CEC Abstract Syntax Tree

Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Abstract

This uses the CEC IR system (responsible for XML serialization of objects) to represent Esterel programs at various stages of compilation. The AST classes represent the program at a syntactic level; the GRC classes represent the program as a control flow graph variant. Many GRC nodes refer to AST symbol tables and whatnot.

This file generates a Bourne shell script that generates .hpp and .cpp for the C++ classes.

Contents

1 The ASTNode Class **2**

2 Symbols and Types **3**
2.1 Module . . . 3
2.2 Signals, Sensors, Traps, Variables, and Constants . . . 3
2.3 Type Symbols . . . 5

3 Symbol Tables **6**

4 Expressions **7**
4.1 Literal . . . 8
4.2 Variables, Signals, and Traps . . . 8
4.3 Operators . . . 9
4.4 Function Call . . . 9
4.5 Delay . . . 10
4.6 CheckCounter . . . 10

5 Modules **10**

6 Statements **12**
6.1 Sequential and Parallel Statement Lists 13
6.2 Nothing, Pause, Halt, Emit, Exit, Sustain, and Assign 14
6.3 Procedure Call 14
6.4 Present, If, and If-Then-Else 15
6.5 Loop and Repeat 15
6.6 Abort, Await, Every, Suspend, Dowatching, and DoUpto 16
6.7 Exec 17
6.8 Trap, Signal, and Var 17
6.9 Run 18
6.10 StartCounter 19

7 GRC Nodes **20**
7.1 GRC control-flow nodes 21
7.1.1 Additional Flow Control 22
7.1.2 Switch 23
7.1.3 Test 23
7.1.4 STSuspend 23
7.1.5 Fork 24
7.1.6 Sync and Terminate 24
7.1.7 Action 24
7.1.8 Enter 25
7.2 Selection Tree Nodes 25

8 The Shell Script **27**

1 The ASTNode Class

All AST nodes are derived from this class; the `Visitor` class takes an `ASTNode` as an argument.

2 ⟨*ASTNode class* 2⟩≡
```
abstract "ASTNode : Node


  virtual Status welcome(Visitor&) = 0;
"
```

2 Symbols and Types

Symbols represent names in the Esterel source code, such as those for signals, functions, variables, and other modules.

3a ⟨*Symbols* 3a⟩≡

```
abstract "Symbol : ASTNode
  string name;

  Symbol(string s) : name(s) {}"
```

2.1 Module

Symbol representing a module.

3b ⟨*Symbols* 3a⟩+≡

```
class "ModuleSymbol : Symbol
  Module *module;

  ModuleSymbol(string s) : Symbol(s), module(0) {}"
```

2.2 Signals, Sensors, Traps, Variables, and Constants

Variable, Trap, and Signal symbols have a type and optional initializing expression, represented by this abstract class.

3c ⟨*Symbols* 3a⟩+≡

```
abstract "ValuedSymbol : Symbol
  TypeSymbol *type;
  Expression *initializer;

  ValuedSymbol(string n, TypeSymbol *t, Expression *e)
    : Symbol(n), type(t), initializer(e) {}"
```

Variables and constants are simply ValuedSymbols. Constants much have an initializing expression. BuiltinConstantSymbol is for the constants true and false.

4a ⟨*Symbols* 3a⟩+≡
```
class "VariableSymbol : ValuedSymbol

  VariableSymbol(string n, TypeSymbol *t, Expression *e)
    : ValuedSymbol(n, t, e) {}"

class "ConstantSymbol : VariableSymbol

  ConstantSymbol(string n, TypeSymbol *t, Expression *i)
     : VariableSymbol(n, t, i) {}"

class "BuiltinConstantSymbol : ConstantSymbol

  BuiltinConstantSymbol(string n, TypeSymbol *t, Expression *i)
     : ConstantSymbol(n, t, i) {}"
```

SignalSymbol represents a signal, trap, sensor, or return signal for a task. Pure signals and traps have a NULL type. The `combine` field points to the "combine" function (e.g., combine integer with +) if there is one, and NULL otherwise.

When valued signals are reincarnated, they may need a separate status to represent their presence, but their values may need to persist. The reincarnation field points to the signal of which this is one is a recinarnation, or is NULL if this is the "master" signal for a group.

4b ⟨*Symbols* 3a⟩+≡
```
class "SignalSymbol : ValuedSymbol
  typedef enum { Input,Output,Inputoutput,Sensor,Return,Local,Trap,Unknown } kinds;
  int kind;
  FunctionSymbol *combine; // combining function, if any
  SignalSymbol *reincarnation; // the signal this is a reincarnation of, if any

  SignalSymbol(string n, TypeSymbol *t, kinds k, FunctionSymbol *f,
               Expression *e, SignalSymbol *r)
    : ValuedSymbol(n, t, e), kind(k), combine(f), reincarnation(r) {}
"
```

For the built-in signal "tick."

4c ⟨*Symbols* 3a⟩+≡
```
class "BuiltinSignalSymbol : SignalSymbol

  BuiltinSignalSymbol(string n, TypeSymbol *t, kinds k, FunctionSymbol *f)
    : SignalSymbol(n, t, k, f, NULL, NULL) {}"
```

2.3 Type Symbols

Esterel's type system provides a way to import types from a host language. A `TypeSymbol` is just a name, while the function and procedure types are for representing functions (return a value) and procedures (do not return a value, but have pass-by-reference parameters).

5a ⟨*Symbols* 3a⟩+≡
```
class "TypeSymbol : Symbol

   TypeSymbol(string s) : Symbol(s) {}"
```

A `BuiltinTypeSymbol` represents one of the five built-in types: boolean, integer, float, double, and string.

5b ⟨*Symbols* 3a⟩+≡
```
class "BuiltinTypeSymbol : TypeSymbol

   BuiltinTypeSymbol(string s) : TypeSymbol(s) {}"
```

A imported function, e.g., "function foo(integer) : boolean;"

5c ⟨*Symbols* 3a⟩+≡
```
class "FunctionSymbol : TypeSymbol
  vector<TypeSymbol*> arguments;
  TypeSymbol *result;

  FunctionSymbol(string s) : TypeSymbol(s), result(NULL) {}"
```

BuiltinFunctionSymbols are used in "combine" declarations or module renamings. Some of them have a null return type because they're polymorphic (e.g., `*`).

5d ⟨*Symbols* 3a⟩+≡
```
class "BuiltinFunctionSymbol : FunctionSymbol

  BuiltinFunctionSymbol(string s) : FunctionSymbol(s) {}"
```

An imported procedure or task, e.g., "procedure bar(integer)(boolean)"

5e ⟨*Symbols* 3a⟩+≡
```
class "ProcedureSymbol : TypeSymbol
  vector<TypeSymbol*> reference_arguments;
  vector<TypeSymbol*> value_arguments;

  ProcedureSymbol(string s) : TypeSymbol(s) {}"
```

5f ⟨*Symbols* 3a⟩+≡
```
class "TaskSymbol : ProcedureSymbol

  TaskSymbol(string s) : ProcedureSymbol(s) {}"
```

3 Symbol Tables

A symbol table is basically a vector of symbols with a linear search function. Although a map might be more efficient, the order in which the symbols appear in the table is important because no forward references are allowed.

The `local_contains` method indicates whether a symbol with the given name is contained in this particular table table. The `contains` method also searches in containing scopes.

The `enter` method adds a symbol to the table. It assumes the table does not already contain a symbol with the same name.

The `get` method returns the symbol with the given name. It assumes the symbol is present in the table.

6 ⟨*SymbolTable* 6⟩≡

```
class "SymbolTable : ASTNode
  SymbolTable *parent;
  typedef vector<Symbol*> stvec;
  stvec symbols;

  SymbolTable() : parent(NULL) {}

  class const_iterator {
    stvec::const_iterator i;
  public:
    const_iterator(stvec::const_iterator ii) : i(ii) {}
    void operator ++(int) { i++; } // int argument denotes postfix
    void operator ++() { ++i; } // int argument denotes postfix
    bool operator !=(const const_iterator &ii) { return i != ii.i; }
    Symbol *operator *() { return *i; }
  };

  const_iterator begin() const { return const_iterator(symbols.begin()); }
  const_iterator end() const { return const_iterator(symbols.end()); }
  size_t size() const { return symbols.size(); }
  void clear() { symbols.clear(); }

  bool local_contains(const string) const;
  bool contains(const string) const;
  void enter(Symbol *);
  Symbol* get(const string);
" "
bool SymbolTable::local_contains(const string s) const {
  for ( stvec::const_iterator i = symbols.begin() ; i != symbols.end() ; i++ ) {
    assert(*i);
    if ( (*i)->name == s) return true;
  }
  return false;
}

bool SymbolTable::contains(const string s) const {
```

```
  for ( const SymbolTable *st = this ; st ; st = st->parent )
    if (st->local_contains(s)) return true;
  return false;
}

void SymbolTable::enter(Symbol *sym) {
  assert(sym);
  assert(!local_contains(sym->name));
  symbols.push_back( sym );
}

Symbol* SymbolTable::get(const string s) {
  for ( SymbolTable *st = this; st ; st = st->parent ) {
    for ( const_iterator i = st->begin() ; i != st->end() ; i++ )
      if ( (*i)->name == s) return *i;
  }
  assert(0); // get should not be called unless contains returned true
}
"
```

4 Expressions

7a ⟨*Expression classes* 7a⟩≡

⟨*Expression* 7b⟩

⟨*Literal* 8a⟩
⟨*LoadVariableExpression* 8b⟩
⟨*LoadSignalExpression* 8c⟩
⟨*LoadSignalValueExpression* 8d⟩

⟨*UnaryOp* 9a⟩
⟨*BinaryOp* 9b⟩
⟨*FunctionCall* 9c⟩
⟨*Delay* 10a⟩
⟨*CheckCounter* 10b⟩

Every `Expression` has a type.

7b ⟨*Expression* 7b⟩≡

```
abstract "Expression : ASTNode
  TypeSymbol *type;

  Expression(TypeSymbol *t) : type(t) {}"
```

4.1 Literal

A literal is an integer, float, double, or string literal value. All are stored as strings to maintain precision.

8a ⟨*Literal* 8a⟩≡

```
class "Literal : Expression
  string value;

  Literal(string v, TypeSymbol *t) : Expression(t), value(v) {}"
```

4.2 Variables, Signals, and Traps

`LoadVariableExpression` is a reference to a variable or constant. It is also used to reference the built-in boolean constants `true` and `false`.

8b ⟨*LoadVariableExpression* 8b⟩≡

```
class "LoadVariableExpression : Expression
  VariableSymbol *variable;

  LoadVariableExpression(VariableSymbol *v)
    : Expression(v->type), variable(v) {}"
```

`LoadSignalExpression` returns the presence/absence of a signal or trap. Used by present, etc. Its `type` should always be the built-in boolean

8c ⟨*LoadSignalExpression* 8c⟩≡

```
class "LoadSignalExpression : Expression
  SignalSymbol *signal;

  LoadSignalExpression(TypeSymbol *t, SignalSymbol *s)
    : Expression(t), signal(s) {}"
```

`LoadSignalValueExpression` returns the value of a valued signal or trap, i.e., the `?` operator for signals, the `??` operator for traps.

8d ⟨*LoadSignalValueExpression* 8d⟩≡

```
class "LoadSignalValueExpression : Expression
  SignalSymbol *signal;

  LoadSignalValueExpression(SignalSymbol *s)
    : Expression(s->type), signal(s) {}"
```

4.3 Operators

Esterel has the usual unary and binary operators. The `op` field represents the actual type of the operator. Its value is the Esterel syntax for the operator, e.g., `<>` for not equal.

9a ⟨*UnaryOp* 9a⟩≡
```
class "UnaryOp : Expression
  string op;
  Expression *source;

  UnaryOp(TypeSymbol *t, string s, Expression *e)
    : Expression(t), op(s), source(e) {}"
```

9b ⟨*BinaryOp* 9b⟩≡
```
class "BinaryOp : Expression
  string op;
  Expression *source1;
  Expression *source2;

  BinaryOp(TypeSymbol *t, string s, Expression *e1, Expression *e2)
    : Expression(t), op(s), source1(e1), source2(e2) {}"
```

4.4 Function Call

This is a function call in an expression. Callee must be defined.

9c ⟨*FunctionCall* 9c⟩≡
```
class "FunctionCall : Expression
  FunctionSymbol *callee;
  vector<Expression*> arguments;

  FunctionCall(FunctionSymbol *s)
    : Expression(s->result), callee(s) {}"
```

4.5 Delay

This is a delay, e.g., the argument of await 5 SECOND. The predicate is a pure signal expression that returns the built-in boolean. The count may be undefined. `is_immediate` is true for expressions such as "await immediate A." The `counter` variable is used when the delay is a counted one, and is 0 for immediate delays.

10a ⟨*Delay* 10a⟩≡
```
class "Delay : Expression
  Expression *predicate;
  Expression *count;
  bool is_immediate;
  Counter *counter;

  Delay(TypeSymbol *t, Expression *e1, Expression *e2,
        bool i, Counter *c)
   : Expression(t), predicate(e1), count(e2), is_immediate(i), counter(c) {}"
```

4.6 CheckCounter

Not part of Esterel's grammar, a CheckCounter expression decrements its counter if the predicate expression is true and returns true if the counter has reached 0. This is generated during the dismantling phase for statements such as `await 5 A`.

10b ⟨*CheckCounter* 10b⟩≡
```
class "CheckCounter : Expression
  Counter *counter;
  Expression *predicate;

  CheckCounter(TypeSymbol *t, Counter *c, Expression *p )
    : Expression(t), counter(c), predicate(p) {}
"
```

5 Modules

10c ⟨*Module classes* 10c⟩≡
  ⟨*Module* 11⟩
  ⟨*InputRelation classes* 12a⟩
  ⟨*Counter* 12b⟩
  ⟨*Modules* 12c⟩

Esterel places signals, types, variables/constants, functions, procedures, tasks, and traps in separate namespaces, so each has its own symbol table here except traps, which are only in scopes.

The `variables` symbol table holds `VariableSymbols` representing signal presence and value, trap status and values, counters, state variables, etc., all generated during the disamantling process.

11 ⟨*Module* 11⟩≡
```
class "Module : ASTNode
  ModuleSymbol *symbol;
  SymbolTable *types;
  SymbolTable *constants;
  SymbolTable *functions;
  SymbolTable *procedures;
  SymbolTable *tasks;
  SymbolTable *signals;
  SymbolTable *variables;
  vector<Counter*> counters;
  vector<InputRelation*> relations;
  ASTNode *body;

  Module() {}
  Module(ModuleSymbol *);
  ~Module();
" "
Module::Module(ModuleSymbol *s) : symbol(s), body(NULL) {
  signals = new SymbolTable();
  constants = new SymbolTable();
  types = new SymbolTable();
  functions = new SymbolTable();
  procedures = new SymbolTable();
  tasks = new SymbolTable();
  variables = new SymbolTable();
}

Module::~Module() {
   delete signals;
   delete types;
   delete constants;
   delete functions;
   delete procedures;
   delete tasks;
   delete body;
   delete variables;
}"
```

Relations are constraints (either exclusion or implication) among two or more input signals.

12a ⟨*InputRelation classes* 12a⟩≡
```
abstract "InputRelation : ASTNode"

class "Exclusion : InputRelation
  vector<SignalSymbol *> signals;"

class "Implication : InputRelation
  SignalSymbol *predicate;
  SignalSymbol *implication;

  Implication(SignalSymbol *ss1, SignalSymbol*ss2)
    : predicate(ss1), implication(ss2) {}"
```

Counters are implicit objects used by counted delays and the *repeat* statement, e.g., `abort halt when 5 A` and `repeat 5 times ... end`. This object is little more than a placeholder. All the action takes place in the StartCounter statement and CheckCounter expressions.

12b ⟨*Counter* 12b⟩≡
```
class "Counter : ASTNode"
```

12c ⟨*Modules* 12c⟩≡
```
class "Modules : ASTNode
  SymbolTable module_symbols;
  vector<Module*> modules;

  void add(Module*);
" "
void Modules::add(Module* m) {
  assert(m);
  assert(m->symbol);
  assert(!module_symbols.contains(m->symbol->name));
  modules.push_back(m);
  module_symbols.enter(m->symbol);
}"
```

6 Statements

12d ⟨*Statements* 12d⟩≡
```
abstract "Statement : ASTNode"
```

The following helper statements are used as parts of other high-level statements or as base classes. A `BodyStatement` is simply one that contains another. A Boolean predicate expression controls the execution of the body of a `PredicatedStatement`. A `CaseStatement` is an abstract notion of a series of choices: if the first predicate is true, execute the first body, else check and execute the second, etc. If none hold, execute the optional default.

13a ⟨*Statements* 12d⟩+≡
```
abstract "BodyStatement : Statement
  Statement *body;

  BodyStatement(Statement *s) : body(s) {}"
```

13b ⟨*Statements* 12d⟩+≡
```
class "PredicatedStatement : BodyStatement
  Expression *predicate;

  PredicatedStatement(Statement *s, Expression *e)
    : BodyStatement(s), predicate(e) {}"
```

13c ⟨*Statements* 12d⟩+≡
```
abstract "CaseStatement : Statement
  vector<PredicatedStatement *> cases;
  Statement *default_stmt;

  CaseStatement() : default_stmt(0) {}
  PredicatedStatement *newCase(Statement *s, Expression *e) {
    PredicatedStatement *ps = new PredicatedStatement(s, e);
    cases.push_back(ps);
    return ps;
  }"
```

6.1 Sequential and Parallel Statement Lists

`StatementList` handles sequences of statements, i.e., those separated by `;`; `ParallelStatementList` handles sequences separated by `||`.

13d ⟨*Statements* 12d⟩+≡
```
class "StatementList : Statement
  vector<Statement *> statements;

  StatementList& operator <<(Statement *s) {
    assert(s);
    statements.push_back(s);
    return *this;
  }"
```

13e ⟨*Statements* 12d⟩+≡
```
class "ParallelStatementList : Statement
  vector<Statement *> threads;"
```

6.2 Nothing, Pause, Halt, Emit, Exit, Sustain, and Assign

Nothing does nothing, pause delays a cycle, halt delays indefinitely, emit emits a signal, perhaps with a value, exit raises a trap, also with an optional value, sustain emits a signal continously, and the assignment statement implements `:=`, assignment to a variable. Emit has a flag for three-valued that marks the signal as being unknown.

14a ⟨*Statements* 12d⟩+≡

```
class "Nothing : Statement"
class "Pause : Statement"
class "Halt : Statement"

class "Emit : Statement
  SignalSymbol *signal;
  Expression *value;
  bool unknown;

  Emit(SignalSymbol *s, Expression *e)
    : signal(s), value(e), unknown(false) {}"

class "Exit : Statement
  SignalSymbol *trap;
  Expression *value;

  Exit(SignalSymbol *t, Expression *e) : trap(t), value(e) {}"

class "Sustain : Emit

  Sustain(SignalSymbol *s, Expression *e) : Emit(s, e) {}"

class "Assign : Statement
  VariableSymbol *variable;
  Expression *value;

  Assign(VariableSymbol *v, Expression *e) : variable(v), value(e) {}"
```

6.3 Procedure Call

Procedure call is a statement that takes a procedure, a collection of pass-by-reference arguments, and a collection of pass-by-value arguments.

14b ⟨*Statements* 12d⟩+≡

```
class "ProcedureCall : Statement
  ProcedureSymbol *procedure;
  vector<VariableSymbol*> reference_args;
  vector<Expression*> value_args;

  ProcedureCall(ProcedureSymbol *ps) : procedure(ps) {}"
```

6.4 Present, If, and If-Then-Else

Conditional statements test their expressions. Esterel draws a textual distinction between testing signals and expressions, but semantically they are the same.

15a ⟨*Statements* 12d⟩+≡
```
class "Present : CaseStatement"
class "If : CaseStatement"
```

The IfThenElse statement is not part of Esterel; it is generated during the dismantling phase.

15b ⟨*low-level classes* 15b⟩≡
```
class "IfThenElse : Statement
  Expression *predicate;
  Statement *then_part;
  Statement *else_part;

  IfThenElse(Expression *e) : predicate(e) , then_part(0), else_part(0) {}
  IfThenElse(Expression *e, Statement *s1, Statement *s2)
    : predicate(e) , then_part(s1), else_part(s2) {}"
```

6.5 Loop and Repeat

15c ⟨*Statements* 12d⟩+≡
```
class "Loop : BodyStatement

  Loop(Statement *s) : BodyStatement(s) {}"

class "Repeat : Loop
  Expression *count;
  bool is_positive;
  Counter *counter;

  Repeat(Statement *s, Expression *e, bool p, Counter *c)
    : Loop(s), count(e), is_positive(p), counter(c) {}"
```

6.6 Abort, Await, Every, Suspend, Dowatching, and DoUpto

16 ⟨*Statements* 12d⟩+≡

```
class "Abort : CaseStatement
  Statement *body;
  bool is_weak;

  Abort(Statement *s, bool i) : body(s), is_weak(i) {}
  Abort(Statement *s, Expression *e, Statement *s1)
    : body(s), is_weak(false) {
    newCase(s1, e);
  }"

class "Await : CaseStatement"

class "LoopEach : PredicatedStatement

  LoopEach(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"

class "Every : PredicatedStatement

  Every(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"

class "Suspend : PredicatedStatement

  Suspend(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"

class "DoWatching : PredicatedStatement
  Statement *timeout;

  DoWatching(Statement *s1, Expression *e, Statement *s2)
    : PredicatedStatement(s1, e), timeout(s2) {}"

class "DoUpto : PredicatedStatement

  DoUpto(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"
```

6.7 Exec

This is for handing the invocation of tasks. It is complex in that many tasks can be initiated at once.

17a ⟨*Statements* 12d⟩+≡
```
class "TaskCall : ProcedureCall
  SignalSymbol *signal;
  Statement *body;

  TaskCall(TaskSymbol *ts) : ProcedureCall(ts), signal(0), body(0) {}
"

class "Exec : Statement
  vector <TaskCall *> calls;"
```

6.8 Trap, Signal, and Var

17b ⟨*Statements* 12d⟩+≡
```
abstract "ScopeStatement : BodyStatement
   SymbolTable *symbols;"
```

The parent symbol table of a `trap` statement is the innermost enclosing `trap`'s symbol table or null.

17c ⟨*Statements* 12d⟩+≡
```
class "Trap : ScopeStatement
  vector<PredicatedStatement *> handlers;

  PredicatedStatement* newHandler(Expression *e, Statement *s) {
    PredicatedStatement *ps = new PredicatedStatement(s, e);
    handlers.push_back(ps);
    return ps;
  }"
```

17d ⟨*Statements* 12d⟩+≡
```
class "Signal : ScopeStatement"
```

The parent symbol table of the `var` statement is either that for the innermost enclosing `var` statement or the `constants` table in its module.

17e ⟨*Statements* 12d⟩+≡
```
class "Var : ScopeStatement"
```

6.9 Run

18 ⟨*Run classes* 18⟩≡
```
abstract "Renaming : ASTNode
  string old_name;

Renaming(string s) : old_name(s) {}"

class "TypeRenaming : Renaming
  TypeSymbol *new_type;

  TypeRenaming(string s, TypeSymbol *t) : Renaming(s), new_type(t) {}"

class "ConstantRenaming : Renaming
  Expression *new_value;

  ConstantRenaming(string s, Expression *e) : Renaming(s), new_value(e) {}"

class "FunctionRenaming : Renaming
  FunctionSymbol *new_func;

  FunctionRenaming(string s, FunctionSymbol *f) : Renaming(s), new_func(f) {}"

class "ProcedureRenaming : Renaming
  ProcedureSymbol *new_proc;

  ProcedureRenaming(string s, ProcedureSymbol *p)
    : Renaming(s), new_proc(p) {}"

class "SignalRenaming : Renaming
  SignalSymbol *new_sig;

  SignalRenaming(string s, SignalSymbol *ss) : Renaming(s), new_sig(ss) {}"
```

The run statement itself is a pair of names (old and new), vectors of renaming, and finally a pointer to the innermost enclosing scope for signals. The Run statement does not own this symbol table, unlike, say, the var statement. This pointer is used by the expander to find the signals referred to in the instantiated module.

19a ⟨*Run classes* 18⟩+≡
```
class "Run : Statement
  string old_name;
  string new_name;
  vector<TypeRenaming *> types;
  vector<ConstantRenaming *> constants;
  vector<FunctionRenaming *> functions;
  vector<ProcedureRenaming *> procedures;
  vector<ProcedureRenaming *> tasks;
  vector<SignalRenaming *> signals;
  SymbolTable *signalScope;

  Run(string s, SymbolTable *ss) : old_name(s), new_name(s), signalScope(ss)
    {}"
```

6.10 StartCounter

Not a part of Esterel's grammar, this statement initializes its counter to the value of the given expression. Statements such as `await 5 A` generate these.

19b ⟨*Statements* 12d⟩+≡
```
class "StartCounter : Statement
  Counter *counter;
  Expression *count;

  StartCounter(Counter *c, Expression *i): counter(c), count(i) {}"
```

7 GRC Nodes

These follow the GRC format defined in Potop-Butcaru's thesis.

The root of the GRC graph. By convention, its first child is the root of the selection tree, the second is the unique EnterGRC node for the imperative part of the graph.

A GRC graph for a program consists of two linked parts: a selection tree representing the state of the program between cycles and a control-flow graph that represents the behavior of the program in a cycle. Certain nodes in the control-flow graph point to nodes in the selection tree.

The `enumerate` method builds two maps: one for GRCNodes (in the control-flow graph) and the other for STNodes (in the selection tree) that assigns each node to a unique integer. These numbers are used primarily for debugging output.

20 ⟨*GRC graph class* 20⟩≡

```
class "GRCgraph : ASTNode
  STNode *selection_tree;
  GRCNode *control_flow_graph;

  GRCgraph(STNode *st, GRCNode *cfg)
     : selection_tree(st), control_flow_graph(cfg) {}

  int enumerate(GRCNode::NumMap &, STNode::NumMap &, int max = 1);
" "
 int GRCgraph::enumerate(GRCNode::NumMap &cfgmap, STNode::NumMap &stmap, int max)
{
  std::set<GRCNode*> cfg_visited;
  std::set<STNode*>   st_visited;

  assert(selection_tree);
  assert(control_flow_graph);

  max = selection_tree->enumerate(stmap, st_visited, max);
  max = control_flow_graph->enumerate(cfgmap, cfg_visited, max);
  return max;
}

"
```

7.1 GRC control-flow nodes

Successors may contain NULL nodes; these are used, e.g., to represent an unused continuation from a parallel synchronizer. Predecessors should all be non-NULL.

The `>>` operator adds a control successor to the given node, i.e., a node that may be executed after the current one terminates. Thus `a >> b` makes `b` a child of `a`.

The `<<` operator adds a data predecessor to the given node, i.e., a node that generates data that is used by the current node. Thus `a << b` means `a` depends on data from node `b`.

Data predecessors point to GRC nodes that emit signals this node cares about. Data successors point to GRC nodes that listen to signals this node emits.

21 ⟨*GRC classes* 21⟩≡

```
abstract "GRCNode : ASTNode
  vector<GRCNode*> predecessors;
  vector<GRCNode*> successors;
  vector<GRCNode*> dataPredecessors;
  vector<GRCNode*> dataSuccessors;

  virtual Status welcome(Visitor&) = 0;

  GRCNode& operator >>(GRCNode*);
  GRCNode& operator <<(GRCNode*);
  typedef map<GRCNode *, int> NumMap;
  int enumerate(NumMap &, std::set<GRCNode *> &, int);
" "
  GRCNode& GRCNode::operator >>(GRCNode *s) {
    successors.push_back(s);
    if (s) s->predecessors.push_back(this);
    return *this;
  }

  GRCNode& GRCNode::operator <<(GRCNode *p) {
    assert(p);
    dataPredecessors.push_back(p);
    p->dataSuccessors.push_back(this);
    return *this;
  }

  int GRCNode::enumerate(NumMap &number, std::set<GRCNode *> &visited, int next) {

    if (visited.find(this) != visited.end()) return next;
    visited.insert(this);
    if (number.find(this) == number.end() || number[this] == 0) {
      number[this] = next++;
    }
    for (vector<GRCNode*>::const_iterator i = successors.begin();
          i != successors.end() ; i++)
```

```
              if (*i) next = (*i)->enumerate(number, visited, next);
      for (vector<GRCNode*>::const_iterator i = predecessors.begin();
            i != predecessors.end() ; i++)
              if(*i) next = (*i)->enumerate(number, visited, next);
      for (vector<GRCNode*>::const_iterator i = dataSuccessors.begin();
            i != dataSuccessors.end() ; i++)
              if(*i) next = (*i)->enumerate(number, visited, next);
      for (vector<GRCNode*>::const_iterator i = dataPredecessors.begin();
            i != dataPredecessors.end() ; i++)
              if(*i) next = (*i)->enumerate(number, visited, next);
      return next;
    }
  "
```

Certain GRC nodes have pointers to the selection tree. The `GRCSTNode` class represents this.

22a ⟨*GRC classes* 21⟩+≡

```
  abstract "GRCSTNode : GRCNode
    STNode *st;

    GRCSTNode(STNode *s) : st(s) {}
  "
```

7.1.1 Additional Flow Control

The `EnterGRC` and `ExitGRC` nodes are placeholders usually placed at the beginning and end of the control-flow graph.

22b ⟨*GRC classes* 21⟩+≡

```
  class "EnterGRC : GRCNode"
  class "ExitGRC : GRCNode"
```

`Nop` is overloaded: it may or may not do anything.

22c ⟨*GRC classes* 21⟩+≡

```
  class "Nop : GRCNode
    int type;
    int code;
    string body;

    Nop(): type(0), code(0) {}

    int isflowin() { return type == 1;}
    void setflowin() { type = 1;}
    // a shorcut Nop gives "up" flow to child 0
    int isshortcut() { return type == 2;}
    void setshortcut() { type = 2;}
  "
```

DefineSignal is used at the beginning of local signal declarations to indicate when a signal enters scope. The is_surface flag is true when this is a surface entry to a scope, meaning the value, if any, should be initialized.

23a ⟨*GRC classes* 21⟩+≡
```
class "DefineSignal : GRCNode
  SignalSymbol *signal;
  bool is_surface;

  DefineSignal(SignalSymbol *s, bool ss) : signal(s), is_surface(ss) {}
"
```

7.1.2 Switch

Multi-way branch on the state of a thread.

23b ⟨*GRC classes* 21⟩+≡
```
class "Switch : GRCSTNode

  Switch(STNode *s) : GRCSTNode(s) {}
"
```

7.1.3 Test

An if-then-else statement.

23c ⟨*GRC classes* 21⟩+≡
```
class "Test : GRCSTNode
  Expression *predicate;

  Test(STNode *s, Expression *e) : GRCSTNode(s), predicate(e) {}
"
```

7.1.4 STSuspend

23d ⟨*GRC classes* 21⟩+≡
```
class "STSuspend : GRCSTNode

  STSuspend(STNode *s) : GRCSTNode(s) {}
"
```

7.1.5 Fork

Sends control to all its successors; just fan-out in the circuit. The sync field, when set, points to the Sync node that joins these threads.

24a ⟨*GRC classes* 21⟩+≡
```
class "Fork : GRCNode
  Sync* sync;

  Fork() : sync(0) {}
  Fork(Sync* sync) : sync(sync) {}
"
```

7.1.6 Sync and Terminate

A parallel synchronizer. Its predecessors should all be Terminate nodes. When executed, it executes one of its successors: the one corresponding to the maximum exit level, i.e., the highest code of the executed terminate nodes preceeding it. Some of its successors may be NULL.

24b ⟨*GRC classes* 21⟩+≡
```
class "Sync : GRCSTNode

  Sync(STNode *s) : GRCSTNode(s) {}
"
```

Terminates a thread with the given completion code. Should have a single successor, a `Sync` node. The `index` field should be zero for all Terminate nodes reachable from the first successor of the corresponding fork, one for those reachable from the second child, and so forth.

24c ⟨*GRC classes* 21⟩+≡
```
class "Terminate : GRCNode
  int code;
  int index;

  Terminate(int c, int i) : code(c), index(i) {}
"
```

7.1.7 Action

Perform an action such as emission or assignment. Should have a single successor.

24d ⟨*GRC classes* 21⟩+≡
```
class "Action : GRCNode
  Statement *body;

  Action(Statement *s) : body(s) {}
"
```

7.1.8 Enter

This represents the activation of a particular statement.

25a ⟨*GRC classes* 21⟩+≡
```
class "Enter : GRCSTNode

  Enter(STNode *s) : GRCSTNode(s) {}
"
```

7.2 Selection Tree Nodes

The selection tree is the part of GRC that controls the state of the program between cycles.

The `enumerate` method is used to assign a unique number to each STNode object, mostly for debugging.

25b ⟨*GRC classes* 21⟩+≡
```
abstract "STNode : ASTNode
  STNode *parent;
  vector<STNode*> children;

  STNode() : parent(0) {}
  virtual Status welcome(Visitor&) = 0;

  STNode& operator >>(STNode*);
  typedef map<STNode *, int> NumMap;
  int enumerate(NumMap &, std::set<STNode*> &visited, int);
" "
  STNode& STNode::operator >>(STNode *s) {
//    assert(s);
    children.push_back(s);
    if(s) s->parent = this;
    return *this;
  }

  int STNode::enumerate(NumMap &number, std::set<STNode*> &visited, int next) {
    if(visited.find(this) != visited.end()) return next;
    visited.insert(this);

    if(number.find(this) == number.end() || number[this] == 0){
        number[this] = next++;
    }
    for (vector<STNode*>::const_iterator i = children.begin() ;
         i != children.end() ; i++) if(*i)
      next = (*i)->enumerate(number, visited,  next);
    return next;
  }
"
```

26a ⟨*GRC classes* 21⟩+≡
```
class "STexcl : STNode"
```

26b ⟨*GRC classes* 21⟩+≡
```
class "STpar : STNode"
```

26c ⟨*GRC classes* 21⟩+≡
```
class "STref : STNode
  int type;

  STref(): type(0) {}

  int isabort() { return type == 1;}
  void setabort() { type = 1;}
  int issuspend() { return type == 2;}
  void setsuspend() { type = 2;}
"
```

26d ⟨*GRC classes* 21⟩+≡
```
class "STleaf : STNode
  int type;

  STleaf(): type(0) {}

  int isfinal() { return type == 1;}
  void setfinal() { type = 1;}
"
```

8 The Shell Script

This generates the AST.hpp and AST.cpp files from the instructions in this file. The overall idea of this came from a similar system in Stanford's SUIF system. This implementation is simpler, less powerful, and with luck, more maintainable since it's implemented in a familiar, portable programming langugage: the Bourne shell.

27 ⟨*AST.sh* 27⟩≡

```
#!/bin/sh

abstract() {
  class "$1" "$2" "abstract"
}

class() {
  # The classname is the string before the : on the first line
  classname=`echo "$1" | sed -n '1 s/ *:.*$//p'`
  # The parent's class name is the string after the : on the first line
  parent=`echo "$1" | sed -n '1 s/^.*: *//p'` ; # String after :
  # The fields come from the second line through the first empty line
  # Each is the identifier just before the semicolon
  # Lines with "typedef" are skipped
  fields=`echo "$1" | sed '/typedef/d' | sed -n '2,/^\$/ s/^.*[^a-zA-Z0-9_]\([a-zA-Z0-9_]*\);.*/\1/p'`
  # The body for the header file starts at the second line
  hppbody=`echo "$1" | sed -n '2,$p'`

  # Any additional methods are defined in the second argument

  #echo "[$classname]"
  #echo "[$parent]"
  #echo "[$fields]"
  #echo "[$hppbody]"

  forwarddefs="$forwarddefs
  class $classname;"

  # Define a default (zero-argument) constructor if one isn't already
  # defined in the body
  if (echo $hppbody | grep -q "$classname()"); then
    defaultconstructor=
  else
    defaultconstructor="$classname() {}
"
  fi

  if test -z "$3"; then
      visitorclassdefs="$visitorclassdefs
  virtual Status visit($classname& n) { assert(0); return Status(); }"
      welcome="
```

```
    IRCLASSDEFS;
  public:
    Status welcome(Visitor&);"
      welcomedef="
  IRCLASS($classname);
  Status $classname::welcome(Visitor &v) { return v.visit(*this); }"
  else
    welcome="public:"
    welcomedef=
  fi

  classdefs="$classdefs

  class $classname : public $parent {
    $welcome
    $copyme
    void read(XMListream &);
    void write(XMLostream &) const;
    $defaultconstructor
$hppbody
  };
"

  if test -n "$fields"; then
    writefields=`echo $fields | sed "s/ / << /g"`;
    writefields="
    w << $writefields;"
    readfields=`echo $fields | sed "s/ / >> /g"`;
    readfields="
    r >> $readfields;"
  else
    readfields=
    writefields=
  fi

  methoddefs="$methoddefs

  void $classname::read(XMListream &r) {
    $parent::read(r); $readfields
  }

  void $classname::write(XMLostream &w) const {
    $parent::write(w); $writefields
  }
$welcomedef
  $2
  "
}
```

⟨*ASTNode class* 2⟩

```
⟨Symbols 3a⟩
⟨SymbolTable 6⟩
⟨Expression classes 7a⟩
⟨Module classes 10c⟩
⟨Statements 12d⟩
⟨Run classes 18⟩
⟨low-level classes 15b⟩
⟨GRC classes 21⟩
⟨GRC graph class 20⟩

######################################################################

echo "#ifndef _AST_HPP
#  define _AST_HPP

/* Automatically generated by AST.sh -- do not edit */

#  include \"IR.hpp\"
#  include <string>
#  include <vector>
#  include <map>
#  include <cassert>
#  include <set>

namespace AST {
  using IR::Node;
  using IR::XMListream;
  using IR::XMLostream;
  using std::string;
  using std::vector;
  using std::map;

  class Visitor;
$forwarddefs

  union Status {
    int i;
    ASTNode *n;
    Status() {}
    Status(int ii) : i(ii) {}
    Status(ASTNode *nn) : n(nn) {}
  };

$classdefs

  class Visitor {
  public:
  virtual ~Visitor() {}
$visitorclassdefs
  };
```

```
}

#endif
" > AST.hpp

echo "/* Automatically generated by AST.sh -- do not edit */
#include \"AST.hpp\"
namespace AST {

$methoddefs

}
" > AST.cpp
```