DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Bases de datos (1er cuatrimestre de 2009)

Grupo 3

Integrante	LU	Correo electrónico
Gonzalez, Emiliano	426/06	xjesse_jamesx@hotmail.com
Martínez, Federico	17/06	federicoemartinez@gmail.com
Ponzoni, Marta	127/06	martaponzoni@gmail.com
Sainz-Trápaga, Gonzalo	454/06	gonzalo@sainztrapaga.com.ar

En este trabajo se presenta la implementación de un analizador de transacciones para un sistema de bases de datos relacionales

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice general

Parte 1

Ejercicio 1

1.1. Enunciado

Determinar si un plan es legal:

Según el modelo de transacciones usado, deberán verificar si un plan cumple con las condiciones para ser legal. Las validaciones necesarias son las mencionadas a continuación:

Sin Locking

- Cada transacción posee como máximo un COMMIT.
- Si T tiene COMMIT, éste es el último paso de la transacción.

Locking Binario

- Cada transacción T posee como máximo un COMMIT.
- Si T tiene COMMIT, éste es el último paso de la transacción.
- Si T hace LOCK A, luego debe hacer UNLOCK A.
- Si T hace UNLOCK A, antes debe haber hecho LOCK A.
- Si T hace LOCK A, no puede volver a hacer LOCK A a menos que antes haya hecho UNLOCK A.
- Si T hace LOCK A, ninguna otra transacción T'puede hacer LOCK A hasta que T libere a A.

Locking Ternario

- Cada transacción T posee como máximo un COMMIT.
- Si T tiene COMMIT, éste es el último paso de la transacción.
- Si T hace RLOCK A o WLOCK A, luego debe hacer UNLOCK A.
- Si T hace UNLOCK A, antes debe haber hecho RLOCK A o WLOCK A.
- Si T hace RLOCK A o WLOCK A, no puede volver a hacer RLOCK A o WLOCK A a menos que antes haya hecho UNLOCK A.
- Si T hace RLOCK A, ninguna otra transacción T'puede hacer WLOCK A hasta que T libere a A.
- Si T hace WLOCK A, ninguna otra transacción T'puede hacer RLOCK A o WLOCK A hasta que T libere a A.

Como resultado, debe devolverse si el plan es legal y, en caso de no serlo, una transacción ilegal (si hubiera más de una, devolver cualquiera) y el motivo por el cual viola las restricciones.

1.2. Legalidad sin Locking

La idea del algoritmo es ir recordando (mediante un diccionario) cuando una transacción hace commit, para verificar que a partir de ese momento no pueda hacer ninguna otra operación.

El código es el siguiente:

```
1 public LegalResult analyzeLegality()
2 {
3     Map<String,Boolean> tuvoCommit = new HashMap<String, Boolean>();
4     // inicialmente nadie hizo commit
5     for (String transaccion:getTransactions()) {
6         tuvoCommit.put(transaccion,false );
7     }
8     for (Action accion : getActions()) {
9         String transaccionActual = accion.getTransaction();
10        if (accion.commits()) {
11            if (tuvoCommit.get(transaccionActual)) {
12                return new LegalResult(false,transaccionActual,"Mas de un commit");
13            }
14            tuvoCommit.put(transaccionActual, true);
15        } else {
16            if (tuvoCommit.get(transaccionActual)) {
17                return new LegalResult(false,transaccionActual,"El commit no es la ultima accion");
18            }
19        }
20    }
21    return new LegalResult(true,null,null);
22
23 }
```

1.3. Legalidad con locking binario

En este caso, además de tener un diccionario para saber quiénes ya hicieron commit, agregamos un segundo diccionario que dado un item nos dice quién lo tiene lockeado (si no está lockeado, el item no estará en el diccionario). De esta manera, podemos controlar que si una transacción hace un unlock, lo haga sobre un item que previamente había lockeado. Por último, también hay que mirar que las transacciones no dejen nada lockeado al terminar, ya que el enunciado pide que si una transacción hace un lock, luego debe hacer el unlock correspondiente.

El código es el siguiente:

```
1 public LegalResult analyzeLegality() {
2
3     Map<String,Boolean> tuvoCommit = new HashMap<String, Boolean>();
4     Map<String, String> lockeados = new HashMap<String, String>();
5     for (String transaccion:getTransactions()) {
6         tuvoCommit.put(transaccion,false );
7     }
8     for (Action accionGenerica:getActions() ) {
9         BinaryLockingAction accion = (BinaryLockingAction) accionGenerica;
10        String transaccionActual = accion.getTransaction();
11        //la transaccion hace un commit
12        if (accion.commits()) {
13            if (tuvoCommit.get(transaccionActual)) {
14                return new LegalResult(false,transaccionActual,"Mas de un commit");
15            }
```

```
16              tuvoCommit.put(transaccionActual, true);
17          }
18
19          else {
20              if (tuvoCommit.get(transaccionActual)) {
21                  if (accion.getType().equals(BinaryLockingActionType.UNLOCK)) {
22                      // si hago unlock el item tiene q estar lockeado y ademas por mi
23                      if ((!(lockeados.containsKey(accion.getItem())) || lockeados.get(accion.getItem()) != ↩
24                          transaccionActual) {
25                        return new LegalResult(false,transaccionActual, "Intento de unlock de un item que no
tiene ↩
26                                                lockeado");
27                      }
28
29                  } else {
30                      return new LegalResult(false,transaccionActual,"Hay una accion que no es un unlock
despues de un ↩
31                                             commit");
32                  }
33              }
34              // Si hago lock tengo que ver que el item no este lockeado previamente
35              if (accion.getType().equals(BinaryLockingActionType.LOCK)) {
36                  if (lockeados.containsValue(accion.getItem())) {
37                      return new LegalResult(false,transaccionActual,"Intento de lockear un item lockeado");
38                  } else {
39                      lockeados.put(accion.getItem(), transaccionActual);
40                  }
41              }
42
43              else if (accion.getType().equals(BinaryLockingActionType.UNLOCK)) {
44                  // si hago unlock el item tiene q estar lockeado y ademas por mi
45                if ((!(lockeados.containsKey(accion.getItem())) || lockeados.get(accion.getItem()) != transac-
cionActual) ↩
46                      {
47                       return new LegalResult(false,transaccionActual, "Intento de unlock de un item que no
tiene lockeado") ↩
48                                             ;
49                  } else {
50                      lockeados.remove(accion.getItem());
51                  }
52              }
53          }
54      }
55      // tiene que quedar todo sin lockear
56      if (lockeados.keySet().isEmpty()) {
57          return new LegalResult(true,null,null);
58      } else {
59       return new LegalResult(false, lockeados.values().iterator().next(), lockeados.values().iterator().next().concat( ↩
60                              " no hace unlock de todo lo que lockeo"));
61      }
62  }
63
```

1.4. Legalidad con locking ternario

En este caso es necesario guardar más información de estado, ya que varias transacciones pueden hacer Rlock sobre un item al mismo tiempo, o una transacción puede hacer Rlock y luego Wlock de un mismo item, entre otros casos a tener en cuenta. Por lo tanto decidimos utilizar una *inner class* ControlAnalisisTernario que se encarga de recordar el estado de la ejecución, conociendo así qué items estan lockeados, con qué tipo de lock y cual es la transacción responsable,

chequeando además que los locks se puedan realizar y haciendo otras verificaciones. En esta implementación soportamos el "lock upgrade" (si una transacción tiene un RLock sobre un item, puede a continuación pedir un WLock sobre el mismo item). Además, chequeamos que al finalizar las ejecuciones se liberen todos los items lockeados, ya que el enunciado dice: *Si T hace RLOCK A o WLOCK A, luego debe hacer UNLOCK A*. Esto se chequea con el metodo estadoLegal de la clase ControlAnalisisTernario.

El código de esta clase es:

```
1  private class ControlAnalisisTernario {
2      private Map<String,Set<String> > rlockeados; //Item -> conj de transacciones
3      private Map<String,String> wlockeados; //Item -> Transaccion
4
5      public ControlAnalisisTernario() {
6          rlockeados = new HashMap<String, Set<String> >();
7          wlockeados = new HashMap<String, String>();
8      }
9      // un estado es legal (en el sentido de posible estado final) si
10     // ninguna transaccion tiene nada loqueado
11     public Boolean estadoLegal() {
12         return rlockeados.isEmpty() && wlockeados.isEmpty();
13     }
14
15     // Devuelve una transaccion que tiene algun item lockeado
16     public String transaccionConLock() {
17         if (!wlockeados.isEmpty()) {
18             return wlockeados.values().iterator().next();
19         } else if (!rlockeados.isEmpty()) {
20             return rlockeados.get(rlockeados.keySet().iterator().next()).iterator().next();
21         } else {
22             return null;
23         }
24
25     }
26     public Boolean estaRLockeado(String item) {
27         if ( rlockeados.containsKey(item)) {
28             return true;
29         } else {
30             return false;
31         }
32     }
33     public Boolean estaWLockeado(String item) {
34         if ( wlockeados.containsKey(item)) {
35             return true;
36         } else {
37             return false;
38         }
39     }
40     public Boolean estaLockeado(String item) {
41         return (rlockeados.containsKey(item) || wlockeados.containsKey(item));
42     }
43     public Boolean puedeRLockear(String item, String transaccion) {
44         /* puede rlockear si:
45          * - el item no esta wlockeado
46          * - la transaccion no lo tiene ya rlockeado
47          */
48         if (wlockeados.containsKey(item)) {
49             return false;
50         }
51         if (rlockeados.containsKey(item)) {
52             if (rlockeados.get(item).contains(transaccion)) {
53                 return false;
54             }
```

```
55          }
56          return true;
57      }
58      public Boolean puedeWLockear(String item, String transaccion) {
59          /* puede wlockear si:
60           * - el item no esta wlockeado
61           *- si el item esta rlockeado, solo lo esta por la transaccion
62           */
63          if (wlockeados.containsKey(item)) {
64              return false;
65          }
66          if (rlockeados.containsKey(item)) {
67              if (rlockeados.get(item).size() == 1 && rlockeados.get(item).contains(transaccion)) {
68                  return true;
69              } else {
70                  return false;
71              }
72          }
73          return true;
74      }
75
76      public void wlockear(String item, String transaccion) {
77          if (puedeWLockear(item,transaccion)) {
78              wlockeados.put(item, transaccion);
79              if (rlockeados.containsKey(item)) {
80                  rlockeados.remove(item);
81              }
82          }
83      }
84
85      public void rlockear(String item, String transaccion) {
86          if (puedeRLockear(item, transaccion)) {
87              if (rlockeados.containsKey(item)) {
88                  rlockeados.get(item).add(transaccion);
89              } else {
90                  Set<String> s = new HashSet<String>();
91                  s.add(transaccion);
92                  rlockeados.put(item, s );
93              }
94          }
95      }
96
97      public Boolean puedeUnlockear(String item, String transaccion) {
98          /* puede wlockear si:
99           * - la transaccion lo tiene rlockeado al item
100           * - la transaccion lo tiene wlockeado
101           */
102          return
103              ((estaRLockeado(item) && rlockeados.get(item).contains(transaccion)) ||
104               (estaWLockeado(item) && wlockeados.get(item) == transaccion));
105      }
106
107      public void unlockear(String item, String transaccion) {
108          if (puedeUnlockear(item, transaccion)) {
109              if (estaWLockeado(item)) {
110                  wlockeados.remove(item);
111              } else {
112                  rlockeados.get(item).remove(transaccion);
113                  if (rlockeados.get(item).isEmpty()) {
114                      rlockeados.remove(item);
115                  }
116              }
117          }
```

```
118
119     }
120
121 }
122
```

Usando esta clase, lo que hacemos es verificar los commit, y para cualquier operación de lock o unlock, consultar al control si se puede llevar a cabo, o si la operación es ilegal.

El código es el siguiente:

```
1  public LegalResult analyzeLegality() {
2
3      ControlAnalisisTernario control = new ControlAnalisisTernario();
4      Set<String> yaComiteo = new HashSet<String>();
5      for (Action accionGenerica: getActions()) {
6
7          TernaryLockingAction accion = (TernaryLockingAction) accionGenerica;
8          String transaccion = accion.getTransaction();
9          String item = accion.getItem();
10         // Chequeo de varios commits
11      if (yaComiteo.contains(transaccion) && !accion.getType().equals(TernaryLockingActionType.UNLOCK))
{
12             return new LegalResult(false,transaccion,transaccion.concat(" intenta realizar operaciones que
no son unlock ↩
13                                 despues del commit"));
14         }
15
16         // caso commit
17         if (accion.getType().equals(TernaryLockingActionType.COMMIT)) {
18             yaComiteo.add(transaccion);
19         }
20
21         // caso rlock
22         else if (accion.getType().equals(TernaryLockingActionType.RLOCK)) {
23             if (control.puedeRLockear(item, transaccion)) {
24                 control.rlockear(item, transaccion);
25             } else {
26                     return new LegalResult(false, transaccion, transaccion.concat(" intenta hacer un rlock
invalido"));
27             }
28         }
29
30         // caso wlock
31         else if (accion.getType().equals(TernaryLockingActionType.WLOCK)) {
32             if (control.puedeWLockear(item, transaccion)) {
33                 control.wlockear(item, transaccion);
34             }
35
36             else {
37                     return new LegalResult(false, transaccion, transaccion.concat(" intenta hacer un wlock
invalido"));
38             }
39
40         }
41
42         // caso unlock
43         else {
44             if (control.puedeUnlockear(item, transaccion)) {
45                 control.unlockear(item, transaccion);
46
```

```
47              } else {
48                  return new LegalResult(false, transaccion, transaccion.concat(" intenta hacer un unlock
invalido"));
49              }
50          }
51
52      }
53
54      // chequeamos que quedaran todos los items liberados
55      if (control.estadoLegal()) {
56          return new LegalResult(true,null,null);
57      } else {
58          String transaccion = control.transaccionConLock();
59          return new LegalResult(false,transaccion, transaccion.concat(" tiene un lock que no libera"));
60      }
61  }
```

Parte 2

Ejercicio 2

2.1. Enunciado

Determinar si un plan es serial:

Un plan es serial cuando las acciones de cada transacción aparecen consecutivas, es decir, no debe haber un entrelazamiento entre las acciones de diferentes transacciones. Notar que para los 3 modelos, la verificación es la misma y se debe resolver el problema en la forma más general posible. Como resultado, debe devolverse si el plan es serial y, en caso de no serlo, una transacción que viole la restricción (si hubiera más de una, devolver cualquiera).

2.2. Resolución

Lo que hacemos es tomar un conjunto donde vamos guardando qué transacciones ya fueron revisadas. Tenemos una transacción actual, y en el momento en que la próxima acción pertenece a otra transacción, guardamos a la que teníamos como actual en el diccionario y cambiamos de actual. Si esta transacción ya estaba en el diccionario es porque tiene operaciones intercaladas y por lo tanto la historia no era serial. Si revisamos con éxito todas las acciones es porque no había operaciones intercaladas, y por lo tanto la historia era serial.

El codigo es el siguiente:

```
1 public SerialResult analyzeSeriality() {
2     /* A medida que encontramos una nueva transaccion la guardamos en el
3        diccionario, y si vuelve a aparecer quiere decir que no era serial */
4     Set<String> yaRevisados = new HashSet<String>();
5     String actual = null;
6     for (Action accion:getActions()) {
7         if (actual == null) {
8             actual=accion.getTransaction();
9             yaRevisados.add(actual);
10        }
11        String nuevaTransaccion = accion.getTransaction();
12        if (nuevaTransaccion != actual) {
13            if (yaRevisados.contains(nuevaTransaccion)) {
14                return new SerialResult(false,nuevaTransaccion,null);
15            } else {
16                actual = nuevaTransaccion;
17                yaRevisados.add(actual);
18            }
19        }
20
```

```
21     }
22     return new SerialResult(true,null,null);
23 }
```

Parte 3

Ejercicio 3

3.1. Enunciado

Determinar si un plan es serializable:

Un plan se dice que es serializable cuando su ejecución produce el mismo resultado que una ejecución serial. A diferencia de los planes seriales, aquí se permite el entrelazamiento de acciones de diferentes transacciones, pero al mismo tiempo, se imponen restricciones sobre cómo entrecruzar transacciones. Típicamente, se construye un grafo con arcos direccionales para determinar la precedencia de las transacciones y, si el grafo no tiene ciclos, el plan se considera serializable. Dependiendo del tipo de modelo de transacciones, la construcción del grafo varía y a continuación presentamos en detalle cuándo deben agregarse arcos entre transacciones T1 y T2 según el modelo:

Sin Locking:

- T1 lee un ítem A y T2 luego escribe A
- T1 escribe un ítem A y T2 luego lee A
- T1 escribe un ítem A y T2 luego escribe A

Locking Binario:

- T1 hace LOCK de un ítem A y luego T2 hace LOCK de A

Locking Ternario:

- T1 hace RLOCK de un ítem A y T2 luego hace WLOCK de A
- T1 hace WLOCK de un ítem A y T2 luego hace RLOCK de A
- T1 hace WLOCK de un ítem A y T2 luego hace WLOCK de A

Una vez construido el grafo en forma particular dependiendo del modelo [1] , se deberá verificar si un grafo tiene ciclos utilizando Teoría de Grafos. Como resultado, debe devolverse el grafo e indicar si el plan es serializable. Además, si es serializable, retornar una lista con las posibles ejecuciones y, en caso contrario, presentar un ciclo (si hubiera más de uno, devolver cualquiera).

3.2. Grafo sin locking

La idea que tuvimos fue separar primero las acciones por item, para facilitar la búsqueda de dependencias. Una vez que hicimos esto recorremos para cada item todas sus acciones viendo qué ejes se generan en el grafo de dependencias.

[1] Como optimización a la hora de implementarlo, pediremos que se omitan arcos que se deduzcan por transitividad.

```
1 public ScheduleGraph buildScheduleGraph() {
2
3
4     ScheduleGraph sg = new ScheduleGraph();
5     for (String transaccion: getTransactions()) {
6         sg.addTransaction(transaccion);
7     }
8
9     // para facilitar el recorrido vamos a separar las acciones por items
10
11   Map<String,List<Par<Action,Integer> > > operXItem = new HashMap<String, List<Par<Action,Integer>
> >();
12     for (String item:getItems()) {
13         operXItem.put(item, new LinkedList<Par<Action,Integer>>());
14     }
15     int i = 1;
16
17     for (Action accion : getActions()) {
18         if (!accion.commits()) {
19             // el i es el indice en la historia, necesario para armar los ejes
20             operXItem.get(accion.getItem()).add(new Par<Action, Integer>(accion, i));
21
22         }
23         i=i+1;
24     }
25
26     // usamos un diccionario para wvitar ejes repetidos
27     Map<String,Set<String>> ejesYaPuestos = new HashMap<String, Set<String>>();
28     for (String transaccion:getTransactions()) {
29         ejesYaPuestos.put(transaccion, new HashSet<String>());
30     }
31
32     for (String item: getItems()) {
33         List<Par<Action,Integer>> operaciones = operXItem.get(item);
34         int tam = operaciones.size();
35         //recorremos cada accion
36         for (int j = 0; j < tam; j++) {
37             Action operacionActual = operaciones.get(0).fst;
38             String transaccionActual = operacionActual.getTransaction();
39             Integer indiceActual = operaciones.get(0).snd;
40             operaciones.remove(0); // removemos la que ya miramos
41
42             //dada una accion empezamos a mirar hacia adelante
43             //para ver si genera algun eje
44             for (int k =0; k < operaciones.size(); k ++) {
45                 // tratamos de agregar un eje para la operacion actual,
46                 // y ademas vemos si hay que seguir agregando
47                 Action proximaAccion = operaciones.get(k).fst;
48                 String proximaTransaccion = proximaAccion.getTransaction();
49                 Integer proximoIndice = operaciones.get(k).snd;
50
51                 if (proximaTransaccion == transaccionActual) {
52                     /* no estoy en el caso caso donde tengo algo asi w1,r1,r2, tengo que saltar r1 y seguir
buscando */
53                     if (!operacionActual.writes() && proximaAccion.reads()) {
54                         break;
55                     }
56
57                 }
58                 /* la transaccion que viene ahora es distinta */
59                 else if (operacionActual.writes()) {
60                     //caso: write vs *
```

```
61                  //pongo el eje si hace falta
62                  if (!ejesYaPuestos.get(transaccionActual).contains(proximaTransaccion)) {
63                      sg.addArc(new ScheduleArc(transaccionActual,
64                                                proximaTransaccion,
65                                                indiceActual,proximoIndice));
66                      ejesYaPuestos.get(transaccionActual).add(proximaTransaccion);
67
68                  }
69                  //caso write vs write: hay que parar de buscar
70                  //caso write vs read: hay que seguir buscando
71                  if (proximaAccion.writes()) {
72                      break;
73                  }
74
75              } else {
76                  // caso: read vs write
77                  // pongo el eje y tengo que parar
78                  if (proximaAccion.writes()) {
79                      if (!ejesYaPuestos.get(transaccionActual).contains(proximaTransaccion)) {
80                          sg.addArc(new ScheduleArc(operacionActual.getTransaction(),
81                                                    proximaTransaccion,
82                                                    indiceActual,proximoIndice));
83                          ejesYaPuestos.get(transaccionActual).add(proximaTransaccion);
84                      }
85                      break;
86                  }
87                  // caso: read vs read
88                  // sigo sin poner ningun eje
89              }
90
91          }
92
93      }
94  }
95  return sg;
96 }
```

3.3. Grafo para locking binario

En este caso también separamos las acciones por items, dejando además solo las acciones de tipo lock, ya que son las que generan ejes. Usamos un diccionario para evitar ejes repetidos. Vamos recorriendo cada acción de cada item y agregamos los ejes que genera, evitando agregar ejes de más (los transitivos no deseados).

```
1 public ScheduleGraph buildScheduleGraph() {
2     ScheduleGraph sg = new ScheduleGraph();
3     for (String transaccion: getTransactions()) {
4         sg.addTransaction(transaccion);
5     }
6
7     Map<String,List<Par<BinaryLockingAction,Integer> > > operXItem;
8     operXItem = new HashMap<String, List<Par<BinaryLockingAction,Integer> > >();
9
10    for (String item:getItems()) {
11        operXItem.put(item, new LinkedList<Par<BinaryLockingAction,Integer>>());
12    }
13    int i = 1;
14    // separo acciones por item para hacer mas facil la busqueda de ejes
15    for (Action accionGenerica : getActions()) {
```

```
16          BinaryLockingAction accion = (BinaryLockingAction) accionGenerica;
17
18          // solo guardamos los locks
19          if ( accion.getType().equals(BinaryLockingActionType.LOCK)) {
20
21              // al igual que antes, el indice sirve para saber en
22              // que parte de la historia esta la accion
23              operXItem.get(accion.getItem()).add(new Par<BinaryLockingAction, Integer>(accion, i));
24
25          }
26          i=i+1;
27      }
28
29      //diccionario para evitar ejes repetidos
30      Map<String,Set<String>> ejesYaPuestos = new HashMap<String, Set<String>>();
31
32      for (String transaccion:getTransactions()) {
33          ejesYaPuestos.put(transaccion, new HashSet<String>());
34      }
35
36      for (String item: getItems()) {
37          List<Par<BinaryLockingAction,Integer>> operaciones = operXItem.get(item);
38          int tam = operaciones.size();
39          for (int j = 0; j < tam-1; j++) {
40              BinaryLockingAction operacionActual = operaciones.get(0).fst;
41              Integer indiceActual = operaciones.get(0).snd;
42              String transaccionActual = operacionActual.getTransaction();
43              operaciones.remove(0);
44              String proximaTransaccion =operaciones.get(0).fst.getTransaction();
45              /* si es la misma transaccion, paro de buscar ejes porque los agarro
46               * la siguiente accion
47               */
48              if (!(proximaTransaccion == transaccionActual)) {
49                  //si se genera un eje nuevo, lo agrego
50                  if (!ejesYaPuestos.get(transaccionActual).contains(proximaTransaccion)) {
51
52                      sg.addArc(new ScheduleArc(transaccionActual,
53                                                proximaTransaccion,
54                                                indiceActual, operaciones.get(0).snd));
55                      ejesYaPuestos.get(transaccionActual).add(proximaTransaccion);
56                  }
57              } else {
58                  break;
59              }
60          }
61      }
62      return sg;
63  }
```

3.4. Grafo para locking ternario

Colocamos un nodo por cada transacción. Luego, para cada acción nos fijamos con qué acciones conflictúa y si aún no existe un eje entre las transacciones correspondientes a esas acciones, lo colocamos.

Dos acciones A1 y A2 conflictúan si no pertenecen a la misma transacción y cumplen alguna de estas condiciones:

- A1 hace RLock o WLock de X y luego A2 es la próxima que hace WLock de X
- A1 hace WLock de X y A2 hace RLock de X antes que cualquier otra haga WLock de X

```
1  public ScheduleGraph buildScheduleGraph() {
2      ScheduleGraph graph = new ScheduleGraph();
3
4      //aca se guarda la info de los ejes ya puestos para no repetirlos
5      Map<String, Set<String> > edges = new HashMap<String, Set<String> >();
6
7      for (String transaccion: getTransactions()) {
8          //se agrega un nodo en el grafo
9          graph.addTransaction(transaccion);
10         //se agrega entrada en el mapa
11         edges.put(transaccion, new HashSet<String>());
12     }
13
14     for (Action actualAction: getActions()) {
15
16         String actualTransaction = actualAction.getTransaction();
17
18         //Si Ti hace WLock de X y Tj (i<>j) hace RLock de X, (antes que cualquier otra
19         //haga WLock de X), se hace un arco Ti -> Tj
20         if (actualAction.writes()) {
21             for (Action action: getActions()) {
22
23                 String transaction = action.getTransaction();
24                 if (canBeAConflictBetween(actualAction, action,
25                                           actualTransaction, transaction)) {
26
27                     if (action.reads() &&
28                             !edges.get(actualTransaction).contains(transaction)) {
29
30                         addEdge(graph, edges, actualAction, action,
31                                 actualTransaction, transaction);
32                     } else if (action.writes()) {
33                         break;
34                     }
35                 }
36             }
37         }
38         //Si Ti hace RLock o WLock de X, y luego Tj es la próxima que hace WLock de X
39         //(i<>j), se hace un arco Ti -> Tj
40         for (Action action: getActions()) {
41             String transaction = action.getTransaction();
42
43             if (canBeAConflictBetween(actualAction, action, actualTransaction,
44                                       transaction) &&   action.writes() &&
45                     !edges.get(actualTransaction).contains(transaction)) {
46
47                 addEdge(graph, edges, actualAction, action,
48                         actualTransaction, transaction);
49                 break;
50             }
51         }
52     }
53     return graph;
54 }
55
56 private boolean canBeAConflictBetween(Action actualAction, Action action,
57                                       String actualTransaction, String transaction) {
58     return
59         //si una accion ocurre despues de la otra
60         getActions().indexOf(action) >
61         getActions().indexOf(actualAction) &&
62         //pertenecen a transacciones distintas
```

```
63          actualTransaction != transaction &&
64          //y actuan sobre el mismo item
65          action.getItem() == actualAction.getItem();
66  }
67
68  private void addEdge(ScheduleGraph graph, Map<String, Set<String>> edges,
69                       Action actualAction, Action action, String actualTransaction,
70                       String transaction) {
71      //se agrega el eje al grafo
72      ScheduleArc arc = new ScheduleArc(actualTransaction,transaction,
73                                        getActions().indexOf(actualAction), getActions().indexOf(action));
74      graph.addArc(arc);
75      //se guarda informacion para no volver a poner un eje entre esas transacciones
76      Set<String> set = edges.get(actualTransaction);
77      set.add(transaction);
78      edges.put(actualTransaction, set);
79  }
80
```

3.5. Análisis del grafo

Para analizar el grafo, creamos en Schedule una *inner class* AnalizadorDeGrafos que se encarga de realizar las operaciones para determinar si un grafo es acíclico, y en caso afirmativo dar todos los ordenes topológicos del mismo, que corresponden a todas las posibles ejecuciones seriales equivalentes a la historia, o exhibir un ciclo en caso contrario.

Para encontrar un ciclo usamos *BFS* en busca de ejes de retroceso. Comenzamos por un nodo y lo marcamos como visitado, y hacemos BFS de forma recursiva sobre sus hijos. Si en algún momento tenemos que pasar a un nodo ya visitado, vemos si este nodo está marcado como seguro: si es así, no hay peligro de ciclo con ese nodo, de otro modo tenemos un ciclo. Un nodo se marca como seguro cuando ya se terminó de visitar (es decir, ya se hizo BFS para todos sus hijos). La idea es que si tenemos un eje a un nodo ya visitado y este no está marcado como seguro, es porque todavía no se revisaron todos sus hijos y por lo tanto hay por lo menos un camino desde alguno de sus hijos hasta el nodo actual. Por ejemplo:

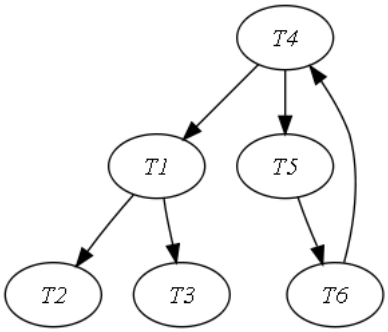
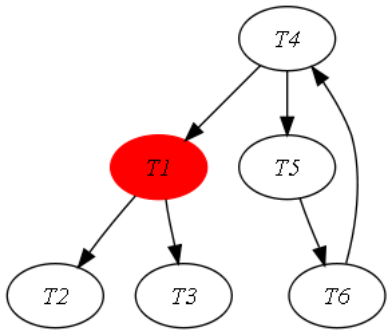
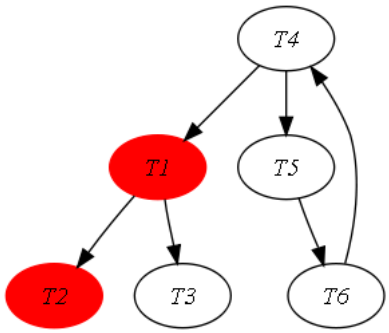


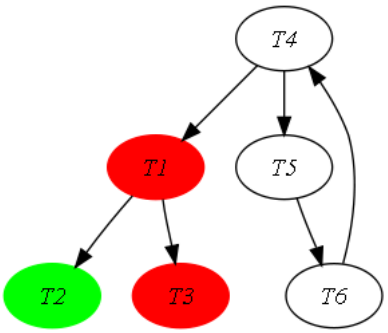
Figura 3.1: Ejemplo de aplicación del algoritmo de busqueda de ciclos



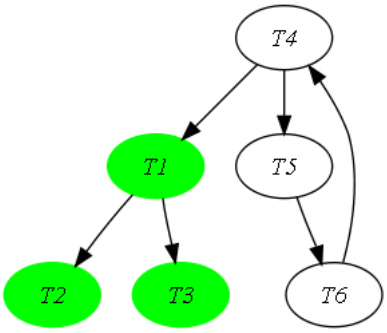
(a) Comenzamos por t1, la marcamos como ya visitada, pero no es segura



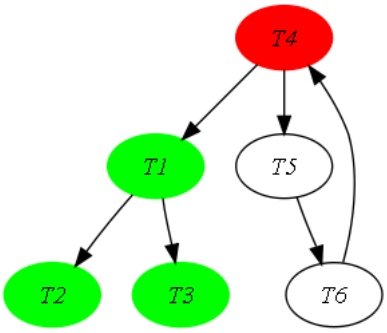
(b) Pasamos a t2, también la marcamos como visitada



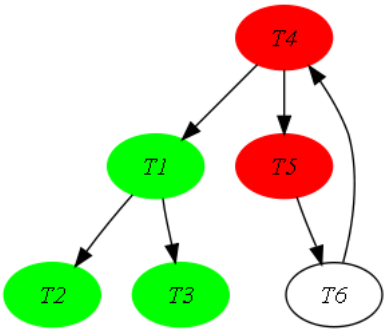
(c) Como t2 no tiene más hijos, la marcamos como segura y pasamos a t3



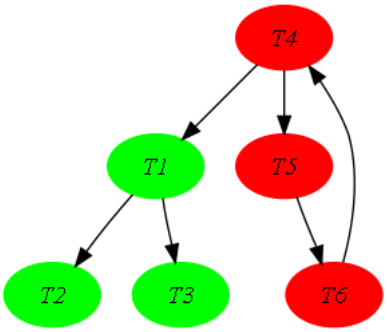
(d) t3 tampoco tiene más hijos, entonces es seguro



(e) La proxima transición sin visitar es t4. t4 tiene un eje a t1 que está ya visitada, pero t1 es segura



(f) Pasamos a t5 y la marcamos también como visitada



(g) llegamos a t6, lo marcamos como visitado. t6 tiene un eje a t4 que está visitado pero no es seguro, por lo tanto hay un ciclo

Figura 3.2: Ejemplo de aplicación del algoritmo

El código del algoritmo está en los métodos esAciclico y el método auxiliar hayUnCiclo. Para obtener un ciclo, como sabemos que el grafo no es acíclico, usamos nuevamente BFS pero agregando cada nodo por donde pasams a una lista. Si llegamos a un nodo ya visitado y no seguro es porque encontramos un ciclo.

Algoritmo 1 Devuelve un ciclo del grafo

```
1: nodosVisitados = ∅
2: seguros = ∅
3: Para cada nodo del grafo hacer
4:    Si  nodo ∉ nodosVisitados entonces
5:       listaRes = []
6:       buscar ciclo a partir del nodo y ponerlo en listaRes
7:    Fin si
8: Fin para
```

Algoritmo 2 busca un ciclo en un grafo a partir de nodo

```
Parámetros: un nodo y una lista donde poner el resultado
1: nodosVisitado = nodosVisitados ⋃ nodo
2: Para cada otro nodo adyacente al nodo hacer
3:    Si el otro nodo ya fue visitado pero no es seguro entonces
4:       agregar a la lista al otro nodo
5:       agregar a la lista al nodo
6:       Devolver
7:    Fin si
8:    Si el otro nodo no fue visitado entonces
9:       buscar ciclo a partir del otro nodo
10:   Fin si
11:   Si la lista que me devolvio la busqueda de un ciclo no esta vacia entonces
12:      Si  si el primer elemento de la lista es igual al ultimo (se completo el ciclo) entonces
13:         Devolver
14:      Si no
15:         me agrego a la lista
16:         Devolver
17:      Fin si
18:   Fin si
19: Fin para
20: seguros = seguros ⋃ nodo
```

Finalmente, para el caso en el que tenemos que devolver todos los órdenes topológicos (posibles ejecuciones), el algoritmo es el siguiente:

Algoritmo 3 devuelve todos los ordenes topologicos de un grafo dirigido

```
1: Si  el grafo tiene un solo nodo entonces
2:    Devolver  [ [ nodo ] ]
3: Fin si
4: Para cada nodo con grado de entrada=0 hacer
5:    sacar al nodo y sus ejes del grafo
6:    armar todos los ordenes topologicos del grafo que queda
7:    para cada orden topologico obtenido, agregar adelante al nodo
8: Fin para
9: Devolver  la lista de listas con los ordenes topologicos
```

El código de estos algoritmos se encuentra en la clase interna AnalizadorDeGrafos. El código completo es el siguiente:

```
1 private class AnalizadorDeGrafos {
2
3    /**
4    * grafo a bindear para analizar
5    */
```

```
6   ScheduleGraph grafo;
7   /**
8    *  Lista de adyacencias del grafo, para hacer bfs y buscar ciclos
9    */
10  Map<String,List<String>> adyacencias;
11  Map<String,List<String>> adyacenciasReversas;
12
13  /**
14   *
15   * @param g : grafo a analizar
16   *
17   */
18  public AnalizadorDeGrafos(ScheduleGraph g) {
19      grafo = g;
20      adyacencias = new HashMap<String, List<String>>();
21      adyacenciasReversas = new HashMap<String, List<String>>();
22      for (String transaccion:grafo.getTransactions()) {
23          adyacencias.put(transaccion, new LinkedList<String>());
24          adyacenciasReversas.put(transaccion, new LinkedList<String>());
25
26      }
27      for (ScheduleArc eje: g.getArcs()) {
28          adyacencias.get(eje.getStartTransaction()).add(eje.getEndTransaction());
29          adyacenciasReversas.get(eje.getEndTransaction()).add(eje.getStartTransaction());
30      }
31  }
32
33  /**
34   *
35   * @return True si no hay ciclos en el grafo
36   */
37  public Boolean esAciclico() {
38      Set<String> yaVisitados = new HashSet<String>();
39      List<String> nodos = grafo.getTransactions();
40
41      Set<String> yaSeguros = new HashSet<String>();
42      for (String nodo:nodos) {
43          if (! yaVisitados.contains(nodo)) {
44              if (hayCiclo(nodo,yaVisitados,yaSeguros)) {
45                  return false;
46              }
47          }
48      }
49      return true;
50  }
51
52  /**
53   *
54   * @param nodo : desde donde parto para buscar un ciclo
55   * @param yaVisitados : nodos que ya fueron visitados
56   * @param yaSeguros : nodos que ya fueron revisados y que no estan en un ciclo
57   * @return True si encontro un ciclo
58   */
59  private boolean hayCiclo(String nodo, Set<String> yaVisitados,
60                           Set<String> yaSeguros) {
61      if (yaVisitados.contains(nodo)) {
62          return false;
63      }
64      yaVisitados.add(nodo);
65      boolean res = false;
66      for (String nodo2: adyacencias.get(nodo)) {
67          if (yaVisitados.contains(nodo2)) {
68              if (!yaSeguros.contains(nodo2) ) {
```

```
69                  return true;
70              }
71          } else {
72              res = res || hayCiclo(nodo2, yaVisitados, yaSeguros);
73          }
74
75      }
76
77      yaSeguros.add(nodo);
78      return res;
79  }
80  /**
81   *
82   * @return un ciclo del grafo si este tiene uno, null sino
83   */
84  public List<String> getCiclo() {
85      Set<String> yaVisitados = new HashSet<String>();
86      List<String> nodos = grafo.getTransactions();
87
88      List<String> res = new LinkedList<String>();
89      Set<String> yaSeguros = new HashSet<String>();
90      for (String nodo:nodos) {
91          if (! yaVisitados.contains(nodo)) {
92              res = armarCiclo(nodo,yaVisitados,yaSeguros);
93              if (! res.isEmpty()) {
94                  return res;
95              }
96
97          }
98      }
99      return null;
100 }
101 /**
102  *
103  * @param nodo : nodo desde el cual se trata de construir un cilo
104  * @param yaVisitados : nodos ya visitados
105  * @param yaSeguros : nodos revisados que no estan en un ciclo
106  * @return un ciclo si lo encuentra, lista vacia sino
107  */
108 private List<String> armarCiclo(String nodo, Set<String> yaVisitados,
109                                 Set<String> yaSeguros) {
110     if (yaVisitados.contains(nodo)) {
111         return new LinkedList<String>();
112     }
113     yaVisitados.add(nodo);
114     List<String> res = new LinkedList<String>();
115     for (String nodo2: adyacencias.get(nodo)) {
116         if (yaVisitados.contains(nodo2)) {
117             if (!yaSeguros.contains(nodo2) ) {
118                 res.add(nodo2);
119                 res.add(0,nodo);
120                 break;
121             }
122         } else {
123             res = armarCiclo(nodo2, yaVisitados, yaSeguros);
124             if (! res.isEmpty()) {
125                 if (res.get(0) != res.get(res.size()-1)) {
126                     res.add(0,nodo);
127                 }
128                 break;
129             }
130         }
131
```

```
132         }
133
134         yaSeguros.add(nodo);
135         return res;
136     }
137
138     public List<List<String>> getEjecuciones() {
139         List<String> nodos = grafo.getTransactions();
140      Map<String,List<String>> adyacenciasAux = new HashMap<String, List<String>>(adyacenciasReversas);
141         return getEjecucionesAux(nodos,adyacenciasAux);
142
143     }
144
145     private List<List<String>> getEjecucionesAux(List<String> nodos,
146             Map<String, List<String>> adyacenciasAux) {
147         if (nodos.size() == 1) {
148             List<List<String>> res = new LinkedList<List<String>>();
149             List<String> lista =new LinkedList<String>();
150             lista.add(nodos.get(0));
151             res.add(lista);
152             return res;
153
154         }
155         List<String> gradoCero = buscarLosDeGrado0(nodos,adyacenciasAux);
156         List<List<String>> res = new LinkedList<List<String>>();
157         for (String nodo:gradoCero) {
158             nodos.remove(nodo);
159             sacarEjes(nodo,adyacenciasAux);
160             List<List<String>> resAux = getEjecucionesAux(nodos, adyacenciasAux);
161             for (List<String> ejecucion : resAux) {
162                 ejecucion.add(0, nodo);
163                 res.add(ejecucion);
164             }
165             nodos.add(nodo);
166             ponerEjes(nodo,adyacenciasAux);
167         }
168         return res;
169     }
170
171     private void ponerEjes(String nodo,
172                         Map<String, List<String>> adyacenciasAux) {
173         for (String nodo2: adyacencias.get(nodo)) {
174             adyacenciasAux.get(nodo2).add(nodo);
175         }
176
177     }
178
179     private void sacarEjes(String nodo,
180                         Map<String, List<String>> adyacenciasAux) {
181
182         for (String nodo2: adyacencias.get(nodo)) {
183             adyacenciasAux.get(nodo2).remove(nodo);
184         }
185
186     }
187
188     private List<String> buscarLosDeGrado0(List<String> nodos,
189                                         Map<String, List<String>> adyacenciasAux) {
190
191         List<String> res = new LinkedList<String>();
192         for (String nodo : nodos) {
193             if (adyacenciasAux.get(nodo).isEmpty()) {
194                 res.add(nodo);
```

```
195             }
196         }
197
198         return res;
199     }
200
201 }
```

Usando esta clase es simple resolver el análisis de serializabilidad:

```
1  public SerializabilityResult analyzeSerializability() {
2      ScheduleGraph graph = buildScheduleGraph();
3      AnalizadorDeGrafos analizador = new AnalizadorDeGrafos(graph);
4
5      if (! analizador.esAciclico()) {
6          System.out.println(analizador.getCiclo());
7          return new SerializabilityResult(false,null,analizador.getCiclo(),null) ;
8      } else {
9          return new SerializabilityResult(true,analizador.getEjecuciones(),null,null);
10     }
11
12 }
```

Parte 4

Ejercicio 4

4.1. Enunciado

Determinar el nivel de recuperabilidad de un plan:

Un plan puede ser no recuperable, recuperable, evitar aborts en cascada o estricto. El nivel de recuperabilidad no depende del modelo de transacciones, sino de analizar si las acciones de cada transacción son lecturas, escrituras o commit y del orden en que estas acciones se dan dentro del plan. Por este motivo, este inciso deberá resolverse en la forma más general posible.

Por definición, se dice que un plan es recuperable si toda transacción T hace COMMIT después de que lo hayan hecho todas las transacciones que escribieron algo que T lee. Por otro lado, se dice que evita aborts en cascada si toda transacción lee de ítems escritos por transacciones que hicieron COMMIT. Por último, es estricto si toda transacción lee y escribe ítems escritos por transacciones que hicieron COMMIT. Como resultado, debe devolverse el nivel de recuperabilidad y, en caso de haber conflictos, un par de transacciones que estén involucradas en el mismo (si hay más de una, devolver cualquiera) además de indicar el motivo del conflicto.

4.2. Resolución

Para resolver este ejercicio, hacemos un chequeo en cascada, realizando los siguientes pasos:

- vemos si la historia es recuperable
- en caso de serlo, vemos si evita aborts en cascada
- en caso de evitarlos, vemos si es estricta

Esto podemos hacerlo garantizando correctitud dado el siguiente teorema:

RC $\subset$ ACA $\subset$ ST

Es decir, si no es recuperable, no puede ser ACA, y si no es ACA no puede ser estricta. Nos evitamos así hacer trabajo de más.

4.2.1. ¿Es recuperable?

Para cada acción A que lee X y luego hace commit, nos fijamos todas las acciones anteriores a A que escribieron X y chequeamos que hayan hecho commit antes que A. Si alguna no lo hizo entonces la historia no es recuperable.

4.2.2. ¿Evita aborts en cascada?

Para cada acción A que lee X, nos fijamos todas las acciones anteriores a A que escribieron X y chequeamos que hayan hecho commit antes que A lea X. Si alguna no lo hizo, entonces la historia no evita aborts en cascada.

4.2.3. ¿Es estricta?

Para cada acción A que lee o escribe X, nos fijamos todas las acciones anteriores a A que escribieron X y chequeamos que hayan hecho commit antes que A lea o escriba X. Si alguna no lo hizo, entonces la historia no es estricta.

4.2.4. Implementación

```
1  public RecoverabilityResult analyzeRecoverability() {
2
3      //en details se guardaran las transacciones (t1 y t2) en conflicto y el mensaje
4      ArrayList<String> details = new ArrayList<String>();
5      details.add(""); //t1
6      details.add(""); //t2
7      details.add(""); //mensaje
8
9      //inicialmente es no recuperable - vemos si es recuperable
10     RecoverabilityType type = RecoverabilityType.NON_RECOVERABLE;
11     Boolean isRecoverable = true;
12     for (Action action : getActions()) {
13         if (!isRecoverable(action, details)) {
14             isRecoverable = false;
15             break;
16         }
17     }
18     //si es recuperable vemos si evita aborts en cascada
19     if (isRecoverable) {
20         type = RecoverabilityType.RECOVERABLE;
21         Boolean avoidsCascadeAborts = true;
22         for (Action action : getActions()) {
23             if (!avoidsCascadeAborts(action, details)) {
24                 avoidsCascadeAborts = false;
25                 break;
26             }
27         }
28         //si evita aborts en cascada vemos si es estricta
29         if (avoidsCascadeAborts) {
30             type = RecoverabilityType.AVOIDS_CASCADING_ABORTS;
31             Boolean isStrict = true;
32             for (Action action : getActions()) {
33                 if (!isStrict(action, details)) {
34                     isStrict = false;
35                     break;
36                 }
37             }
38             if (isStrict) {
39                 type = RecoverabilityType.STRICT;
40             }
41         }
42     }
43     return new RecoverabilityResult(type,details.get(0),details.get(1),details.get(2));
44 }
45
46 private Boolean isRecoverable(Action actualAction, ArrayList<String> details) {
47     if (actualAction.reads()) {
```

```
48          String actualItem = actualAction.getItem();
49
50          for (Action action : getActions()) {
51              String transaction = action.getTransaction();
52              String actualTransaction = actualAction.getTransaction();
53
54              if (canBeARConflictBetween(actualAction, action, actualItem, transaction, actualTransaction))
{
55                  String mensaje = actualTransaction + " hace commit antes que " + transaction
56                          + " que escribe " + actualItem + ".";
57                  fillDetails(details, transaction, actualTransaction, mensaje);
58                  return false;
59              }
60          }
61      }
62      return true;
63  }
64
65  private Boolean avoidsCascadeAborts(Action actualAction, ArrayList<String> details) {
66      if (actualAction.reads()) {
67          String actualItem = actualAction.getItem();
68
69          for (Action action : getActions()) {
70              String transaction = action.getTransaction();
71              String actualTransaction = actualAction.getTransaction();
72
73              if (canBeAConflictBetween(actualAction, action, actualItem, actualTransaction, transaction)) {
74                  String mensaje = actualTransaction + " lee " + actualItem + " de " +
75                          transaction + " que aun no hizo commit.";
76                  fillDetails(details, transaction, actualTransaction, mensaje);
77                  return false;
78              }
79          }
80      }
81      return true;
82  }
83
84
85  private Boolean isStrict(Action actualAction, ArrayList<String> details) {
86      String actualItem = actualAction.getItem();
87
88      for (Action action : getActions()) {
89          String transaction = action.getTransaction();
90          String actualTransaction = actualAction.getTransaction();
91
92          if (canBeAConflictBetween(actualAction, action, actualItem, actualTransaction, transaction)) {
93              String mensaje = actualTransaction + (action.reads() ? " lee " : " escribe ") +
94                      actualItem + " que " + transaction + " escribio y aun no commiteo.";
95              fillDetails(details, transaction, actualTransaction, mensaje);
96              return false;
97          }
98      }
99      return true;
100 }
101
102 private boolean canBeARConflictBetween(Action actualAction, Action action, String actualItem, String
transaction, String ↴
103                                        actualTransaction) {
104     ArrayList<String> transactions = transactionsOrderByCommit();
105     String item = action.getItem();
106
107     return
108         //una escribe
```

```
109         action.writes() &&
110         //y lo hace antes de que la otra
111         getActions().indexOf(action) < getActions().indexOf(actualAction) &&
112         //actuan sobre el mismo item
113         item == actualItem &&
114         //HACK: si la transaccion no es legal, puede haber un unlock desgogado, chequeamos entonces
115         // que la actual haga algo sobre el item
116         (actualAction.reads() || actualAction.writes()) &&
117         //la segunda hizo commit
118         transactions.contains(actualAction.getTransaction()) &&
119         //y la primera no hizo commit
120         (!transactions.contains(action.getTransaction()) ||
121          //o lo hizo despues que la segunda
122          transactions.indexOf(action.getTransaction()) >
123          transactions.indexOf(actualAction.getTransaction()));
124 }
125
126
127 private boolean canBeAConflictBetween(Action actualAction, Action action, String actualItem, String
actualTransaction, ↴
128                                       String transaction) {
129     Action commit = getCommitAction(transaction);
130     String item = action.getItem();
131
132     return
133         //una escribe
134         action.writes() &&
135         //y lo hace antes de que la otra
136         getActions().indexOf(action) < getActions().indexOf(actualAction) &&
137         //HACK: si la transaccion no es legal, puede haber un unlock desgogado, chequeamos entonces
138         // que la actual haga algo sobre el item
139         (actualAction.reads() || actualAction.writes()) &&
140         //actuan sobre el mismo item
141         item == actualItem &&
142         //pertenecen a transacciones diferentes
143         transaction != actualTransaction &&
144         // y si la ultima hace commit lo hace luego de que la primera haga commit
145         (commit == null || getActions().indexOf(commit) > getActions().indexOf(actualAction));
146 }
147
148 //llena details con las transacciones en conflicto y el mensaje
149
150 private void fillDetails(ArrayList<String> details, String transaction, String actualTransaction, String
mensaje) {
151     details.set(0,transaction);
152     details.set(1,actualTransaction);
153     details.set(2, mensaje);
154 }
155
156 //devuelve la lista de transacciones que hacen commit ordenadas temporalmente
157
158 private ArrayList<String> transactionsOrderByCommit() {
159     ArrayList<String> transactions = new ArrayList<String>();
160     for (Action action : getActions()) {
161         if (action.commits()) {
162             transactions.add(action.getTransaction());
163         }
164     }
165     return transactions;
166 }
167
168 //obtiene la accion que hace commit de la transaccion, si no existe devuelve null
169
```

```
170 private Action getCommitAction(String transaction) {
171     Action commit = null;
172     for (Action action : getActions()) {
173         if (action.commits() && action.getTransaction() == transaction) {
174             commit = action;
175         }
176     }
177     return commit;
178 }
```

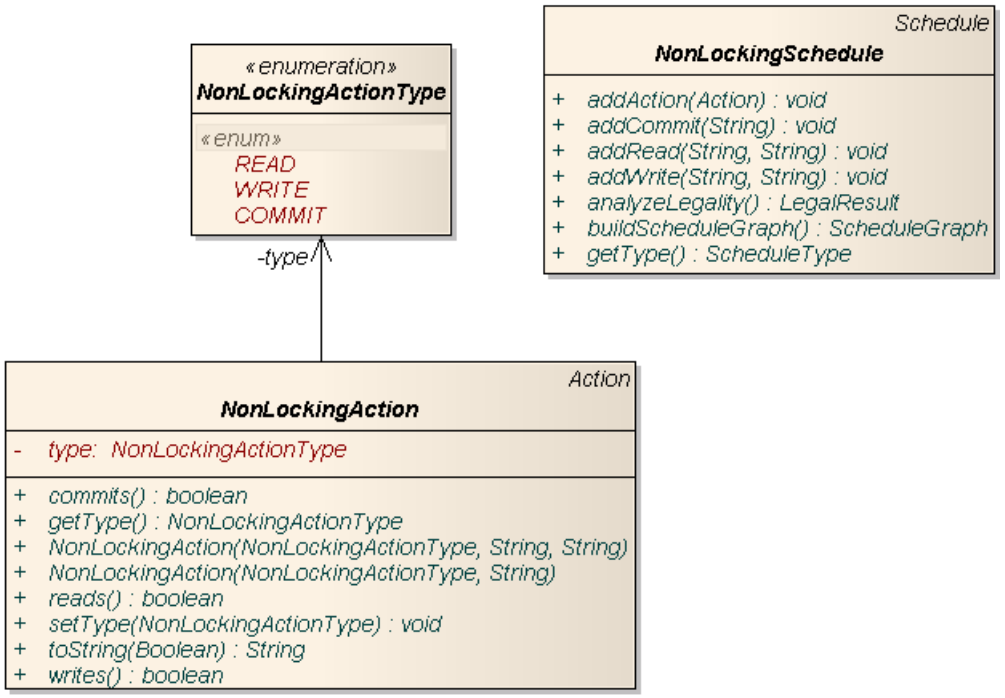
Parte 5

Anexo: Diagramas de clases

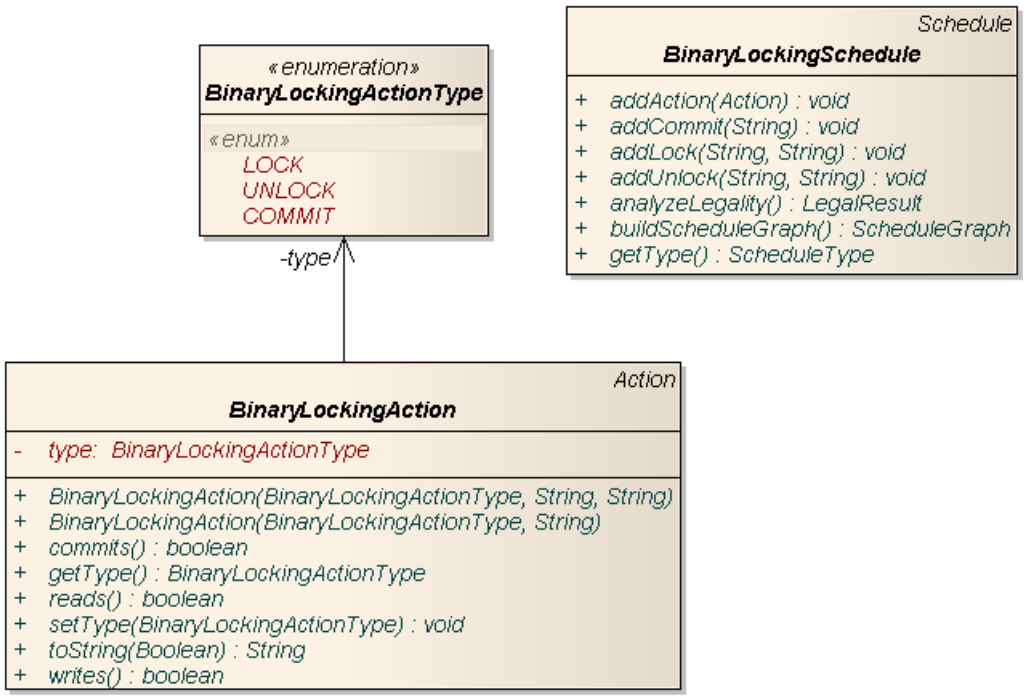
5.1. Common


Schedule
actions: List<Action>
items: List<String>
- transactions: List<String>
+ addAction(Action) : void
+ addItem(String) : void
+ addTransaction(String) : void
+ analyzeLegality() : LegalResult
+ analyzeRecoverability() : RecoverabilityResult
+ analyzeSeriality() : SerialResult
+ analyzeSerializability() : SerializabilityResult
- avoidsCascadeAborts(Action, ArrayList<String>) : Boolean
+ buildScheduleGraph() : ScheduleGraph
- canBeARConflictBetween(Action, Action, String, String) : boolean
- canBeARConflictBetween(Action, Action, String, String, String) : boolean
+ editAction(int, Action) : void
- fillDetails(ArrayList<String>, String, String, String) : void
+ getActions() : List<Action>
- getCommitAction(String) : Action
+ getItems() : List<String>
+ getTransactions() : List<String>
+ getType() : ScheduleType
- isRecoverable(Action, ArrayList<String>) : Boolean
- isStrict(Action, ArrayList<String>) : Boolean
+ removeAction(int) : void
+ Schedule()
+ swapAction(int, int) : void
- transactionsOrderByCommit() : ArrayList<String>
«enumeration»
ScheduleType
«enum»
NON_LOCKING
BINARY_LOCKING
TERNARY_LOCKING
Action
item: String
transaction: String
+ Action(String, String)
+ Action(String)
+ commits() : boolean
+ getItem() : String
+ getTransaction() : String
+ reads() : boolean
+ setItem(String) : void
+ setTransaction(String) : void
+ toString(Boolean) : String
+ writes() : boolean
ScheduleArc
- endIndex: int
- endTransaction: String
- startIndex: int
- startTransaction: String
+ getEndIndex() : int
+ getEndTransaction() : String
+ getStartIndex() : int
+ getStartTransaction() : String
+ ScheduleArc(String, String, int, int)
ScheduleGraph
- arcs: List<ScheduleArc>
- transactions: List<String>
+ addArc(ScheduleArc) : void
+ addTransaction(String) : void
+ getArcs() : List<ScheduleArc>
+ getTransactions() : List<String>
+ imprimir(String) : void
+ ScheduleGraph()
~grafo
T
S
Schedule::
:Par
+ fst: T
+ snd: S
+ Par(T, S)
Schedule::AnalizadorDeGrafos
~ adyacencias: Map<String,List<String>>
~ adyacenciasReversas: Map<String,List<String>>
~ grafo: ScheduleGraph
+ AnalizadorDeGrafos(ScheduleGraph)
- armarCiclo(String, Set<String>, Set<String>) : List<String>
- buscarLosDeGrado0(List<String>, Map<String, List<String>>) : List<String>
- esAciclico() : Boolean
+ getCiclo() : List<String>
+ getEjecuciones() : List<List<String>>
- getEjecucionesAux(List<String>, Map<String, List<String>>) : List<List<String>>
- hayCiclo(String, Set<String>, Set<String>) : boolean
- ponerEjes(String, Map<String, List<String>>) : void
- sacarEjes(String, Map<String, List<String>>) : void

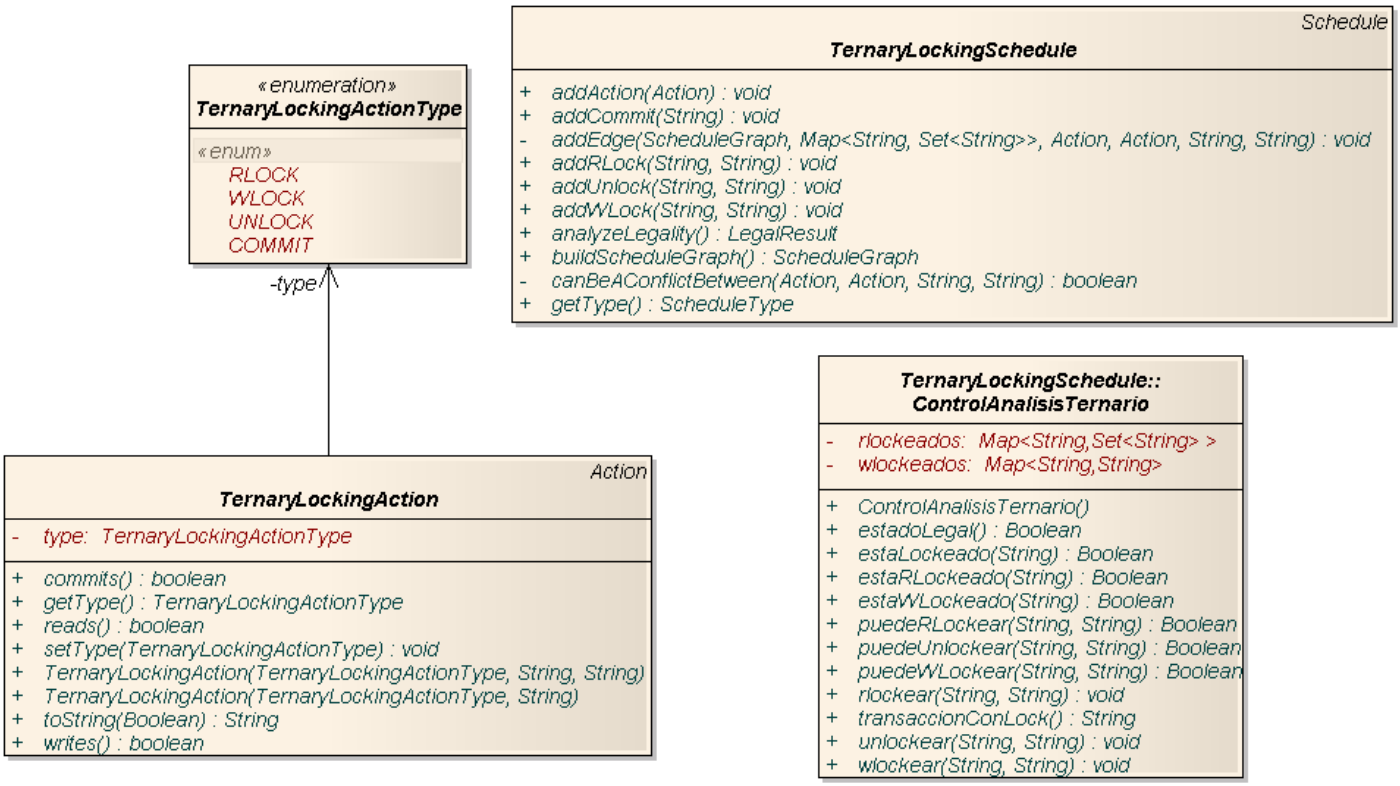
5.2. nonLocking



5.3. binaryLocking



5.4. ternaryLocking

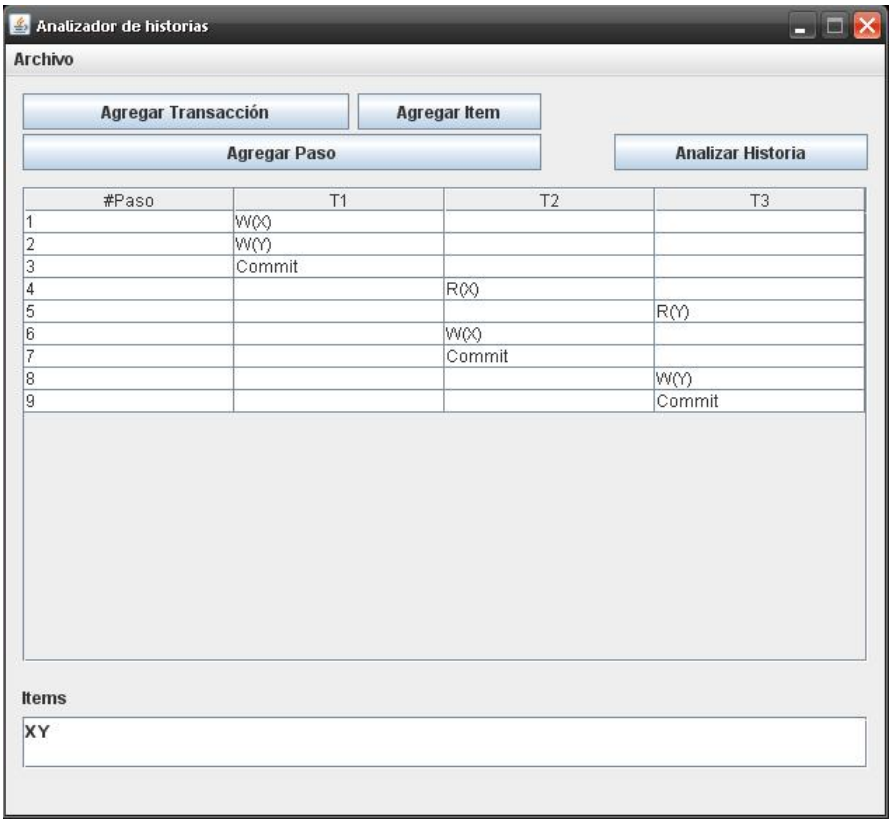


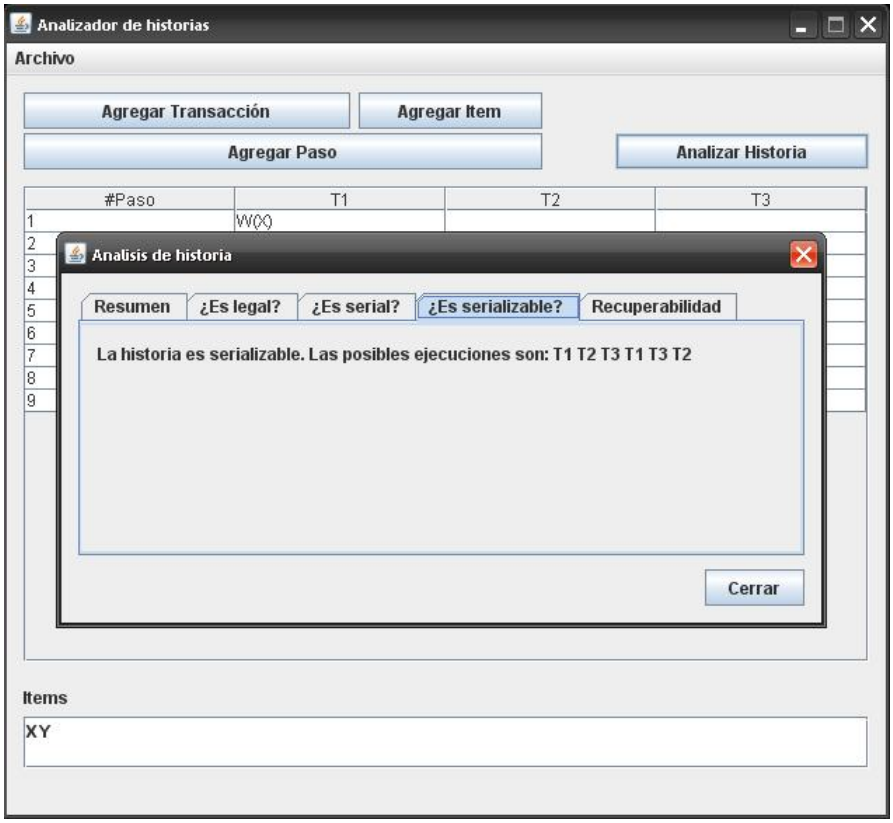
Parte 6

Anexo: Testing

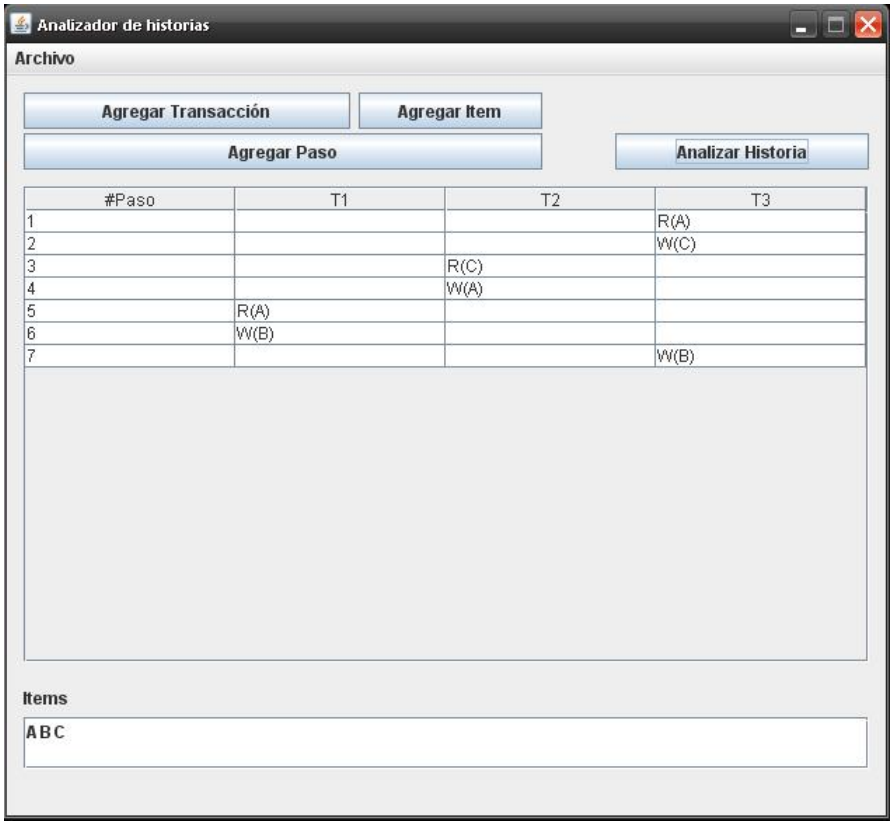
6.1. Sin locking

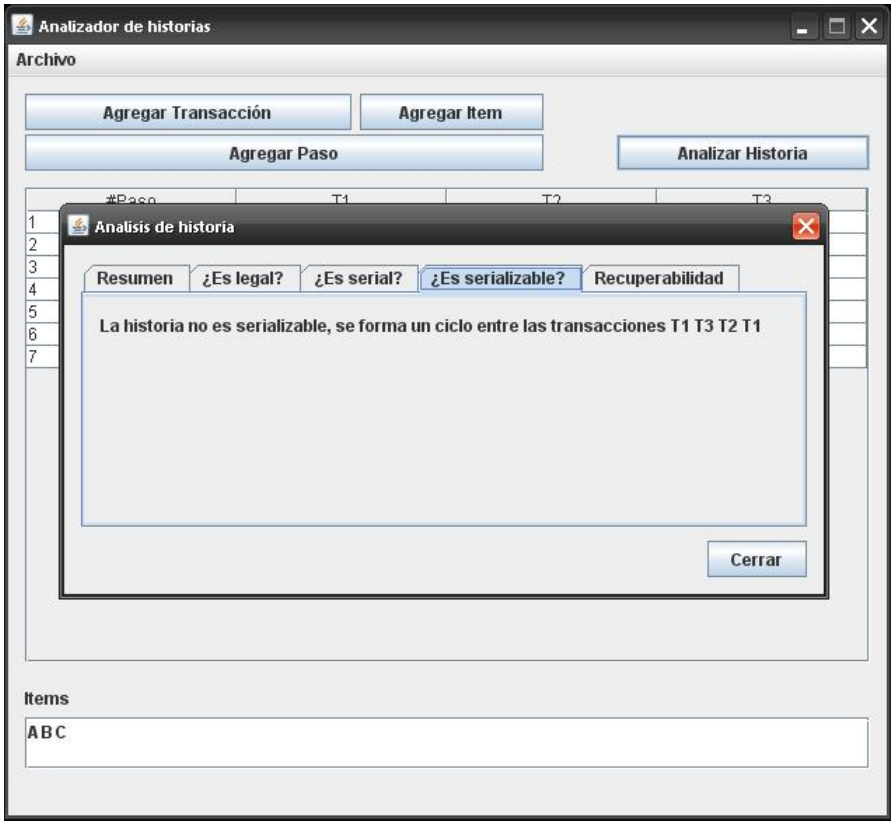
- Caso: historia serializable



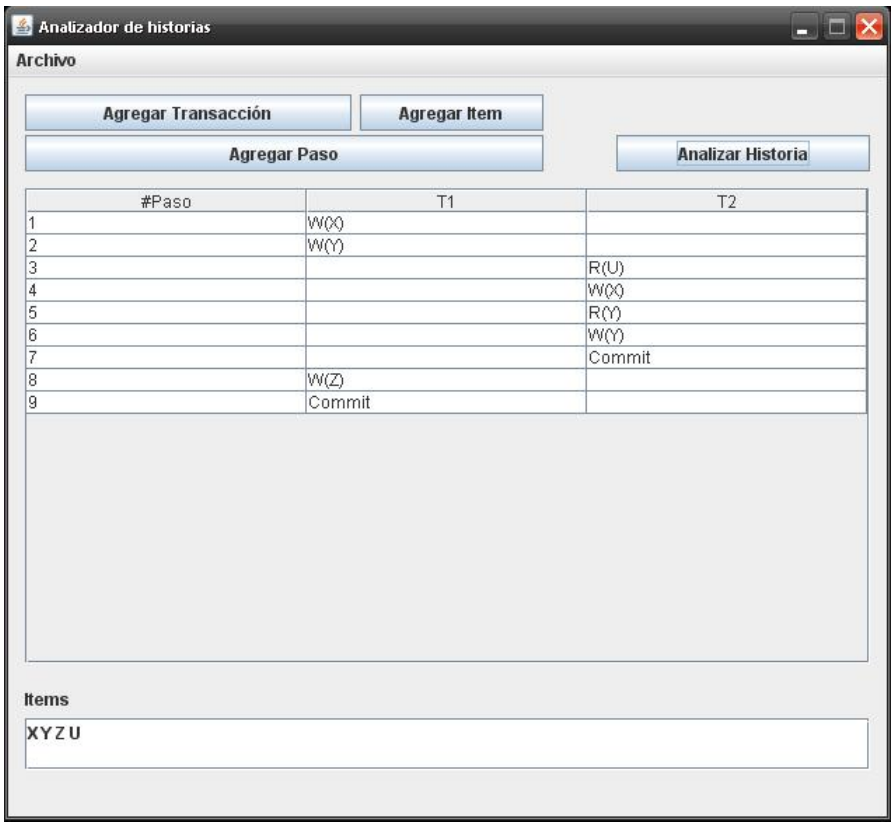


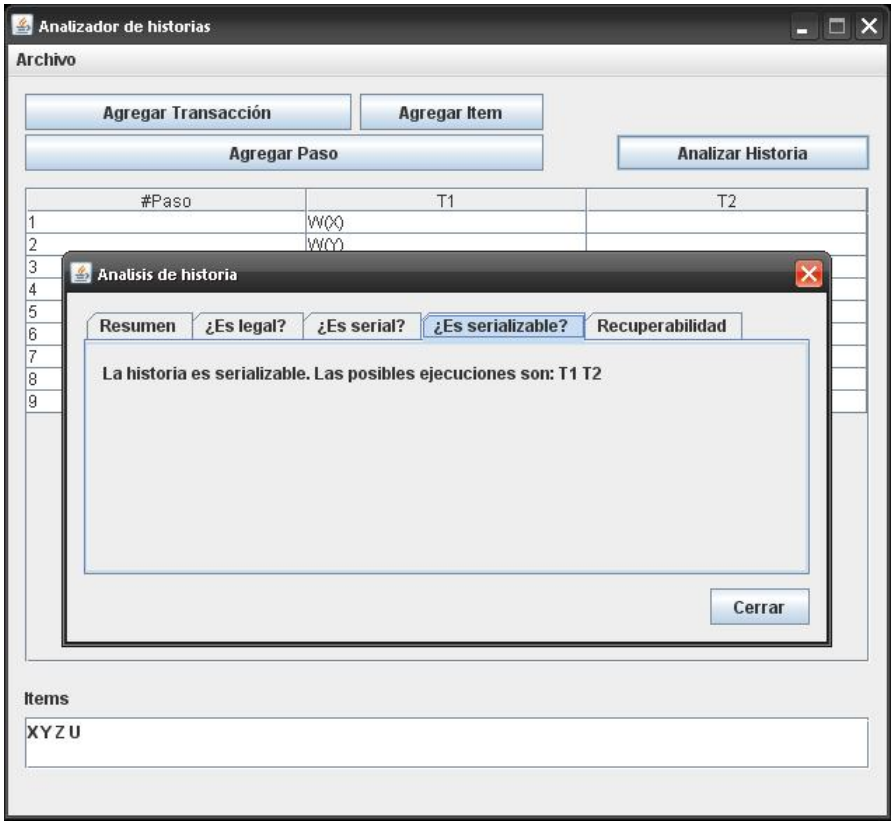
- Caso: historia no serializable

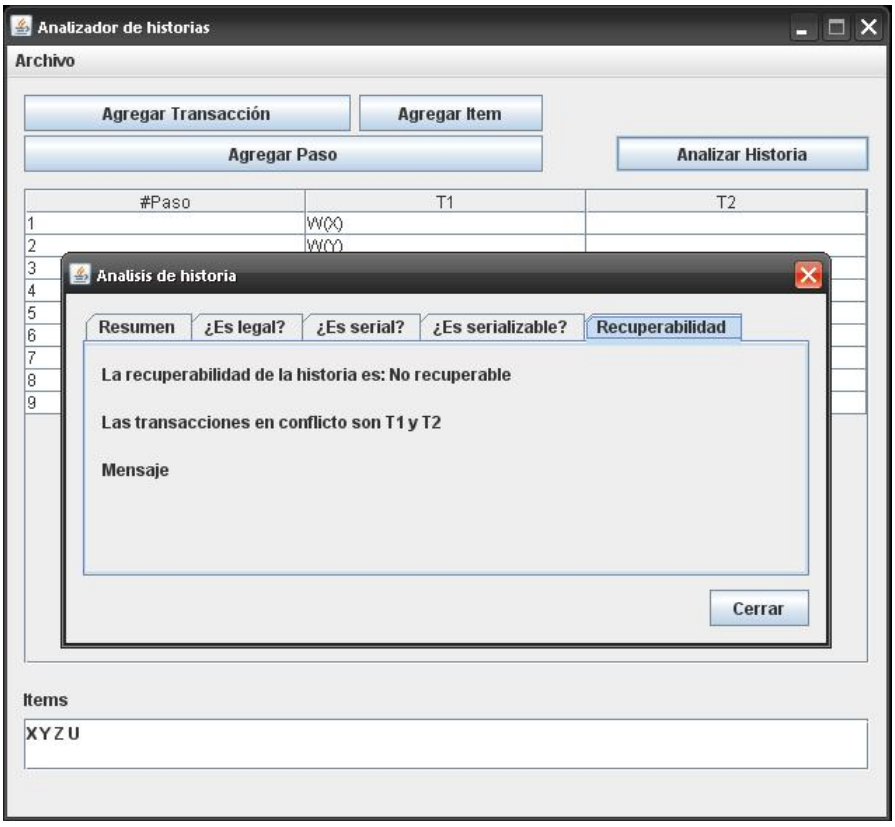




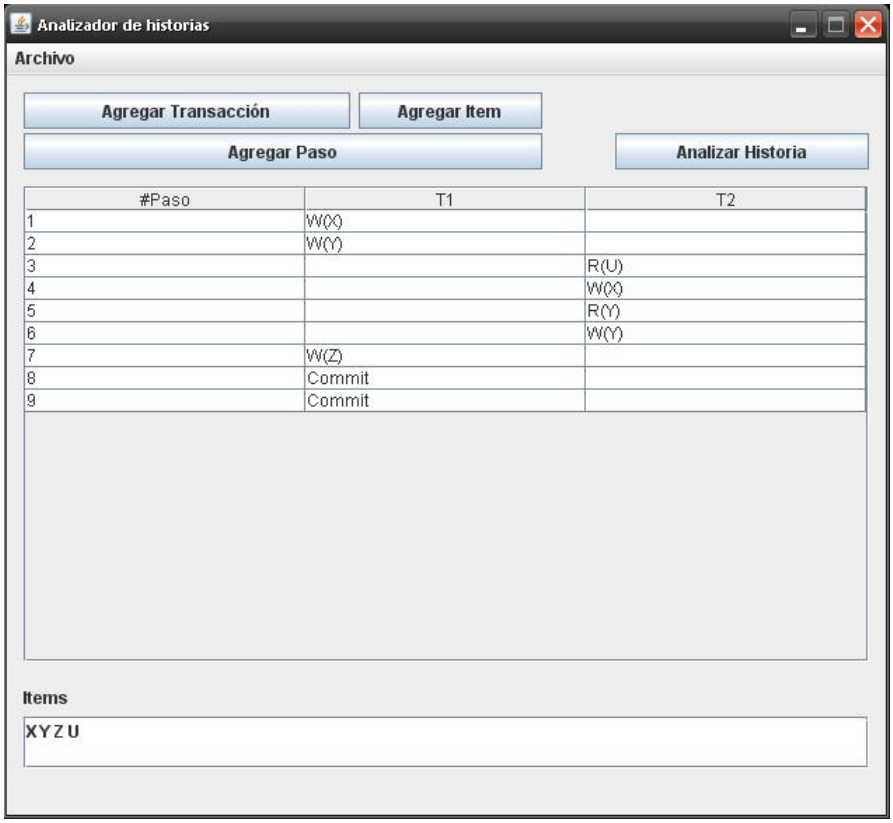
- Caso: historia serializable no recuperable

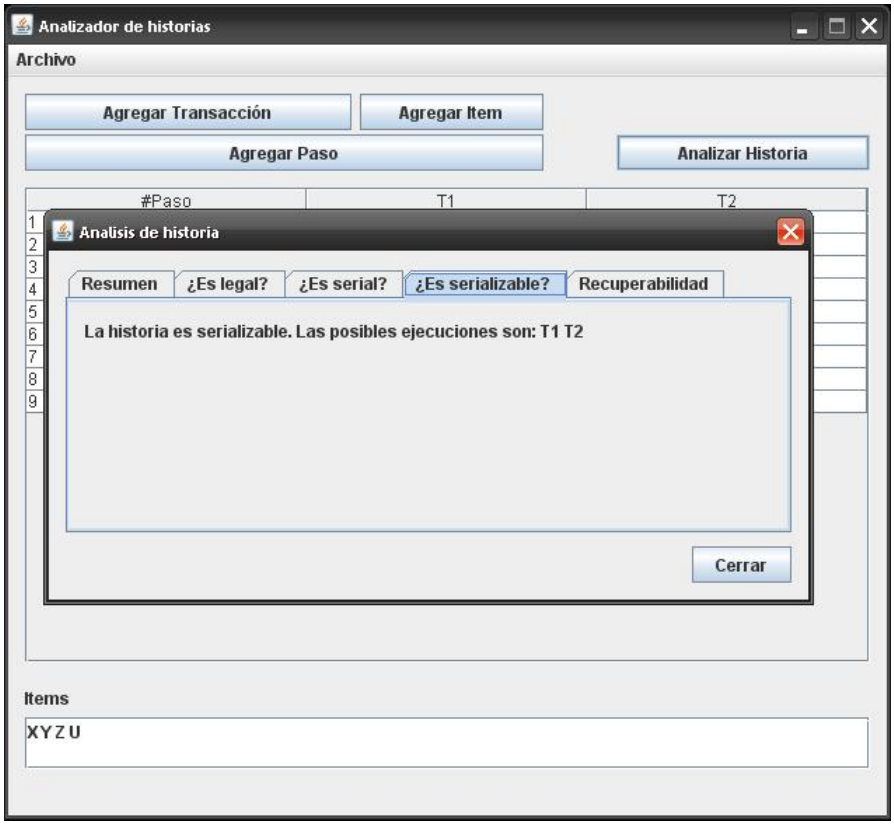


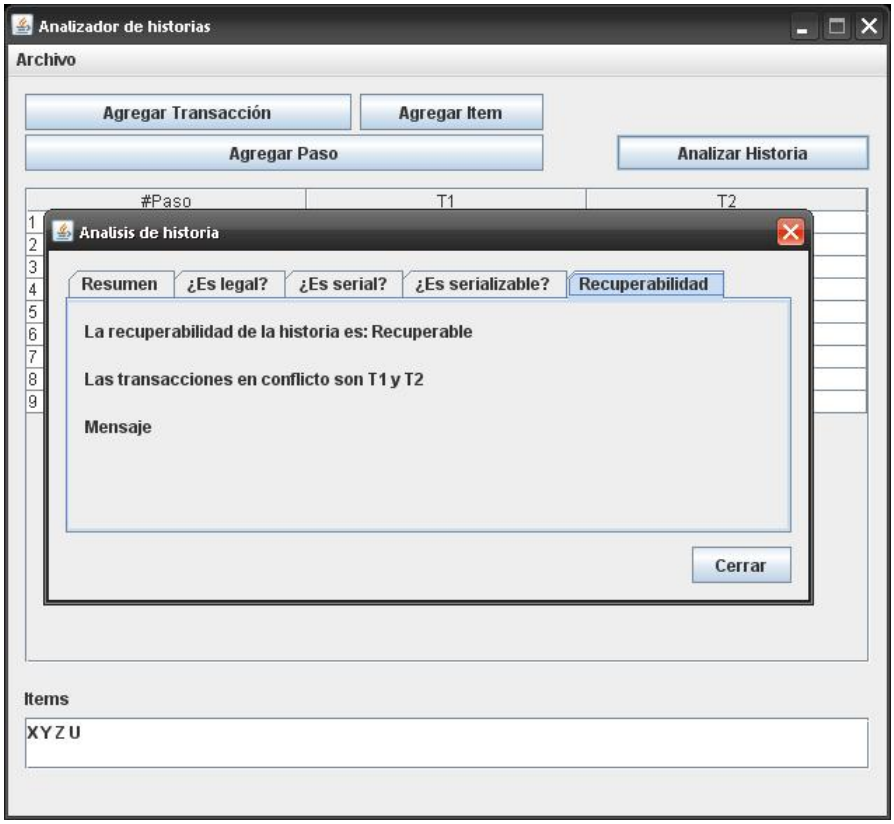




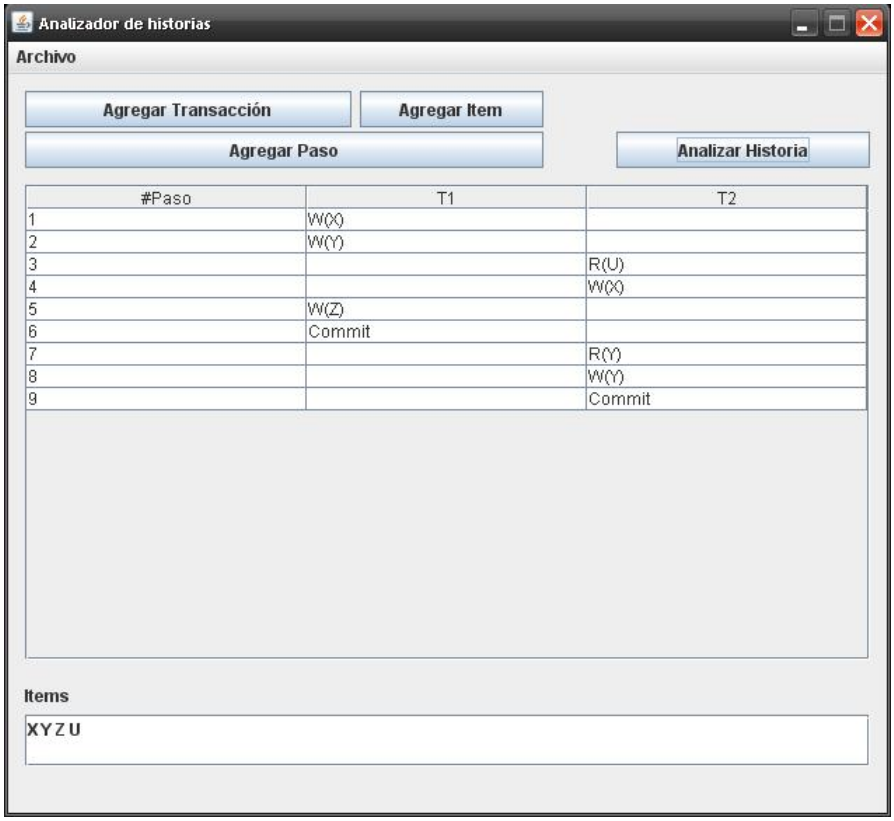
- Caso: historia serializable recuperable

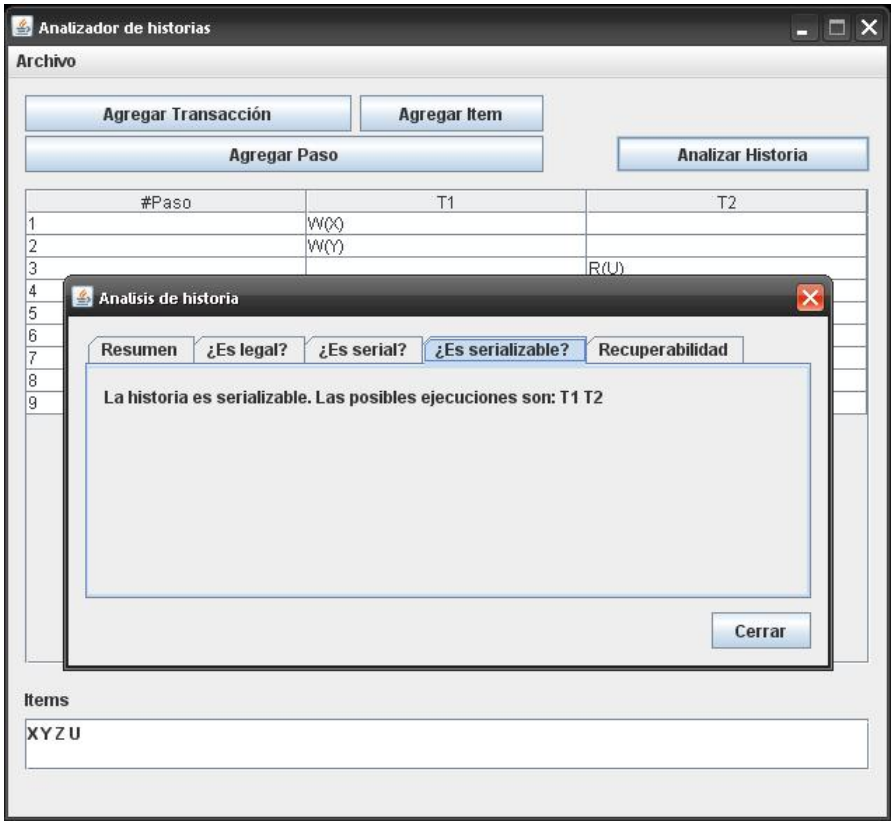

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
1
W(X)
2
W(Y)
3
R(U)
4
W(X)
5
R(Y)
6
W(Y)
7
W(Z)
8
Commit
9
Commit
Items
XYZU

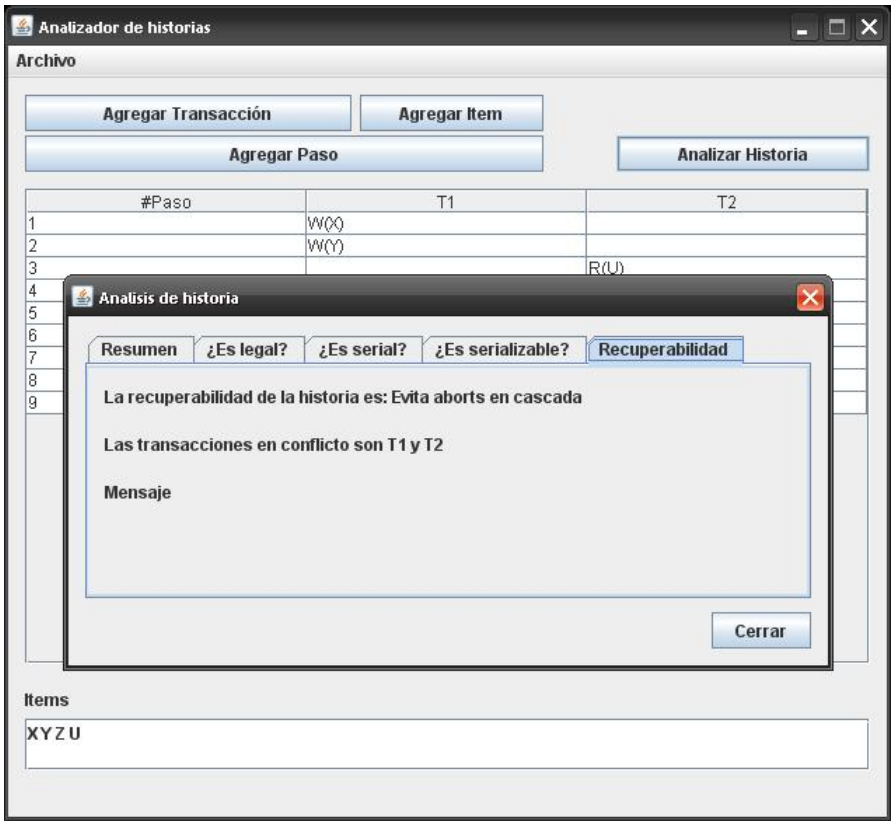

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es serializable. Las posibles ejecuciones son: T1 T2
Cerrar
Items
XYZU



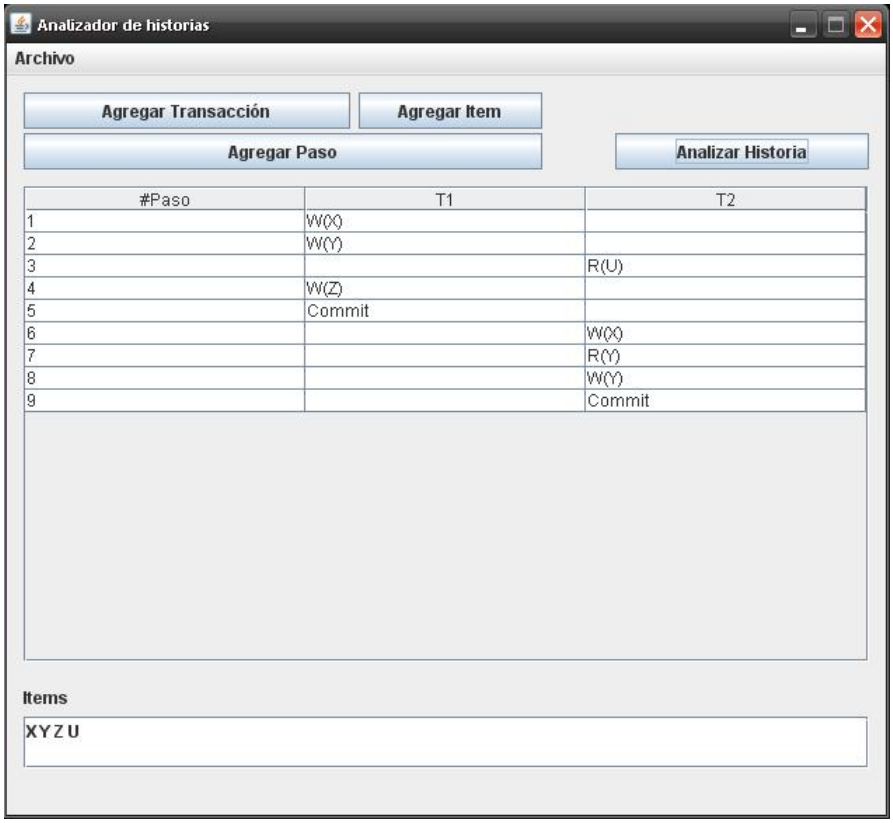
- Caso: historia serializable evita aborts en cascada

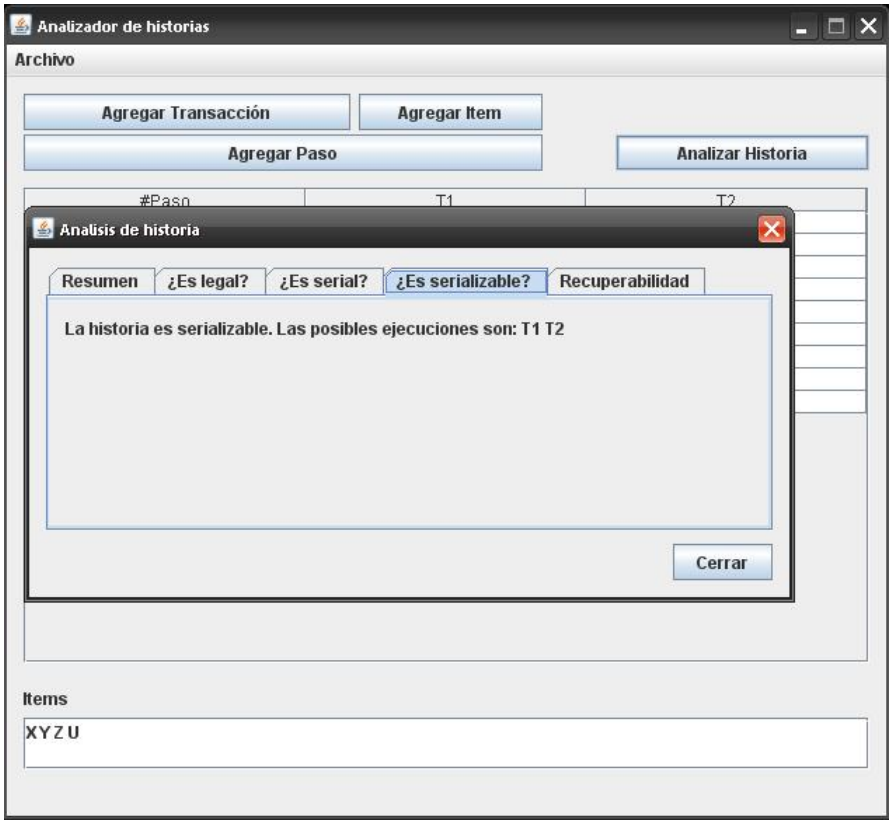


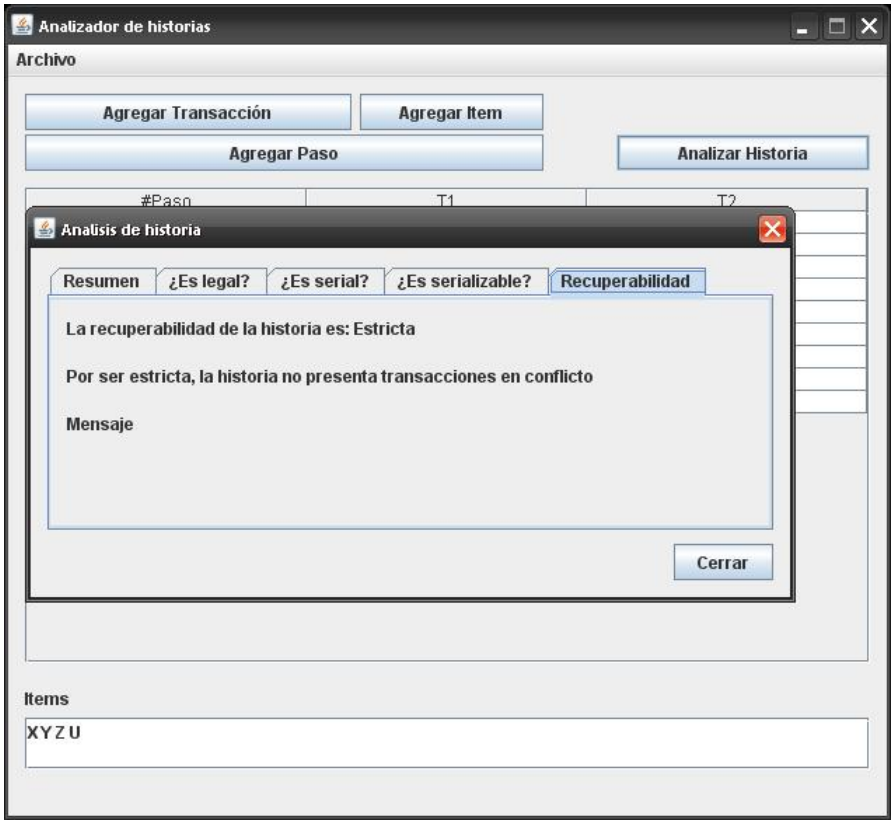




- Caso: historia serializable estricta

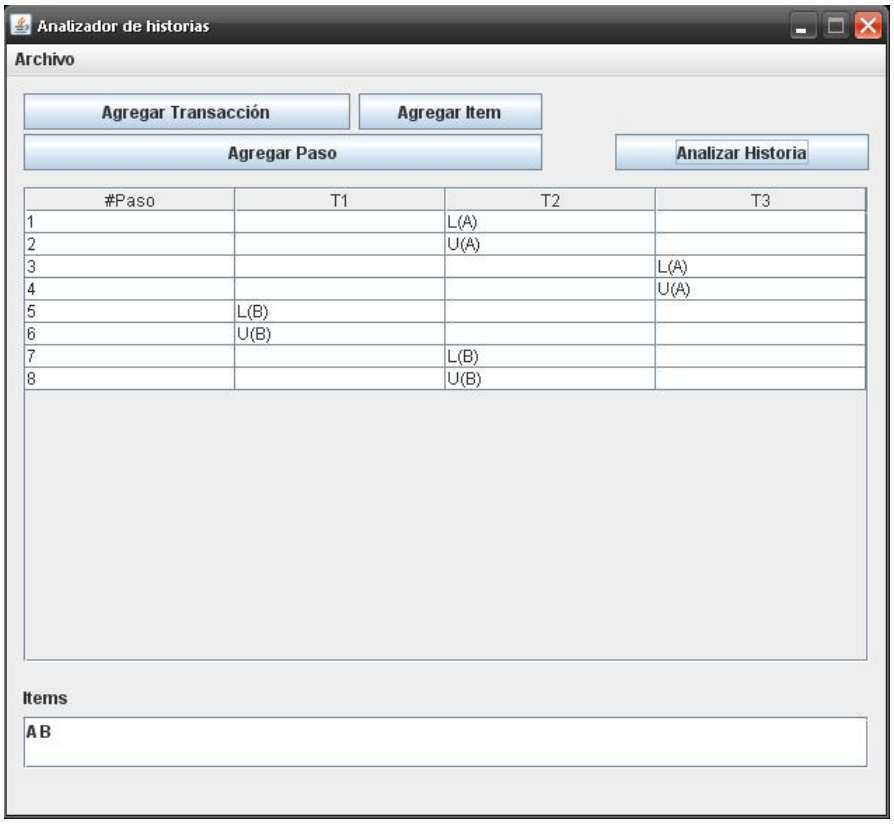

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
1
W(X)
2
W(Y)
3
R(U)
4
W(Z)
5
Commit
6
W(X)
7
R(Y)
8
W(Y)
9
Commit
Items
X Y Z U

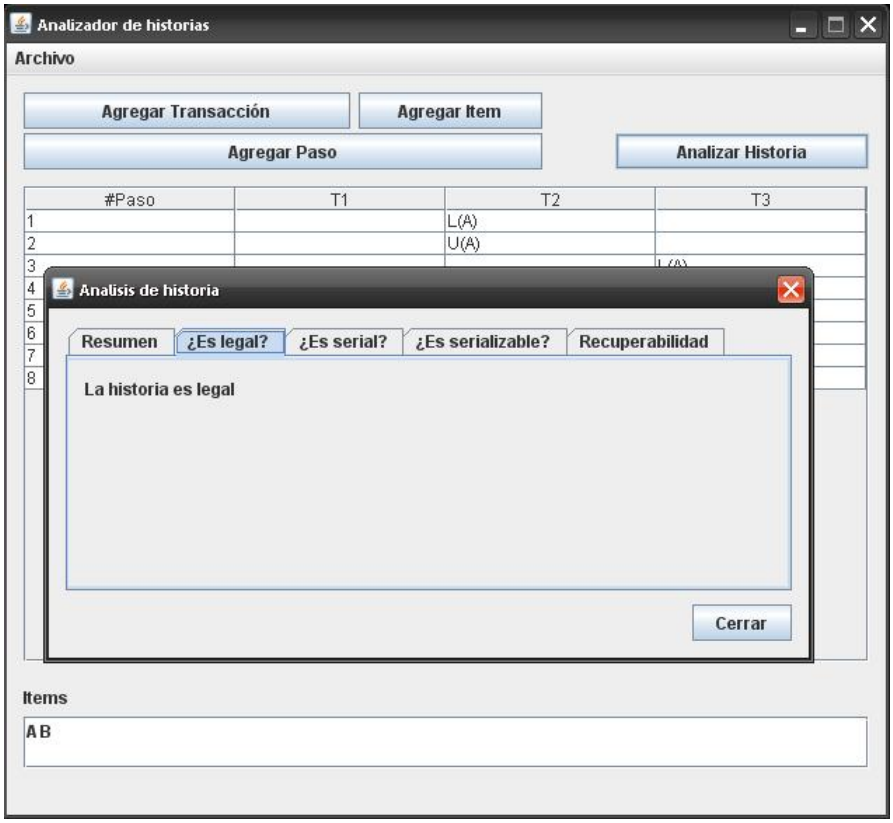

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es serializable. Las posibles ejecuciones son: T1 T2
Cerrar
Items
X Y Z U

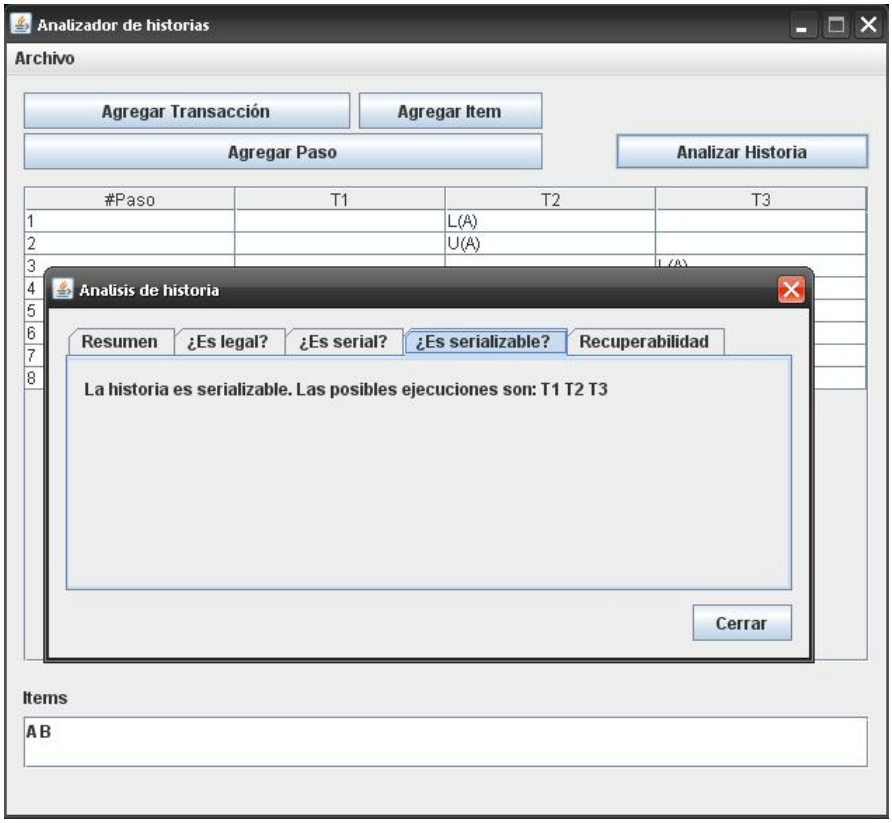


6.2. Locking Binario

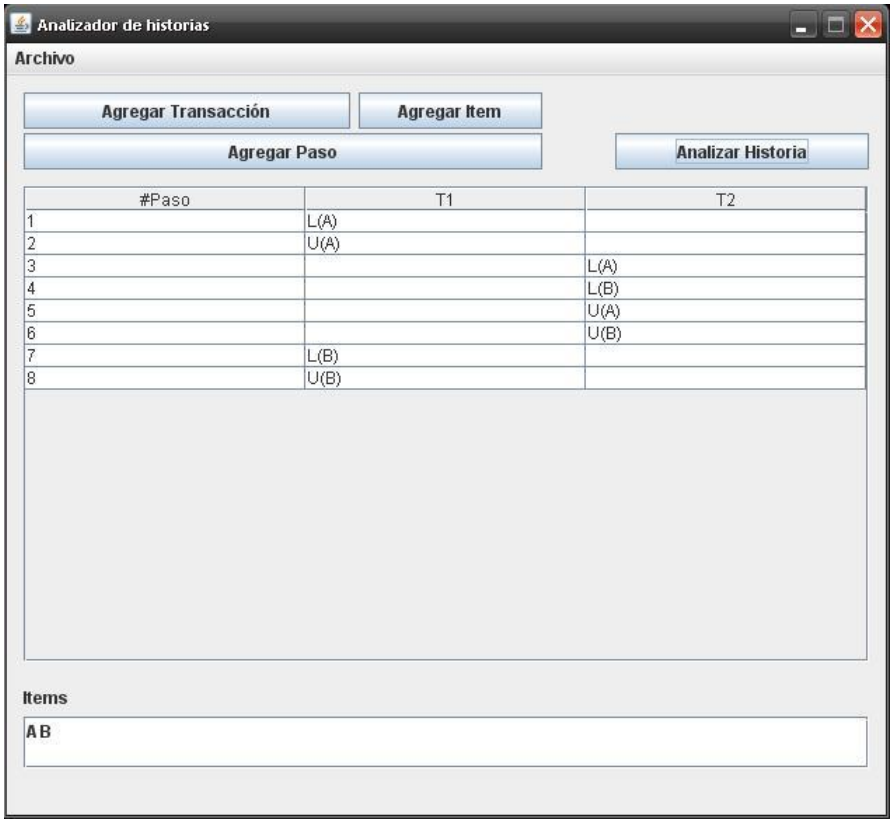
- Caso: historia legal serializable

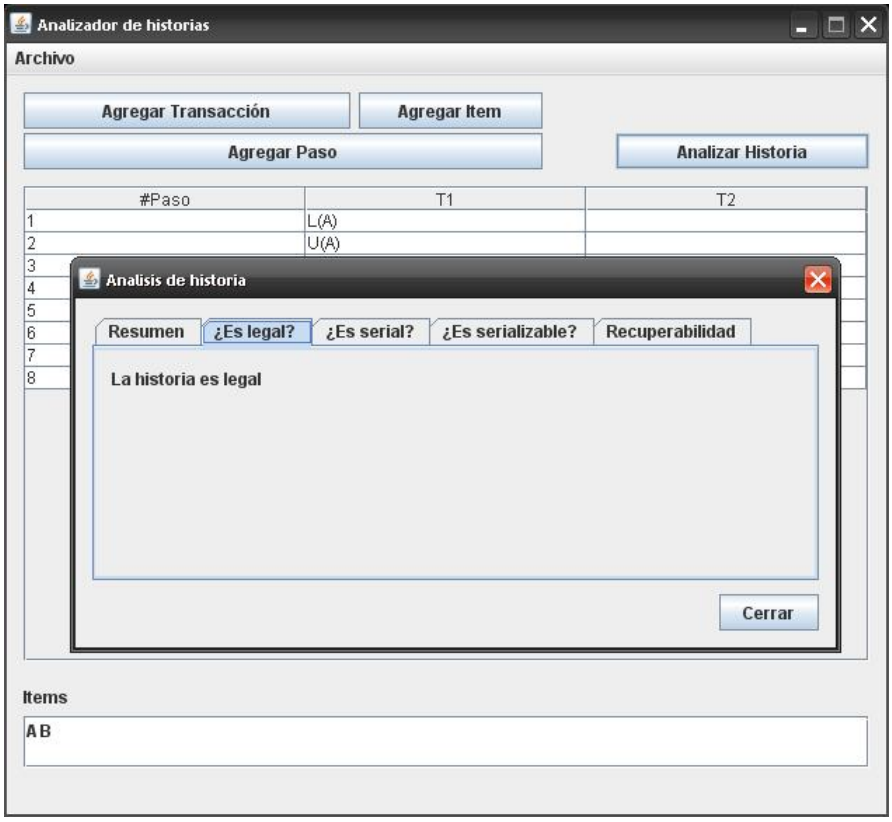


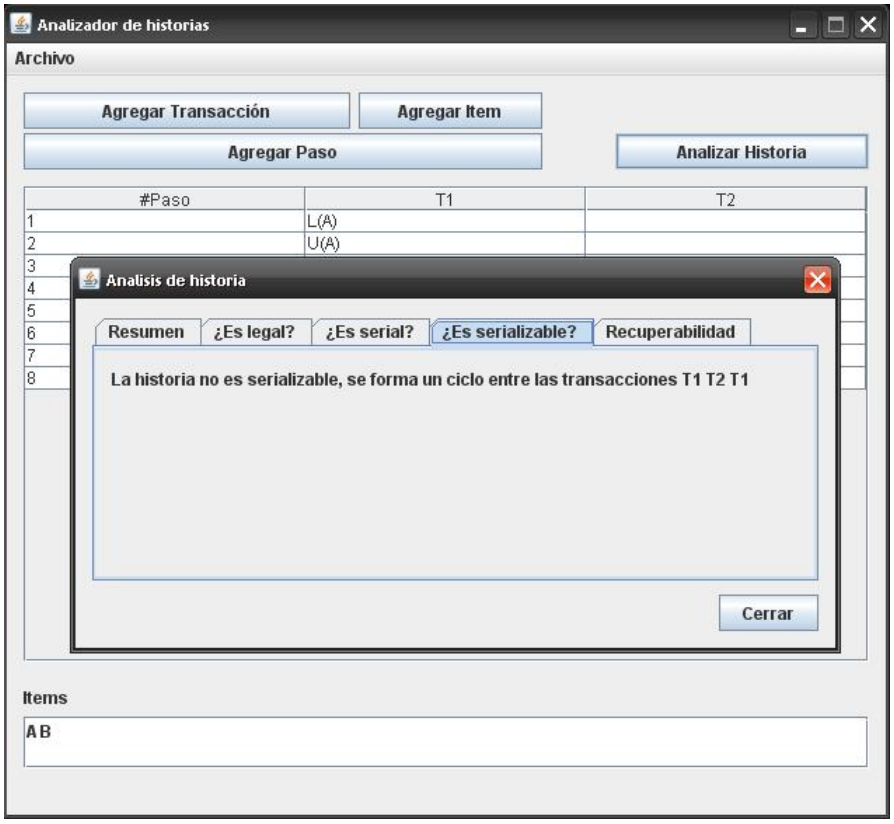




- Caso: historia legal no serializable

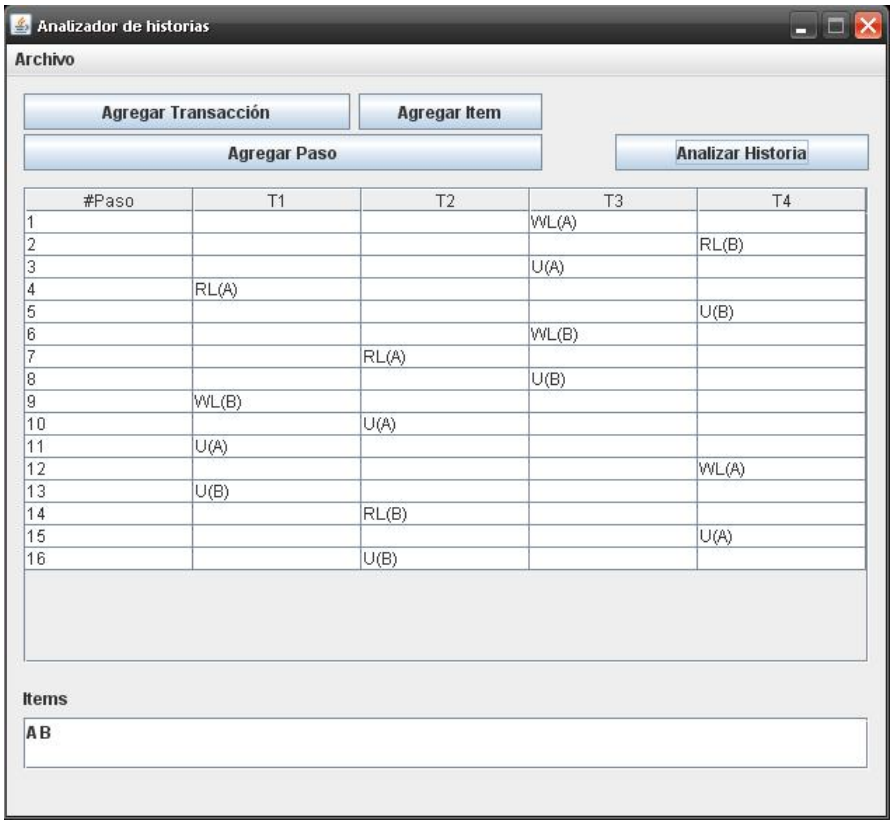

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
1
L(A)
2
U(A)
3
L(A)
4
L(B)
5
U(A)
6
U(B)
7
L(B)
8
U(B)
Items
A B

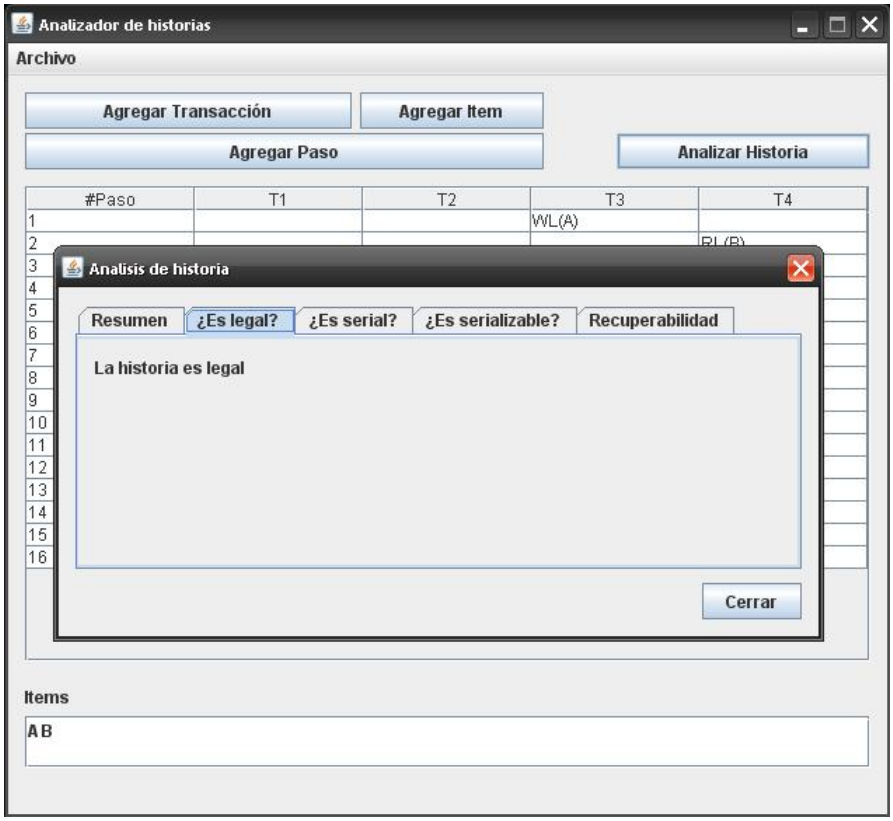

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
1
L(A)
2
U(A)
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es legal
Cerrar
Items
A B

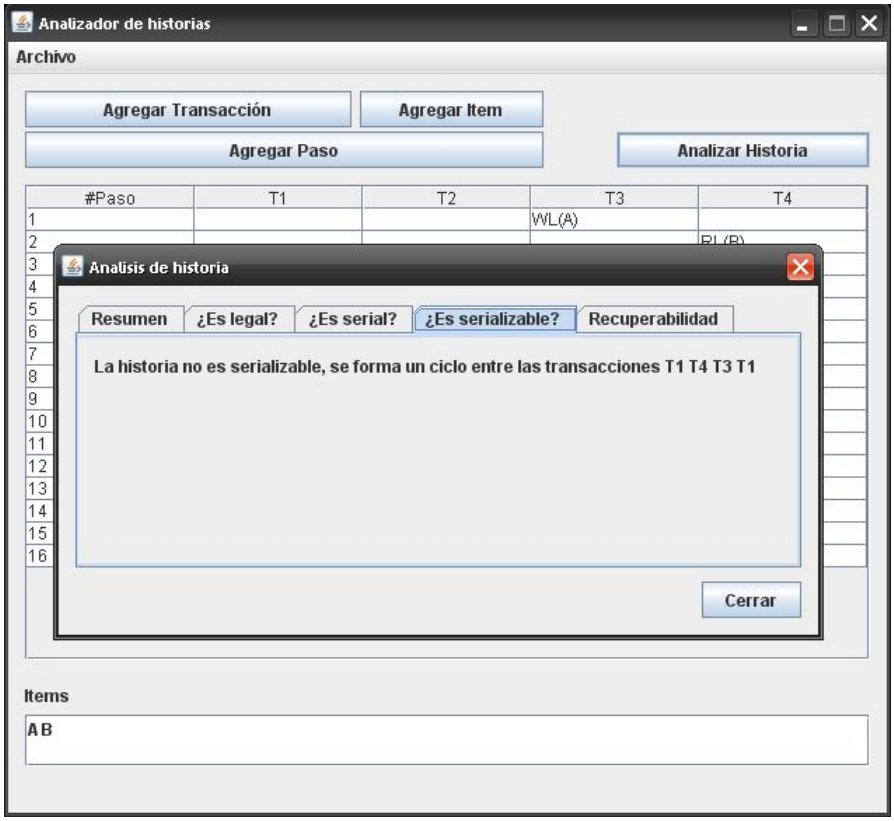


6.3. Locking Ternario

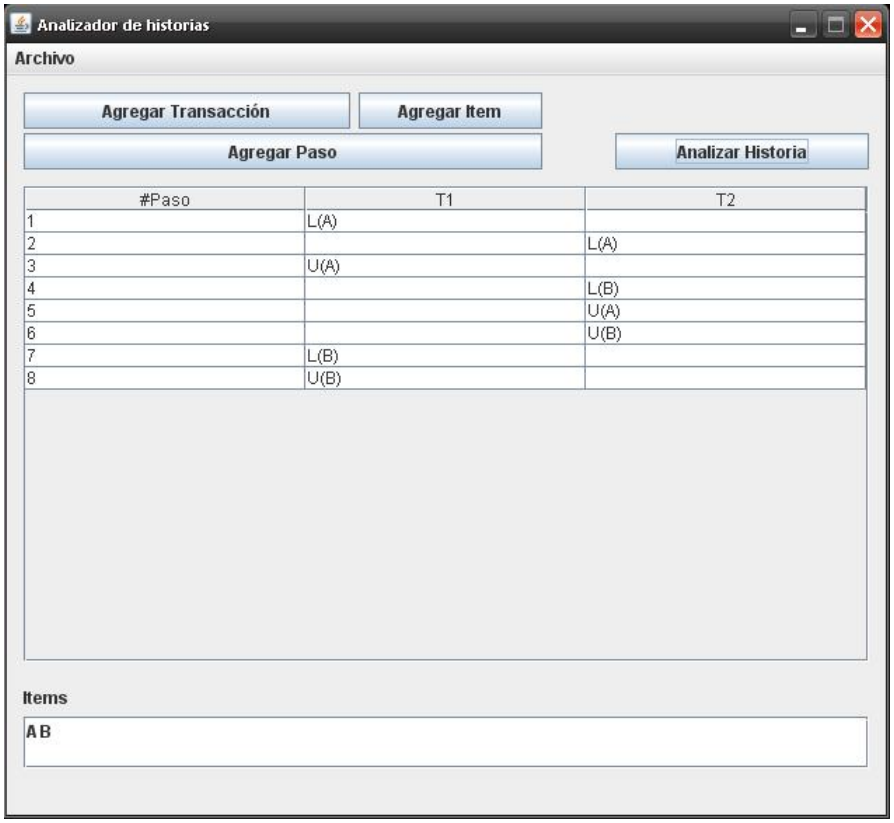
- Caso: historia legal no serializable

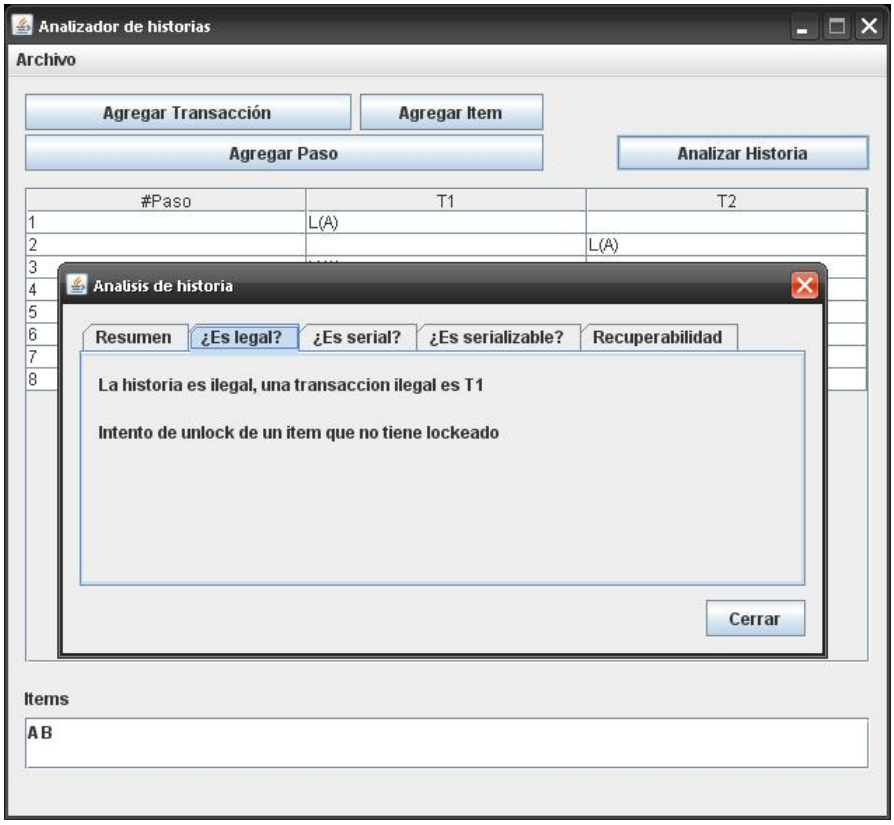


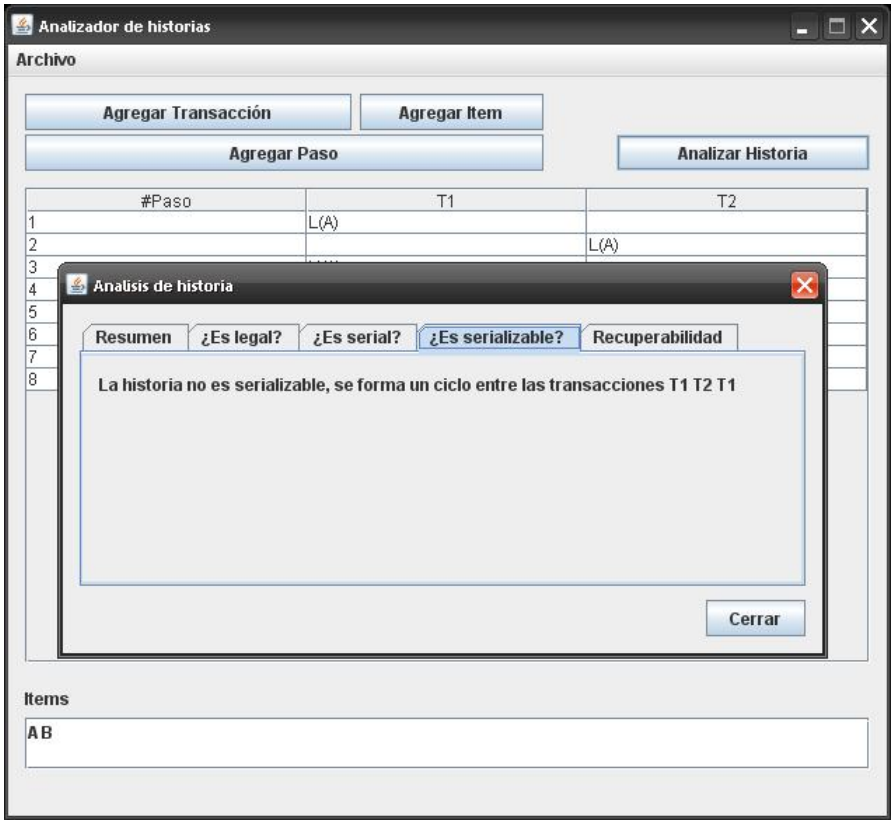




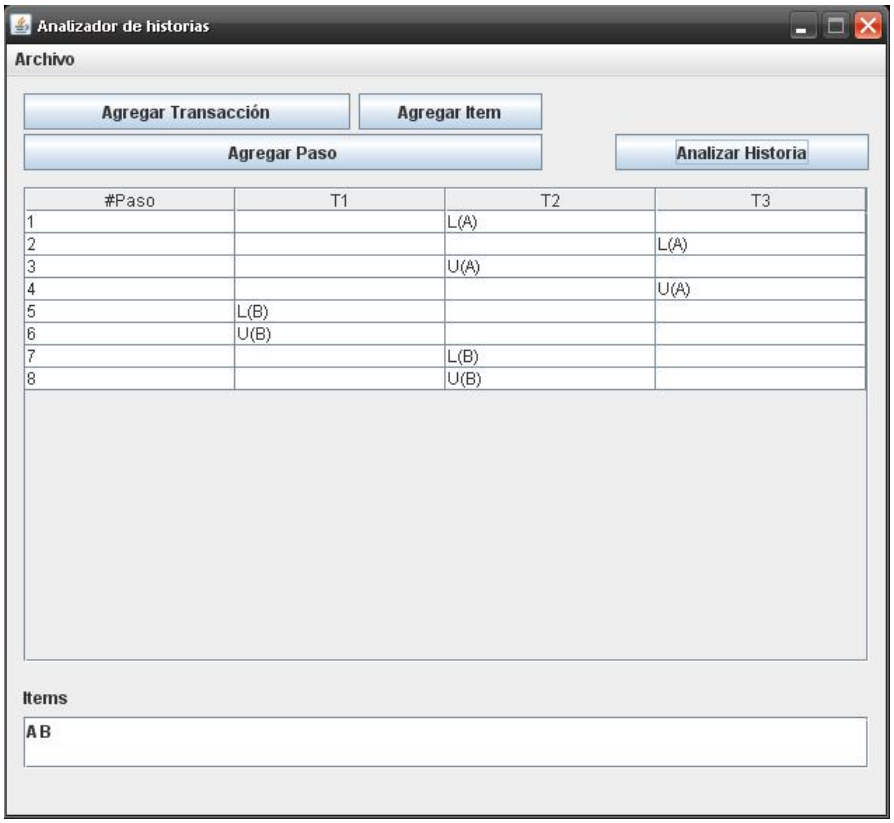
- Caso: historia no legal no serializable

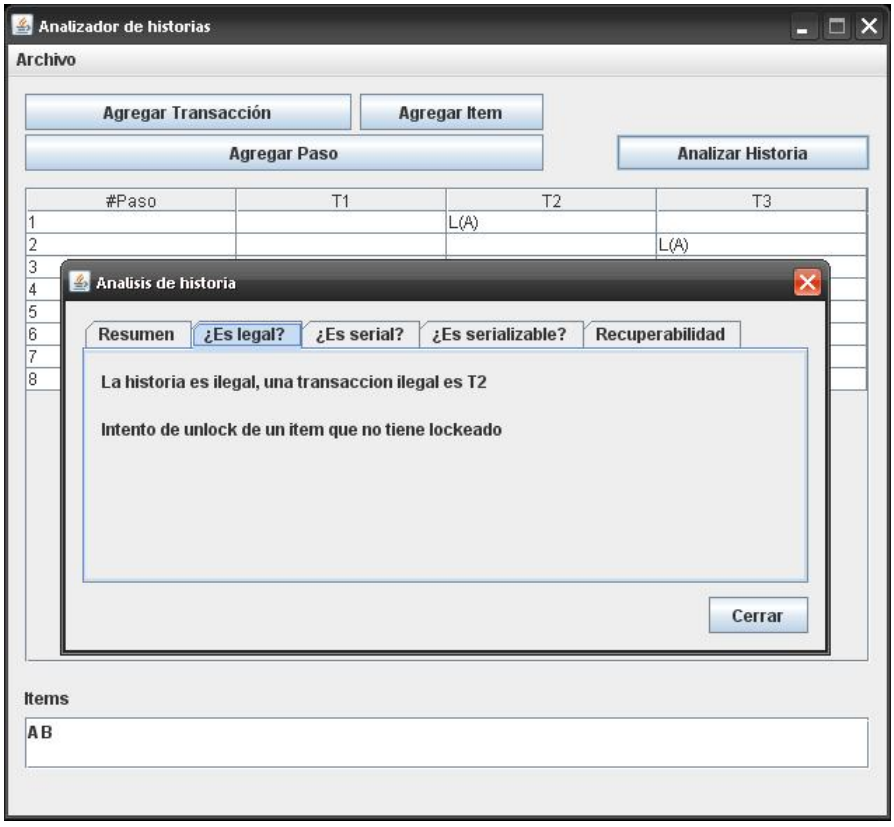

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
1
L(A)
2
L(A)
3
U(A)
4
L(B)
5
U(A)
6
U(B)
7
L(B)
8
U(B)
Items
A B

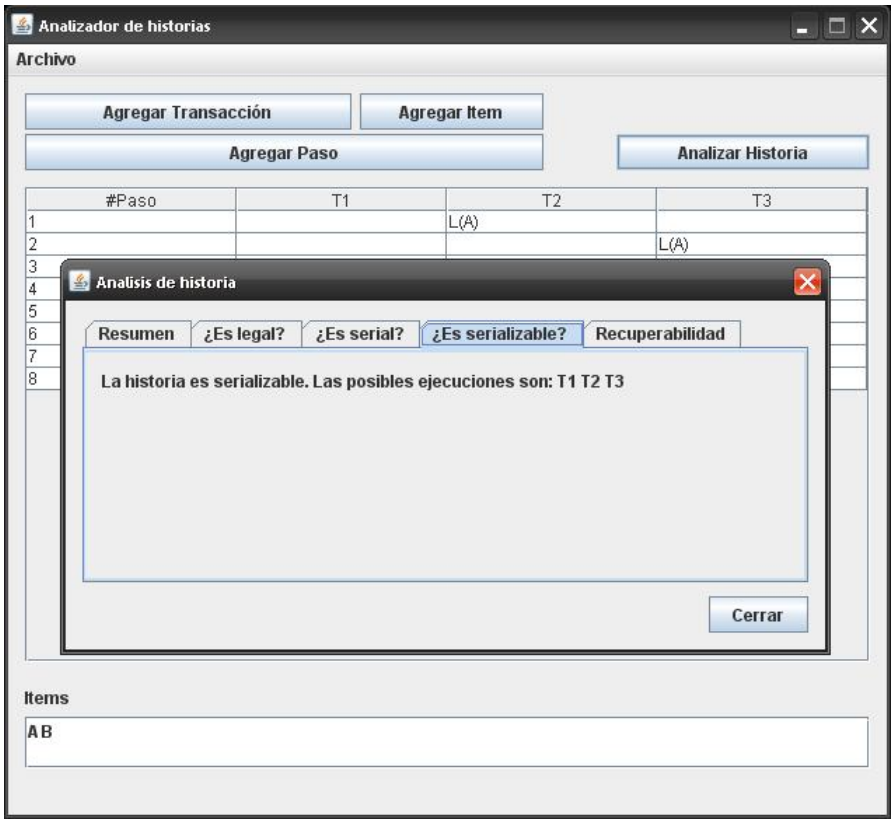

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
T2
1
L(A)
2
L(A)
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es ilegal, una transaccion ilegal es T1
Intento de unlock de un item que no tiene lockeado
Cerrar
Items
A B



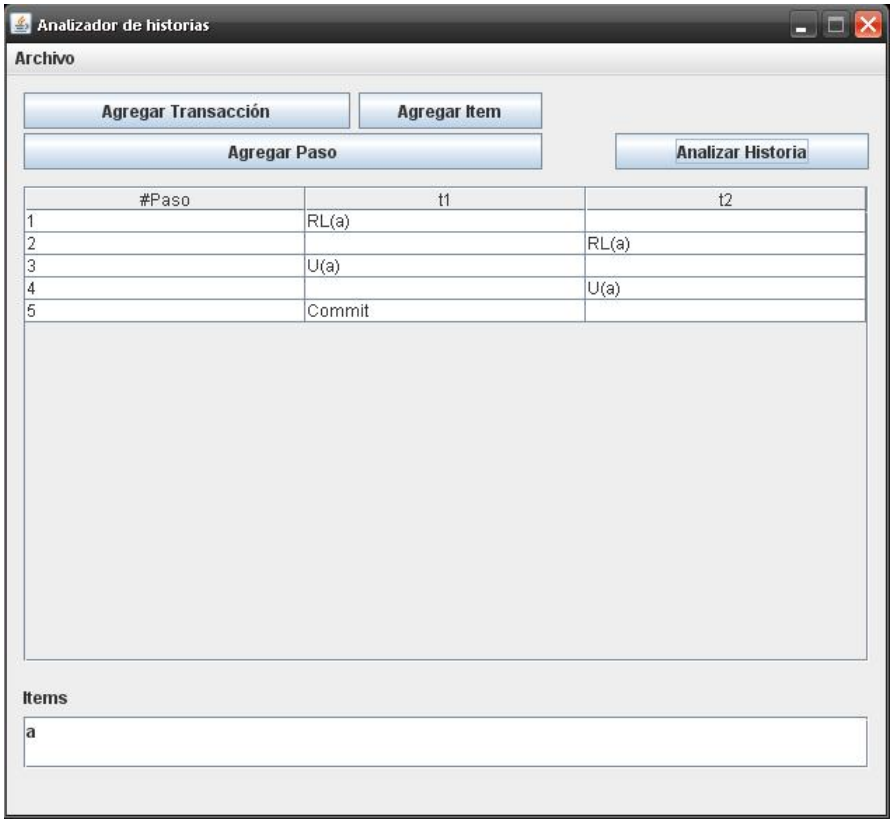
- Caso: historia no legal serializable

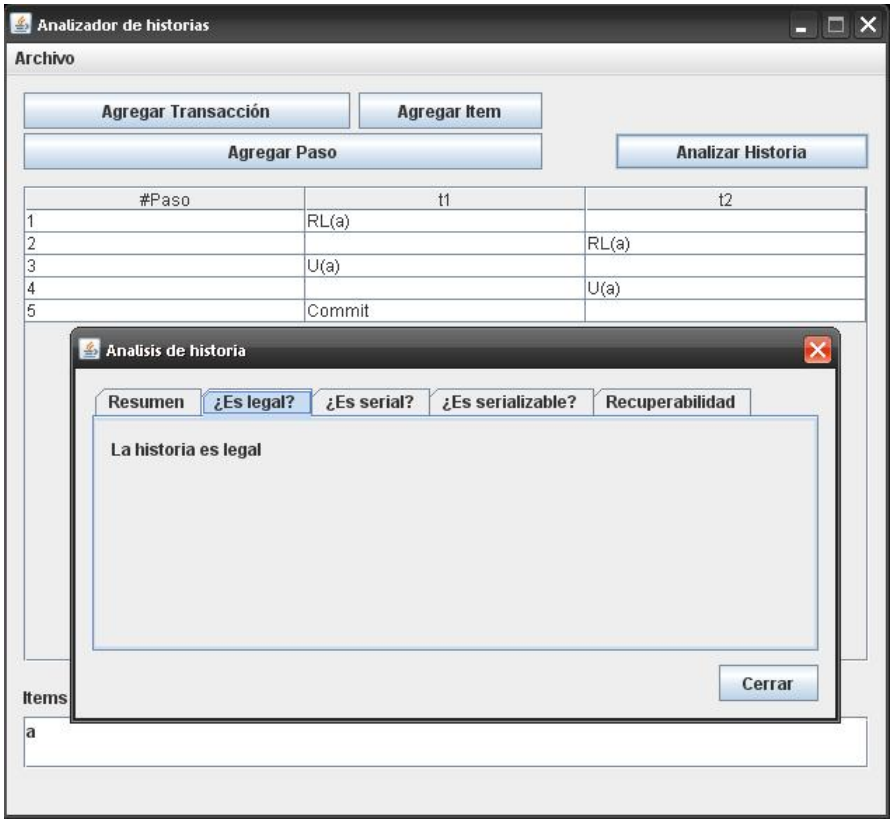


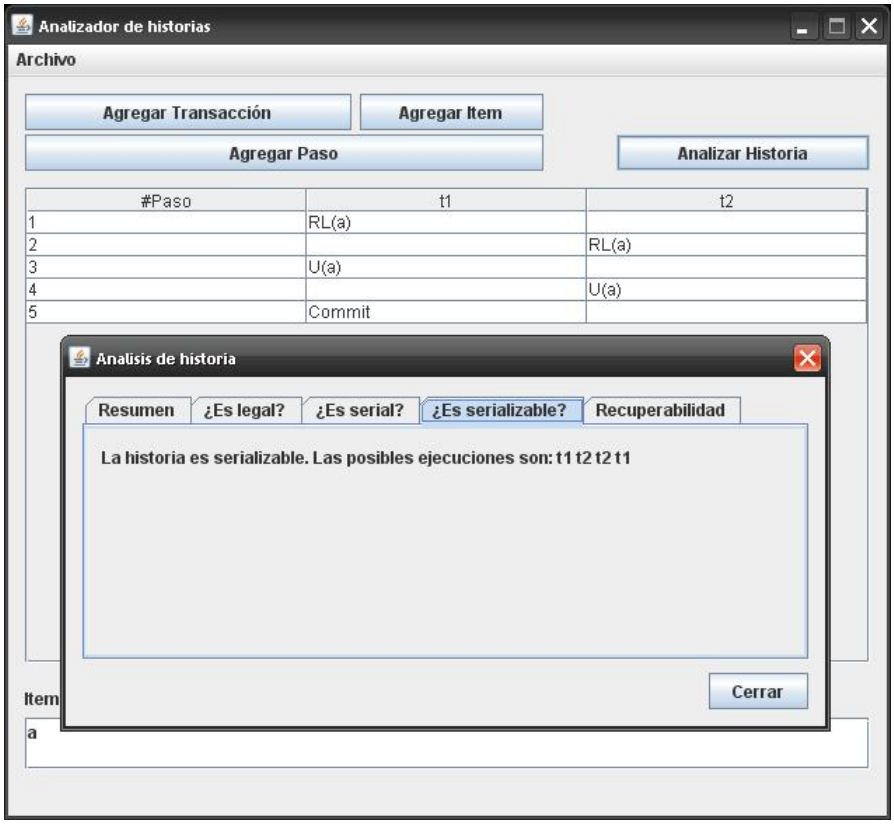




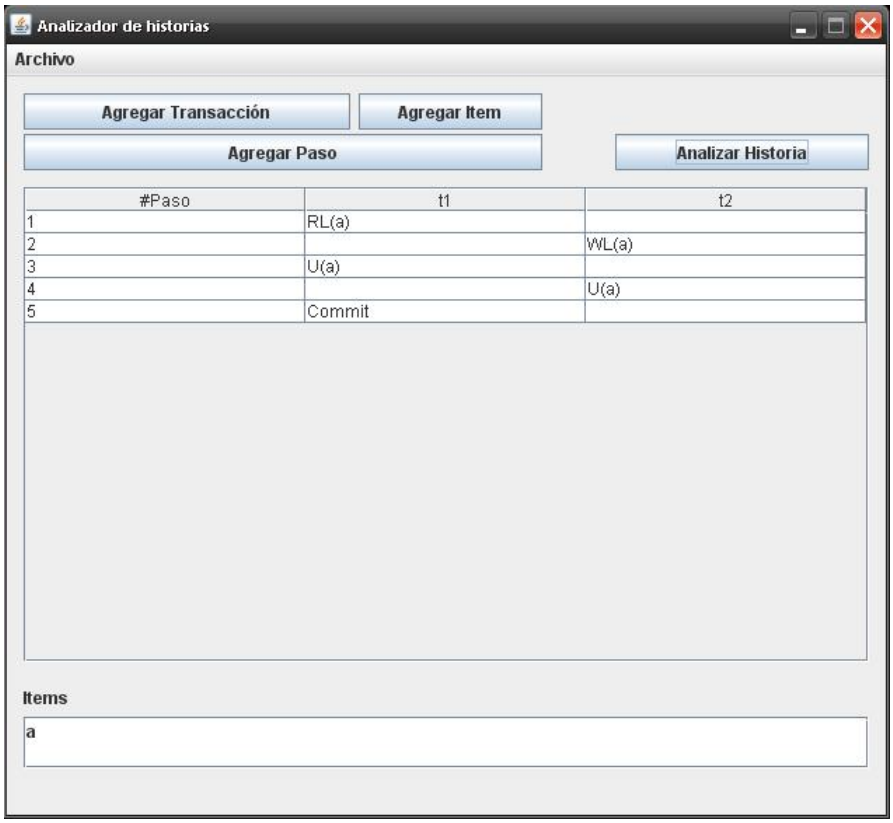
- Caso: historia legal serializable

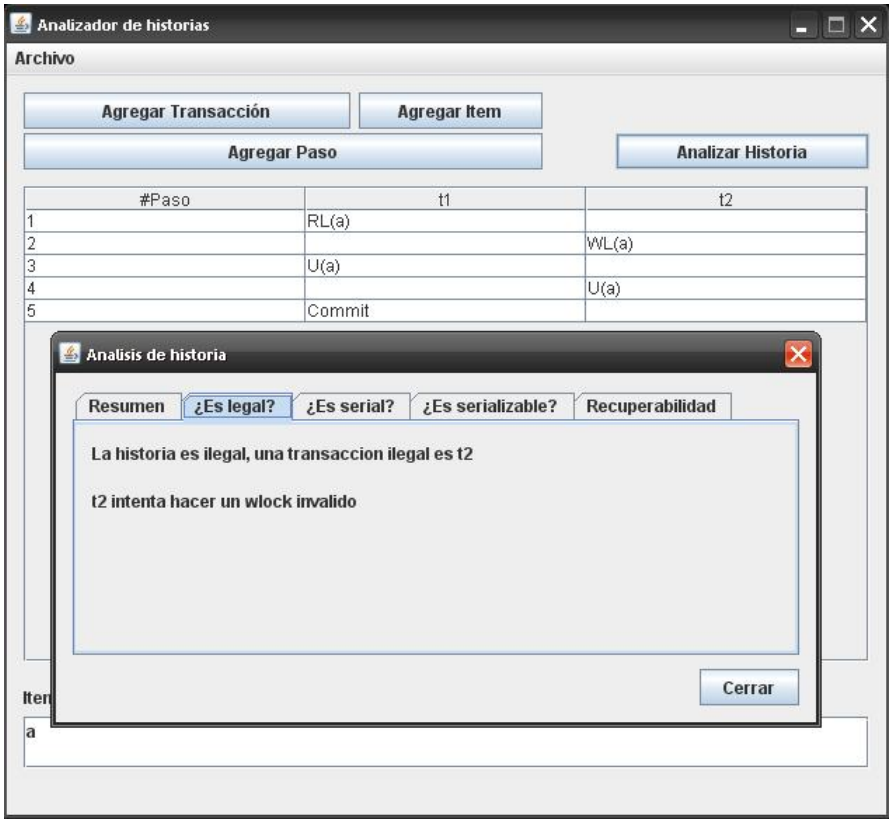

Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
t1
t2
1
RL(a)
2
RL(a)
3
U(a)
4
U(a)
5
Commit
Items
a


Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
t1
t2
1
RL(a)
2
RL(a)
3
U(a)
4
U(a)
5
Commit
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es legal
Cerrar
Items
a

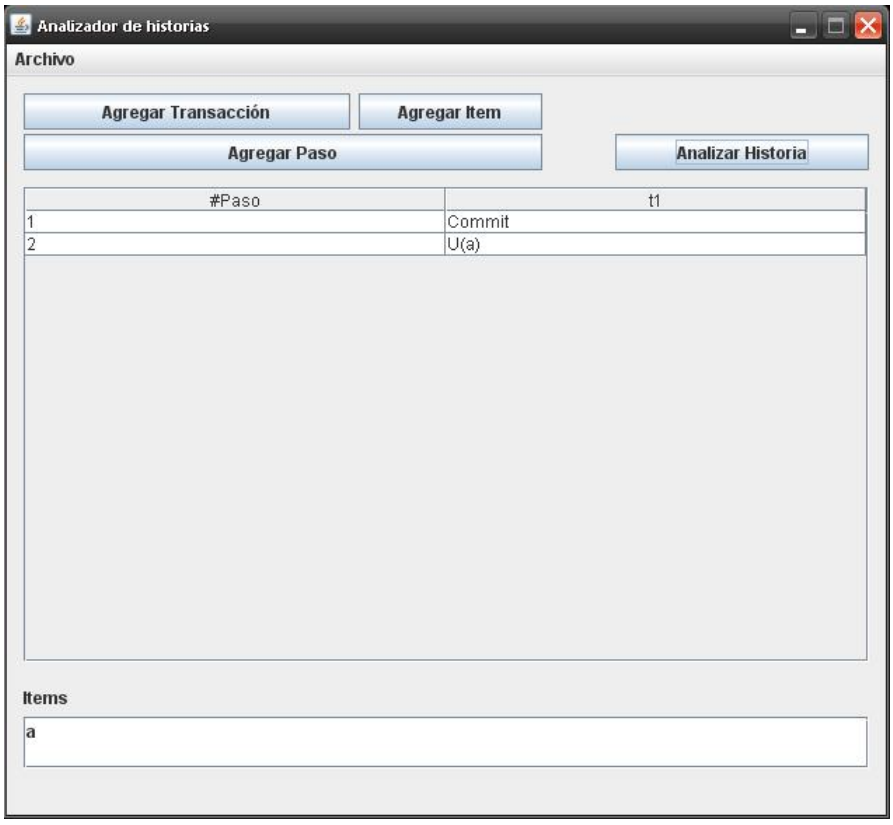


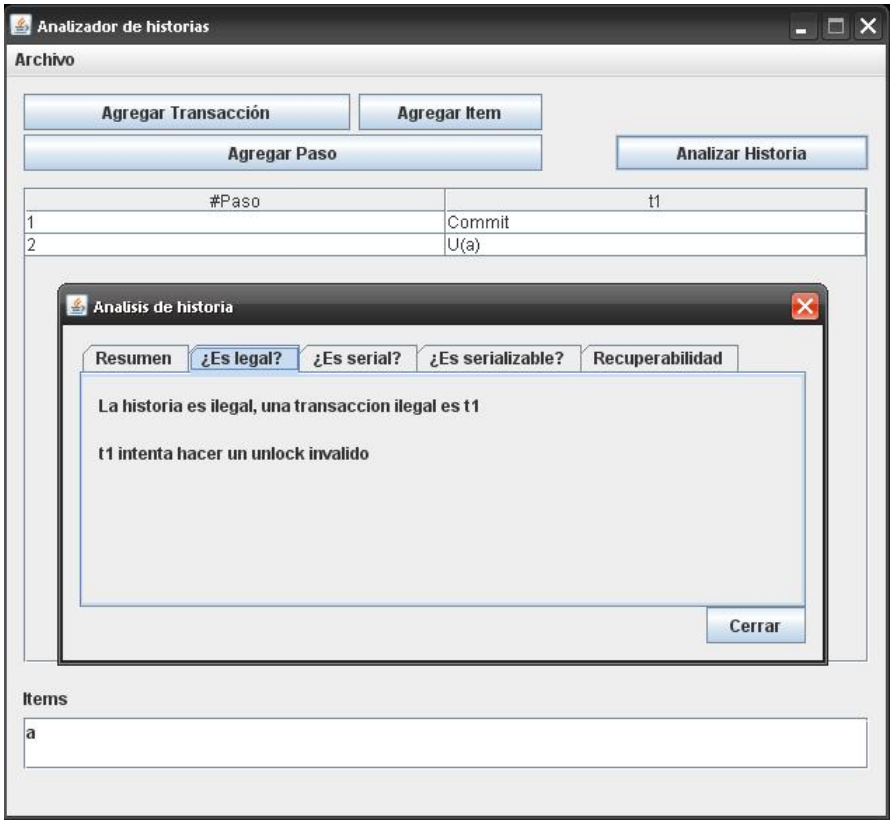
- Caso: historia no legal

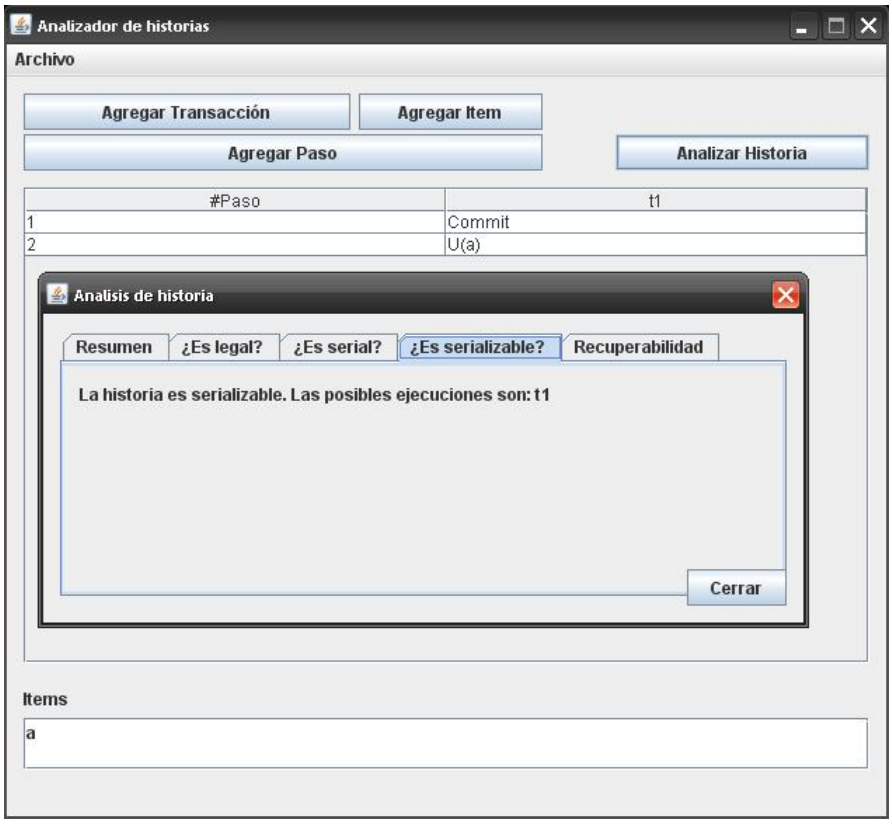




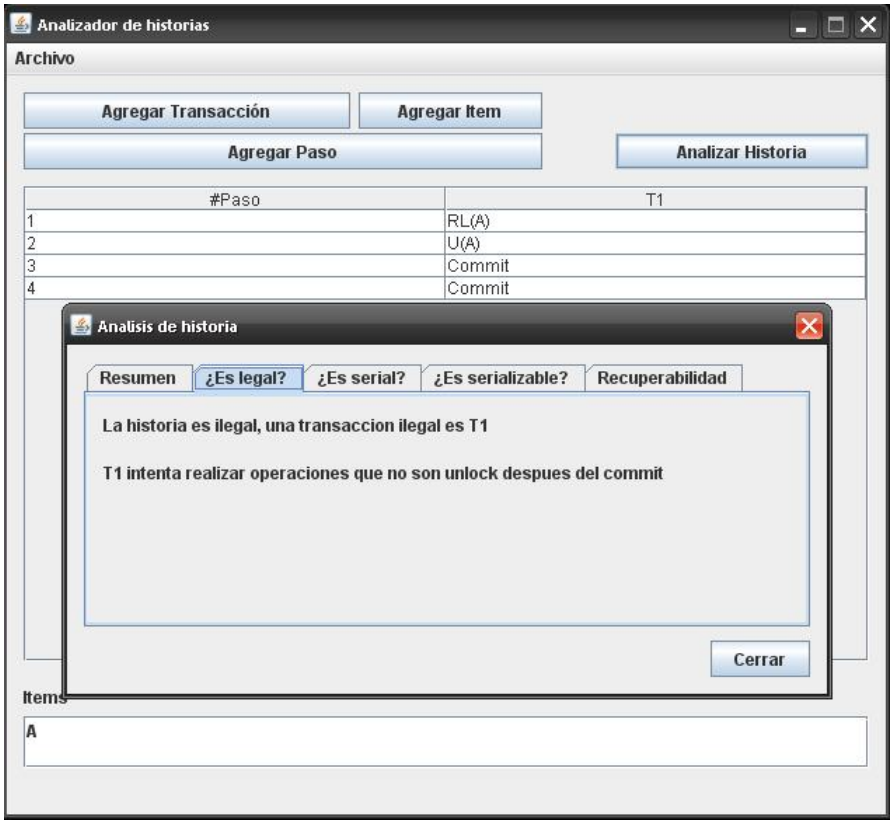
- Caso: historia no legal serializable (mostrando como funciona con una transaccion que hace unlock de algo que no hizo lock)

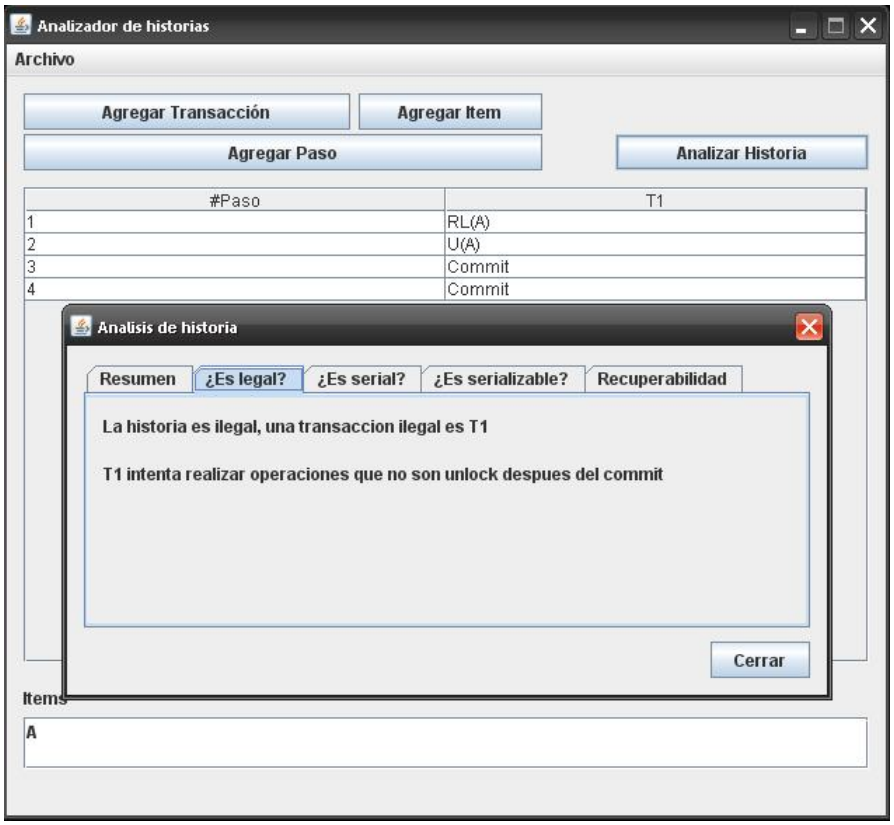






- Caso: historia no legal (mostrando como funciona con operaciones que no son unlock luego de commit)


Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
1
RL(A)
2
U(A)
3
Commit
4
Commit
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es ilegal, una transaccion ilegal es T1
T1 intenta realizar operaciones que no son unlock despues del commit
Cerrar
Items
A


Analizador de historias
Archivo
Agregar Transacción
Agregar Item
Agregar Paso
Analizar Historia
#Paso
T1
1
RL(A)
2
U(A)
3
Commit
4
Commit
Analisis de historia
Resumen
¿Es legal?
¿Es serial?
¿Es serializable?
Recuperabilidad
La historia es ilegal, una transaccion ilegal es T1
T1 intenta realizar operaciones que no son unlock despues del commit
Cerrar
Items
A