CELIA

The CELIA plug-in
Edition 0.1, 14 January 2011

by Cezara Dragoi and Mihaela Sighireanu

Table of Contents

1 CELIA Copying Conditions (LGPL)

The CELIA tool is copyright © by the CELIA project, and its partners.

This license applies to all files distributed in the CELIA and CINV tools, including all source code, libraries, binaries, and documentation.

2 Introduction

CELIA is a plug-in of the Frama-C platform which performs static analysis of C programs manipulating (singly linked) lists. More precisely:

- **CELIA** computes for each line of the program reachable from the main function, an assertion expressing the properties of the lists in the heap of the program.
- **CELIA** verifies assertions on the program heap given in the C file.

For this, **CELIA** does a symbolic reachability analysis based on abstract interpretation techniques [Cousot&Cousot'79] and uses special domains and decision procedures for lists.

CELIA is based on several tools released on LGPL licence:

- Frama-C platform for the parsing, typing, and manipulation of C programs,
- Fixpoint library as symbolic fix-point engine on inter-procedural control flow graphs,
- CINV tool for the abstract domains on lists,
- Apron library for the numerical abstract domains.

3 Installing

3.1 Requirements

The installation procedure is also described in the `README` file of the **CELIA** distribution. It requires the following software:

- Project tools: autoconf and automake
- C libraries: GMP and MPFR
- OCaml system,
- Camlidl system,
- Apron library,
- Fixpoint and camllib libraries,
- FRAMA-C platform version (at least) Boron-20100401

To facilitate the installation, **CELIA** distribution contains packages for:

- Apron library, directory `apron-dist`
- Fixpoint and camllib libraries, directory `interproc-dist`

3.2 Configuring

`automake` tools are used to discover your configuration (Frama-C directory, C compiler, etc.) as follows:

- Generate configure script by executing in the distribution directory
  ```
  > autoconf
  ```
- Configure the `Makefile` with the local settings by executing in the same directory
  ```
  > ./configure
  ```
- Set the environment variable `CINV` to the distribution directory:
  ```
  > export CINV=‘pwd‘
  ```

3.3 Compiling

The distribution file `install.sh` does the installation procedure.

During the installation, the following steps are done:

- to install FIXPOINT
  ```
  > make -C interproc-dist all
  > make -C interproc-dist install
  ```
- to install Apron
  ```
  > make -C apron-dist all
  > make -C apron-dist install
  ```
- to compile CINV abstract domains
  ```
  > make -C shapes all
  > make -C shapes ml
  ```
- to compile CINV engine
  ```
  make src/clim.cma
  ```

- to compile CELIA plug-in

  ```
  make -C frama-c-plugin frama-c-Celia.byte
  ```

The binary of the plug-in, `frama-c-Celia.byte`, is available in the `frama-c-plugin` directory.

3.4 Executing

The **CELIA** plug-in uses dynamic libraries included in the distribution. To find these libraries, please set the environment variable `LD_LIBRARY_PATH` as follows:

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CINV/shapes
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CINV/apron-dist/mlgmpidl/lib
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CINV/apron-dist/apron/lib
```

The plug-in is called as follows:

```
frama-c-Celia.opt -celia <input c files>
```

4 Inputs

CELIA plug-in shall receive as inputs the list of C files containing:

- the C function to be analyzed called the entry point of the analysis (by default `main`) and
- all the C functions called by the entry point.

(See the Chapter 5 [Options], page 6 section to change by default value of the entry point.)

The C functions that can be analyzed with **CELIA** shall satisfy the following constraints:

- they shall use only variables of type `int` and `intlist` (the last defined in the file intlist.h),
- the assignment statements for pointer variables shall be elementary, i.e., a pointer can be assigned only if it is `NULL`, only the pointer left values `x`, `x->data`, and `x->next` are allowed,

  This limitation will be removed in future versions of CELIA.
- the conditions on pointer variables shall also be elementary, i.e., only terms `x` and `x->data` are allowed,

  This limitation will be removed in future versions of CELIA.
- the Boolean conditions used in `if` and`while` statements shall be atomic,

  This limitation will be removed in future versions of CELIA.
- the actual parameters in function calls shall be variables,

  This limitation will be removed in future versions of CELIA.
- the output actual parameters in function calls shall be pointers to lists.

The C file in input of **CELIA** can be annotated with specifications in the ACSL logic of Frama-C. The file `intlist.h` defines several predicates that can be used in these annotations. *Currently, only these predicates are considered by CELIA. Also, only annotations corresponding to function pre-conditions are interpreted.* The semantics of these predicates is explained here.

5 Options

The **CELIA** plugin is called as follows:

```
frama-c-Celia.byte <Frama-C options> -celia <input C files>
```

The Frama-C options external to **CELIA** are explained in the Frama-C documentation.

The options relevant for **CELIA** are:

-celia activates the **CELIA** plug-in in Frama-C,

-celia-cinv-opt
specifies the property file from which the options for the analysis has to be read.

The property file is a list of lines

```
key=value
```

where `key` specifies the option and `value` specifies the value of the option.

The following tuples of keys and values are known to **CELIA**:

domain the abstract domain to use (default: shape).

dwdomain abstract domain used for list segments; possible values are lsum (default), mset, and ucons. If this appear several times with different values, it specifies a product of these domains.

maxanon default maximum number of anonymous nodes per segment (default 0); this number can be changed for each function (see option `funspec`).

maxasegm default maximum number of segments with anonymous nodes (default 1, may only be more); this number can be changed for each function (see option `funspec`).

funspec the property file in which are given specific parameters of analysis for each function, see [FunctionOptions], page 6.

depth (option for Fixpoint) depth of recursive iterations (default 2, may only be more).

guided (option for Fixpoint) if true, guided analysis of Gopand and Reps (default: false).

wdelay (option for Fixpoint) specifies usage of widening delay steps (default: 1).

wdesc specifies usage of widening number of descending steps (default: 2).

main entry point function for the analysis (default: main).

Options specific for functions

The format of this file is a sequence of line, each line having the form:

```
function_name: n s p
```

where:

- `n` gives `maxanon` parameter (positive integer),
- `s` gives `maxsegm` parameter (strictly positive integer),
- `p` gives patterns to be used in the UCONS domain; it is an integer value obtained from the binary or of the following values:
  - 1 for pattern *forall y in n* (default value)
  - 2 for pattern *forall y1 in n1, y2 in n2. y1==y2*
  - 4 for pattern *forall y1,y2 in n1. y1<=y2*

6 Outputs

CELIA outputs the following files:

`pan-nm.c` contains the C code normalized by Frama-C where each statement has a unique identifier (C comment containing `sid`). This file contains the full coded needed by the analysis.

`pan.eq` contains the inter-procedural control flow graph (ICFG) considered by the analysis. Also, it provides (section `penv`) the encoding of each function variable into a domain variable.

`pan.abs` contains the result of the analysis, i.e., for each statement the number of the file `.shp` containing the invariant computed by **CELIA**.

`f_XXX.shp`
shape files containing the invariants computed by **CELIA**. To obtain a dot representation of this file please use the program `com/shp2dot.sh`.

How to read shape files?

A shape file `f_XXX.shp` is a **list** of tuples built from

- **a shape graph** (see CAV'10 publication) which is an abstraction of the heap:
  - the nodes of the graph represent heap sub-lists which are not pointed by pointer program variables; nods are named by `n0` (NULL), `n1`, `n2`, ... and are labelled by the pointer programs variables which points to the head of the sub-list, and
  - the arcs represent reachability relations (via `next` field) between sub-lists.

  For example, the shape graph in the invariant intlist-lib-add.c:sid:6 represents a heap with three sub-lists (nodes `n1, n2, n3`) such that the sub-list stating in `n1` is labeled by the program variables `x2` and `x5` and reaches the sub-list starting in node `n2`; the sub-list starting in `n3` is disjoint from the other sub-lists.
- **a list of constraints** for each data word abstract domain (in LSUM, MSET, UCONS) used in the analysis.

For the domains LSUM and MSET, the constraints attached to the graph are conjunctions of linear constraints involving the following terms:

`xi` represents the value stored in a scalar program variable (see file `pan.eq` to obtain the variable name),

`data(n)` represents the value stored in the head of the sub-list represented by the node `n`,

`len(n)` represents the number of nodes in the sub-list represented by the node `n`,

`sum(n)` represents the sum of values stored in the tail of the sub-list represented by the node `n`,

`mshd(n)` represents the singleton mouldiest containing only the value stored in the first element (head) of the sub-list represented by node `n`,

`mstl(n)` represents the multiset containing the values stored in the tail of the sub-list represented by node `n`,

For example, the constraint

-mstl(n2)+mstl(n3)=0

in the multi-set constraints in the first shape graph of intlist-lib-add.c:sid:6 says that the multisets of tails of sub-lists represented by nodes `n2` and `n3` are equal.

For the domain UCONS, the constraints attached to the graph are conjunction structured as follows:

- the **econs** block are linear constraints on the `data(n)`, `len(n)` and integer program variables `x`,
- the following blocks represent universal formulas in the form, e.g.,

  forall y in n. patter-constraint => data-constraint

  which express properties on the data in the sub-lists starting in the node *n*.

For example, the UCONS constraints in the file intlist-sort-insert:sid:40 contains two universally quantified constraints, both concerning the node `n1`:

- the first constraint expresses the property that for all cells in the sub-lists `n1`, represented by `y`, the data stored in these cells is greater or equal than the data stored in the head of `n1`, and
- the second constraint expresses the property that for all two cells `y1` and `y2` in the sub-lists `n1` such that `y1 <= y2`, the data stored in these cells is ordered.

7 References

- [Apron] Apron project, http://apron.cri.ensmp.fr/library/
- [Cousot&Cousot,79] P. Cousot and R. Cousot, Systematic Design of Program Analysis Frameworks, POPL'79
- [Fixpoint] B. Jeannet, http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/i
- [Frama-C] http://www.frama-c.com/