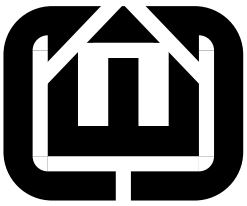
CEC Esterel Prettyprinter



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

Contents

Each `visit` method assumes it gets control right where it should start printing, i.e., that whatever called it added the right number of spaces for indentation. It should print its contents and no following spaces or newlines.

The symbol classes deviate from this: they print a `;` and a newline.

1 ⟨*EsterelPrinter.hpp* 1⟩≡

```
#ifndef _ESTEREL_PRINTER_HPP
#  define _ESTEREL_PRINTER_HPP

#  include "AST.hpp"
#  include <iostream>
#  include <vector>
#  include <map>

namespace AST {

  using std::vector;
  using std::map;

  class EsterelPrinter : public Visitor {
    std::ostream &o;

    unsigned int indentlevel;

    std::vector<int> precedence;
    std::map<string, int> level;
    std::map<string, int> unarylevel;
```

```
public:
  EsterelPrinter(std::ostream &);
  virtual ~EsterelPrinter() {}

  using Visitor::visit; // Bring the Visitor's visit method into scope

  void print(ASTNode* n) { assert(n); n->welcome(*this); }
  void statement(Statement *);
  void expression(Expression *);
  void sigexpression(Expression *);

  static const int sequentialPrecedence = 2;
  static const int parallelPrecedence = 1;

  bool push_precedence(int);
  void pop_precedence();

  Status visit(ModuleSymbol&);
  Status visit(VariableSymbol&);
  Status visit(BuiltinConstantSymbol&);
  Status visit(BuiltinSignalSymbol&);
  Status visit(BuiltinTypeSymbol&);
  Status visit(BuiltinFunctionSymbol&);
  Status visit(TypeRenaming &);
  Status visit(ConstantRenaming &);
  Status visit(FunctionRenaming &);
  Status visit(ProcedureRenaming &);
  Status visit(SignalRenaming &);
  Status visit(PredicatedStatement &);
  Status visit(TaskCall &);

  Status visit(Module&);
  Status visit(Exclusion&);
  Status visit(Implication&);
  Status visit(Modules&);
  Status visit(SymbolTable&);

  Status visit(TypeSymbol&);
  Status visit(ConstantSymbol&);
  Status visit(SignalSymbol&);
  Status visit(FunctionSymbol&);
  Status visit(ProcedureSymbol&);
  Status visit(TaskSymbol&);

  Status visit(StatementList&);
  Status visit(ParallelStatementList&);

  Status visit(Nothing&);
  Status visit(Pause&);
```

```
    Status visit(Halt&);
    Status visit(Emit&);
    Status visit(Sustain&);
    Status visit(Assign&);
    Status visit(StartCounter&);
    Status visit(ProcedureCall&);
    Status visit(Exec&);
    Status visit(Present&);
    Status visit(If&);
    Status visit(Loop&);
    Status visit(Repeat&);
    Status visit(Abort&);
    Status visit(Await&);
    Status visit(LoopEach&);
    Status visit(Every&);
    Status visit(Suspend&);
    Status visit(DoWatching&);
    Status visit(DoUpto&);
    Status visit(Trap&);
    Status visit(Exit&);
    Status visit(Var&);
    Status visit(Signal&);
    Status visit(Run&);

    Status visit(UnaryOp&);
    Status visit(BinaryOp&);
    Status visit(LoadVariableExpression&);
    Status visit(LoadSignalExpression&);
    Status visit(LoadSignalValueExpression&);
    Status visit(Literal&);
    Status visit(FunctionCall&);
    Status visit(Delay&);
    Status visit(CheckCounter&);

    Status visit(IfThenElse&);

    void indent() { indentlevel += 2; }
    void unindent() { indentlevel -= 2; }
    void tab() { for (unsigned int i = 0 ; i < indentlevel ; i++) o << ' '; }
  };
}

#endif
```

4 ⟨*EsterelPrinter.cpp* 4⟩≡

```
#include "EsterelPrinter.hpp"
#include <cassert>

namespace AST {

  EsterelPrinter::EsterelPrinter(std::ostream &oo) : o(oo), indentlevel(0) {
    precedence.push_back(0);

  level["or"] = 1;
  level["and"] = 2;
  unarylevel["not"] = 3;

  level["="] = 4;
  level["<>"] = 4;
  level["<"] = 4;
  level[">"] = 4;
  level["<="] = 4;
  level[">="] = 4;

  level["+"] = 5;
  level["-"] = 5;

  level["*"] = 6;
  level["/"] = 6;
  level["mod"] = 6;

  unarylevel["-"] = 7;
  }

  void EsterelPrinter::statement(Statement *s) {
    assert(s);
    o << '\n';
    indent();
    tab();
    s->welcome(*this);
    o << '\n';
    unindent();
    tab();
  }

  void EsterelPrinter::expression(Expression *e) {
    precedence.push_back(0);
    print(e);
    precedence.pop_back();
  }

  void EsterelPrinter::sigexpression(Expression *e) {
    precedence.push_back(0);
    // write [ ] or not ?
```

```
  if (dynamic_cast<Delay*>(e) || dynamic_cast<LoadSignalExpression*>(e))
      print(e);
  else {
      o<<'[';
      print(e);
      o<<']';
  }
  precedence.pop_back();
}

bool EsterelPrinter::push_precedence(int p) {
  bool needBrackets = p < precedence.back();
  precedence.push_back(p);
  return needBrackets;
}

void EsterelPrinter::pop_precedence() {
  precedence.pop_back();
}

Status EsterelPrinter::visit(ModuleSymbol &) { assert(0); }
Status EsterelPrinter::visit(VariableSymbol &) { assert(0); }
Status EsterelPrinter::visit(TypeRenaming &) { assert(0); }
Status EsterelPrinter::visit(ConstantRenaming &) { assert(0); }
Status EsterelPrinter::visit(FunctionRenaming &) { assert(0); }
Status EsterelPrinter::visit(ProcedureRenaming &) { assert(0); }
Status EsterelPrinter::visit(SignalRenaming &) { assert(0); }
Status EsterelPrinter::visit(PredicatedStatement &) { assert(0); }
Status EsterelPrinter::visit(TaskCall &) { assert(0); }

Status EsterelPrinter::visit(BuiltinConstantSymbol &) { return Status(); }
Status EsterelPrinter::visit(BuiltinSignalSymbol &) { return Status(); }
Status EsterelPrinter::visit(BuiltinTypeSymbol &) { return Status(); }
Status EsterelPrinter::visit(BuiltinFunctionSymbol &) { return Status(); }

Status EsterelPrinter::visit(Module &m) {
  assert(m.symbol);
  tab();
  o << "module " << m.symbol->name << ":\n";
  print(m.types);
  print(m.constants);
  print(m.functions);
  print(m.procedures);
  print(m.tasks);
  print(m.signals);
  for ( vector<InputRelation*>::const_iterator i = m.relations.begin() ;
        i != m.relations.end() ; i++ ) {
    assert(*i);
    print(*i);
  }
```

```
  o << '\n';
  tab();
  print(m.body);
  o << "\n\n";
  tab();
  o << "end module\n";
  return Status();
}

Status EsterelPrinter::visit(Modules &m) {
  vector<Module*>::iterator i = m.modules.begin();
  while ( i != m.modules.end() ) {
    assert(*i);
    assert((*i)->symbol);
    //      assert(m.module_symbols.contains((*i)->symbol->name));
    print(*i);
    i++;
    if ( i != m.modules.end() ) o << '\n';
  }
  return Status();
}

Status EsterelPrinter::visit(Exclusion& e) {
  o << "relation ";
  vector<SignalSymbol*>::const_iterator i = e.signals.begin();
  while ( i != e.signals.end() ) {
    assert(*i);
    o << (*i)->name;
    i++;
    if ( i != e.signals.end() ) o << " # ";
  }
  o << ";\n";
  return Status();
}

Status EsterelPrinter::visit(Implication& e) {
  assert(e.predicate);
  assert(e.implication);
  o << "relation ";
  o << e.predicate->name << " => " << e.implication->name << ";\n";
  return Status();
}

Status EsterelPrinter::visit(SymbolTable &t) {
  for ( SymbolTable::const_iterator i = t.begin() ; i != t.end() ; i++ ) {
    print(*i);
  }
  return Status();
}
```

```
Status EsterelPrinter::visit(TypeSymbol &s) {
  o << "type " << s.name << ";\n";
  return Status();
}

Status EsterelPrinter::visit(ConstantSymbol &s) {
  o << "constant " << s.name;
  if (s.initializer) {
    o << " = ";
    expression(s.initializer);
  }
  assert(s.type);
  o << " : " << s.type->name << ";\n";
  return Status();
}

Status EsterelPrinter::visit(SignalSymbol &s) {
  switch (s.kind) {
  case SignalSymbol::Input: o << "input"; break;
  case SignalSymbol::Output: o << "output"; break;
  case SignalSymbol::Inputoutput: o << "inputoutput"; break;
  case SignalSymbol::Sensor: o << "sensor"; break;
  case SignalSymbol::Return: o << "return"; break;
  default: assert(0); break;
  }
  o << ' ' << s.name;
  if (s.initializer) {
    o << " := ";
    expression(s.initializer);
    assert(s.type);
    assert(s.value);
  }
  if (s.type) {
    o << " : ";
    if (s.combine) o << "combine ";
    o << s.type->name;
    if (s.combine) o << " with " << s.combine->name;
    assert(s.value);
  } else {
    assert(!s.value);
  }
  o << ";\n";
  return Status();
}

Status EsterelPrinter::visit(FunctionSymbol &s) {
  o << "function " << s.name << '(';
  vector<TypeSymbol*>::const_iterator i = s.arguments.begin();
  while (i != s.arguments.end()) {
    assert(*i);
```

```
    o << (*i)->name;
    i++;
    if (i != s.arguments.end()) o << ", ";
  }
  assert(s.result);
  o << ") : " << s.result->name << ";\n";
  return Status();
}

Status EsterelPrinter::visit(ProcedureSymbol &s) {
  o << "procedure " << s.name << '(';
  vector<TypeSymbol*>::const_iterator i = s.reference_arguments.begin();
  while (i != s.reference_arguments.end()) {
    assert(*i);
    o << (*i)->name;
    i++;
    if (i != s.reference_arguments.end()) o << ", ";
  }
  o << ")(";
   i = s.value_arguments.begin();
  while (i != s.value_arguments.end()) {
    assert(*i);
    o << (*i)->name;
    i++;
    if (i != s.value_arguments.end()) o << ", ";
  }
  o << ");\n";
  return Status();
}

Status EsterelPrinter::visit(TaskSymbol &s) {
  o << "task " << s.name << '(';
  vector<TypeSymbol*>::const_iterator i = s.reference_arguments.begin();
  while (i != s.reference_arguments.end()) {
    assert(*i);
    o << (*i)->name;
    i++;
    if (i != s.reference_arguments.end()) o << ", ";
  }
  o << ")(";
   i = s.value_arguments.begin();
  while (i != s.value_arguments.end()) {
    assert(*i);
    o << (*i)->name;
    i++;
    if (i != s.value_arguments.end()) o << ", ";
  }
  o << ");\n";
  return Status();
}
```

```
Status EsterelPrinter::visit(StatementList &l) {
  push_precedence(sequentialPrecedence);
  vector<Statement*>::const_iterator i = l.statements.begin();
  while ( i != l.statements.end() ) {
    print(*i);
    i++;
    if (i != l.statements.end() ) { o << ";\n"; tab(); }
  }
  pop_precedence();
  return Status();
}

Status EsterelPrinter::visit(ParallelStatementList &l) {
  bool needBrackets = push_precedence(parallelPrecedence);

  if (needBrackets) {
    o << "[\n";
    tab();
  }

  vector<Statement*>::const_iterator i = l.threads.begin();
  while ( i != l.threads.end() ) {
    indent();
    o << "  "; // indent doesn't take effect until the next tab()
    print(*i);
    unindent();
    i++;
    if (i != l.threads.end() ) { o << '\n'; tab(); o << "||\n"; tab(); }
  }

  if (needBrackets) {
    o << '\n';
    tab();
    o << ']';
  }

  pop_precedence();
  return Status();
}

Status EsterelPrinter::visit(Nothing &) { o << "nothing"; return Status(); }
Status EsterelPrinter::visit(Pause &) { o << "pause"; return Status(); }
Status EsterelPrinter::visit(Halt &) { o << "halt"; return Status(); }

Status EsterelPrinter::visit(Emit &e) {
  assert(e.signal);
  o << "emit " << e.signal->name;
  if (e.value) {
    o << '(';
```

```
    expression(e.value);
    o << ')';
  }
  return Status();
}

Status EsterelPrinter::visit(StartCounter &s) {
  assert(s.counter);
  o << "StCnt ";
  expression(s.counter->countvalue);
  o <<" ";
  expression(s.counter->predicate);
  return Status();
}

Status EsterelPrinter::visit(Sustain &e) {
  assert(e.signal);
  o << "sustain " << e.signal->name;
  if (e.value) {
    o << '(';
    expression(e.value);
    o << ')';
  }
  return Status();
}

Status EsterelPrinter::visit(Assign &a) {
  assert(a.variable);
  assert(a.value);
  o << a.variable->name << " := ";
  expression(a.value);
  return Status();
}

Status EsterelPrinter::visit(ProcedureCall &c) {
  assert(c.procedure);
  o << "call " << c.procedure->name << '(';
  vector<VariableSymbol*>::const_iterator i = c.reference_args.begin();
  while (i != c.reference_args.end()) {
    o << (*i)->name;
    i++;
    if (i != c.reference_args.end()) o << ", ";
  }
  o << ")(";
  vector<Expression*>::const_iterator j = c.value_args.begin();
  while (j != c.value_args.end()) {
    expression(*j);
    j++;
    if (j != c.value_args.end()) o << ", ";
  }
```

```
  o << ')';
  return Status();
}

Status EsterelPrinter::visit(Exec &e) {
  o << "exec";
  indent();
  for ( vector<TaskCall*>::const_iterator i = e.calls.begin() ;
        i != e.calls.end() ; i++) {
    assert(*i);
    assert((*i)->procedure);
    o << '\n';
    tab();
    o << "case " << (*i)->procedure->name << '(';
    vector<VariableSymbol*>::const_iterator k = (*i)->reference_args.begin();
    while (k != (*i)->reference_args.end()) {
      o << (*k)->name;
      k++;
      if (k != (*i)->reference_args.end()) o << ", ";
    }
    o << ")(";
    vector<Expression*>::const_iterator j = (*i)->value_args.begin();
    while (j != (*i)->value_args.end()) {
      expression(*j);
      j++;
      if (j != (*i)->value_args.end()) o << ", ";
    }
    assert((*i)->signal);
    o << ") return " << (*i)->signal->name;
    if ((*i)->body) {
      o << " do\n";
      indent();
      tab();
      print((*i)->body);
      unindent();
    }
  }
  o << '\n';
  unindent();
  tab();
  o << "end exec";
  return Status();
}

Status EsterelPrinter::visit(Present &p) {
  o << "present ";
  if (p.cases.size() == 1) {
    // Simple if-then-else
    PredicatedStatement *ps = p.cases[0];
    assert(ps);
```

```
    sigexpression(ps->predicate);
    o << ' ';
    if (ps->body) {
      o << "then";
      statement(ps->body);
    }
  } else {
    // Multiple cases
    indent();
    for (vector<PredicatedStatement*>::const_iterator i = p.cases.begin() ;
         i != p.cases.end() ; i++) {
      o << '\n';
      tab();
      o << "case ";
      assert(*i);
      assert((*i)->predicate);
      sigexpression((*i)->predicate);
      if ((*i)->body) {
        o << " do\n";
        indent();
        tab();
        print((*i)->body);
        unindent();
      }
    }
    unindent();
    o << '\n';
    tab();
  }
  if (p.default_stmt) {
    o << "else";
    statement(p.default_stmt);
  }
  o << "end present";
  return Status();
}

Status EsterelPrinter::visit(If &s) {
  o << "if ";
  vector<PredicatedStatement*>::const_iterator i = s.cases.begin();
  assert(i != s.cases.end());
  expression((*i)->predicate);
  o << ' ';
  if ((*i)->body) {
    o << "then";
    statement((*i)->body);
  }
  for ( i++ ; i != s.cases.end() ; i++ ) {
    assert(*i);
    o << "elsif ";
```

```
    expression((*i)->predicate);
    o << " then";
    statement((*i)->body);
  }
  if (s.default_stmt) {
    o << "else";
    statement(s.default_stmt);
  }
  o << "end if";
  return Status();
}

Status EsterelPrinter::visit(Loop &l) {
  o << "loop";
  statement(l.body);
  o << "end loop";
  return Status();
}

Status EsterelPrinter::visit(Repeat &r) {
  if (r.is_positive) o << "positive ";
  o << "repeat ";
  expression(r.count);
  o << " times";
  statement(r.body);
  o << "end repeat";
  return Status();
}

Status EsterelPrinter::visit(Abort &a) {
  if (a.is_weak) o << "weak ";
  o << "abort";
  statement(a.body);
  o << "when";
  if (a.cases.size() == 1) {
    // Simple abort condition
    PredicatedStatement *ps = a.cases[0];
    o << ' ';
    assert(ps);
    sigexpression(ps->predicate);
    if (ps->body) {
      o << " do";
      statement(ps->body);
      goto PrintEnd;
    }
  } else {
    // Abort cases
    indent();
    for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
          i != a.cases.end() ; i++ ) {
```

```
      o << '\n';
      tab();
      assert(*i);
      o << "case ";
      sigexpression((*i)->predicate);
      if ((*i)->body) {
        o << " do\n";
        indent();
        tab();
        print((*i)->body);
        unindent();
      }
    }
    unindent();
    o << '\n';
    tab();
  PrintEnd:
    o << "end";
    if (a.is_weak) o << " weak";
    o << " abort";
  }
  return Status();
}

Status EsterelPrinter::visit(Await& a) {
  o << "await ";
  if (a.cases.size() == 1) {
    // Simple abort condition
    PredicatedStatement *ps = a.cases[0];
    assert(ps);
    sigexpression(ps->predicate);
    if (ps->body) {
      o << " do";
      statement(ps->body);
      o << "end await";
    }
  } else {
    indent();
    for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
          i != a.cases.end() ; i++ ) {
      o << '\n';
      tab();
      assert(*i);
      o << "case ";
      sigexpression((*i)->predicate);
      if ((*i)->body) {
        o << " do\n";
        indent();
        tab();
        print((*i)->body);
```

```
          unindent();
        }
      }
      unindent();
      o << '\n';
      tab();
      o << "end await";
    }
    return Status();
  }

  Status EsterelPrinter::visit(LoopEach &l) {
    o << "loop";
    statement(l.body);
    o << "each ";
    sigexpression(l.predicate);
    return Status();
  }

  Status EsterelPrinter::visit(Every &e) {
    o << "every ";
    sigexpression(e.predicate);
    o << " do";
    statement(e.body);
    o << "end every";
    return Status();
  }

  Status EsterelPrinter::visit(Suspend &s) {
    o << "suspend";
    statement(s.body);
    o << "when ";
    sigexpression(s.predicate);
    return Status();
  }

  Status EsterelPrinter::visit(DoWatching &d) {
    o << "do";
    statement(d.body);
    o << "watching ";
    sigexpression(d.predicate);
//    o << ']';
    if (d.timeout) {
      o << " timeout";
      statement(d.timeout);
      o << "end timeout";
    }
    return Status();
  }
```

```
Status EsterelPrinter::visit(DoUpto &d) {
  o << "do";
  statement(d.body);
  o << "upto ";
  sigexpression(d.predicate);
  return Status();
}

Status EsterelPrinter::visit(Trap &t) {
  assert(t.symbols);
  o << "trap ";
  SymbolTable::const_iterator i = t.symbols->begin();
  while ( i != t.symbols->end() ) {
    Symbol *s = *i;
    assert(s);
    SignalSymbol *ts = dynamic_cast<SignalSymbol*>(s);
    assert(ts);
    assert(ts->kind == SignalSymbol::Trap);
    o << ts->name;
    if (ts->initializer) {
      o << " := ";
      expression(ts->initializer);
    }
    if (ts->type) {
      o << " : " << ts->type->name;
    }
    i++;
    if ( i != t.symbols->end() ) {
      o << ",\n";
      tab();
      o << "     ";
    }
  }
  o << " in";
  statement(t.body);
  for (vector<PredicatedStatement*>::const_iterator i = t.handlers.begin() ;
       i != t.handlers.end() ; i++ ) {
    assert(*i);
    o << "handle ";
    expression((*i)->predicate);
    o << " do";
    statement((*i)->body);
  }
  o << "end trap";
  return Status();
}

Status EsterelPrinter::visit(Exit &e) {
  o << "exit ";
  assert(e.trap);
```

```
  assert(e.trap->kind == SignalSymbol::Trap);
  o << e.trap->name;
  if (e.value) {
    o << '(';
    expression(e.value);
    o << ')';
  }
  return Status();
}

Status EsterelPrinter::visit(Var &v) {
  o << "var ";
  SymbolTable::const_iterator i = v.symbols->begin();
  while ( i != v.symbols->end() ) {
    Symbol *s = *i;
    assert(s);
    VariableSymbol *vs = dynamic_cast<VariableSymbol*>(s);
    assert(vs);
    o << vs->name;
    if (vs->initializer) {
      o << " := ";
      expression(vs->initializer);
    }
    assert(vs->type);
    o << " : " << vs->type->name;
    i++;
    if ( i != v.symbols->end() ) {
      o << ",\n";
      tab();
      o << "    ";
    }
  }
  o << " in";
  statement(v.body);
  o << "end var";
  return Status();
}

Status EsterelPrinter::visit(Signal &s) {
  o << "signal ";
  SymbolTable::const_iterator i = s.symbols->begin();
  while ( i != s.symbols->end() ) {
    Symbol *sy = *i;
    assert(sy);
    SignalSymbol *ss = dynamic_cast<SignalSymbol*>(sy);
    assert(ss);
    o << ss->name;
    if (ss->initializer) {
      o << " := ";
      expression(ss->initializer);
```

```
    }
    if (ss->type) {
      o << " : ";
      if (ss->combine) o << "combine ";
      o << ss->type->name;
      if (ss->combine) o << " with " << ss->combine->name;
    }
    i++;
    if ( i != s.symbols->end() ) {
      o << ",\n";
      tab();
      o << "       ";
    }
  }
  o << " in";
  statement(s.body);
  o << "end signal";
  return Status();
}

Status EsterelPrinter::visit(Run &r) {
  o << "run " << r.new_name;
  if (r.old_name != r.new_name) o << " / " << r.old_name;
  if ( r.types.size() + r.constants.size() + r.functions.size() +
       r.procedures.size() + r.tasks.size() + r.signals.size() > 0 ) {
    o << " [\n";
    indent();
    tab();

    bool more = false;

    if (r.types.size() > 0) {
      o << "type ";
      vector<TypeRenaming*>::const_iterator i = r.types.begin();
      while (i != r.types.end()) {
        assert(*i);
        assert((*i)->new_type);
        o << (*i)->new_type->name << " / " << (*i)->old_name;
        i++;
        if (i != r.types.end()) {
          o << ",\n";
          tab();
          o << "     ";
        }
      }
      more = true;
    }

    if (r.constants.size() > 0 ) {
      if (more) {
```

```
    o << ";\n";
    tab();
  }
  o << "constant ";
  vector<ConstantRenaming*>::const_iterator i = r.constants.begin();
  while (i != r.constants.end()) {
    assert(*i);
    assert((*i)->new_value);
    expression((*i)->new_value);
    o << " / " << (*i)->old_name;
    i++;
    if (i != r.constants.end()) {
      o << ",\n";
      tab();
      o << "          ";
    }
  }
  more = true;
}

if (r.functions.size() > 0) {
  if (more) {
    o << ";\n";
    tab();
  }
  o << "function ";
  vector<FunctionRenaming*>::const_iterator i = r.functions.begin();
  while (i != r.functions.end()) {
    assert(*i);
    assert((*i)->new_func);
    o << (*i)->new_func->name << " / " << (*i)->old_name;
    i++;
    if (i != r.functions.end()) {
      o << ",\n";
      tab();
      o << "          ";
    }
  }
  more = true;
}

if (r.procedures.size() > 0) {
  if (more) {
    o << ";\n";
    tab();
  }
  o << "procedure ";
  vector<ProcedureRenaming*>::const_iterator i = r.procedures.begin();
  while (i != r.procedures.end()) {
    assert(*i);
```

```
      assert((*i)->new_proc);
      o << (*i)->new_proc->name << " / " << (*i)->old_name;
      i++;
      if (i != r.procedures.end()) {
        o << ",\n";
        tab();
        o << "          ";
      }
    }
    more = true;
  }

  if (r.tasks.size() > 0) {
    if (more) {
      o << ";\n";
      tab();
    }
    o << "task ";
    vector<ProcedureRenaming*>::const_iterator i = r.tasks.begin();
    while (i != r.tasks.end() ) {
      assert(*i);
      assert((*i)->new_proc);
      o << (*i)->new_proc->name << " / " << (*i)->old_name;
      i++;
      if (i != r.tasks.end()) {
        o << ",\n";
        tab();
        o << "     ";
      }
    }
    more = true;
  }

  if (r.signals.size() > 0) {
    if (more) {
      o << ";\n";
      tab();
    }
    o << "signal ";
    vector<SignalRenaming*>::const_iterator i = r.signals.begin();
    while (i != r.signals.end() ) {
      assert(*i);
      assert((*i)->new_sig);
      o << (*i)->new_sig->name << " / " << (*i)->old_name;
      i++;
      if (i != r.signals.end()) {
        o << ",\n";
        tab();
        o << "       ";
      }
```

```
      }
    }
    o << " ]";
    unindent();
  }
  return Status();
}

Status EsterelPrinter::visit(UnaryOp &u) {
  assert(unarylevel.find(u.op) != unarylevel.end());
  push_precedence(unarylevel[u.op]); // Highest: parenthesis never needed
  o << u.op;
  o << ' ';
  print(u.source);
  pop_precedence();
  return Status();
}

Status EsterelPrinter::visit(BinaryOp &b) {
  assert(level.find(b.op) != level.end());
  bool needParen = push_precedence(level[b.op]);
  if (needParen) o << '(';
  print(b.source1);
  o << ' ' << b.op << ' ';
  print(b.source2);
  if (needParen) o << ')';
  pop_precedence();
  return Status();
}

Status EsterelPrinter::visit(LoadVariableExpression &e) {
  assert(e.variable);
  o << e.variable->name;
  return Status();
}

Status EsterelPrinter::visit(LoadSignalExpression &e) {
  assert(e.signal);
  o << e.signal->name;
  return Status();
}

Status EsterelPrinter::visit(LoadSignalValueExpression &e) {
  assert(e.signal);
  if (e.signal->kind == SignalSymbol::Trap) o << '?';
  o << '?' << e.signal->name;
  return Status();
}

Status EsterelPrinter::visit(Literal &l) {
```

```
    assert(l.type);
    if (l.type->name == "string") {
      o << '\"';
      for (string::iterator i = l.value.begin() ; i != l.value.end() ; i++) {
        if (*i == '\"') o << '\"';
        o << *i;
      }
      o << '\"';
    }
    else o << l.value;
    return Status();
  }

  Status EsterelPrinter::visit(FunctionCall &e) {
    assert(e.callee);
    o << e.callee->name;
    o << '(';
    vector<Expression*>::const_iterator i = e.arguments.begin();
    while (i != e.arguments.end()) {
      expression(*i);
      i++;
      if ( i != e.arguments.end()) o << ", ";
    }
    o << ')';
    return Status();
  }

  Status EsterelPrinter::visit(Delay &e) {
    if (e.is_immediate) {
      assert(e.count == NULL);
      o << "immediate ";
    } else {
      expression(e.count);
      o << ' ';
    }
//    o << '[';
    sigexpression(e.predicate);
//    o << ']';
    return Status();
  }

  Status EsterelPrinter::visit(CheckCounter &e) {
    assert(e.counter);
    o<<" ChkCnt ";
    expression(e.counter->countvalue);
    o<<" ";
    expression(e.counter->predicate);
    return Status();
  }
```

```
  Status EsterelPrinter::visit(IfThenElse &s) {
    o << "if (";
    expression(s.predicate);
    o << ")";
    if (s.then_part) {
      o << " {";
      statement(s.then_part);
      o << "}";
    }
    if (s.else_part) {
      o << " else {";
      statement(s.else_part);
      o << "}";
    }
    return Status();
  }

}
```