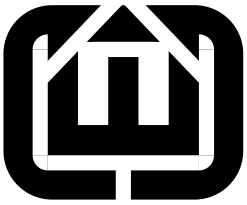
CEC AST-to-GRC Translator



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

Contents

1 Assigning Completion Codes

GRC synthesis, especially related to the *trap* statement, needs to know the completion code of each trap. This class assigns them. Later, the `GrcSynth` class uses this information.

2 ⟨*completion code class* 2⟩≡ (51a)

```
class CompletionCodes : public Visitor {
  int maxOverModule; // Maximum code for this
  map<Abort*, int> codeOfAbort; // Code for each weak abort
  map<SignalSymbol*, int> codeOfTrapSymbol; // Code for each trap symbol
  map<Trap*, int> codeOfTrap; // Code for each trap statement
public:

  void alsoMax(AST::ASTNode *n, int &m) {
     int max = recurse(n);
     if (max > m) m = max;
  }

  int recurse(AST::ASTNode *n) {
     if (n) return n->welcome(*this).i;
     else return 0;
```

```
  }

  ⟨completion code methods 3a⟩
};
```

The constructor takes a module and computes its completion codes.

3a ⟨completion code methods 3a⟩≡ (2) 3b▷

```
CompletionCodes(Module *m)
{
  assert(m);
  assert(m->body);
  maxOverModule = recurse(m->body);
  if (maxOverModule <= 1) maxOverModule = 1;
}

virtual ~CompletionCodes() {}
```

These accessors return codes for the various objects.

3b ⟨completion code methods 3a⟩+≡ (2) ◁3a 3c▷

```
  int max() const { return maxOverModule; }

  int operator [] (Abort *a) {
    assert(codeOfAbort.find(a) != codeOfAbort.end());
    return codeOfAbort[a];
  }

  int operator [] (SignalSymbol *ts) {
    assert(codeOfTrapSymbol.find(ts) != codeOfTrapSymbol.end());
    return codeOfTrapSymbol[ts];
  }

  int operator [] (Trap *t) {
    assert(codeOfTrap.find(t) != codeOfTrap.end());
    return codeOfTrap[t];
  }
```

1.1 Composite Statements

3c ⟨completion code methods 3a⟩+≡ (2) ◁3b 4a▷

```
Status visit(Signal &s) { return recurse(s.body); }
Status visit(Var &s) { return recurse(s.body); }
Status visit(Loop &s) { return recurse(s.body); }
Status visit(Repeat &s) { return recurse(s.body); }
Status visit(Suspend &s) { return recurse(s.body); }
Status visit(PredicatedStatement &s) { return recurse(s.body); }
```

4a ⟨*completion code methods* 3a⟩+≡ (2) ◁3c 4b▷

```
Status visit(StatementList &l) {
  int max = 1;
  for (vector<Statement*>::iterator i = l.statements.begin() ;
       i != l.statements.end() ; i++ ) alsoMax(*i, max);
  return max;
}
```

4b ⟨*completion code methods* 3a⟩+≡ (2) ◁4a 4c▷

```
Status visit(ParallelStatementList &l) {
  int max = 1;
  for (vector<Statement*>::iterator i = l.threads.begin() ;
       i != l.threads.end() ; i++ ) alsoMax(*i, max);
  return max;
}
```

4c ⟨*completion code methods* 3a⟩+≡ (2) ◁4b 4d▷

```
Status visit(IfThenElse& n) {
  int max = 1;
  alsoMax(n.then_part, max);
  alsoMax(n.else_part, max);
  return max;
}
```

1.2 Leaf Statements

None of these needs a completion code, so they all return 0;

4d ⟨*completion code methods* 3a⟩+≡ (2) ◁4c 5a▷

```
Status visit(Emit&) { return Status(0); }
Status visit(Assign&) { return Status(0); }
Status visit(ProcedureCall&) { return Status(0); }
Status visit(TaskCall&) { return Status(0); }
Status visit(Exec&) { return Status(0); }
Status visit(Exit&) { return Status(0); }
Status visit(Run&) { return Status(0); }
Status visit(Pause&) { return Status(0); }
```

1.3 Abort

Weak abort statements use one code for normal termination and one for each case; strong aborts do not use any more.

5a ⟨*completion code methods* 3a⟩+≡ (2) ◁4d 5b▷

```
Status visit(Abort &s) {
  int max = 1;
  alsoMax(s.body, max);
  for (vector<PredicatedStatement*>::iterator i = s.cases.begin() ;
       i != s.cases.end() ; i++ ) alsoMax(*i, max);
  if (s.is_weak) {
    int code = max + 1;
    codeOfAbort[&s] = code;
    assert(code >= 2);
    max += 1 + s.cases.size();
  }
  return max;
}
```

1.4 Trap

Trap is the only statement that consumes completion codes.

5b ⟨*completion code methods* 3a⟩+≡ (2) ◁5a

```
Status visit(Trap &s) {
  int max = 1;
  alsoMax(s.body, max);

  // FIXME: is this the right order?  Should the predicates be
  // considered before or after the code is assigned?

  for (vector<PredicatedStatement*>::iterator i = s.handlers.begin() ;
       i != s.handlers.end() ; i++ ) alsoMax(*i, max);

  max++; // Allocate an exit level for this trap statement

  codeOfTrap[&s] = max;

  assert(s.symbols);
  for (SymbolTable::const_iterator i = s.symbols->begin() ; i !=
       s.symbols->end() ; i++) {
    SignalSymbol *ts = dynamic_cast<SignalSymbol*>(*i);
    assert(ts);
    assert(ts->kind == SignalSymbol::Trap);
    codeOfTrapSymbol[ts] = max;
  }

  return max;
}
```

2 The Cloner class

This is used to duplicate expression trees during GRC synthesis to ensure that they are not shared between surface and depth copies of the same code.

The API for this class is the () operator so you can write code like

```
Cloner clone;

MyObject *oldo = ...;
MyObject *newo = clone(oldo);
```

Within the Cloner class methods, the `clone` template function accomplishes the same thing.

6a ⟨*cloner class* 6a⟩≡ (51a)

```
class Cloner : public Visitor {
public:
  template <class T> T* operator() (T* n) {
    if (!n) return NULL;
    T* result = dynamic_cast<T*>(n->welcome(*this).n);
    assert(result);
    return result;
  }

  ⟨public cloner methods 7a⟩

  virtual ~Cloner() {}

protected:
  template <class T> T* clone(T* n) { return (*this)(n); }

  ⟨cloner methods 6b⟩
};
```

The following map and methods manage renaming local signals.

6b ⟨*cloner methods* 6b⟩≡ (6a) 8a▷

```
map<SignalSymbol*, SignalSymbol*> newsig;
```

Create a new Local signal with a unique name and add it to both the mapping and the symbol table.

7a ⟨*public cloner methods* 7a⟩≡ (6a) 7b▷
```
SignalSymbol *cloneLocalSignal(SignalSymbol *s, SymbolTable *st) {
  assert(s);
  assert(newsig.find(s) == newsig.end()); // Should not already be there
  assert(st);

  string name = s->name;
  int next = 1;
  while (st->contains(name)) {
    char buf[10];
    sprintf(buf, "%d", next++);
    name = s->name + '_' + buf;
  }
  SignalSymbol::kinds kind =
    (s->kind == SignalSymbol::Trap) ? SignalSymbol::Trap : SignalSymbol::Local;
  SignalSymbol *result =
    new SignalSymbol(name, s->type, kind, clone(s->combine),
                     clone(s->initializer), clone(s->presence),
                     clone(s->value));
  st->enter(result);
  newsig[s] = result;
  return result;
}
```

Set a signal to map to itself.

7b ⟨*public cloner methods* 7a⟩+≡ (6a) ◁7a 7c▷
```
void sameSig(SignalSymbol *s) {
  assert(s);
  assert(newsig.find(s) == newsig.end());
  newsig[s] = s;
}
```

`clearSig` deletes a previously-established signal mapping.

7c ⟨*public cloner methods* 7a⟩+≡ (6a) ◁7b
```
void clearSig(SignalSymbol *orig) {
  assert(orig);
  map<SignalSymbol*, SignalSymbol*>::iterator i = newsig.find(orig);
  assert(i != newsig.end());
  newsig.erase(i);
}
```

2.1 Statments: Emit, Assign, and Exit

8a ⟨*cloner methods* 6b⟩+≡ (6a) ◁6b 8b▷

```
Status visit(Emit &s) {
  return new Emit(clone(s.signal), clone(s.value));
}

Status visit(Exit &s) {
  return new Exit(clone(s.trap), clone(s.value));
}

Status visit(Assign &s) {
  return new Assign(clone(s.variable), clone(s.value));
}
```

2.2 Literals, Variable, and Signal references

8b ⟨*cloner methods* 6b⟩+≡ (6a) ◁8a 8c▷

```
Status visit(Literal &s) { return new Literal(s.value, s.type); }

Status visit(LoadVariableExpression &s) {
  return new LoadVariableExpression(clone(s.variable));
}

Status visit(LoadSignalExpression &s) {
  return new LoadSignalExpression(s.type, clone(s.signal));
}

Status visit(LoadSignalValueExpression &s) {
  return new LoadSignalValueExpression(clone(s.signal));
}
```

2.3 Operators

8c ⟨*cloner methods* 6b⟩+≡ (6a) ◁8b 9a▷

```
Status visit(UnaryOp &s) {
  return new UnaryOp(s.type, s.op, clone(s.source));
}

Status visit(BinaryOp &s) {
  return new BinaryOp(s.type, s.op, clone(s.source1), clone(s.source2));
}
```

2.4 Function Call

9a ⟨*cloner methods* 6b⟩+≡ (6a) ◁8c 9b▷

```
Status visit(FunctionCall &s) {
  FunctionCall *c = new FunctionCall(clone(s.callee));
  for (vector<Expression*>::const_iterator i = s.arguments.begin() ;
       i != s.arguments.end() ; i++) {
    assert(*i);
    c->arguments.push_back(clone(*i));
  }
  return c;
}
```

2.5 CheckCounter

9b ⟨*cloner methods* 6b⟩+≡ (6a) ◁9a 9c▷

```
Status visit(CheckCounter &s) {
  return new CheckCounter(s.type, clone(s.counter));
}
```

FIXME: Should the counter object itself be duplicated? Probably not, since it's probably more like a symbol.

9c ⟨*cloner methods* 6b⟩+≡ (6a) ◁9b 9d▷

```
Status visit(Counter &s) { return &s; }
```

2.6 Symbols

These do not clone anything, just return themselves.

9d ⟨*cloner methods* 6b⟩+≡ (6a) ◁9c 9e▷

```
Status visit(VariableSymbol &s) { return &s; }
Status visit(ConstantSymbol &s) { return &s; }
Status visit(BuiltinConstantSymbol &s) { return &s; }
Status visit(BuiltinSignalSymbol &s) { return &s; }
Status visit(FunctionSymbol &s) { return &s; }
Status visit(BuiltinFunctionSymbol &s) { return &s; }
```

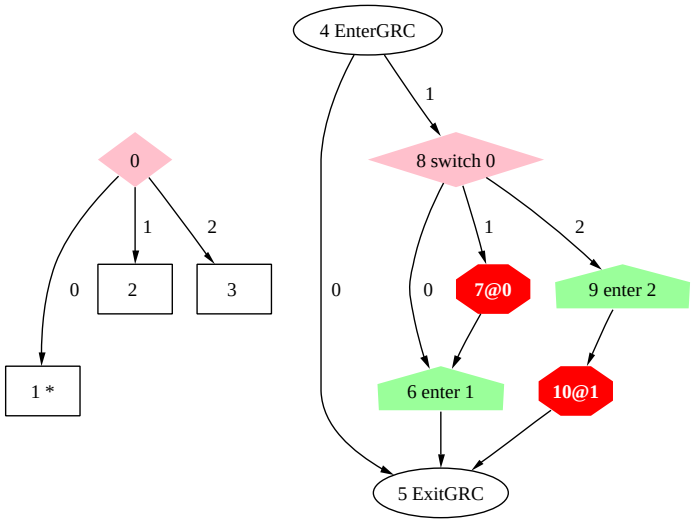
2.7 SignalSymbols

Local signals are cloned during the GRC construction process to separate different incarnations of the same signal (due, i.e., to reincarnation or schizophrenia in Esterel). As such, the cloner maintains a mapping from existing signals to their new versions, which this method returns.

9e ⟨*cloner methods* 6b⟩+≡ (6a) ◁9d

```
Status visit(SignalSymbol &s) {
  assert(newsig.find(&s) != newsig.end()); // should be there
  return newsig[&s];
}
```

```
module onepause:

pause

end module
```



(a) (b)

Figure 1: A small Esterel program and the GRC it generates

3 GRC Synthesis

3.1 The Context class

Used during synthesis. A stack that maintains where termination at levels 0, 1, etc. should branch. There are three main operations: push() starts a new environment by copying all continuations from the current environment, pop() discards the current environment, and the () operator, which returns a reference to the continuation at the given level. This enables statements such as `surface(0) = mynode`.

Note that the maximum exit level must have been computed earlier and passed to the constructor. (This is the `size` field.)

10 ⟨*context class* 10⟩≡ (51a)

```
struct Context {
  int size;
  std::stack<GRCNode**> continuations;

  Context(int sz) : size(sz) {
    assert(sz >= 2); // Must at least have termination at levels 0 and 1
    continuations.push(new GRCNode*[size]);
    for (int i = 0 ; i < size ; i++ ) continuations.top()[i] = 0;
  }
```

```
  ~Context() {}

  void push(Context &c) {
    GRCNode **parent = c.continuations.top();
    continuations.push(new GRCNode*[size]);
    GRCNode **child = continuations.top();
    for ( int i = 0 ; i < size ; i++ ) child[i] = parent[i];
  }

  void push() { push(*this); }

  void pop() {
    delete [] continuations.top();
    continuations.pop();
  }

  GRCNode *& operator ()(int k) {
    assert(k >= 0);
    assert(k < size);
    return continuations.top()[k];
  }
};
```

3.2 The GrcSynth class

Where all the action happens. Tracks contexts for the surface, depth, and selection tree, as well as the surface, depth, and selection tree walkers (the `Surface`, `Depth`, and `SelTree` classes).

The `ast2st` map records which selection tree node is owned by certain AST nodes.

12a ⟨*GrcSynth class* 12a⟩≡ (51a)

```
struct GrcSynth {
  Module *module;
  CompletionCodes &code;

  Cloner clone;

  Context surface_context;
  Context depth_context;

  Surface surface;
  Depth depth;
  SelTree seltree;

  map<const ASTNode*, STNode*> ast2st;

  BuiltinTypeSymbol *integer_type;
  BuiltinTypeSymbol *boolean_type;

  ⟨GrcSynth methods 12b⟩
};
```

The constructor initializes the walkers, finds some built-in types, and initializes the cloner's (trivial) mapping of external signals.

12b ⟨*GrcSynth methods* 12b⟩≡ (12a) 14▷

```
GrcSynth(Module *, CompletionCodes &);
```

13 ⟨*grc synth method definitions* 13⟩≡ (51b)

```
GrcSynth::GrcSynth(Module *m, CompletionCodes &c)
  : module(m), code(c),
    surface_context(code.max() + 1),
    depth_context(code.max() +1),
    surface(surface_context, *this), depth(depth_context, *this),
    seltree(*this)
{
  assert(m);
  assert(m->types);
  integer_type = dynamic_cast<BuiltinTypeSymbol*>(m->types->get("integer"));
  assert(integer_type);
  boolean_type = dynamic_cast<BuiltinTypeSymbol*>(m->types->get("boolean"));
  assert(boolean_type);

  for ( SymbolTable::const_iterator i = m->signals->begin() ;
        i != m->signals->end() ; i++ ) {
    SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
    assert(s);
    clone.sameSig(s);
  }
}
```

The `synthesize` method kicks off the walkers after setting up some environment. See Figure 1 for an example of the scaffolding it generates.

14 ⟨*GrcSynth methods* 12b⟩+≡ (12a) ◁12b

```
GRCgraph *synthesize()
{
  assert(module->body);

  // Set up initial and terminal states in the selection tree

  STexcl *stroot = new STexcl();
  STleaf *boot = new STleaf();
  STleaf *finished = new STleaf();
  finished->setfinal();

  // Set up the root of the GRC

  EnterGRC *engrc = new EnterGRC();
  ExitGRC *exgrc = new ExitGRC();

  Enter *enfinished = new Enter(finished);
  Switch *top_switch = new Switch(stroot);

  *engrc >> exgrc >> top_switch;
  *enfinished >> exgrc;

  enfinished->st = finished;

  Terminate *term0 = new Terminate(0, 0);
  *term0 >> enfinished;
  Terminate *term1 = new Terminate(1, 0);
  *term1 >> exgrc;

  // Set up the context for the surface and the depth: point to term0 and 1

  surface_context(0) = depth_context(0) = term0;
  surface_context(1) = depth_context(1) = term1;

  // Build the selection tree and create the selection tree root

  STNode *synt_seltree = seltree.synthesize(module->body);
  *stroot >> finished >> synt_seltree >> boot;

  // Build the surface and the depth and attach them to the top switch

  GRCNode *synt_surface = surface.synthesize(module->body);
  GRCNode *synt_depth = depth.synthesize(module->body);
  *top_switch >> enfinished >> synt_depth >> synt_surface;

  GRCgraph *result = new GRCgraph(stroot, engrc);
```

```
  return result;
}
```

3.3 The SelTree, Surface, and Depth classes

These do all the actual work. Derived from the AST Visitor class, these recursively walk down the tree and return the nodes they synthesize.

The `synthesize` method is fundamental: it synthesizes the given AST node by calling one of the visitor methods and returns a GRC node.

The `recurse` method is similar but creates a new context and removes it before returning.

The `push_onto` method makes the second argument a successor of the first, then changes the first argument (passed a reference) into the second. Calling it repeatedly with the same variable as a first argument builds a chain whose tail is the variable.

The three workhorse classes are each derived from the `GrcWalker` class.

15 ⟨*grc walker class* 15⟩≡ (51a)

```
class GrcWalker : public Visitor {
protected:
  Context &context;
  GrcSynth &environment;
  Cloner &clone;
public:
  GrcWalker(Context &, GrcSynth &);

  GRCNode *synthesize(ASTNode *n) {
    assert(n);
    n->welcome(*this);
    assert(context(0));
    return context(0);
  }

  GRCNode *recurse(ASTNode *n) {
    context.push();
    GRCNode *nn = synthesize(n);
    context.pop();
    return nn;
  }

  static GRCNode* push_onto(GRCNode *& b, GRCNode* n) {
    *n >> b;
    b = n;
    return b;
  }

  STNode *stnode(const ASTNode &);
};
```

16a ⟨*grc walker methods* 16a⟩≡ (51b) 16b▷
```
GrcWalker::GrcWalker(Context &c, GrcSynth &e)
    : context(c), environment(e), clone(e.clone) {}
```

16b ⟨*grc walker methods* 16a⟩+≡ (51b) ◁16a
```
STNode *GrcWalker::stnode(const ASTNode &n) {
  assert(environment.ast2st.find(&n) != environment.ast2st.end() );
  return environment.ast2st[&n];
}
```

The selection tree is synthesized first, since many nodes in the control-flow portion refer to selection tree nodes, then the surface, then the depth.

16c ⟨*st class* 16c⟩≡ (51a)
```
class SelTree : public Visitor {
protected:
  GrcSynth &environment;
public:
  SelTree(GrcSynth &e): environment(e) {}

  STNode *synthesize(ASTNode *n) {
    assert(n);
    STNode *result = dynamic_cast<STNode*>(n->welcome(*this).n);
    assert(result);
    return result;
  }

  void setNode(const ASTNode &, STNode *);

  ⟨st methods 17a⟩
};
```

16d ⟨*st method definitions* 16d⟩≡ (51b) 17b▷
```
void SelTree::setNode(const ASTNode &n, STNode *sn) {
  assert(sn);
  environment.ast2st[&n] = sn;
}
```

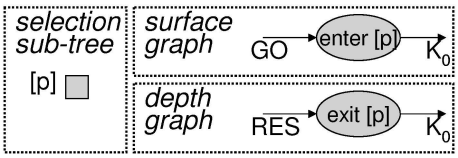
16e ⟨*surface class* 16e⟩≡ (51a)
```
class Surface : public GrcWalker {
public:
  Surface(Context &c, GrcSynth &e) : GrcWalker(c, e) {}
  ⟨surface methods 17c⟩
};
```

16f ⟨*depth class* 16f⟩≡ (51a)
```
class Depth : public GrcWalker {
public:
  Depth(Context &c, GrcSynth &e) : GrcWalker(c, e) {}
  ⟨depth methods 17e⟩
};
```

3.4 Statement Translators

Each of these define the `visit` methods for their AST nodes for the selection tree synthesis phase, the surface synthesis, and the depth synthesis.

3.4.1 Pause



The selection tree fragment is a single leaf node.

17a ⟨*st methods* 17a⟩≡ (16c) 18a▷

```
Status visit(Pause &);
```

17b ⟨*st method definitions* 16d⟩+≡ (51b) ◁16d 19e▷

```
Status SelTree::visit(Pause &s){
  STleaf *leaf = new STleaf();
  setNode(s, leaf);
  return Status(leaf);
}
```

The surface of a `pause` Enters and terminates at level 1.

17c ⟨*surface methods* 17c⟩≡ (16e) 18b▷

```
Status visit(Pause &);
```

17d ⟨*surface method definitions* 17d⟩≡ (51b) 18c▷

```
Status Surface::visit(Pause &s) {
  Enter *en = new Enter(stnode(s));
  assert(en->st);
  *en >> context(1);
  context(0) = en;
  return Status();
}
```

The depth is empty.

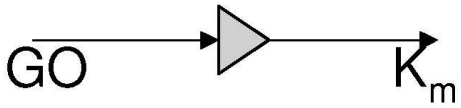
17e ⟨*depth methods* 17e⟩≡ (16f) 18d▷

```
Status visit(Pause &);
```

17f ⟨*depth method definitions* 17f⟩≡ (51b) 20c▷

```
Status Depth::visit(Pause &s) {
  return Status();
}
```

3.4.2 Exit



This "emits" the trap and sends the incoming activation to the code for the exit.

18a ⟨*st methods* 17a⟩+≡ (16c) ◁17a 18e▷
```
Status visit(Exit &s) {
  return Status(new STref());
}
```

18b ⟨*surface methods* 17c⟩+≡ (16e) ◁17c 18f▷
```
Status visit(Exit &);
```

18c ⟨*surface method definitions* 17d⟩+≡ (51b) ◁17d 20a▷
```
Status Surface::visit(Exit &s) {
  assert(s.trap);
  context(0) = context(environment.code[s.trap]);
  push_onto(context(0), new Action(clone(&s)));
  return Status();
}
```

18d ⟨*depth methods* 17e⟩+≡ (16f) ◁17e 18g▷
```
Status visit(Exit &) { return Status(); }
```

3.4.3 Emit

18e ⟨*st methods* 17a⟩+≡ (16c) ◁18a 19a▷
```
Status visit(Emit &) {
  return Status(new STref());
}
```

This becomes an action in the surface; the depth is vacuous.

18f ⟨*surface methods* 17c⟩+≡ (16e) ◁18b 19b▷
```
Status visit(Emit &s) {
  push_onto(context(0), new Action(clone(&s)));
  return Status();
}
```

18g ⟨*depth methods* 17e⟩+≡ (16f) ◁18d 19c▷
```
Status visit(Emit &) { return Status(); }
```

3.4.4 Assign

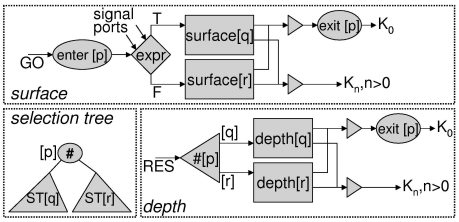
This also becomes an action with a vacuous depth.

19a ⟨*st methods* 17a⟩+≡ (16c) ◁18e 19d▷
```
Status visit(Assign &s) {
  return Status(new STref());
}
```

19b ⟨*surface methods* 17c⟩+≡ (16e) ◁18f 19f▷
```
Status visit(Assign &s) {
  Action *a = new Action(clone(&s));
  *a >> context(0);
  context(0) = a;
  return Status();
}
```

19c ⟨*depth methods* 17e⟩+≡ (16f) ◁18g 20b▷
```
Status visit(Assign &) { return Status(); }
```

3.4.5 IfThenElse



19d ⟨*st methods* 17a⟩+≡ (16c) ◁19a 20d▷
```
Status visit(IfThenElse &);
```

19e ⟨*st method definitions* 16d⟩+≡ (51b) ◁17b 21a▷
```
Status SelTree::visit(IfThenElse &s) {
  STexcl *ite = new STexcl();
  setNode(s, ite);

  *ite >> (s.else_part ? synthesize(s.else_part) : new STref())
       >> (s.then_part ? synthesize(s.then_part) : new STref());

  return Status(ite);
}
```

The surface depth of if-then-else is a test node.

19f ⟨*surface methods* 17c⟩+≡ (16e) ◁19b 21b▷
```
Status visit(IfThenElse &);
```

20a ⟨*surface method definitions* 17d⟩+≡ (51b) ◁18c 21c▷

```
Status Surface::visit(IfThenElse &s) {
  Enter *en;
  assert(s.predicate);
  Test *t = new Test(stnode(s), clone(s.predicate));
  *t >> ( (s.else_part != 0) ? recurse(s.else_part) : context(0))
     >> ( (s.then_part != 0) ? recurse(s.then_part) : context(0));
  context(0) = t;
  en = new Enter(stnode(s));
  push_onto(context(0), en);
  return Status();
}
```

The depth of an if-then-else node is a Switch that remembers which branch, if any, is still running.

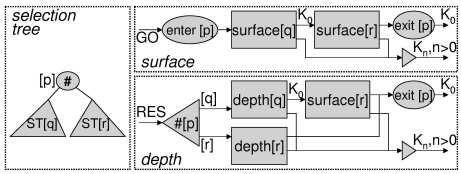
20b ⟨*depth methods* 17e⟩+≡ (16f) ◁19c 21d▷

```
Status visit(IfThenElse &);
```

20c ⟨*depth method definitions* 17f⟩+≡ (51b) ◁17f 22a▷

```
Status Depth::visit(IfThenElse &s) {
  Switch *sw = new Switch(stnode(s));
  *sw >> ( (s.else_part != 0) ? recurse(s.else_part) : context(0))
      >> ( (s.then_part != 0) ? recurse(s.then_part) : context(0));
  context(0) = sw;
  return Status();
}
```

3.4.6 StatementList



Sequencing is slightly difficult because of need to handle reincarnation.

20d ⟨*st methods* 17a⟩+≡ (16c) ◁19d 22b▷

```
Status visit(StatementList &s);
```

21a ⟨*st method definitions* 16d⟩+≡ (51b) ◁19e 22c▷

```
Status  SelTree::visit(StatementList &s)
{
  STexcl *excl = new STexcl();
  setNode(s, excl);

  for ( vector<Statement*>::reverse_iterator i = s.statements.rbegin() ;
        i != s.statements.rend() ; i++ ){
    assert(*i);
    *excl >> synthesize(*i);
  }

  return Status(excl);
}
```

21b ⟨*surface methods* 17c⟩+≡ (16e) ◁19f 22d▷

```
Status visit(StatementList &);
```

21c ⟨*surface method definitions* 17d⟩+≡ (51b) ◁20a 23a▷

```
Status Surface::visit(StatementList &s) {

  for ( vector<Statement*>::reverse_iterator i = s.statements.rbegin() ;
        i != s.statements.rend() ; i++ ) {
    assert(*i);
    context(0) = synthesize(*i);
  }

  push_onto(context(0), new Enter(stnode(s)));
  return Status();
}
```

21d ⟨*depth methods* 17e⟩+≡ (16f) ◁20b 23b▷

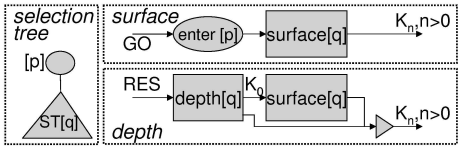
```
Status visit(StatementList &);
```

Can be optimized to remove dead code (?)

22a ⟨*depth method definitions* 17f⟩+≡ (51b) ◁20c 23c▷

```
Status Depth::visit(StatementList &s) {
  Switch *sw;
  if (!s.statements.empty()) {
    sw = new Switch(stnode(s));
    environment.surface_context.push(context);
    vector<Statement*>::reverse_iterator final = s.statements.rend();
    final--;
    for ( vector<Statement*>::reverse_iterator i = s.statements.rbegin() ;
          i != s.statements.rend() ; i++ ) {
      assert(*i);
      *sw >> synthesize(*i); // Build the depth
      // Build the surface
      if (i != final ) context(0) = environment.surface.synthesize(*i);
    }
    environment.surface_context.pop();
    context(0) = sw;
  }
  return Status();
}
```

3.4.7 Loop



Loops duplicate their surface.

22b ⟨*st methods* 17a⟩+≡ (16c) ◁20d 23d▷

```
Status visit(Loop &s);
```

22c ⟨*st method definitions* 16d⟩+≡ (51b) ◁21a 23e▷

```
Status SelTree::visit(Loop &s) {
  STref *lp = new STref();
  setNode(s, lp);
  *lp >> synthesize(s.body);
  return Status(lp);
}
```

22d ⟨*surface methods* 17c⟩+≡ (16e) ◁21b 23f▷

```
Status visit(Loop &);
```

23a ⟨*surface method definitions* 17d⟩+≡ (51b) ◁21c 23g▷
```
Status Surface::visit(Loop &s) {
  context(0) = synthesize(s.body);
  Enter *en = new Enter(stnode(s));
  push_onto(context(0), en);
  return Status();
}
```

23b ⟨*depth methods* 17e⟩+≡ (16f) ◁21d 24a▷
```
Status visit(Loop &);
```

23c ⟨*depth method definitions* 17f⟩+≡ (51b) ◁22a 24b▷
```
Status Depth::visit(Loop &s) {
  environment.surface_context.push(context);
  // Synthesize the surface
  context(0) = environment.surface.synthesize(s.body);
  // Synthesize the depth
  context(0) = synthesize(s.body);
  environment.surface_context.pop();
  return Status();
}
```

3.4.8 Repeat

This is a counted loop. It behaves much like `Loop`, except a counter is added. It assumes the body is NOT instantaneous (i.e. the body surface CAN'T terminate at level 0)

23d ⟨*st methods* 17a⟩+≡ (16c) ◁22b 28a▷
```
Status visit(Repeat &);
```

23e ⟨*st method definitions* 16d⟩+≡ (51b) ◁22c 25▷
```
Status SelTree::visit(Repeat &s) {
  STref *lp = new STref();
  setNode(s, lp);
  *lp >> synthesize(s.body);
  return Status(lp);
}
```

23f ⟨*surface methods* 17c⟩+≡ (16e) ◁22d 28b▷
```
Status visit(Repeat &);
```

23g ⟨*surface method definitions* 17d⟩+≡ (51b) ◁23a 26▷
```
Status Surface::visit(Repeat &s) {
  context(0) = synthesize(s.body);
  StartCounter *stcnt = new StartCounter(s.counter, stnode(s));
  push_onto(context(0), new Action(stcnt));
  Enter *en = new Enter(stnode(s));
  push_onto(context(0), en);
  return Status();
}
```

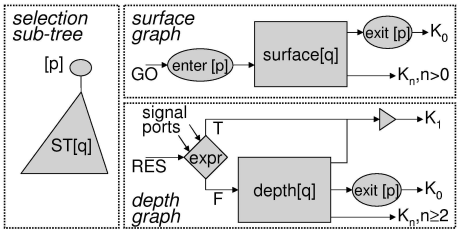
24a ⟨*depth methods* 17e⟩+≡ (16f) ◁23b 28c▷

```
Status visit(Repeat &);
```

24b ⟨*depth method definitions* 17f⟩+≡ (51b) ◁23c 27▷

```
Status Depth::visit(Repeat &s) {

  // std::cerr<<"depth visit\n";

  // Synthesize the surface
  environment.surface_context.push(context);
  GRCNode *restart = environment.surface.synthesize(s.body);
  environment.surface_context.pop();

  Test *tst = new Test(stnode(s), new CheckCounter(environment.boolean_type, s.counter));
  *tst >> restart >> context(0);
  //Synthesize the depth
  context(0) = tst;
  context(0) = synthesize(s.body);

  // std::cerr<<"visit ok\n";

  return Status();
}
```

3.4.9 Suspend



The selection tree fragment for a *suspend* consists of an exclusive node whose first child is a reference to the *suspend* statement and whose second child is a leaf if the suspend's predicate is immediate. The child of the reference is the selection tree for the body of the suspend.

25 ⟨*st method definitions* 16d⟩+≡ (51b) ◁23e 29▷

```
Status SelTree::visit(Suspend &s)
{
  STexcl *ex = new STexcl();
  STref *sp = new STref();
  sp->setsuspend();

  *ex >> sp;
  Delay *d = dynamic_cast<Delay*>(s.predicate);
  if (d && d->is_immediate) *ex >> new STleaf();

  assert(s.body);
  *sp >> synthesize(s.body);

  setNode(s, ex);
  return Status(ex);
}
```

26 ⟨*surface method definitions* 17d⟩+≡ (51b) ◁23g 30▷

```
Status Surface::visit(Suspend &s)
{
  assert(s.body);
  GRCNode *start = recurse(s.body);

  assert(stnode(s)->children.size() >= 1); // Should have at least one child
  STNode *bodytree = stnode(s)->children.front();
  assert(bodytree); // First child is STref for the body

  // Put an Enter for the suspend's body just before the code for the body
  push_onto(start, new Enter(bodytree));

  Delay *d = dynamic_cast<Delay*>(s.predicate);
  if (d) {
    if (d->is_immediate) {

      // An immediate predicate (e.g., suspend .. when immediate A)
      STNode *imleaf = stnode(s)->children.back();
      Enter *enimleaf = new Enter(imleaf);
      *enimleaf >> context(1); // Enter the immediate additional leaf
      Test *tst = new Test(bodytree, clone(d->predicate));
      *tst >> start >> enimleaf;
      start = tst;

    } else {

      // A counted suspend (e.g., suspend .. when 5 A)
      StartCounter *scnt = new StartCounter(d->counter, bodytree);
      push_onto(start, new Action(scnt));

    }
  }

  // Put an Enter for the suspend statement itself at the beginning
  push_onto(start, new Enter(stnode(s)));

  context(0) = start;
  return Status();
}
```

27 ⟨*depth method definitions* 17f⟩+≡ (51b) ◁24b 32▷

```
Status Depth::visit(Suspend &s) {

  assert(s.predicate);
  assert(s.body);
  assert(stnode(s));

  assert(stnode(s)->children.size() >= 1); // Should have at least one child
  STNode *bodytree = stnode(s)->children.front();
  assert(bodytree); // First child is STref for the body

  Switch *swimm = new Switch(stnode(s));

  GRCNode *start = synthesize(s.body);

  Delay *d = dynamic_cast<Delay*>(s.predicate);
  Expression *pred =
    (d != NULL) ?
      ( d->is_immediate ?
        clone(d->predicate) :
        new CheckCounter(environment.boolean_type, d->counter)
      ) :
    clone(s.predicate);

  // the depth test: body is already started

  Test *t = new Test(bodytree, pred);
  STSuspend *sts = new STSuspend(bodytree);
  *sts >> context(1);
  *t >> start >> sts;
  Enter *hold = new Enter(bodytree);
  *hold >> t;
  *swimm >> hold;

  // If the predicate is immediate then it's possible that the surface
  // still needs to start in the depth of the suspend (i.e., when
  // it was suspended in the first cycle)
  //
  // This section builds that portion of the code

  if (d && d->is_immediate) {
    assert(stnode(s)->children.size() == 2); // build in selection tree
    Enter *enimleaf = new Enter( stnode(s)->children.back() );
    Enter *en = new Enter( bodytree );
    *enimleaf >> context(1);
    t = new Test(NULL, clone(pred));
    start = environment.surface.recurse(s.body);
    push_onto(start, en);
    *t >> start >>enimleaf;
    *swimm >> t;
```

```
    }

    context(0) = swimm;
    return Status();
  }
```

28a ⟨*st methods* 17a⟩+≡ (16c) ◁23d 33a▷

```
Status visit(Suspend &);
```

28b ⟨*surface methods* 17c⟩+≡ (16e) ◁23f 33b▷

```
Status visit(Suspend &);
```

28c ⟨*depth methods* 17e⟩+≡ (16f) ◁24a 33c▷

```
Status visit(Suspend &);
```

3.4.10 Abort

The selection tree fragment for an *abort* consists of an exclusive node whose children are the body of the *abort* followed by the body of each of the non-vacuous handlers. The body of the abort is an STref node whose sole child is the tree for the body of the abort.

29 ⟨*st method definitions* 16d⟩+≡ (51b) ◁25 34a▷

```
Status SelTree::visit(Abort &s) {

  // The selection tree for the body of the abort:
  //  An STref node whose only child is the tree
  //  for the body of the abort

  assert(s.body); // Any abort should have a body
  STref *bodytree = new STref();
  bodytree->setabort();
  *bodytree >> synthesize(s.body);

  // The root of the tree for the abort: an exclusive
  // whose first child is the tree for the body of the abort

  STexcl *exclusive = new STexcl();
  *exclusive >> bodytree;

  // Attach the selection tree for each non-vacuous handler
  // under the exclusive node at the top of the abort

  for ( vector<PredicatedStatement*>::const_iterator i = s.cases.begin() ;
          i != s.cases.end() ; i++ ) {
      assert(*i);
      assert((*i)->predicate);
      if ((*i)->body) *exclusive >> synthesize((*i)->body);
  }

  setNode(s, exclusive);
  return Status(exclusive);
}
```

The surface fragment for an *abort* consists of an enter node followed by tests for any immediate predicates and initialization of any counted predicates finally followed by the surface for the body of the abort.

30 ⟨*surface method definitions* 17d⟩+≡ (51b) ◁26 35a▷

```
Status Surface::visit(Abort &s) {
  if (s.is_weak) throw IR::Error("weak abort.  Did the dismantler run?");

  // Synthesize the surface of the body

  context.push();
  assert(s.body);
  GRCNode *start = recurse(s.body);
  context.pop();

  // Add an enter node for STref node under the abort

  assert(stnode(s)->children.size() >= 1); // Should be at least one child
  assert(stnode(s)->children.front()); // First child is STref for this body

  // The selection tree node for the body of the abort
  STNode *bodytree = stnode(s)->children.front();

  push_onto(start, new Enter(bodytree));

  // Add a check for each immediate predicate and "initialize counter"
  // for each counted predicate

  for ( vector<PredicatedStatement*>::reverse_iterator i = s.cases.rbegin() ;
        i != s.cases.rend() ; i++ ) {
    assert(*i);
    assert((*i)->predicate);
    Delay *d = dynamic_cast<Delay*>((*i)->predicate);
    if (d) {
      if (d->is_immediate) {

        // An immediate predicate: add a test an a handler

        assert(d->counter == NULL); // immediate delays shoudn't be counted
        assert(d->predicate);
        // FIXME: does the Test need this reference to the abort STref node?
        Test *tst = new Test( bodytree, clone(d->predicate) );
        assert(tst->st);
        *tst >> start
          // If the predicate has a body, send control there
             >> ( ((*i)->body) ? recurse((*i)->body)
                  : context(0) );
        start = tst;

      } else {
```

```
        // A counted predicate: add code that initializes the counter

        push_onto(start, new Action(new StartCounter(d->counter, bodytree)));

      }
    }
  }

  // Topmost node in the surface is an enter for the whole abort
  push_onto(start, new Enter(stnode(s)));

  context(0) = start;

  return Status();
}
```

The depth fragment is rooted at a switch node that selects among the depth of the body of the abort and any handlers. The depth for the "body" actually begins with tests for each of the predicates, which may include counter decrements, and branches to the surface of each of the handlers.

32 ⟨*depth method definitions* 17f⟩+≡ (51b) ◁27 36▷

```
Status Depth::visit(Abort &s) {
  if (s.is_weak) throw IR::Error("weak abort.  Did the dismantler run?");

  // Synthesize the depth of the body

  context.push();
  assert(s.body);
  GRCNode *resume = recurse(s.body);//
  context.pop();

  // The selection tree node for the body of the abort

  STNode *bodytree = stnode(s)->children.front();
  push_onto(resume, new Enter(bodytree));

  // Add a check for each predicate that branches to the surface of
  // the handler.  Also add the depth of each handler

  for ( vector<PredicatedStatement*>::reverse_iterator i = s.cases.rbegin() ;
        i != s.cases.rend() ; i++ ) {
    assert(*i);
    assert((*i)->predicate);

    // Get the predicate expression: either the simple predicate,
    // the predicate of an immediate, or a counter check for counted
    // predicates

    Delay *d = dynamic_cast<Delay*>((*i)->predicate);
    Expression *pred =
      d ?
      (d->is_immediate ?
       clone(d->predicate) : new CheckCounter(environment.boolean_type, d->counter))
      :  clone((*i)->predicate);

    // Add a test for the predicate

    Test *tst = new Test( bodytree, pred );
    GRCNode *handler;
    if ((*i)->body) {
      environment.surface_context.push(context);
      handler = environment.surface.synthesize((*i)->body);
      environment.surface_context.pop();
    } else handler = context(0);
```

```
    *tst >> resume >> handler;
    resume = tst;
  }

  // The switch at the top of the depth for the abort

  Switch *topswitch = new Switch( stnode(s) );

  // Its first child is the code for the body of the abort

  *topswitch >> resume;

  // Its remaining children are the bodies of the handlers

  for ( vector<PredicatedStatement*>::const_iterator i = s.cases.begin() ;
        i != s.cases.end() ; i++ ) {
    assert(*i);
    assert((*i)->predicate);
    if ((*i)->body) *topswitch >> recurse((*i)->body);
  }

  context(0) = topswitch;

  return Status();
}
```

33a ⟨*st methods* 17a⟩+≡ (16c) ◁28a 33d▷

```
Status visit(Abort &);
```

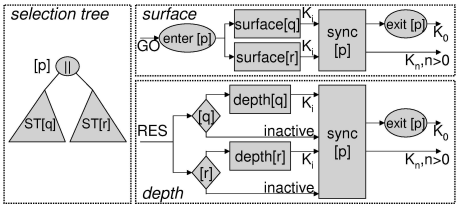
33b ⟨*surface methods* 17c⟩+≡ (16e) ◁28b 34b▷

```
Status visit(Abort &);
```

33c ⟨*depth methods* 17e⟩+≡ (16f) ◁28c 35b▷

```
Status visit(Abort &);
```

3.4.11 Parallel



33d ⟨*st methods* 17a⟩+≡ (16c) ◁33a 43a▷

```
Status visit(ParallelStatementList &);
```

34a ⟨*st method definitions* 16d⟩+≡ (51b) ◁29 38▷

```
Status SelTree::visit(ParallelStatementList &s)
{
  STpar *par = new STpar();
  setNode(s, par);

  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++ ) {
    assert(*i);
    STexcl *ex = new STexcl();
    *par >> ex;
    STleaf *term = new STleaf();
    term->setfinal();
    *ex >> term;
    *ex >> synthesize(*i);
  }

  return Status(par);
}
```

34b ⟨*surface methods* 17c⟩+≡ (16e) ◁33b 43b▷

```
Status visit(ParallelStatementList &);
```

35a ⟨*surface method definitions* 17d⟩+≡ (51b) ◁30 40▷

```
Status Surface::visit(ParallelStatementList &s) {
  Fork *fork = new Fork();
  Sync *sync = new Sync(stnode(s));
  Terminate *t;
  int nthr;

  GRCNode **outer = context.continuations.top();
  assert(outer);
  context.push();

  // Create a new terminate for every possible exit level
  // and link each from the sync node

  // Synthesize each thread's surface
  for ( vector<Statement*>::iterator i = s.threads.begin() ;i != s.threads.end() ; i++ ) {
    assert(*i);
    nthr=i-s.threads.begin();

    for(int tl=0; tl<context.size; tl++){
            t = new Terminate(tl, nthr);
            *t >> sync;
            context(tl)=t;

            if(tl == 0){ // this is the self looping enter
              Enter *en = new Enter( stnode(s)->children[nthr]->children[0] );
              assert(en->st);
              push_onto(context(tl), en);
            }
    }

    *fork >> recurse(*i); // it links thread to terminates, but each thread should have its own "enter"
  }

  // Connect each Terminate node with predecessors (i.e., that was
  // used by the threads) to the Sync and delete the rest.

  for ( int i = 0 ; i < context.size ; i++ ){
        *sync >> outer[i];
  }

  context.pop();
  context(0) = fork;
  push_onto(context(0), new Enter(stnode(s)));
  return Status();
}
```

35b ⟨*depth methods* 17e⟩+≡ (16f) ◁33c 43c▷

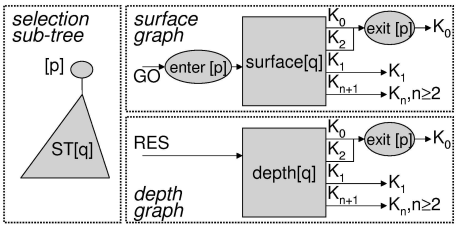
```
Status visit(ParallelStatementList &);
```

36 ⟨*depth method definitions* 17f⟩+≡ (51b) ◁32 42▷

```
Status Depth::visit(ParallelStatementList &s) {
  Fork *fork = new Fork();
  Sync *sync = new Sync(stnode(s));
  Enter *en;
  int nthr;
  Terminate *t;

  GRCNode **outer = context.continuations.top();
  assert(outer);
  context.push();

  // Create a new terminate for every possible exit level
  // and link each from the sync node


  // Synthesize each thread's surface
  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++) {
    assert(*i);
    nthr=i-s.threads.begin();
    Switch *sw = new Switch( stnode(s)->children[nthr] );
    assert(sw->st);
    *fork >> sw;

    for(int tl = 0; tl < context.size; tl++){
        t = new Terminate(tl, nthr);
        context(tl)=t;
        *t >> sync;
        if(tl == 0){
         // this is the self looping enter
          en=new Enter( stnode(s)->children[nthr]->children[0] );
          assert(en->st);
          *sw >> en;
          push_onto(context(tl), en);
        }
    }

   *sw >> recurse(*i);
  }

  // Connect each Terminate node with predecessors (i.e., that was
  // used by the threads) to the Sync and delete the rest.

  for ( int i = 0 ; i < context.size ; i++ )
        *sync >> outer[i];

  context.pop();
  context(0) = fork;
  push_onto(context(0), new Enter(stnode(s))); // hold
```

```
  return Status();
}
```

3.4.12 Trap



The selection tree fragment for a *trap* consists of an exclusive node whose children are the body of the *trap* followed by the body of the handler, if any. Multiple handlers should have been dismantled into a single handler by the dismantler. The body of the abort is an STref node whose sole child is the tree for the body of the abort.

38 ⟨*st method definitions* 16d⟩+≡ (51b) ◁34a 43d▷

```
Status SelTree::visit(Trap &s) {

  // Create the topmost exclusive node

  STexcl *exclusive = new STexcl();

  // Create the subtree for the body of the Trap; attach it to the top

  assert(s.body);
  STref *bodytree = new STref();
  *bodytree >> synthesize(s.body);
  *exclusive >> bodytree;

  // Create the subtree for the handler, if any

  switch (s.handlers.size()) {
  case 0:
    // No handler; nothing to do
    break;
  case 1:
    // Single handler: add the selection tree for it to the exclusive node
    assert(s.handlers.front());
    assert(s.handlers.front()->body);
    *exclusive >> synthesize(s.handlers.front()->body);
    break;
  default:
    // Esterel permits multiple handlers, but the dismantler should
    // have removed them
    throw IR::Error("Multiple trap handler.  Did the dismantler run?");
    break;
  }

  setNode(s, exclusive);
```

```
  return Status(exclusive);
}
```

The surface for a Trap consists of two enter nodes (one for the trap as a whole, the other for the body). Between the two enters are DefineSignal nodes for each of the traps. After the two enters follows the surface for the body of the trap followed by the surface of the handler, if any, connected through the exit level of the trap.

40 ⟨*surface method definitions* 17d⟩+≡ (51b) ◁35a 44a▷

```
Status Surface::visit(Trap &s) {

  assert(s.symbols);
  assert(s.symbols->begin() != s.symbols->end());
  SignalSymbol *ts = dynamic_cast<SignalSymbol*>((*(s.symbols->begin())));
  assert(ts);

  for (SymbolTable::const_iterator i = s.symbols->begin() ;
       i != s.symbols->end() ; i++ ) {
    SignalSymbol *ss = dynamic_cast<SignalSymbol*>(*i);
    assert(ss);
    assert(ss->kind == SignalSymbol::Trap);
    clone.cloneLocalSignal(ss, environment.module->signals);
  }

  int level = environment.code[ts];
  assert(level > 1); // Should have been assigned by the dismantler

  // Esterel permits multiple handlers, but the dismantler should
  // have removed them
  if (s.handlers.size() > 1)
    throw IR::Error("Multiple trap handler.  Did the dismantler run?");

  // Surface for the handler is either a normal termination or
  // the handler for the body
  assert( s.handlers.empty() || s.handlers.front() );
  GRCNode *handlerSurface =
     s.handlers.empty() ? context(0) : recurse(s.handlers.front()->body);

  // Synthesize the body
  context.push();

  assert(handlerSurface);
  context(level) = handlerSurface;

  assert(s.body);
  GRCNode *surface = synthesize(s.body);

  context.pop();

  assert(stnode(s));
  assert(stnode(s)->children.front());
  push_onto(surface, new Enter(stnode(s)->children.front()));
```

```
  // Add "DefineSignal" nodes for each of the traps

  for (SymbolTable::const_iterator i = s.symbols->begin() ;
       i != s.symbols->end() ; i++ ) {
    SignalSymbol *ss = dynamic_cast<SignalSymbol*>(*i);
    assert(ss);
    assert(ss->kind == SignalSymbol::Trap);
    push_onto(surface, new DefineSignal(clone(ss)));
    clone.clearSig(ss);
  }

  push_onto(surface, new Enter(stnode(s)));

  context(0) = surface;

  return Status();
}
```

The depth of a trap consists of a switch that decides between the depth of the body or the depth of the handler. The depth of the body is connected to another copy of the surface of the handler at the appropriate exit level.

42 ⟨*depth method definitions* 17f⟩+≡ (51b) ◁36 44b▷

```
Status Depth::visit(Trap &s) {

  assert(s.symbols);
  assert(s.symbols->begin() != s.symbols->end());
  SignalSymbol *ts = dynamic_cast<SignalSymbol*>((*(s.symbols->begin())));
  assert(ts);
  int level = environment.code[ts];
  assert(level > 1); // Should have been assigned by the dismantler

  // Clone each of the trap signals

  for (SymbolTable::const_iterator i = s.symbols->begin() ;
       i != s.symbols->end() ; i++ ) {
    SignalSymbol *ss = dynamic_cast<SignalSymbol*>(*i);
    assert(ss);
    assert(ss->kind == SignalSymbol::Trap);
    clone.cloneLocalSignal(ss, environment.module->signals);
  }

  // Esterel permits multiple handlers, but the dismantler should
  // have removed them
  if (s.handlers.size() > 1)
    throw IR::Error("Multiple trap handler.  Did the dismantler run?");

  // Surface for the handler is either a normal termination or
  // the handler for the body
  assert( s.handlers.empty() || s.handlers.front() );
  GRCNode *handlerSurface = context(0);
  if (!s.handlers.empty()) {
    environment.surface_context.push(context);
    handlerSurface = environment.surface.synthesize(s.handlers.front()->body);
    environment.surface_context.pop();
  }

  GRCNode *handlerDepth =
     s.handlers.empty() ? 0 : recurse(s.handlers.front()->body);

  // Synthesize the body
  context.push();

  assert(handlerSurface);
  context(level) = handlerSurface;

  assert(s.body);
  GRCNode *depth = synthesize(s.body);
```

```
    context.pop();

    Switch *topswitch = new Switch( stnode(s) );
    *topswitch >> depth;

    if (handlerDepth) *topswitch >> handlerDepth;

    // Delete the mapping for each of the traps
    for (SymbolTable::const_iterator i = s.symbols->begin() ;
         i != s.symbols->end() ; i++ ) {
      SignalSymbol *ss = dynamic_cast<SignalSymbol*>(*i);
      assert(ss);
      assert(ss->kind == SignalSymbol::Trap);
      clone.clearSig(ss);
    }

    context(0) = topswitch;
    return Status();
  }
```

43a ⟨*st methods* 17a⟩+≡ (16c) ◁33d 44c▷

```
  Status visit(Trap &);
```

43b ⟨*surface methods* 17c⟩+≡ (16e) ◁34b 44d▷

```
  Status visit(Trap &);
```

43c ⟨*depth methods* 17e⟩+≡ (16f) ◁35b 44e▷

```
  Status visit(Trap &);
```

3.4.13 Signal

Signal statements introduce a new scope for signals. Both the surface and the depth start with DefineSignal nodes that reset to absent all of the new local signals.

43d ⟨*st method definitions* 16d⟩+≡ (51b) ◁38 45a▷

```
  Status SelTree::visit(Signal &s) {
    STNode *st = new STref();
    *st >> synthesize(s.body);
    return Status(st);
  }
```

44a ⟨*surface method definitions* 17d⟩+≡ (51b) ◁40 45b▷

```
Status Surface::visit(Signal &s) {
  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i != s.symbols->end() ; i++ ) {
    SignalSymbol *sig = dynamic_cast<SignalSymbol*>(*i);
    assert(sig);
    clone.cloneLocalSignal(sig, environment.module->signals);
  }

  context(0) = synthesize(s.body);

  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i!= s.symbols->end() ; i++ ) {
    SignalSymbol *sig = dynamic_cast<SignalSymbol*>(*i);
    assert(sig);
    push_onto(context(0), new DefineSignal(clone(sig)));
    clone.clearSig(sig);
  }
  return Status();
}
```

44b ⟨*depth method definitions* 17f⟩+≡ (51b) ◁42 45c▷

```
Status Depth::visit(Signal &s) {
  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i != s.symbols->end() ; i++ ) {
    SignalSymbol *sig = dynamic_cast<SignalSymbol*>(*i);
    assert(sig);
    clone.cloneLocalSignal(sig, environment.module->signals);
  }

  context(0) = synthesize(s.body);
   for ( SymbolTable::const_iterator i = s.symbols->begin() ;
         i!= s.symbols->end() ; i++ ) {
     SignalSymbol *sig = dynamic_cast<SignalSymbol*>(*i);
     assert(sig);
     push_onto(context(0), new DefineSignal(clone(sig)));
     clone.clearSig(sig);
  }
  return Status();
}
```

44c ⟨*st methods* 17a⟩+≡ (16c) ◁43a 45d▷

```
Status visit(Signal &);
```

44d ⟨*surface methods* 17c⟩+≡ (16e) ◁43b 45e▷

```
Status visit(Signal &);
```

44e ⟨*depth methods* 17e⟩+≡ (16f) ◁43c 45f▷

```
Status visit(Signal &);
```

3.4.14 Var

The *var* statement introduces a scope for new local variables. Its translation is trivial.

45a ⟨*st method definitions* 16d⟩+≡ (51b) ◁43d
```
Status SelTree::visit(Var &s) {
  STNode *st = new STref();
  *st >> synthesize(s.body);
  return Status(st);
}
```

45b ⟨*surface method definitions* 17d⟩+≡ (51b) ◁44a
```
Status Surface::visit(Var &s) {
  context(0) = synthesize(s.body);
  return Status();
}
```

45c ⟨*depth method definitions* 17f⟩+≡ (51b) ◁44b
```
Status Depth::visit(Var &s) {
  context(0) = synthesize(s.body);
  return Status();
}
```

45d ⟨*st methods* 17a⟩+≡ (16c) ◁44c 45g▷
```
Status visit(Var &);
```

45e ⟨*surface methods* 17c⟩+≡ (16e) ◁44d 45h▷
```
Status visit(Var &);
```

45f ⟨*depth methods* 17e⟩+≡ (16f) ◁44e 45i▷
```
Status visit(Var &);
```

3.5 Unimplemented statements

3.5.1 Exec

45g ⟨*st methods* 17a⟩+≡ (16c) ◁45d 45j▷
```
Status visit(Exec &) { return Status(new STref()); }
```

45h ⟨*surface methods* 17c⟩+≡ (16e) ◁45e 46a▷
```
Status visit(Exec &) { return Status(); }
```

45i ⟨*depth methods* 17e⟩+≡ (16f) ◁45f 46b▷
```
Status visit(Exec &) { return Status(); }
```

3.5.2 Procedure Call

45j ⟨*st methods* 17a⟩+≡ (16c) ◁45g
```
Status visit(ProcedureCall &) { return Status(new STref()); }
```

46a ⟨*surface methods* 17c⟩+≡ (16e) ◁45h
```
Status visit(ProcedureCall &) { return Status(); }
```

46b ⟨*depth methods* 17e⟩+≡ (16f) ◁45i
```
Status visit(ProcedureCall &) { return Status(); }
```

4 Signal Dependency Calculator Class

The GRC synthesis class produces a control-flow graph only. This class annotates it with two types of dependencies: those between signal emissions and tests, and those from terminate nodes to their sync node successors.

46c ⟨*dependency class* 46c⟩≡ (51a)
```
class Dependencies : public Visitor {
protected:
  set<GRCNode *> visited;
  GRCNode *current;

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

  map<SignalSymbol *, SignalNodes> dependencies;

  ⟨dependency methods 47c⟩
  Dependencies() {}
  virtual ~Dependencies() {}
public:
  static void compute(GRCNode *);
};
```

47a ⟨*dependency method definitions* 47a⟩≡ (51b) 47b▷

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

4.1 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.

47b ⟨*dependency method definitions* 47a⟩+≡ (51b) ◁47a 48a▷

```
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);
}
```

47c ⟨*dependency methods* 47c⟩≡ (46c) 48b▷

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

4.2 Action

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

48a ⟨*dependency method definitions* 47a⟩+≡ (51b) ◁47b 50a▷

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

48b ⟨*dependency methods* 47c⟩+≡ (46c) ◁47c 48c▷

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

4.3 DefineSignal

This is an "unemit" for a signal and therefore a writer.

48c ⟨*dependency methods* 47c⟩+≡ (46c) ◁48b 48d▷

```
Status visit(DefineSignal &d) {
  dependencies[d.signal].writers.insert(current);
  return Status();
}
```

4.4 Test

This descends down its predicate, possibly adding signal testers

48d ⟨*dependency methods* 47c⟩+≡ (46c) ◁48c 49a▷

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

4.5 Expressions

49a ⟨*dependency methods* 47c⟩+≡ (46c) ◁48d 49b▷

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

Status visit(LoadSignalValueExpression &e) {
  dependencies[e.signal].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.counter->predicate->welcome(*this);
  return Status();
}

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

4.5.1 Vacuous Expression Nodes

49b ⟨*dependency methods* 47c⟩+≡ (46c) ◁49a 50b▷

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

4.6 Sync

A terminate node is ignored, but a sync node connects a dependency from each of its predecessors, all of which much be terminate nodes.

50a ⟨*dependency method definitions* 47a⟩+≡ (51b) ◁48a

```
Status Dependencies::visit(Sync &s)
{
  for ( vector<GRCNode*>::const_iterator i = s.predecessors.begin() ;
        i != s.predecessors.end() ; i++ ) {
    // Every predecessor should be a terminate node
    assert( dynamic_cast<Terminate*>(*i) );
    s << *i;
  }
  return Status();
}
```

50b ⟨*dependency methods* 47c⟩+≡ (46c) ◁49b 50c▷

```
Status visit(Sync &);
```

4.7 Trivial visitors

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

50c ⟨*dependency methods* 47c⟩+≡ (46c) ◁50b

```
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(); }
```

5 ASTGRC.hpp and .cpp

51a ⟨*ASTGRC.hpp* 51a⟩≡

```
#ifndef _ASTGRC_HPP
#  define _ASTGRC_HPP

#  include "AST.hpp"
#  include <assert.h>
#  include <stack>
#  include <map>
#  include <set>

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

  class GrcSynth;

  ⟨completion code class 2⟩

  ⟨cloner class 6a⟩

  ⟨context class 10⟩
  ⟨grc walker class 15⟩
  ⟨surface class 16e⟩
  ⟨depth class 16f⟩
  ⟨st class 16c⟩
  ⟨GrcSynth class 12a⟩

  ⟨dependency class 46c⟩
}
#endif
```

51b ⟨*ASTGRC.cpp* 51b⟩≡

```
#include "ASTGRC.hpp"
#include <cstdio>

namespace ASTGRC {
  ⟨grc synth method definitions 13⟩
  ⟨grc walker methods 16a⟩
  ⟨surface method definitions 17d⟩
  ⟨depth method definitions 17f⟩
  ⟨st method definitions 16d⟩
  ⟨dependency method definitions 47a⟩
}
```

52 ⟨*cec-astgrc.cpp* 52⟩≡

```
#include "IR.hpp"
#include "AST.hpp"
#include "ASTGRC.hpp"
#include <iostream>
#include <vector>

int main(int argc, char *argv[])
{
  try {
    IR::Node *root;
    IR::XMListream r(std::cin);
    r >> root;

    AST::Modules *mods = dynamic_cast<AST::Modules*>(root);
    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);
      // Compute completion codes for this module
      ASTGRC::CompletionCodes cc(*i);
      // Synthesize GRC for this module and replace it
      ASTGRC::GrcSynth synth(*i, cc);
      (*i)->body = synth.synthesize();
      assert((*i)->body);

      AST::GRCgraph *g = dynamic_cast<AST::GRCgraph *>((*i)->body);
      assert(g);
      assert(g->control_flow_graph);

      // Add dependencies
      ASTGRC::Dependencies::compute(g->control_flow_graph);
    }

    IR::XMLostream w(std::cout);
    w << mods;

  } catch (IR::Error &e) {
    std::cerr << e.s << std::endl;
    return -1;
  }

  return 0;
}
```