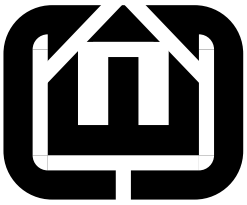
CEC GRC-to-PDG Converter



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

Abstract

This converts the control-flow graph portion of the GRC graph into a program dependence graph using the algorithm described by Cytron et al. in their 1991 TOPLAS article.

Contents

1 Utilities

1.1 contains

Return true if the set contains the object.

2a ⟨*utilities* 2a⟩≡ (30) 2b▷
```
template <class T> bool contains(set<T> &s, T o) {
  return s.find(o) != s.end();
}
```

2b ⟨*utilities* 2a⟩+≡ (30) ◁2a
```
template <class T, class U> bool contains(map<T, U> &m, T o) {
  return m.find(o) != m.end();
}
```

2 The STDPS class

3 ⟨*stdps class* 3⟩≡ (30)

```
class STDPS {
  EnterGRC *entergrc;
  set<GRCNode *> visited;
  map<GRCNode *, set<GRCNode *> > enter_nodes; //enter nodes under, for node with multi-par only

  public:

  STDPS(EnterGRC *entergrc): entergrc(entergrc) {}
  ~STDPS() {}

  Status execute() {
        visited.clear();
        variable_dfs(entergrc);
        return Status();
  }

  private:

  set<GRCNode *> variable_dfs(GRCNode *n)
  {
    set<GRCNode *> RET;

    if (!n)
      return RET;
    if (visited.count(n) >0){
      assert(enter_nodes.count(n)>0);
      return enter_nodes[n];
    }

    visited.insert(n);

    for (vector<GRCNode *>::iterator i = n->successors.begin();
         i != n->successors.end(); i++){
      set<GRCNode *> ch_set = variable_dfs(*i);
      if (ch_set.size()>0){// if find some enters under
        RET.insert(ch_set.begin(), ch_set.end());
      }
      if (dynamic_cast<Enter *>(n)){// if it's an enter, decide whether to add dps
        for (set<GRCNode *>::iterator j = ch_set.begin();
             j != ch_set.end(); j++){
          if (same_sstp(n,*j)){
            **j << n;
          }
        }
        RET.insert(n);
      }
      if ((dynamic_cast<STSuspend *>(n))
```

```
          ||(dynamic_cast<Switch *>(n))) {//if it's suspend or switch, decide whether to add dp
        for (set<GRCNode *>::iterator j = ch_set.begin();
             j != ch_set.end(); j++){
//        if (same_stp(n,*j)){
          if(st_ancestor(n,*j)){
            **j << n;
          }
        }
      }
    }

    if (n->predecessors.size()>1){
      enter_nodes[n].insert(RET.begin(), RET.end());
    }

    return RET;
  }

  //test if two nodes have the same st pointer, n1-suspend, n2-enter
  bool same_stp(GRCNode *n1, GRCNode *n2)
  {
    STSuspend *s;
    Enter *e;

    s = dynamic_cast<STSuspend *>(n1);
    e = dynamic_cast<Enter *>(n2);

    if (s->st == e->st)
      return true;

    return false;
  }

  bool same_sstp(GRCNode *n1, GRCNode *n2)
  {
    Enter *e1, *e2;

    e1 = dynamic_cast<Enter *>(n1); assert(e1);
    e2 = dynamic_cast<Enter *>(n2); assert(e2);

    //if they point to the same stnode, not need to add constrain btw them
    if(e1->st == e2->st)
      return false;

    if (e1->st->parent == e2->st->parent) {
      if (dynamic_cast<STexcl *>(e1->st->parent))
        return true;
    }
    return false;
  }
```

```
  bool st_ancestor(GRCNode *p, GRCNode *c)
  {

    GRCSTNode *pp = dynamic_cast<GRCSTNode *>(p); assert(p);
    GRCSTNode *cc = dynamic_cast<GRCSTNode *>(c); assert(cc);

    STNode *stp = pp ->st;
    STNode *stc = cc ->st;

    while(stc != NULL){
        if(stp==stc) return true;
        stc = stc->parent;
    }
    return false;
  }

};
```

3 Signal Dependency Calculator Class

This class removes & re-computes dependencies between signal emissions and tests.

⟨*dependency class* 5⟩≡ (30)

```
class Dependencies : public Visitor {
protected:
  set<GRCNode *> visited;
  GRCNode *current;
  map<GRCNode *, bool> par_label;

  struct SignalNodes {
    set<GRCNode *> writers;
    set<GRCNode *> readers;
  };

  map<SignalSymbol *, SignalNodes> dependencies;

  ⟨dependency methods 8b⟩
  void mark_par(GRCNode* n);
  bool have_comm_pp_gen(GRCNode* n, GRCNode* m);
  bool have_comm_pp(GRCNode* n, GRCNode* m);

public:

  Dependencies() {}
  virtual ~Dependencies() {}
  void compute(GRCNode *);
};
```

6a ⟨*dependency method definitions* 6a⟩≡ (30) 6b▷

```
void Dependencies::compute(GRCNode *root)
{
  assert(root);

  Dependencies depper;

  depper.dfs(root);

  for ( map<SignalSymbol *, SignalNodes>::const_iterator i =
          depper.dependencies.begin() ; i != depper.dependencies.end() ;
        i++ ) {
    const SignalNodes &sn = (*i).second;
    if (!sn.writers.empty() && !sn.readers.empty()) {
      for ( set<GRCNode*>::const_iterator j = sn.writers.begin() ;
            j != sn.writers.end() ; j++ ){
        visited.clear();
        par_label.clear();
        mark_par(*j);
        for ( set<GRCNode*>::const_iterator k = sn.readers.begin() ;
              k != sn.readers.end() ; k++ ){
          visited.clear();
          if (have_comm_pp_gen((*k),(*j)))
            **k << *j;
        }
      }
    }
  }
}
```

6b ⟨*dependency method definitions* 6a⟩+≡ (30) ◁6a 7a▷

```
void Dependencies::mark_par(GRCNode* n)
  {
    int sz, i;

    if (visited.count(n) > 0)
      return;

    sz = n->predecessors.size();
    for (i = 0; i < sz; i++){
      if ( par_label[n->predecessors[i]] == false){
        par_label[n->predecessors[i]] = true;
        mark_par(n->predecessors[i]);
      }
    }

    //also mark n itself as its parent
    par_label[n] = true;
    visited.insert(n);
  }
```

7a ⟨*dependency method definitions* 6a⟩+≡ (30) ◁6b 7b▷

```
//test if two nodes n & m have parallel first-comm-parent
//where m's parents have been labeled
bool Dependencies::have_comm_pp_gen(GRCNode* n, GRCNode* m)
{
  if (par_label[n])
    return true;

  return have_comm_pp(n,m);
}
```

7b ⟨*dependency method definitions* 6a⟩+≡ (30) ◁7a 8a▷

```
bool Dependencies::have_comm_pp(GRCNode* n, GRCNode* m)
{

  assert(n);
  if (visited.count(n)>0)
    return false;

  visited.insert(n);

  if (par_label[n]){//found a first_comm_parent
    if ((dynamic_cast<Fork *>(n)) //is it a parallel node?
        || (n == m))//or, is it the emitter corsp?
      return true;
  }
  else {
    for (vector<GRCNode *>::iterator it =n->predecessors.begin();
         it != n->predecessors.end(); it++){
      if (have_comm_pp(*it, m))
        return true;
    }
  }

  return false;

}
```

3.1 DFS

This is the core dispatch procedure for the walker. It verifies it has not already visited the given node, visits it, then calls itself recursively on its successors.

8a ⟨*dependency method definitions* 6a⟩+≡ (30) ◁7b 8c▷

```
void Dependencies::dfs(GRCNode *n)
{
  if (!n || visited.find(n) != visited.end() ) return;

  visited.insert(n);

  current = n;
  n->welcome(*this);

  for (vector<GRCNode*>::const_iterator i = n->successors.begin() ;
       i < n->successors.end() ; i++ ) dfs(*i);
}
```

8b ⟨*dependency methods* 8b⟩≡ (5) 8d▷

```
void dfs(GRCNode *);
```

3.2 Action

An action may be an emit or exit statement, which emit signals.

8c ⟨*dependency method definitions* 6a⟩+≡ (30) ◁8a 9a▷

```
Status Dependencies::visit(Action &act)
{
    Emit *emt = dynamic_cast<Emit *>(act.body);
    if (emt) {
      dependencies[emt->signal].writers.insert(current);
      current->dataSuccessors.clear();
    } else {
      // Traps are treated the same as signals
      Exit *ex = dynamic_cast<Exit *>(act.body);
      if (ex) {
        dependencies[ex->trap].writers.insert(current);
        current->dataSuccessors.clear();
      }
    }
    return Status();
}
```

8d ⟨*dependency methods* 8b⟩+≡ (5) ◁8b 9b▷

```
Status visit(Action &);
```

3.3 DefineSignal

The DefineSignal node is like an emit.

9a ⟨*dependency method definitions* 6a⟩+≡ (30) ◁8c

```
Status Dependencies::visit(DefineSignal &ds)
{
  assert(ds.signal);
  dependencies[ds.signal].writers.insert(current);
  current->dataSuccessors.clear();
  return Status();
}
```

9b ⟨*dependency methods* 8b⟩+≡ (5) ◁8d 9c▷

```
Status visit(DefineSignal &);
```

3.4 Test

This descends down its predicate, possibly adding signal testers

9c ⟨*dependency methods* 8b⟩+≡ (5) ◁9b 10a▷

```
Status visit(Test &t) {
  t.predicate->welcome(*this); return Status();
}
```

3.5 Expressions

10a ⟨*dependency methods* 8b⟩+≡ (5) ◁9c 10b▷

```
Status visit(LoadSignalExpression &e) {
  dependencies[e.signal].readers.insert(current);
  current->dataPredecessors.clear();
  return Status();
}

Status visit(LoadSignalValueExpression &e) {
  dependencies[e.signal].readers.insert(current);
  current->dataPredecessors.clear();
  return Status();
}

Status visit(BinaryOp &e) {
  e.source1->welcome(*this);
  e.source2->welcome(*this);
  return Status();
}

Status visit(UnaryOp &e) {
  e.source->welcome(*this);
  return Status();
}

Status visit(CheckCounter &e) {
  e.predicate->welcome(*this);
  return Status();
}

Status visit(Delay &d) {
  d.predicate->welcome(*this);
  return Status();
}
```

3.5.1 Vacuous Expression Nodes

10b ⟨*dependency methods* 8b⟩+≡ (5) ◁10a 11▷

```
Status visit(Literal &) { return Status(); }
Status visit(LoadVariableExpression &) { return Status(); }
Status visit(FunctionCall &) { return Status(); }
```

3.6 Trivial visitors

These nodes have no dependency implications and hence do nothing when visited.

11 ⟨*dependency methods* 8b⟩+≡ (5) ◁10b

```
Status visit(EnterGRC &) { return Status(); }
Status visit(ExitGRC &) { return Status(); }
Status visit(Nop &) { return Status(); }
Status visit(Switch &) { return Status(); }
Status visit(STSuspend &) { return Status(); }
Status visit(Fork &) { return Status(); }
Status visit(Terminate &) { return Status(); }
Status visit(Enter &) { return Status(); }
Status visit(Sync &s) { return Status(); }
```

4 The GRCPDG class

12 ⟨*grcpdg class* 12⟩≡ (30)

```
class GRC2PDG {

  CFGmap &dotrefmap;

  map<GRCNode *, int> nodenum; // RDFS numbering (index) of each node
  vector<GRCNode*> vert; // nodes in RDFS order

  vector<int> parent; // index of the RDFS spanning tree parent of
                      // each node

  vector<int> ancestor;
  vector<int> semi; // Semi-dominator of each node

  vector<int> idom; // The immediate dominator of each node
  vector<set<int> > ichild; // The nodes immediately dominated by each node

  vector<set<int> > df; // Dominance frontier for each node
  vector<set<int> > cd; // Nodes control dependent on each node

  map<int, vector<int> > succmap;
  map<int, vector<int> > predmap;
  map<int, bool> reachability;
  set<int> visited;
  int N; // Total number of nodes
  int nullnum;

  EnterGRC *enternode;
  ExitGRC *exitnode;

public:
  ⟨method declarations 13⟩
};
```

5 The Constructor

This uses the algorithm described in Cytron et al. [1] to calculate control dependence relationship and transform the GRC concurrent control-flow graph into a program dependence graph.

13 ⟨*method declarations* 13⟩≡ (12) 14▷

```
GRC2PDG(GRCNode *top, CFGmap &dotrefmap) : dotrefmap(dotrefmap)
{
  assert(top);
  enternode = dynamic_cast<EnterGRC *>(top);
  assert(enternode);
  exitnode = dynamic_cast<ExitGRC *>(enternode->successors[0]);
  assert(exitnode);

  N = 0; // Used to number the nodes during reverse DFS
  reverse_dfs(NULL, exitnode);

  build_dominance_tree();

  df.resize(N);
  compute_dominance_frontier(nodenum[exitnode]);
  //print_df();

  cd.resize(N);
  compute_control_dependence();
  //print_CD();

  //cerr<<"start building pdg\n";
  build_pdg();
  //print_PDG();

  visited.clear();
  opt_dfs(enternode);
  //cerr<<"finished\n";
}
```

6 Depth-first search on the reverse graph

Depth-first search on the reverse graph. Number all the nodes.

14 ⟨*method declarations* 13⟩+≡ (12) ◁13 15▷

```
void reverse_dfs(GRCNode *p, GRCNode *n)
{
  if (!n || contains(nodenum,n) ) return;

  nodenum[n] = N;
  vert.push_back(n);
  parent.push_back(p ? nodenum[p] : -1);
  N++;

  if ( n != enternode )
    for (vector<GRCNode*>::iterator i = n->predecessors.begin() ;
         i != n->predecessors.end() ; i++)
      reverse_dfs(n, *i);
}
```

7 Build Dominance Tree

Build the dominance tree for the reverse graph.

15 ⟨*method declarations* 13⟩+≡ (12) ◁14 16▷

```
void build_dominance_tree()
{
  ancestor.resize(N,-1);
  semi.resize(N,-1);
  idom.resize(N,-1);
  vector<int> samedom;
  samedom.resize(N,-1);

  vector<set<int> > bucket;
  bucket.resize(N);

  ichild.resize(N);

  for ( int n = N-1 ; n > 0 ; n-- ) {

    assert(dotrefmap.count(vert[n])>0); // FIXME: ??

    int p = parent[n];
    int s = p;

    for( vector<GRCNode*>::iterator iv = vert[n]->successors.begin() ;
         iv != vert[n]->successors.end() ; iv++ ) {
      if (*iv) {
        int v = nodenum[*iv];
        int s1 = (v <= n) ? v : semi[ancestor_lowest_semi(v)];
        if ( s1 < s ) s = s1;
      }
    }

    semi[n] = s;
    if ( !contains(bucket[s], n) ) bucket[s].insert(n);
    ancestor[n] = p;

    for( set<int>::iterator iv = bucket[p].begin() ;
         iv != bucket[p].end() ; iv++ ) {
      int v = *iv;
      int y = ancestor_lowest_semi(v);
      if (semi[y] == semi[v]) idom[v] = p;
      else samedom[v] = y;
    }

    bucket[p].clear();
  }

  for (int n = 1 ; n < N ; n++ )
```

```
      if ( samedom[n] != -1 )
        idom[n] = idom[samedom[n]];

    for (int n = 1 ; n < N ; n++)
      if ( idom[n] != -1 )
        ichild[idom[n]].insert(n);
  }
```

7.1 ancestor lowest semi

16 ⟨*method declarations* 13⟩+≡ (12) ◁15 17a▷

```
  int ancestor_lowest_semi(int v)
  {
    int u = v;
    while ( ancestor[v] != -1 ) {
      if ( semi[v] < semi[u] ) u = v;
      v = ancestor[v];
    }

    return u;
  }
```

8 Compute Dominance Frontier

This is Fig. 10 from Cytron et al. [1]. It builds the `df` sets.

17a ⟨*method declarations* 13⟩+≡ (12) ◁16 17b▷

```
void compute_dominance_frontier(int n)
{
  for(set<int>::iterator iz = ichild[n].begin(); iz != ichild[n].end() ; iz++)
    compute_dominance_frontier(*iz);

  int enternodeidx = nodenum[enternode];

  if ( n != enternodeidx ) {
    for (vector<GRCNode*>::iterator i = vert[n]->predecessors.begin() ;
         i != vert[n]->predecessors.end(); i++ ) {
      assert(contains(nodenum, *i));
      int y = nodenum[*i];
      if ( idom[y] != n && !contains(df[n], y) ) {
        assert( contains(dotrefmap, *i) );
        df[n].insert(y);
      }
    }
  }

  for( set<int>::iterator iz = ichild[n].begin() ;
       iz != ichild[n].end() ; iz++) {
    int z = *iz;
    for( set<int>::iterator iy = df[z].begin() ; iy != df[z].end() ; iy++ ) {
      int y = *iy;
      if(idom[y] != n && !contains(df[n], y) ) df[n].insert(y);
    }
  }
}
```

9 Compute control dependence

This is Fig. 11 from Cytron et al. [1]. It builds the `cd` sets.

17b ⟨*method declarations* 13⟩+≡ (12) ◁17a 18▷

```
void compute_control_dependence()
{
  for( int y = 0 ; y < N ; y++ )
    for(set<int>::iterator ix=df[y].begin() ; ix!=df[y].end() ; ix++) {
      int x = *ix;
      if ( !contains(cd[x], y) ) cd[x].insert(y);
    }

  //a trick - force EnterGRC's child[1] to be CD of EnterGRC
  cd[nodenum[enternode]].insert(nodenum[enternode->successors[1]]);
}
```

10 Build PDG

18 ⟨*method declarations* 13⟩+≡ (12) ◁17b 21a▷

```
void build_pdg()
{
  copy_conn();
  remove_conn();

  int counter = N;

  //for each node i
  for (int i = 0; i < N; i++ ) {
    //cerr<<"for node "<<dotrefmap[vert[i]]<<"\n";
    GRCNode *n = vert[i];

    assert(dotrefmap.count(vert[i])>0);

    if ( n == exitnode ) {

      // n is ExitGRC; ignore it

    } else if ((dynamic_cast<Fork *>(n))
               ||
               (n == enternode && (cd[i].size() < 2)) ) {

      // A parallel node or EnterGRC with a single child:
      // Make each CD member a child, disregard its original child number
      // If n is EnterGRC with 1 child, take it as a parallel node
      // **** something may happen, if one can exit in two branches

      for( set<int>::iterator iy = cd[i].begin() ; iy != cd[i].end() ; iy++) {
        GRCNode *y = vert[*iy];
        if ( y != exitnode && ((*iy) != i) ) {
          n->successors.push_back(y);
          y->predecessors.push_back(n);
        }
      }

    } else if ( n == enternode ) {

      // EnterGRC with more than 1 child

      Fork *reg = new Fork();
      for (set<int>::iterator iy = cd[i].begin() ; iy != cd[i].end() ; iy++) {
        GRCNode *y = vert[*iy];
        if (y != exitnode && ((*iy) != i)) {
          reg->successors.push_back(y);
          y->predecessors.push_back(reg);
        }
      }
```

```
  //new region node
  nodenum[reg] = counter++;
  vert.push_back(reg);
  n->successors.push_back(reg);
  reg->predecessors.push_back(n);

} else {

  // else, for each successor ic of i, make a region node reg

  //cerr<<" build regions for ic succ:\n";
  for(vector<int>::iterator ic = succmap[i].begin();
      ic != succmap[i].end(); ic++) {

    // NULL node
    if (*ic == -1){
      n->successors.push_back(NULL);
      //cerr<<"  null succ\n";
      continue;
    }
    if (dynamic_cast<ExitGRC *>(vert[*ic])){
      //cerr<<"  exit grc succ\n";
      continue;
    }

    Fork *reg = new Fork();

    //cerr<<"  real succ IC "<<dotrefmap[vert[*ic]]<<"\n";

    //for each node iy in CD set of node i,
    // check if iy is reachable from brunch ic
    for(set<int>::iterator iy=cd[i].begin(); iy!=cd[i].end(); iy++){
      if ((dynamic_cast<ExitGRC *>(vert[*iy])) || ((*iy) == i))
        continue;

      //cerr<<" IY "<<dotrefmap[vert[*iy]]<<"\n";

      reachability.clear();
      //cerr<<"testing reachablility...";
      if (reachable((*ic), (*iy))) {
        // if yes, add it as a child of the brunch region node reg
        reg->successors.push_back(vert[*iy]);
        vert[*iy]->predecessors.push_back(reg);
      }
      //cerr<<" finshed\n";
    }

    //place the region node reg as n's child
    // if reg only has one child, add this child directly
```

```
        switch (reg->successors.size()){
        case 0:
          //if n is sync|switch|test,instead of reg, place a null node there
          if ((dynamic_cast<Switch *>(n)) || (dynamic_cast<Sync *>(n))
              || (dynamic_cast<Test *>(n)))
            n->successors.push_back(NULL);
          break;
        case 1:
          n->successors.push_back(reg->successors[0]);
          reg->successors[0]->predecessors.pop_back();
          reg->successors[0]->predecessors.push_back(n);
          reg->successors.clear();
          break;
        default:
          //cerr<<"add new reg node: "<<sz<<"\n";
          nodenum[reg] = counter++;
          vert.push_back(reg);
          n->successors.push_back(reg);
          reg->predecessors.push_back(n);
          break;
        }
      }
      //cerr<<"N"<<dotrefmap[vert[i]]<<" is finished\n";
    }
  }
}
```

11 copy conn

21a ⟨*method declarations* 13⟩+≡ (12) ◁18 21b▷

```
void copy_conn()
{
  nullnum = 0;

  for (int i = 0; i < N; i++){
    for (vector<GRCNode *>::iterator ic = vert[i]->successors.begin();
         ic != vert[i]->successors.end(); ic++){
      if (*ic)
        succmap[i].push_back(nodenum[*ic]);
      else{
        succmap[i].push_back(-1);
        nullnum++;
      }
    }
    for (vector<GRCNode *>::iterator ip = vert[i]->predecessors.begin();
         ip != vert[i]->predecessors.end(); ip++){
      predmap[i].push_back(nodenum[*ip]);
    }
  }
}
```

12 remove conn

21b ⟨*method declarations* 13⟩+≡ (12) ◁21a 22▷

```
void remove_conn()
{
  for (int i = 0; i < N; i++){
    vert[i]->successors.clear();
    if (vert[i] != enternode)
      vert[i]->predecessors.clear();
  }
}
```

13 reachable

22 ⟨*method declarations* 13⟩+≡ (12) ◁21b 24▷

```
bool reachable(int from, int to)
{
  //cerr<<" dfs "<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"\n";

  if (reachability.count(from) > 0)
    return reachability[from];

  if (from == 0){
    //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  NO2\n";
    reachability[from] = false;
    return false;
  }

  if (from == -1){
    //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES2\n";
    reachability[from] = true;
    return true;
  }

  if (to == from){
    //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES1\n";
    reachability[from] = true;
    return true;
  }

  assert(vert[from]);

  //for fork node, reachable from any one of the children is reable
  if (dynamic_cast<Fork *>(vert[from])){
    for (vector<int>::iterator ic = succmap[from].begin();
         ic != succmap[from].end(); ic++){
      if (reachable((*ic), to)){
        //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES3\n";
        reachability[from] = true;
        return true;
      }
    }
    //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  NO3\n";
    reachability[from] = false;
    return false;
  }

  //for else node, reachable means can be reached from all of the children
  for (vector<int>::iterator ic = succmap[from].begin();
       ic != succmap[from].end(); ic++){
    if (!reachable((*ic), to)){
      //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  NO4\n";
```

```
        reachability[from] = false;
        return false;
      }
    }
    //cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES4\n";
    reachability[from] = true;
    return true;
  }
```

14 opt dfs

24 ⟨*method declarations* 13⟩+≡ (12) ◁22 25▷

```
void opt_dfs(GRCNode *n)
{
  vector<GRCNode *>::iterator i,j,k;
  GRCNode *ch;
  bool pis_fork = false;

  if (!n)
    return;

  if (visited.count(nodenum[n]) > 0){
    return;
  }

  if (dynamic_cast<Fork *>(n)){
    pis_fork = true;
  }

  i = n->successors.begin();
  while ( i != n->successors.end()){

    if ((*i) == NULL){
      i++;
      continue;
    }

    opt_dfs(*i);

    // remove node with only null children
    if (all_child_null(*i)){
      //for sync, test, switch node, replace i with null child
      if ((dynamic_cast<Sync *>(n))
          ||(dynamic_cast<Switch *>(n)) || (dynamic_cast<Test *>(n))){
        rm_predecessor(*i, n);
        (*i) = NULL;
        i++;
      }
      //else remove child i  ::FIXME -- is it always true?
      else{
        if ((*i)->predecessors.size() == 1) //FIXME
          rm_datadps(*i);
        rm_predecessor(*i, n);
        i = n->successors.erase(i);
        //i--;
      }
      continue;
    }
```

```
    // & continuous fork nodes with the same control dependence
    if ((dynamic_cast<Fork *>(*i)) && (pis_fork)){
      //cerr<<dotrefmap[n]<<"'s child "<<dotrefmap[*i]<<" is also a fork\n";
      if((*i)->predecessors.size() == 1){
        //cerr<<" SAME control flow, merge them\n";
        ch = *i;
        rm_predecessor(*i, n);
        k = n->successors.erase(i);

        for (j = ch->successors.begin(); j != ch->successors.end(); j++){
          k = n->successors.insert(k, *j);
          replace_par(*j, ch, n);
          k++;
        }
        if (ch->predecessors.size() == 0)
          ch->successors.clear();
        i = k-1;
      }
      //else,it has other control dependences, do nothing currently
    }
    i++;
  }

  visited.insert(nodenum[n]);

}
```

15 rm predecessor

25 ⟨*method declarations* 13⟩+≡ (12) ◁24 26a▷

```
void rm_predecessor(GRCNode *n, GRCNode *par)
{
  vector<GRCNode *>::iterator i;

  assert(n);
  assert(par);
  for (i = n->predecessors.begin(); i != n->predecessors.end(); i++)
    if (*i == par){
      n->predecessors.erase(i);
      return;
    }
}
```

16 rm datadps

26a ⟨*method declarations* 13⟩+≡ (12) ◁25 26b▷

```
void rm_datadps(GRCNode *n)
{
  vector<GRCNode *>::iterator i,j;

  for (i = n->dataPredecessors.begin(); i != n->dataPredecessors.end(); i++){
    for (j = (*i)->dataSuccessors.begin();
         j != (*i)->dataSuccessors.end(); j++)
      if ((*j) == n){
        (*i)->dataSuccessors.erase(j);
        break;
      }
  }

  for (i = n->dataSuccessors.begin(); i != n->dataSuccessors.end(); i++){
    for (j = (*i)->dataPredecessors.begin();
         j != (*i)->dataPredecessors.end(); j++)
      if ((*j) == n){
        (*i)->dataPredecessors.erase(j);
        break;
      }
  }

  n->dataPredecessors.clear();
  n->dataSuccessors.clear();
}
```

17 all child null

26b ⟨*method declarations* 13⟩+≡ (12) ◁26a 27a▷

```
bool all_child_null(GRCNode *n)
{
  int sz;

  assert(n);
  sz = n->successors.size();

  if (sz == 0)
    return false;

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

  return true;
}
```

18 replace par

27a ⟨*method declarations* 13⟩+≡ (12) ◁26b

```
void replace_par(GRCNode *n, GRCNode *org_par, GRCNode *new_par)
{
  vector<GRCNode *>::iterator i;

  for (i = n->predecessors.begin(); i != n->predecessors.end(); i++)
    if ((*i) == org_par){
      //*i = new_par;
      i = n->predecessors.erase(i);
      n->predecessors.insert(i,new_par);
      return;
    }
}
```

19 Printing methods

27b ⟨*printing method declarations* 27b⟩≡ 27c▷

```
void print_df()
{
  int i;

  cerr<<"DF\n";
  for(i=0; i<N ;i++){
    cerr<<"Node"<<dotrefmap[vert[i]]<<": ";
    for(set<int>::iterator iy=df[i].begin(); iy!=df[i].end(); iy++){
      cerr<<dotrefmap[vert[(*iy)]]<<" ";
    }
    cerr<<"\n";
  }
  }
```

27c ⟨*printing method declarations* 27b⟩+≡ ◁27b 28a▷

```
void print_CD()
{
  int i;

  cerr<<"CD\n";
  for(i=0; i<N ;i++){
    cerr<<"Node"<<dotrefmap[vert[i]]<<": ";
    for(set<int>::iterator iy=cd[i].begin(); iy!=cd[i].end(); iy++)
      cerr<<dotrefmap[vert[*iy]]<<" ";
    cerr<<"\n";
  }
}
```

28a ⟨*printing method declarations* 27b⟩+≡ ◁27c 28b▷

```
void print_conn()
{
  cerr<<"Connectivity:\n";
  for(int i=0; i<N ;i++){
    cerr<<"Node"<<dotrefmap[vert[i]]<<": ";
    for(vector<int>::iterator iy=succmap[i].begin(); iy!=succmap[i].end(); iy++)
      if (*iy > -1)
        cerr<<dotrefmap[vert[*iy]]<<" ";
      else
        cerr<<"NULL ";
    cerr<<"\n";
  }
}
```

28b ⟨*printing method declarations* 27b⟩+≡ ◁28a

```
void print_PDG()
{
  cerr<<"PDG:\n";

  for (int i = 0; i < (int)(vert.size()); i++){

    if (!(vert[i]))
      continue;

    cerr<<"Node"<<dotrefmap[vert[i]]<<": ";
    for(vector<GRCNode *>::iterator iy=vert[i]->successors.begin();
        iy!=vert[i]->successors.end(); iy++)
      cerr<<dotrefmap[*iy]<<" ";
    cerr<<"\n";
  }
}
```

20 Main function

29 ⟨*main function* 29⟩≡ (30)

```
int main(int argc, char* argv[])
{
  IR::XMListream f(std::cin);
  IR::Node *n;
  f >> n;

  Modules *mods = dynamic_cast<AST::Modules*>(n);
  if (!mods) {
    std::cerr<<"Root node is not a module object\n";
    exit(-2);
  }

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

    GRCgraph *gf = dynamic_cast<GRCgraph*>((*i)->body);
    assert(gf);
    GRCNode *top = gf->control_flow_graph;

    CFGmap dotrefmap;
    STmap strefmap;

    EnterGRC *engrc = dynamic_cast<EnterGRC*>(top);
    assert(engrc);

    // compute the data dependencies between Enter & STsuspend nodes
    //   remove & recompute dps between variables
    Dependencies vardps;
    vardps.compute(engrc);

    STDPS compdps(engrc);
    compdps.execute();

    // Convert the GRC graph into a PDG
    gf->enumerate(dotrefmap, strefmap);
    GRC2PDG converter(top, dotrefmap);
  }

  IR::XMLostream o(std::cout);
  o << n;

  return 0;
}
```

30 ⟨*cec-grcpdg.cpp* 30⟩≡

```
#include "IR.hpp"
#include "AST.hpp"

#include <iostream>
#include <fstream>
#include <set>
#include <map>
#include <vector>

using namespace AST;
using namespace std;

typedef map<GRCNode *, int> CFGmap;
typedef map<STNode *, int> STmap;
```

⟨*utilities* 2a⟩

⟨*stdps class* 3⟩

⟨*dependency class* 5⟩

⟨*dependency method definitions* 6a⟩

⟨*grcpdg class* 12⟩

⟨*main function* 29⟩

References

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.