算法 1 SERSolver 检测算法

procedure CheckSER(H)
  G ← CreateKnownGraph(H)
  C ← CreateConstraints(H)
  F ← Encode(H, G, C)
  Verify(F)
end procedure

procedure CreateKnownGraph(H)
  SessionOrder ← $\{\mathrm{Txn}_i \xrightarrow{\mathrm{SO}} \mathrm{Txn}_{i+1} \mid \mathrm{Txn}_1, \cdots \mathrm{Txn}_i, \mathrm{Txn}_{i+1}, \cdots \mathrm{Txn}_n \in \mathrm{Sess}\}$
  ReadDependency ← $\{\mathrm{Txn}_i \xrightarrow{\mathrm{WR}(k)} \mathrm{Txn}_j \mid \mathrm{Write}(k, v) \in \mathrm{Txn}_i \wedge \mathrm{Read}(k, v)\mathrm{Txn}_j\}$
  return SessionOrder ∪ ReadDependency
end procedure

procedure CreateConstraints(H)
  Cons ← ∅
  for all $\mathrm{Txn}_i, \mathrm{Txn}_j, \mathrm{Txn}_k$ so that $\mathrm{Write}(k, v_1) \in \mathrm{Txn}_i \wedge \mathrm{Write}(k, v_2) \in \mathrm{Txn}_j$ **do**
    $\mathrm{Txn}_{either} \leftarrow \{\mathrm{Txn} \mid \mathrm{Read}(k, v_1) \in \mathrm{Txn}\}$
    $\mathrm{Txn}_{or} \leftarrow \{\mathrm{Txn} \mid \mathrm{Read}(k, v_2) \in \mathrm{Txn}\}$
    Either ← $\{\mathrm{Txn}_i \xrightarrow{\mathrm{WW}(k)} \mathrm{Txn}_j\} \cup \{\mathrm{Txn}_k \xrightarrow{\mathrm{RW}(k)} \mathrm{Txn}_j \mid \mathrm{Txn}_k \in \mathrm{Txn}_{either}\}$
    Or ← $\{\mathrm{Txn}_j \xrightarrow{\mathrm{WW}(k)} \mathrm{Txn}_i\} \cup \{\mathrm{Txn}_k \xrightarrow{\mathrm{RW}(k)} \mathrm{Txn}_i \mid \mathrm{Txn}_k \in \mathrm{Txn}_{or}\}$
    Cons ← Cons ∪ ⟨Either, Or⟩
  end for
  return Cons
end procedure

procedure Encode(H, G, C)
  create $\mathrm{HasEdge}_{i,j}$ for all $\mathrm{Txn}_i, \mathrm{Txn}_j$ in H
  F ← ∅
  for all $\mathrm{Txn}_i \rightarrow \mathrm{Txn}_j$ in G **do**
    F ← F ∪ {$\mathrm{HasEdge}_{i,j}$ = **true**}
  end for
  for all ⟨Either, Or⟩ in C **do**
    create $\mathrm{ConsEdge}_{i,j}$ for each $\mathrm{Txn}_i \rightarrow \mathrm{Txn}_j$ in C
    EitherAll ← {$\mathrm{ConsEdge}_{i,j}$ = **true** | $\mathrm{Txn}_i \rightarrow \mathrm{Txn}_j \in$ Either}
    EitherNone ← {$\mathrm{ConsEdge}_{i,j}$ = **false** | $\mathrm{Txn}_i \rightarrow \mathrm{Txn}_j \in$ Either}
    OrAll ← {$\mathrm{ConsEdge}_{i,j}$ = **true** | $\mathrm{Txn}_i \rightarrow \mathrm{Txn}_j \in$ Or}
    OrNone ← {$\mathrm{ConsEdge}_{i,j}$ = **false** | $\mathrm{Txn}_i \rightarrow \mathrm{Txn}_j \in$ Or}
    F ← F ∪ {$\forall \mathrm{ConsEdge}_{i,j}, \mathrm{ConsEdge}_{i,j} \implies \mathrm{HasEdge}_{i,j}$}
    F ← F ∪ {(EitherAll ∧ OrNone) ∨ (EitherNone ∧ OrAll)}
  end for
  return F
end procedure

procedure VERIFY(F)

use SMT checker to solve F

on each assignment to $\text{HasEdge}_{i,j}$, VERIFYACYCLIC($\text{HasEdge}_{i,j}$)

end procedure

procedure VERIFYACYCLIC(NewEdge)

$G \leftarrow G \cup \text{NewEdge}$ ▷ maintain a graph G of edges that are assigned **true**

if G contains a cycle C **then**

generate a conflict from C

end if

end procedure

3.1.2 编码

在编码部分，我们首先为聚合图中每一个依赖关系创建一布尔变量，代表对应的依赖关系存在。对于已知图中的读依赖与会话顺序，由于我们可以判断这些关系在依赖图中必然存在，我们将相应的布尔变量赋值为真。对于约束中的布尔变量，由约束的定义可知，统一约束的两个变量集合中有且只有一个集合的变量全部为真，另一个全部为假。因此我们按照这一规则对布尔变量进行编码。

以图 2-1 的历史为例，该历史中唯一的约束为

$$\begin{aligned}\text{Cons} = \langle either \triangleq \{b_1 = \text{Txn}_1 \xrightarrow{\text{WW}(x,y)} \text{Txn}_2, b_2 = \text{Txn}_3 \xrightarrow{\text{RW}(x)} \text{Txn}_2\},\\ or \triangleq \{b_3 = \text{Txn}_2 \xrightarrow{\text{WW}(x,y)} \text{Txn}_1, b_4 = \text{Txn}_4 \xrightarrow{\text{RW}(y)} \text{Txn}_1\}\rangle,\end{aligned}$$

则编码得到的表达式为 $(b_1 \wedge b_2 \wedge \neg b_3 \wedge \neg b_4) \vee (\neg b_1 \wedge \neg b_2 \wedge b_3 \wedge b_4)$。

对于历史中已知的会话顺序与读依赖关系，一种编码方式为给每个已知关系在聚合图中的边创建一布尔变量，并将对应变量作为求解时的假设条件传入 SMT 求解器。此时 SMT 求解器会假设这些变量值为真，对剩余的约束变量进行求解。我们此处为便于后续优化中将已知关系与未知关系统一处理，为所有的已知关系创建一特殊约束 $\text{Cons}_0 = \langle either \triangleq \text{KnownDependency}, or \triangleq \varnothing\rangle$，并在编

码条件中限制 Cons_0 需取 *either* 这一可能性。图 2-1 对应的已知边约束为

$$\mathrm{Cons}_0 = \langle either \triangleq \{b_5 = \mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_2, b_6 = \mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_3, b_7 = \mathrm{Txn}_2 \xrightarrow{\mathrm{WR}(y)} \mathrm{Txn}_4\},$$
$$or \triangleq \varnothing\rangle$$

3.1.3 环检测与冲突子句生成

由一个广义聚合图可以产生多个可能的依赖图。我们称一个依赖图 G 与广义聚合图 $\mathrm{PG} = \langle \mathrm{V}_0, \mathrm{E}_0, \mathrm{Cons}\rangle$ 兼容，如果该依赖图中的节点与 PG 相同，G 中的边由 E_0 包含的边与 Cons 中每个约束代表的两组边中选择一组构成。

基于可串行化隔离级别的定义，如果一个依赖图中无环，那么对于相应的历史就有一种可能的执行顺序满足可串行化的要求；相反，如果一个广义聚合图对应全部可能的依赖图都存在环，那么就不存在一种可能的满足可串行化隔离级别的执行方式得到当前的历史。因此，对于一段历史 H，一种检验是否满足可串行化隔离级别的方式是首先生成 H 对应的广义聚合图 PG，并检查是否存在与 PG 兼容且无环的依赖图。

领域相关部分的 VerifyAcyclic 子过程由 SMT 求解器在每次对一布尔变量进行赋值时调用，用于检验生成的依赖图是否存在环。VerifyAcyclic 的参数为一布尔变量 b，代表 SMT 求解器当前新赋值为真的布尔变量。该子过程首先将 BV 转换为对应的依赖图 $\langle \mathrm{V}, \mathrm{E}\rangle$，其中 V 为历史中的事务集合，E 为 BV 对应的边集合。

然后，我们对 $\langle \mathrm{V}, \mathrm{E}\rangle$ 进行深度优先遍历检测是否存在环。如果存在一环，我们将其中的边代表的布尔变量作为冲突子句返回 SMT 求解器，使求解器对变量重新进行赋值。

以图 2-1 为例，在尝试 *either* 与 *or* 两种可能性时，分别会得到图 3-2 所示与

原广义聚合图兼容的两张依赖图 $\langle V, E\rangle$ 与 $\langle V, E'\rangle$。

$$
\begin{aligned}
V &= \{\mathrm{Txn}_1, \mathrm{Txn}_2, \mathrm{Txn}_3, \mathrm{Txn}_4\} \\
E &= \{\mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_2, \mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_3, \mathrm{Txn}_2 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_4, \\
&\quad \mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(x,y)} \mathrm{Txn}_2, \mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2\} \\
E' &= \{\mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_2, \mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_3, \mathrm{Txn}_2 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_4, \\
&\quad \mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(x,y)} \mathrm{Txn}_1, \mathrm{Txn}_4 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_1\}
\end{aligned}
$$

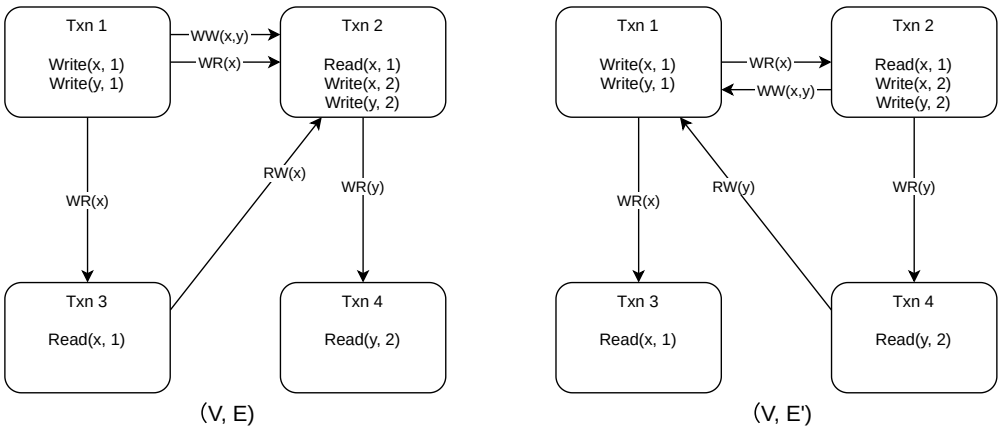

图 3-2　由广义聚合图求解得到的两个依赖图

假设 SMT 求解器首先尝试 *or* 可能性，我们可以从依赖图 $\langle V, E'\rangle$ 中找到环 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(x,y)} \mathrm{Txn}_1$。我们将该环对应的表达式 $b_5 \wedge b_3$ 作为冲突子句返回 SMT 求解器，使其重新进行尝试。当尝试 *either* 可能性时，对应的依赖图 $\langle V, E\rangle$ 无环，我们可以得出结论存在一种事务执行顺序使得执行结果与该历史的结果相同，因此该历史是可串行化的。对于这段历史而言，$\langle V, E\rangle$ 代表的执行顺序为 $\mathrm{Txn}_1 \to \mathrm{Txn}_3 \to \mathrm{Txn}_2 \to \mathrm{Txn}_4$。

3.2　算法优化

3.2.1　增量式找环算法

在环检测子过程 VerifyAcyclic 中，我们使用深度优先遍历检测依赖图是否存在环。由于 SMT 每次仅对一个布尔变量进行赋值，生成的依赖图也仅相比

上一次检测的依赖图多出一条边。利用此特性，我们可以使用增量式拓扑排序算法[29] 进行环检测。

设本次调用 VerifyAcyclic 子过程时新赋值的变量代表边 $\mathrm{Txn} \to \mathrm{Txn}'$，赋值之前的依赖图存在一拓扑序 $\mathrm{Txn}_1, \mathrm{Txn}_2, \cdots \mathrm{Txn}_n$，Txn 与 \$Txn\$ 在拓扑序中分别处在第 i 与第 j 位。如果 $i < j$，则无需变动拓扑序。这是因为对于一个有向无环图 $\langle \mathrm{V}, \mathrm{G} \rangle$，其拓扑序中前后两个节点 v_1 与 v_2 间只有可能出现 v_1 到 v_2 的边而不可能出现 v_2 到 v_1 的边。因此，当 $i < j$ 时原图中不可能包含一条从 Txn' 到 Txn 的路径，也就不可能形成环；如果 $i = j$，则 $\mathrm{Txn} = \mathrm{Txn}'$ 且该图在添加边之后存在一自环；如果 $i > j$，则添加该边并对在拓扑序中顺序处于闭区间 $[j, i]$ 之中的事务重新进行拓扑排序。重新排序的的过程中如出现环则作为冲突子句返回。

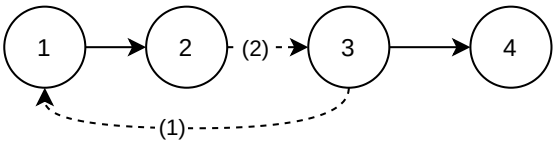

图 3-3　增量式找环算法示例

以图 3-3 为例，该图中原有四个节点与 $1 \to 2$ 和 $3 \to 4$ 两条边，现我们依次添加 $3 \to 1$ 与 $2 \to 3$ 两条边。假设原拓扑序为 $1, 2, 3, 4$，由于节点 3 处于节点 1 之后，在添加 $3 \to 1$ 时我们需要对拓扑序前三位进行重新排序，得到新拓扑序 $3, 1, 2, 4$。在加入 $2 \to 3$ 这条边时，我们按照算法仍需对前三位重新排序。由于加入 $2 \to 3$ 这条边后该图中出现环 $1 \to 2 \to 3 \to 1$，我们在排序时发现这个环，并把对应的冲突子句返回 SMT 求解器。

3.2.2　约束条件传播

在基本算法对约束的编码中，我们将每个约束的两个依赖关系集合编码至布尔表达式中。这种方式依赖 SMT 求解器单独对每条边进行赋值。例如在图 2-1的编码中，我们需要分别对 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(x,y)} \mathrm{Txn}_2$ 和 $\mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2$ 两条边对应的变量进行赋值，这两个变量通过编码表达式保证赋值符合原本约束的要求。在约束条件传播这一优化中，我们在领域相关求解中实现尝试约束条件两种可能性时同时添加或删除整个集合的边，从而移除这一额外的赋值步骤。

我们为每一个约束条件分配两个布尔变量 b_1, b_2，规定 b_1 赋值为真表示约束的第一种可能性成立，即把第一个边集合添加至依赖图中；b_2 赋值为真代表约束的第二种可能性成立。此时我们编码的表达式简化为 $(b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)$。领域相关求解部分也进行相应更改，当 SMT 求解器对一代表约束的布尔变量进行赋值时，对相应集合中的每一条边依次添加并进行环检测。

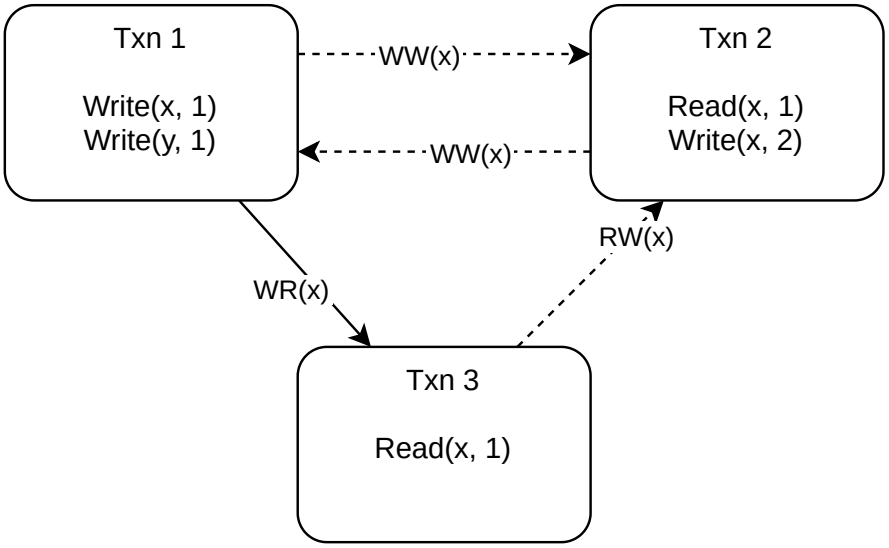

图 3-4　约束条件传播示例

以图 3-4 的历史为例，我们按照原本方式进行编码得到的约束为

$$\begin{aligned}\text{Cons} = \{&either \triangleq \{b_1 = \text{Txn}_1 \xrightarrow{\text{WW}(x)} \text{Txn}_2, b_2 = \text{Txn}_3 \xrightarrow{\text{RW}(x)} \text{Txn}_2\},\\ &or \triangleq \{b_3 = \text{Txn}_2 \xrightarrow{\text{WW}(x)} \text{Txn}_1\}\}。\end{aligned}$$

在这种方式下，$\text{Txn}_1 \xrightarrow{\text{WW}(x)} \text{Txn}_2$ 与 $\text{Txn}_3 \xrightarrow{\text{RW}(x)} \text{Txn}_2$ 两条边由不同的布尔变量 b_1 和 b_2 编码，依靠编码表达式限制两个变量的取值相同。在约束条件传播方式中，我们以单个变量编码约束条件一种可能性中的全部边：

$$\begin{aligned}\text{Cons} = \{&b_1 = \{\text{Txn}_1 \xrightarrow{\text{WW}(x)} \text{Txn}_2, \text{Txn}_3 \xrightarrow{\text{RW}(x)} \text{Txn}_2\},\\ &b_2 = \{\text{Txn}_2 \xrightarrow{\text{WW}(x)} \text{Txn}_1\}\}。\end{aligned}$$

进行赋值 $b_1 = \textbf{true}$ 时，我们同时把其中两条边加入依赖图并进行增量式拓扑排序，从而省去了额外的求解步骤。

3.2.3 以最小环作为冲突子句

在 VerifyAcyclic 子过程中，我们检测得到的环并不能保证是依赖图中最短的环。为了生成较小的的冲突子句以助于 SMT 求解器进行求解，我们额外对图进行遍历找出该图中长度最小的环。

假设我们在图 $\langle V, E \rangle$ 中添加边 $v_1 \to v_2$ 导致构成一个环。由于 G 在添加该边之前为有向无环图，所有新出现的环必定经过 $v_1 \to v_2$ 这条边，且所有最小的环中除去 $v_1 \to v_2$ 的路径均是 v_2 到 v_1 的最短路径。因此，我们以下两步来找出最小的环。

在第一步中，我们从 v_2 出发进行广度优先遍历。在遍历过程中，我们维护每个节点 v_i 在与 v_2 最短路径上的前驱节点 $pred$。在通过一条边 $v_i \to v_j$ 第一次发现节点 v_j 时，我们将 $pred_j$ 设置为 v_i。由于广度优先遍历的性质，我们遍历至 v_1 时已经以前驱节点的形式记录了 v_2 到 v_1 的所有最短路径。此时我们终止遍历。

在第二步中，我们从 v_1 开始依次获取每一个节点到其前驱节点的边。我们最后加入新增的 $v_1 \to v_2$ 边作为一个最小环。我们将所有这样的最小环对应的变量作为冲突子句返回至 SMT 求解器。

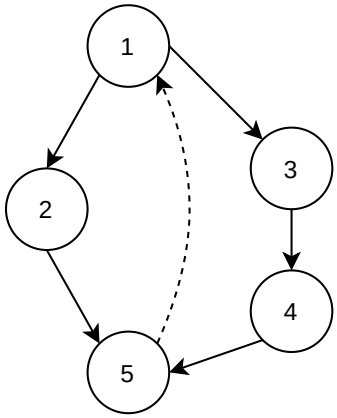

图 3-5　生成最小环对应的冲突子句示例

我们以图 3-5 为例说明找出最小环的过程。在该图中原有 $1 \to 2$，$2 \to 5$，$1 \to 3$，$3 \to 4$，$4 \to 5$ 共 5 条边，现加入 $5 \to 1$ 边并形成环。在第一步广度优先遍历中，我们得到的前驱节点分别为 $pred_2 = pred_3 = 1, pred_4 = 3, pred_5 = 2$。在第二步中，我们沿 5 号节点按前驱节点顺序分别找出 $2 \to 5$ 与 $1 \to 2$ 两条边，并加入 $5 \to 1$ 得到最小环。

3.2.4 写依赖传播

在扩展聚合图中,每一个约束 Cons 都由一对写了同一键值对的事务 $\mathrm{Txn}_1, \mathrm{Txn}_2$ 生成。构成 Cons 的两个依赖关系集合 S_1 和 S_2 分别包含了 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_2$ 以及 $\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_1$ 两个可能的依赖关系。利用这一对依赖关系涉及同一对事务且方向恰好相反的特性，我们可以在 SMT 求解过程中由当前赋值的变量推导出 Cons 应取哪一种可能性。

我们维护一传播表 $\mathrm{P} = \{\langle b \rightarrow \{b_1, \cdots, b_n\}\rangle, \cdots\}$，代表在 b 取值为真时 $b_1, \cdots, b_n$ 取值均应为真。P 的构造方式如下：

在编码约束条件的过程中，我们得到编码变量与其对应的边集合 $\mathrm{S} = \{\langle b \rightarrow \{\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_2, \cdots\}\rangle, \cdots\}$。我们首先将 S 中每个边集合惟一的写依赖边映射至对应的变量，构建出 $\mathrm{S}' = \{\langle (\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_2) \rightarrow b\rangle, \cdots\}$。然后，我们忽略 S 与 S' 中依赖关系的类型，将 S 中每一条边由 S' 映射为对应变量（若 S' 中无对应变量则从结果中去除）得到 P。

对于编码同一依赖的布尔变量 b, b'，S' 中 $\langle (\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_2) \rightarrow b\rangle$ 也有其相反方向的依赖关系 $\langle (\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_1) \rightarrow b'\rangle$，当与 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_2$ 平行的边被添加至依赖图时，如果再添加 $\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_1$ 必定构成一个环。在 SMT 求解器赋值一变量导致添加 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(\cdots)} \mathrm{Txn}_2$ 的平行边时，为了使最终的依赖图无环，必须将 b 赋值为真，b' 相应赋值为假。因此，我们在进行求解时可以利用 P 传播赋值。

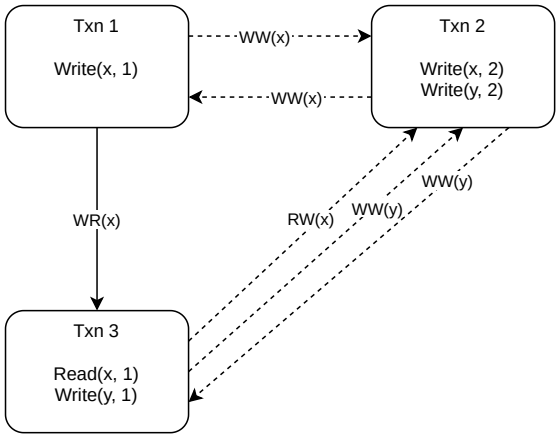

图 3-6　写依赖传播示例

我们以图 3-6 的历史为例介绍写依赖传播的过程。该图包含两个依赖

$$\mathrm{Cons}_1 = \{b_1 = \{\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_2, \mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2\}, b_2 = \{\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_2\}\}$$

$$\mathrm{Cons}_2 = \{b_3 = \{\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(y)} \mathrm{Txn}_3\}, b_4 = \{\mathrm{Txn}_3 \xrightarrow{\mathrm{WW}(y)} \mathrm{Txn}_2\}\},$$

其中 $\mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2$ 与 $\mathrm{Txn}_3 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_2$ 相互平行。假设我们尝试进行赋值 $b_1 =$ **true**，由于 $\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_3$ 同时与 $\mathrm{Txn}_3 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_2$ 和 $\mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2$ 方向相反，我们可以推导出此时必定有 $b_3 =$ **false**，$b_4 =$ **true**，否则会在依赖图中产生环。根据以上过程，我们在对该图建立传播表 $\mathrm{P} = \{b_1 \to \{b_4\}\}$，当发生赋值 $b_1 =$ **true** 时自动对 b_4 进行赋值。

3.2.5 约束剪枝

检测器在进行 SMT 求解之前会生成许多的约束。然而，许多的约束包含的两个边集合中至少有一个和已知的依赖关系冲突。这意味着我们通过已知图便可求解出这些约束的取值，而无需在 SMT 求解器中进行试错。我们在求解之前通过一个约束消除的预处理步骤来消除这些约束。

给定一个已知图 G 和一组依赖 $\mathrm{Cons} = \{\mathrm{Cons}_1, \cdots, \mathrm{Cons}_n\}$，我们首先计算出 G 的可达性关系 G^*。然后，对于每一个依赖 $\mathrm{Cons}_p = \langle either \triangleq \{\mathrm{Txn}_1 \to \mathrm{Txn}_2, \cdots\}, or\rangle$，我们对 *either* 集合中的每一条边 $\mathrm{Txn}_i \to \mathrm{Txn}_j$ 检查 G^* 是否包含 $\mathrm{Txn}_j \to \mathrm{Txn}_i$。如果包含，那么 $\mathrm{Txn}_i \to \mathrm{Txn}_j$ 选择 *either* 这种可能就会导致其中的边与已知图 G 中的边构成环。因此我们从 Cons 中去除 Cons_p，并将 *or* 集合中的边加入已知图 G 中。如果 *either* 集合不存在构成环的边，我们以同样的方式对 *or* 集合进行检查。

向已知图添加边可能导致其中出现环。我们在消除依赖集合中的依赖后检查 G 是否存在一个环。如果存在，那么导致出现环的依赖 Cons_p 无论取哪种可能性都会存在环，因此这个历史不符合可串行化隔离级别。

在求解出一部分依赖后，新加入已知图 G 的边可能导致其它依赖。此时同样可以通过上述方法解出更多的依赖。因此，我们对依赖集合 Cons 进行多次遍历并尝试求解其中的依赖，直到无法求解出更多依赖为止。

我们以图 3-7 的历史为例介绍约束剪枝的过程。该图中四个事务对主键 x 和

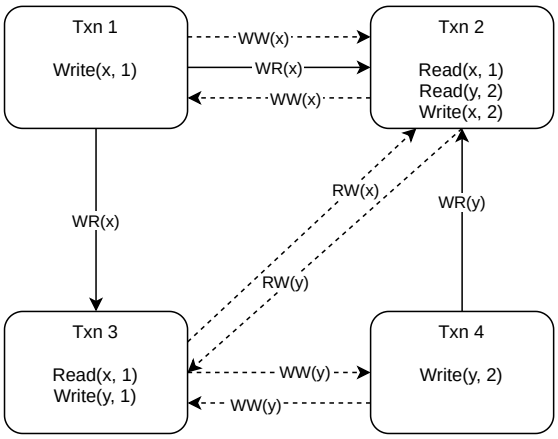

图 3-7　约束剪枝示例

y 进行了读写，我们由该历史生成约束

$$\mathrm{Cons}_1 = \{b_1 = \{\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_2, \mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2\}, b_2 = \{\mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_1\}\}$$

$$\mathrm{Cons}_2 = \{b_3 = \{\mathrm{Txn}_3 \xrightarrow{\mathrm{WW}(y)} \mathrm{Txn}_4\}, b_4 = \{\mathrm{Txn}_4 \xrightarrow{\mathrm{WW}(y)} \mathrm{Txn}_3, \mathrm{Txn}_2 \xrightarrow{\mathrm{RW}(y)} \mathrm{Txn}_3\}\}。$$

由于存在已知边 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_2$，在 $b_2 = \textbf{true}$ 的条件下依赖图中会产生环 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WR}(x)} \mathrm{Txn}_2 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_1$。因此我们解出 $b_1 = \textbf{true}, b_2 = false$，并将 $\mathrm{Txn}_1 \xrightarrow{\mathrm{WW}(x)} \mathrm{Txn}_2$ 与 $\mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2$ 两条边加入已知图。进一步地，我们由新加入的 $\mathrm{Txn}_3 \xrightarrow{\mathrm{RW}(x)} \mathrm{Txn}_2$ 与 Cons_2 中的 $\mathrm{Txn}_2 \xrightarrow{\mathrm{RW}(y)} \mathrm{Txn}_3$ 形成环可以解出 $b_3 = \textbf{true}, b_4 = \textbf{false}$。在这个例子中，我们无需进行 SMT 求解步骤便求解出了全部的约束。

3.3　程序设计与算法实现

3.3.1　开发技术选择

我们使用 C++ 语言将上述算法实现为一命令行工具。在语言标准上，我们使用目前最新的 C++ 20 标准编写项目，以利用范围、概念等新特性更简洁地实现算法的每个步骤。检测器算法中涉及到的最重要的一个数据结构是图。我们以 Boost Graph 图论库作为图数据结构的实现，使用其中的邻接表存储扩展多态图与依赖图。除此之外，我们还使用 Boost Log 日志库输出程序运行过程中产生的调试信息，使用 argparse 库处理程序参数。在项目开发过程中，我们使用 AddressSanitizer 检测程序中的内存使用错误，使用 gprof 与 perf 工具分析程序运

行状况并进行优化。该项目使用 Meson 构建系统进行编译。

本项目使用 C++ 语言进行编写主要有三点考虑。第一点在于 C++ 语言由于其接近底层机器语言的特性常常有着更高的效率，而 C++ 语言以没有额外开销的抽象为重要设计理念，在保持程序效率的同时也能高效编写出高层次的算法逻辑。例如，C++ 标准库中的可变数组、队列、哈希表等容器均以模板类的方式提供，在编译时会根据元素类型特化出单独的实现，使得编译器可以对容器操作进行最大程度的优化。C++ 的高效使得它对我们追求算法效率的这一项目来说是一个理想的编程语言。第二个原因在于由于 C++ 语言发展时间较早的特点，其生态系统包含了大量高质量的程序库。我们在编写项目的过程中可以直接使用现成的程序库，而不用重复编写代码。例如，Boost 是 C++ 的一个重要程序库，它包含了容器、日志、模板元编程等多方面的高质量实现，极大地扩展了 C++ 标准库中原有的工具。本项目使用了 Boost Graph 与 Boost Log 两个部分作为图数据结构的实现和记录日志，并在测试中使用了 Boost Test 库。此外，面向 C++ 语言的调试工具与性能分析工具也十分成熟，如本项目在编写过程中使用了 gdb 进行调试，也使用了 gprof 与 perf 两工具分析性能瓶颈。第三个原因是本项目依赖 Z3 求解器特有的用户自定义领域相关求解功能来实现对约束的求解，而 Z3 求解器同样使用 C++ 语言进行编写。虽然 Z3 求解器除 C++ 外还提供了 Java、Python 等语言的接口，但面向这些语言的接口并未提供自定义领域相关求解的功能，无法直接使用。

本项目代码在编写过程中除使用的库外还多次用到容器类型转换、日志记录等相关代码。我们将这些代码的共同之处封装为工具函数、工具类与宏函数以减少重复编码。目前的工具包括 `as_range`、`to` 两个模板函数与 `_uz`、`_z`、`_u64` 三个自定义字面量后缀。相关工具算法如列表 3-1 所示。

本项目代码中大量使用了 C++ 20 范围库编写算法，但 Boost 图论库中获得图节点与节点出边等函数均返回一对迭代器而非一个范围。因此，我们通过工具函数 `as_range` 将迭代器对转化为范围。同样地，本项目中有大量将范围转换为标准库容器的操作，而 C++ 20 未提供相应的转换函数。因此，我们编写一模板类 `ToContainer` 来封装转换逻辑，并重载其管道运算符和函数调用运算符，分别提供转换为对应容器和以相应参数构造容器并插入元素的功能。

本项目中多处用到 `size_t`、`ptrdiff_t`、`uint64_t` 三种类型的整数

```cpp
template <std::input_iterator I>
auto as_range(std::pair<I, I> p) -> decltype(auto);

template <typename C>
struct ToContiner {
  template <std::ranges::range R>
  friend auto operator|(R &&r, ToContainer) -> C;

  template <typename... Args>
  auto operator()(Args &&...args) const;
};

constexpr size_t operator"" _uz(unsigned long long n);
constexpr ptrdiff_t operator"" _z(unsigned long long n);
constexpr uint64_t operator"" _u64(unsigned long long n);

#define CHECKER_LOG_COND(level, var_name)
```

列表 3-1: 本项目中所用工具类与函数

常量，但 C++ 标准库中未提供这三种常量的字面量后缀。我们通过用户自定义后缀解决需要手动进行类型转换的问题。

本项目开发过程中需要在程序内部多处输出日志，但 Boost Log 库仅提供输出单行日志的宏函数，需要通过循环输出多个元素时使用不便。我们编写了 `CHECKER_LOG_COND` 宏函数完成绑定输出流与检查日志级别的操作，使输出日志更为便捷。

3.3.2 历史解析与预处理

在进行检测之前，我们首先需要读取需要检测的历史。SERSolver 将实现读取历史、生成读依赖、会话顺序与约束的相关操作统一实现在命名空间中。相关的数据结构与函数声明如图 3-8 所示：

检测器支持读取 dbcop[12] 生成的历史记录。dbcop 使用 bincode 格式序列化历史，其中每次操作被记录为 ⟨isWrite, key, value, success⟩ 的形式，每个事务同样包含一 success 参数表示事务是否提交成功。在解析历史时，我们忽略所有未成功执行的操作和未成功提交的事务。`parse_dbcop_history` 函数接收一 C++ 输入流，可从文件中读取历史数据。

`DependencyGraph` 类存储在预处理过程中得到的已知图，该类包含分别由读依赖、写依赖、反向依赖和会话顺序四种关系组成的子图。我们从历史中

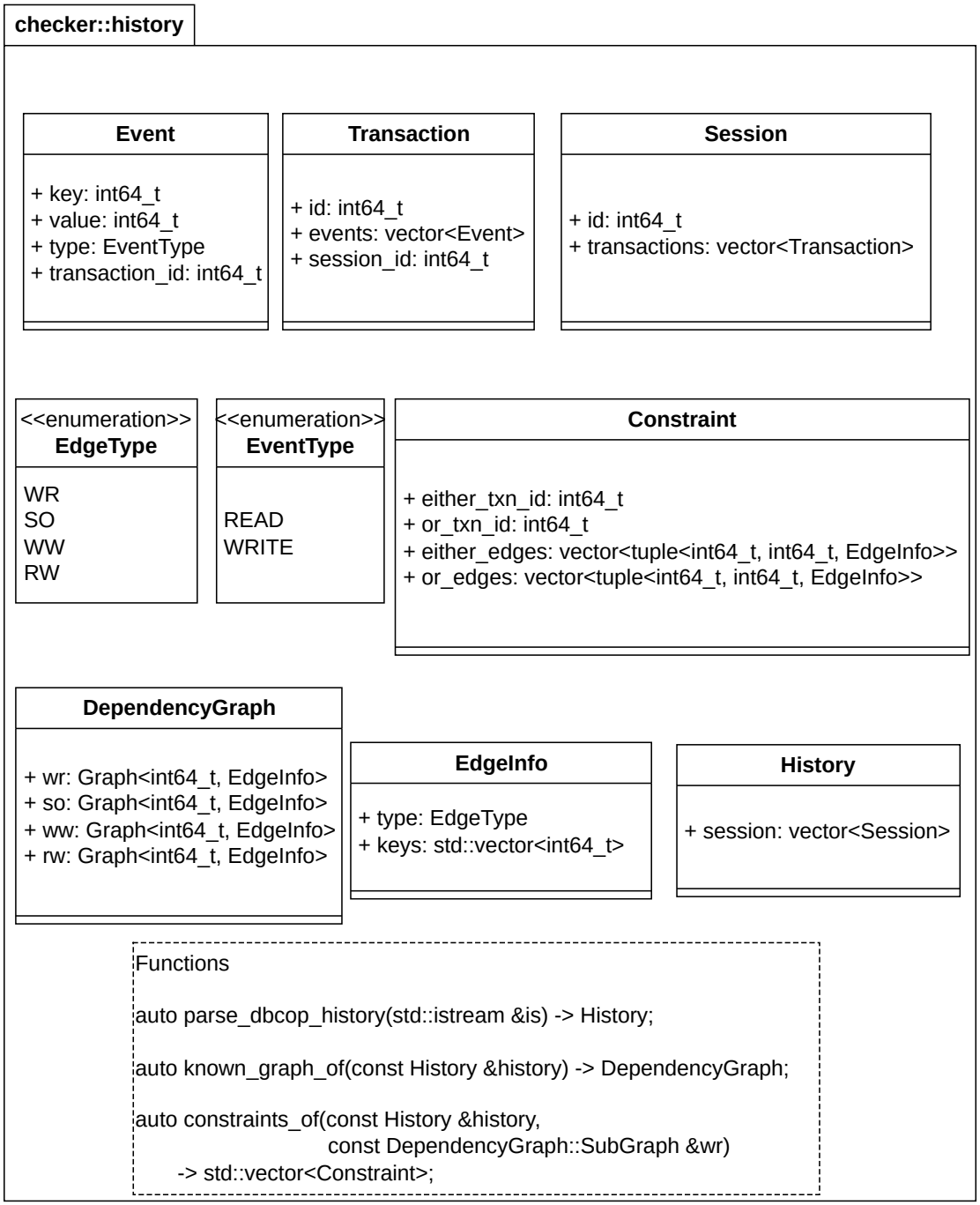

图 3-8　历史相关数据结构

解析得到的已知依赖图仅包含读依赖与会话顺序，但预处理过程中后续可以加入根据已知图推导约束取值的步骤，将推导出的关系分别加入写依赖与反向依赖子图中。

我们通过 `constriants_of` 函数构建历史中的约束。对于每一个约束，我们将其中两个写操作所在事务分别保存至 `either_txn_id` 与 `or_txn_id` 中，并使用 `either_edges` 与 `or_edges` 保存两种可能性下依赖图的依赖关系集合。

3.3.3 对接 Z3 SMT 接口

我们实现的检测器使用 Z3 作为 SMT 求解器。Z3 为领域相关求解提供的相关接口及领域相关部分的类图与领域相关求解器中的成员变量见图 3-9。

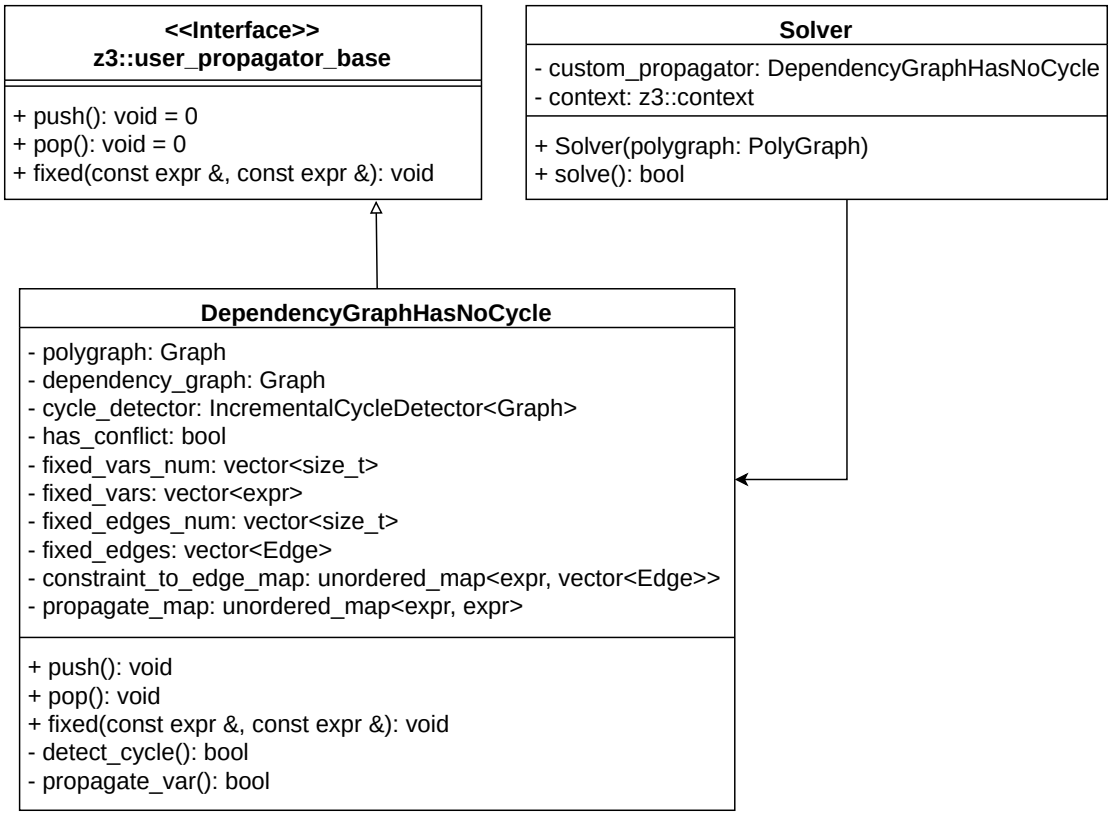

图 3-9 领域相关求解器类图

Z3 为领域相关求解器提供了 `z3::user_propagator_base` 基类作为接口，通过覆写接口方法的方式在 SMT 求解过程中注册回调函数。该接口主要有 `push`、`pop` 与 `fixed` 三个方法组成。

SMT 求解器对布尔变量进行赋值的过程主要由三部分组成。在第一部分中，求解器首先选择一变量尝试性地进行赋值。在赋值之前，领域相关求解器需要保存当前状态以在未来回滚时恢复状态。保存状态的操作通过 push 方法完成，然后通过 fixed 方法告知领域相关求解器赋值结果。在对一变量赋值之后，SMT 求解器尝试通过单元变量传播确定其它变量的值。由于被单元变量传播确定赋值的变量当前状态下无法改变赋值，对这些变量进行赋值之前无需调用 push 方法。若 SMT 求解器在尝试赋值与单元变量传播的过程中发现冲突，需要取消赋值这些变量，同时调用领域相关求解器的 pop 方法将对应的状态出栈，恢复赋值以前的结果。

我们将领域相关求解实现为一 DependencyGraphHasNoCycle 类，继承自 Z3 提供的 user_propagator_base 基类。该类实现了 push、pop、decide 三个方法，其中 push 与 pop 分别在 SMT 求解器改变变量赋值时维护依赖图数据结构。decide 方法实现领域相关部分求解，分别调用 detect_cycle 与 propagate_var 两方法实现环检测与约束条件传播。

DependencyGraphHasNoCycle 类需维护领域相关求解器的内部状态。其中，polygraph 为编码得到的扩展聚合图，dependency_graph 为当前赋值对应的依赖图，是 polygraph 的一个子图。cycle_detector 封装了进行环检测的相关状态与方法。fixed_vars 与 fixed_edges 分别为当前赋值为真的变量与对应依赖图中的边，每一个变量可对应多个边。fixed_vars_num 与 fixed_edges_num 记录了每次进行 push 操作时这两个栈的大小，用于在回滚时删除相应的变量与边。constraint_to_edge_map 记录每一变量对应的边集合，在变量赋值为真时将边集合中的边添加至依赖图中。我们使用哈希表 propagate_map 记录进行写依赖传播时每个变量传播的变量集合。

我们在领域相关求解求解部分实现的优化也会和 SMT 求解的过程产生交互，因此在实现优化的同时常常需要维护领域相关求解器的内部状态，避免产生内部状态与 Z3 求解器状态不一致导致的错误。

在增量式找环算法中，我们需要维持当前保存的拓扑序与依赖图一致。由于从依赖图中删除边的操作并不会影响拓扑序的正确性，我们在 Z3 求解器回滚时并不需要修改拓扑序；在添加一条边导致依赖图出现环时，由于我们返回冲突会导致 Z3 求解器回滚，因此我们需要在这种情况下保证增量拓扑排序算法在出

现环时不会修改拓扑序。

在写依赖传播中，我们会在 Z3 求解器对特定变量进行赋值时通过传播条件控制其单元变量传播过程，求解器对我们希望传播的变量进行赋值。然而，Z3 求解器在进行单元变量传播时并不受领域相关求解器新返回的冲突影响。因此，我们即使在之后的过程中报告了冲突仍有可能继续赋值而非回滚。增量拓扑排序算法假设在添加一条边之前的图是一个有向无关图，这一假设可能在单元变量传播的特殊条件下得到违反。为了解决这一问题，我们在领域相关求解器中额外维护一 `has_conflict` 条件，发现冲突时将该条件置为真，直到 Z3 求解器进行回滚。在 `has_conflict` 为真时，`fixed` 方法将会直接返回，避免破坏内部状态。我们同时在 `push` 方法中添加断言，保证在发现冲突的情况下 Z3 求解器不会继续尝试赋值。

3.3.4 SERSolver 工具测试

在开发过程中，我们经常需要验证某一部分算法的实现是否正确。我们通过测试的方式来保证算法实现的正确性。我们分别为两部分算法编写了测试。第一部分是整体的检测算法，我们构造一个完整的历史并执行完整的检测流程，将检测结果与预期结果进行比较；第二部分是检测器核心的增量式拓扑排序算法，我们向一个图中依次添加边并执行拓扑排序，测试得到的拓扑序与预期的顺序是否一致。我们使用了 Boost Test 库提供的框架来编写测试用例。

在对整体算法进行的测试中，我们按照违反可串行化隔离级别的方式构造不同的历史运行检测算法。其中一例如图 3-10 所示：

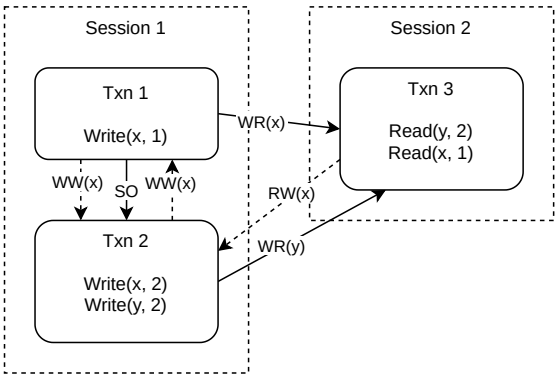

图 3-10　测试数据集中一例违反已提交读的历史

由图可见，该历史中 Txn_2 同时修改了 x 与 y 的值，但 Txn_3 只读到了对 y 的

修改，因此存在一处违反可串行化隔离级别的情况。我们预期检测算法会找出该历史中存在的冲突，并对算法输出进行检查。

在对拓扑排序算法的测试中，我们使用图 3-11 来对排序算法进行测试：

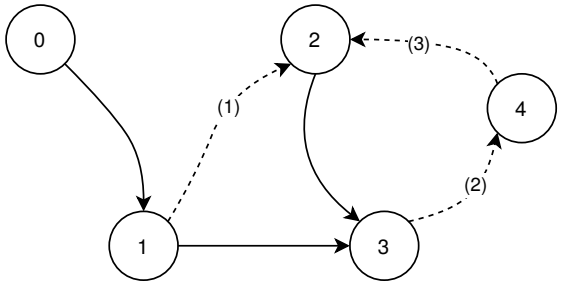

图 3-11　拓扑排序测试用例

在该图中，总共有 0 至 4 共五个节点，且创建时初始有 $0 \to 1$、$1 \to 3$、$2 \to 3$ 共三条边。然后，我们依次向该图中添加 $1 \to 2$、$3 \to 4$ 和 $4 \to 2$ 三条边。在添加第一条边后，该图中 0 至 3 号节点的顺序确定为 $0, 1, 2, 3$；在添加前两条边后，该图仍为有向无环图，且五个节点有唯一的拓扑序 $0, 1, 2, 3, 4$。我们此时检查拓扑序与预期的顺序相同。在添加第三条边后，$2, 3, 4$ 三个节点之间出现一环。此时我们检查拓扑排序算法是否报告了这个环。

对写依赖传播、找最小环、约束剪枝等其它优化，我们同样以图 3-6、图 3-5、图 3-7 等用例进行测试，并通过检查内部日志、检查返回值等方式判断这些优化方式是否发挥作用。