pointer analysis algorithm:

1 $i \leftarrow 1$
2 **for each** program variable x
3 $\quad x : \tau_i = \textbf{ref}(\bot_i)$
4 $\quad i \leftarrow i + 1$
5 **end for**
6 **for each** program statement S
7 $\quad$ **apply** the inference rule for S
8 **end for**

Inference rules:

`x = y;`
let x: **ref**(τ_i)
let y: **ref**(τ_j)
if $\tau_i \neq \tau_j$
$\quad$ **cjoin**(τ_i, τ_j)
end if

`x = &y;`
let x: **ref**(τ_i)
let y: τ_j
if $\tau_i \neq \tau_j$
$\quad$ **join**(τ_i, τ_j)
end if

`x = *y;`
let x: **ref**(τ_i)
let y: **ref**(τ_j)
if $\tau_j = \bot$
$\quad \tau_j \leftarrow$ **ref**($\bot_j$)
end if
let τ_j = **ref**(τ_k)
if $\tau_i \neq \tau_k$
$\quad$ **cjoin**(τ_i, τ_k)
end if

`x = y + z;`
let x: **ref**(τ_i)
let y: **ref**(τ_j)
let z: **ref**(τ_k)
if $\tau_i \neq \tau_j$
$\quad$ **cjoin**(τ_i, τ_j)
end if
if $\tau_i \neq \tau_k$
$\quad$ **cjoin**(τ_i, τ_k)
end if

`x = alloc(y);`

let x: **ref**(τ_i)
if $\tau_i = \bot$
    $\tau_i \leftarrow$ **ref**($\bot$)
end if

`*x = y;`

let x: **ref**(τ_i)
let y: **ref**(τ_j)
if $\tau_i = \bot$
    $\tau_i \leftarrow$ **ref**($\bot_i$)
end if
let τ_i = **ref**(τ_k)
if $\tau_k \neq \tau_j$
    cjoin(τ_k, τ_j)
end if

`x = f(a1, ..., an);`

let `f(p1, ..., pn):r`
let `p1`: **ref**(τ_{p1}), ..., `pn`: **ref**(τ_{pn})
let `r`: **ref**(τ_r)
let `a1`: **ref**(τ_{a1}), ..., `an`: **ref**(τ_{an})
let `x`: **ref**(τ_x)
if $\tau_x \neq \tau_r$
    cjoin(τ_x, τ_r)
end if
for $i = 1, ..., n$
    if $\tau_{pi} \neq \tau_{ai}$
        cjoin(τ_{pi}, τ_{ai})
    end if
end for

Unification algorithms:

cjoin(τ_1, τ_2)
1 **if** $\tau_2 = \bot_2$
2 pending[$\bot_2$] $\leftarrow$ pending[$\bot_2$] $\bigcup\{\tau_1\}$
3 **else**
4 **join**(τ_1, τ_2)
5 **end if**

join(τ_1, τ_2)
1 $\tau_3 \leftarrow$ **unify**(τ_1, τ_2)
2 **if** $\tau_1 = \bot_1 \wedge \tau_2 = \bot_2$
3 let $\bot_3 = \tau_3$
4 pending$[\bot_3] \leftarrow$ pending$[\bot_1] \bigcup$ pending$[\bot_2]$
5 **end if**
6 **if** $\tau_1 = \bot_1 \wedge \tau_2 \neq \bot$
7 **for each** τ_i in pending$(\bot_1)$
8 **join**(τ_3, τ_i)
9 **end for**
10 **end if**
6 **if** $\tau_1 \neq \bot \wedge \tau_2 = \bot_2$
7 **for each** τ_i in pending$(\bot_2)$
8 **join**(τ_3, τ_i)
9 **end for**
10 **end if**

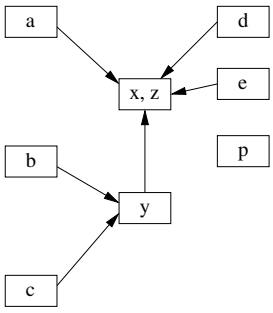
unify(τ_1, τ_2)
1 **if** $\tau_1 = \bot_1 \wedge \tau_2 = \bot_2$
2 **return** new $\bot(\bot_1, \bot_2)$
3 **else**
4 **return** new $\tau(\tau_1, \tau_2)$
5 **end if**

Computational complexity: the algorithm examines each program statement just once. The application of the inference rules to a statement may trigger the execution of one or more **join** operations: the algorithm may recurse if non empty pending lists are present. However, the types that are unified during the recursion cannot appear any longer during successive recursions, and therefore the total number of **join** operations that are executed is at most $O(N)$ (where N is the program size). Therefore, on average each statement triggers 1 **join**, the computational cost of which is $O(\alpha(N, N))$ if fast union/find data structures are used for the types, where α is the inverse Ackermann's function (generalized logarithm). The resulting computational complexity is equal to $O(N\alpha(N, N))$.

Example of well typed program:

```
main()
{
1   d = a;
2   e = a;
3   a = &x;
4   b = &y;
5   if (p)
6      y = &z;
7   else
8      y = &x;
9   c = &y;
}
```

a: $\tau_1 = \mathbf{ref}(\tau_4)$
b: $\tau_2 = \mathbf{ref}(\tau_5)$
c: $\tau_3 = \mathbf{ref}(\tau_5)$
x: $\tau_4 = \mathbf{ref}(\bot)$
y: $\tau_5 = \mathbf{ref}(\tau_4)$
z: $\tau_4 = \mathbf{ref}(\bot)$
p: $\tau_6 = \mathbf{ref}(\bot)$
d: $\tau_7 = \mathbf{ref}(\tau_4)$
e: $\tau_8 = \mathbf{ref}(\tau_4)$



Storage Shape Graph (SSG): each node of this graph is associated with a type (different from $\bot$) and is labelled with the list of variables of that type. Edges represent the **ref** relationship between types. If a program is well typed, its SSG is a conservative representation of the *points-to* relationship between memory locations of the program for any execution with any input.

2.6.2 Flow sensitive pointer analysis

Alias: two or more different names are said to be *aliases* of each other if they identify the same program location (e.g., `*p` and `x`, if `p = &x`). Aliases are represented by unordered pairs: $\langle *p, x \rangle$.

Flow sensitivity:

```
1    p =&x;      {<*p, x>}
2    q = p;      {<*p, x>, <*q, x>, <*p, *q>}
3    p = &y;     {<*p, y>, <*q, x>}
4    r = p;      {<*p, y>, <*q, x>, <*r, *p>, <*r, y>}
```

Conditional alias: an alias is said conditional if its validity depends on the validity of another alias (condition). Conditional aliases are represented as a pair of aliases, the former of which is the condition. For example, $(\langle *p, x \rangle, \langle *q, x \rangle)$ after the statement `q = p` represents the fact that `*q` is an alias of `x` only under the condition that `*p` be an alias of `x`. This may for example occur when `p` is the formal parameter of a procedure (`f(int* p);`). If such a procedure is invoked with `&x` as the actual parameter (`f(&x);`), the condition is satisfied and the alias $\langle *q, x \rangle$ is valid. Otherwise, if the calling context of `f` is different (e.g., `f(&y);`), the condition is no longer true and the alias $\langle *q, x \rangle$ becomes invalid.

Context sensitivity:

```
int* q;
1    f(&x);
...
2    f(&y);

f(int* p) {
3    q = p;      {(0, <*p, *q>),
                  (<*p, x>, <*p, x>), (<*p, x>, <*q, x>),
                  (<*p, y>, <*p, y>), (<*p, y>, <*q, y>)}
}
```

1: BIND$[\emptyset] = \{\langle *p, x \rangle\}$
2: BIND$[\emptyset] = \{\langle *p, y \rangle\}$

Examples of pointer analysis:

```
main() {
1   f(&a);
2   f(&b);
}

f(int** y) {
3   *y = &c;
}
```

1: BIND$[\emptyset] = \{\langle *y, a\rangle\}$
2: BIND$[\emptyset] = \{\langle *y, b\rangle\}$

Aliases at exit of node 2: $(\emptyset, \langle *a, c\rangle)$, $(\emptyset, \langle *b, c\rangle)$.

```
main() {
1   f(&a);
2   f(&b);
}

f(int** y) {
3   g(y);
}

g(int** z) {
4   *z = &c;
}
```

1: BIND$[\emptyset] = \{\langle *y, a\rangle\}$
2: BIND$[\emptyset] = \{\langle *y, b\rangle\}$
3: BIND$[\langle *y, a\rangle] = \{\langle *z, a\rangle\}$
3: BIND$[\langle *y, b\rangle] = \{\langle *z, b\rangle\}$
3: BIND$[\emptyset] = \{\langle *y, *z\rangle\}$

Aliases at exit of node 2: $(\emptyset, \langle *a, c\rangle)$, $(\emptyset, \langle *b, c\rangle)$.

```
main() {
1   s = &a;
2   t = &s;
3   f(t);
4   exit(0);
}

f(int** w) {
5   *w = &b;
6   return;
}
```

3: BIND$[\emptyset] = \{\langle *t, *w\rangle, \langle s, *w\rangle, \langle * * w, a\rangle\}$
5: $(\langle *w, *t\rangle, \langle *w, *t\rangle), (\emptyset, \langle **w, b\rangle), (\langle *w, *t\rangle, \langle **t, b\rangle), (\langle s, *w\rangle, \langle s, *w\rangle), (\langle s, *w\rangle, \langle *s, b\rangle)$

Aliases at exit of node 4: $(\emptyset, \langle *t, s\rangle)$, $(\emptyset, \langle *s, b\rangle)$, $(\emptyset, \langle * * t, b\rangle)$.

2.7 Interprocedural flow analysis

Valid (aka realizable) interprocedural path: Let us assume for convenience that all nodes which are not a procedure call or return are eliminated from the path. Moreover, each pair of call and return nodes associated to a given call site are assigned a same incremental index (e.g., c_i, r_i).

1. An empty path is valid.
2. Given a path containing a non empty sequence of call and return nodes, if the first return node is preceded immediately by a call node with same index, invoking the same procedure, and the remaining part of the path (obtained after deleting these two nodes) is valid, then the whole path is valid.

Note: the set of all the paths in a program can be generated by a regular grammar, while the set of all the valid paths can only be generated by a context free grammar. Example: `f() { if (c) f(); return; }`. Paths: $(e_2e_4) * e_1(e_3e_5)*$. Valid paths: $F ::= e_1|e_2e_4Fe_3e_5$.

2.7.1 Call string approach

The general idea consists of labelling the flow information with a string which encodes the history of the procedure calls that led to propagate such flow information up to a given program point.

Call string: given a path $p =< q_1, c_1, e_1, q_2, ..., c_j, e_j, q_{j+1} >$, where $q_1, ..., q_{j+1}$ are interprocedural valid paths and each call node $c_1, ..., c_j$ is followed by the initial node $e_1, ..., e_j$ of the called procedure, the call string associated with p is:

$\gamma =$ "$c_1...c_j$"

The flow information x to be propagated is labelled with γ: (x, γ). The confluence operator is redefined in the following way:

$(x_1, \gamma_1) \bigwedge (x_2, \gamma_2) = (x_1 \bigwedge x_2, \gamma)$ if $\gamma_1 = \gamma_2 = \gamma$.

In all other cases the confluence is undefined.

The call strings are updated during flow propagation. In particular, when propagating from a call node c to the initial node of the called procedure:

$\gamma' = \gamma + "c"$

When propagating from the exit node of a procedure to the call node c:

$\gamma' = \gamma -$ LAST$[\gamma]$ if LAST$[\gamma] = c$ (otherwise γ' is undefined).

When γ' is undefined, the flow information labelled by γ is NOT propagated to the call node (invalid interprocedural path).

Problem: in presence of recursion the call strings have infinite length, thus making this approach apparently infeasible.

Solution: let $M = K(|U| + 1)^2$, where K is the number of call nodes in the program and $|U|$ is the size of the set containing every possible flow information propagated in the CFG. The maximum allowed length for a call string γ is M. If the concatenation $\gamma + "c"$ produces a string whose length exceeds M, the resulting string γ' is considered undefined.

It can be shown that the solution obtained by flow analysis with this set of call strings is conservative, and it is coincident with the interprocedural MOP solution (restricted to valid paths) if the framework is distributive.

The value of M can be further decreased for specific cases. In general, it is always possible replacing K with K', the maximum number of distinct nested calls in every possible interprocedural path (maximum call stack size without recursion). The framework for the reaching definitions admits a further reduction, since it can be decomposed as follows: every node n which defines a variable x can be associated to a sub-framework in which 0 is propagated if x was defined by nodes different from n and 1 if x was defined by n. The overall framework can be obtained by just recombining the sub-frameworks. For each sub-framework $U = \{0, 1\}$ and consequently it is sufficient to set $M = 9K$ (or, even better, $M = 9K'$).

Example of interprocedural reaching definitions:

```
main() {
1   a = 0;        ({a: 1}, "")
2   f();          ({a: 1, 7}, "")
3   a = 1;        ({a: 3}, "")
4   f();          ({a: 3, 7}, "")
5   exit(0);      ({a: 3, 7}, "")
}

f() {
6   if (c)        ({a: 1}, "2") ({a: 3}, "4")
7      a++;       ({a: 7}, "2") ({a: 7}, "4")
8   return;       ({a: 1, 7}, "2") ({a: 3, 7}, "4")
}
```

2.7.2 Functional approach

The effect of a procedure q on the flow information is summarized by a composite transfer function ϕ_q, given by the function ϕ_{e_N} computed at the exit statement of the procedure q by means of iterated propagation until fixpoint. Given the transfer functions $f_n(x)$, the composite functions ϕ_n can be obtained as the minimum fixpoint of the following equations:

- $\phi_{e_1}(x) = f_{e_1}(x)$
- $\phi_n(x) = f_n(\bigwedge_{p \in pred(n)} \phi_p(x))$ if $n \neq e_1$ is not a call node.
- $\phi_n(x) = \phi_q(\bigwedge_{p \in pred(n)} \phi_p(x))$ if $n \neq e_1$ is a call node and q is the called procedure.

If the flow analysis framework is distributive, the following relation holds: $f_n(\bigwedge_{p \in pred(n)} \phi_p(x)) = \bigwedge_{p \in pred(n)} f_n(\phi_p(x))$

Let us consider the flow analysis frameworks whose transfer functions f_n have the following structure:

$f_n(x) = \text{GEN}[n] \bigcup (x \setminus \text{KILL}[n])$

and the meet operator is the union. The equations for the computation of the composite functions become:

- $\phi_{e_1}(x) = \text{GEN}[e_1] \bigcup (x \setminus \text{KILL}[e_1])$
- $\phi_n(x) = \text{GEN}[\phi_n] \bigcup (x \setminus \text{KILL}[\phi_n])$ if $n \neq e_1$ is not a call node, where:
  - $\text{GEN}[\phi_n] = (\bigcup_{p \in pred(n)} \text{GEN}[\phi_p] \setminus \text{KILL}[n]) \bigcup \text{GEN}[n]$
  - $\text{KILL}[\phi_n] = \bigcap_{p \in pred(n)} \text{KILL}[\phi_p] \bigcup \text{KILL}[n]$
- $\phi_n(x) = \phi_q(\bigcup_{p \in pred(n)} \phi_p(x))$ if $n \neq e_1$ is a call node and q is the called procedure.

Example of interprocedural reaching definitions:

```
main() {
1   a = 0;        {a: 1}
2   f();          {a: 1, 7}
3   a = 1;        {a: 3}
4   f();          {a: 3, 7}
5   exit(0);      {a: 3, 7}
}

f() {
6   if (c)        {a: 1, 3}
7      a++;       {a: 7}
8   return;       {a: 1, 3, 7}
}
```

$\phi_f(x) = x \bigcup \{\texttt{a:7}\}$

The iterative computation of the functions $\phi_q(x)$ may be infeasible for flow analysis frameworks which do not admit a compact representation of the transfer functions and of the meet operator. In such cases, the functions $\phi_q(x)$ can be computed just on the values $\overline{x}$ which can actually reach the initial node of q. Call nodes will be able to use the composite functions $\phi_q(x)$ only if their computation on $\overline{x}$ was already performed. Otherwise, it will be necessary to wait until a successive iteration, when $\overline{x}$ has been propagated from the entry to the exit of q, making thus $\phi_q(x)$ available for the specific value $\overline{x}$.

In the previous example, the values of $\phi_f(x)$ to be determined are:
$\phi_f : \{\texttt{a} : 1\} \mapsto \{\texttt{a} : 1, 7\}, \{\texttt{a} : 3\} \mapsto \{\texttt{a} : 3, 7\}$

2.8 Abstract interpretation

The aim of abstract interpretation is assigning an abstract value to each program variable at each program statement. Abstract values group concrete values into equivalence classes. The outcome of abstract interpretation holds for every possible program execution (conservative results).

First, syntax and semantics of programs will be described at a high level, with no reference to a specific programming language. Then, the static semantics of a program will be introduced. Finally, a generalization of the static semantics, called abstract interpretation, will be presented.

2.8.1 Abstract syntax

Abstractly, a program consists of a set of *Nodes* and *Arcs*. Nodes can be partitioned into:

- Entries
- Assignments
- Tests
- Junctions
- Exits

Entries: An *entry* node n has no predecessor ($|n\_pred(n)|=0$) and one successor ($|n\_succ(n)|=1$).

Assignments: An *assignment* node n has one predecessor ($|n\_pred(n)|=1$) and one successor ($|n\_succ(n)|=1$). *Expr(n)* gives the right hand side expression, *id(n)* gives the left hand side identifier.

Tests: A *test* node n has one predecessor ($|n\_pred(n)|=1$) and two successors ($|n\_succ(n)|=2$). The *true* and *false* successors are respectively denoted *n_succ_t(n)* and *n_succ_f(n)*. *Test(n)* gives the boolean expression associated with the test node n.

Junctions: A *junction* node n has more than one predecessor ($|n\_pred(n)| > 1$) and one successor ($|n\_succ(n)| = 1$).

Exits: An *exit* node n has one predecessor ($|n\_pred(n)| = 1$) and no successor ($|n\_succ(n)| = 0$).

The set of Arcs can be defined as:

$$Arcs = \{< n, m > | (n \in Nodes) \wedge (m \in n\_succ(n))\}$$

Origin(a) and *end(a)* give n and m for an arc $a =< n, m >$.

a_succ(n), *a_pred(n)*, *a_succ_t(n)*, *a_succ_f(n)* give successor and predecessor arcs for a node n (the latter, true and false branches).

2.8.2 Program semantics

Let *Values* be the set of all possible variable values.

An environment e maps identifiers to their values:

$$e : Env = Ident \rightarrow Values$$

The meaning of an expression *expr* in the environment e is given by $val[[expr]](e)$, where:

$$val : Expr \rightarrow [Env \rightarrow Values]$$

A state $s =< cs(s), env(s) >$ consists of a control edge $cs(s)$ and an environment $env(s)$.

A value/variable substitution in the environment e is:

$$e[v/x] = \lambda y.cond(y = x, v, e(y))$$

State transition is described by means of the function *n_state(s)*, where s is a state:

n_state(s: State): State
1 **let** n be end(cs(s)), e be env(s)
2 **case** n in
3 Assignments: **return** <a_succ(n), e[val[[expr(n)]](e)/id(n)]>
4 Tests: **return**
5 cond(val[[test(n)]](e), <a_succ_t(n), e>, <a_succ_f(n), e>)
6 Junctions: **return** <a_succ(n), e>
7 Exits: **return** s
8 **end case**

2.8.3 Static semantics

The static semantics determines the set of all environments e which may be associated to a program edge q in any possible computation. Such a set is called a *context* $C(q)$:

$$C(q) = \{e \in Env | (\exists n \geq 0, \exists i_s \in InitStates | < q, e >= n\_state^n(i_s))\}$$

Computation of a new context for an arc r, given an existing context C, can be achieved by means of the function *n_context(r, C)*:

n_context(r: Arc, C: Context): Context
1 **case** origin(r) in
2 Entries: **return** {InitEnv}
3 Tests, Assignments, Junctions: **return**
4 $\bigcup_{q \in} a\_pred(origin(r)) \bigcup_{e \in C(q)} env(n\_state(< q, e >))$
5 **end case**

The static semantics (context) of a program is given by the fixpoint of the function *n_context(r, C)* over all arcs, that is, the value of C such that, for all arcs r, the following condition holds:

$$C(r) = n\_context(r, C)$$

2.8.4 Abstract interpretation

In an *abstract interpretation* of a program, actual values are replaced by *abstract values*, which belong to a so called *abstract context*. The main requirement on an abstract context is that it must be a complete semi-lattice (with ordering "$\leq$").

An abstract interpretation is a function *Int*, such that:

$$Int : Arcs \times AbsContext \rightarrow AbsContext$$

under the constraint that it is order-preserving:

$$\forall r \in Arcs, \forall (C', C'') \in AbsContext \times AbsContext,$$
$$C' \leq C'' \Rightarrow Int(r, C') \leq Int(r, C'')$$

The abstract interpretation of a program determines the abstract context associated to each program edge r, such that $C(r) = Int(r, C)$, i.e., the fixpoint of the abstract interpretation function *Int*.

The static semantics of a program is a special case of abstract interpretation with:

AbsContext = 2^{Env}
Int = n_context

References

(Aho 1985) A. V. Aho, R. Sethi and J. D. Ullman. *Compilers. Principles, Techniques, and Tools.* Chapter 10, Addison Wesley, Reading, MA, 1985.

(Cousot 1977) P. Cousot and R. Cousot. *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* Proc. of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977.

(Ferrante 1987) J. Ferrante, K. J. Ottenstein and J. D. Warren. *The Program Dependence Graph and its use in Optimization.* In ACM Transactions on Programming Languages and Systems, vol. 3, n. 9, pp. 319-349, July 1987.

(Steensgaard 1996) B. Steensgaard. *Points-to Analysis in Almost Linear Time.* In Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32-41, January 1996.

(Landi 1992) W. Landi and B. G. Ryder. *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing.* In Proc. of the ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation, pp. 235-248, 1992.

(Sharir 1981) M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis.* In Program Flow Analysis: Theory and Applications, pp. 189-233, Prentice Hall, 1981.

Chapter 3

Program slicing

3.1 Static slicing

Program slicing is a decomposition technique aimed at determining the subset of statements relevant to a computation of interest. It provides the program statements which directly or indirectly contribute to the value of a given variable x at a given statement n.

Applications of program slicing:

- Debugging.
- Testing (regression).
- Parallelization.
- Integration (differencing).
- Safety (functional diversity).
- Understanding (decomposition, interface slicing).
- Maintenance.
- Metrics (cohesion).

Definition:
A **static slice** $S(x, n)$ of program P on variable x at statement n is a subprogram of P computing the same value of x at n for every input on which P terminates normally.

The pair (x, n) is called *slicing criterion*. The slice can be obtained by deleting zero or more statements from P which are assured not to affect the value of x at n.

In some applications of program slicing, the requirement that the resulting subprogram be still executable may be dropped (e.g., debugging).

Given a program P, a static slice $S(x, n)$ can be computed as the transitive closure of the data dependences of P (DU) at n on x, and of the control dependences of P (CD) on n.

$S(x, n) \leftarrow \{n' : (n', n, x) \in DU\}$
while $S(x, n)$ increases
$\quad S(x, n) \leftarrow S(x, n) \bigcup \{n'' : \exists n' \in S(x, n), \exists y \in V : (n'', n', y) \in DU$
$\qquad\qquad\qquad\qquad \vee\ \exists n' \in S(x, n) : (n'', n') \in CD\}$
end while

Slicing example:

```
main() {
1   inword = NO;
2   nl = 0;
3   nw = 0;
4   nc = 0;
5   c = getchar();
6   while (c != EOF) {
7     nc = nc + 1;
8     if (c == '\n')
9        nl = nl + 1;
10    if (c == ' ' || c == '\n' || c == '\t')
11       inword = NO;
12    else if (inword == NO) {
13       inword = YES;
```

```
14        nw = nw + 1;
       }
15     c = getchar();
     }
16   printf("%d \n", nl);
17   printf("%d \n", nw);
18   printf("%d \n", nc);
}
```

S(`nl`, 16) = {2, 5, 6, 8, 9, 15}
S(`nw`, 17) = {1, 3, 5, 6, 10, 11, 12, 13, 14, 15}
S(`nc`, 18) = {4, 5, 6, 7, 15}

Program Dependence Graph (PDG): graph the nodes of which are program statements and the edges of which are either control or data dependences between statements. Data dependence edges can be labelled with the variable on which the dependence holds, while control dependence edges can be labelled with the truth value which determines the execution of the target node. A fictitious *entry* node is connected via (true) control dependences to each node that is not controlled by any other different node. Slice computation on the PDG is straightforward:

$S(x, n) = \{m \in P | \exists p =< m, ..., n > \text{in PDG } \}$

with USE(n) = $\{x\}$ (this can always be achieved by some simple transformations of the original program).

Slicing unstructured programs:
The problem is determining which `goto` statements have to be included in the slice. This can be achieved through the following steps:

1. For each label a associated with a statement n, add a pseudo-label a to all statements that are always executed after n (basic block statements), until a different label is encountered.

2. When a statement m, whose (pseudo-)label is a, is added to the slice $S(x, n)$, add all the statements of the type `goto a` to $S(x, n)$. In case the statement labelled with a is not eventually in $S(x, n)$, a null statement labelled with a has to be inserted so as to have the label available in the right place.

3.1.1 Interprocedural slicing

System Dependence Graph (SDG): a system dependence graph consists of the program dependence graphs associated with all procedures in the system, the call dependences between call nodes and called procedures, the parameter-in (parameter-out) dependences between actual and formal (formal and actual) parameters passed to (returned from) each procedure at the call site and transitive dependences (called *summary edges*).

A *summary edge* connects an actual-in to an actual-out parameter node if the latter is reachable from the former in the called procedure along data and transitive dependences.

Interprocedural slicing algorithm:

Step 1 Determine the transitive closure over data, control, call, parameter-in and transitive dependences (but not parameter-out dependences).

Step 2 Determine the transitive closure over data, control, parameter-out and transitive dependences (but not parameter-in and call dependences).

Intuitively, Step 1 focuses on calling procedures while Step 2 focuses on called procedures.

Interprocedural slicing example:

```
main() {
1   int s = 0;
2   int i = 1;
3   while (i < 11)
4      f(s, i);
5   printf("%d\n", s);
}

f(int& x, int& y) {
6   add(x, y);
7   inc(y);
```

```
}

add(int& a, int b) {
8   a = a + b;
}

inc(int& z) {
9   add(z, 1);
}
```

S(z, 9[out]) = {2, 3, 4, 7, 8, 9}

Interprocedural slicing in presence of aliasing:

The problem is that a procedure behaves differently when some of its parameters are aliased. A solution to this problem consists of creating a copy of a procedure for each possible aliasing configuration under which it can be called and treat the copies as different procedures.

3.2 Decomposition slicing

Decomposition slice: let OutTerm(x) be the set of statements that output variable x or terminate the computation performed on x. The decomposition slice on x, denoted as $S(x)$, is defined as:

$S(x) = \bigcup_{n \in} {}_{OutTerm(x)} S(x, n)$

The decomposition slice on a variable captures all the computation performed by the program on that variable.

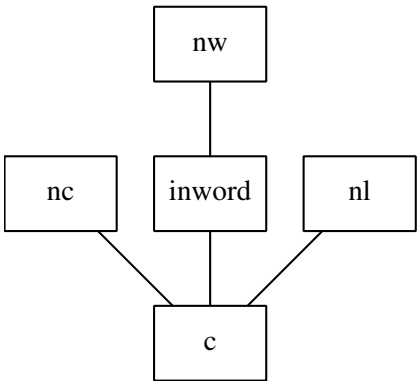
Dependence relationship for decomposition slices: two decomposition slices $S(x)$ and $S(y)$ are *independent* if their intersection is empty ($S(x) \cap S(y) = \emptyset$). Two decomposition slices $S(x)$ and $S(y)$ are *weakly dependent* if their intersection is non empty ($S(x) \cap S(y) \neq \emptyset$). Two decomposition slices $S(x)$ and $S(y)$ are *strongly dependent* if the former is contained in the latter ($S(x) \subset S(y)$).

Maximal decomposition slices: a decomposition slice that is not strongly dependent on any other decomposition slice is said to be *maximal*.

Decomposition slice graph: the strong dependence relationship between decomposition slices defines a graph, called the *decomposition slice graph*, the nodes of which are decomposition slices and the edges of which represent the containment relationship between slices, such that no intermediate slice exists: $(x, y) \in E \Leftrightarrow S(x) \subset S(y) \wedge \nexists z \in V : S(x) \subset S(z) \subset S(y)$.

Example:

S(`nl`) = {2, 5, 6, 8, 9, 15}
S(`nw`) = {1, 3, 5, 6, 10, 11, 12, 13, 14, 15}
S(`inword`) = {1, 5, 6, 10, 11, 12, 13, 15}
S(`nc`) = {4, 5, 6, 7, 15}
S(`c`) = {5, 6, 15}



Dependent statements: given two decomposition slices $S(x)$ and $S(y)$, the statements in $S(x) \cap S(y)$ are called *dependent statements*. The other statements of $S(x)$ and $S(y)$ are *independent*. Given a decomposition slice $S(x)$, its *dependent statements* are those shared with some maximal decomposition slice different from $S(x)$. All other statements are *independent*: they are not shared with other maximal decomposition slices. Consequently, only maximal decomposition slices may possess independent statements. Given a program and its maximal decomposition slices, the union of the statements that are independent in each maximal slice $S(x)$ are the *independent statements* of the program. The statements that are dependent in some

maximal slice $S(x)$ are the *dependent statements* of the program. Independent statements represent the computation that is unique to some maximal slice. Dependent statements capture the computation that different maximal slices share. When modifying a program, dependent statements cannot be changed without ripple effects outside the focus of interest (i.e., the maximal slice to be changed).

It should be noted that statements 1, 10, 11, 12, 13 are independent statements of $S(\texttt{nw})$, but dependent statements of $S(\texttt{inword})$.

Complement of a decomposition slice: the complement of a decomposition slice $S(x)$, denoted as $\Sigma(x)$, is obtained by deleting from the program all independent statements of $S(x)$ and all the output statements of the program. The result is the subprogram which computes the rest of the program specification not computed by $S(x)$. The whole program can thus be decomposed into the direct sum of a slice of interest and of its complement. The complement can in turn be decomposed into a maximal slice and its complement, until all maximal decomposition slices are obtained. Their direct sum gives the original program.

Dependent variables: a variable that is defined in a dependent statement is called a *dependent variable*. If all statements defining a variable are independent, the variable is said to be an *independent variable*.

Rules for program modification:

1. Independent statements can be deleted from a decomposition slice without ripple effects.

2. Statements defining independent variables can be added anywhere in a decomposition slice (a new variable is always independent).

3. Logical expressions and output statements can be added anywhere in a decomposition slice.

4. New control statements can surround only independent statements.

If this list of rules is respected, the complement of the decomposition slice of interest remains unaffected by ripple effects. In other words, only the functionality selected for change is modified, while the rest of the program

continues to behave exactly as before. A direct consequence is that regression testing can be limited to the decomposition slice that was changed (i.e., the complement needs not be re-tested).

If the change to be performed involves a dependent variable x and we want to preserve the behavior of the complement (no ripple effects), two solutions are possible:

1. A new variable is introduced. The variable of interest is assigned to it and then the new variable can be manipulated as needed.

2. Enlarge the decomposition slice, focus of the change, if possible, in the following way: if $S(x)$ is not maximal, add all maximal slices containing $S(x)$. If $S(x)$ is maximal, add all slices in which x is a dependent variable. In the enlarged slice x is assured to be independent.

3.3 Dynamic slicing

Definition:
A **dynamic slice** $S_d(x, n, I)$ of program P on variable x at statement n with input I is a subprogram of P computing the same value of x at n when I is provided as input.

A dynamic slice contains the statements affecting the value of a variable at a program point for a particular execution of the program, rather than for every execution in which the program terminates normally.

Definition:
An **execution trace** (aka execution history or trajectory) is the sequence of statements executed for a given input I, with each statement labelled by an occurrence counter (e.g., a loop statement numbered 2 will appear in the trace as 2^1 the first time it is executed, 2^2, 2^3, etc. the next times it is entered). When unnecessary, the occurrence counter will be omitted.

Given a program P, a dynamic slice $S_d(x, n, I)$ can be computed as the transitive closure of the dynamic data dependences of P (DU_d) at n on x, of the dynamic control dependences of P (CD_d) on n, and of the identity relationship (ID).

If not specified differently, the starting point for slice computation, n, is actually n^k, with k the largest occurrence counter for n. Otherwise, it is possible to explicitly specify the occurrence of n on which to compute the slice.

Dynamic data dependence (aka dynamic def-use chain): a dynamic data dependence exists between nodes n and m on variable x if n defines x, m uses x, n appears before m in the execution trace T and no intermediate node in T defines x:

$(n, m, x) \in DU_d \Leftrightarrow$
$(n, x) \in DEF$,
$(m, x) \in USE$,
$\exists p =< n, \ldots, k, \ldots, m >$ subseq of T: $k \neq n \Rightarrow (k, x) \notin DEF$

Dynamic control dependence: node n holds a dynamic control dependence on node m if there is a static control dependence between n and m, and if there is a path in the execution trace T from n to m such that n holds a transitive dynamic control dependence on all intermediate nodes in the path.

$(n, m) \in CD_d \Leftrightarrow$
$(n, m) \in CD$,
$\exists p =< n, \ldots, k, \ldots, m >$ subseq of T: $k \neq n \Rightarrow (n, k) \in {CD_d}^*$

Identity: node n^i is identical to node m^j if n and m are two successive occurrences of the same node in the execution trace T.

$(n^i, m^j) \in ID \Leftrightarrow \quad m = n \ \wedge \ i < j \ \wedge \ \exists p =< n^i, \ldots, m^j >$ subseq of T

The identity relationship is included in slice computation only if *executable slices* are being determined. When the objective is determining the statements affecting a value at a given execution point, the identity relationship can be ignored. The result will be a *non executable slice*.

Dynamic program slicing algorithm:

$S_d(x, n, I) \leftarrow \{n' : (n', n, x) \in DU_d \vee (n', n) \in CD_d\}$
while $S_d(x, n, I)$ increases
$S_d(x, n, I) \leftarrow S_d(x, n, I) \bigcup \{n'' : \exists n' \in S_d(x, n, I), \exists y \in V : (n'', n', y) \in DU_d$
$\vee \ \exists n' \in S_d(x, n, I) : (n'', n') \in CD_d \cup ID\}$
end while

Dynamic slicing example:

```
main() {
1   scanf("%d", &x);
2   for (i = 0 ; i < x ; i++) {
3       a = 2;
4       if (i > 0)
5          if (i == 1)
6             a = 4;
           else
7             a = 6;
8       z = a;
    }
9   printf("%d\n", z);
}
```

Slice	Executable	Non executable
(z, 9, [x=1])	{1, 2, 3, 8}	{1, 2, 3, 8}
(z, 9, [x=2])	{1, 2, 3, 4, 5, 6, 8}	{1, 2, 4, 5, 6, 8}
(z, 9, [x=3])	{1, 2, 3, 4, 5, 6, 7, 8}	{1, 2, 4, 5, 7, 8}

Dynamic Dependence Graph (DDG): graph the nodes of which are occurrences of program statements from an execution trace and the edges of which are either dynamic control dependences or dynamic data dependences between statement occurrences. Data dependence edges can be labelled with the variable on which the dependence holds, while control dependence edges can be labelled with the truth value which determines the execution of the target node. A fictitious *entry* node is connected via (true) control dependences to each node that is not controlled by any other different node. Slice computation on the DDG is straightforward:

$S_d(x, n, I) = \{m \in P | \exists p =< m, ..., n > \text{ in DDG } \}$

Since statements in the DDG are replicated for each occurrence, the space requirements of the related algorithm may increase without bound. To mitigate this problem, it is possible to introduce a *merge* operation between DDG nodes, to be executed during DDG construction each time there is no difference between the new occurrence of a statement and a previous one, i.e., the respective dynamic slices are the same.

Dynamic slicing in presence of arrays and pointers:
Since the subscript value used to access an array component is known at run time as well as the location pointed by a pointer variable, it is possible to apply the same slicing algorithm even in presence of arrays and pointers. Run time information will be exploited to determine the exact component or memory location which is defined or used at a statement, thus allowing for a precise computation of the data dependences.

3.4 Amorphous slicing

Traditional program slices are obtained by deleting commands from the original program. While the restriction of command deletion as the only allowed transformation is appropriate for applications such as debugging, for other uses of slicing such as program comprehension and reuse less restrictive requirements on the allowed transformations may lead to simpler and more meaningful slices. The theoretical framework supporting the extension of traditional slicing so as to permit less restrictive code transformations is the *program projection* framework:

Simplicity measure:
A simplicity measure is a pair $(F, \sqsubseteq)$, where F is a total function from programs to a set which is partially ordered by $\sqsubseteq$.

Projection:
Given a simplicity measure $(F, \sqsubseteq)$ and an equivalence relation $\sim$, a program P is a projection of a program Q if and only if $F(P) \sqsubseteq F(Q)$ and $P \sim Q$.

Minimality:
Given a simplicity measure $(F, \sqsubseteq)$ and an equivalence relation $\sim$, the program P is minimal if there exists no program Q such that $Q \sim P$ and $F(Q) \sqsubset F(P)$.

Minimal projection:
Given a simplicity measure $(F, \sqsubseteq)$ and an equivalence relation $\sim$, a program P is a minimal projection of a program Q if and only if P is minimal and P is a projection of Q.

Syntactic subset simplicity:
Let F be the mapping from programs to the set of statements they contain. The *syntactic subset simplicity* measure is $(F, \subseteq)$.

Amorphous simplicity:
Let F be the mapping from programs to the number of nodes in the control flow graph. The *amorphous simplicity* measure is $(F, \leq)$.

Static equivalence:
Given two programs P and Q and a slicing criterion (x, n), P is equivalent to Q on (x, n) if and only if in all executions of P and Q with the same input, the state σ (mapping names to values) reached along each respective trajectory ending at n maps x to a same value.

A *traditional static slice* respects the syntactic subset simplicity and the static equivalence properties. In fact, both original and sliced programs produce the same value of x at n, with the slice being a subset of the original program.

An *amorphous static slice* is a program projection that respects the amorphous simplicity and the static equivalence properties. An amorphous slice produces the same value of x at n as the original program, but contains less statements. Moreover, the statements it includes are not required to be a subset of the statements in the original program. Amorphous static slicing subsumes traditional static slicing in that syntactic subset simplicity implies amorphous simplicity. Thus, traditional static slices are also amorphous slices (but the inverse is not true).

Proposition:
Minimal amorphous slices are not generally computable.

Proof:
Consider the following program fragment P, where `f` and `g` have no effect upon the global variable `x`:

```
1    {
2      scanf("%d", &x);
3      if (f(x) != g(x))
4         z = 1;
5    }
```

The minimal amorphous slice on $(z, 5)$ is P if `f` and `g` do not return the same value on every input `x`, but it is $\emptyset$ if `f` and `g` are equivalent. Since the problem of deciding if two functions are equivalent is undecidable, minimal amorphous slices are not generally computable. □

Amorphous program slices can be obtained from a given program by applying semantic preserving transformation rules that eventually reduce the number of statements in the resulting program. This can be done in combination with traditional slicing. Specifically, it can be executed as a post-processing after the traditional slice has been determined.

General purpose transformation rules usable to produce amorphous slices are difficult to define. On the other side, it is possible to determine transformation rules that are likely to reduce the program size when they are focused on a specific domain (i.e., when amorphous slicing is used to answer questions of a specific, well defined category).

As an example, let us consider the problem of analyzing the allocation of dynamic memory. This specific domain can be made amenable to slicing by making the implicit state associated with memory allocation explicit. To this aim an additional program variable `hp` is introduced to record the bytes allocated or deallocated from the heap. A subset of the transformation rules that can be defined in this domain to achieve amorphous slicing follows:

Unfold Assignment:

$$\frac{e_3 = \text{replace}(e_2, i, e_1)}{[[i = e_1; i = e_2]] \Rightarrow [[i = e_3]]}$$

Collapse Incremental For Loop:

$$\frac{\{i_1, i\} \cap \text{USE}(e_3) = \emptyset}{[[\texttt{for}\ (i = e_1; i < e_2; i = i + 1) \quad i_1 = i_1 + e_3;]] \Rightarrow [[i_1 = i_1 + (e_2 - e_1) * e_3;]]}$$

Pull Read:

$$\frac{i \notin (\text{USE}(c) \cup \text{DEF}(c))}{[[c\ \texttt{scanf("\%d", \&i);}]] \Rightarrow [[\ \texttt{scanf("\%d", \&i);}\ c]]}$$

Example of amorphous slicing:

```
1*    hp = 0;
2     pass = NULL;
3     fail = NULL;
4     passnum = 0;
5     printf("Number of students?");
6     scanf("%d", &num);

7     for (i = 0 ; i < num ; i = i + 1) {
8*        hp = hp + 32;
9         e = (exam) malloc(32);
10        printf("\n%d name:   ", i + 1);
11        scanf("%s", &name);
12        printf("\n%d mark:   ", i + 1);
13        scanf("%s", &mark);
14        strcpy(e->name, name);
15        e->mark = mark;
```

```
16*       hp = hp + 10;
17        r = (list) malloc(10);
18        r->datum = e;
19        if (mark >= 18) {
20           r->next = pass;
21           pass = r;
22           passnum = passnum + 1;
          } else {
23           r->next = fail;
24           fail = r;
          }
      }
25    printf("\n\n");
26    for (i = 0 ; i < passnum ; i = i + 1) {
27        e = (exam) pass->datum;
28        printf("\n%d name:   %s", e->name);
29        printf("\n%d mark:   %s", e->mark);
30        pass = pass->next;
      }
31    exit(0);
```

After traditional slicing:

```
1     hp = 0;
2     scanf("%d", &num);
3     for (i = 0 ; i < num ; i = i + 1) {
4         hp = hp + 32;
5         hp = hp + 10;
      }
```

After amorphous slicing (Unfold, Collapse, Pull, Unfold):

```
1     scanf("%d", &num);
2     hp = num * 42;
```

3.5 Conditioned slicing

Definition:
A **conditioned slice** $C(x, n, F)$ of program P on variable x at statement n under condition F is a subprogram of P computing the same value of x at n for every input satisfying F.

F is a first order logic formula on the input variables V_{in} of the program. The triple (x, n, F) is called *conditioned slicing criterion*.

Computation of conditioned slices:

1. Determine the conditioned program by means of symbolic execution.
2. Mark data and control dependences traversed during symbolic execution.
3. Compute the transitive closure of marked data and control dependences.

Symbolic execution of a program consists of producing a set of symbolic states for each statement encountered during CFG traversal. A *symbolic state* is a pair *(state, path-condition)*, where the *state* maps each variable to a symbolic value (i.e., an algebraic expression containing symbolic constants), while the *path-condition* gives the predicate that must be true to reach and execute the current statement.

The logic formula F in the slicing criterion can be easily transformed into an initial path condition, which is assumed at the beginning of symbolic execution. Due to such an initial path condition, some of the branches in the program are never traversed and the associated statements can be discarded. The resulting program is a subprogram of the initial program called *conditioned program*.

Problems can arise during the symbolic execution of loops, when the path condition does not allow deciding on the loop termination. To continue execution past the end of the loop, the *loop invariant* can be determined and

exploited. Alternatively, when such a computation is not possible, a human can drive loop execution, forcing termination when the path condition cannot further restrict the program to be considered.

During symbolic execution not all statically possible data and control dependences are traversed, even if the related statements are acually traversed. Let us consider the following code fragment:

```
1    input(a, b);
2    w = 0;
3    x = b + 2;
4    if (a > 0)
5       x = b * 2;
     else
6       w = b + a;
7    z = x + w;
```

If $F = (a > 0)$, the symbolic execution of this program fragment never traverses the edge (4, 6). As a consequence only the data dependence $(5, 7, x)$ is active in every possible execution satisfying F. Nevertheless, statement 3 is traversed in every symbolic execution, and therefore belongs to the conditioned program, while statement 6 does not. To discard control and data dependences that are not allowed by symbolic execution, a simple marking procedure is activated: last definition of each variable is recorded during symbolic execution of each statement. When a variable is used, the data dependence associated with the last definition recorded for such a variable is marked. Similarly, control dependences are marked when a controlled statement is executed after traversing the controlling statement. Transitive closure on data and control dependences is then limited to the marked dependences only.

Example of conditioned slicing:

```
1   main () {
2     int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3     scanf("%d", &test0); scanf("%d", &n);
4     i = posprod = negprod = 1;
5     possum = negsum = 0;
6     while (i <= n) {
```

```
7         scanf("%d", &a);
8         if (a > 0) {
9             possum += a;
10            posprod *= a; }
11        else if (a < 0) {
12                 negsum -= a;
13                 negprod *= (-a); }
14             else if (test0) {
15                     if (possum >= negsum)
16                          possum = 0;
17                     else negsum = 0;
18                     if (posprod >= negprod)
19                          posprod = 1;
20                     else negprod = 1; }
21        i++; }
22    if (possum >= negsum)
23         sum = possum;
24    else sum = negsum;
25    if (posprod >= negprod)
26         prod = posprod;
27    else prod = negprod;
28    printf("%d \n", sum);
29    printf("%d \n", prod); }
```

Let us consider the slicing criterion: (`sum`, 28, $(\forall i, 1 \leq i \leq n, a_i > 0)$).

If the symbolic state pairs the variables with the following symbolic constants: (`n`, γ), ($\mathtt{a}_1$, α_1), ..., ($\mathtt{a}_\gamma$, α_γ), the initial path condition becomes: $(\forall i, 1 \leq i \leq \gamma, \alpha_i > 0)$

Let us consider the execution of statement 8, in the symbolic state: ({($\mathtt{a}_1$, α_1), (`test0`, β), (`n`, γ), (`i`, 1), (`posprod`, 1), (`negprod`, 1), (`possum`, 0), (`negsum`, 0), (`prod`, *undef*), (`sum`, *undef*)}, $(1 \leq \gamma)$)

When no initial path condition is considered, the two following symbolic states are produced and propagated along the two branches of the conditional statement:

({($\mathtt{a}_1$, α_1), (`test0`, β), (`n`, γ), (`i`, 1), (`posprod`, 1), (`negprod`, 1), (`possum`,

0), (`negsum`, 0), (`prod`, *undef*), (`sum`, *undef*)}, $(1 \leq \gamma \wedge \alpha_1 > 0))$

({(`a`$_1$, α_1), (`test0`, β), (`n`, γ), (`i`, 1), (`posprod`, 1), (`negprod`, 1), (`possum`, 0), (`negsum`, 0), (`prod`, *undef*), (`sum`, *undef*)}, $(1 \leq \gamma \wedge \alpha_1 \leq 0))$

If the initial path condition is added to the initial symbolic state, only the former symbolic state is produced and statement 11 is never reached, since only the `then` branch of the conditional is traversed.

C(`sum`, 28, $(\forall i, 1 \leq i \leq n, a_i > 0)) = \{3, 4, 5, 6, 7, 8, 9, 21, 22, 23\}$

C(`prod`, 29, $(\forall i, 1 \leq i \leq n, a_i > 0)) = \{3, 4, 6, 7, 8, 10, 21, 25, 26\}$

3.6 Other variants of program slicing

Dicing: the searched statements belong to some computations (e. g., affected by a bug), but not to another set of computations (e. g., unaffected by the bug). A dice can be obtained by intersecting the slices over the first set and subtracting the slices associated with the second set.

Chopping: *chop(t, s)* includes all statements affected by the computation at t that in turn affect the computation at s. Intraprocedural chopping can be achieved by intersecting the forward slice starting at t with the backward slice that starts at s. Interprocedural chopping requires more sophisticated algorithms.

References

(Binkley 1996) D. Binkley and K. B. Gallagher. *Program slicing.* In Advances in Computers, vol. 43, pp. 1-50, 1996.

(Gallagher 1991) K. B. Gallagher and J. R. Lyle. *Using Program Slicing in Software Maintenance.* In IEEE Transactions on Software Engineering, vol 17. n. 8. pp. 751-761, August 1991.

(Harman 1997) M. Harman and S. Danicic. *Amorphous Program Slicing.* In Proc. of the International Workshop on Program Comprehension (IWPC'97), pp. 70-79, Dearborn, Michigan, May 1997.

(Harman 1998) M. Harman, Y. Sivagurunathan and S. Danicic. *Analysis of Dynamic Memory Access using Amorphous Slicing.* In Proc. of the International Conference on Software Maintenance (ICSM'98), pp. 336-345, Bethesda, Maryland, November 1998.

(Canfora 1998) G. Canfora, A. Cimitile and A. De Lucia. *Conditioned Program Slicing.* In Information and Software Technology, vol. 40, pp. 595-607, Elsevier Science, 1998.

Chapter 4

Reverse engineering

4.1 Pattern matching

Recognition and transformation of patterns in a source program can be conducted at increasing levels of abstraction:

Text level: the source program is represented as a sequence of characters and pattern matching is based on string matching. Regular expressions can be employed to avoid some of the problems related to textual variants and spacing/indentation.

Examples:

`"if (x > 100)"` $\longrightarrow$ `"if (x >= 100)"`

`"if\s*(\s*x\s*>\s*100\s*)"` $\longrightarrow$ `"if (x >= 100)"`

With this approach only simple patterns can be expressed and surface variantes, such as the presence of interleaved comments, may invalidate them even in the simplest cases. Moreover, syntactic and semantic constraints cannot be checked at the textual level. For example, the presence of two loop statements *at the same nesting level* cannot be expressed as a regular expression over the textual representation of the program. Specifications about data declarations are independent from the ordering of the declarations. A regular expression for such cases should include all possible permutations, which becomes quickly unfeasible.

Syntactic level: the source program is represented as the Abstract Syntax Tree (AST) produced by the parser. Only the logical structure of the program is preserved, while surface features, such as spaces, comments, etc., are disregarded. Patterns at this level are parsed into trees, and a match occurs when a corresponding subtree is found in the program's AST.

Syntactic patterns may include *pattern variables*, which are matched when it is possible to bind them to objects in the program's AST. They are expressed in a *pattern language*, i.e. an extension of the source code programming language with wildcards. An example of named wildcard matching any subtree consists of an identifier prefixed by the character `@`. A more detailed presentation of pattern languages is provided below.

Example:

`"if (@expr > 100)"` $\longrightarrow$ `"if (@expr >= 100)"`

The tree for the left hand side pattern is constructed and searched in the program's AST. If a match is found, the pattern variable `@expr` is bound to a program expression (represented as a subtree), which is copied into the transformed pattern (right hand side).

Semantic level: semantic properties of the program may be used to improve the expressiveness of patterns. The AST representation of the program is annotated with semantic information. Examples of this kind of information include the control and data dependences between statements. Each node in the AST is decorated with additional attributes storing such semantic relationships.

Example:

`if-stmt-100:"if (@expr > 100)"` $\longrightarrow_C$ `"if (@expr >= 100)"`
`where`
C `= control-dep(while-stmt, if-stmt-100),`
`while-stmt: "while (@cond)"`

```
if (i > 100) {
  ...
}
```

```
while (c) {
   if (j > 100) {
     ...
  }
}
```

Concept level: extraction of information from the source code can be further improved by exploiting knowledge about abstract concepts, concerning programming, problem solving and application domain, that are expected to be implemented in the program. Additional annotations are attached to the AST nodes to represent knowledge about the abstract concepts to which a program fragment contributes.

Example:

if-stmt-100:"if (@expr > 100)" $\longrightarrow_C$ "if (@expr >= 100)"
where
C = control-dep(while-stmt, if-stmt-100) and
part-of(while-stmt, sequential-search),
while-stmt: "while (@cond)"

```
while (c) {
   if (k > 100) {
     ...
   }
}
found = false;
i = 0;
while (!found) {
   if (a[i].equals(s))
      found = true;
   if (j > 100) {
     ...
   }
   i++;
}
```

The concepts contained in a program can be classified into:

- *programming concepts:* typical organizations of the instructions according to the manipulated data structures and the algorithms working on them (e.g., sequential search in an array);
- *architectural concepts:* realization of interconnections between the components of a system. Examples include module interfaces, communication between concurrent processes, data base access, etc.
- *domain concepts:* application domain and business logic functions implemented in the code.

Concepts can be described in terms of their *attributes*, the *sub-concepts* or *patterns* they consist of, and the *constraints* that must hold when a match is found.

Concept recognition cannot be as complete and accurate as the syntactic analysis, so that in general its outcome is not expected to yield a tree structure covering the entire program (as with the AST). Rather, a set of hierarchies of concepts is produced (no single concept is assigned to the entire program), covering only portions ("islands") of the code.

Example:

```
plan sequential-search(
       array:  @arr,
       object:  @obj,
       index:  @i,
       loop:  @w,
       incr-index:  @inc)
consists of
   init-index:  assign(lhs:  @i, rhs:  0)
   loop:  @w = while-loop(pattern:
          "while (!@f)")
   condition:  if-stmt(pattern:
          "if (@arr[@i].equals(@obj)) @f = true;")
   incr-index:  @inc = incr-stmt(inc-var:  @i)
such that
   data-dep(init-index, loop, @i)
```

```
    data-dep(incr-index, loop, @i)
    data-dep(init-index, incr-index, @i)
    data-dep(incr-index, incr-index, @i)
    control-dep(@w, condition)
    control-dep(@w, incr-index)
```

```
plan incr-stmt(inc-var:  @x)
consists of
    assign(lhs:  @x, rhs:  "@x + 1")
```

```
plan incr-stmt(inc-var:  @x)
consists of
    incr(pattern:  "@x++;")
```

Transformations can be defined on top of pattern matching. Example:

```
transformation handle-index-out-of-bounds
consists of
    search-obj:  sequential-search(
        array:  @arr,
        object:  @obj,
        index:  @i,
        loop:  @w,
        incr-index:  @inc)
then do
    insert-after(@inc, "if (@i >= @arr.length) break;")
    @ww = clone(@w)
    insert-before(@w, "if (@arr.length > 0) @ww")
    delete(@w)
```

If this transformation is applied to the code fragment presented above, the following statements are produced:

```
while (c) {
   if (k > 100) {
     ...
   }
}
found = false;
i = 0;
if (a.length > 0)
   while (!found) {
      if (a[i].equals(s))
         found = true;
      if (j > 100) {
        ...
      }
      i++;
      if (i >= a.length)
         break;
   }
```

4.1.1 Pattern languages

Pattern languages extend the source code programming language with a set of symbols that can be used as substitutes for syntactic entities. In the examples above a simple extension based on named subtrees, indicated as `@`*name*, where *name* is bound to the matching subtree, has been used. A richer set of primitive wildcards can be introduced to express more sophisticated patterns:

Entity	Symbol
Declaration	$d_{name}
Declaration set	$*d_{name}
Type	$t_{name}
Variable	$v_{name}
Variable set	$*v_{name}
Function	$f_{name}
Expression	#_{name}
Expression set	#*_{name}
Statement	@_{name}
Statement sequence	@*_{name}
Fixed nesting	@{{...{}...}}
Arbitrary nesting	@{**}
Alternative	@[stmt-type1 | stmt-type2]
Reference to identifiers	@<id-1, id-2, ...>, #<...>, $f<...>

The suffix _{*name*} is present only in *named wildcards*, i.e., wildcards that reference a *same* subtree in different parts of the pattern. Otherwise, it is not necessary to specify such a suffix.

Examples of patterns:

Find all `while` *statements where the condition of the* `while` *statement is a relational expression of the form non-equal-to zero.*

```
while (# != 0) @;
```

Find all occurrences of three consecutive `if` *statements.*

```
if # @;
if # @;
if # @;
```

Find all `if` *statements where '=' has been mistakenly used in place of '==' in the condition.*

```
if (# = #) @;
```

Find all declarations of the variable x.

```
$t x;
```

Find all statements which are procedure calls.

```
$f(#*);
```

Find all functions which return values of type `complex`*.*

```
complex $f($*d) {@*;}
```

Find a sequence of statements such that three or more `if` *statements occur, possibly with other statements between them.*

```
if # @;
@*;
if # @;
@*;
if # @;
```

Find a set of declarations, one of which is a declaration of a variable `maxval` *of type* `int`*.*

```
int maxval;
$*d;
```

Find all instances where a variable of type `int` *is incremented by 1.*

```
int $v_x;
$v_x = $v_x + 1;
```

Find situations where the values of two variables are being swapped.

```
$v_tmp = $v_x;
@*;
$v_x = $v_y;
@*;
$v_y = $v_tmp;
```

Find all `struct` *declarations containing a field whose type is recursively defined.*

```
$t_1 {
  $*d;
  $t_1 *$v;
}
```

Find all functions that have references to the identifier `xmax`*.*

```
$t $f <xmax> ($*d) { @*; }
```

Find a structure of three nested loops.

```
@[while | for | dowhile] {@*;
  @[while | for | dowhile] {@*;
    @[while | for | dowhile] {@*;
    }
  }
}
```

Find the statements which determines the maximum value in an array.

```
$t $f ($*d)
{*
   @[while | for | dowhile] {*
      if ($v_2[#] > $v_3)
          $v_3 = $v_2[#];
   *}
*}
```

Example of matching code:

```
int find_max(int arr, int N) {
  int i;
  int maxstore = arr[0];
  for (i = 1 ; i < N ; i++) {
     if (arr[i] > maxstore)
        maxstore = arr[i];
  }
  return maxstore;
```

Pattern matching algorithm: the pattern is transformed into a *code pattern automaton* (CPA), a special purpose nondeterministic finite state automaton. A CPA *interpreter* runs the CPA with the AST as input. A match occurs whenever the CPA reaches a final state. The interpreter maintains information about bindings of named wildcards in a set of data structures called *binding tables*.

A *code pattern automaton* (CPA) is a 6-tuple of the form $\langle Q, \Sigma, A, \Gamma, q_0, F \rangle$, where:

1. Q is the set of states.

2. Σ is the input alphabet consisting of AST nodes. $\Sigma = \Sigma_N \cup \Sigma_L$, where Σ_N represents the internal nodes of the AST and Σ_L represents the leaves.

3. A is the set of AST navigation functions given by $A = \{mlc, mrs, mp\}$, where mlc is interpreted as *move to left child*, mrs as *move to right sibling*, and mp as *move to parent*.

4. Γ is the set of transition functions given by $\Gamma = \{CAT, VAL\}$, where:

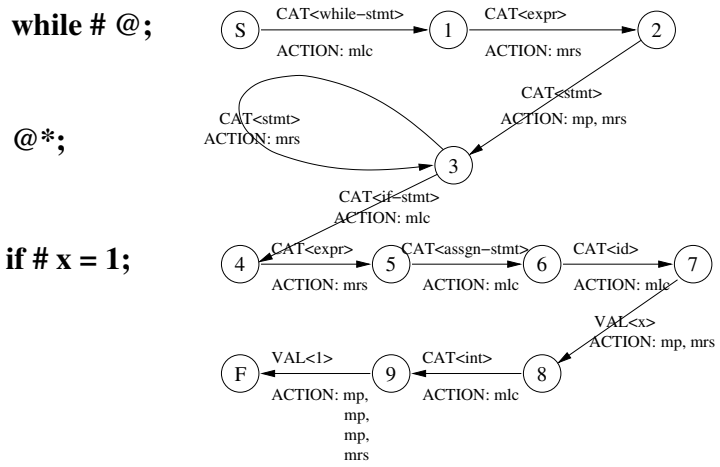
   - $CAT : Q \times \Sigma_N \longrightarrow 2^{Q \times A^+}$
     $CAT\ \langle category \rangle$*; ACTION:* $\langle actions \rangle$
     CAT arcs are traversed when the input node is an internal node of the syntactic category specified as $CAT\ \langle category \rangle$. The actions executed when the input is consumed are specified as *ACTION:* $\langle actions \rangle$.

- $VAL : Q \times \Sigma_L \longrightarrow 2^{Q \times A^+}$
  VAL $\langle value \rangle$*; ACTION:* $\langle actions \rangle$
  VAL arcs are traversed when the input node is a leaf whose value is that specified in *VAL* $\langle value \rangle$. The actions executed when the input is consumed are specified as *ACTION:* $\langle actions \rangle$.

5. q_0 is the initial state.

6. $F \subseteq Q$ is the set of final states.

Example:



When the pattern contains named wildcards, the interpreter uses *binding tables* to keep track of the associations between named wildcards and AST subtrees. Since the matching is nondeterministic, more than one match are explored at any given time. Correspondingly, more binding tables, each associated with a distinct exploration of the CPA, are maintained.

4.1.2 Graph parsing

The main difficulties in reverse engineering based on pattern matching are:

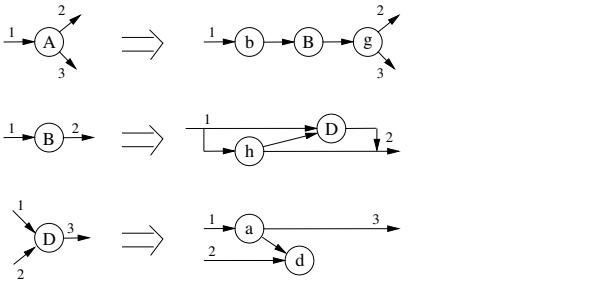
- *Syntactic variation*: the same behavior can be achieved by writing the code in many different ways.
- *Non-contiguousness*: portions of a concept may appear in statements that are scattered through the program.
- *Implementation variations*: an abstraction can be implemented by exploiting several different algorithms and data structures described in the literature.
- *Overlapping implementations*: two or more distinct concepts may share portions of the respective implementations.
- *Unrecognizable code*: not every program statements can be reduced to a known concept.

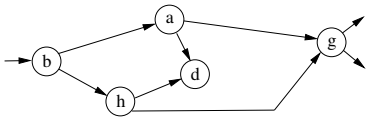
A way to overcome some of these problems consists of encoding the program and the concepts (called *"clichès"* in this context) as flow graphs, and exploiting graph parsing algorithms for their recognition in the code.

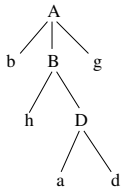
Flow graphs: a flow graph is a labeled, directed, acyclic graph in which edges connect a node's input and output ports. Node labels identify node types; each node type has a fixed number of input and output ports. More than one edge can converge into a single input port (fan-in allowed) and more than one edge can diverge out of a single output port (fan-out allowed).

Flow graph grammar: a flow graph grammar is a set of rewriting rules (productions) of the type $L \rightarrow R$, specifying how a left hand side flow graph L containing a single non terminal node can be replaced by a right hand side flow graph R, containing terminal and possibly non terminal nodes. A binary *embedding relation* specifies the correspondence between the ports in L and the ports in R.

The derivation sequence of non terminal nodes that ends up with the parsed graph can be arranged into a tree called the *design tree* of the program.





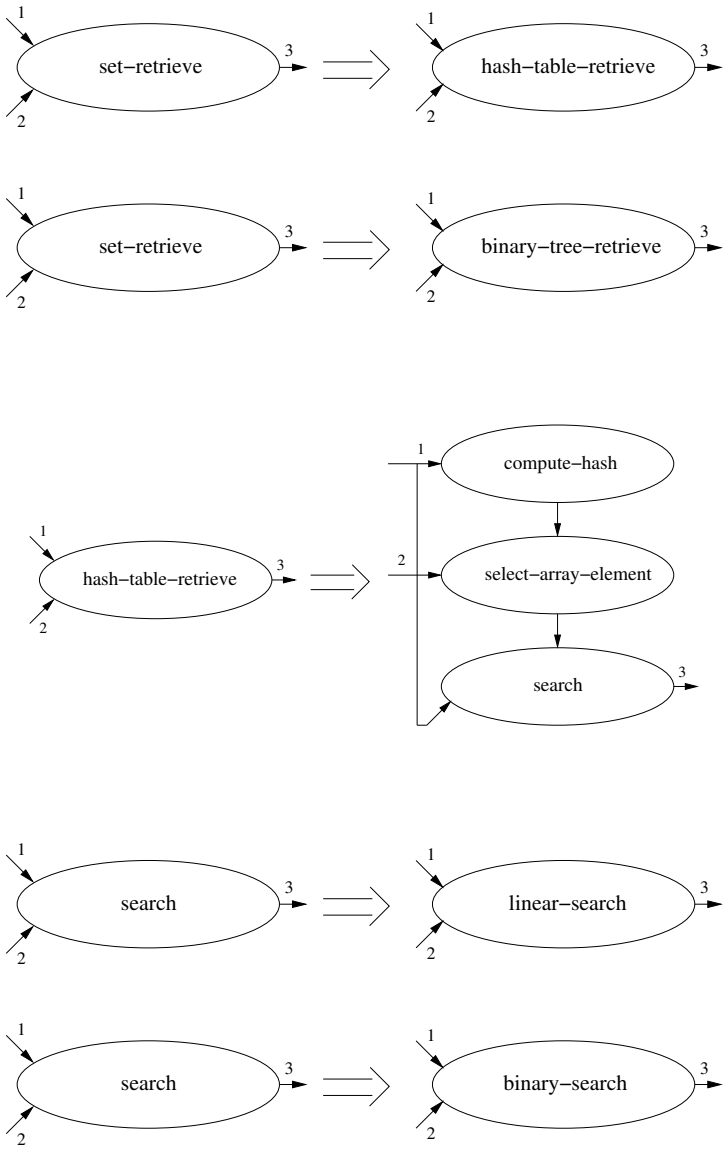


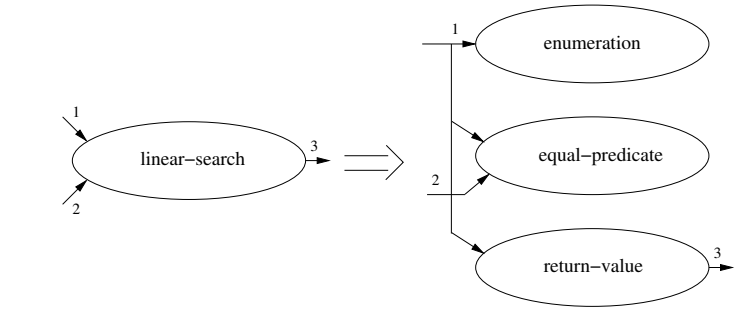
Flow graph Design tree

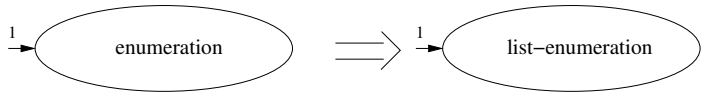
Partial flow graph parsing: given a flow graph representing the program's data flow and a clichè library encoded as a flow graph grammar, the derivations of the grammar that parse (a portion of) the input flow graph solve the problem of matching the clichès in the code. The result is coded in the forest of design trees otained from parsing.

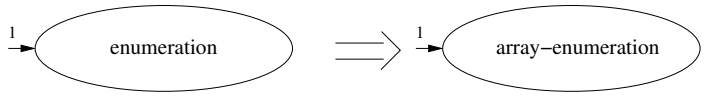
```
    CHART-PARSING
1   insert the grammar's productions (items) into the chart
2   while chart can be extended
3       select a partial item i to be extended
4       if i contains an unmarked terminal node n
5           for each flow graph's node m matching n
6               create a new item i' by marking node n as matched with m
7               insert i' into the chart, if not already present
8           end for
9       end if
10      if i contains a non terminal A
11          for each complete item j derived from A
12              create a new item i' by replacing A with j
13              insert i' into the chart, if not already present
14          end for
15      end if
16  end while
```

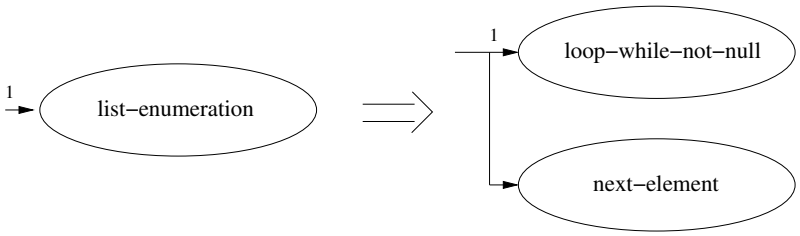
Example. Productions:




linear−search
1
2
3
enumeration
equal−predicate
return−value
1
2
3


enumeration
1
list−enumeration
1

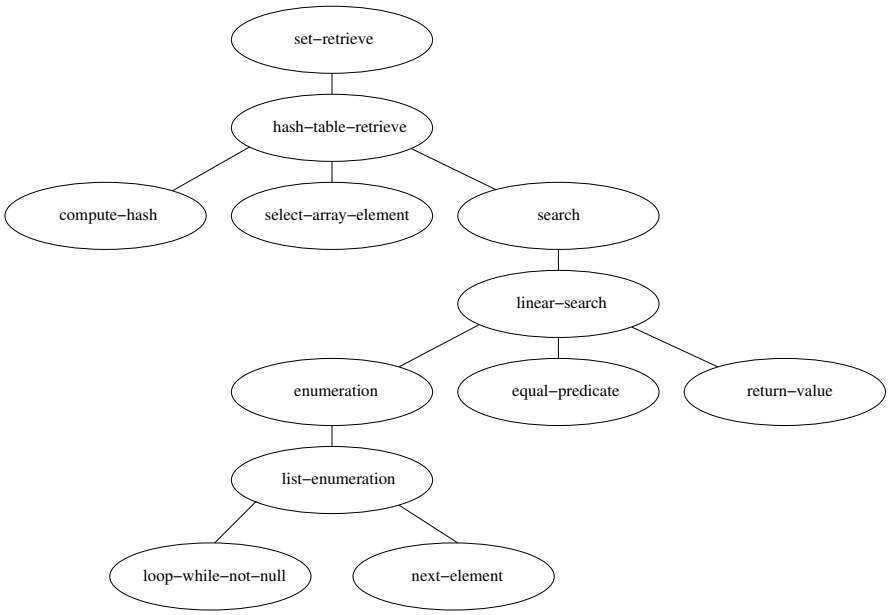

enumeration
1
array−enumeration
1


list−enumeration
1
loop−while−not−null
next−element
1

Input code:

```
Entry* table_lookup(Key key, Entry *table[]) {
1     int k = hash(key);
2     Entry *bucket = table[k];
3     while (bucket != 0) {
4        if (equal(bucket->key, key))
5            return bucket;
6        bucket = bucket->next;
      }
7     return 0;
}
```

Design tree:



Textual representation of the design tree, obtainable by automatically filling-in a predefined template:

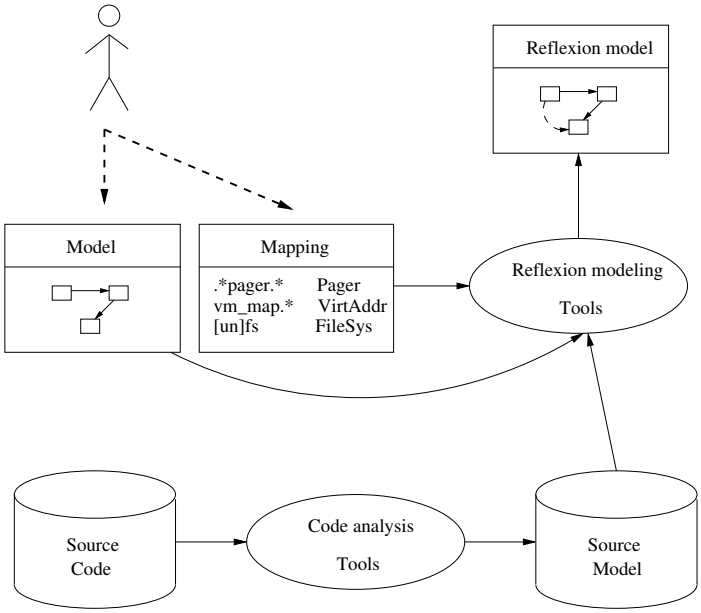
```
------------------------------------------------------------
table_lookup performs a set-retrieve operation. The set being
searched is table, while the element to be searched is
```

```
identified by key.

The set table is implemented as a hash table, the hashing function
of which is hash. Each hash table entry is a bucket, implemented as
a list. The selected entry is assigned to variable bucket.

A linear search is performed on the list bucket to find the
first element with field key equal to key. The comparison
function between list elements is equal.
-------------------------------------------------------------
```

4.2 Reflexion models



Software systems consist of collections of artifacts that are derived from each other. Design diagrams and source code are an example. During evolution, such artifacts are manipulated independently. Consequently they tend to "drift" apart over time. In fact, maintaining the consistency of these artifacts over time is time consuming, difficult, and rarely the highest priority.

Reflexion models help software engineers produce a high level model of the

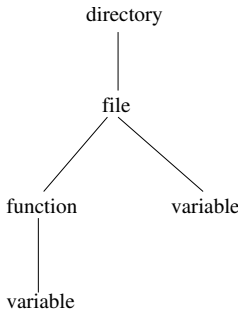
system consistent with the current implementation, which can be evolved from a (possibly outdated) model of the system, if any is available. An iterative refinement process leads to the final model of the system.

The user specifies an initial model, based on his/her beliefs about the organization of the system and on any available documentation. The model needs not be complete: only the portion of the system which is interesting for the ongoing task is represented in the model. This initial model consists of entities (e.g., modules) and relations between entities (e.g., calls), capturing the interactions between entities.

Then, the user specifies a mapping between structural information (called the *source model*), that is extracted from the source code by means of code analysis tools, and the entities in the initial, high level model. Only entities in the subsystems of interest have to be indicated in the map. The physical (e.g., directory and file) and logical (e.g., functions and variabes) structure of the source code can be used to group entities of the source model to be mapped. Such a structure complies with a *source entity naming tree*, which defines a naming mechanism to denote the organization of the entities in the source model. Moreover, regular expressions can be used to specify all names that share a common format.

Example of mapping:

```
[function=main        mapTo=main]
[directory=syntax     mapTo=translator]
[file=.*scan.*\.c     mapTo=lexer]
[function=optimize    mapTo=optimizer]
[function=gen.*       mapTo=generator]
```



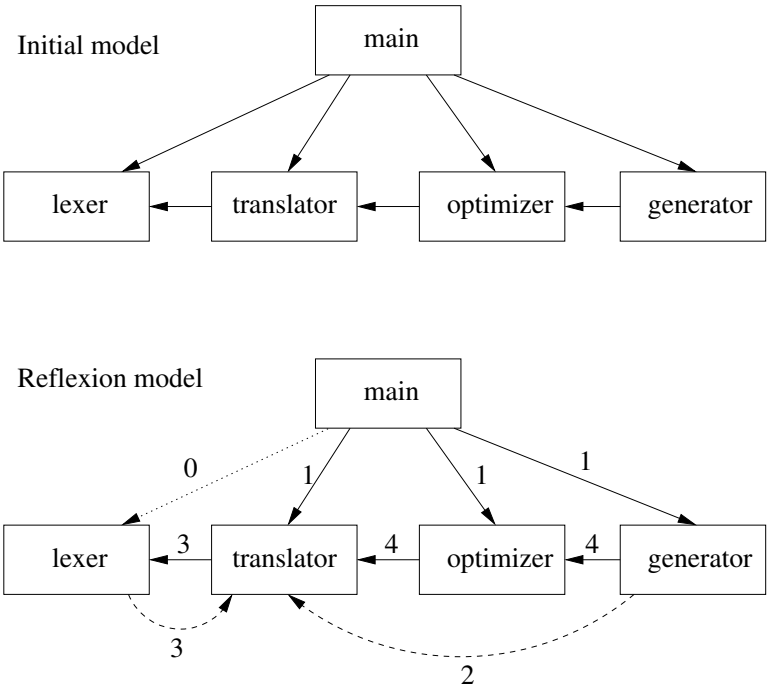
Source entity naming tree: each node in a naming tree defines a keyword that describes one aspect of the organizational information of the source model. The tree permits naming both physical and logical entities, and any combination of keywords can be used to describe the mapping between high level model and source model entities. Usage of the naming tree as well as of regular expressions allows providing the mapping in a concise way.

The text lines specifying the mapping between high level and source model are considered as a sequence. Consequently, their ordering *is* relevant. When an entry is encountered that permits mapping a source model entity to a high level entity, such a mapping is recorded and the presence of successive entries that map the given entity differently is ignored. Only the first mapping encountered for each source entity is considered. Consequently, it is possible to define a coarse-grain mapping, for example by means of regular expressions or directory information, and then to refine it by prepending lines in the mapping file, which handle the exceptions to the general, approximate rule. In the example above, if `main` is in directory `syntax`, it is mapped to `main` instead of `translator` thanks to the first line of the mapping. Moreover, a final mapping rule may be appended at the end, to map all entities that are otherwise unmatched by the previous cases.

Then, the reflexion modeling tools compute the *reflexion model*, which displays the *convergences* – source interactions that were expected by the developer, the *divergences* – source interactions that were not expected by the developer, and the *absences* – source interactions that were expected but not found. Divergences (but also convergences) can be queried by the user to investigate the source model relationships contributing to them. Queries can also be issued to determine the source model entities that are included in the reflexion model and those that are not. In the first case, the actual mapping exploited can be accessed. A coverage measure can also be provided, giving the proportion of source entities mapped to some high level entity.

Information displayed in the reflexion model and the output of the queries can be used to refine the initial model into a new model which better reflects the entities and the relations between entities as implemented in the source code.

Example:



4.3 Software reconnaissance

Two program understanding strategies are possible: the systematic and the opportunistic strategy. A programmer using the *systematic* strategy attempts to understand the program's overall organization and full range of behavior, while a programmer using the *opportunistic* approach tries to locate as soon as possible the specific parts of the program which need to be changed to implement the desired modification. It has been reported that the latter strategy may not provide sufficient knowledge about the interactions between components in the program, thus leading to unsuccessful modifications. However, in real world programming it is often impossible to allocate sufficient time to follow the systematic strategy, especially for large scale programs. The software reconnaissance method aims at isolating the code portions relevant to a given functionality, subject to modification, thus making opportunistic understanding less error-prone. The military definition of *reconnaissance* as a 'preliminary survey of enemy terrain' corresponds

to a view of the program to be modified as an enemy whose terrain has to be explored to plan the attack.

The end-user sees a program as providing a set of *functionalities* (aka features or functional requirements). The software engineer sees it as a set of *components* (e.g., procedures, functions, classes, blocks of statements, etc.). Let us consider a set of test cases T, partitioned into a subset V_f, containing all test cases that exhibit a given functionality f, and $V_f^c = T \setminus V_f$, containing all test cases that do not exhibit f. Execution of the test cases in T allows determining the following interesting sets of components:

- **CCOMPS** (*common components*): components that are executed in all test cases (T). Typically, utility components.
- **ICOMPS(f)** (*potentially involved components*): components that are executed in at least one test case from V_f.
- **IICOMPS(f)** (*indispensably involved components*): components that are executed in all test cases from V_f.
- **UCOMPS(f)** (*uniquely involved components*): components that are executed in some test case from V_f but are not executed in any test case from V_f^c.

UCOMPS(f) is generally a quite small fraction of the overall program, and probably provides a good place to start when trying to apply the opportunistic strategy.

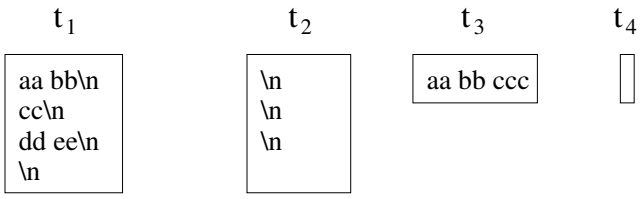
Components: the obvious unit to consider as a component is the subroutine, but this may be unsatisfactory when subroutines contain code that implements more than one functionality. A finer level of decomposition could be based, for example, on the arcs of the control flow graph or on the program statements.

Test cases: in practice, very useful results can be obtained with only a few test cases carefully chosen and classified.

The main limitation of the method is related to functionalities that are activated in every reasonable test case. They cannot be isolated since they are necessarily classified as utility coponents which belong to CCOMPS.

Example:

```
main() {
1   inword = NO;
2   nl = 0;
3   nw = 0;
4   nc = 0;
5   c = getchar();
6   while (c != EOF) {
7     nc = nc + 1;
8     if (c == '\n')
9        nl = nl + 1;
10    if (c == ' ' || c == '\n' || c == '\t')
11       inword = NO;
12    else if (inword == NO) {
13       inword = YES;
14       nw = nw + 1;
      }
15    c = getchar();
    }
16  printf("%d \n", nl);
17  printf("%d \n", nw);
18  printf("%d \n", nc);
}
```



- Functionalities of interest: counting the number of lines (f_1) and counting the number of words (f_2).
- Test cases exhibiting f_1: t_1, t_2.
- Test cases exhibiting f_2: t_1, t_3.

CCOMPS = {(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 16), (16, 17), (17, 18)}
ICOMPS(f_1) = E
IICOMPS(f_1) = {(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10), (10, 11), (11, 15), (15, 6), (6, 16), (16, 17), (17, 18)}
UCOMPS(f_1) = {(8, 9), (9, 10)}
ICOMPS(f_2) = E
IICOMPS(f_2) = {(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 10), (10, 11), (11, 15), (10, 12), (12, 15), (12, 13), (13, 14), (14, 15), (15, 6), (6, 16), (16, 17), (17, 18)}
UCOMPS(f_2) = {(8, 10), (10, 12), (12, 13), (12, 15), (13, 14), (14, 15)}

where E is the set of all control flow graph edges.

4.4 Concept analysis

Concept analysis is a branch of lattice theory that provides a way to identify maximal groupings of objects that have common attributes.

A *context* is a triple $C = (O, A, R)$, where O and A are finite sets (the objects and the attributes, respectively), and R is a binary relation between O and A: $R \subseteq O \times A$.

Let $X \subseteq O$ and $Y \subseteq A$. The mappings $\sigma(X) = \{a \in A | \forall o \in X : (o, a) \in R\}$ (the common attributes of X) and $\tau(Y) = \{o \in O | \forall a \in Y : (o, a) \in R\}$ (the common objects of Y) form a *Galois connection*, that is, the mappings are *antimonotone*:

$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1)$

$Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1)$

and *extensive*:

$X \subseteq \tau(\sigma(X))$ and $Y \subseteq \sigma(\tau(Y))$

A *concept* is a pair of sets (X, Y) – a set of objects, X, called the *extent* and a set of attributes, Y, called the *intent* – such that:

$Y = \sigma(X)$ and $X = \tau(Y)$

That is, a concept is a maximal collection of objects sharing common attributes.

A concept $c_0 = (X_0, Y_0)$ is a *subconcept* of concept $c_1 = (X_1, Y_1)$ ($c_0 \leq c_1$) if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$).

The subconcept relation forms a complete partial order (the *concept lattice*) over the set of concepts.

The fundamental theorem for concept lattices relates subconcepts and superconcepts as follows:

$\sqcup_{i \in I}(X_i, Y_i) = (\tau(\bigcap_{i \in I} Y_i), \bigcap_{i \in I} Y_i)$

The least upper bound (*supremum*) of a set of concepts can be computed by intersecting their intents, and by finding the common objects of the resulting intersection.

Dually, the largest lower bound (*infimum*) can be computed as follows:

$\sqcap_{i \in I}(X_i, Y_i) = (\bigcap_{i \in I} X_i, \sigma(\bigcap_{i \in I} X_i))$

The complete information about each node n in the concept lattice, associated with a concept $c = (X, Y)$, is given by the pair (X, Y). However, it is possible to represent the same information in a more compact and readable form by marking a node n with an object $o \in X$ or an attribute $a \in Y$ only if it is associated with the most special (respectively, general) concept c having o (resp., a) in the extent (resp., intent). The (unique) node of the concept lattice marked with a given object o is thus:

$$\gamma(o) = \inf\{n \in L | o \in Ext[n]\}$$

where *inf* gives the infimum of a set of concepts. Similarly, the unique lattice node marked with a given attribute a is:

$$\mu(a) = \sup\{n \in L | a \in Int[n]\}$$

where *sup* gives the supremum of a set of concepts. The objects in the extent of a lattice node n are then obtained as the set of objects at or below n, while the attributes in its intent are those marking n or any node above n.

A *concept partition* is a set of concepts whose extents form a partition of O. That is, $P = \{(X_1, Y_1), ..., (X_k, Y_k)\}$ is a concept partition iff the extents of the concepts cover the object set (i.e., $\bigcup X_i = O$) and are pairwise disjoint ($X_i \bigcap X_j = \emptyset$ for $i \neq j$).

Bottom-up algorithm for computing the concept lattice for a given context:

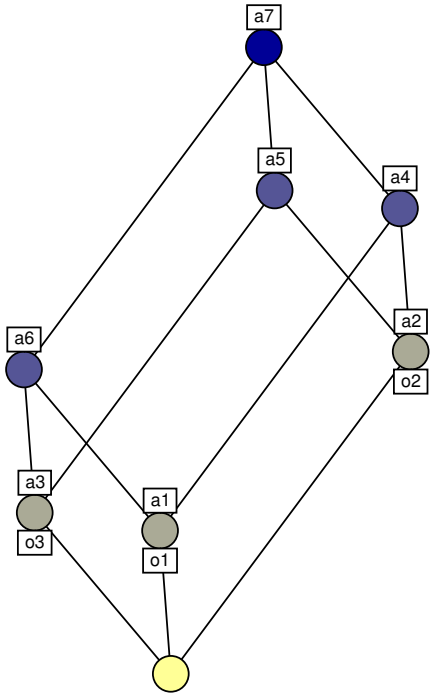
1. Compute the bottom element of the concept lattice: $(\tau(\sigma(\emptyset)), \sigma(\emptyset))$, with $\sigma(\emptyset) = A$.
2. Compute the *atomic* concepts – smallest concepts with extent obtained by treating each object as a singleton: $(\tau(\sigma(\{o\})), \sigma(\{o\})), o \in O$
3. Close the set of atomic concepts under join (**AtomicConceptClosure**).

AtomicConceptClosure

```
1   worklist ← {(c', c)|c ⋠ c' ∧ c' ⋠ c}
2   while worklist ≠ ∅
3       (c0, c1) ← RemoveFirst(worklist)
4       c'' = c0 ⨆ c1
5       if c'' is yet to be discovered
6           for each pairs of concepts (c'', c)
7               if c ⋠ c'' ∧ c'' ⋠ c
8                   Add(worklist, (c'', c))
9               end if
10          end for
11      end if
12  end while
```

Example:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
o_1	×			×		×	×
o_2		×		×	×		×
o_3			×		×	×	×



4.4.1 Feature location

The goal of this application of concept analysis is to identify the computational units that specifically implement a feature of interest. Execution traces obtained by running the program under given scenarios provide the input data (dynamic analysis).

Scenario: Sequence of user inputs to be provided during the execution of a program. Such inputs trigger actions with observable results (e.g., draw a circle by diameter and move it).

Feature: Realized functional requirement of a system. A feature is an observable behavior that can be triggered by the user (e.g., drawing an object).

Computational unit: An executable part of a system (e.g., block of statements, function, class, module).

Concept analysis is applied to a context where:

- Objects are computational units.
- Attributes are scenarios.
- A pair (o, a) is in relation R if computational unit o is executed when scenario a is performed.

Information about which computational unit is executed for each scenario can be collected by gathering execution traces of the program (dynamic analysis). This can be achieved by means of a *tracer* tool, or by instrumenting the program.

In addition to the context described above, it is assumed that a **scenario-feature mapping** be available. Such a mapping states which features are exercised in each scenario. Formally, it can be modeled by representing a scenario a as a set of features: $a = \{f_1, f_2, f_3\}$.

A concept in the resulting concept lattice groups all computational units executed by all scenarios in the intent. Given the scenario-feature mapping, the set of **feature-specific concepts** for a feature f can be defined as:

$$c_f = (X, Y) \;\; where \;\; \bigcap_{a \in Y} a = \{f\}$$

The following relationships between computational units and features can then be derived:

SPEC (Specific computational units): All computational units o for which $\gamma(o) = c_f$ holds.

RLVT (Relevant computational units): All computational units o for which $\gamma(o) = c'$ and $c' < c_f$ holds.

CSPC (Conditionally specific computational units): All computational units o for which $\gamma(o) = c'$ and $c_f < c'$ holds.

SHRD (Shared computational units): All computational units o for which $\gamma(o)$ and c_f are not comparable, but o is in the extent of a concept c' such that $c_f < c'$ holds.

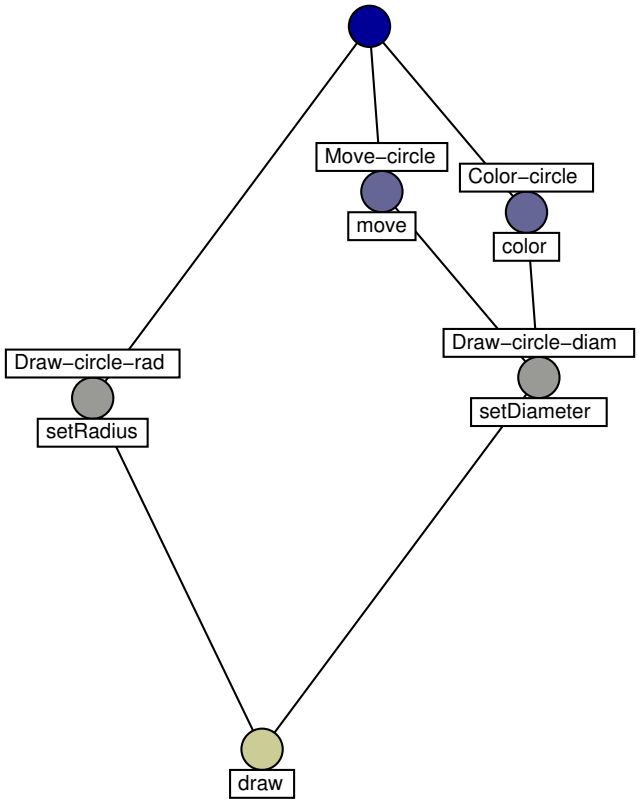
IRLVT (Irrelevant computational units): All computational units not belonging to any of the categories above.

Example of context:

	Move-circle	Draw-circle-diam	Draw-circle-rad	Color-circle
move	×			
setDiameter	×	×		×
setRadius			×	
draw	×	×	×	×
color				×

Features:

	Move-circle	Draw-circle-diam	Draw-circle-rad	Color-circle
draw-circle	×	×	×	×
color-circle				×
move-circle	×			



The three concepts at the bottom contain the computational units specific to *draw-circle*. No feature-specific concept exists for *color-circle* and *move-circle*. It is possible to isolate such features in the concept lattice by adding more discriminating scenarios, such as *Load-circle-and-color* = {*load-circle, color-circle*} and *Load-circle-and-move* = {*load-circle, move-circle*}.

References

(Kozaczynski'1992) V. Kozaczynski and J. Q. Ning and A. Engberts. *Program concept recognition and transformation.* In IEEE Transactions on Software Engineering, vol. 18, n. 12, pp. 1065-1075, December 1992.

(Rich'1990) C. Rich and L. Wills. *Recognizing a Program's Design: A Graph Parsing Approach.* In IEEE Software, vol. 7, n. 1, pp. 82-89, January 1990.

(Paul'1994) S. Paul and A. Prakash. *A Framework for Source Code Search Using Program Patterns.* In IEEE Transactions on Software Engineering, vol. 20, n. 6, pp. 463-475, June 1994.

(Murphy'2001) G. C. Murphy, D. Notkin, and K. Sullivan. *Software Reflexion Models: Bridging the Gap Between Design and Implementation.* In IEEE Transactions on Software Engineering, vol. 27, n. 4, pp. 364-380, April 2001.

(Wilde'1995) N. Wilde. *Software Reconnaissance: Mapping Program Features to Code.* In Journal of Software Maintenance, vol. 7, pp. 49-62, 1995.

(Eisenbarth'2003) T. Eisenbarth, R. Koschke and D. Simon. *Locating Features in Source Code.* In IEEE Transactions on Software Engineering, vol. 29, n. 3, pp. 195-209, March 2003.

Chapter 5

Restructuring

5.1 Clone detection

The typical scenario leading to a phenomenon called *software cloning* involves the following steps:

- When a component is modified to add a functionality or remove a defect, a different component similar to the needed one is identified.
- The new functionality is sufficiently different from the existing one, so that it is not possible to reuse it as it is or with minor modifications.
- The analysis of the current usage of the existing component and of the impact of a potential change aimed at reusing it is lengthy and difficult, and substantial regression testing would be required to ensure that the previous functionality is not adversly affected by the change.
- To meet the deadline, instead of changing and reusing the existing component, a copy is created (via "copy-and-paste"), names are changed to avoid conflicts and modifications are introduced where necessary.
- Since not all internal features of the cloned component are fully understood, some of them are preserved unchanged, although not used at all ("dead code").

An *idiom* is a program fragment that implements a recognizable concept (data structure or computation). Clones often occur when idioms are not factored into separate functions, but are copied and optionally edited.

Software cloning has a number of negative effects on software maintenance:

- If a defect is detected in a component, all its clones should be repaired as well. If some update is not performed, the same defect will be still present in the system and may result in similar failures during successive executions.
- Unused code is inserted into the system.
- Errors in renaming the cloned items may lead to unintended aliases, which could originate new errors.
- The code size grows quickly, making the system increasingly difficult to maintain. The overall result is a sort of "software aging", and its main consequence is that performing even small design changes becomes a hard task, due to the possibility of undesired ripple effects.

5.1.1 Clone detection by substring matching

The approach to clone detection based on substring matching aims at determining (maximal) sequences of lines of code that are the same (or very similar).

Maximal exact match: two sections of code are said to be a *maximal exact match* if their lines match exactly character by character but the preceding lines and the following lines do not match.

The steps of clone detection based on substring matching can be summarized as follows:

1. Text-to-text source transformation.
2. Generation of candidate substrings.
3. Identification of raw matches.
4. Simplification and presentation of results.

Text-to-text source transformation: it aims at eliminating irrelevant differences between code fragments that are potentially clones of each other.

Examples of such transformations are:

- Remove all white space characters (blank, tab, carriage return, line feed, etc.). The resulting matches will be independent from the code layout.
- Remove all white spaces except for line separators. Only layout differences internal to the code lines are considered irrelevant.
- Replace maximal sequences of white characters with a single blank.
- Remove comments.
- Retain only comments.
- Replace each identifier with an identifier marker.
- Various combinations of the above.

Generation of candidate substrings:

All or a sample of all substrings with a given length (e.g., 50 lines of code) are extracted from the transformed code. Sampling can be used to reduce the number of candidate substrings, for example based on some characters appearing before or after the considered string (e.g., only strings following '{' or preceding '}' are considered).

Identification of raw matches:

Candidate substrings are sorted. Equal (exact match) or similar (edit distance below a given threshold) substrings are considered to match.

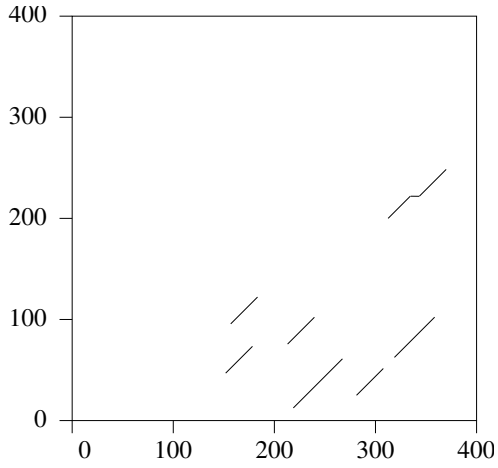
To make string comparison tractable for large systems, fingerprinting can be exploited. The content of each substring is summarized into a small sequence of bytes (e.g., hashing), which can be easily compared. If the underlying substrings do match, fingerprints will also match, while non matching substrings generate the same fingerprints with low probability.

Simplification and presentation of results:

Matching substrings may overlap. In such cases, the length of the matching substrings is extended as much as possible, so as to detect maximal (exact) matches.

Since the number of matching substrings may be overwhelming for a human in charge of reviewing them, proper presentation strategies have to be adopted. For example, clones may be ranked by length (from longer to shorter) and by number of instances found (from higher to lower), so that those that are more likely to represent relevant problems are presented earlier in the final list of candidate code clones.

Scatter plot: Maximal exact matches can be represented in a scatter plot diagram, in which diagonal segments connect matching code sections with a length above a predefined threshold (e.g., 20 lines of code).



- The length of the segments corresponds to the size of the cloned code.
- The plot can be made more readable by eliminating segments with length below a given threshold.
- Segments may be not strictly diagonal if blank lines and comments are skipped.
- Gaps between segments may be associated with code that was copied and then modified in the middle.
- Closing braces preceding a clone are reported as part of the clone, although their inclusion is not meaningful. They can be eliminated during the generation of the candidate substrings, or by a postprocessing, verifying if any of the found matches start with closing braces.

5.1.2 Clone detection by parameterized matching

This approach aims at determining maximal sections (i.e., sequences of lines) of code over a threshold length that are the same except for a global substitution of names of parameters such as variables and constants.

P-match: two sections of code are said to be a *parameterized match (p-match)* if there is a one-to-one function that maps the set of parameters in one section onto the set of parameters in the second section, such that the text of the first section is transformed into the text of the second by textually substituting all parameter occurrences according to this function.

Two sections of code are a *maximal p-match* if they are a p-match which cannot be extended to the preceding or following lines.

P-matches can also be represented in a scatter plot diagram. The scatter plot of the exact matches is contained in the scatter plot of the p-matches.

P-match detection:
P-matches can be identified by transforming each line of code into a p-string, by encoding the p-string and finally comparing it with the other encoded p-strings.

P-string: a *parameterized string (p-string)* results from concatenating the representation of the line of code with non-parameter symbols and the ordered list of the parameter symbols.

For example, the p-string of the line of code:

```
x = x + y;
```

is:

```
"P = P + P; | x, x, y"
```

Encoded p-string: a p-string is encoded by replacing the first occurence of each parameter with a 0 and each later occurrence with the distance from the previous occurrence (spaces excluded).

For example, the p-strings:

```
"P = P + P; | x, x, y"
"P = P + P; | a, a, b"
```

are both encoded as:

```
0=2+0;
```

A p-match holds between two lines of code *iff* they are associated to the same encoding of the respective p-strings.

In order to efficiently compute all maximal p-matches of a program it is possible to exploit a tree representation of the encoded p-strings called *p-suffix tree* and find repetead substrings during a tree visit.

- In the scatter plot, overlapping segments are associated with p-matches involving conflicting parameter substitutions (i.e., a different parameter assignment is performed in the two segments).

- P-matches involving a high number of parameters that have to be remapped may be difficult to factorize. Therefore, a threshold can be introduced to exclude p-matches involving too many remapped parameters (e.g., when remapped parameters are more than half the number of lines of code).

5.1.3 Clone detection using metrics

This clone detection technique works at the level of whole functions. It compares the metrics determined for the functions in the system, to infer the likelihood of a cloning relationship between them. Metrics can be considered as belonging to different categories. For the purpose of clone detection, it is convenient to classify metrics as *Layout metrics*, *Expression metrics* and *Control Flow metrics*. Examples of these metrics are:

Layout metrics	
Description	**Delta**
Number of alphanum. characters in declaration comments	10
Number of alphanum. characters in exe. section comments	10
Number of logical comments	5
Number of non-blank lines of code	5
Average variable name length	2

Expression metrics	
Description	**Delta**
Total number of call sites	5
Number of distinct invoked functions	2
Average complexity of decisions	2
Number of declaration statements	2
Number of executable statements	5

Control Flow metrics	
Description	**Delta**
Number of nodes	2
Number of arcs	2
Number of decisions	2
Average span of the branches of decisions	5
Number of arc crossings	2
Number of loops	2
Number of return points	2
Average nesting level	2
Number of independent paths	100
Number of control statements	2
Number of violations to structured programming	2

For each metric, deltas represent the maximum difference expected to occur between clones. They are defined on the basis of the distribution of the selected metrics on a proper sample of existing software.

Within each metrics category C, functions can be compared and classified as *Equal*, *Similar* or *Distinct*, according to the following definitions:

Equal $[C]$ $(f = g)$: $\forall i \in [1, ..., n] : |m_i(f) - m_i(g)| = 0$

Similar $[C]$ $(f \sim g)$: $\forall i \in [1, ..., n] : |m_i(f) - m_i(g)| \leq \delta_i$

Distinct $[C]$ $(f \neq g)$: $\exists i \in [1, ..., n] : |m_i(f) - m_i(g)| > \delta_i$

where the metrics category C is either *Layout*, *Expression* or *Control Flow*. $m_1, ..., m_n$ are the metrics computed for functions f and g within a given metrics category C, while $\delta_1, ..., \delta_n$ are the respective deltas.

Function pairs can now be ranked against an ordinal scale from the most likely to the less likely clones:

Ordinal scale of clones				
Scale	**Name**	**Layout**	**Expr.**	**Control**
1 ExactCopy	$=$	$=$	$=$	$=$
2 DistinctName	$\neq$	$=$	$=$	$=$
3 SimilarLayout	$\times$	$\sim$	$=$	$=$
4 DistinctLayout	$\times$	$\neq$	$=$	$=$
5 SimilarExpression	$\times$	$\times$	$\sim$	$=$
6 DistinctExpression	$\times$	$\times$	$\neq$	$=$
7 SimilarControlFlow	$\times$	$\times$	$\times$	$\sim$
8 DistinctControlFlow	$\times$	$\times$	$\times$	$\neq$

Clone groups:
Since the cloning relationship is transitive, groups of functions that are pairwise clones can be formed. For example, if the following level 1 clone pairs are found: $\{f_1, f_2\}, \{f_1, f_3\}, \{f_2, f_3\}$, it is possible to construct the level 1 clone group: $\{f_1, f_2, f_3\}$.

Small functions are likely to have close metrics values even if they are not clones of each other. To avoid such false positives, a filter can be applied which excludes from the analysis all functions with a size below a given threshold. They can be handled separately by means of different techniques.

Once clones have been identified, it may be convenient to remove them from the system by factoring them out into a library function. The degree of parameterization that has to be implemented in this function largely

depends on the similarity level between the clones. Factoring out a level 1 pair of functions is typically easier than for levels 2, 3 etc., since no difference or only a few differences are expected to be related to the function behavior.

It should be noted that even level 1 clones are not ensured to be a same function. In fact, the name may be coincident and all considered metrics may be equal, but the functions may behave differently or contain entities named differently, if the same summary information, represented by the metrics, is associated to them.

5.1.4 Clone detection using abstract syntax trees

This technique exploits the Abstract Syntax Tree (AST) generated by parsing the input program to detect the clones it contains. More specifically, subtrees of the AST are compared to identify recurrent instances.

Clone detection is achieved in three steps. First, the basic clone detection algorithm is applied to determine equal subtrees above a given size. Then, sequences of cloned subtrees are recognized as a unique clone. Finally, clones are generalized to near-clones by computing the similarity between subtrees containing clones.

Basic clone detection algorithm:

```
1   Clones ← ∅
2   for each subtree i of the AST
3       if nodes[i] ≥ nodeThreshold
4           hash i to bucket
5       end if
6   end for
7   for each subtree i and j in the same bucket
8       if equal(i, j)
9           for each subtree s of i ∨ j
10              if s ∈ Clones
11                  Remove(Clones, s)
12              end if
13          end for
14          Add(Clones, (i, j))
15      end if
16  end for
```

Commutative operators can be accounted for by making the function *equal* insensitive to the order of the children for nodes associated with such operators. Moreover, the hash function has to produce a value that does not depend on the order of the children for these nodes.

Sequences of cloned statements correspond to sequences of cloned subtrees, the parents of which are not necessarily clones. Therefore, the next step in clone detection is the identification of clone sequences. Maximum length sequences subsume the smaller clones they contain.

To find clone sequences, the statement sequences inside a function are represented in a *list structure*, consisting of the sequences of hash codes generated by the hash function for the subtrees in the sequence. Hashing of subsequences is then achieved by hashing the hash codes in the associated list structure.

Sequence clone detection algorithm:

```
1   Build the list structure for the subtree sequences
2   for k ← minSeqLength to maxSeqLength
3       hash all subsequences of length k to bucket
4   end for
5   for each subsequence i and j in the same bucket
6       if equal(i, j)
7           for each subtree/subsequence s of i ∨ j
8               if s ∈ Clones
9                   Remove(Clones, s)
10              end if
11          end for
12          Add(Clones, (i, j))
13      end if
14  end for
```

Example:

```
void f() {                    void g() {
   x = 0;                        y = 2;
   a = 1;                        a = 1;
   b = 2;                        b = 2;
   c = 3;                        c = 3;
   w = 4;                        i = 5;
}                             }
```

List structures:

f: tree hash codes = 121, 153, 321, 442, 701
subsequence hash code(153, 321, 442) = 1023

g: tree hash codes = 752, 153, 321, 442, 811
subsequence hash code(153, 321, 442) = 1023

In order to detect near-clones (i.e., clones that have been slightly modified by the programmer), in addition to comparing trees for equality, it is necessary to determine the degree of similarity of pairs of trees. The following similarity measure can be adopted:

$Similarity = \frac{2S}{L+2S+R}$

where S is the number of shared nodes, L is the number of different nodes in the first tree, and R is the number of different nodes in the second tree.

Near clones are determined by visiting the parent nodes of already identified clones and measuring their similarity.

Near-clone detection algorithm:

```
1  ClonesToGeneralize ← Clones
2  while ClonesToGeneralize ≠ ∅
3      (i, j) ← RemoveFirst(ClonesToGeneralize)
4      if similarity(parent(i), parent(j)) ≥ simThreshold
5          Remove(Clones, (i, j))
6          Add(Clones, (parent(i), parent(j)))
7          Add(ClonesToGeneralize, (parent(i), parent(j)))
8      end if
9  end while
```

Example: similarity(f, g) = 18/32

5.2 Migration to Object-Oriented code

Many existing systems do not incorporate object oriented features and design principles. They can be restructured so as to enforce some level of *modularization* or *object orientation*, encapsulating the implementative details, increasing the internal cohesion of the modules, and reducing the interdependencies.

A cohesive module is a collection of functions (perhaps along with data structure) having common properties. Therefore, it is possible to employ concept analysis to solve the modularization problem, by associating objects with functions. More flexibility is given in the choice of appropriate attributes. An example of attribute is the fact that a given function manipulates a data structure of the program. Concepts will correspond to highly cohesive module candidates, to be used to drive the restructuring of the code.

The steps to identify potential classes (modules) in an existing system are the following:

1. Build a context, where objects are functions and attributes are properties of those functions.

2. Construct the concept lattice.

3. Identify concept partitions. Each concept partition corresponds to a possible modularization of the input program, in which every function in the program is assigned to exactly one module.

There are a wide variety of attributes that can be used to identify classes (modules) in a program. Some possibilities are:

- Manipulation of a structured (e.g., C `struct`) data type.
- Usage of global variables.
- Data flow and slicing information.
- Information obtained from type inference.
- Disjunction, negation and other combinations of the above.

Example:

```
#define QUEUE_SIZE 10

struct stack { int *base, *sp, size; };
struct queue { struct stack *front, *back; };

struct stack* initStack(int sz) {
  struct stack *s = (struct stack*) malloc(sizeof(struct stack));
  s->base = s->sp = (int*)malloc(sz * (sizeof(int)));
  s->size = sz;
  return s;
}

struct queue* initQ() {
  struct queue* q = (struct queue*) malloc(sizeof(struct queue));
  q->front = initStack(QUEUE_SIZE);
  q->back = initStack(QUEUE_SIZE);
  return q;
}
```

```
int isEmptyStack(struct stack* s) {
  return (s->sp == s->base);
}

int isEmptyQ(struct queue* q) {
  return (isEmptyStack(q->front) && isEmptyStack(q->back));
}

void push(struct stack* s, int i) {
  *(s->sp) = i;
  s->sp++;
}

void enq(struct queue* q, int i) {
  push(q->front, i);
}

int pop(struct stack* s) {
  if (isEmptyStack(s))
     return -1;
  s->sp--;
  return (*(s->sp));
}

int deq(struct queue* q) {
  if (isEmptyQ(q))
     return -1;
  if (isEmptyStack(q->back))
     while (!isEmptyStack(q->front))
        push(q->back, pop(q->front));
  return pop(q->back);
}
```

θ_0	`initStack`
θ_1	`initQ`
θ_2	`isEmptyStack`
θ_3	`isEmptyQ`
θ_4	`push`
θ_5	`enq`
θ_6	`pop`
θ_7	`deq`

α_0	return type is `struct stack*`
α_1	return type is `struct queue*`
α_2	has argument of type `struct stack*`
α_3	has argument of type `struct queue*`
α_4	uses fields of `struct stack`
α_5	uses fields of `struct queue`

	α_0	α_1	α_2	α_3	α_4	α_5
θ_0	$\surd$				$\surd$	
θ_1		$\surd$				$\surd$
θ_2			$\surd$		$\surd$	
θ_3				$\surd$		$\surd$
θ_4			$\surd$		$\surd$	
θ_5				$\surd$		$\surd$
θ_6			$\surd$		$\surd$	
θ_7				$\surd$		$\surd$

top	$(\{\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}, \emptyset)$	universal concept
c_5	$(\{\theta_1, \theta_3, \theta_5, \theta_7\}, \{\alpha_5\})$	queue concept
c_4	$(\{\theta_0, \theta_2, \theta_4, \theta_6\}, \{\alpha_4\})$	stack concept
c_3	$(\{\theta_3, \theta_5, \theta_7\}, \{\alpha_3, \alpha_5\})$	`isEmptyQ, enq, deq`
c_2	$(\{\theta_2, \theta_4, \theta_6\}, \{\alpha_2, \alpha_4\})$	`isEmptyStack, push, pop`
c_1	$(\{\theta_1\}, \{\alpha_1, \alpha_5\})$	`initQ`
c_0	$(\{\theta_0\}, \{\alpha_0, \alpha_4\})$	`initStack`
bot	$(\emptyset, \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\})$	empty concept

P_1	$\{\text{top}\}$
P_2	$\{c_4, c_5\}$
P_3	$\{c_0, c_1, c_2, c_3\}$

```
const int QUEUE_SIZE = 10;

class stack {
private:
  int *base, *sp, size;

public:
  stack(int sz) {
    base = sp = new int[sz];
    size = sz;
  }

  int isEmpty() {
    return (sp == base);
  }

  void push(int i) {
    sp = i;
    sp++;
  }

  int pop() {
    if (isEmpty())
       return -1;
    sp--;
    return (*sp);
  }
};
```

```
class queue {
private:
  stack *front, *back;

public:
  queue() {
    front = new stack(QUEUE_SIZE);
    back = new stack(QUEUE_SIZE);
  }

  int isEmpty() {
    return (front->isEmpty() && back->isEmpty());
  }

  void enq(int i) {
    front->push(i);
  }

  int deq() {
    if (isEmpty())
       return -1;
    if (back->isEmpty())
       while (!front->isEmpty())
          back->push(front->pop());
    return back->pop();
  }
};
```

5.3 Clustering

Clustering is a key activity in reverse engineering and re-engineering to discover a better design of the system or to extract significant concepts from the code. *Clustering* consists of gathering the software entities (modules, procedures, etc.) that compose the system into meaningful (highly cohesive) and independent (loosely coupled) groups.

Three main issues are central to the clustering techniques:

Entity description: Build an abstraction of the real world in which the entities to be clustered are described according to a set of properties characterizing them.

Entity grouping: Define when a pair of entities (or, more generally, entity clusters) should be clustered together to make a cohesive unit.

Clustering algorithm: Select an algorithm to cluster the entities according to their features and to the grouping criterion.

Clustering techniques do not *discover* some hidden or unknown structure in a system, but rather *impose* a structure on the set of entities they are given. They (arbitrarily) decide to ignore some links and favour others. The structures imposed by the different algorithms have different qualities. Some are definitely useless, because they do not correspond to a reasonable view of the system. Others may be interesting in different ways.

When deciding how to group entities together, two main approaches can be followed: the direct link and the sibling link. In the *direct link* approach entities that depend on one another are put together, while in the *sibling link* approach entities are put together if they have similar features.

In the direct link approach, the coupling between two entities will be stronger if they have more links between themselves. For finer tuning, links can be weighted. Entity grouping is based on the level of coupling.

In the sibling link approach, a similarity metric is computed out of the features describing the entities. The more similar the entities, the higher the coupling between them.

Descriptive features of the entities can be classified into *formal* and *informal*, according to their dependence on the program semantics.

Examples of formal descriptive features

Type: User defined types referred to by the entity.

Var: Global variables referred to (assigned or read) by the entity.

Proc: Procedures called by the entity.

File: Files included by the entity.

Macro: Macros used by the entity.

Examples of informal descriptive features

Ident: References to words in identifiers declared or used in the entity. Identifiers can be decomposed into words according to simple word markers (e.g., the underscore, an uppercase letter preceded by a lowercase, a digit).

Cmt: References to words in comments describing the entity.

Informal features may be unreliable because they are not up to date or they do not correspond to the behavior of the software. On the other hand, they are more abstract than the formal ones, they are closer to human understanding, they are quite easy to extract, and they do not depend on the specific programming language of choice.

Similarity metrics are typically based on a description of each entity by means of a *feature vector*, i.e., a vector in which the dimensions correspond to the selected features, while the coordinate values are the number of references to such features found in the entity described.

Examples of alternative similarity metrics:

Normalized product: Normalized vector product of the feature vectors: $sim(X, Y) = X^T Y/(\|X\|\|Y\|)$

Association coefficients: Derived metrics are based on the following coefficients:
$a = \|X \cap Y\|$
$b = \|X \setminus Y\|$
$c = \|Y \setminus X\|$
$d = \|\mathcal{F} \setminus (X \cup Y\|$

Jaccard: $sim(X, Y) = a/(a + b + c)$

Simple Matching: $sim(X, Y) = (a + d)/(a + b + c + d)$

Sørensen-Dice: $sim(X, Y) = 2a/(2a + b + c)$

Distance coefficients: Entities are considered as geometrical points and the distance between these points is computed:

Taxonomic: $sim(X, Y) = 1 - \sqrt{\frac{1}{N} \sum (x_i - y_i)^2/(x_i^2 + y_i^2)}$

Camberra: $sim(X, Y) = 1 - \sqrt{\frac{1}{N} \sum |x_i - y_i|/(x_i + y_i)}$

Correlation coefficients: The linear correlation between values for all dimensions is computed:
$sim(X, Y) = 1 - \sqrt{(1 - r)/2} \qquad r = \frac{\sum x_i y_i - \frac{1}{N} \sum x_i \sum y_i}{\sqrt{(\sum x_i^2 - \frac{1}{N}(\sum x_i)^2)(\sum y_i^2 - \frac{1}{N}(\sum y_i)^2)}}$

Probabilistic coefficients: The probability that two entities are similar given their respective vectors is computed.

For each entity, the related feature vector is typically very sparse. This may raise problems, especially in presence of:

Empty description: Some entities have no references at all for the selected feature (feature vector equal to zero).

Quasi-empty description: Some entities have very few references for the selected feature (feature vector close to zero – typically, only one or two non-zero dimensions).

Entities with an empty description cannot be clustered together, although they have identical descriptions, neither they can be clustered with any other entity. It is possible to decide to treat them as singleton clusters

or (equivalently) to exclude them from clustering. They indicate that the chosen features are not sufficient to characterize all entities in the system.

Entities with quasi-empty description pose similar problems: they tend to be clustered together because their descriptions result to be very similar. They may form a core cluster which attracts all other entities, ultimately leading to a single, useless cluster. As with the empty descriptions, they indicate that the chosen features are inadequate.

The next choice is the clustering algorithm to use. One of the most widely used with the sibling link approach is called *agglomerative hierarchical clustering*. This algorithm starts from singleton clusters containing the individual entities. Then, singletons are gathered into small clusters, which are in turn gathered into larger clusters up to one final cluster containing everything. This results in a binary tree of clusters.

Variants of this algorithm are differentiated by the way they compute the distance of a new cluster from all the others. Given two clusters C_1 and C_2, containing respectively n and m elements (sub-clusters or entities), their distance is computed from the distances $d_{1,1}, ..., d_{i,j}, ..., d_{n,m}$ between their elements. The following are examples of how to carry out this computation:

Single linkage (or closest neighbor): $d = \min_{i,j}(d_{i,j})$

Complete linkage (or furthest neighbor): $d = \max_{i,j}(d_{i,j})$

Weighted average linkage: $d = \frac{1}{N}\sum d_{i,j}$

Unweighted average linkage: $d = \sum \|C_i \cup C_j\| d_{i,j} / \sum \|C_i \cup C_j\|$

Single linkage is known to give less coupled clusters, while complete linkage gives more cohesive ones (*cohesion* measures the average similarity between any two entities clustered together, while *coupling* measures the average similarity between any two entities belonging to different clusters). Unweighted and weighted average linkages stand in between, on the scale: complete, weighted, unweighted, single linkage.

Since feature vectors tend to be sparse, coupling naturally tends to be good. As a consequence, more importance is typically given to cohesion, when working with the sibling link approach.

For software remoduarization, a partition of the system instead of a hierarchy of clusters is typically desired. The partition can be obtained by cutting the hierarchy at an appropriate height and considering only the top most clusters. Successions of cuts at different hights are usually generated and assessed.

The problems that have to be overcome when designing a clustering technique are the generation of a *black hole*, in which one subsystem absorbs everything, or, at the other extreme, the generation of a *gas cloud*, in which all singleton clusters tend to remain almost unchanged until the final grouping into a single final cluster.

5.3.1 Modularity optimization

Software systems can be described as sets of components and relationships between components. Typical software *components* include classes, modules, variables, macros and structures, while common *relationships* include import, export, inherit, procedure invocation, and variable access.

The goal of clustering is optimizing the level of modularity, so that the resultant organization concurrently minimizes *coupling* (i.e., the connections between components of distinct clusters) while maximizing *cohesion* (i.e., the connections between the components of the same cluster).

Measures of cohesion and coupling:

Cohesion: $A_i = \frac{\mu_i}{N_i^2}$

Coupling: $E_{i,j} = \frac{\epsilon_{i,j}}{2N_iN_j}$

where μ_i is the number of dependencies internal to cluster i, and $\epsilon_{i,j}$ is the number of dependencies between clusters i and j. If auto-loops cannot occur, the denominator of A_i becomes $N_i(N_i - 1)$. If $i = j$, $E_{i,j} = 0$.

A_i and $E_{i,j}$ are between 0 and 1, being 0 when no dependency hold and 1 when the graph is fully connected.

The Modularization Quality(MQ), which will be used as the objective function of the optimization process, is defined as a measurement of the modularity in terms of cohesion and coupling:

$$MQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} E_{i,j}$$

If $k = 1$, $MQ = A_1$. The MQ measurement is bounded between -1 (no cohesion, maximum coupling) and 1 (no coupling, maximum cohesion). Given a dependency graph of the source code, the modularization algorithm partitions the software system so as to maximize MQ.

Given a set S that contains n elements, the number $S_{n,k}$ of distinct k-partitions (i.e., partitions consisting of k non-empty clusters) satisfies the recurrence equation:

$$S_{n,k} = \begin{cases} 1 & \text{if } k = 1 \text{ or } k = n \\ S_{n-1,k-1} + kS_{n-1,k} & \text{otherwise} \end{cases}$$

The entries $S_{n,k}$ are called *Stirling numbers* and grow exponentially with the size of S. A dependency graph with 5 nodes is associated with 52 distinct partitions, while a graph with 15 nodes is associated with 1,382,958,545 distinct partitions.

Optimal clustering algorithm:

```
1   S ← {M_1, ..., M_n}, max ← -1
2   let MDG be the graph representing the module dependencies
3   for each partition P of S
4       if MQ(P) > max
5           max ← MQ(P), P_max ← P
6       end if
7   end for
```

This algorithm is applicable to systems up to about 15 modules. Beyond that, sub-optimal techniques must be employed. Such techniques can be

based on the notion of neighboring partitions, obtained by moving modules between the clusters of the partition, so as to improve *MQ*.

A partition *NP* is a *neighbor* of a partition P if it is the same as P except for a single element that belongs to different clusters in the two partitions.

Hill-climbing clustering algorithm:

```
1   S ← {M_1, ..., M_n}
2   let MDG be the graph representing the module dependencies
3   P ← GenerateRandomPartition(S)
4   repeat
5       BNP ← BetterNeighboringPartitions(P)
6       if BNP ≠ ∅
7           P ← SelectRandomly(BNP)
8       end if
9   until P does not change
10  P_max ← P
```

Genetic clustering algorithm:

```
1   S ← {M_1, ..., M_n}
2   let MDG be the graph representing the module dependencies
3   Pop ← GeneratePopulationOfRandomPartitions(S)
4   repeat
5       SubPop ← SelectSubpopulationRandomly(Pop)
6       BNPop ← BetterNeighboringPartitions(SubPop)
7       Pop ← Replace(Pop, SubPop, BNPop)
8       Pop ← SelectRandomlyAccordingToMQ(Pop)
9   until no better partition is generated for t generations or
        all partitions in Pop converged to their maximum MQ or
        the maximum number of generations has been reached.
10  P_max ← MaximumMQPartition(Pop)
```

The procedure *SelectRandomlyAccordingToMQ* performs a random selection with replacement, using probabilities of extraction proportional to the values of MQ.

Hierarchical clustering:

When a large software system is analyzed, the number of clusters in the (sub-)optimal partition may be large. In this case, it makes sense to cluster the clusters, thus creating a hierarchy of subsystems.

The first step in the hierarchical clustering process is to apply the modularization algorithm to the dependence graph. This produces a partition of the components. A new higher-level graph can then be built by treating each cluster as a single component. If there exists at least one edge between the components in two clusters, then there is an edge between their representative nodes in the new graph. The clustering algorithm is then applied to the new graph in order to discover the next higher-level graph, and so on, until all components have coalesced into a single cluster.

The main limitation of the presented approach is that it does not take into account the *interconnection strength* of the relationships that exist between the software components, i.e., the number of dependencies holding between the components. When the interconnection strength differs significantly from module to module, clustering should weight more stronger dependencies. On the contrary, the algorithms described above weight all dependencies uniformly.

5.4 Refactoring

Refactoring is the processs of changing a software system in such a way that it does not alter the external behavior of the code, while it improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. The design of the system is improved after the code has been written. Design, rather than occurring only at the beginning of the software life cycle, occurs continuously during development. As a consequence, the risks associated with the production of a good design from the very beginning are reduced.

```
class Movie {
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;
  public static final int CHILDREN = 2;

  private String _title;
  private int _priceCode;
  ...
}

class Rental {
  private Movie _movie;
  private int _daysRented;
  ...
}

class Customer {
  private String _name;
  private Vector _rentals = new Vector();
  ...
}
```

```
class Customer {
  ...
  public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();
      switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
          thisAmount += 2;
          if (each.getDaysRented() > 2)
              thisAmount += (each.getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          thisAmount += each.getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
             thisAmount += (each.getDaysRented() - 3) * 1.5;
          break;
      }
      frequentRenterPoints++;
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRented() > 1) frequentRenterPoints++;

      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;
  }
}
```

Method `statement` is:

- not well designed – it has too many responsibilities;
- not object oriented – computation is not delegated to objects.

Yet, it works: the compiler does not signal any error and testing does not reveal any defect. The problem is with its evolution: a poorly designed system is hard to change. The compiler does not care whether the code is ugly or clean, but humans do care, and humans are going to understand the existing code and introduce modifications.

Let us consider the following examples of changes to this program:

- statements should also be printable in HTML format;
- the classification of movies changes.

A quick-and-dirty solution could be "cloning" the `statement` method into a new one, to be named `HTMLStatement`, and modify it where necessary. Consequences of such a choice are that when the charging rules change, it is necessary to fix both `statement` and `HTMLStatement`, and the two fixes should be consistent. If later the movie classification scheme changes, again any change to `statement` should be consistently reflected into `HTMLStatement`. As the rules complexity and the system size grow, it is hard to figure out where to make the changes and how to avoid mistakes.

The program is making programmer's life difficult, because it is difficult to evolve. This is where refactoring comes in: *when a feature has to be added to a program, if the code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.*

Before starting refactoring, it is important to have a solid test suite, with self-checking test cases. In fact, after refactoring the program, it is convenient to perform regression testing automatically, relying on its output to gain some confidence that bugs have not been introduced.

The first operation on the example is the extraction of the code that computes the amount due for a given rental. This is achieved by applying the *Extract Method* refactoring.

Refactoring *Extract Method*: When a sequence of logically related statements can be grouped together, they can be turned into the body of a method, whose name should explain the isolated behavior. Referenced variables should be made available as parameters and/or return values, if not visible.

```
class Customer {
  ...
  public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      double thisAmount = amountFor(each);

      frequentRenterPoints++;
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRented() > 1) frequentRenterPoints++;

      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;
  }
}
```

```
class Customer {
  ...
  private double amountFor(Rental each) {
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
          thisAmount += 2;
          if (each.getDaysRented() > 2)
              thisAmount += (each.getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          thisAmount += each.getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
             thisAmount += (each.getDaysRented() - 3) * 1.5;
          break;
    }
    return thisAmount;
  }
}
```

Some names in the extracted method are not meaningful. Good code should communicate what it is doing clearly, and variable names are a key to clear code. Anybody can write code that a computer can understand. Good programmers write code that humans can understand.

Refactoring *Renaming*: If the name of an entity does not reveal its purpose, it should be changed. All references to such an entity must be changed accordingly. Moreover, conflicts with existing entities must be avoided when choosing the new name.

```
class Customer {
  ...
  private double amountFor(Rental aRental) {
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
          result += 2;
          if (aRental.getDaysRented() > 2)
              result += (aRental.getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          result += aRental.getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          result += 1.5;
          if (aRental.getDaysRented() > 3)
             result += (aRental.getDaysRented() - 3) * 1.5;
          break;
    }
    return result;
  }
}
```

The new method, `amountFor`, uses only information from class `Rental` but belongs to class `Customer`. It is appropriate to move it to the former class.

Refactoring *Move Method*: If a method is, or will be, using or used by more features of another class than the class in which it is defined, a new method with a similar body can be created in the class it uses most. The old method can either be turned into a simple delegation, or it can be removed altogether.

```
class Rental {
  ...
  private double getCharge() {
    double result = 0;
    switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
          result += 2;
          if (getDaysRented() > 2)
              result += (getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          result += getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          result += 1.5;
          if (getDaysRented() > 3)
             result += (getDaysRented() - 3) * 1.5;
          break;
    }
    return result;
  }
}

class Customer {
  ...
  public String statement() {
    ...
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      double thisAmount = each.getCharge();
    ...
}
```

Temporary variables can be seen only in the context of the method in which they are used. As a consequence, they tend to encourage longer methods, since the method body is the only place where the value of the local variable can be accessed. By replacing a temporary variable with a query method, any method in the class can reach the related information.

In the working example, the local variable `thisAmount` is redundant and can be replaced with a call to method `getCharge` from class `Rental`. Of course, there is a performance price to pay, but it is possible to optimize the query inside class `Rental`, if really necessary. In general, such optimizations can be performed more effectively when the code is properly factored.

Refactoring *Replace Temporary Variable with Query*: When a temporary variable is used to hold the result of an expression, it is possible to extract the expression into a query method. All references to the temporary variable can be replaced with invocations to the query method. Moreover, the new method can be used in other methods.

```
class Customer {
  ...
  public String statement() {
    ...
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints++;
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRented() > 1) frequentRenterPoints++;

      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(each.getCharge()) + "\n";
      totalAmount += each.getCharge();
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;
  }
}
```

The computation of the frequent renter points can be refactored similarly to the computation of the charge for a rental. This can be achieved by applying the *Move Method* refactoring to the related code fragment.

```
class Customer {
  ...
  public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();
      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(each.getCharge()) + "\n";
      totalAmount += each.getCharge();
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;
  }
}

class Rental {
  ...
  public int getFrequentRenterPoints() {
    if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && getDaysRented() > 1)
       return 2;
    else
       return 1;
  }
}
```

The two temporary variables `totalAmount` and `frequentRenterPoints` can be replaced with query methods, but in this case the computation performed is more complicated than for `getCharge`. Nevertheless, it is worth refactoring the local variables into methods, so that the design is cleaned up and the body of the methods is shortened. Moreover, since the totals stored in `totalAmount` and `frequentRenterPoints` are necessary also to the `HTMLStatement` method to be developed, factoring them into separate methods avoids the creation of potentially dangerous clones.

```
class Customer {
  ...
  public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(each.getCharge()) + "\n";
    }
    result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " + String.valueOf(getTotalFrequentRenterPoints()) +
              " frequent renter points";
    return result;
  }

  public double getTotalCharge() {
    double result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getCharge();
    }
    return result;
  }
}
```

```
class Customer {
  ...
  public int getTotalFrequentRenterPoints() {
    int result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getFrequentRenterPoints();
    }
    return result;
  }
}
```

Most refactorings reduce the amount of code, while the previous one increased it. This is because some fixed statements are needed to set up a summing loop. It is an *idiom*, obvious to any programmer, which therefore does not affect the program comprehension difficulties.

The other concern may be about performances. In the old code the "while" loop was executed once, the new code executes it three times. Until the profiler is run, it is not possible to say if the time spent in the loop and the frequency of execution of the loop are such as to affect the overall performances of the system. When optimization is taken into consideration and the profiler is executed, decisions will be made on how to optimize methods. Having refactored the system typically simplifies such interventions.

Now, queries for the totals are available to any user code written in the `Customer` class and, being *public*, also to any other part of the system needing this information. This makes user methods simpler and concentrates mantenance interventions to these two new methods. It is therefore possible to write the new method `HTMLStatement` without creating any undesirable code clones.

```
class Customer {
  ...
  public String HTMLStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + getName() +
                    "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getMovie().getTitle() + ": " +
                String.valueOf(each.getCharge()) + "<BR>\n";
    }
    result += "<P>You owe <EM>" +
              String.valueOf(getTotalCharge()) + "</EM><P>\n";
    result += "You earned <EM>" +
              String.valueOf(getTotalFrequentRenterPoints()) +
              "</EM> frequent renter points<P>";
    return result;
  }
}
```

It is now easy to prepare any alternative kind of statement. Since calculations have been separated from the formatting of the statement, their maintenance remains local to the classes containing them and unexpected ripple effects or missed updates cannot occur.

Let us now consider some refactorings that can be applied to anticipate any change in the classification scheme of the movies. At the moment, making such changes is awkward. The logic of the code of `getCharge` and `getFrequentRenterPoints` has to be modified according to the new film classification. The difficulty with this modification is related to the "switch" statement of `getCharge` and to the "if" statement of `getFrequentRenterPoints`. It is not convenient to do a switch based on a type attribute of an object (the movie type), because this introduces a strong dependence on the persistence of such an attribute. When it changes, the conditionals are necessarily impacted by the change.

Polymorphism allows avoiding an explicit conditional when the behavior of an object depends on its type. If conditional code is present, each time a new type is added, all conditionals sparsed in the code have to be found

and updated. On the contrary, if conditional code is replaced with polymorphism, it is sufficient creating a new subclass and providing the appropriate methods. Clients of a class don't need to know about the subclasses, thus reducing the dependencies in the system and simplifying its update. Their code continues to work untouched.

To replace the conditional logic with polymorphism in the working example, the `getCharge` and `getFrequentRenterPoints` are first moved to the class containg the type attribute used in the conditionals, i.e., the `Movie` class.

```
class Rental {
  ...
  public double getCharge() {
    return _movie.getCharge(_daysRented);
  }

  public int getFrequentRenterPoints() {
    return _movie.getFrequentRenterPoints(_daysRented);
  }
}
```

```
class Movie {
  ...
  public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case REGULAR:
          result += 2;
          if (daysRented > 2)
              result += (daysRented - 2) * 1.5;
          break;
        case NEW_RELEASE:
          result += daysRented * 3;
          break;
        case CHILDREN:
          result += 1.5;
          if (daysRented > 3)
             result += (daysRented - 3) * 1.5;
          break;
    }
    return result;
  }

  public int getFrequentRenterPoints(int daysRented) {
    if ((getPriceCode() == NEW_RELEASE) && daysRented > 1)
       return 2;
    else
       return 1;
  }
}
```

A naive way to replace conditionals with polymorphism consists of introducing three subclasses of `Movie`, namely `RegularMovie`, `ChildrenMovie` and `NewReleaseMovie`. This does not work in our case, since while a movie can change its classification over its lifetime, an object cannot. The solution is adopting the *State* design pattern. Each movie can be in a state in which it is classified either as a *new release*, or as a *children movie*, or as a *regular movie*. Modeling such a state as a separate object allows changing it over time. Since in our case, the state characterizes the price of the film, it can be implemented by means of three subclasses of class `Price`,

namely `RegularPrice`, `ChildrenPrice` and `NewReleasePrice`. Each movie will be associated with its price, which in turn can be *regular*, *children* or *new release*. To accomplish this transformation, the *Replace Type Code with State/Strategy* refactoring can be used.

Refactoring *Replace Type Code with State/Strategy*: When a type code affects the behavior of a class, but subclassing cannot be used, the type code can be replaced with a state object.

```
abstract class Price {

  abstract int getPriceCode();

  public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case Movie.REGULAR:
          result += 2;
          if (daysRented > 2)
              result += (daysRented - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          result += daysRented * 3;
          break;
        case Movie.CHILDREN:
          result += 1.5;
          if (daysRented > 3)
             result += (daysRented - 3) * 1.5;
          break;
    }
    return result;
  }
}
class ChildrenPrice extends Price {
  int getPriceCode() {
     return Movie.CHILDREN;
  }
}
```

```
class NewReleasePrice extends Price {
  int getPriceCode() {
     return Movie.NEW_RELEASE;
  }
}
class RegularPrice extends Price {
  int getPriceCode() {
     return Movie.REGULAR;
  }
}

class Movie {
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;
  public static final int CHILDREN = 2;

  private String _title;
  private Price _price;

  public void setPrice(int priceCode) {
    switch (priceCode) {
        case REGULAR:
          _price = new RegularPrice();
          break;
        case NEW_RELEASE:
          _price = new NewReleasePrice();
          break;
        case CHILDREN:
          _price = new ChildrenPrice();
          break;
    }
  }

  public double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
  }
}
```

Now it is possible to replace the conditionals with polymorphism.

Refactoring *Replace Conditional with Polymorphism*: When a conditional statement chooses a different behavior depending on the type of an object, each leg of the conditional can be turned into an overriding method of a subclass associated with the object type. The original method becomes abstract.

```
abstract class Price {
  abstract double getCharge(int daysRented);
}
class ChildrenPrice extends Price {
  public double getCharge(int daysRented) {
    double result = 1.5;
    if (daysRented > 3)
       result += (daysRented - 3) * 1.5;
    return result;
  }
}
class NewReleasePrice extends Price {
  public double getCharge(int daysRented) {
    return daysRented * 3;
  }
}
class RegularPrice extends Price {
  public double getCharge(int daysRented) {
    double result = 2;
    if (daysRented > 2)
       result += (daysRented - 2) * 1.5;
    return result;
  }
}
```

The same sequence of refactorings can be applied to `getFrequentRenterPoints` from class `Movie`.

```
abstract class Price {
  abstract double getCharge(int daysRented);
  public int getFrequentRenterPoints(int daysRented) {
      return 1;
  }
}
class NewReleasePrice extends Price {
  ...
  public int getFrequentRenterPoints(int daysRented) {
    return (daysRented > 1) ? 2 : 1;
  }
}

class Movie {
  ...
  public double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
  }
  public int getFrequentRenterPoints(int daysRented) {
    return _price.getFrequentRenterPoints(daysRented);
  }
}
```

Changing any price's behavior, adding new prices, or adding extra price-dependent behavior, have become much easier tasks. In a more complex application, this would make a big difference, affecting positively the system's maintainability and evolvability. Refactoring led to better-distributed responsibilities and code that is easier to change. The original code was step by step migrated from being procedure-oriented to being truly object-oriented.

5.5 Code transformation

Automatic program transformations can be employed to restructure or reengineer existing systems, with the purpose of improving their syntactic appearence, their architecture or to move from a programming language/paradigm to another one.

5.5.1 Rewrite rules

A *term* is either a variable, a constant, a tuple of one or more terms, or an application of a function symbol to one or more terms, as summarized by the following grammar:

$t ::= x \mid c \mid (t_1, ..., t_n) \mid f(t_1, ..., t_n)$

A *rewrite rule* has the form: $l \rightarrow r$, where l and r are terms, respectively called *redex* (reducible expression) and *reduct*. The reduct may also be $\uparrow$, which denotes failure.

Examples:

`Mem1:` $\texttt{Member}(x, \texttt{Nil}) \rightarrow \texttt{False}$
`Mem2:` $\texttt{Member}(x, \texttt{Cons}(x, ys)) \rightarrow \texttt{True}$
`Mem3:` $\texttt{Member}(x, \texttt{Cons}(y, ys)) \rightarrow \texttt{Member}(x, ys)$

A rewrite rule specifies a single step transformation of a term:

`Member(A, Cons(B, Cons(A, Nil)))` $\xrightarrow{\text{Mem3}}$ `Member(A, Cons(A, Nil))`

Operational semantics of the rewrite rule $l \rightarrow r$:

$$t \rightarrow t' \quad if \;\; \exists \sigma : \sigma(l) = t \wedge \sigma(r) = t'$$
$$t \rightarrow \uparrow \quad if \;\; \nexists \sigma : \sigma(l) = t$$

In order for a rule to be applicable, a mapping σ from variables to terms, called *substitution*, must exist, such that t and t' are obtained by mapping l and r with σ.

Given a *rewriting system* (i.e., a set of rewrite rules), a *rewrite sequence* $t_1 \rightarrow t_2 \ ... \rightarrow t_k$ can be obtained by sequentially applying a rewrite rule to the result of the previous rewrite step. An arbitrary length rewrite sequence is indicated as $t \stackrel{*}{\rightarrow} t'$.

A term t is in *normal form* if there is no t' such that $t \rightarrow t'$, according to some rewrite rule.

A rewriting system is *confluent* if for all t, we have that $t \stackrel{*}{\rightarrow} t_1$ and $t \stackrel{*}{\rightarrow} t_2$ implies that there exists a term t' such that $t_1 \stackrel{*}{\rightarrow} t'$ and $t_2 \stackrel{*}{\rightarrow} t'$. In other words, the transitive closure of the single-step transformations leads always to a unique final term in normal form. The associated graph, representing the transitive closure of the single-step transformations, is called the *reduction graph*.

A rewriting system is *terminating* if there are no infinite rewrite sequences. Confluence and termination are generally undecidable properties.

Example:

```
        Member(A, Cons(B, Cons(C, Nil)))
Mem3 ⟶  Member(A, Cons(C, Nil))
Mem3 ⟶  Member(A, Nil)
Mem1 ⟶  False
```

5.5.2 Strategies

A *strategy* is an algorithm for selecting a path in the reduction graph. If a rewriting system is terminating and confluent, the choice of a particular strategy does not affect the final result of the transformation – but of course affects the efficiency of the computation.

Rewrite rules are atomic strategies that describe a path of length one. They can be combined into more complex strategies by means of the following basic strategy *combinators*:

Sequential composition:

$$\frac{t \xrightarrow{s_1} t' \qquad t' \xrightarrow{s_2} t''}{t \xrightarrow{s_1 ; s_2} t''}$$

Non-deterministic choice:

$$\frac{t \xrightarrow{s_1} t'}{t \xrightarrow{s_1 + s_2} t'} \qquad \frac{t \xrightarrow{s_2} t'}{t \xrightarrow{s_1 + s_2} t'}$$

Deterministic *or* left choice:

$$\frac{t \xrightarrow{s_1} t'}{t \xrightarrow{s_1 \leftarrow\!\!+ s_2} t'} \qquad \frac{t \xrightarrow{s_1} \uparrow \qquad t \xrightarrow{s_2} t'}{t \xrightarrow{s_1 \leftarrow\!\!+ s_2} t'}$$

Recursion:

$$\frac{t \xrightarrow{s[x:=\mu x(s)]} t'}{t \xrightarrow{\mu x(s)} t'}$$

Examples:

`(Mem1 + Mem2)` $\leftarrow\!\!+$ `Mem3`
μx`((Mem1 + Mem2)` $\leftarrow\!\!+$ `(Mem3;` x`))`

Strategy definitions are introduced to name common patterns of strategies. A *strategy definition* $\varphi(x_1, ..., x_n) = s$ introduces a new n-ary operator which, once applied to n strategies $s_1, ..., s_n$, instantiates the body s of the definition with them: $s[x_1 := s_1, ..., x_n := s_n]$.

Examples:

`repeat`$(s) = \mu x((s; x) \leftarrow\!\!+ \epsilon)$
`repeat1`$(s) = s$; `repeat`(s)

where ϵ is the *identity* strategy, which always succeeds without changing the term it is applied to.

Backtracking: the non-deterministic choice operator randomly chooses one strategy and, if that fails, it backtracks and attempts to apply the other one. However, backtracking is local. When the first strategy succeeds, the second will never be attempted. The deterministic choice also backtracks locally.

5.5.3 Tree traversal

The rewrite rules presented so far can only be applied to the root of the tree representing a program, since their left part has to match the whole term to which they are applied. To be able to apply transformations at arbitrary depth in a term, the following tree traversal operators are introduced:

Indexed child:

$$\frac{t_i \xrightarrow{s} t_i'}{f(t_1, ..., t_i, ..., t_n) \xrightarrow{i(s)} f(t_1, ..., t_i', ..., t_n)}$$

Congruence:

$$\frac{t_1 \xrightarrow{s_1} t_1' \ \ ... \ \ t_n \xrightarrow{s_n} t_n'}{f(t_1, ..., t_n) \xrightarrow{f(s_1,...,s_n)} f(t_1', ..., t_n')}$$

One child:

$$\frac{t_i \xrightarrow{s} t_i'}{f(t_1, ..., t_i, ..., t_n) \xrightarrow{\Diamond(s)} f(t_1, ..., t_i', ..., t_n)}$$

All children:

$$\frac{t_1 \xrightarrow{s} t_1' \ \ ... \ \ t_n \xrightarrow{s} t_n'}{f(t_1, ..., t_n) \xrightarrow{\Box(s)} f(t_1', ..., t_n')}$$

Some children:

$$\frac{\exists j \forall i : i, j \in \{1, ..., n\} \wedge P(t_i) = \begin{cases} t_i' & \text{if } t_i \xrightarrow{s} t_i' \\ t_i & \text{if } t_i \xrightarrow{s} \uparrow \wedge i \neq j \end{cases}}{f(t_1, ..., t_n) \xrightarrow{\boxtimes(s)} f(P(t_1), ..., P(t_n))}$$

Operators *one child* and *some children* choose non-deterministically the contained term(s) to which the transformation(s) is(are) applied.

Examples:

$\texttt{Cnc1}$: $\texttt{Conc}(\texttt{Nil}, xs) \to xs$
$\texttt{Cnc2}$: $\texttt{Conc}(\texttt{Cons}(x, xs), ys) \to \texttt{Cons}(x, \texttt{Conc}(xs, ys))$

$$\begin{array}{ll} & \texttt{Conc(Cons(1, Nil), Cons(2, Nil))} \\ \xrightarrow{\text{Cnc2}} & \texttt{Cons(1, Conc(Nil, Cons(2, Nil)))} \\ \xrightarrow{\text{2(Cnc1)}} & \texttt{Cons(1, Cons(2, Nil))} \end{array}$$

$\texttt{Cons}(\epsilon, \texttt{Cnc1})$ is equivalent to $\texttt{2(Cnc1)}$.

$\texttt{Cnc3}$: $\mu x(\texttt{Cnc1} + (\texttt{Cnc2}; \texttt{Cons}(\epsilon, x)))$

$$\begin{array}{ll} & \texttt{Conc(Cons(1, Cons(2, Nil)), 3)} \\ \xrightarrow{\text{Cnc3}} & \texttt{Cons(1, Cons(2, 3))} \end{array}$$

$\texttt{map}(s) = \mu x(\texttt{Nil} + \texttt{Cons}(s, x))$

$\texttt{topdown}(s) = \mu x(s; \Box(x))$
$\texttt{bottomup}(s) = \mu x(\Box(x); s)$
$\texttt{downup}(s) = \mu x(s; \Box(x); s)$

$\texttt{topdown}((\texttt{Cnc1} + \texttt{Cnc2}) \leftarrow\!\!\!+ \epsilon)$

$\texttt{oncetd}(s) = \mu x(s \leftarrow\!\!\!+ \Diamond(x))$
$\texttt{oncebu}(s) = \mu x(\Diamond(x) \leftarrow\!\!\!+ s)$

$\texttt{oncetd}(\texttt{Cnc1} + \texttt{Cnc2})$

$\texttt{somebu}(s) = \mu x(⧈(x) \leftarrow\!\!\!+ s)$
$\texttt{manybu}(s) = \mu x((⧈(x); (s \leftarrow\!\!\!+ \epsilon)) \leftarrow\!\!\!+ s)$

$\texttt{sometd}(s) = \mu x(s \leftarrow\!\!\!+ \square(x))$
$\texttt{manytd}(s) = \mu x((s; \square(x \leftarrow\!\!\!+ \epsilon)) \leftarrow\!\!\!+ \square(x))$

$\texttt{reduce}(s) = \texttt{repeat}(\mu x(\diamond(x) + s))$
$\texttt{outermost}(s) = \texttt{repeat}(\texttt{oncetd}(s))$
$\texttt{innermost}(s) = \texttt{repeat}(\texttt{oncebu}(s))$
$\texttt{innermost'}(s) = \mu x(\square(x); ((s; x) \leftarrow\!\!\!+ \epsilon))$

5.5.4 Advanced strategies

For real code transformation applications, additional features are required, such as conditions, contexts, variable renaming, list matching, and matching modulo associativity and commutativity. To achieve this, it is convenient to break down rewrite rules into their primitives: *match* and *build*. An atomic rewrite rule $l \rightarrow r$ is turned into the sequence of these two primitives: $\texttt{match}(l)$; $\texttt{build}(r)$:

$$t : \mathcal{E} \overset{\text{match(l)}}{\longrightarrow} t : \mathcal{E}' \quad if \;\; \mathcal{E}' \sqsupseteq_l \mathcal{E} \wedge \mathcal{E}'(l) = t$$

$$t : \mathcal{E} \overset{\text{build(r)}}{\longrightarrow} t' : \mathcal{E} \quad if \;\; vars(r) \subseteq dom(\mathcal{E}) \wedge t' = \mathcal{E}(r)$$

Given a term t and the left hand side of a rewrite rule l, a *match* occurs between l and t if an environment $\mathcal{E}'$, mapping variables to ground terms, exists such that $\mathcal{E}'(l) = t$. Moreover, $\mathcal{E}'$ is required to extend the initial environment $\mathcal{E}$ (notation $\mathcal{E}' \sqsupseteq_l \mathcal{E}$) with the additional variables (if any) in l.

Given a term t and the right hand side of a rewrite rule r, the term obtained by applying the *build* primitive is $\mathcal{E}(r)$, where the environment $\mathcal{E}$ maps all variables in r to ground terms. Such an environment is the result of a *match*, when the sequence $\texttt{match}(l)$; $\texttt{build}(r)$ is executed. Transformations can now be rewritten in terms of the environment produced by *match* and used by *build*: $t : \mathcal{E} \overset{s}{\rightarrow} t' : \mathcal{E}'$.

Example:

$\texttt{in} = \texttt{match}((x, y))$; $\texttt{test}$ ($\texttt{build}(y)$; $\texttt{oncetd}(\texttt{match}(x))$)

where $\texttt{test}(s)$ succeeds if $t \xrightarrow{s} t'$, but does not change t.

Conditional rules:

$$\frac{t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'}{t : \mathcal{E} \xrightarrow{\text{where } s} t : \mathcal{E}'}$$

$l \to r \quad where\ \ s$

corresponds to the strategy:

$\texttt{match}(l)$; $\texttt{where}(s)$; $\texttt{build}(r)$

Example:

$\texttt{Dead1:}\quad \texttt{Let}(\texttt{Vdec}(x, t, e_1),\, e_2) \to e_2 \qquad$ where $\neg(\langle \texttt{in} \rangle(\texttt{Var}(x),\, e_2))$

with $\langle s \rangle t$ a shorthand for $\texttt{build}(t)$; s.

Contexts:

Context-dependent rules match a top pattern and some inner patterns which occur in *contexts*.

$$l[c[l']] \to r[c[r'](\varphi)]$$

The notation $t[t']$ indicates a term t with an occurrence of a subterm t'. The strategy operator φ that is specified in conjunction with the contexts indicates the strategy to be used for the traversal.

Example:

`Sel1:` $\texttt{Let}(\texttt{Vdec}(x, t, ses),\ e[\texttt{Select}(i, x)]) \rightarrow$
$\texttt{Let}(\texttt{Vdec}(x, t, ses),\ e[\langle\texttt{index}\rangle(i, ses)](\texttt{sometd}))$

where the strategy `index` gives the i-th element of the list ses.

References

(Johnson 1994) J. H. Johnson. *Substring Matching for Clone Detection and Change Tracking.* In Proc. of the International Conference on Software Maintenance (ICSM'94), pp. 120-126, Victoria, British Columbia, Canada, September 1994.

(Baker 1995) B. S. Baker. *On Finding Duplication and Near-Duplication in Large Software Systems.* In Proc. of the 2nd Working Conference on Reverse Engineering (WCRE'95), pp. 86-95, Toronto, Ontario, Canada, July 1995.

(Mayrand 1996) J. Mayrand, C. Leblanc and E. Merlo. *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics.* In Proc. of the International Conference on Software Maintenance (ICSM'96), pp. 244-253, Monterey, California, November 1996.

(Baxter 1998) I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier, *Clone Detection Using Abstract Syntax Trees.* In Proc. of the International Conference on Software Maintenance (ICSM'98), pp. 368-377, Bethesda, Maryland, November 1998.

(Siff 1997) M. Siff and T. Reps. *Identifying Modules Via Concept Analysis.* In Proc. of the International Conference on Software Maintenance (ICSM'97), pp. 170-179, Bari, Italy, October 1997.

(Anquetil 1999) N. Anquetil and T. C. Lethbridge. *Experiments with Clustering as a Software Remodularization Method.* In Proc. of the 6th Working Conference on Reverse Engineering (WCRE'99), pp. 235-255, Atlanta, Georgia, USA, October 1999.

(Mancoridis 1998) S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen and E. R. Gansner. *Using Automatic Clustering to Produce High-Level System Organizations of Source Code.* In Proc. of the 6th International Workshop on Program Comprehension (IWPC'98), pp. 45-52, Ischia, Italy, June 1998.

(Fowler 1999) M. Fowler. *Refactoring – Improving the Design of Existing Code.* (pp. 1-52), Addison-Wesley, December 1999.

(Visser 1998) E. Visser, Z. Benaissa and A. Tolmach. *Building Program Optimizers with Rewriting Strategies.* In Proc. of the International Conference on Functional Programming (ICFP'98), Baltimore, MD, September, 1998.

Chapter 6

Testing

Black box (functional) testing: given a program P with intended function F and input domain D, a sequence of input values (test cases) is drawn from D and applied to P. The output is compared with the expected behavior indicated by F. Any deviation from the intended function is designated as a failure.

6.1 Structural testing

White box (structural) testing: it indicates a family of testing techniques based on the selection of a set of test paths through the program. The aim is reaching a given level of *coverage* (for example, selecting enough paths so as to assure that every source statement – or, alternatively, every branch – has been executed at least once).

Structural is most applicable to unit testing. It requires complete knowledge of the program's structure and therefore it is most often used by programmers to unit test their own code.

Coverage criteria for structural testing:

Statement (aka node) coverage: every statement in the program is executed at least once in some test case.

Branch (aka link) coverage: for every decision point in the program, each outgoing branch is chosen at least once in some test case.

Condition coverage: every boolean expression appearing in a decision is made both true and false in some test case.

Branch/condition coverage: every condition and every decision are made both true and false in some test case.

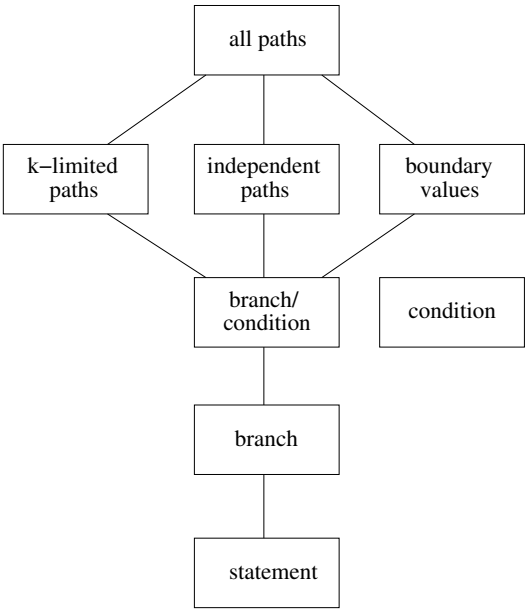
Path coverage: every path in the control flow graph is traversed at least once in some test case.

In the presence of loops, the number of paths may be very high or even infinite, thus making the path coverage criterion impractical. It is weakened by limiting the number of loop traversals, while maintaining the requirement of covering all non-looping sub-paths:

- The number of iterations for each loop is $0, ..., k$ (hence, the name of *k-limiting*). For every loop, there must exist at least a test case in which the loop is traversed $0, ..., k$ times.
- Only *independent paths* are considered: for every independent path, there must exist at least a test case traversing it.
- Loop *boundary values* are considered: for every loop which is executed at most M times, there must exist at least a test case in which the loop is traversed 0, 1, 2, $M - 1$ and M times. This reduces to 2-limiting when M is unbounded.

A typical value of k in k-limiting is 2. It is justified by the fact that some data flows, holding between two statements inside a loop, can be exercised only if the loop is traversed at least twice. This is also the rationale for the 2-traversals requirement of the boundary values restriction.

Relative strength of the coverage criteria:



While input values that allow traversing all statements, branches or conditions are always expected to exist, this is not the case for path coverage, even if restricted to a weaker variant. Some paths (called *infeasible paths*) may be associated with conditions that can never be satisfied simultaneously by any input value.

Input vector: sequence of values that the program acquires from the external environment (user input, database records, constants, data from files, etc.) along a given execution path, which are fixed before starting the execution.

The conjunction of predicates, expressed in terms of input variables (i.e., variables storing input values), whose truth values determines the traversal of a given path is called *path predicate* (aka *path condition*, *path expression*).

A path is *feasible* if there exists an input vector satisfying its path predicate. Otherwise, it is called *infeasible*.

Since the predicates evaluated along a path may contain variables that are

only indirectly dependent on the input vector, it is necessary to perform a preliminary step of *symbolic execution* (aka *predicate interpretation*) to express them in terms of the input variables. Moreover, since the same variables can be reassigned new values (for example, during successive iterations of a loop or at different statements) along the path of interest, they are indexed progressively so as to distinguish them from each other.

The basic steps to be performed when the path predicate is determined and values are symbolically propagated are:

- if an input variable x is encountered, which is assigned an input value, set its symbolic value to `*input*` and index it progressively (x_k);
- if a variable x is assigned an expression, replace each non-input variable in the expression with its symbolic value (use latest indexes);
- if a condition is encountered, join it to the path predicate after replacing non-input variables with the related symbolic values.

Examples:

```
1     scanf("%d", &x);
2     a = x + 1;
3     b = x - 1;
4     if (a < 10)
5        x++;
6     if (b > 20)
7        x--;
8     printf("%d\n", x);
```

Path: 1, 2, 3, 4, 5, 6, 7, 8

x_0	a_0	b_0	x_1	x_2
`*input*`	$x_0 + 1$	$x_0 - 1$	$x_0 + 1$	x_0

Path predicate: $x_0 < 9 \wedge x_0 > 21$

```
1     x = 1;
2     while (x > 0) {
3        scanf("%d", &x);
4        if (x > 10)
5           x++;
6        else x--;
      }
7     printf("%d\n", x);
```

Path: 1, 2, 3, 4, 5, 2, 3, 4, 6, 2, 7

x_0	x_1	x_2	x_3	x_4
1	`*input*`	$x_1 + 1$	`*input*`	$x_3 - 1$

Path predicate: $x_1 > 10 \wedge x_3 \leq 1$

Path sensitization: the computation of a solution to the constraints in the path predicate is called *path sensitization*. Given a path predicate, it is usually convenient to turn it into a normalized sum-of-products form, before trying to solve any of the sets of alternative constrains.

Example:

$(A + BC)(D + E)FGH$
$\longrightarrow \quad ADFGH + AEFGH + BCDFGH + BCEFGH$

When a test case is executed, two alternative techniques can be adopted to determine the path that is traversed:

- execution of the program from a tracer (or a debugger);
- instrumentation of the program (at each branch, a statement is added which either appends the link label to the current path string, or increments the link counter).

Data-flow testing: a family of test strategies based on the manipulation of the data performed by the program. The aim is checking the flows that go from each computation performed on every data structure to its subsequent usage at a different program point.

Coverage criteria for data-flow testing:

All def-use (aka all-*du*): every def-clear path from every definition to every use of all program variables is exercised in some test case.

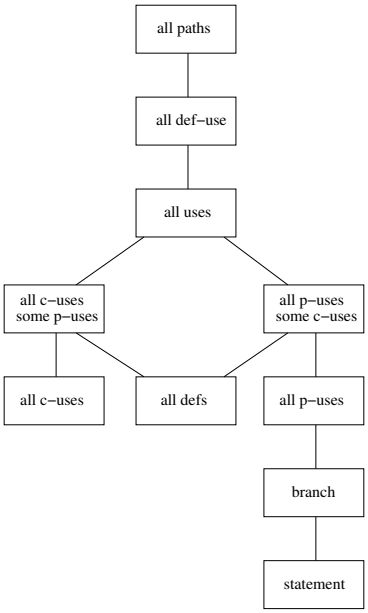
All uses: at least one def-clear path (if any exist) from every definition to every use of all program variables is exercised in some test case.

All definitions: at least one def-clear path (if any exist) from every definition to at least one use of all program variables is exercised in some test case.

All *p*-uses/some *c*-uses: at least one def-clear path (if any exist) from every definition to every *p*-use (use in a predicate) of all program variables is exercised in some test case. If there are definitions that remain not covered by the previous prescription, the coverage of at least a path from such definitions to some *c*-use (computational use) is required. Predicates in *p*-uses must evaluate both to true and false.

All *c*-uses/some *p*-uses: at least one def-clear path (if any exist) from every definition to every *c*-use of all program variables is exercised in some test case. If there are definitions that remain not covered by the previous prescription, the coverage of at least a path from such definitions to some *p*-use is required.

Relative strength of the coverage criteria:



Example:

```
main() {
1    scanf("%d", &a);          {3, 4, 7}
2    scanf("%d", &b);          {4, 8, 13}
3    if (a == 4) {
4       a = a - b;             {5, 6, 7}
5       if (a == 3)
6          a--;                {7}
     }
7    a--;                      {9, 10, 12}
8    if (b)
9       while (a)
10         a--;                {9, 10, 12}
11   else a = 2;               {12}
12   printf("%d\n", a);
13   printf("%d\n", b);
```

Statement coverage: $\{(4,1),(4,0)\}$
Branch coverage: $\{(4,1),(4,0),(1,0)\}$
All-uses coverage: $\{(4,1),(4,0),(1,0),(1,1),(3,1)\}$
The path $< 1,2,3,4,5,6,7,8,11,12,13 >$ is infeasible.

6.2 Statistical testing

Statistical testing: it is based on the adoption of a *usage model* of the software. A probability distribution is assigned to individual (or groups of) inputs from D, according to an estimated operational frequency of use for each input. Test cases are obtained by sampling input values from such a distribution, thus replicating the real world usage environment in the testing environment. As a consequence, the number of observed failures f can be used to derive a measure of the reliability of the software (R) and of the mean time between failure (MTBF):

$$R = 1 - \frac{f}{n}$$

$$MTBF = \frac{n}{f}$$

where n is the total number of test cases executed.

If discovered defects are fixed over the observation period, reliability is expected to grow over time. Such a trend can be analyzed by means of a *software reliability growth model*. If, for example, the failure arrival proces is assumed to be a non-homogeneous Poisson process, the expected cumulative number of failures, $m(t)$, over time t is given by the formula:

$$m(t) = N(1 - e^{-bt})$$

where the model constants N (total number of defects in the system) and b (model curvature) are estimated from the observation data. Of course, $R(t) = m(t)/N$.

Usage Markov chain: a usage model exploiting the Markov chains can be constructed by identifying a set of *states*, i.e., externally visible modes of operation of the software, and *state transitions*, i.e., links between states, labelled with the input causing the transition and the probability of having such an input. Transition probabilities are assumed to be uniform when no usage information is available, but may be non uniform if usage patterns are known.

Test case generation is achieved as a stochastic traversal of the Markov chain. At each state, a transition is selected according to the probabilities of the outgoing edges, until the termination state is reached. This allows estimating the reliability and MTBF of the software. Alternatively, the usage Markov chain can be exploited to exercise only the most likely execution paths, thus ensuring that only paths with very low probability are left untested in the delivered application. To achieve this, path probabilities are computed and only paths with probability above a given threshold are exercised.

Important statistical properties of the software under test can be derived from the Markov chain used to model it. Some assume a *recurrent* Markov chain, where the termination states are connected to the initial one, while others assume an *absorbing* Markov chain, where the termination states are self-looping with probability 1.

Recurrent Markov chain:

Asymptotic appearence rate of state j in a large number of sequences:

$$\pi_j = \sum_i \pi_i U_{ij}$$

Mean number of state transitions between two occurrences of state j in a large number of sequences:

$$m_{jj} = \frac{1}{\pi_j}$$

Mean number of occurrences of state i between two occurrences of state j in a large number of sequences:

$$m_{jj}\pi_i = \frac{\pi_i}{\pi_j}$$

Mean number of state transitions until state j occurs, starting from state i:

$$m_{ij} = 1 + \sum_{k \neq j} U_{ik} m_{kj}$$

Absorbing Markov chain:

Probability that state j occurs in a single sequence (from the initial state i to an absorbing state):

$$y_{ij} = U_{ij} + \sum_{k \in \tau} U_{ik} y_{kj}$$

Mean number of sequences until state j occurs:

$$h_j = \frac{1}{y_{ij}}$$

Probability that arc (j, k) occurs in a single sequence (from the initial state i to an absorbing state):

$$z_{jk} = y_{ij} U_{jk}$$

Mean number of sequences until arc (j, k) occurs:

$$h_{jk} = \frac{1}{z_{jk}}$$

Mean number of occurrences of state j in a single sequence (from the initial state i to an absorbing state):

$$s_{ij} = \sum_{k \in \tau} U_{ik} s_{kj} + \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

An additional (implicit) constraint of the equations above is that probabilities sum up to 1.

Testing Markov chain: a stochastic model of the test history, to be used as a suitable stopping point, as well as to estimate reliability and MTBF. Before any input sequence is generated and provided to P, the test history is empty and the initial testing chain T_0 is a copy of the usage chain U with all arc frequencies set to 0. When a new input sequence is applied, the existing

testing chain T_k is updated into a new one T_{k+1}. If no failure occurs, such an update consists of incrementing the arc frequencies along the path exercised by the test case. When a failure f_j occurs, a new state labelled f_j is added to T_{k+1}, with incoming and outgoing arcs having a frequency count equal to 1. The outgoing arc from f_j may either be directed to the termination state or to a normal state, where execution is resumed after the failure.

The testing chain sequence T_k tend to converge to the usage chain U, since frequencies converge to the probabilities in the long run and failures disappear due to defect fixing. To test if the two stochastic processes T_k, U, are *equivalent* (i.e., indistinguishable as sequence generators), the *log likelihoood ratio* expectation, called the *discriminant*, is measured:

$$D(\lambda_1, \lambda_2) = \lim_{n\to\infty} \frac{1}{n}[\log_2 p(d_1, d_2, ..., d_n|\lambda_1) - \log_2 p(d_1, d_2, ..., d_n|\lambda_2)]$$

where $d_1, d_2, ..., d_n$ is the sequence generated by the two processes λ_1 and λ_2. For two Markov chains U and T, the discriminant can be computed as:

$$D(U, T) = \sum_{i,j} \pi_i p_{ij} \log_2 \frac{p_{ij}}{\hat{p}_{ij}}$$

where $\hat{p}_{ij}$ is the transition probability computed from the frequencies of the testing chain T.

When the discriminant drops below some predefined threshold and experiences little change for an extended period, it is implied that additional test sequences will not significantly impact the statistics of the testing model, and testing can stop.

The probability R of a failure free realization of the testing chain is the probability that a sequence of states starting at the initial state i and terminating at the final state f does not contain a failure state:

$$R = R_{if} = \hat{p}_{if} + \sum_{j\in\tau} \hat{p}_{ij} R_{jf}$$

where τ is the set of transient, non absorbing, states (failure states are made absorbing).

The expected number of steps between two failures is the expected number of state transitions encountered between occurrences of two failure states in the testing chain:

$$M = \sum_{i \in f_1,...,f_m} v_i(\sum_{j \in u_1,...,u_n} \hat{p}_{ij}(m_j + 1))$$

where $u_1, ..., u_n$ are the usage states, $f_1, ..., f_m$ are the failure states, v_i is the conditional long run probability for failure state f_i, given that the process is in a failure state, m_j is the mean number of steps until the first failure state from j.

6.3 Mutation testing

A large number of faulty versions (*mutants*) of a program are created, by automatically altering program statements (e.g., changing an operator or a variable with another one in expressions). The quality of a test suite is assessed by measuring whether it can detect the artificial defects (*kill the mutants*). If it cannot, new test cases have to be added, to increase the fault exposing potential of the test suite, until all of the mutants are killed.

An important issue to consider in mutation testing involves the *equivalent mutant* problem. If a mutant of a program is semantically equivalent to the original program, it can never be killed by any test case. Since the problem of identifying equivalent mutants is undecidable, it is often not possible to exclude them from the mutant population. In such cases, total coverage (i.e., kill of all mutants) cannot be reached.

6.4 Regression testing

Running all of the test cases in a test suite during regression testing may require a large amount of effort (the order of several weeks for medium size programs). If the time interval allocated to regression testing is limited, only a fraction of the test suite can be executed, with the risk of missing test cases that are able to reveal defects not yet discovered. Various techniques can be employed to attack this problem:

Test case selection: the cost of regression testing is reduced by selecting a subset of the existing test suite based on information about the program, modified version and test suite.

Test suite minimization: the test suite is reduced to a minimal subset that maintains the same coverage as the original test suite with respect to a given coverage criterion.

Test case prioritization: test cases are ordered so that those with the highest priority are executed earlier, for example with the objective of achieving code coverage at the fastest possible rate, or of exercising the modules according to their propensity to fail.

6.4.1 Test case selection

Let P be a procedure or program and let P' be a modified version of P, the *selective re-test techniques* (in contrast to the *re-test all* approach) address two problems:

1. the problem of selecting tests from an existing test suite;
2. the problem of determining where additional test may be required.

Given the test suite T for P, a typical selective re-test technique proceeds as follows:

1. Select $T' \subseteq T$, a set of tests to execute on P' (*regression test selection problem*, *obsolete* tests).
2. Test P' with T', establishing P''s correctness with respect to T' (*test suite execution problem*).
3. If necessary, create T'', a set of new functional or structural tests for P' (*coverage identification problem*).
4. Test P' with T'', establishing P''s correctness with respect to T'' (*test suite execution problem*).
5. Create T''', a new test suite for P', from T, T', T'' (*test suite maintenance problem*).

Regression testing is performed either in the *preliminary phase* or in the *critical phase* (limited by the deadline for product release). Moreover, it can be conducted within a *big bang* integration process or an *incremental* integration process.

procedure Compare(n, n')
1 mark n "n'-visited"
2 **for each** edge (n, m)
3 L = the label of (n, m) – (*true*, *false* or ϵ)
4 m' = the target node of the edge (n', m') labeled L
5 **if** m is not marked "m'-visited"
6 **if not** LexicallyEquivalent(m, m')
7 $T' = T' \bigcup$ TestsOnEdge((n, m))
8 **else**
9 Compare(m, m')
10 **end if**
11 **end if**
12 **end for**

Example: **Program** `Avg`

```
1    count = 0;
2    fscanf(fileptr, "%d", &n);
3    while (!feof(fileptr)) {
4       if (n < 0) {
5           exit(1);
        } else {
6           numarray[count] = n;
7           count++;
        }
8       fscanf(fileptr, "%d", &n);
     }
9    avg = calcavg(numarray, count);
10   return avg;
```

Program `Avg'`

```
1'   count = 0;
2'   fscanf(fileptr, "%d", &n);
3'   while (!feof(fileptr)) {
4'      if (n < 0) {
5a         printf("bad input\n");
5'         exit(1);
       } else {
6'         numarray[count] = n;
       }
8'     fscanf(fileptr, "%d", &n);
     }
9'   avg = calcavg(numarray, count);
10'  return avg;
```

Test cases:
t1: empty file
t2: -1
t3: 1 2 3

$T' = \{t2, t3\}$

Example: **Program** `TwoVisits`

```
1    if (x == 0)
2        goto a;
     else
3        a: printf("1\n");
4    exit(0);
```

Program `TwoVisits'`

```
1'   if (x == 0)
2'       goto a;
     else
3'       printf("1\n");
4'   exit(0);
5'   a: printf("2\n");
```

If 3 were marked just "visited" (instead of "3'-visited"), 3 and 5' would not be compared lexically when visiting node 2.

It can be shown that under controlled regression testing, the modification-traversing tests are a superset of the fault-revealing tests. Since the test case selection algorithm described above selects all modification-traversing tests (this can be also demonstrated), it is a *safe* algorithm, i.e., it does not discard any test case that could potentially reveal a fault in the modified program P'.

Interprocedural test selection

In order to avoid comparing every pair of corresponding procedures, the interprocedural *Compare* algorithm starts at the *main* and proceeds with an intraprocedural visit of the *main*. Whenever a call is encountered, one of the following cases occur:

1. The called procedure is not marked – then, it must be visited using the *Compare* algorithm.
2. The called procedure is marked *visited* – it is not necessary to re-visit it.
3. The called procedure is marked *selects_all* – this means all test cases reaching the call node must be selected, thus *Compare* can stop (no recursive call).

A procedure is marked *selects_all* if after visiting it, the exit node is not marked "m'-visited" for some m', i.e., if every test that enters this procedure is modification-traversing on every possible path within the procedure.

Example:

```
main() {
1    s1;
2    f();
3    s2;
4    h();
5    s3;
6    g();
}
f() {
7    s4;
8    h();
9    if (x == 0)
10      f();
     else
11      g();
12   return;
}
g() {
13   s5;
}
h() {
14   s6;
}
```

If only 1' is modified, all test cases are selected without visiting *f(), g(), h()*. If only 14' is modified, all test cases are selected without visiting *g()* and without proceeding below 8 (in *f()*) and 2 (in *main()*).

6.4.2 Test case prioritization

The Test Case Prioritization Problem:
Given: T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real numbers;
Problem: find $T' \in PT$ such that $\forall T'' \in PT : f(T') \geq f(T'')$.

Function f is the objective function to be maximized by the ordering. It depends on the goal of the test case prioritization. Some possible goals are:

- Increasing the rate of fault detection of a test suite – that is, the likelihood of revealing faults earlier during regression testing.
- Achieving code coverage at a fast rate, thus allowing a code coverage criterion to be met earlier during regression test.
- Increasing the confidence in the reliability of the system under test.
- Increasing the rate at which high risk faults are detected.
- Revealing faults related to critical code changes earlier.

Depending upon the choice of f, the test case prioritization problem may be intractable or undecidable. This is the case of the function f that quantifies whether a test suite achieves statement coverage at the fastest possible rate, as well as the function f that quantifies whether a test suite detects faults at the fastest possible rate. In such cases, test case prioritization must resort to heuristics.

Let us consider the first goal listed above, i.e., improving the *rate of fault detection* of the test suite. A possible quantification of the related objective function f is the weighted Average of the Percentage of Faults Detected (APFD), given by the relative area below the linear interpolation of the cumulative percentage of detected faults.

Since the faults to be exposed by regression testing are unknown, it is impossible to determine the optimal prioritization a priori. However, a hypothetical a priori knowledge of such faults would require an exponential algorithm, in the worst case, to determine the optimal ordering of test cases maximizing the APFD, since all possible test case orderings may have to

be considered. A greedy algorithm, which iteratively selects the test case which exposes the most faults not yet exposed by a selected test case, is only sub-optimal. In fact, it may produce the non optimal sequence (t_1, t_2, t_3) on the following example:

Test case	Fault			
	1	2	3	4
t_1	X	X		
t_2	X			X
t_3		X	X	

Some heuristic techniques that can be adopted to maximize the APFD are:

Total statement coverage prioritization: test cases are prioritized by counting the number of statements covered by each test case and then sorting the test cases in descending order.

Additional statement coverage prioritization: test cases are repeatedly selected that yield the greatest coverage of statements not yet covered by previously selected test cases. When the selected test cases cover all of the statements, the selection procedure is restarted and additional statement coverage prioritization is reapplied to the remaining test cases, after resetting the coverage measure.

Total branch coverage prioritization: test cases are prioritized by counting the number of branches covered by each test case and then sorting the test cases in descending order.

Additional branch coverage prioritization: test cases are repeatedly selected that yield the greatest coverage of branches not yet covered by previously selected test cases. When the selected test cases cover all of the branches, the selection procedure is restarted and additional branch coverage prioritization is reapplied to the remaining test cases, after resetting the coverage measure.

Total Fault-Exposing-Potential (FEP) prioritization: test cases are sorted in decreasing order of Fault-Exposing-Potential (FEP) – the ability of a test case to expose a fault. Values of FEP are estimated by means of mutation analysis, as described below.

Additional Fault-Exposing-Potential (FEP) prioritization: test cases are repeatedly selected that yield the greatest increment of *confidence* – an estimate of the probability that each program statement is correct. The detailed procedure is illustrated below.

FEP estimation: Given a program P and a test suite T, a set of mutants is created for each statement s_j in P. Then, each test case $t_i \in T$ is executed on each mutant. The value of $\text{FEP}(t_i, s_j)$ per statement s_j is computed as the ratio of mutants of s_j killed by t_i. Note that if t_i does not execute s_j, this ratio is zero. A cumulative value of FEP is then assigned to each test case: $\text{FEP}(t_i) = \sum_j \text{FEP}(t_i, s_j)$.

Confidence increment estimation: The confidence $C(s_j)$ in a statement s_j is assigned a value 0 prior to executing the test suite. In other words, no confidence in the correctness of each program statement is assumed before testing it. Then, after execution of the test case t_i, if t_i exposes no fault in s_j, the increment of confidence in statement s_j is:

$$\Delta C(s_j) = (1 - C(s_j))FEP(t_i, s_j)$$

The total confidence increment is obtained as the sum over all of the statements: $\Delta C(t_i) = \sum_j \Delta C(s_j)$.

Example:

```
    main() {
1     s1;
2     while (c1) {
3        if (c2)
4           s2;
         else
5           s3;
6        s4;
      }
7     s5;
8     s6;
9     s7;
    }
```

Statement coverage			
Statement	Test case 1	Test case 2	Test case 3
1	X	X	X
2	X	X	X
3		X	X
4		X	
5			X
6			X
7	X		X
8	X		X
9	X		X

Branch coverage			
Branch	Test case 1	Test case 2	Test case 3
entry	X	X	X
2-true		X	X
2-false	X		X
3-true		X	
3-false			X

FEP(s,t) values			
Statement	Test case 1	Test case 2	Test case 3
1	.4	.5	.3
2	.5	.9	.4
3		.01	.4
4		1.0	
5			.4
6			.1
7	.5		.2
8	.6		.3
9	.3		.1
	2.3	2.41	2.2

$\Delta C(s)$ – initial values				
Statement	$C(s)$	Test case 1	Test case 2	Test case 3
1	0	.4	.5	.3
2	0	.5	.9	.4
3	0		.01	.4
4	0		1.0	
5	0			.4
6	0			.1
7	0	.5		.2
8	0	.6		.3
9	0	.3		.1
		2.3	2.41	2.2

$\Delta C(s)$ – after executing test case 2				
Statement	$C(s)$	Test case 1	Test case 2	Test case 3
1	.5	.2		.15
2	.9	.05		.04
3	.01			.4
4	1.0			
5	0			.4
6	0			.1
7	0	.5		.2
8	0	.6		.3
9	0	.3		.1
		1.65		1.69

6.5 Testing of object-oriented programs

The smallest unit of testing in object oriented programming is the class. Unit testing of a class cannot exploit the standard paradigms, since object creations and method invocations may be performed externally with respect to the class under test. Moreover, reuse of the given class in a different context may be associated to a different order of object creations and method invocations.

A possible solution consists of specifying the class in terms of states and state transitions, and building a test suite which covers all possible transitions from every possible state, verifying that the next state reached is the correct one. This is the basic idea behind **state based testing**.

State of an object: combined values of all its data members at a given point of execution.

State based testing: all transitions from each state of an object of the class under test is exercised at least once by some test case. Testing is passed if the next states reached after exercising each transition are those represented in the state diagram for the given class.

Example:

```
class CounterBase4 {
  int counter;
public:
  CounterBase4() {
     counter = 0;
  }
  void reset() {
     counter = 0;
  }
  void next() {
     if (counter < 3)
        counter++;
     else
        counter = 0;
  }
  int getValue() {
     return counter;
  }
};
```

Since the number of states increases combinatorially with the number of data members of a class, it is often convenient to consider partial states, consisting of substates. A set of state diagrams corresponding to the considered partial states will be produced and tested instead of a single, huge state diagram.

Substate of an object: value of a particular data member at a given point of execution.

Partial state: combined values of a set of substates.

Rather than associating every possible value of a data member with a substate, it may be more appropriate to introduce two types of substate values:

specific substate values: substate values carrying a special significance;

generic substate values: when a group of substate values can be considered in the same manner, since they are expected to be handled equivalently, one representative element is chosen arbitrarily.

When a data member `p` is a pointer, there are at least two values that hold special significance and should be associated to two distinct substates: `p == 0` and any `p != 0`. However, if the pointer references an object, in turn each state of the referenced object may induce a new distinct state of the given object. Such states are distinguished according to the values of the data members of the referenced object. For example, we could have the following substate values: `(p == 0)`, `(p != 0 && p->d == 0)`, `(p != 0 && p->d == 1)`, `(p != 0 && p->d > 1)`.

Example:

```
class List {
  Item *head;
  Item *tail;
public:
  void append(Item *it);
  Item* remove();
};

class Item {
  char *data;
  Item *next;
  Item *prev;
};
```

Substates associated with the data member `tail`:

S_1 : `tail == 0`
S_2 : `tail != 0 && tail->prev == 0`
S_3 : `tail != 0 && tail->prev != 0`

Transitions:

$t_0 : \rightarrow S_1$: `constructor`
$t_1 : S_1 \rightarrow S_2$: `append`
$t_2 : S_2 \rightarrow S_3$: `append`
$t_3 : S_2 \rightarrow S_1$: `remove`
$t_4 : S_3 \rightarrow S_3$: `append`

$t_5 : S_3 \rightarrow S_2$: `remove [tail->prev->prev == 0]`
$t_6 : S_3 \rightarrow S_3$: `remove [tail->prev->prev != 0]`

Since the events (messages) associated with the transitions can define and/or use varibales, the data flow testing framework can also be applied to state based testing.

Transition	DEF	USE
t_0	`tail`	
t_1	`tail`	`tail`
t_2	`tail`	`tail`
t_3	`tail`	`tail`
t_4	`tail`	`tail`
t_5	`tail`	`tail`
t_6	`tail`	`tail`

Def-use chains can be conveniently computed on the dual graph (with edges and nodes exchanged). In the example above, they are coincident with the edges in the dual graph:

(t_0, t_1)
$(t_1, t_2), (t_1, t_3)$
$(t_2, t_4), (t_2, t_5), (t_2, t_6)$
(t_3, t_1)
$(t_4, t_4), (t_4, t_5), (t_4, t_6)$
$(t_5, t_2), (t_5, t_3)$
$(t_6, t_4), (t_6, t_5), (t_6, t_6)$

6.6 Automatic test case generation

6.6.1 Path generation

Path expressions are an algebraic representation of all the paths in a graph. They can be used to generate sequences of paths which satisfy a given testing criterion. Such sequences become test cases once input values are determined (possibly through symbolic execution) for them.

Edges are given a unique identifier (lowercase letter, by convention).

Concatenation: path concatenation is represented through the product operator (e.g., *abc*). It is associative but not commutative. The exponent is used for a given number of concatenations of a same path (e.g., a^2). The *identity*, $a^0 = 1$, is the path of length 0 (associated to a graph node).

Alternative: path alternative is represented through the sum operator (e.g., $a+b+c$). It is associative and commutative. The *identity*, 0, represents the empty path (associated to an empty graph).

Distributive law: product and sum are distributive:
$a(b + c) = ab + ac$
$(a + b)c = ac + bc$.

Absorption rule: identical alternative paths are absorbed: $a + a = a$.

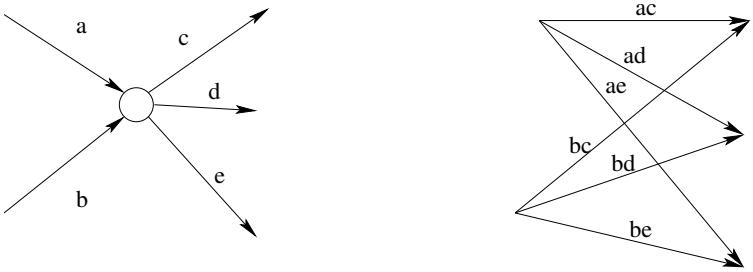
Loops: the set of all paths through a loop (consisting of the edge a) is indicated as a^*: $a^* = a^0 + a^1 + a^2 + a^3 + ...$ If a loop is traversed at least once, the associated algebraic expression is a^+. Evidently: $a^+ = aa^* = a^*a$.

Identity elements:
$1 + 1 = 1$
$1a = a1 = a$
$1^n = 1$
$a + 0 = 0 + a = a$
$a0 = 0a = 0$

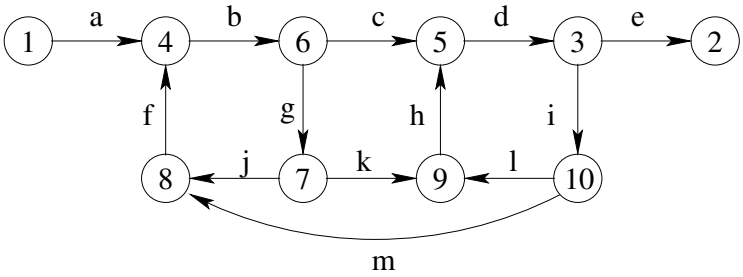
Node reduction algorithm:

```
1   combine serial links by multiplying them
2   combine parallel links by adding them
3   remove self loops X by concatenating X* to the outgoing edges
4   while number of links is greater than 1
5       select a node n
6       apply the cross-term step to n
7       combine serial links as in 1
8       combine parallel links as in 2
9       remove self loops as in 3
10  end while
```

Cross-term step:



Example:



$$a(bgjf)^*b(c+gkh)d((ilhd)^*imf(bgjf)^*b(c+gkh)d)^*(ilhd)^*e$$

Graph matrix: a graph can be represented as a matrix $A = [a_{ij}]$, the elements of which contain the label of the edge from node n_i to node n_j (0 if no edge connects n_i to n_j).

The *matrix power* A^n contains all paths between every pair of nodes of length n.

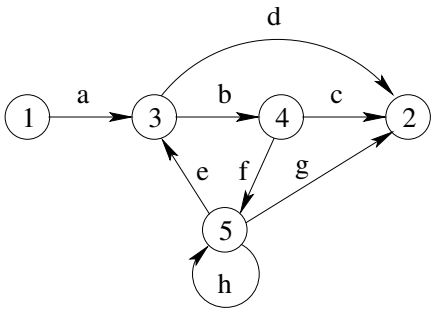
The sum $\sum_{k=1}^{n} A^k$ gives all paths between every pair of nodes of length lower than or equal to n.

Loops (diagonal elements) can be removed by pre-multiplying all elements in the same raw by the * power. For example, if $a_{ii} = h$, it is possible to replace it with 0 by setting $a_{ij} \leftarrow h^* a_{ij}, \forall j \neq i$.

Node reduction algorithm on the graph matrix:

```
1   build a graph matrix with raws 1 and 2 associated
        to the entry and exit nodes
2   while number of raws r > 2
3       if a_rr ≠ 0
4           a_rj ← a*_rr a_rj, ∀j ≠ r
5       end if
6       foreach i ∈ [1, ..., r − 1], j ∈ [1, ..., r − 1]
7           a_ij ← a_ij + a_ir a_rj
8       end foreach
9       delete raw r and column r from A
10  end while
```

Example:



$$a(bfh^*e)^*(d + bc + bfh^*g)$$

Test case generation: the path expression can be used to generate a set of paths according to a given path generation strategy. Each path is associated with a test case once input values for traversing it are determined (possibly by means of symbolic execution). Strategies for path generation specify how alternatives and loops have to be handled. For example, loops may be traversed 1 and 2 times, while the alternatives covering branches not yet considered are selected, if any is present.

6.6.2 Evolutionary testing

A population of individuals, representing the test cases, is evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a given coverage criterion. The chromosomes of the individuals being evolved consist of sequences of input values to use in test case execution.

```
testCaseGeneration(progUnderTest: Program)
1   targetsToCover ← targets(progUnderTest)
2   curPopulation ← generateRandomPopulation(popSize)
3   while targetsToCover ≠ ∅ and
                executionTime() < maxExecutionTime
4       t ← selectTarget(targetsToCover), attempts ← 0
5       while not covered(t) and attempts < maxAttempts
6           execute test cases in curPopulation
7           update targetsToCover
8           if covered(t) break
9           compute fitness[t] for test cases in curPopulation
10          extract newPopulation from curPopulation
                according to fitness[t]
11          mutate newPopulation
12          curPopulation ← newPopulation
13          attempts ← attempts + 1
14      end while
15  end while
```

With reference to the procedure *testCaseGeneration* given above, input data are generated so as to satisfy a given coverage criterion for the program under test. The set of targets (e.g., branches) to be covered is determined at line 1. Then, a set of test cases is randomly generated (*curPopulation*). Each test case (individual) consists of a sequence of input values (chromosome).

The algorithm generates new test cases as long as there are targets to be covered or a maximum execution time is reached (line 3). For each target to be covered, selected at line 4, a maximum number (*maxAttempts*) of successive generations are produced out of the initial population (loop at line 5). Test cases in the current population are executed at line 6, possibly covering some of the previously uncovered targets. The targets still to be covered are updated at line 7, and if the currently selected target (t) has been covered, the most internal loop is exited and a new target is selected. If the current target was not covered by any execution of the test cases in the current population, a measure of proximity of each test case to the target is computed (*fitness*[t], line 9). A new population is randomly extracted with replacement from the current population. Each test case has a probability of being selected which is proportional to its fitness with respect to the current target t. In other words, test cases that come closer to the target are more likely to be extracted in the formation of the new population. Then, the new population is mutated according to the mutation operators. Two widely used mutation operators are: *Change value*, which randomly changes one of the values in the input sequence, and *Crossover*, which swaps the tails of two chromosomes cut at a randomly selected middle point.

During the execution of the algorithm above, each time a previously uncovered target is covered by a test case in the current population, the test case is saved as one of those necessary to achieve the final level of coverage. The output of the algorithm is thus the set of test cases that allowed covering at least one new target. Such a set may be redundant, in that test cases that are inserted later in the resulting set may cover targets that were previously associated to a different test case. Thus, if all the targets covered by a given test case are also covered by test cases added later to the resulting set, the former does not contribute to the final level of coverage any longer and can be removed from the final set. In order to cope with such a situation, a post-processing of the set of resulting test cases is performed, aimed at minimizing it. The minimization procedure can be, for example, a simple greedy algorithm, which iteratively selects the test case that gives the largest

increase in the number of covered targets. Such a test case is added to a minimized set of test cases (initially empty), until the final coverage level is reached. All test cases not added during the post-processing are redundant and can be dropped.

The two critical choices in the evolutionary testing algorithm are the measure of fitness of each test case and the mutation operators. These are the two factors that drive the evolution of the current population into a new population that has more chances to cover the current target. The fitness determines the probability of an individual (test case) to survive and participate, in an evolved form, in the new population, while the mutation operators determine how new individuals (test cases) are generated out of the existing ones.

Assuming that the adopted coverage criterion is structural (e.g., branch coverage), the fitness of a test case with respect to a given target can be obtained from the control and call dependence edges that are traversed during its execution. Specifically, given the transitive set of all control and call dependences that lead to the given target, the proportion of such edges that is exercised during the execution of a test case measures its fitness. Thus, the fitness will be close to 1 for the test cases which traverse most of the control and call dependence edges that lead to the target, while it will be close to zero when the execution follows a path that does not intersect with edges leading to the target.

The parameters of the algorithm above that can be fine tuned externally for each program under test are the maximum execution time (*maxExecutionTime*) and the maximum number of attempts per target (*maxAttempts*), as well as the population size(*popSize*). They can be augmented when the achieved coverage level is not satisfactory.

References

(Beizer'1990) B. Beizer. *Software Testing Techniques, 2nd edition.* Chapters 3, 5, 8, 12, 13. International Thomson Computer Press, 1990.

(Whittaker'1994) J. A. Whittaker and M. G. Thomason. *A Markov Chain Model for Statistical Software Testing.* In IEEE Transactions on Software Engineering, vol. 20, n. 10, pp. 812-824, October 1994.

(Rothermel'1997) G. Rothermel, M. J. Harrold. *A Safe, Efficient Regression Test Selection Technique.* ACM Transactions on Software Engineering and Methodology, vol. 5, n. 2, pp. 173-210, 1997.

(Rothermel'2001) G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. *Prioritizing Test Cases for Regression Testing.* In IEEE Transactions on Software Engineering, vol. 27, n. 10, pp. 929-948, October 2001.

(Turner'1993) C. D. Turner and D. J. Robson. *The State-Based Testing of Object-Oriented Programs.* In proc. of the Conference on Software Maintenance, Montreal, Canada, pp. 302-310, September, 1993.

(Pargas'1999) Roy Pargas, Mary Jean Harrold and Robert Peck, *Test-Data Generation Using Genetic Algorithms.* In Journal of Software Testing, Verifications, and Reliability, vol. 9, pp. 263-282, September 1999.