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⟩≡
```
template <class T> bool contains(set<T> &s, T o) {
  return s.find(o) != s.end();
}
```

2b ⟨*utilities* 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*⟩≡

```
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 = RET.begin();
        j != RET.end(); j++){
          if (same_sstp(n,*j)){
            **j << 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 = RET.begin();
         j != RET.end(); j++){
      if(st_ancestor(n,*j)){
        **j << n;
      }
    }
  }

  if (dynamic_cast<Enter *>(n))
    RET.insert(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.

6 ⟨*dependency class* 6⟩≡

```
class Dependencies : public Visitor {

protected:
  set<GRCNode *> visited;

  GRCNode *current;
  map<GRCNode *, bool> par_label;

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

  //writers & readers for signals
  map<SignalSymbol *, SignalNodes> dependencies;

  //writers & readers for variables
  map<VariableSymbol *, SignalNodes> v_dependencies;

  //procedure calls & function calls
  set<GRCNode *> all_calls;

  ⟨dependency methods 11c⟩
  void mark_par(GRCNode* n);
  bool have_comm_pp_gen(GRCNode* n, GRCNode* m);
  bool have_comm_pp(GRCNode* n, GRCNode* m);
  bool have_dps(GRCNode *n, GRCNode *m);
  void find_mra(GRCNode *n, const SignalNodes &sn, bool rw);
  void find_mra_calls(GRCNode *n);

public:

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

7 ⟨*dependency method definitions* 7⟩≡

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

  dfs(root);

  //add dps on signals
  for ( map<SignalSymbol *, SignalNodes>::const_iterator i =
          dependencies.begin() ; i != 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)) && !have_dps(*j, *k))
            **k << *j;
        }
      }
    }
  }


  //add dps on variables
  for (map<VariableSymbol *, SignalNodes>::const_iterator j =
       v_dependencies.begin(); j != v_dependencies.end();
       j++ ) {
    //VariableSymbol *var = (*j).first;
    const SignalNodes &sn = (*j).second;
    for (set<GRCNode *>::const_iterator i = sn.writers.begin();
         i != sn.writers.end(); i++){
      //looking for most-recent-ancestor of readers/writer on var
      // and save them in visited set
      visited.clear();
      find_mra(*i,sn,true);
      for(set<GRCNode *>::const_iterator k = visited.begin();
          k != visited.end(); k++){
        if (!have_dps(*k, *i))
          **i << *k;
      }
    }

    for (set<GRCNode *>::const_iterator i = sn.readers.begin();
         i != sn.readers.end(); i++){
```

```
      //looking for most-recent-ancestor of writer on var
      visited.clear();
      find_mra(*i,sn,false);
      //if(visited.size() > 1)
        //cerr<<"Warning: reader "<<*i<<" has more than one pre-writers\n";
      for(set<GRCNode *>::const_iterator k = visited.begin();
          k != visited.end(); k++){
        if (!have_dps(*k, *i))
          **i << *k;
      }
    }
  }

  //add dps btw function/procedure calls
  for (set<GRCNode *>::const_iterator i = all_calls.begin();
       i != all_calls.end(); i++){
    visited.clear();
    find_mra_calls(*i);
    for(set<GRCNode *>::const_iterator k = visited.begin();
        k != visited.end(); k++){
      if (!have_dps(*k, *i))
        **i << *k;
    }
  }

}
```

8 ⟨*dependency method definitions* 7⟩+≡

```
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);
  }
```

9a ⟨*dependency method definitions* 7⟩+≡

```
//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);
}
```

9b ⟨*dependency method definitions* 7⟩+≡

```
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;

}
```

10a ⟨*dependency method definitions* 7⟩+≡

```
//test if two nodes n & m have data dependency already
bool Dependencies::have_dps(GRCNode* n, GRCNode* m)
{
  vector<GRCNode *>::iterator i;
  bool found = false;

  for (i = n->dataSuccessors.begin(); i != n->dataSuccessors.end(); i++){
    if (*i == m){
      found = true;
      break;
    }
  }

  return found;
}
```

10b ⟨*dependency method definitions* 7⟩+≡

```
//find the most recent ancestor of n which R/W var
void Dependencies::find_mra(GRCNode *n, const SignalNodes &sn, bool rw)
{
  vector<GRCNode *>::const_iterator i;

  if (!n)
    return;

  for (i = n->predecessors.begin(); i != n->predecessors.end(); i++){

    if (sn.writers.find(*i) != sn.writers.end()) {//looking for writer
      if ((!rw) || (visited.size() == 0)){
        visited.insert(*i);
        return;
      }
    }
    else if (rw) {//looking for reader also
      if (sn.readers.find(*i) != sn.readers.end()){
        visited.insert(*i);
      }
    }
    find_mra(*i, sn, rw);
  }
}
```

11a ⟨*dependency method definitions* 7⟩+≡
```
//find most recent ancestor of n which includes a function/procedure call
void Dependencies::find_mra_calls(GRCNode *n)
{
  vector<GRCNode *>::const_iterator i;

  if (!n)
    return;

  for (i = n->predecessors.begin(); i != n->predecessors.end(); i++){
    if (all_calls.find(*i) != all_calls.end()) {
        visited.insert(*i);
        return;
    }
    else{
      find_mra_calls(*i);
    }
  }
}
```

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.

11b ⟨*dependency method definitions* 7⟩+≡
```
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);
}
```

11c ⟨*dependency methods* 11c⟩≡
```
void dfs(GRCNode *);
```

3.2 Action

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

12a ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(Action &act)
{
    act.body->welcome(*this);
    return Status();
}
```

12b ⟨*dependency methods* 11c⟩+≡
```
Status visit(Action &);
```

3.3 Emit

An emit statement, which emits a signal.

12c ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(Emit &emt)
{
    dependencies[emt.signal].writers.insert(current);
    current->dataSuccessors.clear();
    if (emt.value)
      emt.value->welcome(*this);

    return Status();
}
```

12d ⟨*dependency methods* 11c⟩+≡
```
Status visit(Emit &);
```

3.4 Exit

An exit statement, which exits a trap.

12e ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(Exit &ext)
{
    dependencies[ext.trap].writers.insert(current);
    current->dataSuccessors.clear();
    if (ext.value)
      ext.value->welcome(*this);
    return Status();
}
```

12f ⟨*dependency methods* 11c⟩+≡
```
Status visit(Exit &);
```

3.5 Assign & asn

An assign statement, which assigns.

13a ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(Assign &asn)
{
    v_dependencies[asn.variable].writers.insert(current);
    //not sure whether need to clear dataSucc or dataPred yet
    if (asn.value)
      asn.value->welcome(*this);
    return Status();
}
```

13b ⟨*dependency methods* 11c⟩+≡
```
Status visit(Assign &);
```

3.6 DefineSignal

The DefineSignal node is like an emit.

13c ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(DefineSignal &ds)
{
  assert(ds.signal);
  dependencies[ds.signal].writers.insert(current);
  current->dataSuccessors.clear();
  return Status();
}
```

13d ⟨*dependency methods* 11c⟩+≡
```
Status visit(DefineSignal &);
```

3.7 Test

This descends down its predicate, possibly adding signal testers

13e ⟨*dependency methods* 11c⟩+≡
```
Status visit(Test &t) {
  t.predicate->welcome(*this); return Status();
}
```

3.8 StartCounter

Do nothing.

13f ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(StartCounter &sct)
{
    return Status();
}
```

14a ⟨*dependency methods* 11c⟩+≡
```
Status visit(StartCounter &);
```

3.9 ProcedureCall

Data dependencies are added by looking at the ref/value parameters of a ProcedureCall.

14b ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(ProcedureCall &prc)
{
  all_calls.insert(current);

  for(vector<Expression *>::const_iterator i = prc.value_args.begin() ;
       i != prc.value_args.end() ; i++) {
    (*i)->welcome(*this);
  }
  for(vector<VariableSymbol *>::const_iterator i = prc.reference_args.begin() ;
       i != prc.reference_args.end() ; i++) {
    v_dependencies[*i].readers.insert(current);
    v_dependencies[*i].writers.insert(current);
  }
  return Status();
}
```

14c ⟨*dependency methods* 11c⟩+≡
```
Status visit(ProcedureCall &);
```

3.10 FunctionCall

Here we add data dependency by looking at the parameters of a FunctionCall.

14d ⟨*dependency method definitions* 7⟩+≡
```
Status Dependencies::visit(FunctionCall &func)
{
  all_calls.insert(current);
  for (vector<Expression*>::const_iterator i = func.arguments.begin() ;
       i != func.arguments.end() ; i++) {
    (*i)->welcome(*this);
  }
  return Status();
}
```

14e ⟨*dependency methods* 11c⟩+≡
```
Status visit(FunctionCall &);
```

3.11 Expressions

15a ⟨*dependency methods* 11c⟩+≡
```
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(LoadVariableExpression &e) {
  v_dependencies[e.variable].readers.insert(current);
  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();
}
```

3.11.1 Vacuous Expression Nodes

15b ⟨*dependency methods* 11c⟩+≡
```
Status visit(Literal &) { return Status(); }
```

3.12 Trivial visitors

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

16 ⟨*dependency methods* 11c⟩+≡

```
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

17 ⟨*grcpdg class* 17⟩≡

```
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;

  int debug, debug2;

public:
  ⟨method declarations 18⟩
};
```

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.

18 ⟨*method declarations* 18⟩≡

```
GRC2PDG(GRCNode *top, CFGmap &dotrefmap) : dotrefmap(dotrefmap)
{
  debug=0;debug2=0;

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

6 Depth-first search on the reverse graph

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

19 ⟨*method declarations* 18⟩+≡

```
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.

20 ⟨*method declarations* 18⟩+≡

```
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

21 ⟨*method declarations* 18⟩+≡

```
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.

22a ⟨*method declarations* 18⟩+≡

```
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.

22b ⟨*method declarations* 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

23 ⟨*method declarations* 18⟩+≡

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

  int counter = N;

  //for each node i
  for (int i = 0; i < N; i++ ) {
    if(debug) 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

  if(debug) 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);
      if(debug) cerr<<"  null succ\n";
      continue;
    }
    if (dynamic_cast<ExitGRC *>(vert[*ic])){
      if(debug) cerr<<"  exit grc succ\n";
      continue;
    }

    Fork *reg = new Fork();

    if(debug) 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;

      if(debug) cerr<<" IY "<<dotrefmap[vert[*iy]]<<"\n";

      reachability.clear();
      if(debug) 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);
      }
      if(debug) 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:
          if(debug) cerr<<"add new reg node: "<<counter<<"\n";
          nodenum[reg] = counter++;
          vert.push_back(reg);
          n->successors.push_back(reg);
          reg->predecessors.push_back(n);
          break;
        }
      }
      if(debug) cerr<<"N"<<dotrefmap[vert[i]]<<" is finished\n";
    }
  }
}
```

11 copy conn

26a ⟨*method declarations* 18⟩+≡

```
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

26b ⟨*method declarations* 18⟩+≡

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

13 reachable

27 ⟨*method declarations* 18⟩+≡

```
bool reachable(int from, int to)
{
  if(debug2) cerr<<" dfs "<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"\n";

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

  if (from == 0){
    if(debug2) cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  NO2\n";
    reachability[from] = false;
    return false;
  }

  if (from == -1){
    if(debug2) cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES2\n";
    reachability[from] = true;
    return true;
  }

  if (to == from){
    if(debug2) 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)){
        if(debug2) cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES3\n";
        reachability[from] = true;
        return true;
      }
    }
    if(debug2) 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)){
      if(debug2) cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  NO4\n";
```

```
      reachability[from] = false;
      return false;
    }
  }
  if(debug2) cerr<<dotrefmap[vert[from]]<<"->"<<dotrefmap[vert[to]]<<"  YES4\n";
  reachability[from] = true;
  return true;
}
```

14 Remove nodes with all null successors, and null nodes under forks

29 ⟨*method declarations* 18⟩+≡

```
void removeJunkNull(GRCNode *n)
{
  vector<GRCNode *>::iterator i;
  vector<GRCNode *> newch;
  bool isfork = false;

  if (!n)
    return;

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

  for (i = n->successors.begin(); i != n->successors.end(); i++)
    removeJunkNull(*i);

  if (dynamic_cast<Fork *>(n))
    isfork = true;

  if (n->successors.size() == 0)
    return;

  for (i = n->successors.begin(); i != n->successors.end(); i++){
    if (!*i){
      if (isfork){
        rm_invect((*i)->predecessors, n);
        continue;
      }
    }
    else if ((dynamic_cast<Fork *>(*i)) && ((*i)->successors.size() == 0)){
      rm_invect((*i)->predecessors, n);
      rm_datadps(*i);
      continue;
    }
    else if (all_child_null(*i)){
      if (isfork){
        rm_invect((*i)->predecessors, n);
        rm_datadps(*i);
        continue;
      }
      else
        *i = NULL;
    }
    newch.push_back(*i);
  }
```

```
    n->successors = newch;

  }
```

15 Remove consequencial fork nodes

30 ⟨*method declarations* 18⟩+≡

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

    if (!n)
      return;

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

    for (i = n->successors.begin(); i != n->successors.end(); i++)
      removeJunkFork(*i);

    if (dynamic_cast<Fork *>(n)){
      assert(n->successors.size()>0);
      for (i = n->successors.begin(); i != n->successors.end(); i++)
        if ((dynamic_cast<Fork *>(*i)) && ((*i)->predecessors.size() == 1)){
          for (j = (*i)->successors.begin(); j != (*i)->successors.end(); j++){
            newch.push_back(*j);
            rm_invect((*j)->predecessors, *i);
            (*j)->predecessors.push_back(n);
          }
          (*i)->predecessors.clear();
          (*i)->successors.clear();
        }
        else
          newch.push_back(*i);
      n->successors = newch;
    }
  }
```

16 remove element in vector

31a ⟨*method declarations* 18⟩+≡
```
void rm_invect(vector<GRCNode *> &vec, GRCNode *n)
{
  assert(n);
  vector<GRCNode *>::iterator i = vec.begin();
    while (i != vec.end()){
      if ((*i) == n)
        i = vec.erase(i);
      else
        i++;
    }
}
```

17 rm datadps

31b ⟨*method declarations* 18⟩+≡
```
void rm_datadps(GRCNode *n)
{
  vector<GRCNode *>::iterator i;

  for (i = n->dataPredecessors.begin(); i != n->dataPredecessors.end(); i++){
    rm_invect((*i)->dataSuccessors, n);
  }

  for (i = n->dataSuccessors.begin(); i != n->dataSuccessors.end(); i++){
    rm_invect((*i)->dataPredecessors, n);
  }

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

18 all child null

32a ⟨*method declarations* 18⟩+≡

```
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;
}
```

19 Printing methods

32b ⟨*printing method declarations* 32b⟩≡

```
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";
  }
  }
```

33a ⟨*printing method declarations* 32b⟩+≡
```
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";
  }
}
```

33b ⟨*printing method declarations* 32b⟩+≡
```
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";
  }
}
```

33c ⟨*printing method declarations* 32b⟩+≡
```
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

34 ⟨*main function* 34⟩≡

```
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;
}
```

35 ⟨*cec-grcpdg.cpp* 35⟩≡

```
#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 6⟩

⟨dependency method definitions 7⟩

⟨grcpdg class 17⟩

⟨main function 34⟩
```

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.