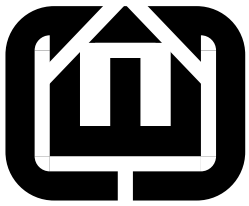
CEC GRC dot format printer



Stephen A. Edwards, Jia Zeng, Cristian Soviani
Columbia University
sedwards@cs.columbia.edu

Contents

1 Overview

2 Node printers

2.1 Control-flow graph nodes

The basic operation for a control-flow graph node is to create a new graph node, use the visitor to label it, draw its data dependencies, and draw arcs and recurse on its successors. Since it may be a DAG, this is done as a depth-first search with the `reached` set indicating which nodes have been visited.

1 ⟨*declarations* 1⟩≡ (8) 3▷

```
void visit_cfg(GRCNode *);
```

2 ⟨*definitions* 2⟩≡ (9) 4▷

```
void GRCDP::visit_cfg(GRCNode *n) {
  assert(n);

  if (reached.find(n) == reached.end()) {
    reached.insert(n);

    // Print a definition for the node
    assert(cfgnum.find(n) != cfgnum.end());

    mynum = cfgnum[n]; // used by most visitors
    o << 'n' << mynum << ' ';
    n->welcome(*this);

    // Draw data dependencies

    for (vector<GRCNode *>::const_iterator k = n->dataPredecessors.begin() ;
         k != n->dataPredecessors.end() ; k++) {
      assert(cfgnum.find(*k) != cfgnum.end());
      o << 'n' << cfgnum[*k] << " -> n" << cfgnum[n]
        << " [color=red constraint=false]\n";
    }

    // Draw control dependencies

    for ( vector<GRCNode*>::iterator j = n->successors.begin() ;
          j != n->successors.end() ; j++ ) {
      if (*j) {
        o << 'n' << cfgnum[n] << " -> n" << cfgnum[*j];
        if ( n->successors.size() > 1)
          o << " [label=\"" << j - n->successors.begin() << "\"]";
        o << '\n';
      } else {
        o << 'n'<< cfgnum[n] << " -> n" << nextnum << "[label=\""
          << j-n->successors.begin()<<"\"]"<<'\n';
        o << 'n' << nextnum++ << " [shape=octagon style=filled color=black]" << '\n';
      }
    }

    // Visit control successors and predecessors

    for ( vector<GRCNode*>::iterator j = n->successors.begin() ;
          j != n->successors.end() ; j++ )
      if (*j) visit_cfg(*j);

    for ( vector<GRCNode*>::iterator j = n->predecessors.begin() ;
      j != n->predecessors.end() ; j++ ) visit_cfg(*j);

    // Visit data successors and predecessors
```

```
    for ( vector<GRCNode*>::iterator j = n->dataSuccessors.begin() ;
          j != n->dataSuccessors.end() ; j++ ) visit_cfg(*j);

    for ( vector<GRCNode*>::iterator j = n->dataPredecessors.begin() ;
      j != n->dataPredecessors.end() ; j++ ) visit_cfg(*j);
  }
}
```

3 ⟨*declarations* 1⟩+≡ (8) ◁1 5▷

```
Status visit(Switch &);
Status visit(Test &);
Status visit(Terminate &);
Status visit(Sync &);
Status visit(Fork &);
Status visit(Action &);
Status visit(Enter &);
Status visit(STSuspend &);
Status visit(EnterGRC &);
Status visit(ExitGRC &);
Status visit(Nop &);
Status visit(DefineSignal &);
```

4 ⟨*definitions* 2⟩+≡ (9) ◁2 6a▷

```
Status GRCDP::visit(Switch &s) {
  o << "[label=\"" << mynum << " switch " << stnum[s.st]
    << "\" shape=diamond color=pink style=filled]\n";
  drawSTlink(&s,s.st);
  return Status();
}

Status GRCDP::visit(Test &s) {
  o << "[label=\"" << mynum << " test ";
  s.predicate->welcome(ep);
  o << "\" shape=diamond]\n";
  return Status();
}

Status GRCDP::visit(STSuspend &s){
  o << "[label=\"" << mynum << " Suspend "<<stnum[s.st];
  o << "\" shape=egg]\n";
  return Status();
}

Status GRCDP::visit(Terminate &s) {
  o << "[label=\"" << mynum << ' ' << s.index << '@' << s.code
    << "\" shape=octagon color=red style=filled "
       "fontcolor=white fontname=\"Times-Bold\"]\n";
  return Status();
}

Status GRCDP::visit(Sync &s) {
  o << "[label=\"" << mynum << " sync";
  if (s.st) o << " " << stnum[s.st];
  o<<"\" shape=invtriangle]\n";
  return Status();
}

Status GRCDP::visit(Fork &s) {
  o << "[label=\"" << mynum << " fork\" shape=triangle]\n";
  return Status();
}

Status GRCDP::visit(Action &s) {
  o << "[label=\"" << mynum << " action ";
  s.body->welcome(ep);
  o << "\" shape=box]\n";
  return Status();
}

Status GRCDP::visit(Enter &s) {
  o << "[label=\"" << mynum << " enter "<<stnum[s.st];
  o << "\" shape=house color=palegreen1 style=filled]\n";
```

```
  // drawSTlink(&s, s.st);
  return Status();
}

Status GRCDP::visit(EnterGRC &s){
   o << "[label=\"" << mynum << " EnterGRC\"]\n";
   return Status();
}

Status GRCDP::visit(ExitGRC &s){
  o << "[label=\"" << mynum << " ExitGRC\"]\n";
  return Status();
}

Status GRCDP::visit(Nop &s){
  o << "[label=\"" << mynum << " ";
  if (s.isflowin()) o << "*"; else
    if (s.isshortcut()) o << "#";
    else o << "\\n" << s.code;
  o << "\" shape=circle]\n";
  return Status();
}

Status GRCDP::visit(DefineSignal &s){
  o << "[label=\"" << mynum << " DefS\\n";
  o<< s.signal->name;
  o<< "\"]\n";
  return Status();
}
```

2.2 Selection Tree Nodes

The basic operation for a selection tree node is to print the node and its label using the visitor, then recurse on its children. This is a simple recursive walk because the selection tree is a tree.

5 ⟨*declarations* 1⟩+≡ (8) ◁3 6b▷

```
void visit_st(STNode *);
```

6a ⟨*definitions* 2⟩+≡ (9) ◁4 7▷

```
void GRCDP::visit_st(STNode *n) {
  assert(n);

  mynum = stnum[n];
  o << 'n' << mynum << ' ';
  n->welcome(*this);

  // Visit children

  for ( vector<STNode*>::const_iterator i = n->children.begin() ;
        i != n->children.end() ; i++ ) if(*i) {
    visit_st(*i);
    o << 'n' << stnum[n] << " -> n" << stnum[*i]
      << " [label=\"" << (i - n->children.begin()) << "\"]\n";
  } else {
    o << 'n'<< stnum[n] << " -> n" << nextnum << "[label=\""
      << i - n->children.begin() << "\"]"<<'\n';
    o << 'n' << nextnum++ << " [shape=octagon style=filled color=black]" << '\n';
  }
}
```

6b ⟨*declarations* 1⟩+≡ (8) ◁5

```
Status visit(STexcl &);
Status visit(STref &);
Status visit(STpar &);
Status visit(STleaf &);
```

7 ⟨*definitions 2*⟩+≡ (9) ◁6a

```
Status GRCDP::visit(STexcl &s) {
  o << "[label=\"" << mynum << "\" shape=diamond color=pink style=filled]\n";
  return Status();
}

Status GRCDP::visit(STref &s) {
  o << "[label=\"" << mynum << " ";
  if(s.isabort()) o << "A";
  if(s.issuspend()) o << "S";
  o << "\" ]\n";
  return Status();
}

Status GRCDP::visit(STpar &s) {
  o << "[label=\"" << mynum << "\" shape=triangle]\n";
  return Status();
}

Status GRCDP::visit(STleaf &s) {
  o << "[label=\"" << mynum << " ";
  if(s.isfinal()) o << "*";
  o << "\" shape=box]\n";
  return Status();
}
```

3 Topmost files

8 ⟨*GRCPrinter.hpp* 8⟩≡

```
#ifndef _GRC_PRINTER_HPP
#  define _GRC_PRINTER_HPP
#  include "AST.hpp"
#  include "EsterelPrinter.hpp"
#  include <iostream>
#  include <map>
#  include <set>

namespace AST {
  using std::map;
  using std::set;

  typedef map<GRCNode *, int> CFGmap;
  typedef map<STNode *, int> STmap;
  void GRCDot(std::ostream &, GRCgraph *, Module *);
  void GRC27Dot(std::ostream &, GRCgraph *, Module *,
        CFGmap &, STmap &, int);

  class GRCDP : public Visitor {
    std::ostream &o;
    CFGmap &cfgnum; // Node numbers for control-flow graph
    STmap &stnum; // Node numbers for selection tree
    set<GRCNode *> reached; // Used during DFS of CFG

    int nextnum;
    int mynum;

    bool drawstlink;

    EsterelPrinter ep;
  public:
    GRCDP(std::ostream &oo, CFGmap &cm, STmap &sm, int nextnum) :
      o(oo), cfgnum(cm), stnum(sm), nextnum(nextnum),
      drawstlink(false), ep(oo) {}

    virtual ~GRCDP() {}

    ⟨declarations 1⟩

    void drawSTlink(GRCNode *, STNode *);
  };
};
#endif
```

9 ⟨*GRCPrinter.cpp* 9⟩≡

```
#include "GRCPrinter.hpp"
#include <cassert>

namespace AST {

  ⟨definitions 2⟩

  void GRCDot(std::ostream &o, GRCgraph *g, Module *m)
  {
    assert(g);
    assert(m);
    assert(m->symbol);

    CFGmap cfgmap;
    STmap stmap;

    int mxnode = g->enumerate(cfgmap, stmap);

    GRCDP visitor(o, cfgmap, stmap, mxnode+1);

    o << "digraph " << m->symbol->name << " {" << std::endl;
    o << "size=\"7.5,10\"\n";

    visitor.visit_st(g->selection_tree);
    visitor.visit_cfg(g->control_flow_graph);

    o << "}" << std::endl;
  }

  void GRC27Dot(std::ostream &o, GRCgraph *g, Module *m,
        CFGmap &cfgmap, STmap &stmap, int mxnode)
  {
    assert(g);
    assert(m);
    assert(m->symbol);

    GRCDP visitor(o, cfgmap, stmap, mxnode+1);

    o << "digraph " << m->symbol->name << " {" << std::endl;
    o << "size=\"7.5,10\"\n";

    visitor.visit_st(g->selection_tree);
    visitor.visit_cfg(g->control_flow_graph);

    o << "}" << std::endl;
  }
```

```
void GRCDP::drawSTlink(GRCNode *g, STNode *s)
{
  o << "{ rank=same; n" << cfgnum[g] << "; n" << stnum[s] << " }\n";
  if (!drawstlink) return;

  assert( stnum.find(s) != stnum.end() );

  o << 'n' << cfgnum[g] << " -> n" << stnum[s];
  o << "[color=blue constraint=false]";
  o << '\n';
}
}
```

10 ⟨*cec-grcdot.cpp* 10⟩≡

```
#include "IR.hpp"
#include "AST.hpp"
#include "GRCPrinter.hpp"
#include <iostream>
#include <stdlib.h>

int main()
{
  try {
    IR::XMListream r(std::cin);
    IR::Node *n;
    r >> n;

    AST::Modules *mods = dynamic_cast<AST::Modules*>(n);
    if (!mods) throw IR::Error("Root node is not a Modules object");

    for ( std::vector<AST::Module*>::iterator i = mods->modules.begin() ;
          i != mods->modules.end() ; i++ ) {
      assert(*i);

      AST::GRCgraph *g = dynamic_cast<AST::GRCgraph*>((*i)->body);
      if (!g) throw IR::Error("Module is not in GRC format");

      AST::GRCDot(std::cout, g, *i);
    }
  } catch (IR::Error &e) {
    std::cerr << e.s << std::endl;
    exit(-1);
  }
  return 0;
}
```