COHO Reachability Analysis Tool Manual

Chao Yan

December 5, 2014 (Version 1.0)

Contents

Chapter 1

Introduction

CRA (Coho Reachability Analysis tool) is a reachability analysis tool from the University of British Columbia. It is based on the novel representation method projectagon[7, 3, 2]. It is originally developed for the Coho circuit verification platform. It is extended as a standalone tool for reachability analysis for AMS verification, hybrid systems, control systems, *etc.*

CRA provides two interfaces for users: the high-level *hybrid automata* interface and the detailed *projectagon* interface. The projectagon interface provides basic projectagon operations which enable users to perform customized reachability computations. The hybrid automata interface accepts a user-provided hybrid system modeled by hybrid automata and computes all reachable system states automatically. It is recommended to use the hybrid automata interface for most users.

1.1 Installation

CRA is open-sourced. Users can download from *github* by:

```
git clone https://github.com/dreamable/cra.git
```

CRA supports Linux, Unix and MacOS. It requires Matlab (R2008+) and Java (5.0+) installed in the system. It can be installed by:

```
cd cra
sh install.sh
```

Note that the installation script ask users a question that if the CPLEXlinear program solver available in the system or not. The CPLEX solver could improve performance. To enable the solver, users must configure CPLEX and system to support the CPLEXINT package[1].

[1] http://control.ee.ethz.ch/ hybrid/cplexint.php

1.2 Simple Usage

CRA is a Matlab package. To use it, please first start Matlab, then run your Matlab codes by:

```
cra_open
%user_code
cra_close
```

1.3 Examples

Examples are available under the `example` directory in the code. Please see chapter 5 for details.

1.4 Organization

chapter 2 presents the projectagon structure and operations. chapter 3 describe the hybrid automata interface. It is recommended to use the hybrid automata interface for most users, while the projectagon interface are used for highly customized reachability computation. CRA also implemented some packages which could be useful for some users, e.g. linear program solver and polygon operations based on arbitrary precision rational numbers. chapter 4 provides the APIs of these functionalities. Examples that uses CRA are described in chapter 5.

1.5 Learn More

There are several paper published which are good sources to understand the higher level ideas.

To understand implementation details, please use the Matlab help files by

```
help funcName
```

Table 1.1: Publications

Publication	**Note**
[7]	This is Chao Yan's PhD thesis. It is a comprehensive document with most details. Reachability analysis are covered mostly in chapter 4 and chapter 2.
[2, 3]	The initial idea of projectagon is presented in these papers. They are good documents to understand the basic idea of projectagon. But there are a little dated, especially the implementation details.
[14, 6]	These documents provides details of projection algorithm, polygon operations and linear program solver, especially arbitrary precision rational computation employed to solve numerical problems.
[9, 10, 11, 13, 16, 12, 15]	Examples of how CRA been used to verify circuits.
[8]	COHO Circuit Modeling Tool Manual.

Chapter 2

Projectagon

2.1 Introduction

Reachability analysis completely explores the state space of a system by solving both continuous and discrete dynamics. A fundamental problem of reachability analysis is to computes over-approximated results of differential inclusions

$$\begin{aligned} \dot{x} &\in F(x) \\ X_0 &\subseteq \Omega \end{aligned}$$

CRA solves the problem based on the projectagon representation. It over-approximates $F(x)$ by linear differential inclusions and Ω by projectagon.

Table 2.1 compares reachability analysis and simulation algorithms. We distinguish two kinds of reachable regions in the document: a *reachable set* is the set of states occupied by trajectories at some specified time, and a *reachable tube* is the set of states traversed by those same trajectories over all times in a closed or unbounded interval.

Table 2.1: Simulation v.s. Reachability Analysis

	Simulation	Reachability Analysis
Dynamics	Differential equations: $\dot{x} = f(x)$	Differential inclusions: $\dot{x} \in F(x)$
Initial	One point: $x_0 \in \Omega$	A region: $X_0 \subseteq \Omega$
Solution	Approximated: $\hat{x}(t) \approx x(t), \forall t \in R^+$	Over-approximated: ***reachable set***: $\{X(t) \| x(t, x_0), \forall x_0 \in X_0\}$ ***reachable tube***: $\{X(t) \| x(t, x_0), \forall x_0 \in X_0, \forall t \in [t_l, t_h]\}$

2.2 Projectagon structure and operations

projectagons are a data structure for representing high dimensional polyhedra by their projections onto two-dimensional planes, where these projection polygons are not required to be convex. The representation is accurate and efficient: it can represent non-convex polyhedra accurately, projectagon operations can be implemented by efficient polygon operations. For more details, please see [7, 2].

The function *ph_create* is to construct a projectagon.

```
ph = ph_create(dim,planes,hulls[,polys[,type]]);
```

To create a projectagon, users need to provide

- dim: number of dimensions
- planes: projection planes, should be a $dimx2$ matrix
- polys: projection polygons, should be a cell, each item should be a polygon by *poly_create* (see section 4.3).
- hulls: convex hull of projection polygons, should be a cell, each item should be a convex polygon.
- type: projectagon types.

For example, to create a projectagon a unit cube,

```
polys{1} = poly_create([0,0,1,1;0,1,1,0]);
polys{2} = poly_create([0,0,1,1;0,1,1,0]);
polys{3} = poly_create([0,0,1,1;0,1,1,0]);
ph = ph_create(3,[1,2;1,3;2,3],polys,polys);
```

CRA supports three types of projectagon: 1) general (or non-convex) projectagon, 2) convex projectagon, 3) bounding box. General projectagon is the most accurate representation with most complex operations; while bounding box projectagon has most simple operations with largest error. This provide users a way to get a trade-off between accuracy and performance for different applications. Usually, bounding box projectagon has so large approximation error that can be rarely used for non-trivial problem. Convex projectagon is more efficient for most simple problems with acceptable approximation error. General projectagon is used for complex problems that require small approximation error. Convex projectagon can be constructed from linear programs. Bounding box projectagon can be constructed from intervals. For example:

```
lp = lp_create([1,0;-1,0;0,1;0,-1],[1;0;1;0]);
ph = ph_createByLP(dim,planes,lp);
bbox = [0,1;0,1];
ph = ph_createByBox(dim,planes,bbox);
```

Different types of projectagons can be converted by

```
ph = ph_convert(ph,new_type);
```

CRA supports the operations shown in Table 2.2.

Table 2.2: Projectagon Operations

Operations	**Functions**	Note
Union	ph = ph_union({set of ph})	All ph must have same dim and planes
Intersect	ph = ph_intersect({set of ph})	All ph must have same dim and planes
Intersect with line	ph = ph_intersect(ph, line)	Result is bloated to be a projectagon
Intersect with LP	ph = ph_intersect(ph, lp)	
Empty	isempty = ph_isempty(ph)	check if the projectagon has feasible region or not
Simplify	ph = ph_simplify(ph)	Simplify the projection polygons by over-approximate the region slightly
Projection	ph = ph_project(ph,plane)	Project the ph onto two-dimensional subspace.
Contain	iscontain = ph_contain(ph1,ph2)	Check if ph2 is contained by ph1
Contain Point	iscontain= ph_containPts(ph,pts)	Check if points are contained in the ph or not
Canonical	ph = ph_canon(ph)	make the projectagon canonical
MinkSum	ph = ph_minkSum(ph1,ph2)	Only for bounding box projectagons
Change planes	ph = ph_chplanes(ph,planes)	Update ph to use a new set of projection planes

2.3 Reachability algorithm and configuration

In CRA, reachable set and reachable tubes are computed by

```
new_ph = ph_advance(ph,opt);   % reachable set
ph_tube = ph_succ(ph,new_ph);  % reachable tube
```

Reachable tubes for time interval $[t_0, t_1]$ is over-approximated by bloating convex hull of reachable set for time t_0 and t_1.

$$ph_{[t_0,t_1]} \in bloat(convex(ph_{t_0}, ph_{t_1}));$$

User must define the system dynamics to compute reachable regions by

```
cra_cfg(set,'modelFunc',modelFunc)
```

where $modelFunc$ is a function handle of the format

```
models = modelFunc(lp)
```

The function accepts a CohoLP as input (see section 4.2 for details of Coho LP). The return of the function must be a structure with fields A, b, u, representing a linear differential inclusion model (LDI)[1]. A LDI mode is of the format:

$$\dot{x} \in Ax + b \pm u$$

To reduce linearization error, users can use the intersection of several linear differential inclusion models, by returning a cell of models.

The reachability algorithm accepts options which should be specified by a structure with the following fields:

- Parameters for computing models. A crucial step of reachability algorithm is to compute time step to be advance and corresponding maximum moving distance of all trajectories during the time interval. The choice of $timeStep, maxBloat$ pair affects both accuracy and performance significantly.

[1]see section 4.1 for details

Fields	Note
model	Three ways to compute the pair of *timeStep*, *maxBloat*. – `guess_verify`: Guess a pair of *timeStep*, *maxBloat* and verify them at the end. – `bloatAmt`: Compute *timeStep* from *maxBloat*. – `timeStep`: Use user provided *maxStep* with *maxBloat*. Usually, the `guess_verify` provides the best result for both accuracy and performance.
maxBloat	Maximum moving distance of all trajectories in a single advance step. It is used to compute *timeStep* for `bloatAmt` method. Users can specify different values for each variable and direction (increase or decrease)
maxStep	Maximum time step.
bloatAmt	The fixed bloating amount. It is only for the `bloatAmt` method.
timeStep	The fixed user-provided time step. It is only for `timeStep` method.
prevBloatAmt, prevTimeStep	Interval usage for `guess_verify`, should not be changed by users.
ntries	Maximum number of trying to guess a valid pair of *timeStep*, *bloatAmt* for `guess_verify` method.

- Parameters for finding projectagon faces to advance in each step.

Fields	Note
object	Methods to compute object to advance. Valid value includes – `face-bloat`: Advance projectagon faces individually. Faces are bloated outward for soundness. – `face-height`: Advance projectagon faces individually. The height of faces are increased for soundness. Sometimes it has smaller error than `face-bloat`. – `face-none`: Advance projectagon face individually. Faces are not updated for soundness. It has smaller error than `face-bloat` and `face-height`, but may not be sound. It requires non-zero error term in the linear differential inclusion model. – `face-all`: Advance all faces and project them onto all slices. The result is sound, usually with large error bad slow performance. – `ph`: Advance the whole projectagon. Only for convex or bounding-box projectagon.
maxEdgeLen	Maximum length of polygon edges. When object is not `ph`, projection polygons are broken into short edges to reduce error. Larger value can reduce the number of faces but may increase model error.
useInterval	Enable/disable the *interval closure* method to find more accurate faces for *non-convex* projectagon.

- Error control.

Fields	Note
tol	tolerance used to simplify the polygons, see *ph_simplify* for details
riters,reps	To reduce model error, *ph_advance* repeats computation with smaller bloatAmt. The loop exits either the number of iterations is greater than *riters* or the change of bloatAmt is greater than *reps*
constraintLP	Global contraint of reachable region. It can help to reduce approximation error.
canonOpt	The parameters of *ph_canon* function.
intervalOpt	The parameters of *ph_interval* function

To simplify the configuration, we provide recommended value by *ph_getOpt*(*opt*), which provides value for *opt* as:

- `default`: the default template of opt (default value of type)
- `fast`: optimize the performance
- `accurate`: optimize for accuracy.
- `stable`: use the most numerical stable algorithms.

Instead change the structure directly, it's highly recommend to update options by the function *ph_setOpt*:

```
opt = ph_setOpt(opt,filed,value);
```

2.4 Functions

This package is the core of CRA. It has two parts: projectagon operations and reachability computation algorithms. Here lists all functions can be used by users with short descriptions. For details, please check in Matlab help document.

2.4.1 Projectagon

2.4.2 Reachability computation

Table 2.3: Projectagon Functions

Functions	**Description**
ph_create	create a general (non-convex) projectagon from polygons.
ph_createByLP	create a convex projectagon from lps.
ph_createByBox	create a bbox projectagon from bounding box.
ph_rand	generate a random projectagon (mainly for test purpose).
ph_convert	convert the type of projectagon.
ph_get	get structure info.
ph_isempty	check if the projectagon is empty or not
ph_intersect	intersection of two or more projectagons
ph_union	union of two or more projectagons
ph_intersectLP	intersection of a projectagon and LP
ph_intersectLine	intersection of a projectagon and line (result is bloated to be a projectagon)
ph_simplify	simplify the projectagon.
ph_contain	check if a projectagon contained by another one
ph_containPts	check if points are contained by a projectagon
ph_project	project a projectagon onto 2D subspaces
ph_canon	make the projectagon canonical
ph_minkSum	MinkSum operations (only support bbox now)
ph_chplanes	change the project planes a projectagon
ph_promote	promote a set of projectagons to have the same planes
ph_interval	interval closure calculation
ph_regu	make ill-conditioned projectagon a normal one
ph_display	display a projectagon
ph_display3d	display a 3D projectagon in 3D space
phs_display	display a set of projectagon

Table 2.4: Reachability Analysis Functions

Functions	**Description**
ph_advance	the main function to compute advanced projectagon
ph_advanceSafe	this function caches exceptions from *ph_advance* and try with different options to continue the computations. Exceptions may from over-approximation or computation error.
ph_succ	compute the reachable tube during time [t1,t2]
ph_getOpt	get default options
ph_checkOpt	check if the option is correct
ph_setOpt	update the options

Chapter 3

Hybrid Automata

Hybrid automata is widely used mathematical model for hybrid systems. CRA provides a general hybrid automata interface in MATLAB. Given a hybrid automaton, CRA can perform reachability analysis automatically. The interface also enables users to easily improve performance, reduce approximation, automate various calculations, *etc.*

3.1 A Quick Start

A hybrid automaton in CRA consist of *states*, *transitions*, *source states*, *initial regions*, and *global invariants*. In this chapter, we use a simple example as demo to show the basic flow of creating and using hybrid automata in CRA. More examples are available in the *example* directory of the CRA codes.

3.1.1 Problem

The example has three variables x, y, z. The initial region is $[x, y, z] \in [0, 0.1] \otimes [0, 0.1] \otimes [0, 0.1]$. The system dynamic has three modes:

1. $\dot{x} = 1, \dot{y} = \dot{z} = 0$, if $[x, y, z] \in [0, 1] \otimes [0, 0.1] \otimes [0, 0.1]$
2. $\dot{y} = 1, \dot{x} = \dot{z} = 0$, if $[x, y, z] \in [0.9, 1] \otimes [0, 1] \otimes [0, 0.1]$
3. $\dot{z} = 1, \dot{x} = \dot{y} = 0$, if $[x, y, z] \in [0.9, 1] \otimes [0.9, 1] \otimes [0, 1]$

So x will increase first and reach the value of 1, followed by the increase of y and then z.

We use the example to show the basic work flow of CRA.

3.1.2 System dynamics

One of the most important step is to build the system dynamics. Users must provide a function which over-approximate the dynamics by LDI models($\dot{x} \in$

$Ax + b \pm u$, see section 4.1 for details). For the example above, it is trivial: A and u are always zeros[1], the value of b depends on the mode. The code is:

```
function ldi = ex_demo_model(lp,mode)
  A = zeros(3,3);
  b = zeros(3,1); b(mode) = 1;
  u = 1e-9;  % to avoid empty projection
  ldi = int_create(A,b,u);
```

3.1.3 Build hybrid automata

The next step is to translate the hybrid system into a hybrid automaton for CRA.

First, we need to create automata states. For the example, apparently, there are three states, each one corresponds to a dynamics mode. A state needs a *name*, *dynamics* and *state invariants*. We use "s1", "s2", and "s3" as state names. State dynamics is from the function above. State invariants are straightforward from dynamic modes.

Second, we need to create state transitions. For the example, it is obviously that computation should be performed in state "s1". The result is used as starting point for computation in state "s2". So there is a transition from state "s1" to state "s2". Similarly, the transition from state "s2" to state "s3" is added to the automata.

Third, we need to add initial states and initial regions. For the example, state "s1" is the initial states. The initial region is the cube $[0, 0.1] \otimes [0, 0.1] \otimes [0, 0.1]$. The region should be represented by a projectagon. Here, three projections planes $[x, y], [y, z], [x, z]$ are used to create the projectagon. For more details about creating projectagon, please check section 2.2.

The code is:

```
function ha = ex_demo_ha
  % initial region
  x = 1; y = 2; z = 3; dim = 3; planes = [x,y;x,z;y,z];
  bbox = [0,0.1;0,0.1;0,0.1];
  initPh = ph_createByBox(dim,planes,bbox);
  initPh = ph_convert(initPh,'convex');

  % states
  bbox1 = [0,1;0,0.1;0,0.1]; inv1 = lp_createByBox(bbox1);
  bbox2 = [0.9,1;0,1;0,0.1]; inv2 = lp_createByBox(bbox2);
  bbox3 = [0.9,1;0.9,1;0,1]; inv3 = lp_createByBox(bbox3);
  states(1) = ha_state('s1',@(lp)(ex_demo_model(lp,1)),inv1);
  states(2) = ha_state('s2',@(lp)(ex_demo_model(lp,2)),inv2);
  states(3) = ha_state('s3',@(lp)(ex_demo_model(lp,3)),inv3);
```

[1] u is increased slightly to make it non-zero to avoid empty projection error during reachability computation.

```
% trans
trans(1) = ha_trans('s1','s2');
trans(2) = ha_trans('s2','s3');

% source
source = 's1';

% create hybrid automaton
ha = ha_create('demo',states,trans,source,initPh);
```

3.1.4 Perform computation

Given the automaton, CRA computes reachable sets and reachable tubes[2] in all automata states automatically. The code below shows how to perform computation and show computation reachable region.

```
cra_open;
ha = ex_demo_ha;     % get hybrid automaton
ha = ha_reach(ha);  % perform reachability analysis
ha_reachOp(ha,@(reachData)(phs_display(reachData.sets))); % result
cra_close;
```

3.2 Hybrid Automata

CRA provides a MATLAB function for creating an automaton:

```
ha = ha_create(name,states,trans,sources,initials,[inv,[path]]);
```

A hybrid automaton consists of

- **Name**: a string.
- **States**: automaton states created by *ha_state*, *ha_stableState* or *ha_transState*.
- **Transitions**: transitions between states, created by *ha_trans*.
- **Sources**: source states, could be multiple states. Reachable computation started from source states.
- **Initials**: initial regions for each source states. The initial region must be a projectagon.
- **Invariants**: global invariants for all states. The invariant must be specified by COHO linear programs. By default, invariant is empty.
- **Path**: place to save reachable computation results, e.g. reachable sets. By default, the result is save in the current directory.

[2]See section 2.3 for details

3.2.1 Automata states

CRA provides the following function to create an automata state:

```
state = ha_state(name,modelFunc,[inv,[phOpt,[callBacks]]]);
```

A state consists of

- **Name**: a unique string
- **State dynamics**: system dynamics in the state. It must be specified by a function of the form

  $$ldi = \text{modelFunc}(lp);$$

  where lp is a COHO linear program as shown in section 4.2, and ldi is a linear differential inclusion mode as shown in section 4.2.
- **State invariants**: the invariant region for the state. It must be specified by a COHO linear program. Each constraint of the linear program defines a *gate*, which is used for state transition. The intersection of reachable region and *gate* is calculated during the reachable computation. The result is used as initial region for other states. By default, the state invariant is empty.
- **phOpt**: user can apply different configuration for projectagon. This includes
  - type: projectagon types (see section 2.2 for details).
  - planes: projectagon planes.
  - fwdOpt: configurations for *ph_advance* (see section 2.3 for details).
- **CallBacks**: Users can provide functions which will be performed during reachable computation. Current supported call backs include
  - exitCond: This function decides when to terminate the reachable computation in the state.
  - sliceCond: This function decides when to slice reachable regions by *gates*.
  - beforeComp: This function is executed before the reachable computation.
  - afterComp: This function is executed after the reachable computation.
  - beforeStep: This function is executed before each step of reachable computation.
  - afterStep: This function is executed after each step of reachable computation.

  For details, please see subsection 3.3.2 and the algorithm in subsection 3.2.3.

3.2.2 Automata transitions

CRA provides the following function to create an automata transition:

```
state = ha_trans(source, target, [gate, [resetMap]]);
```

A transition brides a source state and a target state. It consists of

- Source state: the name of source state must be provided.
- Target state: the name of target state must be provided.
- Gates: The gate ID of source state. By default, it is zero. The virtual gate 0 means the reachable regions for the source state are used as initial regions of the target state. Otherwise, the intersection of reachable regions and gate are used as initial regions of the target state.
- Reset Map: a function to update the initial region for target state. It is of the form
$$ph = resetMap(ph).$$

3.2.3 Reachable computation algorithm

The reachable computation is performed by the function:

```
ha = ha_reach(ha);
```

The reachable computation flow is illustrated in the pseudo-code below:

```
For each state
  % computation initial regions
  Find all source states by transitions
  Compute initial regions by transition source and gates.

  % Specify state dynamics
  cra_cfg('set','modelFunc',state.modelFunc);

  % Preform reachability computation
  state.beforeComp;                   % callback
  while(~state.exitCond)
    prevPh = ph;
    state.beforeStep;                 % callback
    ph = ph_advance(ph,state.phOpt)   % compute reachable set
    tube = ph_succ(ph,prevPh);        % compute reachable tube
    state.afterStep;                  % callback
    if(state.sliceCond)
      % slice reachable tube
      state.slices = ph_intersect(tube,state.inv);
    end
  end
  state.afterComp;                    % callback
end
```

```
% save all reachable data onto disk
```

3.3 Advanced Configuration

3.3.1 Performance v.s Accuracy

Obtaining a good balance between performance and accuracy is an important step for many reachability analysis problems. CRA provides several options to control performance and accuracy.

Reachability computation in each automata state could be specified individually by setting the parameter *pOpt*. It includes

- phOpt.type: Users can use different projectagon types in automata states. Generally speaking, convex projectagon are suitable in most case. Non-convex projectagon are use to optimize accuracy with slower computation. Bounding box projectagon have bad accuracy, may only suitable for extremely simple problems.
- phOpt.plane: Users can specify projectagon planes for each automata state. For a n-dimensional system, the number of planes is in the range of $[n-1,\, n(n-1)/2]$. Generally speaking, the more planes used, the better accuracy is the computation result with the cost of more computation time. Usually, if dynamics of x_i and x_j highly depends on each others, it is recommended to include the plane x_i, x_j to get better accuracy.
- phOpt.fwdOpt.model: User can specify the way to compute advance time step. Usually, the default value of `guess_verify` provides better performance and accuracy. For more details, please check section 2.3.
- phOpt.fwdOpt.object: Generally speaking, `ph` is much faster than other methods with relatively larger error than `face-none`,`face-bloat`,`face-height`. The performance of `face-none`,`face-bloat` and `face-height` are similar. `Face-none` has less error than the other two methods, but doesn't guarantee soundness as the other two method do. Note `face-none` requires that the error term of the linear differential inclusion model provided by users can not be zeros. `Face-height` usually has slightly smaller error than `face-bloat`, especially for high-dimensional system. `Face-all` has the largest error and slowest computation, it's not recommended to be used by users. For more details, please check section 2.3.

Slicing is a useful method to reduce error and improve performance. Usually, if a variable x changes rapidly in large range $[x_l, x_h]$ monotonically, slicing the variable into smaller intervals helps to reduce accumulated error and improve performance. Of course, the number of hybrid automata states increase, thus could increase total running time.

Linearization error is an important part of computation error. It is highly recommended to reduce the error term of the linear differential inclusion model computed in the user-provided function as possible. Employing multiple models can also reduce linearization error to obtain more accurate result. However, this usually increases the computation time.

3.3.2 Callbacks

Callbacks provide user the ability to execute their own MATLAB functions during reachability computation. During reachability computation in each state, users can provide functions

- exitCond: Condition to terminate the reachability computation in the state. It is of the format: exitCond = exitCond(info), where info is a structure with fields "ph","prevPh","fwdStep","fwdT",'compT".
- sliceCond: Condition to slice reachable tubes with invariant faces/gates. It s of the format: sliceCond = sliceCond(info), where "info" has fields "complete", "ph", "prevPh", "fwdStep", "fwdT", "compT".
- beforeComp: called before the reachability computation. It is of the format: beforeComp(info), where info has fields "initPh".
- afterComp: called at the end of reachability computation. It is of the format: afterComp(info), where info has the fields "sets","tubes",'timeSteps","faces".
- beforeStep: called before each computation step. It is of the format: ph = beforeStep(info), where info has the following fields: "ph", "prevPh", "fwdStep", "fwdT", "compT".
- afterStep: called after each computation step. It is of the format: ph = afterStep(info), where info has the fields: "ph", "prevPh", "fwdStep", "fwdT", "compT ".

During state transitions, users can also provide functions to update initial regions.

To simplify the usage of call backs, CRA provides templates of callbacks by

```
func = ha_callBacks(callback, method, ...)
```

For example, displaying reachable regions after each computation step could be specified by

```
callBacks.afterStep = ha_callBacks('afterStep', 'display');
```

3.4 Functions

We list the main functions for users with short descriptions. For details, please check in MATLAB help document.

Table 3.1: Hybrid Automata Functions

Functions	**Description**
ha_create	create a hybrid automata
ha_state	create an automata state
ha_stableState	create a stable automata state. The reachable region will not leave the invariant region. Computation is terminated when reachable region converges. Slicing is only performed on the last step.
ha_transState	create a transit automata state. The reachable region will leave the invariant region. Computation is terminated when reachable region leaves the invariant,
ha_trans	create an automata transition
ha_reach	Perform the reachability analysis on the automata
ha_callBacks	Templates for callbacks used in the automata state
ha_get	Get automata information
ha_op	Perform an operation on the automata
ha_reachOp	Perform an operation on reachable data of the automata

Chapter 4

Others

4.1 Linear Differential Inclusion

CRA over-approximates system dynamics by linear differential inclusion on-the-fly. A linear differential inclusion is a structure with fields A, b, u, representing a inclusion of the form

$$\dot{x} = Ax + b \pm u \tag{4.1}$$

It is recommended to construct a linear differential inclusion by

```
ldi = int_create(A,b,u);
```

4.2 Linear Programming

CRA supports COHO linear programs of the form:

$$A\dot{x} \leq b \tag{4.2}$$

where A is a matrix with only one or two non-zero elements on each row. COHO linear programs corresponds to convex hull of projectagons. A COHO linear program can be constructed by

```
lp = lp_create(A,b);
```

CRA implements an efficient solver for COHO linear programs based on arbitrary precision rational numbers. Users can use the solver by

```
[optV,optPt,status] = lp_solve(lp,optDir);
```

Beside the built-in linear program solver, CRA also support MATLAB built-in solver or the CPLEX solvers. These solvers (especially the CPLEX solver) could be faster than our solver which is implemented in JAVA. But these solvers suffer from numerical problems. CRA support hybrid method which tries CPLEX (MATLAB) solver first, and re-solves the problem by our JAVA if failed. The default linear program solver can be configured by users as shown in section 4.4.

CRA also implements a solver to project a COHO linear programs onto two-dimensional subspace. Users can use the solver by

```
hull = lp_project(lp,planes);
```

CRA also implements another solver based on the MATLAB linear program solver. But it is not numerical stable, thus not recommend to use.

4.3 Polygon Operations

CRA implements a package for polygon operations using arbitrary precision rational numbers. It supports operations as shown in Table 4.1.

Table 4.1: Polygon Operations

Operations	**Functions**	Note
Intersect	poly = poly_intersect(set of polys)	intersection of two/more polygons
Union	poly = poly_union(set of polys)	union of two/more polygons
Simplify	poly = poly_simplify(poly,tol)	simplify the polygon (reduce number of vertices)
Convex Hull	hull = poly_convexHull(poly)	convex hull of a polygon
Contain	isc = poly_contain(polyA,polyB)	check if polygon A contains polygons B
Contain points	isc = poly_containPts(polyA,pts)	check if polygon A contains points
Intersect with line	seg = poly_intersectLine(poly,line)	intersection of polygon and lines/segments

4.4 Global Configurations

CRA provide global configuration by

```
value = cra_cfg('get',filed); % check global config value
cra_cfg('set',filed,value);   % set global config
```

Supported configurations and valid values are listed in Table 4.2.

Table 4.2: CRA Global Configurations

Field	**Values**	**Default**	**Note**
modelFunc	models = modelFunc(lp)	@model_create(lp)	function handle for the dynamic system, returns one or more LDI models.
dataPath	valid directory	/var/tmp/user/coho/cra/data/	path to save computation data
phOpt		ph_getOpt	configuration structure for projectagon package. See section 2.3 for details.
lpSolver	`java, matlab, cplex, cplexjava, matlabjava`	`java` (wo cplex) or `cplexjava` (with cplex)	LP solver: recommend to use `java` or `cplexjava`; `matlab` usually has numerical problems
projSolver	`java, matlab, javamatlab, matlabjava`	`javamatlab`	projection solver: recommend to use `java`
polySolver	`java, matlab`(or `saga`);	`java`	polygon operation solver: recommend to use `java`
polyApproxEn	1/0	1	enable over approximation in polygon package
javaFormat	`hex`, `dec`	`hex`	Format of numbers passed between Java and Matlab threads
tol	positive value	1e-6	error tolerance

Chapter 5

Examples

CRA has been applied to solve problems listed in Table 5.1.

Table 5.1: CRA Examples

Problem	**Code**	**Dim**	**Source**	**Note**
Sink	ex_2sink	2	[6]	Two dimensional sink example
TDO	ex_2tdo	2	[4]	Tunnel Diode Oscillator circuit
VDP2	ex_2vdp	2	[3]	Two dimensional Van der Pol oscillator
VDP3	ex_3vdp	3	[6]	Three dimensional Van der Pol oscillator
DM	ex_3dm	3	[3]	Dang and Maler's example
PD	ex_3pd	3	[3]	Play-Doh example
VCO	ex_3vco	3	[1]	Voltage controlled oscillator circuit
PLL	ex_3pll	3	[5]	A digital PLL circuit

Bibliography

[1] Goran Frehse, Bruce H. Krogh, and Rob A. Rutenbar. Verifying analog oscillator circuits using forward/backward abstraction refinement. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 257–262, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 29

[2] Mark R. Greenstreet and Ian Mitchell. Integrating projections. In *HSCC '98: Proceedings of the First International Workshop on Hybrid Systems*, pages 159–174. Springer Verlag, 1998. 5, 7, 10

[3] Mark R. Greenstreet and Ian Mitchell. Reachability analysis using polygonal projections. In *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, pages 103–116, London, UK, 1999. Springer-Verlag. 5, 7, 29

[4] Smriti Gupta, Bruce H. Krogh, and Rob A. Rutenbar. Towards formal verification of analog designs. In *Proceedings of 2004 IEEE/ACM International Conference on Computer Aided Design*, pages 210–217, November 2004. 29

[5] Jijie Wei, Mark R. Greenstreet, Yan Peng, and Ge Yu. Verifying global convergence for a digital phase-locked loop. In *FMCAD13*, 2013. 29

[6] Chao Yan. Coho: A verification tool for circuit verification by reachability analysis. Master's thesis, The University of British Columbia, August 2006. 7, 29

[7] Chao Yan. *Reachability Analysis Based Circuit-Level Formal Verification*. PhD thesis, The University of British Columbia, 2011. 5, 7, 10

[8] Chao Yan. Coho circuit modeling (coho-reach) tool manual. 2014. 7

[9] Chao Yan and Mark R. Greenstreet. Circuit level verification of a high-speed toggle. In *FMCAD*, pages 199–206, Washington, DC, USA, November 2007. IEEE Computer Society. 7

[10] Chao Yan and Mark R. Greenstreet. Faster projection based methods for circuit level verification. In *ASP-DAC*, pages 410–415, Los Alamitos, CA, USA, January 2008. IEEE Computer Society Press. 7

[11] Chao Yan and Mark R. Greenstreet. Verifying an arbiter circuit. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–9, Piscataway, NJ, USA, November 2008. IEEE Press. 7

[12] Chao Yan and Mark R Greenstreet. Oscillator verification with probability one. In Jason Baumgartner and Mary Sheeran, editors, *FMCAD*, pages 165–172. IEEE Press, October 2012. 7

[13] Chao Yan, Mark R. Greenstreet, and Jochen Eisinger. Formal verification of arbiters. *The 16th IEEE International Symposium on Asynchronous Circuits and Systems*, May 2010. 7

[14] Chao Yan, Mark R. Greenstreet, and Marius Laza. A robust linear program solver for reachability analysis. In *Proceedings of the First International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS)*, pages pp231–242, Beijing, China, July 2006. 7

[15] Chao Yan, MarkR. Greenstreet, and Suwen Yang. Verifying global start-up for a mbius ring-oscillator. *Formal Methods in System Design*, pages 1–27, 2013. 7

[16] Chao Yan, Florent Ouchet, Laurent Fesquet, and Katell Morin-Allory. Formal verification of c-element circuits. *The 17th IEEE International Symposium on Asynchronous Circuits and Systems*, April 2011. 7