1 Overview

This is the documentation of Changming Yang's implementation of his master thesis on the topic "Effective Control Synthesis under Partial observation" the whole work is based on the cpp library libFAUDES, and is also implemented in cpp

2 Problem Definition

According to the article, the goal is to design a complete, deadlock-free suprevisor V for plant G such that

$$S(V/G) \subseteq E \tag{1}$$

where the plant is designed to be a Rabin Automata with a trivial acceptance condition, which can be simplfied into a Buechi automata as follow:

$$\begin{aligned} G &= (\Sigma, X_G, \delta_G, x_{0G}, (X_G, X_G)) \\ G &= (\Sigma, X_G, \delta_G, x_{0G}) \end{aligned} \tag{2}$$

and the Specification is a Rabin Automata with a single acceptance condition:

$$E = (\Sigma, X_E, \delta_E, x_{0E}, (R_E, I_E)) \tag{3}$$

3 Operation

3.1 Define RabinPairs

Because libfaudes has a template of Generator, I can construct a Rabin Automata by only defining a Rabin Acceptance Condition (one single rabin pair is utilised in our case), in the practical construction, multiple RabinPairs can be inserted.

Constructors and Destructors:

```
class AttributeRabinPairs :  public AttributeVoid
```

Default Constructor:

```
AttributeRabinPairs()
```

Copy Constructor:

```
AttributeRabinPairs(const AttributeRabinPairs& rOther)
```

Destructor:

```
virtual  AttributeRabinPairs()
```

Assignment operator:

```
AttributeRabinPairs& operator=(const AttributeRabinPairs& rOther)
```

Equality comparison operator:

```
bool operator==(const AttributeRabinPairs& rOther) const
```

Inequality comparison operator:

```
bool operator!=(const AttributeRabinPairs& rOther) const
```

where the Rabin Acceptance condition is defined with:
$Inf(\pi) \bigcap R \neq \emptyset$, There exists at least one element in R-set can be infinitely visited.
$Inf(\pi) \bigcap I = \emptyset$, All the elements in R-set are finitely visited.

Add a Rabin pair of R-set and I-set:

```
void AddRabinPair(const StateSet& rRSet, const StateSet& rISet)
```

Set Rabin pairs attribute for a Generator:

```
void SetRabinPairsAttribute(Generator& rGen, const AttributeRabinPairs& rAttr)
```

Get Rabin pairs attribute from a Generator:

```
AttributeRabinPairs GetRabinPairsAttribute(const Generator& rGen)
```

Write Generator to console, including Rabin pairs:

```
void CustomDWrite(const Generator& rGen)
```

3.2 Build Controlled System

Before construct a Controlled System, a control pattern **C** must be designed by extending the alphabet Σ, then a Controlled System with a trivial Acceptance condition can be designed as below, furthermore, the Specification is also extended with the same alphabet.

$$G_C = (\Sigma \times C, X_G, \delta_{GC}, x_{0G}) \tag{4}$$

$$E_C = (\Sigma \times C, X_E, \delta_{EC}, x_{0E}, (R_{EC}, I_{EC})) \tag{5}$$

Apply control patterns to a Generator, preserving Rabin pairs:

```
void ApplyControlPattern(const Generator& rPlantGen, const EventSet&
rControllableEvents, Generator& rControlledGen)
   @rPlantGen: The plant Generator
   @rControllableEvents: Set of controllable events
   @rControlledGen: Output controlled Generator
```

3.3 Compute A_C

A_C is the Product of controlled system and Specification $\mathbf{G_C} \times \mathbf{E_C}$

$$A_C = (\Sigma \times C, X_G \times X_E, \delta_c, (x_{0G}, x_{0E}), (R_c, I_c)) \tag{6}$$

one thing need to be take care is, the Rabin Pairs result of a universal Product of two Rabin Automata should be:

$$\begin{aligned} R_c &= R_1 \times X_2 \cup X_1 \times R_2 \\ I_c &= I_1 \times X_2 \cup X_1 \times I_2 \end{aligned} \tag{7}$$

In our case, since there is not a R_1 and I_1, so the final rabin pair can be simplfied to:

$$\begin{aligned} R_c &= X_1 \times R_2 = X_G \times R_E \\ I_c &= X_1 \times I_2 = X_G \times I_E \end{aligned} \tag{8}$$

To simplfy the code, I use the function $Product(Plant, Specification)$ from libFAUDES to compute the product, return a new system GC; Seperately compute product of Rabin pairs from two Generators, then insert the new RabinPairs into GC:

$Product(Plant, Specification, ompositionMap, GC)$

```
void ProductRabinPair(const Generator& rPlant, const Generator& rSpec,
const std::map<std::pair<Idx, Idx>, Idx>& rCompositionMap, Generator& GC)
  @rPlant: First Generator
  @rSpec: Second Generator
  @rCompositionMap: Mapping from state pairs to product states
  @GC: Output product Generator
```

3.4 Compute the Projection without determinization A_o

The natural Projection is used to simplfy the Observation Mask, $RabinProjectNonDet()$ is imitated based on the function $ProjectNonDet()$ from libFAUDES to compute the Projection **without** determinization, I will later use another algorithm to determinize.

$$A_o = (\Sigma \times C, X_G \times X_E, \delta_o, (x_{0G}, x_{0E}), (R_c, I_c)) \tag{9}$$

```
void RabinProjectNonDet(const Generator& rGen, const EventSet&
rProjectAlphabet, rProjectAlphabet,Generator& rResGen)
  @rGen: Input Generator
  @rProjectAlphabet: Alphabet to project onto
  @rResGen: Output projected Generator
```

3.5 PseudoDeterminization

We can now construct a deterministic automaton accepting a language having the same π_ϵ -projection as $R(A_v)$ by adapting the construction of the article "A. Emerson and C. S. Jutla, On simultaneously determinizing and complementing ω- automata"

```
struct TreeNode {
  StateSet stateLabel;           // Set of states from the original automaton
  std::set<NodeIdx> aSet;        // A-set
  std::set<NodeIdx> rSet;        // R-set
  std::vector<NodeIdx> children;} // child node
  enum Color { WHITE, RED, GREEN } color;  //Node color
```

```
class LabeledTree {
  NodeIdx createNode() {}
  void deleteNode(NodeIdx nodeId) {}  \\Methods for node manipulation}
```

Main Algorithm Steps

1.Initialization

Retrieves Rabin pairs from the input Generator
Creates an initial tree with root node containing the initial state

2.State Processing Loop

Uses a queue to process pending states For each state, processes all possible event transitions

3.Tree Transformation Rules (9 steps)

Step 1: Colors all nodes white
Step 2: Updates state labels based on transitions
Step 3: Creates new nodes for potential acceptance violations
Step 4: Maintains state disjointness between sibling nodes
Step 5: Removes nodes with empty state labels
Step 6: Detects "red breakpoints" (state label equals union of children's labels)
Step 7: Updates A-sets and R-sets
Step 8: Handles green coloring (when A-set is empty and node isn't red)
Step 9: Maintains tracking of red nodes

4.Tree Signature Computation and State Mapping

Computes signatures to detect previously encountered tree structures
Maps signatures to existing states or creates new ones

5.Rabin Pair Construction

Creates Rabin pairs for the output Generator based on node colors
Red nodes correspond to R-sets, green nodes to I-sets