

=====

ITEM #141 — Missing-Strands Reconstruction Algorithm

Conversation Title: ACLM 横向变异算法讨论

Date: 20251205

Authors: Sizhe Tan & GPT-Obot

=====

ME:

请生成 ITEM #141 — Missing-Strands Reconstruction Algorithm

ChatGPT said:

下面直接给出 ITEM #141 — Missing-Strands Reconstruction Algorithm 的正式稿。

我按你一贯风格写成：算法级技术条目，中英文各一版，并与 #140 自然衔接。

---

## ■ ITEM #141 — 缺失语义股重建算法 (Missing-Strands Reconstruction Algorithm)

### 一、定位：承接 ITEM #140 的“核心子引擎”

在 ITEM #140 — ACLM Horizontal Mutation: Operation State Completion Engine 中，我们将：

- 上下游 Operation 之间的对接差异

- 抽象为“多股语义绳（multi-strand semantics rope）”中的 **缺失语义股（missing strands）** 问题。

ITEM #141 就是为此定义一个 **可操作、可进化、可实现** 的算法核心：

**如何从 Calling Graph 与 CCC 知识结构中，自动重建这些缺失语义股？**

换言之：

#141 = “**缺股重建核（Missing-Strands Reconstruction Core）**”

是 #140 Operation State Completion Engine 的中央心脏。

---

## 二、问题形式化：什么是“缺失语义股”？

设：

- 上游 Operation :  $Op\_up$
- 下游 Operation :  $Op\_down$
- Strands ( $Op$ ) 表示 Operation 所具备的语义股集合。每条 Strand 是一个结构化元素：

Strand=(Type,StateSignature,CCCNODE,TransformChain)\text{Strand} = (\text{Type}, \text{StateSignature}, \text{CCCNODE}, \text{TransformChain})

则：

- 下游所需的语义股集合：

$R=RequiredStrands(Op\_down)$   
 $R = \text{RequiredStrands}(Op\_down)$

- 上游实际提供的语义股集合：

$P = \text{ProvidedStrands(Op\_up)}$   
 $P = \setminus \text{ProvidedStrands}(Op\_{up})$

定义：

$M = R \setminus PM = R \setminus \setminus PM = R \setminus P$

其中：

- $MM = \text{Missing-Strands Set}$
  - 每个  $m \in M$  就是一条需要被自动重建的 缺失语义股。
- 

### 三、算法目标：从“语义目标”到“可执行路径”

对每条缺失语义股  $mm$ ，算法目标是：

在 ACLM 的 Calling Graph 与 CCC Functional Graph 中，  
找到一条或多条 生成  $mm$  所需状态/语义 的路径，  
并在必要时通过纵向变异修复或合成，最终形成一个可连接到  $Op\_down$  的 Composite Strand。

这等价于：

- 在结构图中做一次 目标驱动的反向搜索 (goal-directed backward search) ；
  - 并在搜索中动态使用 四大 Vertical Mutations 修补缺口。
- 

### 四、核心数据结构

#### 1. Calling Graph (调用图)

$CG = (V, E)$   
 $CG = (V, E)$

- VVV : Operations / Functions / Methods
- EEE : 调用边 ( $A \rightarrow B$  表示 A 调用 B)

## 2. CCC Functional Graph (功能知识图)

$FG = (C, L) FG = (C, L) FG = (C, L)$

- CCC : CCC 节点 (概念/功能/语义单元)
- LLL : CCC 之间的功能关系 (依赖、推导、组合)

## 3. Strand Descriptor (语义股描述)

- 类型 (data / state / logic / semantic)
- 目标 CCC 节点
- 所需 State Signature (类型 + 约束条件)

## 4. Path Candidate (候选路径)

- ops\_chain: Operation 序列
- ccc\_chain: 概念/语义节点序列
- cost: 复杂度 / 风险 / 长度估计
- repair\_ops: 在路径中需要使用 Vertical Mutations 的点

---

# 五、缺失语义股重建的算法步骤

下面是针对 单条缺失语义股  $m$  的 Reconstruction Algorithm。

多条缺失股的情况见后续“多股处理”章节。

---

## Step 1 : 构造缺失语义股的“目标签名” (Target Signature)

从  $mmm$  中抽取：

1. 目标数据/状态类型 :  $TargetStateType(m)$
2. 对应 CCC 语义节点 :  $CCC(m)$

### 3. 必要上下文约束（如：必须在某种模式、某种安全条件下）

得到一个 目标签名结构：

$\text{Target}(m) = (\text{StateType}, \text{CCCNode}, \text{ContextConstraints})$   
 $\text{Target}(m) = (\text{\textbackslash text\{StateType\}},$   
 $\text{\textbackslash text\{CCCNode\}},$   
 $\text{\textbackslash text\{ContextConstraints\}})$   
 $\text{Target}(m) = (\text{StateType}, \text{CCCNode}, \text{ContextConstraints})$

这是整个搜索的“灯塔”。

---

## Step 2：在 CCC Functional Graph 中定位候选“语义来源”

在 FG 上执行：

“有哪些 CCC 节点可以 直接或间接产生  $\text{Target}(m)$ ？”

这一步的输出是一个 候选 CCC 节点集合：

$\text{CandidCCC}(m) = \{c \in C | c \Rightarrow \text{CCC}(m)\}$   
 $\text{CandidCCC}(m) = \{c \in C | c \mid\!\! \rightarrow \text{CCC}(m)\}$   
 $\text{CandidCCC}(m) = \{c \in C | c \Rightarrow \text{CCC}(m)\}$

其中  $c \Rightarrow \text{CCC}(m)$  表示：

- $c$  能直接产出目标语义；或
  - $c$  能通过一小段 CCC 组合推导生成目标语义。
- 

## Step 3：从候选 CCC 节点映射到具体 Operation 节点

对每个  $c \in \text{CandidCCC}(m)$  :

- 查找在 Calling Graph 中实现该 CCC 功能的 Operations :

$\text{Ops}(c) = \{op \in V | op \text{ implements or approximates } c\}$   
 $\text{Ops}(c) = \{op \in V | op \mid\!\! \rightarrow \text{implements or approximates } c\}$   
 $\text{Ops}(c) = \{op \in V | op \text{ implements or approximates } c\}$

合并得到：

$$\begin{aligned} \text{CandidOps}(m) &= \bigcup_{c \in \text{CandidCCC}(m)} \text{Ops}(c) \\ \text{Ops}(c) \text{CandidOps}(m) &= c \in \text{CandidCCC}(m) \cup \text{Ops}(c) \end{aligned}$$

这些是 潜在能生成目标语义股 m 的 Operation 节点。

---

#### Step 4：在 Calling Graph 中做“反向路径搜索”（Backward Path Search）

目标：

在 Calling Graph 中，

从  $\text{CandidOps}(m)$  反向/正向，

找到一条能 连接到当前上下文（Op\_up / Op\_down 之间）的路径，

并能按顺序生成所需的 State/CCC 语义。

可以采用：

- 反向 BFS / Dijkstra / A\* （代价函数可加入：路径长度、变异次数、复杂度）
- 限制搜索层数以避免爆炸
- 加入 “context-compatible” 过滤条件：
  - 类型相容
  - 状态不会破坏已有 invariants
  - 无明显 side-effect 冲突

输出若干条 路径候选：

$$\begin{aligned} \text{PathCandidates}(m) &= \{\text{path\_1}, \text{path\_2}, \dots, \text{path\_k}\} \\ \text{PathCandidates}(m) &= \{\text{path\_1}, \text{path\_2}, \dots, \text{path\_k}\} \end{aligned}$$

---

#### Step 5：路径中的断点与局部修复（调用 Vertical Mutations）

每条路径 `path_i` 可能并非完全可用：

它可能存在：

- 类型不完全匹配
- 语义链有缺口
- 中间 Operation 不存在合适参数
- 某个中间功能只有类似而非完全一致

此时，ITEM #140 中的机制启动：

- 使用 **Edit Mutation** 修补参数/签名校验差异
- 使用 **Calling Graph Metaprogramming** 增加/重排中间 Operation
- 使用 **Puzzle-Bridging** 弥合中间语义断点
- 使用 **Functional CCC Metaprogramming** 生成新的函数节点

我们可以为每个修复点增加一个 **修复代价 cost\_repair**，用于后续路径排序。

最终，每条 `path_i` 被转化为一个带修复信息的结构：

`RepairedPathi=(ops_chain,ccc_chain,total_cost,repair_ops)`  
`RepairedPath_i = (ops\_chain,`  
`ccc\_chain, total\_cost, repair\_ops)`  
`RepairedPathi=(ops_chain,ccc_chain,total_cost,repair_ops)`

---

## Step 6：路径选择（Path Selection）

从所有 `RepairedPath_i` 中选取 代价最优的一个或若干个：

- 优先级可按以下排序：
  1. 修复次数少 (Vertical Mutation 调用少)
  2. 路径长度短 (Ops 链条短)
  3. 与现有代码相似度高 (便于复用与解释)
  4. Side-effect 风险低 (不会破坏已有行为)

记最优路径为：

$\text{BestPath}(m) = \text{RepairedPath} \setminus * \text{BestPath}(m) = \text{RepairedPath} \wedge * \text{BestPath}(m) = \text{RepairedPath} \setminus *$

---

## Step 7：合成缺失语义股的 Composite Strand

沿  $\text{BestPath}(m)$  :

1. 抽取数据流与状态流
2. 计算传递后的 State Signature
3. 将多步变换与修复合并为一条 复合语义股 :

$\text{CompStrand}(m) = (\text{StateType}^*, \text{CCCNode}^*, \text{TransformChain}^*)$   
 $\text{CompStrand}(m) = (\text{\textbackslash text}\{\text{StateType}\}^*, \text{\textbackslash text}\{\text{CCCNode}\}^*,$   
 $\text{\textbackslash text}\{\text{TransformChain}\}^*)$   
 $\text{CompStrand}(m) = (\text{StateType}^*, \text{CCCNode}^*, \text{TransformChain}^*)$

这条 CompStrand 就是 :

对缺失股 m 的 结构化重建结果 , 可以被挂接回  $\text{op\_down}$  的输入侧。

---

## 六、多股缺失的协同重建 (Multi-Strand Reconstruction)

当 MMM 中存在多条缺失语义股时 :

- 不能完全独立处理 , 否则可能出现 :
  - 重复计算
  - 相互冲突的路径
  - 状态覆盖问题

改用 :

1. 对所有  $m \in M$  先完成 Step 1–2 : 目标签名定位与 CCC 候选
2. 在 CCC 图中做一次 多目标联合搜索 , 避免重复路径
3. 在 Calling Graph 中优先寻找可以 同时服务多条缺失股 的公共路径

4. 对多条 CompStrand(m) 做一次 冲突检测与合成：
  - 若两个 strand 在中间状态冲突，需加入额外修复 operation
  - 若两个 strand 可共用部分路径，可抽取为共享子路径（共用 helper 函数）

这时候，ITEM #141 的算法升级为一类 多目标结构规划（multi-goal structural planning）问题，与 DBM 的“多目标 What-If 引擎”是同一类结构。

---

## 七、失败模式与退火策略

如果 Missing-Strands Reconstruction 算法失败或代价过高，可以：

1. 降级策略（Degradation）
  - 标记为“部分补全”
  - 用 simpler heuristic 或 stub 实现（比如返回默认值、占位对象）
2. 延迟策略（Deferral）
  - 把当前缺失股列为未来 Mutation 的优先目标
  - 收集更多上下文信息后再进行重建
3. 交互策略（Interactive ACLM）
  - 向人类程序员发出“提示问题卡”，说明缺失股的结构与可能路径
  - 人类选择或修改路径后，ACLM 记入 Mutation 经验库

---

## 八、总结：ITEM #141 在 ACLM/DBM 演化中的位置

可以用一句话总结：

**ITEM #141 — Missing-Strands Reconstruction Algorithm**  
是在 ACLM 自动编程中，  
从“知道自己缺什么”（缺失股检测），  
到“从知识图与调用图中构造出缺的那一股”（结构重建），

再到“将重建结构压缩成一条可挂接的复合语义股”  
的完整算法闭环。

它让：

- ITEM #140 中的 Horizontal Mutation
  - 不仅是“概念上的状态补全”，
  - 而成为一个有 明确步骤、可编码实现、可演化优化 的 算法级引擎。
- 

## ITEM #141 — Missing-Strands Reconstruction Algorithm

(English Edition)

---

### 1. Positioning: The Core Sub-Engine of ITEM #140

In ITEM #140, we introduced the **ACLM Horizontal Mutation: Operation State Completion Engine**, which views:

- Mismatch between upstream and downstream Operations
- As **missing semantic strands** in a multi-strand semantic rope.

ITEM #141 defines the **algorithmic core** that answers:

**How can ACLM reconstruct those missing strands  
using the Calling Graph and the CCC Functional Graph?**

---

### 2. Formalization: What is a “Missing Strand”?

Let:

- Upstream operation:  $\text{Op\_up}$
- Downstream operation:  $\text{Op\_down}$

Define the semantic strands of an Operation:

Strand=(Type,StateSignature,CCCNODE,TransformChain)  
Strand = (Type, StateSignature, CCCNODE, TransformChain)  
Strand=(Type,StateSignature,CCCNODE,TransformChain)

Then:

- Required strands by  $Op\_down$ :

$R=RequiredStrands(Op\_down)R =$   
 $RequiredStrands(Op\_{down})R=RequiredStrands(Op\_down)$

- Provided strands by  $Op\_up$ :

$P=ProvidedStrands(Op\_up)P = ProvidedStrands(Op\_{up})P=ProvidedStrands(Op\_up)$

Define the missing strands set:

$M=R\setminus P M = R \setminus P M=R\setminus P$

Each  $m \in M$  is a **Missing Semantic Strand** to be reconstructed.

---

### 3. Algorithm Goal

For each missing strand  $m \in M$ , the goal is:

To find one or more operation paths in the Calling Graph and CCC Functional Graph that can produce the required state and semantics,  
repair gaps using Vertical Mutations if needed,  
and compress the result into a **Composite Strand** that can be attached to  $Op\_down$ .

This is essentially a:

- **Goal-directed backward search** in the structural graphs
  - With dynamic use of **Vertical Mutations** for local repair.
- 

### 4. Core Data Structures

#### 1. Calling Graph

$CG=(V,E)CG = (V, E)CG=(V,E)$

- VVV: operations / functions
  - EEE: call edges
- 2. CCC Functional Graph**

$FG=(C,L)FG = (C, L)FG=(C,L)$

- CCC: CCC nodes (concepts, functions, semantic units)
  - LLL: functional relations among CCC nodes
- 3. Strand Descriptor**
- Type (data / state / logic / semantic)
  - Target CCC node
  - State signature constraints
- 4. PathCandidate**
- ops\_chain
  - ccc\_chain
  - cost
  - repair\_ops
- 

## 5. Reconstruction Steps for a Single Missing Strand

### Step 1: Build the Target Signature

From mmm, extract:

- StateType = TargetStateType(m)
- CCCNode = CCC(m)
- ContextConstraints

Form a target descriptor:

$Target(m)=(StateType, CCCNode, ContextConstraints)$   
 $Target(m) = (\text{\text{StateType}}, \text{\text{CCCNode}}, \text{\text{ContextConstraints}})$   
 $Target(m)=(StateType, CCCNode, ContextConstraints)$

---

### Step 2: Locate Candidate Semantic Sources in CCC Graph

In the CCC Functional Graph FG, search for nodes that can directly or indirectly produce  $CCC(m)$ :

$CandidCCC(m)=\{c \in C | c \Rightarrow CCC(m)\}$   
 $CandidCCC(m) = \{ c \mid c \in C \wedge c \Rightarrow CCC(m) \}$   
 $\} CCC(m)=\{c \in C | c \Rightarrow CCC(m)\}$

Here  $c \Rightarrow CCC(m)$  means:

- $c$  can directly yield  $\text{CCC}(m)$ , or
  - $c$  can derive  $\text{CCC}(m)$  via a small CCC composition.
- 

### Step 3: Map CCC Candidates to Operations

For each  $c \in \text{CandidCCC}(m) \setminus \text{CandidCC}(m) \cup \text{CandidCC}(m)$ :

$\text{Ops}(c) = \{op \in V \mid op \text{ implements or approximates } c\}$

$$\text{Ops}(c) = \{op \in V \mid \exists o \in \text{CandidCC}(m) \text{ such that } op \text{ implements or approximates } o \text{ and } o \text{ implements or approximates } c\}$$

Combine:

$\text{CandidOps}(m) = \bigcup_{c \in \text{CandidCCC}(m)} \text{Ops}(c)$

$$\text{CandidOps}(m) = \bigcup_{c \in \text{CandidCCC}(m)} \{op \in V \mid \exists o \in \text{CandidCC}(m) \text{ such that } op \text{ implements or approximates } o \text{ and } o \text{ implements or approximates } c\}$$

These are potential producers of the missing strand.

---

### Step 4: Backward Path Search in Calling Graph

Goal:

Find paths in the Calling Graph  
that connect  $\text{CandidOps}(m)$  to the current context (around  $\text{Op\_up}/\text{Op\_down}$ )  
while producing the required state and semantics.

Use:

- Backward BFS / Dijkstra / A\*
- Context compatibility constraints
- Depth limits

Output:

$\text{PathCandidates}(m) = \{\text{path}_1, \text{path}_2, \dots, \text{path}_k\}$

$$\text{PathCandidates}(m) = \{\text{path}_1, \text{path}_2, \dots, \text{path}_k\}$$


---

### Step 5: Repair Gaps with Vertical Mutations

Each path  $\text{path}_i$  may be incomplete or inconsistent:

- Type mismatches

- Missing intermediate semantics
- Incomplete parameters
- Only approximate functionality available

Invoke Vertical Mutations:

- **Edit Mutations** for signature-level repair
- **Calling-Graph Metaprogramming** for inserting/reordering ops
- **Puzzle-Bridging** for semantic gaps
- **Functional CCC Metaprogramming** for creating new functional nodes

Each repair adds a `cost_repair`. The repaired path becomes:

`RepairedPathi=(ops_chain,ccc_chain,total_cost,repair_ops)`  
`RepairedPath_i = (ops\_chain,`  
`ccc\_chain, total\_cost, repair\_ops)`  
`RepairedPathi=(ops_chain,ccc_chain,total_cost,repair_ops)`

---

## Step 6: Path Selection

Select best path according to:

1. Fewer repairs
2. Shorter operation chain
3. Higher similarity to existing code
4. Lower side-effect risk

Let:

`BestPath(m)=RepairedPath\*BestPath(m) = RepairedPath^\*BestPath(m)=RepairedPath\*`

---

## Step 7: Composite Strand Synthesis

Along `BestPath(m)`:

- Merge data/state transforms
- Compute resulting StateSignature
- Compose into a **Composite Strand**:

`CompStrand(m)=(StateType\*,CCCNode\*,TransformChain\*)`  
`CompStrand(m) = (\text{StateType}^\*, \text{CCCNode}^\*,`  
`\text{TransformChain}^\*)`  
`CompStrand(m)=(StateType\*,CCCNode\*,TransformChain\*)`

`CompStrand(m)` is the reconstructed missing strand, ready to be attached to `Op_down`.

---

## 6. Multi-Strand Reconstruction

When multiple missing strands exist:

- Independent reconstruction may cause redundancy or conflicts.
- Instead, perform:
  1. Joint target analysis in CCC space
  2. Multi-goal search for shared CCC and operation paths
  3. Conflict detection among strands (state overwrites, incompatible invariants)
  4. Shared sub-path extraction for efficiency and coherence

Thus, ITEM #141 naturally upgrades to a **multi-goal structural planning** engine, closely aligned with DBM's multi-goal What-If engines.

---

## 7. Failure Modes and Fallback Strategies

If reconstruction fails or is too costly:

1. **Degradation**
  - Provide a partial implementation or stub
  - Mark the strand for future refinement
2. **Deferral**
  - Postpone reconstruction until more context is available
3. **Interactive Mode**
  - Present candidate paths to a human engineer
  - Learn from human choices into the mutation knowledge base

---

## 8. Summary

### ITEM #141 — Missing-Strands Reconstruction Algorithm

is the algorithmic core that turns

“we know which semantic strands are missing”

into

“we can reconstruct them from our structural and knowledge graphs and compress them into attachable composite strands.”

Together with ITEM #140, it upgrades ACLM from:

- Merely **detecting mismatches**
- To **repairing and completing program structures autonomously**.

---

兄弟，这就是 **ITEM #141** 的完整算法级定义稿，中英文已成型，可以直接进 DBM/ACLM 总清单。

如果你愿意，下一步我们可以：

- 写一个 **彩色结构图**：  
“Missing-Strands Reconstruction Engine Diagram”
- 再来一个 **Java 8 骨架接口**，例如：  
IMissingStrandReconstructor + 若干策略实现类，用小玩具例子跑通一条“缺股  
→ 搜索 → 补全 → 接入”的流程。