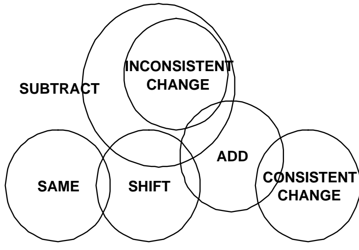

Figure 1: The relationship among evolution patterns

1. MODEL OF CLONE GENEALOGY

To study clone evolution structurally and semantically rather than quantitatively, we defined a model of clone genealogy. The genealogy of code clones describes how groups of code clones change over multiple versions of a program. In a clone's genealogy, the origin of a group to which the clone belongs is traced to the previous version. The model associates related clone groups that have originated from the same ancestor clone group. In addition, the genealogy contains information about how each element in a group of clones has changed with respect to other elements in the same group.

We wrote our model in the Alloy modeling language [?] to check whether several evolution patterns can describe all possible changes to a clone group and to clarify the relationship among evolution patterns. (Our entire model is available on the web [?].)

The basic unit in our model is a Code Snippet, which has two attributes, Text and Location. Text is an internal representation of code that a clone detector uses to compare code snippets. For example, when using CCFinder [?], text is a parametrized token sequence, whereas when using *CloneDr* [?], text is an isomorphic AST. A Location is used to trace code snippets across multiple versions of a program; thus, every code snippet in a particular version of a program has a unique location. To determine how much the text of a code snippet has changed across versions, we define a TextSimilarity function that measures the text similarity between two texts $t1$ and $t2$ ($0 \leq$ TextSimilarity$(t1, t2) \leq 1$). To trace a code snippet across versions, we define a LocationOverlapping function that measures how much two locations $l1$ and $l2$ overlap each other ($0 \leq$ LocationOverlapping$(l1, l2) \leq 1$). A Clone Group is a set of code snippets with <u>identical</u> text. $CG.text$ is a syntactic sugar for the text of any code snippet in a clone group CG. A Cloning Relationship is defined between two clone groups CG_1 and CG_2 if and only if TextSimilarity$(CG_1.text, CG_2.text) \geq sim_{th}$, where sim_{th} is a constant between 0 and 1. An Evolution Pattern is defined between an old clone group OG in the $k-1^{th}$ version and a new clone group NG in the kth version such that there exists a cloning relationship between NG and OG.

We defined several evolution patterns that describe all possible changes to a clone group. The relationship among evolution patterns is shown in the Venn diagram in Figure 1.

- Same: all code snippets in NG did not change from OG.

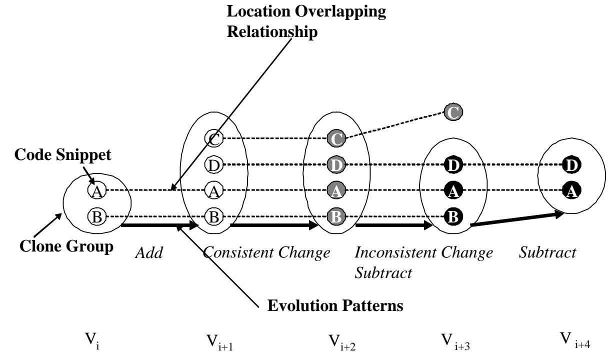

Figure 2: An example clone lineage

TextSimilarity(NG.text,OG.text) $= 1$
all cn:CodeSnippet | some co:CodeSnippet | cn in NG $\Rightarrow$
co in OG && LocationOverlapping(cn,co) $= 1$
all co:CodeSnippet | some cn:CodeSnippet | co in OG $\Rightarrow$
cn in NG && LocationOverlapping(cn,co) $= 1$

- Add: at least one code snippet in NG is a newly added one. For example, programmers added a new code snippet to NG by copying an old code snippet in OG.
TextSimilarity(NG.text,OG.text) $\geq sim_{th}$
some cn:CodeSnippet | all co:CodeSnippet | co in OG $\Rightarrow$
cn in NG && LocationOverlapping(cn,co) $= 0$

- Subtract: at least one code snippet in OG does not appear in NG. For example, programmers refactored or removed a code clone.
TextSimilarity(NG.text,OG.text) $\geq sim_{th}$
some co:CodeSnippet | all cn:CodeSnippet | cn in NG $\Rightarrow$
co in OG && LocationOverlapping(cn,co) $= 0$

- Consistent Change: all code snippets in OG have changed consistently; thus they belong to NG together. For example, programmers applied the same change consistently to all code clones in OG.
$sim_{th} \leq$TextSimilarity(NG.text,OG.text)< 1
all co:CodeSnippet | some cn:CodeSnippet | co in OG $\Rightarrow$
cn in NG && LocationOverlapping(cn,co) > 0

- Inconsistent Change: at least one code snippet in OG changed inconsistently; thus it does not belong to NG anymore. For example, a programmer forgot to change one code snippet in OG.
$sim_{th} \leq$TextSimilarity(NG.text,OG.text)< 1
some co:CodeSnippet | all cn:CodeSnippet | cn in NG $\Rightarrow$
co in OG && LocationOverlapping(cn,co) $= 0$

- Shift: at least one code snippet in NG partially overlaps with at least one code snippet in OG.[1]
TextSimilarity(NG.text,OG.text) $= 1$
some cn:CodeSnippet | some co:CodeSnippet | cn in NG
&& co in OG && ($1 >$LocationOverlapping(cn,co) > 0)

[1]This unintuitive pattern was found when we used Alloy to check whether the combination of patterns can describe all possible changes to a clone group.

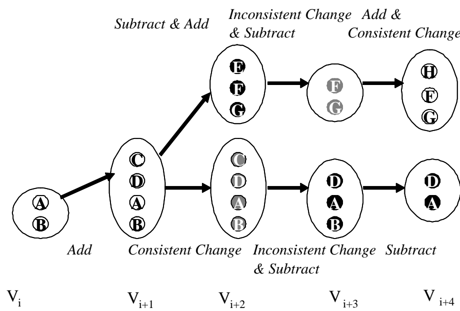

Figure 3: An example clone genealogy

A Clone Lineage is a directed acyclic graph that describes the evolution history of a sink node (clone group). In a clone lineage, a clone group in the kth version is connected by an evolution pattern from a clone group in the $k-1^{th}$ version. For example, Figure 2 shows a clone lineage including Add, Subtract, Consistent Change, and Inconsistent Change. In the figure, code snippets with the same text are filled with the same color.

A Clone Genealogy is a set of clone lineages that have originated from the same clone group. A clone genealogy is a connected component where every clone group is connected by at least one evolution pattern.[2] A clone genealogy approximates how programmers create, propagate, and evolve code clones. For example, Figure 3 shows a clone genealogy that comprises two clone lineages.

2. ALLOY CODE

```
module clonelineage
open std/ord

sig Text{}
fun similarhigh (t1:Text,t2:Text) {
  t1=t2 // exactly the same
}
fun similar (t1:Text,t2:Text) {
  #OrdPrevs(t1) = # OrdPrevs(t2) +1 ||
  #OrdPrevs(t2) = #OrdPrevs(t1)+1 // similar
}
fun notsimilar (t1:Text, t2:Text) {
! similarhigh(t1,t2) &&
  !similar(t1,t2) // not similar
}

//test functions
run similarhigh for 3
run similar for 3
run notsimilar for 3

sig Location{}
fun overlaphigh (o1:Location,o2:Location) {
  o1=o2 // exactly the same location
}
fun overlap (o1:Location,o2:Location) {
  #OrdPrevs(o1) = # OrdPrevs(o2) +1
  || #OrdPrevs(o2) = #OrdPrevs(o1)+1
  // partially overlap
}
fun notoverlap (o1:Location, o2:Location) {
  ! overlaphigh(o1,o2) &&
  !overlap(o1,o2) // does not overlap at all
}

// test functions
run overlaphigh for 3
run overlap for 3
run notoverlap for 3

// code is identified with its text and its location
sig Code{
  text: Text,
  location: Location
}
// clone group is a set of code with the same text.
// within a clone group,
// every code has a unique location.
sig Group{
  group: set Code
}{
  all c1,c2:Code |
    c1 in group && c2 in group
    => c1.text = c2.text
  # group > 1
  all disj c1,c2:Code |
    c1 in group && c2 in group
    => c1.location!=c2.location
}
// clone relationship is defined between
// one clone group in a old version and
// one clone groiup in a new version.

sig Relationship {
new : Group,
old : Group
}{
  new!=old
}

// clone genealogy is a graph which describes
// evolution of a code snippet. this graph
// is a direct graph where all nodes are
// connected by at least one edge if edges
// are present.
sig Genealogy {
nodes : set Group,
edges : set Relationship
}{
#nodes >0
#nodes>1 => (nodes = edges.new+edges.old)
}
fun testlineage () {
Genealogy= univ[Genealogy]
}
```

[2] A clone genealogy is a connected component in the sense that there exists an undirected path for every pair of clone groups. Although a clone genealogy is often an inverted tree in practice, it is a connected component in theory because the in-degree of a new clone group can be greater than one when it is ambiguous to determine the most likely origin of a new clone group.

```
// evolution patterns
fun SAME (r:Relationship){
  similarhigh(r.new.group.text,r.old.group.text)
  all csn:Code | some cso:Code |
    csn in r.new.group => cso in r.old.group &&
    overlaphigh(csn.location,cso.location)
  all cso:Code | some csn:Code |
    cso in r.old.group => csn in r.new.group &&
    overlaphigh(csn.location,cso.location)
}
//run SAME for 5

fun SHIFT (r:Relationship) {
  similarhigh(r.new.group.text,r.old.group.text)
  some csn:Code | some cso:Code |
    csn in r.new.group && cso in r.old.group &&
    overlap(csn.location,cso.location)
}
//run SHIFT for 5

fun ADD (r:Relationship) {
  (similarhigh(r.new.group.text, r.old.group.text) ||
   similar(r.new.group.text,r.old.group.text))
   some csn:Code | all cso:Code |
   cso in r.old.group => csn in r.new.group &&
   notoverlap(csn.location,cso.location)
}
//run ADD for 5

fun SUBTRACT(r:Relationship) {
  (similarhigh(r.new.group.text, r.old.group.text) ||
   similar(r.new.group.text,r.old.group.text))
   some cso:Code | all csn:Code |
   csn in r.new.group => cso in r.old.group &&
   notoverlap(csn.location,cso.location)
}
//run SUBTRACT for 5

fun CONSISTENT(r:Relationship) {
  similar(r.new.group.text,r.old.group.text)
  all cso:Code | some csn:Code |
  cso in r.old.group => csn in r.new.group && (
    overlap(csn.location,cso.location) ||
    overlaphigh(csn.location,cso.location))
}
//run CONSISTENT for 5

fun INCONSISTENT(r:Relationship) {
  similar(r.new.group.text,r.old.group.text)
  some cso:Code | all csn:Code |
  csn in r.new.group => cso in r.old.group &&
    notoverlap(csn.location,cso.location)
}
//run INCONSISTENT for 5

assert ALL_EXHAUSTIVE {
  all r:Relationship |
    !notsimilar(r.new.group.text,r.old.group.text) =>
       ADD(r)|| SHIFT(r) || SAME(r) ||
       SUBTRACT(r) || CONSISTENT(r) || INCONSISTENT(r)
}
//check ALL_EXHAUSTIVE for 5
//proved true

assert SAME_IN_SHIFT {
  all r:Relationship | SAME(r) => SHIFT(r)
}
//check SAME_IN_SHIFT for 5

assert SHIFT_IN_SAME {
  all r:Relationship | SHIFT(r) => SAME(r)
}
//check SHIFT_IN_SAME for 5

fun SHIFT_AND_SAME (r:Relationship) {
  SHIFT(r) && SAME(r)
}
//run SHIFT_AND_SAME for 5

assert INCONSISTENT_IN_SUBTRACT {
  all r:Relationship | INCONSISTENT(r)
  => SUBTRACT(r)
}
//check INCONSISTENT_IN_SUBTRACT for 5
```