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.

3a ⟨*completion code class* 3a⟩≡

```
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 3b⟩
};
```

The constructor takes a module and computes its completion codes.

3b ⟨*completion code methods* 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.

4a ⟨*completion code methods* 3b⟩+≡
```
  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

4b ⟨*completion code methods* 3b⟩+≡
```
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(Every &s) { return recurse(s.body); }
Status visit(Suspend &s) { return recurse(s.body); }
Status visit(PredicatedStatement &s) { return recurse(s.body); }
```

4c ⟨*completion code methods* 3b⟩+≡
```
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;
}
```

4d ⟨*completion code methods* 3b⟩+≡
```
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;
}
```

5a ⟨*completion code methods* 3b⟩+≡
```
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;

5b ⟨*completion code methods* 3b⟩+≡
```
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.

5c ⟨*completion code methods* 3b⟩+≡
```
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.

6 ⟨*completion code methods* 3b⟩+≡

```
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. It also handles local signal cloning for reincarnation.

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.

7 ⟨*cloner class* 7⟩≡

```
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 11a⟩

  virtual ~Cloner() {}

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

  /* For each signal symbol in the AST's symbol tables,
     the master signal symbol in the expanded graph.  Used to
     set up the reincarnation field of the cloned signals. */
  map<SignalSymbol *, SignalSymbol *> master_signal;

  ⟨cloner methods 8a⟩
};
```

2.1 Statments: Emit, Assign, and Exit

8a ⟨*cloner methods* 8a⟩≡
```
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* 8a⟩+≡
```
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* 8a⟩+≡
```
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* 8a⟩+≡

```
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 Procedure Call

9b ⟨*cloner methods* 8a⟩+≡

```
Status visit(ProcedureCall &s) {
  ProcedureCall *c = new ProcedureCall(clone(s.procedure));
  for (vector<VariableSymbol*>::const_iterator i = s.reference_args.begin() ;
       i != s.reference_args.end() ; i++) {
    assert(*i);
    c->reference_args.push_back(clone(*i));
  }
  for (vector<Expression*>::const_iterator i = s.value_args.begin() ;
       i != s.value_args.end() ; i++) {
    assert(*i);
    c->value_args.push_back(clone(*i));
  }

  return c;
}
```

2.6 CheckCounter

9c ⟨*cloner methods* 8a⟩+≡

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

2.7 Symbols

These do not clone anything, just return themselves.

10a ⟨*cloner methods* 8a⟩+≡
```
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(ProcedureSymbol &s) { return &s; }
Status visit(BuiltinFunctionSymbol &s) { return &s; }
Status visit(Counter &s) { return &s; }
```

2.8 Local Signal Renaming

The following map and methods manage renaming local signals.

10b ⟨*cloner methods* 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.

11a ⟨*public cloner methods* 11a⟩≡
```
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 *reincarnation = 0;
  if (master_signal.find(s) != master_signal.end()) {
    reincarnation = master_signal[s];
    assert(reincarnation);
  }
  SignalSymbol *result =
    new SignalSymbol(name, s->type, kind, clone(s->combine),
                     clone(s->initializer), reincarnation);
  if (!reincarnation)
    master_signal[s] = result;
  assert(master_signal.find(s) != master_signal.end());
  // std::cerr << "cloning " << s->name << std::endl;
  st->enter(result);
  newsig[s] = result;
  return result;
}
```

Set a signal to map to itself.

11b ⟨*public cloner methods* 11a⟩+≡
```
void sameSig(SignalSymbol *s) {
  assert(s);
  assert(newsig.find(s) == newsig.end());
  newsig[s] = s;
}
```

`clearSig` deletes a previously-established signal mapping.

11c ⟨*public cloner methods* 11a⟩+≡
```
void clearSig(SignalSymbol *orig) {
  assert(orig);
  map<SignalSymbol*, SignalSymbol*>::iterator i = newsig.find(orig);
  assert(i != newsig.end());
  newsig.erase(i);
}
```

2.9 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.

12a ⟨*cloner methods* 8a⟩+≡
```
Status visit(SignalSymbol &s) {
  assert(newsig.find(&s) != newsig.end()); // should be there
  return newsig[&s];
}
```

2.10 Local Variable Renaming

The following are responsible for hoisting local variables from their local symbol tables up to the topmost one and renaming them if necessary.

12b ⟨*public cloner methods* 11a⟩+≡
```
map<VariableSymbol*, VariableSymbol*> newvar;
```

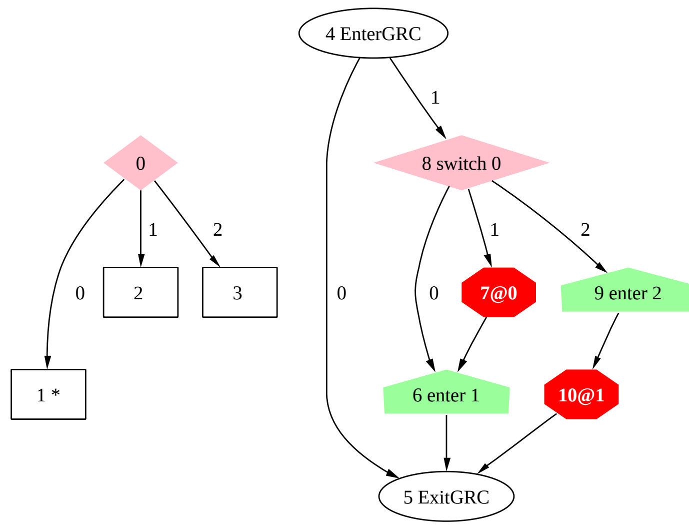
12c ⟨*public cloner methods* 11a⟩+≡
```
VariableSymbol *hoistLocalVariable(VariableSymbol *s, SymbolTable *st) {
  assert(s);
  assert(newvar.find(s) == newvar.end()); // should not be remapped yet
  assert(st);

  // Add a suffix to the name to make it unique, if necessary

  string name = s->name;
  int next = 1;
  while (st->contains(name)) {
    char buf[10];
    sprintf(buf, "%d", next++);
    name = s->name + '_' + buf;
  }

  VariableSymbol *result =
    new VariableSymbol(name, s->type, clone(s->initializer));
  st->enter(result);
  newvar[s] = result;
  return result;
}
```

12d ⟨*public cloner methods* 11a⟩+≡
```
void sameVar(VariableSymbol *s) {
  assert(s);
  assert(newvar.find(s) == newvar.end());
  newvar[s] = s;
}
```




```
module onepause:

pause

end module
```

(a) (b)

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

2.11 VariableSymbols

13 ⟨*cloner methods* 8a⟩+≡

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

3 Duplicating GRC Synthesis

This algorithm closely follows the rules described in Potop's thesis. Surfaces are frequently duplicated to remove any threat of schizophrenia.

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.)

14 ⟨*context class* 14⟩≡
```
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.

15a ⟨*GrcSynth class* 15a⟩≡
```
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;
  BuiltinConstantSymbol *true_symbol;

  ⟨GrcSynth methods 15b⟩
};
```

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

15b ⟨*GrcSynth methods* 15b⟩≡
```
GrcSynth(Module *, CompletionCodes &);
```

16 ⟨*grc synth method definitions* 16⟩≡

```
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);
  true_symbol = dynamic_cast<BuiltinConstantSymbol*>(m->constants->get("true"));
  assert(true_symbol);

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

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

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

17 ⟨*GrcSynth methods* 15b⟩+≡
```
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

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

  // Add DefineSignal statements for every output signal.  This clears
  // their presence and initializes their values if an initializer was given
```

```
  for (SymbolTable::const_iterator i = module->signals->begin() ;
       i != module->signals->end() ; i++) {
    SignalSymbol *ss = dynamic_cast<SignalSymbol*>(*i);
    assert(ss);
    if (ss->kind == SignalSymbol::Output) {
      DefineSignal *ds = new DefineSignal(ss, true);
      *ds >> synt_surface;
      synt_surface = ds;
    }
  }

  *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.

19a ⟨*grc walker class* 19a⟩≡
```
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 &);
};
```

19b ⟨*grc walker methods* 19b⟩≡
```
GrcWalker::GrcWalker(Context &c, GrcSynth &e)
    : context(c), environment(e), clone(e.clone) {}
```

20a ⟨*grc walker methods* 19b⟩+≡
```
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.

20b ⟨*st class* 20b⟩≡
```
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 21a⟩
};
```

20c ⟨*st method definitions* 20c⟩≡
```
void SelTree::setNode(const ASTNode &n, STNode *sn) {
  assert(sn);
  environment.ast2st[&n] = sn;
}
```

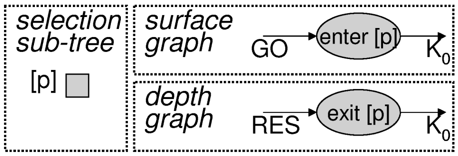
20d ⟨*surface class* 20d⟩≡
```
class Surface : public GrcWalker {
public:
  Surface(Context &c, GrcSynth &e) : GrcWalker(c, e) {}
  ⟨surface methods 21c⟩
};
```

20e ⟨*depth class* 20e⟩≡
```
class Depth : public GrcWalker {
public:
  Depth(Context &c, GrcSynth &e) : GrcWalker(c, e) {}
  ⟨depth methods 21e⟩
};
```

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.

21a ⟨*st methods* 21a⟩≡
```
Status visit(Pause &);
```

21b ⟨*st method definitions* 20c⟩+≡
```
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.

21c ⟨*surface methods* 21c⟩≡
```
Status visit(Pause &);
```

21d ⟨*surface method definitions* 21d⟩≡
```
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.

21e ⟨*depth methods* 21e⟩≡
```
Status visit(Pause &);
```

21f ⟨*depth method definitions* 21f⟩≡
```
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.

21g ⟨*st methods* 21a⟩+≡
```
Status visit(Exit &s) {
  return Status(new STref());
}
```

22a ⟨*surface methods* 21c⟩+≡
```
Status visit(Exit &);
```

22b ⟨*surface method definitions* 21d⟩+≡
```
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();
}
```

22c ⟨*depth methods* 21e⟩+≡
```
Status visit(Exit &) { return Status(); }
```

3.4.3 Emit

22d ⟨*st methods* 21a⟩+≡
```
Status visit(Emit &) {
  return Status(new STref());
}
```

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

22e ⟨*surface methods* 21c⟩+≡
```
Status visit(Emit &s) {
  push_onto(context(0), new Action(clone(&s)));
  return Status();
}
```

22f ⟨*depth methods* 21e⟩+≡
```
Status visit(Emit &) { return Status(); }
```

3.4.4 Assign

This also becomes an action with a vacuous depth.

22g ⟨*st methods* 21a⟩+≡
```
Status visit(Assign &s) {
  return Status(new STref());
}
```

22h ⟨*surface methods* 21c⟩+≡
```
Status visit(Assign &s) {
  Action *a = new Action(clone(&s));
  *a >> context(0);
  context(0) = a;
  return Status();
}
```

22i ⟨*depth methods* 21e⟩+≡
```
Status visit(Assign &) { return Status(); }
```

3.4.5 IfThenElse

23a ⟨*st methods* 21a⟩+≡
```
Status visit(IfThenElse &);
```

23b ⟨*st method definitions* 20c⟩+≡
```
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.

23c ⟨*surface methods* 21c⟩+≡
```
Status visit(IfThenElse &);
```

23d ⟨*surface method definitions* 21d⟩+≡
```
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.

23e ⟨*depth methods* 21e⟩+≡
```
Status visit(IfThenElse &);
```

23f ⟨*depth method definitions* 21f⟩+≡
```
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.

24a ⟨*st methods* 21a⟩+≡
```
Status visit(StatementList &s);
```

24b ⟨*st method definitions* 20c⟩+≡
```
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);
}
```

24c ⟨*surface methods* 21c⟩+≡
```
Status visit(StatementList &);
```

24d ⟨*surface method definitions* 21d⟩+≡
```
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();
}
```

24e ⟨*depth methods* 21e⟩+≡
```
Status visit(StatementList &);
```

Can be optimized to remove dead code (?)

25a ⟨*depth method definitions* 21f⟩+≡
```
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.

25b ⟨*st methods* 21a⟩+≡
```
Status visit(Loop &s);
```

25c ⟨*st method definitions* 20c⟩+≡
```
Status SelTree::visit(Loop &s) {
  STref *lp = new STref();
  setNode(s, lp);
  *lp >> synthesize(s.body);
  return Status(lp);
}
```

25d ⟨*surface methods* 21c⟩+≡
```
Status visit(Loop &);
```

25e ⟨*surface method definitions* 21d⟩+≡
```
Status Surface::visit(Loop &s) {
  context(0) = synthesize(s.body);
  Enter *en = new Enter(stnode(s));
  push_onto(context(0), en);
  return Status();
}
```

25f ⟨*depth methods* 21e⟩+≡
```
Status visit(Loop &);
```

26a ⟨*depth method definitions* 21f⟩+≡
```
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 Every

26b ⟨*st methods* 21a⟩+≡
```
Status visit(Every &);
```

26c ⟨*st method definitions* 20c⟩+≡
```
Status SelTree::visit(Every &s) {

  STref *ab = new STref(); ab->setabort(); setNode(s, ab);
  STexcl *excl = new STexcl();
  STleaf *halt = new STleaf();

  *excl >> halt >> synthesize(s.body);
  *ab >> excl;

  return Status(ab);

}
```

26d ⟨*surface method definitions* 21c⟩+≡
```
Status visit(Every &);
```

27a ⟨*surface method definitions* 21d⟩+≡
```
Status Surface::visit(Every &s) {

  STNode *halt = stnode(s)->children[0]->children[0];
  Enter *enhalt = new Enter(halt);
  GRCNode *start;

  *enhalt >> context(1);

  Delay *d = dynamic_cast<Delay*>(s.predicate);
  if (d) {
    if (d->is_immediate) {
      context(0) = enhalt;
      Test *tst = new Test(stnode(s), clone(d->predicate));
      *tst >> enhalt >> synthesize(s.body);
      start = tst;

    } else {
      // A counted every
      assert(d->counter);
      StartCounter *scnt = new StartCounter(d->counter, clone(d->count));
      start = new Action(scnt);
      *start >> enhalt;
    }
  } else start = enhalt;

  push_onto(start, new Enter(stnode(s)));
  context(0) = start;

  return Status();
}
```

27b ⟨*depth methods* 21e⟩+≡
```
Status visit(Every &);
```

28a ⟨*depth method definitions* 21f⟩+≡
```
Status Depth::visit(Every &s) {

  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(d->predicate))
      ) :
    clone(s.predicate);

   Test *tst = new Test(stnode(s), pred);
   Enter *enhalt = new Enter(stnode(s)->children[0]->children[0]);
   *enhalt >> context(1);
   context(0) = enhalt;

   Switch *sw = new Switch(stnode(s)->children[0]);
   *sw >> enhalt >> recurse(s.body);
   *tst >> sw;

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

   if(d && !d->is_immediate) {
      assert(d->counter);
      push_onto(restart, new Action(new StartCounter(d->counter,
                                                     clone(d->count))));
   }
   *tst >> restart;

   Enter *hold = new Enter(stnode(s));
   *hold >> tst;
   context(0) = hold;

  return Status();
}
```

3.4.9 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)

28b ⟨*st methods* 21a⟩+≡
```
Status visit(Repeat &);
```

29a ⟨*st method definitions* 20c⟩+≡
```
Status SelTree::visit(Repeat &s) {
  STref *lp = new STref();
  setNode(s, lp);
  *lp >> synthesize(s.body);
  return Status(lp);
}
```

29b ⟨*surface methods* 21c⟩+≡
```
Status visit(Repeat &);
```

29c ⟨*surface method definitions* 21d⟩+≡
```
Status Surface::visit(Repeat &s) {
  context(0) = synthesize(s.body);
  assert(s.counter);
  StartCounter *stcnt = new StartCounter(s.counter, clone(s.count));
  push_onto(context(0), new Action(stcnt));
  Enter *en = new Enter(stnode(s));
  push_onto(context(0), en);
  return Status();
}
```

29d ⟨*depth methods* 21e⟩+≡
```
Status visit(Repeat &);
```

29e ⟨*depth method definitions* 21f⟩+≡
```
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,
         new LoadVariableExpression(environment.true_symbol)));
  *tst >> restart >> context(0);
  //Synthesize the depth
  context(0) = tst;
  context(0) = synthesize(s.body);

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

  return Status();
}
```

3.4.10 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.

30 ⟨*st method definitions* 20c⟩+≡
```
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);
}
```

31 ⟨*surface method definitions* 21d⟩+≡

```
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)
      assert(d->counter);
      StartCounter *scnt = new StartCounter(d->counter, clone(d->count));
      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();
}
```

32 ⟨*depth method definitions* 21f⟩+≡

```
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(d->predicate))
      ) :
    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));
    environment.surface_context.push(context);
    start = environment.surface.synthesize(s.body);
    environment.surface_context.pop();
```

```
      push_onto(start, en);
      *t >> start >>enimleaf;
      *swimm >> t;
    }

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

33a ⟨*st methods* 21a⟩+≡

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

33b ⟨*surface methods* 21c⟩+≡

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

33c ⟨*depth methods* 21e⟩+≡

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

3.4.11 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.

34 ⟨*st method definitions* 20c⟩+≡
```
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.

35 ⟨*surface method definitions* 21d⟩+≡
```
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

        assert(d->counter);
        push_onto(start, new Action(new StartCounter(d->counter,
                                                     clone(d->count))));

      }
    }
  }

  // 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.

37 ⟨*depth method definitions* 21f⟩+≡

```
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(d->predicate)))
      :  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();
  }
```

38a ⟨*st methods* 21a⟩+≡

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

38b ⟨*surface methods* 21c⟩+≡

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

38c ⟨*depth methods* 21e⟩+≡

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

3.4.12 Parallel

38d ⟨*st methods* 21a⟩+≡

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

39a ⟨*st method definitions* 20c⟩+≡

```
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);
}
```

39b ⟨*surface methods* 21c⟩+≡

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

40a ⟨*surface method definitions* 21d⟩+≡

```
Status Surface::visit(ParallelStatementList &s) {
  Sync *sync = new Sync(stnode(s));
  Fork *fork = new Fork(sync);
  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 != 1) {
              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 ow
  }

  // Connect the sync with outer context nodes

  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();
}
```

40b ⟨*depth methods* 21e⟩+≡

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

41 ⟨*depth method definitions* 21f⟩+≡

```
Status Depth::visit(ParallelStatementList &s) {
  Sync *sync = new Sync(stnode(s));
  Fork *fork = new Fork(sync);
  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;
        // this is the self looping enter
        if(tl == 0){
          en=new Enter( stnode(s)->children[nthr]->children[0] );
          assert(en->st);
          push_onto(context(tl), en);
          *sw >> en;
        }
    }

   *sw >> recurse(*i);
  }

  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();
}
```

3.4.13 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.

42 ⟨*st method definitions* 20c⟩+≡
```
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.

43 ⟨*surface method definitions* 21d⟩+≡

```
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 completion codes class

  // 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), true));
    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.

45 ⟨*depth method definitions* 21f⟩+≡
```
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();
  }
```

46a ⟨*st methods* 21a⟩+≡

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

46b ⟨*surface methods* 21c⟩+≡

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

46c ⟨*depth methods* 21e⟩+≡

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

3.4.14 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.

46d ⟨*st method definitions* 20c⟩+≡

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

47a ⟨*surface method definitions* 21d⟩+≡
```
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), true));
    clone.clearSig(sig);
  }
  return Status();
}
```

47b ⟨*depth method definitions* 21f⟩+≡
```
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), false));
     clone.clearSig(sig);
  }
  return Status();
}
```

47c ⟨*st methods* 21a⟩+≡
```
Status visit(Signal &);
```

47d ⟨*surface methods* 21c⟩+≡
```
Status visit(Signal &);
```

47e ⟨*depth methods* 21e⟩+≡
```
Status visit(Signal &);
```

3.4.15 Var

The *var* statement introduces a scope for new local variables. It hoists the local variables to the topmost scope.

48a ⟨*st method definitions* 20c⟩+≡
```
Status SelTree::visit(Var &s) {
  STNode *st = new STref();
  *st >> synthesize(s.body);
  return Status(st);
}
```

48b ⟨*surface method definitions* 21d⟩+≡
```
Status Surface::visit(Var &s) {
  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i != s.symbols->end() ; i++ ) {
    VariableSymbol *vs = dynamic_cast<VariableSymbol*>(*i);
    assert(vs);
    clone.hoistLocalVariable(vs, environment.module->variables);
  }
  s.symbols->clear(); // Make sure we do not do this again

  context(0) = synthesize(s.body);
  return Status();
}
```

48c ⟨*depth method definitions* 21f⟩+≡
```
Status Depth::visit(Var &s) {
  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i != s.symbols->end() ; i++ ) {
    VariableSymbol *vs = dynamic_cast<VariableSymbol*>(*i);
    assert(vs);
    clone.hoistLocalVariable(vs, environment.module->variables);
  }
  s.symbols->clear(); // Make sure we do not do this again

  context(0) = synthesize(s.body);
  return Status();
}
```

48d ⟨*st methods* 21a⟩+≡
```
Status visit(Var &);
```

48e ⟨*surface methods* 21c⟩+≡
```
Status visit(Var &);
```

48f ⟨*depth methods* 21e⟩+≡
```
Status visit(Var &);
```

3.5 Unimplemented statements

3.5.1 Exec

49a ⟨*st methods* 21a⟩+≡
```
Status visit(Exec &) { return Status(new STref()); }
```

49b ⟨*surface methods* 21c⟩+≡
```
Status visit(Exec &) { return Status(); }
```

49c ⟨*depth methods* 21e⟩+≡
```
Status visit(Exec &) { return Status(); }
```

3.5.2 Procedure Call

49d ⟨*st methods* 21a⟩+≡
```
Status visit(ProcedureCall &) { return Status(new STref()); }
```

49e ⟨*surface methods* 21c⟩+≡
```
Status visit(ProcedureCall &s) {
  push_onto(context(0), new Action(clone(&s)));
  return Status();
}
```

49f ⟨*depth methods* 21e⟩+≡
```
Status visit(ProcedureCall &) { return Status(); }
```

4 Non-duplicating GRC Synthesis

This assumes schizophrenia has been dealt with earlier. No surfaces are duplicated. The synthesis procedure is now a simple recursive walk.

50a ⟨*RecursiveSynth class* 50a⟩≡
```
class RecursiveSynth : public Visitor {
public:
  Module *module;
  CompletionCodes &code;

  Cloner clone;

  Context context;

  BuiltinTypeSymbol *boolean_type;
  BuiltinConstantSymbol *true_symbol;

  // The visitors set these pointers when they return

  STNode *stnode;
  GRCNode *surface;
  GRCNode *depth;

  void synthesize(ASTNode *n) {
    assert(n);

    stnode = NULL; // Assignments not strictly necessary: for safety
    surface = depth = NULL;

    n->welcome(*this);
    assert(stnode);
    assert(surface);
    assert(depth);
  }

  // Run n then b, replacing b
  static void run_before(GRCNode *& b, GRCNode *n) { *n >> b; b = n; }

  ⟨RecursiveSynth declarations 50b⟩
  virtual ~RecursiveSynth() {}
};
```

50b ⟨*RecursiveSynth declarations* 50b⟩≡
```
RecursiveSynth(Module *, CompletionCodes &);
```

51a ⟨*RecursiveSynth definitions* 51a⟩≡

```
RecursiveSynth::RecursiveSynth(Module *m, CompletionCodes &c)
  : module(m), code(c), context(code.max() + 1) {
  assert(m);

  assert(m->types);
  boolean_type = dynamic_cast<BuiltinTypeSymbol*>(m->types->get("boolean"));
  assert(boolean_type);
  true_symbol = dynamic_cast<BuiltinConstantSymbol*>(m->constants->get("true"));
  assert(true_symbol);


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

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

51b ⟨*RecursiveSynth declarations* 50b⟩+≡

```
GRCgraph *synthesize();
```

52 ⟨*RecursiveSynth definitions* 51a⟩+≡

```
GRCgraph *RecursiveSynth::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;

  context(0) = term0;
  context(1) = term1;

  // Synthesize the trees

  synthesize(module->body);

  *stroot >> finished >> stnode >> boot;
  *top_switch >> enfinished >> depth >> surface;

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

  // Verify the control-flow graph is acyclic

  visit(engrc);

  return result;
}
```

The visitor methods observe the following invariants:

- The stnode, surface, and depth members point to nodes for the statement
- The context is generally not modified. In particular, context(0) is not set the surface or depth.

4.1 Check Acyclic

This simple DFS verifies that the control-flow graph is acyclic.

53a ⟨*RecursiveSynth declarations* 50b⟩+≡

```
map<GRCNode *, bool> visiting;
void visit(GRCNode *);
```

53b ⟨*RecursiveSynth definitions* 51a⟩+≡

```
void RecursiveSynth::visit(GRCNode *n)
{
  assert(n);
  if (visiting.find(n) != visiting.end() && visiting[n])
    throw IR::Error("Cyclic control-flow!");

  visiting[n] = true;

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

  visiting[n] = false;
}
```

4.2 Pause

53c ⟨*RecursiveSynth declarations* 50b⟩+≡

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

53d ⟨*RecursiveSynth definitions* 51a⟩+≡

```
Status RecursiveSynth::visit(Pause &s)
{
  stnode = new STleaf();

  surface = new Enter(stnode);
  *surface >> context(1);

  depth = context(0);

  return Status();
}
```

4.3 Exit

54a ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(Exit &);
```

54b ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(Exit &s)
{
  assert(s.trap);
  stnode = new STref();
  surface = new Action(clone(&s));
  *surface >> context(code[s.trap]);
  depth = context(0);
  return Status();
}
```

4.4 Emit

54c ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(Emit &);
```

54d ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(Emit &s)
{
  stnode = new STref();
  surface = new Action(clone(&s));
  *surface >> context(0);
  depth = context(0);
  return Status();
}
```

4.5 Assign

54e ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(Assign &);
```

54f ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(Assign &s)
{
  stnode = new STref();
  surface = new Action(clone(&s));
  *surface >> context(0);
  depth = context(0);
  return Status();
}
```

4.6 IfThenElse

54g ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(IfThenElse &);
```

55a ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(IfThenElse &s)
{
  if (s.then_part) {
    context.push();
    synthesize(s.then_part);
    context.pop();
  }
  STNode *stthen = s.then_part ? stnode : new STref();
  GRCNode *surfacethen = s.then_part ? surface : context(0);
  GRCNode *depththen = s.then_part ? depth : context(0);

  if (s.else_part) {
    context.push();
    synthesize(s.else_part);
    context.pop();
  }
  STNode *stelse = s.else_part ? stnode : new STref();
  GRCNode *surfaceelse = s.else_part ? surface : context(0);
  GRCNode *depthelse = s.else_part ? depth : context(0);

  stnode = new STexcl();
  *stnode >> stelse >> stthen;

  surface = new Test(stnode, clone(s.predicate));
  *surface >> surfaceelse >> surfacethen;

  run_before(surface, new Enter(stnode));

  depth = new Switch(stnode);
  *depth >> depthelse >>depththen;

  return Status();
}
```

4.7 Statement List

55b ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(StatementList &);
```

56a ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(StatementList &s)
{
  if ( s.statements.empty() ) {
    stnode = new STref();
    surface = depth = context(0);
    run_before(surface, new Enter(stnode));
    return Status();
  }

  STexcl *stroot = new STexcl();
  Switch *depthswitch = new Switch(stroot);

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

  stnode = stroot;
  depth = depthswitch;
  run_before(surface, new Enter(stnode));
  return Status();
}
```

4.8 Loop

Esterel prohibits the surface of a loop from terminating instantly (i.e., at level 0), so setting context(0) will only apply to the depth. When the depth terminates, it passes control to a Nop node that immediately passes control back to the surface.

Arbitrary, correct loops may experience schizophrenia. This translation assumes schizophrenia has been eliminated by a preprocessor.

56b ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(Loop &);
```

57a ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(Loop &s)
{
  STref *lp = new STref();

  Nop *loop_bottom = new Nop();

  context(0) = loop_bottom;

  assert(s.body);
  synthesize(s.body);
  *loop_bottom >> surface;

  *lp >> stnode;
  stnode = lp;

  run_before(surface, new Enter(lp));
  return Status();
}
```

4.9 Every

57b ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(Every &);
```

58 ⟨*RecursiveSynth definitions* 51a⟩+≡

```
Status RecursiveSynth::visit(Every &s)
{
  STref *stroot = new STref();
  stroot->setabort();
  STexcl *excl = new STexcl();
  STleaf *halt = new STleaf();

  Enter *enhalt = new Enter(halt);
  *enhalt >> context(1);

  context.push();
  context(0) = enhalt;
  synthesize(s.body);
  context.pop();

  *excl >> halt >> stnode;
  *stroot >> excl;

  GRCNode *bsurface = surface;

  Expression *predicate = NULL;
  Delay *d = dynamic_cast<Delay*>(s.predicate);
  if (d) {
    if (d->is_immediate) {
      assert(d->predicate);
      Test *tst = new Test(stroot, clone(d->predicate));
      *tst >> enhalt >> surface;
      surface = tst;
      predicate = clone(d->predicate);
    } else {
      assert(d->counter);
      StartCounter *scnt = new StartCounter(d->counter, clone(d->count));
      surface = new Action(scnt);
      *surface >> enhalt;
      predicate =
         new CheckCounter(boolean_type, d->counter, clone(d->predicate));
    }
  } else {
    surface = enhalt;
    predicate = clone(s.predicate);
  }

  Test *tst = new Test(stroot, predicate);
  Switch *sw = new Switch(excl);
  *sw >> enhalt >> depth;
  *tst >> sw >> bsurface;

  depth = tst;
```

```
    run_before(surface, new Enter(stroot));
    run_before(depth, new Enter(stroot));

    stnode = stroot;
    return Status();
  }
```

4.10 Repeat

59a ⟨*RecursiveSynth declarations* 50b⟩+≡

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

59b ⟨*RecursiveSynth definitions* 51a⟩+≡

```
  Status RecursiveSynth::visit(Repeat &s)
  {
    STref *stroot = new STref();

    GRCNode *context0 = context(0);

    Test *tst = new Test(stroot, new CheckCounter(boolean_type, s.counter,
                           new LoadVariableExpression(true_symbol)));
    context(0) = tst;

    assert(s.body);
    synthesize(s.body);
    *tst >> surface >> context0;

    *stroot >> stnode;
    stnode = stroot;

    assert(s.counter);
    run_before(surface, new Action(new StartCounter(s.counter, clone(s.count))));
    run_before(surface, new Enter(stroot));
    return Status();
  }
```

4.11 Suspend

59c ⟨*RecursiveSynth declarations* 50b⟩+≡

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

60 ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(Suspend &s)
{
  synthesize(s.body);

  // Selection tree node for the body of the suspend
  STref *stbody = new STref();
  stbody->setsuspend();
  *stbody >> stnode;
  stnode = stbody;

  run_before(surface, new Enter(stbody));

  Test *immediate_test = NULL;

  Expression *predicate = NULL;
  Delay *d = dynamic_cast<Delay*>(s.predicate);
  if (d) {
    if (d->is_immediate) {
      STNode *imm = new STleaf();

      STexcl *ex = new STexcl();
      *ex >> stnode >> imm;
      stnode = ex;

      // Machinery for the surface: an extra test
      Enter *en = new Enter(imm);
      *en >> context(1);
      immediate_test = new Test(stbody, clone(d->predicate));
      *immediate_test >> surface >> en;
      surface = immediate_test;
      run_before(surface, new Enter(stnode));

      predicate = clone(d->predicate);

    } else {
      // A counted suspend (suspend .. when 5 A)
      assert(d->counter);
      run_before(surface, new Action(new StartCounter(d->counter,
                                                      clone(d->count))));
      predicate = new CheckCounter(boolean_type, d->counter,
                                   clone(d->predicate));
    }
  } else {
    predicate = clone(s.predicate);
  }
  assert(predicate);

  // Machinery for the depth: a test that sends control to either
  // an ST suspend node to context(1) (e.g., the suspend condition)
```

```
    // or the depth of the body

    STSuspend *sts = new STSuspend(stbody);
    *sts >> context(1);
    Test *t = new Test(stbody, predicate);
    *t >> depth >> sts;
    depth = t;
    run_before(depth, new Enter(stbody));

    if (immediate_test) {
      Switch *sw = new Switch(stnode);
      *sw >> depth >> immediate_test;
      depth = sw;
    }

    return Status();
  }
```

4.12 Abort

61 ⟨*RecursiveSynth declarations* 50b⟩+≡

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

62 ⟨*RecursiveSynth definitions* 51a⟩+≡

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

  context.push();
  synthesize(s.body);
  context.pop();

  STref *stbody = new STref();
  stbody->setabort();
  *stbody >> stnode;

  run_before(surface, new Enter(stbody));
  run_before(depth, new Enter(stbody));

  GRCNode *bsurface = surface;
  GRCNode *bdepth = depth;

  STexcl *stroot = new STexcl();
  *stroot >> stbody;

  Switch *depth_root = new Switch(stroot);
  Nop *start_depth = new Nop();
  *depth_root >> start_depth;

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

    if ((*i)->body) {
      context.push();
      synthesize((*i)->body);
      context.pop();
      *stroot >> stnode;
      *depth_root >> depth;
    }

    assert((*i)->predicate);
    Expression *predicate = NULL;
    Delay *d = dynamic_cast<Delay*>((*i)->predicate);
    if (d) {
      if (d->is_immediate) {
        // An immediate delay

        assert(d->counter == NULL);
        assert(d->predicate);

        Test *t = new Test(stbody, clone(d->predicate));
        *t >> bsurface >> ( (*i)->body ? surface : context(0) );
```

```
        bsurface = t;

        predicate = clone(d->predicate);

      } else {
        // A counted delay
        assert(d->counter);
        run_before(bsurface, new Action(new StartCounter(d->counter,
                                                         clone(d->count))));

        predicate =
          new CheckCounter(boolean_type, d->counter, clone(d->predicate));
      }
    } else {
      predicate = clone((*i)->predicate);
    }
    assert(predicate);

    Test *t = new Test(stbody, predicate);
    *t >> bdepth >> ( (*i)->body ? surface : context(0) );
    bdepth = t;

  }

  stnode = stroot;
  run_before(bsurface, new Enter(stroot));
  surface = bsurface;
  *start_depth >> bdepth;
  depth = depth_root;

  return Status();
}
```

4.13 Parallel Statement List

63 ⟨*RecursiveSynth declarations* 50b⟩+≡

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

64 ⟨*RecursiveSynth definitions* 51a⟩+≡

```
Status RecursiveSynth::visit(ParallelStatementList &s)
{
  STpar *stroot = new STpar();

  Sync *sync = new Sync(stroot);
  Fork *surface_fork = new Fork(sync);
  Fork *depth_fork = new Fork(sync);

  context.push();

  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++ ) {
    assert(*i);

    int threadnum = i - s.threads.begin();

    STleaf *term = new STleaf();
    term->setfinal();

    for ( int l = 0 ; l < context.size ; l++ ) {
      Terminate *t = new Terminate(l, threadnum);
      *t >> sync;
      context(l) = t;
      // This extra enter should only be necessary for level 0 terminates,
      // since higher levels necessarily terminate the thread group,
      // however, they appear to be necessary so the optimizer doesn't get
      // too aggressive on potentially vacuous depth
      if (l != 1) run_before(context(l), new Enter(term));
    }

    GRCNode *terminate0 = context(0);

    synthesize(*i);

    // Selection tree fragment: track whether the thread has terminated

    STexcl *ex = new STexcl();
    *stroot >> ex;
    *ex >> term >> stnode;

    run_before(surface, new Enter(ex));

    *surface_fork >> surface;

    Switch *sw = new Switch(ex);
    *sw >> terminate0 >> depth;
    *depth_fork >> sw;
  }
```

```
    context.pop();

    // Connect the sync to every continuation in the context

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

    surface = surface_fork;
    run_before(surface, new Enter(stroot));

    depth = depth_fork;
    run_before(depth, new Enter(stroot)); // Actually, a "hold"

    stnode = stroot;
    return Status();
  }
```

4.14 Trap

65 ⟨*RecursiveSynth declarations* 50b⟩+≡

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

66 ⟨*RecursiveSynth definitions* 51a⟩+≡

```
Status RecursiveSynth::visit(Trap &s)
{
  assert(s.body);
  assert(s.symbols->size());
  if (s.handlers.size() >= 2)
    throw IR::Error("Multiple trap handler.  Did the dismantler run?");

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

  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, module->signals);
  }

  GRCNode *handlerSurface = context(0);
  GRCNode *handlerDepth = NULL;

  STref *bodytree = new STref(); // Selection tree for the body of the trap
  STNode *stroot = bodytree;
  STNode *topExclusive = NULL;

  if ( !s.handlers.empty() ) {
    assert(s.handlers.front());

    context.push();
    synthesize(s.handlers.front()->body);
    handlerSurface = surface;
    handlerDepth = depth;

    stroot = topExclusive = new STexcl();
    *topExclusive >> bodytree >> stnode;

    context.pop();
  }

  context.push();
  context(level) = handlerSurface;
  synthesize(s.body);
  context.pop();

  *bodytree >> stnode;

  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);
    run_before(surface, new DefineSignal(ss, true));

    clone.clearSig(ss);
  }

  run_before(surface, new Enter(bodytree));

  if (handlerDepth) {
    run_before(surface, new Enter(stroot));
    Switch *depth_switch = new Switch(stroot);
    *depth_switch >> depth >> handlerDepth;
    depth = depth_switch;
  }

  stnode = stroot;
  return Status();
}
```

4.15 Signal and Var

67 ⟨*RecursiveSynth declarations* 50b⟩+≡

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

68a ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::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, module->signals);
  }

  synthesize(s.body);

  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i!= s.symbols->end() ; i++ ) {
    SignalSymbol *sig = dynamic_cast<SignalSymbol*>(*i);
    assert(sig);
    run_before(surface, new DefineSignal(clone(sig), true));
    run_before(depth, new DefineSignal(clone(sig), false));
    clone.clearSig(sig);
  }

  return Status();
}

Status RecursiveSynth::visit(Var &s)
{
  for ( SymbolTable::const_iterator i = s.symbols->begin() ;
        i != s.symbols->end() ; i++ ) {
    VariableSymbol *vs = dynamic_cast<VariableSymbol*>(*i);
    assert(vs);
    clone.hoistLocalVariable(vs, module->variables);
  }

  synthesize(s.body);
  return Status();
}
```

4.16 Procedure Call

68b ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(ProcedureCall &);
```

69a ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(ProcedureCall &s)
{
  stnode = new STref();
  surface = new Action(clone(&s));
  *surface >> context(0);
  depth = context(0);
  return Status();
}
```

4.17 Exec

Unimplemented: does nothing.

69b ⟨*RecursiveSynth declarations* 50b⟩+≡
```
Status visit(Exec &);
```

69c ⟨*RecursiveSynth definitions* 51a⟩+≡
```
Status RecursiveSynth::visit(Exec &s)
{
  stnode = new STref();
  surface = context(0);
  depth = context(0);
  return Status();
}
```

5 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.

70a ⟨*dependency class* 70a⟩≡
```
class Dependencies : public Visitor {
protected:
  set<GRCNode *> visited;
  GRCNode *current;

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

  map<SignalSymbol *, SignalNodes> dependencies;

  ⟨dependency methods 71b⟩
  Dependencies() {}
  virtual ~Dependencies() {}

  static void compute(GRCNode *);
};
```

70b ⟨*dependency method definitions* 70b⟩≡
```
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;
    }
  }
}
```

5.1 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.

71a ⟨*dependency method definitions* 70b⟩+≡
```
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);
}
```

71b ⟨*dependency methods* 71b⟩≡
```
void dfs(GRCNode *);
```

5.2 Action

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

71c ⟨*dependency method definitions* 70b⟩+≡
```
Status Dependencies::visit(Action &act)
{
  assert(act.body);
  act.body->welcome(*this);
  return Status();
}
```

71d ⟨*dependency methods* 71b⟩+≡
```
Status visit(Action &);
```

71e ⟨*dependency methods* 71b⟩+≡
```
Status visit(Emit &e) {
  dependencies[e.signal].writers.insert(current);
  if (e.value) e.value->welcome(*this);
  return Status();
}
```

71f ⟨*dependency methods* 71b⟩+≡
```
Status visit(Exit &e) {
  dependencies[e.trap].writers.insert(current);
  if (e.value) e.value->welcome(*this);
  return Status();
}
```

72a ⟨*dependency methods* 71b⟩+≡
```
Status visit(Assign &a) {
  a.value->welcome(*this);
  return Status();
}
```

72b ⟨*dependency methods* 71b⟩+≡
```
Status visit(ProcedureCall &c) {
  for ( vector<Expression*>::const_iterator i = c.value_args.begin() ;
        i != c.value_args.end() ; i++ )
    (*i)->welcome(*this);
  return Status();
}
```

The following actions never have data dependencies.

72c ⟨*dependency methods* 71b⟩+≡
```
Status visit(Pause &) { return Status(); }
Status visit(StartCounter &) { return Status(); }
```

5.3 DefineSignal

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

72d ⟨*dependency methods* 71b⟩+≡
```
Status visit(DefineSignal &d) {
  dependencies[d.signal].writers.insert(current);
  return Status();
}
```

5.4 Test

This descends down its predicate, possibly adding signal testers

72e ⟨*dependency methods* 71b⟩+≡
```
Status visit(Test &t) { t.predicate->welcome(*this); return Status(); }
```

5.5 Expressions

73a ⟨*dependency methods* 71b⟩+≡

```
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.predicate->welcome(*this);
  return Status();
}

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

Status visit(FunctionCall &c) {
    for ( vector<Expression*>::const_iterator i = c.arguments.begin() ;
        i != c.arguments.end() ; i++ )
    (*i)->welcome(*this);
  return Status();
}
```

5.5.1 Vacuous Expression Nodes

73b ⟨*dependency methods* 71b⟩+≡

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

5.6 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.

74a ⟨*dependency method definitions* 70b⟩+≡
```
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();
}
```

74b ⟨*dependency methods* 71b⟩+≡
```
Status visit(Sync &);
```

5.7 Trivial visitors

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

74c ⟨*dependency methods* 71b⟩+≡
```
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(); }
```

6 ASTGRC.hpp and .cpp

75a ⟨*ASTGRC.hpp* 75a⟩≡

```
#ifndef _ASTGRC_HPP
#  define _ASTGRC_HPP

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

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

  class GrcSynth;

  ⟨completion code class 3a⟩

  ⟨cloner class 7⟩

  ⟨context class 14⟩
  ⟨grc walker class 19a⟩
  ⟨surface class 20d⟩
  ⟨depth class 20e⟩
  ⟨st class 20b⟩
  ⟨GrcSynth class 15a⟩
  ⟨RecursiveSynth class 50a⟩

  ⟨dependency class 70a⟩
}
#endif
```

75b ⟨*ASTGRC.cpp* 75b⟩≡

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

namespace ASTGRC {
  ⟨grc synth method definitions 16⟩
  ⟨grc walker methods 19b⟩
  ⟨surface method definitions 21d⟩
  ⟨depth method definitions 21f⟩
  ⟨st method definitions 20c⟩
  ⟨dependency method definitions 70b⟩
  ⟨RecursiveSynth definitions 51a⟩
}
```

76 ⟨*cec-astgrc.cpp* 76⟩≡

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

int main(int argc, char *argv[])
{

  bool expand = true;

  if ( argc > 1) {
    if (argc == 2 && strcmp(argv[1], "-s") == 0) {
      expand = false;
    } else {
      std::cerr << "Usage: cec-astgrc [-s]\n";
      return 1;
    }
  }

  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
      if (expand) {
        ASTGRC::GrcSynth synth(*i, cc);
        (*i)->body = synth.synthesize();
      } else {
        ASTGRC::RecursiveSynth 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;
}
```