CallGraph

Shin Sahara

SCSK Corporation,
Hosei University

目次

1 AnalyserCommon

1.1 Responsibility

Utilities of analyser.

```
class
AnalyserCommon
values
public
  DbgLevel = 5;
  fileName_of_CallGraph = "CallGraph.txt";
  levelString = " − − − − − − − − − − − − − − − − − − − − − ";
protected
  io = new IO();
protected
  util = new VDMUtil()
types
  public IdentifierName = seq of char
functions
protected
  MakeCallTree[@T] : AnalysedObject * @T * [seq of char * nat * nat * IdentifierName *
IdentifierName] +> bool
  MakeCallTree(anAnalysedObject, val, p) ==
    let wClassName = anAnalysedObject.GetClassName(),
        mk_(keyName, dispName) = getNames[@T](wClassName, val) in
    (if anAnalysedObject.GetOutputLevel() <= 7 and not anAnalysedObject.IsRoutineName(dispName)
     then false
     else let wLevel = anAnalysedObject.GetLevel(),
              lineNum_set = anAnalysedObject.GetLineNum(keyName),
              wDefinedLineNum = anAnalysedObject.GetDefinedLineNum(dispName),
              mk_(-, line_num, column_num, -, fnop_name) = p,
              wSourceInfos =
                  if DbgLevel >= 9
                  then util.val2seq_of_char[nat * nat * IdentifierName](mk_(line_num, column_num, fnop_name
                  else "" in
          anAnalysedObject.SetCallGraphData(keyName, dispName, wLevel, wDefinedLineNum, lineNum_set, wSourc
operations
protected
```

```
  prCallTree2 : AnalysedObject ==> ()
  prCallTree2 (anAnalysedObject) ==
    for wCG in anAnalysedObject.GetCallGraphDataSeq ()
    do let wDispName = wCG.fDispName in
       let wLevel = wCG.fLevel,
           indents = getIndents (wLevel),
           wDefinedLineNum = wCG.fDefinedLineNum,
           wDefinedLineNums = util.val2seq_of_char[set of nat] (wDefinedLineNum),
           wlineNum_set = wCG.fLineNum_set,
           wLineNums = util.val2seq_of_char[set of nat] (wlineNum_set),
           wSourceInfos = wCG.fSourceInfos,
           s9 = indents^wDispName^wDefinedLineNums^wLineNums^"\t"^wSourceInfos^"\n",
           - = printf (s9) in
       skip
functions
protected
  print[@T] : AnalysedObject * @T +> bool
  print (anAnalysedObject, val) ==
    let wLevel = anAnalysedObject.GetLevel (),
        indents = getIndents (wLevel + 1),
        s = indents^util.val2seq_of_char[@T] (val)^"\n",
        - = io.fecho (fileName_of_CallGraph, s, <append> ) in
    io.echo (s);
protected
  printf : seq of char +> bool
  printf (s) ==
    let - = io.fecho (fileName_of_CallGraph, s, <append> ) in
    io.echo (s);
protected
  printHeader : seq of IdentifierName -> bool
  printHeader (aClassName_seq) ==
    def s = "+++ CallGraph +++";
        - = io.echo (s);
        - = io.fecho (fileName_of_CallGraph, s, <start> );
        wClassName_str = util.val2seq_of_char[seq of IdentifierName] (aClassName_seq) in
    r;
public
```

```
  pr : seq of char -> bool
  pr (s) ==
    io.echo (s);
  getIndents : AnalysedObject`Level -> seq of char
  getIndents (aLevel) ==
    if aLevel > 0
    then levelString (1,...,aLevel)
    else "";
  getNames[@T] : IdentifierName * @T -> IdentifierName * IdentifierName
  getNames (aClassName,val) ==
    if is_(val,IdentifierName)
    then mk_ (aClassName^"`"^val,val)
    elseif is_(val,seq of IdentifierName)
    then if len val = 2
         then mk_ (hd val^"`"^hd tl val,hd val^"`"^hd tl val)
         else mk_ (aClassName^"`"^hd val,hd val)
    else mk_ (aClassName^"`"^val,val);
protected
  GetName : AS`Name -> AS`Ids
  GetName (aNAME) ==
    let mk_AS`Name (ids,-) = aNAME in
    ids
end
AnalyserCommon
```

Test Suite : vdm.tc

Class : AnalyserCommon

Name	#Calls	Coverage
AnalyserCommon`pr	28	√
AnalyserCommon`print	0	0%
AnalyserCommon`printf	738	√
AnalyserCommon`GetName	1158	√
AnalyserCommon`getNames	5758	86%
AnalyserCommon`getIndents	737	√
AnalyserCommon`prCallTree2	1	√
AnalyserCommon`printHeader	1	√
AnalyserCommon`MakeCallTree	5758	√
Total Coverage		**90%**

2 Analyser

2.1 Responsibility

Analyse VDM++ specifications.

```
class
Analyser is subclass of AnalyserCommon
instance variables
  protected iWzd : VDMToolsWizard := new VDMToolsWizard();

operations
public
  analyseClass : AnalysedObject * IdentifierName ==> seq of AnalysedObject
  analyseClass(anAnalysedObject, aClassName) ==
    def - = pr("Analysing "^aClassName^"...\n");
        - = anAnalysedObject.ClearLevel();
        wClass = iWzd.classAST(aClassName);
        wClassToFileMap = iWzd.classToFileMap();
        wTokenInfo_seq = iWzd.docTokenInfoSeq(wClassToFileMap(aClassName));
        wAnalysedObject = anAnalysedObject.SetTokenInfo(wTokenInfo_seq);
        wContextNodeInfo_seq = iWzd.docContextNodeInfoSeq(wClassToFileMap(aClassName), nil);
        wAnalysedObject1 = wAnalysedObject.SetContextNodeInfo(wContextNodeInfo_seq);
        wAnalysedObject2 = wAnalysedObject1;
        wAnalysedObject3 = wAnalysedObject2.SetName2LineNum(aClassName);
        defs = wClass.defs;
        nm = wClass.nm;
        ids = GetName(nm);
        wLineNum = wAnalysedObject3.GetLine(nm.cid);
        wAnalysedObject4 = wAnalysedObject3;
        wAnalysedObject5 = wAnalysedObject4.SetRoutineName2DefinedLineNum(ids, wLineNum);
        - = MakeCallTree[IdentifierName](wAnalysedObject5, aClassName, util.P());
        r = analyseDefinitions(wAnalysedObject5, defs);
        - = wAnalysedObject5.ClearLevel() in
    return r
functions
```

```
analyseDefinitions : AnalysedObject * AS`Definitions -> seq of AnalysedObject
analyseDefinitions (anAnalysedObject, aDefinitions) ==
  let valuem = aDefinitions.valuem,
      fnm = aDefinitions.fnm,
      opm = aDefinitions.opm,
      cnm = anAnalysedObject.GetClassName (),
      wValue_sec = analyseValueDefs (anAnalysedObject, cnm, valuem),
      wFunc_seq = analyseFnms (anAnalysedObject, fnm),
      wOp_seq = analyseOpms (anAnalysedObject, opm),
      r = wFunc_seq^wOp_seq^wValue_sec in
  r;
analyseFnms : AnalysedObject * map AS`Name to AS`FnDef -> seq of AnalysedObject
analyseFnms (anAnalysedObject, aFnm) ==
  let wFnDef_seq = util.set2seq[AS`FnDef] (rng aFnm),
      r = conc [analyseFnm (anAnalysedObject, wFnDef_seq (i))|i in set inds wFnDef_seq] in
  r;
analyseOpms : AnalysedObject * map AS`Name to AS`OpDef -> seq of AnalysedObject
analyseOpms (anAnalysedObject, aOpm) ==
  let wOpDef_seq = util.set2seq[AS`OpDef] (rng aOpm),
      r = conc [analyseOpm (anAnalysedObject, wOpDef_seq (i))|i in set inds wOpDef_seq] in
  r;
analyseFnm : AnalysedObject * AS`FnDef -> seq of AnalysedObject
analyseFnm (anAnalysedObject, aFnDef) ==
  if is_AS`ExplFnDef (aFnDef)
  then analyseExplFnDef (anAnalysedObject, aFnDef)
  elseif is_AS`ImplFnDef (aFnDef)
  then analyseImplFnDef (anAnalysedObject, aFnDef)
  elseif is_AS`ExtExplFnDef (aFnDef)
  then analyseExtExplFnDef (anAnalysedObject, aFnDef)
  else [anAnalysedObject];
analyseOpm : AnalysedObject * AS`OpDef -> seq of AnalysedObject
analyseOpm (anAnalysedObject, aOpDef) ==
  if is_AS`ExplOpDef (aOpDef)
  then analyseExplOpDef (anAnalysedObject, aOpDef)
  elseif is_AS`ImplOpDef (aOpDef)
  then analyseImplOpDef (anAnalysedObject, aOpDef)
  elseif is_AS`ExtExplOpDef (aOpDef)
  then analyseExtExplOpDef (anAnalysedObject, aOpDef)
  else [anAnalysedObject];
```

```
analyseExplFnDef : AnalysedObject * AS`FnDef -> seq of AnalysedObject
analyseExplFnDef (anAnalysedObject, aFnDef) ==
  let tp = aFnDef.tp,
      body = aFnDef.body,
      nm = aFnDef.nm,
      fnpre = aFnDef.fnpre,
      fnpost = aFnDef.fnpost,
      ids = GetName (nm),
      wLineNum = anAnalysedObject.GetLine (nm.cid),
      wAnalysedObject = anAnalysedObject.SetRoutineName2DefinedLineNum (ids, wLineNum),
      wAnalysedObject2 = wAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject2, ids, util.P ()),
      - = if DbgLevel >= 8
          then print[AS`FnType] (wAnalysedObject2, tp)
          else false,
      r =
          analyseFnBody (wAnalysedObject2, ids, body)^
          analyseExpr (wAnalysedObject2, ids, fnpre)^
          analyseExpr (wAnalysedObject2, ids, fnpost),
      - = wAnalysedObject2.DecLevel () in
  r;
```

```
analyseImplFnDef : AnalysedObject * AS`FnDef -> seq of AnalysedObject
analyseImplFnDef (anAnalysedObject, aFnDef) ==
  let nm = aFnDef.nm,
      ids = GetName (nm),
      wLineNum = anAnalysedObject.GetLine (nm.cid),
      wAnalysedObject = anAnalysedObject.SetRoutineName2DefinedLineNum (ids, wLineNum),
      resnmtps = aFnDef.resnmtps,
      fnpre = aFnDef.fnpre,
      fnpost = aFnDef.fnpost,
      wAnalysedObject2 = wAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject2, ids, util.P ()),
      - = if DbgLevel >= 8
          then print[seq of AS`NameType] (wAnalysedObject2, resnmtps)
          else false,
      r =
          analyseExpr (wAnalysedObject2, ids, fnpre)^
          analyseExpr (wAnalysedObject2, ids, fnpost),
      - = wAnalysedObject.DecLevel () in
  r;
```

```
analyseExtExplFnDef : AnalysedObject * AS`FnDef -> seq of AnalysedObject
analyseExtExplFnDef (anAnalysedObject, aFnDef) ==
  let nm = aFnDef.nm,
      ids = GetName (nm),
      wLineNum = anAnalysedObject.GetLine (nm.cid),
      wAnalysedObject = anAnalysedObject.SetRoutineName2DefinedLineNum (ids, wLineNum),
      resnmtps = aFnDef.resnmtps,
      body = aFnDef.body,
      fnpre = aFnDef.fnpre,
      fnpost = aFnDef.fnpost,
      wAnalysedObject2 = wAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject2, ids, util.P ()),
      - = if DbgLevel >= 8
          then print[seq of AS`NameType] (wAnalysedObject2, resnmtps)
          else false,
      r =
          analyseFnBody (wAnalysedObject2, ids, body)^
          analyseExpr (wAnalysedObject2, ids, fnpre)^
          analyseExpr (wAnalysedObject2, ids, fnpost),
      - = wAnalysedObject2.DecLevel () in
  r;
analyseExplOpDef : AnalysedObject * AS`OpDef -> seq of AnalysedObject
analyseExplOpDef (anAnalysedObject, aOpDef) ==
  let nm = aOpDef.nm,
      ids = GetName (nm),
      wLineNum = anAnalysedObject.GetLine (nm.cid),
      wAnalysedObject = anAnalysedObject.SetRoutineName2DefinedLineNum (ids, wLineNum),
      body = aOpDef.body,
      oppre = aOpDef.oppre,
      oppost = aOpDef.oppost,
      wAnalysedObject2 = wAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject2, ids, util.P ()),
      r =
          analyseOpBody (wAnalysedObject2, ids, body)^
          analyseExpr (wAnalysedObject2, ids, oppre)^
          analyseExpr (wAnalysedObject2, ids, oppost),
      - = wAnalysedObject2.DecLevel () in
  r;
```

```
analyseImplOpDef : AnalysedObject * AS`OpDef -> seq of AnalysedObject
analyseImplOpDef (anAnalysedObject, aOpDef) ==
  let nm = aOpDef.nm,
      ids = GetName (nm),
      wLineNum = anAnalysedObject.GetLine (nm.cid),
      wAnalysedObject = anAnalysedObject.SetRoutineName2DefinedLineNum (ids, wLineNum),
      partps = aOpDef.partps,
      oppre = aOpDef.oppre,
      oppost = aOpDef.oppost,
      wAnalysedObject2 = wAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject2, ids, util.P ()),
      r1 = analyseParameterTypes (anAnalysedObject, ids, partps),
      r2 = analyseExpr (anAnalysedObject, ids, oppre),
      r3 = analyseExpr (anAnalysedObject, ids, oppost),
      r = r1^r2^r3,
      - = wAnalysedObject2.DecLevel () in
  r;
analyseExtExplOpDef : AnalysedObject * AS`OpDef -> seq of AnalysedObject
analyseExtExplOpDef (anAnalysedObject, aOpDef) ==
  let nm = aOpDef.nm,
      ids = GetName (nm),
      wLineNum = anAnalysedObject.GetLine (nm.cid),
      wAnalysedObject = anAnalysedObject.SetRoutineName2DefinedLineNum (ids, wLineNum),
      partps = aOpDef.partps,
      body = aOpDef.body,
      oppre = aOpDef.oppre,
      oppost = aOpDef.oppost,
      excps = aOpDef.excps,
      wAnalysedObject2 = wAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject2, ids, util.P ()),
      r1 = analyseOpBody (wAnalysedObject2, ids, body),
      r2 = analyseParameterTypes (anAnalysedObject, ids, partps),
      r3 = analyseExpr (wAnalysedObject2, ids, oppre),
      r4 = analyseExpr (wAnalysedObject2, ids, oppost),
      r5 = conc [analyseError (anAnalysedObject, ids, excps (i))|i in set inds excps],
      r = r1^r2^r3^r4^r5,
      - = wAnalysedObject2.DecLevel () in
  r;
```

```
analyseFnBody : AnalysedObject * AS`Ids * AS`FnBody -> seq of AnalysedObject
analyseFnBody (anAnalysedObject, anIds, aFnBody) ==
  if is_(aFnBody.body, AS`Expr)
  then analyseExpr (anAnalysedObject, anIds, aFnBody.body)
  else [anAnalysedObject];
analyseOpBody : AnalysedObject * AS`Ids * AS`OpBody -> seq of AnalysedObject
analyseOpBody (anAnalysedObject, anIds, aOpBody) ==
  if is_(aOpBody.body, AS`Stmt)
  then analyseStmt (anAnalysedObject, anIds, aOpBody.body)
  else [anAnalysedObject];
analyseExprs : AnalysedObject * AS`Ids * seq of [AS`Expr] -> seq of AnalysedObject
analyseExprs (anAnalysedObject, anIds, exprs) ==
  let r = conc [analyseExpr (anAnalysedObject, anIds, exprs (i))|i in set inds exprs] in
  r;
```

```
analyseExpr : AnalysedObject * AS`Ids * [AS`Expr] -> seq of AnalysedObject
analyseExpr (anAnalysedObject, anIds, anExpr) ==
  let wAnalysedObject = anAnalysedObject.IncLevel (),
      r =
          if is_(anExpr, AS`LetExpr)
          then analyseLetExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`LetBeSTExpr)
          then analyseLetBeSTExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`DefExpr)
          then analyseDefExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`ApplyExpr)
          then analyseApplyExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`IfExpr)
          then analyseIfExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`CasesExpr)
          then analyseCasesExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`FieldSelectExpr)
          then analyseFieldSelectExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`SetComprehensionExpr)
          then analyseSetComprehensionExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`SeqComprehensionExpr)
          then analyseSeqComprehensionExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`BracketedExpr)
          then analyseBracketedExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`UnaryExpr)
          then analyseUnaryExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`BinaryExpr)
          then analyseBinaryExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`SetEnumerationExpr)
          then analyseSetEnumerationExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`SeqEnumerationExpr)
          then analyseSeqEnumerationExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`Maplet)
          then analyseMaplet (anAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`MapEnumerationExpr)
          then analyseMapEnumerationExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`MapComprehensionExpr)
          then analyseMapComprehensionExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`QuantExpr)
          then analyseQuantExpr (wAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`FctTypeInstExpr)
          then analyseFctTypeInstExpr (anAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`IotaExpr)
          then analyseIotaExpr (anAnalysedObject, anIds, anExpr)
          elseif is_(anExpr, AS`SetRangeExpr)
          then analyseSetRangeExpr (wAnalysedObject, anIds, anExpr)
```

```
analyseStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseStmt (anAnalysedObject, anIds, aStmt) ==
  let wAnalysedObject = anAnalysedObject.IncLevel (),
      r =
          if is_(aStmt, AS`BlockStmt)
          then analyseBlockStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`AssignStmt)
          then analyseAssignStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`LetStmt)
          then analyseLetStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`DefStmt)
          then analyseDefStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`CallStmt)
          then analyseCallStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`ReturnStmt)
          then analyseReturnStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`IfStmt)
          then analyseIfStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`CasesStmt)
          then analyseCasesStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`SetForLoopStmt)
          then analyseSetForLoopStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`SeqForLoopStmt)
          then analyseSeqForLoopStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`WhileLoopStmt)
          then analyseWhileLoopStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`IndexForLoopStmt)
          then analyseIndexForLoopStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`TrapStmt)
          then analyseTrapStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`RecTrapStmt)
          then analyseRecTrapStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`AlwaysStmt)
          then analyseAlwaysStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`NonDetStmt)
          then analyseNonDetStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`AtomicAssignStmt)
          then analyseAtomicAssignStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`ExitStmt)
          then analyseExitStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`StartStmt)
          then analyseStartStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`StartListStmt)
          then analyseStartListStmt (wAnalysedObject, anIds, aStmt)
          elseif is_(aStmt, AS`SpecificationStmt)
          then analyseSpecificationStmt (wAnalysedObject, anIds, aStmt)
```

```
analyseLetExpr : AnalysedObject * AS`Ids * [AS`LetExpr] -> seq of AnalysedObject
analyseLetExpr (anAnalysedObject, anIds, anExpr) ==
  let localdef = anExpr.localdef,
      body = anExpr.body in
  analyseLocaldefs (anAnalysedObject, anIds, localdef)^
  analyseExpr (anAnalysedObject, anIds, body);
analyseLetBeSTExpr : AnalysedObject * AS`Ids * [AS`LetBeSTExpr] -> seq of AnalysedObject
analyseLetBeSTExpr (anAnalysedObject, anIds, anExpr) ==
  let lhs = anExpr.lhs,
      St = anExpr.St,
      In = anExpr.In in
  analyseBindList (anAnalysedObject, anIds, lhs)^
  analyseExpr (anAnalysedObject, anIds, St)^
  analyseExpr (anAnalysedObject, anIds, In);
analyseDefExpr : AnalysedObject * AS`Ids * [AS`DefExpr] -> seq of AnalysedObject
analyseDefExpr (anAnalysedObject, anIds, anExpr) ==
  let Def = anExpr.Def,
      In = anExpr.In in
  analyseDef (anAnalysedObject, anIds, Def)^
  analyseExpr (anAnalysedObject, anIds, In);
analyseLocaldefs : AnalysedObject*AS`Ids*seq of [AS`LocalDef] -> seq of AnalysedObject
analyseLocaldefs (anAnalysedObject, anIds, anLocalDef) ==
  let r = conc [if is_(anLocalDef, seq of AS`FnDef)
          then analyseFnm (anAnalysedObject, anLocalDef (i))
          elseif is_(anLocalDef, seq of AS`ValueDef)
          then analyseValueDef (anAnalysedObject, anIds, anLocalDef (i))
          else []|
             i in set inds anLocalDef] in
  r;
analyseValueDefs : AnalysedObject*IdentifierName*seq of [AS`ValueDef] -> seq of AnalysedObject
analyseValueDefs (anAnalysedObject, aClassName, aValueDef_sec) ==
  let s1 = conc [analyseValueDef (anAnalysedObject, [aClassName], aValueDef_sec (i))|i in set inds aValue
  s1;
```

```
analyseDef : AnalysedObject*AS‘Ids*seq of (AS‘PatternBind*AS‘Expr) -> seq of AnalysedObject
analyseDef (anAnalysedObject, anIds, anDef) ==
  let mk_(wPatternBind_sec, wExpr_sec) = Sequence‘Unzip[AS‘PatternBind, AS‘Expr] (anDef),
      s1 = conc [analysePatternBind (anAnalysedObject, anIds, wPatternBind_sec (i))|i in set inds wPatter
      s2 = conc [analyseExpr (anAnalysedObject, anIds, wExpr_sec (i))|i in set inds wExpr_sec],
      r = s1^s2 in
  r;
analysePatternBind : AnalysedObject * AS‘Ids * [AS‘PatternBind] -> seq of AnalysedObject
analysePatternBind (anAnalysedObject, anIds, aPatternBind) ==
  if is_(aPatternBind, AS‘Pattern)
  then analysePattern (anAnalysedObject, anIds, aPatternBind)
  elseif is_(aPatternBind, AS‘Bind)
  then analyseBind (anAnalysedObject, anIds, aPatternBind)
  else [];
analyseBind : AnalysedObject * AS‘Ids * [AS‘Bind] -> seq of AnalysedObject
analyseBind (anAnalysedObject, anIds, aBind) ==
  if is_(aBind, AS‘SetBind)
  then analyseSetBind (anAnalysedObject, anIds, aBind)
  elseif is_(aBind, AS‘TypeBind)
  then analyseTypeBind (anAnalysedObject, anIds, aBind)
  else [];
analyseSetBind : AnalysedObject * AS‘Ids * [AS‘SetBind] -> seq of AnalysedObject
analyseSetBind (anAnalysedObject, anIds, aBind) ==
  let pat = aBind.pat,
      Set = aBind.Set,
      r =
          analysePattern (anAnalysedObject, anIds, pat)^
          analyseExpr (anAnalysedObject, anIds, Set) in
  r;
analyseTypeBind : AnalysedObject * AS‘Ids * [AS‘TypeBind] -> seq of AnalysedObject
analyseTypeBind (anAnalysedObject, anIds, aBind) ==
  let pat = aBind.pat in
  analysePattern (anAnalysedObject, anIds, pat);
```

```
analysePattern : AnalysedObject * AS`Ids * [AS`Pattern] -> seq of AnalysedObject
analysePattern (anAnalysedObject, anIds, aPattern) ==
  if is_(aPattern, AS`PatternName)
  then []
  elseif is_(aPattern, AS`MatchVal)
  then analyseExpr (anAnalysedObject, anIds, aPattern.val)
  elseif is_(aPattern, AS`SetPattern)
  then analyseSetPattern (anAnalysedObject, anIds, aPattern)
  elseif is_(aPattern, AS`SeqPattern)
  then analyseSeqPattern (anAnalysedObject, anIds, aPattern)
  elseif is_(aPattern, AS`MapPattern)
  then analyseMapPattern (anAnalysedObject, anIds, aPattern)
  elseif is_(aPattern, AS`TuplePattern)
  then analyseTuplePattern (anAnalysedObject, anIds, aPattern)
  elseif is_(aPattern, AS`RecordPattern)
  then analyseRecordPattern (anAnalysedObject, anIds, aPattern)
  elseif is_(aPattern, AS`MapletPattern)
  then analyseMapletPattern (anAnalysedObject, anIds, aPattern)
  else [];
analyseSetPattern : AnalysedObject * AS`Ids * [AS`SetPattern] -> seq of AnalysedObject
analyseSetPattern (anAnalysedObject, anIds, aSetPattern) ==
  if is_(aSetPattern, AS`SetEnumPattern)
  then analyseSetEnumPattern (anAnalysedObject, anIds, aSetPattern)
  elseif is_(aSetPattern, AS`SetUnionPattern)
  then analyseSetUnionPattern (anAnalysedObject, anIds, aSetPattern)
  else [];
analyseSetEnumPattern : AnalysedObject*AS`Ids*[AS`SetEnumPattern] -> seq of AnalysedObject
analyseSetEnumPattern (anAnalysedObject, anIds, aSetEnumPattern) ==
  let Elems = aSetEnumPattern.Elems,
      r = conc [analysePattern (anAnalysedObject, anIds, Elems (i))|i in set inds Elems] in
  r;
analyseSetUnionPattern : AnalysedObject*AS`Ids*[AS`SetUnionPattern] -> seq of AnalysedObject
analyseSetUnionPattern (anAnalysedObject, anIds, aSetUnionPattern) ==
  let lp = aSetUnionPattern.lp,
      rp = aSetUnionPattern.rp,
      r =
          analysePattern (anAnalysedObject, anIds, lp)^
          analysePattern (anAnalysedObject, anIds, rp) in
  r;
```

```
analyseSeqPattern : AnalysedObject * AS`Ids * [AS`SeqPattern] -> seq of AnalysedObject
analyseSeqPattern (anAnalysedObject, anIds, aSeqPattern) ==
  if is_(aSeqPattern, AS`SeqEnumPattern)
  then analyseSeqEnumPattern (anAnalysedObject, anIds, aSeqPattern)
  elseif is_(aSeqPattern, AS`SeqConcPattern)
  then analyseSeqConcPattern (anAnalysedObject, anIds, aSeqPattern)
  else [];
analyseSeqEnumPattern : AnalysedObject*AS`Ids*[AS`SeqEnumPattern] -> seq of AnalysedObject
analyseSeqEnumPattern (anAnalysedObject, anIds, aSeqEnumPattern) ==
  let els = aSeqEnumPattern.els in
  conc [analysePattern (anAnalysedObject, anIds, els (i))|i in set inds els];
analyseSeqConcPattern : AnalysedObject*AS`Ids*[AS`SeqConcPattern] -> seq of AnalysedObject
analyseSeqConcPattern (anAnalysedObject, anIds, aSeqConcPattern) ==
  let lp = aSeqConcPattern.lp,
      rp = aSeqConcPattern.rp,
      r =
          analysePattern (anAnalysedObject, anIds, lp)^
          analysePattern (anAnalysedObject, anIds, rp) in
  r;
analyseMapPattern : AnalysedObject * AS`Ids * [AS`MapPattern] -> seq of AnalysedObject
analyseMapPattern (anAnalysedObject, anIds, aMapPattern) ==
  if is_(aMapPattern, AS`MapEnumPattern)
  then analyseMapEnumPattern (anAnalysedObject, anIds, aMapPattern)
  elseif is_(aMapPattern, AS`MapMergePattern)
  then analyseMapMergePattern (anAnalysedObject, anIds, aMapPattern)
  else [];
analyseMapEnumPattern : AnalysedObject*AS`Ids*[AS`MapEnumPattern] -> seq of AnalysedObject
analyseMapEnumPattern (anAnalysedObject, anIds, aMapEnumPattern) ==
  let mls = aMapEnumPattern.mls,
      r = conc [analyseMapletPattern (anAnalysedObject, anIds, mls (i))|i in set inds mls] in
  r;
analyseMapletPattern : AnalysedObject*AS`Ids*[AS`MapletPattern] -> seq of AnalysedObject
analyseMapletPattern (anAnalysedObject, anIds, aMapletPattern) ==
  let dp = aMapletPattern.dp,
      rp = aMapletPattern.rp,
      r =
          analysePattern (anAnalysedObject, anIds, dp)^
          analysePattern (anAnalysedObject, anIds, rp) in
  r;
```

```
analyseMapMergePattern : AnalysedObject*AS`Ids*[AS`MapMergePattern] -> seq of AnalysedObject
analyseMapMergePattern (anAnalysedObject, anIds, aMapMergePattern) ==
  let lp = aMapMergePattern.lp,
      rp = aMapMergePattern.rp,
      r =
          analysePattern (anAnalysedObject, anIds, lp)^
          analysePattern (anAnalysedObject, anIds, rp) in
  r;
analyseTuplePattern : AnalysedObject*AS`Ids*[AS`TuplePattern] -> seq of AnalysedObject
analyseTuplePattern (anAnalysedObject, anIds, aTuplePattern) ==
  let fields = aTuplePattern.fields,
      r = conc [analysePattern (anAnalysedObject, anIds, fields (i))|i in set inds fields] in
  r;
analyseRecordPattern : AnalysedObject*AS`Ids*[AS`RecordPattern] -> seq of AnalysedObject
analyseRecordPattern (anAnalysedObject, anIds, aRecordPattern) ==
  let fields = aRecordPattern.fields,
      r = conc [analysePattern (anAnalysedObject, anIds, fields (i))|i in set inds fields] in
  r;
analyseParameterTypes : AnalysedObject*AS`Ids*[AS`ParameterTypes] -> seq of AnalysedObject
analyseParameterTypes (anAnalysedObject, anIds, aParameterTypes) ==
  conc [analysePatTypePair (anAnalysedObject, anIds, aParameterTypes (i))|i in set inds aParameterTypes]
analysePatTypePair : AnalysedObject * AS`Ids * [AS`PatTypePair] -> seq of AnalysedObject
analysePatTypePair (anAnalysedObject, anIds, aPatTypePair) ==
  let pats = aPatTypePair.pats in
  conc [analysePattern (anAnalysedObject, anIds, pats (i))|i in set inds pats];
analyseValueDef : AnalysedObject * AS`Ids * [AS`ValueDef] -> seq of AnalysedObject
analyseValueDef (anAnalysedObject, anIds, anValueDef) ==
  let pat = anValueDef.pat,
      val = anValueDef.val in
  analysePattern (anAnalysedObject, anIds, pat)^analyseExpr (anAnalysedObject, anIds, val);
analyseApplyExpr : AnalysedObject * AS`Ids * [AS`ApplyExpr] -> seq of AnalysedObject
analyseApplyExpr (anAnalysedObject, anIds, anExpr) ==
  let fct = anExpr.fct,
      arg = anExpr.arg in
  analyseExpr (anAnalysedObject, anIds, fct)^
  analyseExprs (anAnalysedObject, anIds, arg);
```

```
analyseIfExpr : AnalysedObject * AS`Ids * [AS`IfExpr] -> seq of AnalysedObject
analyseIfExpr (anAnalysedObject, anIds, anExpr) ==
  let test = anExpr.test,
      cons = anExpr.cons,
      elsif = anExpr.elsif,
      altn = anExpr.altn in
  analyseExpr (anAnalysedObject, anIds, test)^
  analyseExpr (anAnalysedObject, anIds, cons)^
  analyseElseifExprs (anAnalysedObject, anIds, elsif)^
  analyseExpr (anAnalysedObject, anIds, altn);
analyseCasesExpr : AnalysedObject * AS`Ids * [AS`CasesExpr] -> seq of AnalysedObject
analyseCasesExpr (anAnalysedObject, anIds, anExpr) ==
  let sel = anExpr.sel,
      altns = anExpr.altns,
      Others = anExpr.Others,
      r1 = analyseExpr (anAnalysedObject, anIds, sel),
      r2 = conc [analyseCaseAltn (anAnalysedObject, anIds, altns (i))|i in set inds altns],
      r3 =
          if Others = nil
          then []
          else analyseExpr (anAnalysedObject, anIds, Others) in
  r1^r2^r3;
analyseCaseAltn : AnalysedObject * AS`Ids * AS`CaseAltn -> seq of AnalysedObject
analyseCaseAltn (anAnalysedObject, anIds, anCaseAltn) ==
  let match = anCaseAltn.match,
      r1 = conc [analysePattern (anAnalysedObject, anIds, match (i))|i in set inds match],
      body = anCaseAltn.body,
      r2 = analyseExpr (anAnalysedObject, anIds, body) in
  r1^r2;
```

```
analyseFieldSelectExpr : AnalysedObject*AS`Ids*[AS`FieldSelectExpr] -> seq of AnalysedObject
analyseFieldSelectExpr (anAnalysedObject, anIds, anExpr) ==
  let rec = anExpr.rec,
      nm = anExpr.nm,
      ids =
          if is_(nm, AS`Name)
          then GetName (nm)
          elseif is_(nm, AS`FctTypeInstExpr)
          then GetName (nm.polyfct)
          else GetName (nm),
      - = MakeCallTree[AS`Ids] (anAnalysedObject, ids, util.P ()) in
  analyseExpr (anAnalysedObject, anIds, rec);
analyseSetComprehensionExpr : AnalysedObject*AS`Ids*AS`SetComprehensionExpr -> seq of AnalysedObject
analyseSetComprehensionExpr (anAnalysedObject, anIds, anExpr) ==
  let elem = anExpr.elem,
      bind = anExpr.bind,
      pred = anExpr.pred,
      r1 = analyseExpr (anAnalysedObject, anIds, elem),
      r2 = analyseBindList (anAnalysedObject, anIds, bind),
      r3 =
          if pred = nil
          then []
          else analyseExpr (anAnalysedObject, anIds, pred) in
  r1^r2^r3;
analyseSeqComprehensionExpr : AnalysedObject*AS`Ids*AS`SeqComprehensionExpr -> seq of AnalysedObject
analyseSeqComprehensionExpr (anAnalysedObject, anIds, anExpr) ==
  let elem = anExpr.elem,
      bind = anExpr.bind,
      pred = anExpr.pred,
      r1 = analyseExpr (anAnalysedObject, anIds, elem),
      r2 = analyseSetBind (anAnalysedObject, anIds, bind),
      r3 =
          if pred = nil
          then []
          else analyseExpr (anAnalysedObject, anIds, pred) in
  r1^r2^r3;
```

```
analyseBracketedExpr : AnalysedObject*AS`Ids*AS`BracketedExpr -> seq of AnalysedObject
analyseBracketedExpr (anAnalysedObject, anIds, anExpr) ==
  let expr = anExpr.expr in
  analyseExpr (anAnalysedObject, anIds, expr);
analyseUnaryExpr : AnalysedObject * AS`Ids * AS`UnaryExpr -> seq of AnalysedObject
analyseUnaryExpr (anAnalysedObject, anIds, anExpr) ==
  analysePrefixExpr (anAnalysedObject, anIds, anExpr);
analysePrefixExpr : AnalysedObject * AS`Ids * AS`PrefixExpr -> seq of AnalysedObject
analysePrefixExpr (anAnalysedObject, anIds, anExpr) ==
  let arg = anExpr.arg in
  analyseExpr (anAnalysedObject, anIds, arg);
analyseBinaryExpr : AnalysedObject * AS`Ids * AS`BinaryExpr -> seq of AnalysedObject
analyseBinaryExpr (anAnalysedObject, anIds, anExpr) ==
  let left = anExpr.left,
      right = anExpr.right in
  analyseExpr (anAnalysedObject, anIds, left)^
  analyseExpr (anAnalysedObject, anIds, right);
analyseSetEnumerationExpr : AnalysedObject*AS`Ids*AS`SetEnumerationExpr -> seq of AnalysedObject
analyseSetEnumerationExpr (anAnalysedObject, anIds, anExpr) ==
  let els = anExpr.els,
      r1 = conc [analyseExpr (anAnalysedObject, anIds, els (i))|i in set inds els] in
  r1;
analyseSeqEnumerationExpr : AnalysedObject*AS`Ids*AS`SeqEnumerationExpr -> seq of AnalysedObject
analyseSeqEnumerationExpr (anAnalysedObject, anIds, anExpr) ==
  let els = anExpr.els,
      r1 = conc [analyseExpr (anAnalysedObject, anIds, els (i))|i in set inds els] in
  r1;
analyseMapEnumerationExpr : AnalysedObject*AS`Ids*AS`MapEnumerationExpr -> seq of AnalysedObject
analyseMapEnumerationExpr (anAnalysedObject, anIds, anExpr) ==
  let els = anExpr.els,
      r1 = conc [analyseMaplet (anAnalysedObject, anIds, els (i))|i in set inds els] in
  r1;
```

```
analyseMapComprehensionExpr : AnalysedObject*AS`Ids*AS`MapComprehensionExpr -> seq of AnalysedObject
analyseMapComprehensionExpr (anAnalysedObject, anIds, anExpr) ==
  let elem = anExpr.elem,
      bind = anExpr.bind,
      pred = anExpr.pred,
      r1 = analyseMaplet (anAnalysedObject, anIds, elem),
      r2 = analyseBindList (anAnalysedObject, anIds, bind),
      r3 = analyseExpr (anAnalysedObject, anIds, pred) in
  r1^r2^r3;
analyseQuantExpr : AnalysedObject * AS`Ids * AS`QuantExpr -> seq of AnalysedObject
analyseQuantExpr (anAnalysedObject, anIds, anExpr) ==
  if is_(anExpr, AS`AllOrExistsExpr)
  then analyseAllOrExistsExpr (anAnalysedObject, anIds, anExpr)
  elseif is_(anExpr, AS`ExistsUniqueExpr)
  then analyseExistsUniqueExpr (anAnalysedObject, anIds, anExpr)
  else [];
analyseFctTypeInstExpr : AnalysedObject*AS`Ids*AS`FctTypeInstExpr -> seq of AnalysedObject
analyseFctTypeInstExpr (anAnalysedObject, -, anExpr) ==
  let wName = GetName (anExpr.polyfct),
      - = MakeCallTree[AS`Ids] (anAnalysedObject, wName, util.P ()) in
  [anAnalysedObject];
analyseIotaExpr : AnalysedObject * AS`Ids * AS`IotaExpr -> seq of AnalysedObject
analyseIotaExpr (anAnalysedObject, anIds, anExpr) ==
  let bind = anExpr.bind,
      pred = anExpr.pred,
      r1 = analyseBind (anAnalysedObject, anIds, bind),
      r2 = analyseExpr (anAnalysedObject, anIds, pred) in
  r1^r2;
analyseAllOrExistsExpr : AnalysedObject*AS`Ids*AS`AllOrExistsExpr -> seq of AnalysedObject
analyseAllOrExistsExpr (anAnalysedObject, anIds, anExpr) ==
  let bind = anExpr.bind,
      pred = anExpr.pred,
      r1 = analyseBindList (anAnalysedObject, anIds, bind),
      r2 = analyseExpr (anAnalysedObject, anIds, pred) in
  r1^r2;
```

```
analyseExistsUniqueExpr : AnalysedObject*AS`Ids*AS`ExistsUniqueExpr -> seq of AnalysedObject
analyseExistsUniqueExpr (anAnalysedObject, anIds, anExpr) ==
  let bind = anExpr.bind,
      pred = anExpr.pred,
      r1 = analyseBind (anAnalysedObject, anIds, bind),
      r2 = analyseExpr (anAnalysedObject, anIds, pred) in
  r1^r2;
analyseSetRangeExpr : AnalysedObject*AS`Ids*[AS`SetRangeExpr] -> seq of AnalysedObject
analyseSetRangeExpr (anAnalysedObject, anIds, anExpr) ==
  let lb = anExpr.lb,
      ub = anExpr.ub,
      r1 = analyseExpr (anAnalysedObject, anIds, lb),
      r2 = analyseExpr (anAnalysedObject, anIds, ub) in
  r1^r2;
analyseSubSequenceExpr : AnalysedObject*AS`Ids*[AS`SubSequenceExpr] -> seq of AnalysedObject
analyseSubSequenceExpr (anAnalysedObject, anIds, anExpr) ==
  let sequence = anExpr.sequence,
      frompos = anExpr.frompos,
      topos = anExpr.topos,
      r1 = analyseExpr (anAnalysedObject, anIds, sequence),
      r2 = analyseExpr (anAnalysedObject, anIds, frompos),
      r3 = analyseExpr (anAnalysedObject, anIds, topos) in
  r1^r2^r3;
analyseSeqModifyMapOverrideExpr : AnalysedObject*AS`Ids*[AS`SeqModifyMapOverrideExpr] -> seq of Analys
analyseSeqModifyMapOverrideExpr (anAnalysedObject, anIds, anExpr) ==
  let seqmap = anExpr.seqmap,
      mapexp = anExpr.mapexp,
      r1 = analyseExpr (anAnalysedObject, anIds, seqmap),
      r2 = analyseExpr (anAnalysedObject, anIds, mapexp) in
  r1^r2;
analyseTupleConstructorExpr : AnalysedObject*AS`Ids*[AS`TupleConstructorExpr] -> seq of AnalysedObject
analyseTupleConstructorExpr (anAnalysedObject, anIds, anExpr) ==
  let fields = anExpr.fields,
      r = conc [analyseExpr (anAnalysedObject, anIds, fields (i))|i in set inds fields] in
  r;
```

```
analyseRecordConstructorExpr : AnalysedObject*AS`Ids*[AS`RecordConstructorExpr] -> seq of AnalysedObje
analyseRecordConstructorExpr (anAnalysedObject, anIds, anExpr) ==
  let fields = anExpr.fields,
      r = conc [analyseExpr (anAnalysedObject, anIds, fields (i))|i in set inds fields] in
  r;
analyseRecordModifierExpr : AnalysedObject*AS`Ids*[AS`RecordModifierExpr] -> seq of AnalysedObject
analyseRecordModifierExpr (anAnalysedObject, anIds, anExpr) ==
  let rec = anExpr.rec,
      modifiers = anExpr.modifiers,
      r1 = analyseExpr (anAnalysedObject, anIds, rec),
      r2 = conc [analyseRecordModification (anAnalysedObject, anIds, modifiers (i))|i in set inds modifie
  r1^r2;
analyseIsExpr : AnalysedObject * AS`Ids * [AS`IsExpr] -> seq of AnalysedObject
analyseIsExpr (anAnalysedObject, anIds, anExpr) ==
  let arg = anExpr.arg in
  analyseExpr (anAnalysedObject, anIds, arg);
analyseNarrowExpr : AnalysedObject * AS`Ids * [AS`NarrowExpr] -> seq of AnalysedObject
analyseNarrowExpr (anAnalysedObject, anIds, anExpr) ==
  let expr = anExpr.expr in
  analyseExpr (anAnalysedObject, anIds, expr);
analyseTupleSelectExpr : AnalysedObject*AS`Ids*[AS`TupleSelectExpr] -> seq of AnalysedObject
analyseTupleSelectExpr (anAnalysedObject, anIds, anExpr) ==
  let tuple = anExpr.tuple in
  analyseExpr (anAnalysedObject, anIds, tuple);
analyseTypeJudgementExpr : AnalysedObject*AS`Ids*[AS`TypeJudgementExpr] -> seq of AnalysedObject
analyseTypeJudgementExpr (anAnalysedObject, anIds, anExpr) ==
  let expr = anExpr.expr in
  analyseExpr (anAnalysedObject, anIds, expr);
analysePreConditionApplyExpr : AnalysedObject*AS`Ids*[AS`PreConditionApplyExpr] -> seq of AnalysedObje
analysePreConditionApplyExpr (anAnalysedObject, anIds, anExpr) ==
  let fct = anExpr.fct,
      arg = anExpr.arg,
      r1 = analyseExpr (anAnalysedObject, anIds, fct),
      r2 = conc [analyseExpr (anAnalysedObject, anIds, arg (i))|i in set inds arg] in
  r1^r2;
```

```
analyseNewExpr : AnalysedObject * AS`Ids * [AS`NewExpr] -> seq of AnalysedObject
analyseNewExpr (anAnalysedObject, anIds, anExpr) ==
  let args = anExpr.args,
      r1 = conc [analyseExpr (anAnalysedObject, anIds, args (i))|i in set inds args] in
  r1;
analyseIsOfClassExpr : AnalysedObject*AS`Ids*[AS`IsOfClassExpr] -> seq of AnalysedObject
analyseIsOfClassExpr (anAnalysedObject, anIds, anExpr) ==
  let arg = anExpr.arg in
  analyseExpr (anAnalysedObject, anIds, arg);
analyseSameBaseClassExpr : AnalysedObject*AS`Ids*[AS`SameBaseClassExpr] -> seq of AnalysedObject
analyseSameBaseClassExpr (anAnalysedObject, anIds, anExpr) ==
  let expr1 = anExpr.expr1,
      expr2 = anExpr.expr2,
      r1 = analyseExpr (anAnalysedObject, anIds, expr1),
      r2 = analyseExpr (anAnalysedObject, anIds, expr2) in
  r1^r2;
analyseSameClassExpr : AnalysedObject*AS`Ids*[AS`SameClassExpr] -> seq of AnalysedObject
analyseSameClassExpr (anAnalysedObject, anIds, anExpr) ==
  let expr1 = anExpr.expr1,
      expr2 = anExpr.expr2,
      r1 = analyseExpr (anAnalysedObject, anIds, expr1),
      r2 = analyseExpr (anAnalysedObject, anIds, expr2) in
  r1^r2;
analyseTokenConstructorExpr : AnalysedObject*AS`Ids*[AS`TokenConstructorExpr] -> seq of AnalysedObject
analyseTokenConstructorExpr (anAnalysedObject, anIds, anExpr) ==
  let field = anExpr.field in
  analyseExpr (anAnalysedObject, anIds, field);
analyseLambdaExpr : AnalysedObject * AS`Ids * [AS`LambdaExpr] -> seq of AnalysedObject
analyseLambdaExpr (anAnalysedObject, anIds, anExpr) ==
  let parm = anExpr.parm,
      body = anExpr.body,
      r1 = conc [analyseTypeBind (anAnalysedObject, anIds, parm (i))|i in set inds parm],
      r2 = analyseExpr (anAnalysedObject, anIds, body) in
  r1^r2;
```

```
analyseRecordModification : AnalysedObject*AS`Ids*[AS`RecordModification] -> seq of AnalysedObject
analyseRecordModification (anAnalysedObject, anIds, anExpr) ==
  let field = anExpr.field,
      new' = anExpr.new',
      r1 = analyseExpr (anAnalysedObject, anIds, field),
      r2 = analyseExpr (anAnalysedObject, anIds, new') in
  r1^r2;
analyseMaplet : AnalysedObject * AS`Ids * AS`Maplet -> seq of AnalysedObject
analyseMaplet (anAnalysedObject, anIds, anExpr) ==
  let mapdom = anExpr.mapdom,
      maprng = anExpr.maprng,
      r1 = analyseExpr (anAnalysedObject, anIds, mapdom),
      r2 = analyseExpr (anAnalysedObject, anIds, maprng) in
  r1^r2;
analyseBindList : AnalysedObject * AS`Ids * AS`BindList -> seq of AnalysedObject
analyseBindList (anAnalysedObject, anIds, aBindList) ==
  let r = conc [analyseMultBind (anAnalysedObject, anIds, aBindList (i))|i in set inds aBindList] in
  r;
analyseMultBind : AnalysedObject * AS`Ids * AS`MultBind -> seq of AnalysedObject
analyseMultBind (anAnalysedObject, anIds, aMultBind) ==
  if is_(aMultBind, AS`MultSetBind)
  then analyseMultSetBind (anAnalysedObject, anIds, aMultBind)
  elseif is_(aMultBind, AS`MultTypeBind)
  then analyseMultTypeBind (anAnalysedObject, anIds, aMultBind)
  else [];
analyseMultSetBind : AnalysedObject * AS`Ids * AS`MultSetBind -> seq of AnalysedObject
analyseMultSetBind (anAnalysedObject, anIds, aMultSetBind) ==
  let pat = aMultSetBind.pat,
      Set = aMultSetBind.Set,
      r1 = conc [analysePattern (anAnalysedObject, anIds, pat (i))|i in set inds pat],
      r2 = analyseExpr (anAnalysedObject, anIds, Set) in
  r1^r2;
analyseMultTypeBind : AnalysedObject * AS`Ids * AS`MultTypeBind -> seq of AnalysedObject
analyseMultTypeBind (anAnalysedObject, anIds, aMultTypeBind) ==
  let pat = aMultTypeBind.pat,
      r1 = conc [analysePattern (anAnalysedObject, anIds, pat (i))|i in set inds pat] in
  r1;
```

```
analyseBlockStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseBlockStmt (anAnalysedObject, anIds, aStmt) ==
  let stmts = aStmt.stmts in
  analyseBlockStmts (anAnalysedObject, anIds, stmts);
analyseBlockStmts : AnalysedObject * AS`Ids * seq of [AS`Stmt] -> seq of AnalysedObject
analyseBlockStmts (anAnalysedObject, anIds, stmts) ==
  let r = conc [analyseStmt (anAnalysedObject, anIds, stmts (i))|i in set inds stmts] in
  r;
analyseAssignStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseAssignStmt (anAnalysedObject, anIds, aStmt) ==
  let rhs = aStmt.rhs in
  analyseExpr (anAnalysedObject, anIds, rhs);
analyseReturnStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseReturnStmt (anAnalysedObject, anIds, aStmt) ==
  let val = aStmt.val in
  analyseExpr (anAnalysedObject, anIds, val);
analyseCallStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseCallStmt (anAnalysedObject, -, aStmt) ==
  let obj = aStmt.obj,
      oprt = aStmt.oprt,
      ids = GetName (oprt),
      args = aStmt.args,
      r1 = analyseExpr (anAnalysedObject, ids, obj),
      r2 = conc [analyseExpr (anAnalysedObject, ids, args (i))|i in set inds args],
      wAnalysedObject = anAnalysedObject.IncLevel (),
      - = MakeCallTree[AS`Ids] (wAnalysedObject, ids, util.P ()),
      - = anAnalysedObject.DecLevel () in
  r1^r2;
analyseDefStmt : AnalysedObject * AS`Ids * [AS`DefStmt] -> seq of AnalysedObject
analyseDefStmt (anAnalysedObject, anIds, aStmt) ==
  let value = aStmt.value,
      In = aStmt.In,
      r =
          analyseDef (anAnalysedObject, anIds, value)^
          analyseStmt (anAnalysedObject, anIds, In) in
  r;
```

```
analyseLetStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseLetStmt (anAnalysedObject, anIds, aStmt) ==
  let localdef = aStmt.localdef,
      In = aStmt.In,
      r1 = analyseLocaldefs (anAnalysedObject, anIds, localdef),
      r2 = analyseStmt (anAnalysedObject, anIds, In) in
  r1^r2;
analyseIfStmt : AnalysedObject * AS`Ids * [AS`Stmt] -> seq of AnalysedObject
analyseIfStmt (anAnalysedObject, anIds, aStmt) ==
  let test = aStmt.test,
      cons = aStmt.cons,
      elsif = aStmt.elsif,
      altn = aStmt.altn in
  analyseExpr (anAnalysedObject, anIds, test)^
  analyseStmt (anAnalysedObject, anIds, cons)^
  analyseElseifStmts (anAnalysedObject, anIds, elsif)^
  if altn <> nil
  then analyseStmt (anAnalysedObject, anIds, altn)
  else [];
analyseCasesStmt : AnalysedObject * AS`Ids * [AS`CasesStmt] -> seq of AnalysedObject
analyseCasesStmt (anAnalysedObject, anIds, aStmt) ==
  let sel = aStmt.sel,
      altns = aStmt.altns,
      Others = aStmt.Others,
      r1 = analyseExpr (anAnalysedObject, anIds, sel),
      r2 = conc [analyseCasesStmtAltn (anAnalysedObject, anIds, altns (i))|i in set inds altns],
      r3 =
          if Others = nil
          then []
          else analyseStmt (anAnalysedObject, anIds, Others) in
  r1^r2^r3;
analyseSetForLoopStmt : AnalysedObject*AS`Ids*[AS`SetForLoopStmt] -> seq of AnalysedObject
analyseSetForLoopStmt (anAnalysedObject, anIds, aStmt) ==
  let cv = aStmt.cv,
      fset = aStmt.fset,
      body = aStmt.body in
  analysePattern (anAnalysedObject, anIds, cv)^
  analyseExpr (anAnalysedObject, anIds, fset)^
  analyseStmt (anAnalysedObject, anIds, body);
```

```
analyseSeqForLoopStmt : AnalysedObject*AS`Ids*[AS`SeqForLoopStmt] -> seq of AnalysedObject
analyseSeqForLoopStmt (anAnalysedObject, anIds, aStmt) ==
  let cv = aStmt.cv,
      fseq = aStmt.fseq,
      body = aStmt.body in
  analysePatternBind (anAnalysedObject, anIds, cv)^
  analyseExpr (anAnalysedObject, anIds, fseq)^
  analyseStmt (anAnalysedObject, anIds, body);
analyseWhileLoopStmt : AnalysedObject*AS`Ids*[AS`WhileLoopStmt] -> seq of AnalysedObject
analyseWhileLoopStmt (anAnalysedObject, anIds, aStmt) ==
  let test = aStmt.test,
      body = aStmt.body,
      r1 = analyseExpr (anAnalysedObject, anIds, test),
      r2 = analyseStmt (anAnalysedObject, anIds, body) in
  r1^r2;
analyseIndexForLoopStmt : AnalysedObject*AS`Ids*[AS`IndexForLoopStmt] -> seq of AnalysedObject
analyseIndexForLoopStmt (anAnalysedObject, anIds, aStmt) ==
  let lb = aStmt.lb,
      ub = aStmt.ub,
      By = aStmt.By,
      body = aStmt.body,
      r1 = analyseExpr (anAnalysedObject, anIds, lb),
      r2 = analyseExpr (anAnalysedObject, anIds, ub),
      r3 = analyseExpr (anAnalysedObject, anIds, By),
      r4 = analyseStmt (anAnalysedObject, anIds, body) in
  r1^r2^r3^r4;
analyseTrapStmt : AnalysedObject * AS`Ids * [AS`TrapStmt] -> seq of AnalysedObject
analyseTrapStmt (anAnalysedObject, anIds, aStmt) ==
  let pat = aStmt.pat,
      Post = aStmt.Post,
      body = aStmt.body,
      r1 = analysePatternBind (anAnalysedObject, anIds, pat),
      r2 = analyseStmt (anAnalysedObject, anIds, Post),
      r3 = analyseStmt (anAnalysedObject, anIds, body) in
  r1^r2^r3;
```

```
analyseRecTrapStmt : AnalysedObject * AS`Ids * [AS`RecTrapStmt] -> seq of AnalysedObject
analyseRecTrapStmt (anAnalysedObject, anIds, aStmt) ==
  let traps = aStmt.traps,
      body = aStmt.body,
      r1 = conc [analyseTrap (anAnalysedObject, anIds, traps (i))|i in set inds traps],
      r2 = analyseStmt (anAnalysedObject, anIds, body) in
  r1^r2;
analyseTrap : AnalysedObject * AS`Ids * [AS`Trap] -> seq of AnalysedObject
analyseTrap (anAnalysedObject, anIds, aTrap) ==
  let match = aTrap.match,
      trappost = aTrap.trappost,
      r1 = analysePatternBind (anAnalysedObject, anIds, match),
      r2 = analyseStmt (anAnalysedObject, anIds, trappost) in
  r1^r2;
analyseAlwaysStmt : AnalysedObject * AS`Ids * [AS`AlwaysStmt] -> seq of AnalysedObject
analyseAlwaysStmt (anAnalysedObject, anIds, aStmt) ==
  let Post = aStmt.Post,
      body = aStmt.body,
      r1 = analyseStmt (anAnalysedObject, anIds, Post),
      r2 = analyseStmt (anAnalysedObject, anIds, body) in
  r1^r2;
analyseNonDetStmt : AnalysedObject * AS`Ids * [AS`NonDetStmt] -> seq of AnalysedObject
analyseNonDetStmt (anAnalysedObject, anIds, aStmt) ==
  let stmts = aStmt.stmts,
      r1 = conc [analyseStmt (anAnalysedObject, anIds, stmts (i))|i in set inds stmts] in
  r1;
analyseAtomicAssignStmt : AnalysedObject*AS`Ids*[AS`AtomicAssignStmt] -> seq of AnalysedObject
analyseAtomicAssignStmt (anAnalysedObject, anIds, aStmt) ==
  let atm = aStmt.atm,
      r1 = conc [analyseAssignStmt (anAnalysedObject, anIds, atm (i))|i in set inds atm] in
  r1;
analyseExitStmt : AnalysedObject * AS`Ids * [AS`ExitStmt] -> seq of AnalysedObject
analyseExitStmt (anAnalysedObject, anIds, aStmt) ==
  let expr = aStmt.expr in
  analyseExpr (anAnalysedObject, anIds, expr);
analyseStartStmt : AnalysedObject * AS`Ids * [AS`StartStmt] -> seq of AnalysedObject
analyseStartStmt (anAnalysedObject, anIds, aStmt) ==
  let expr = aStmt.expr in
  analyseExpr (anAnalysedObject, anIds, expr);
```

```
analyseStartListStmt : AnalysedObject*AS`Ids*[AS`StartListStmt] -> seq of AnalysedObject
analyseStartListStmt (anAnalysedObject, anIds, aStmt) ==
  let expr = aStmt.expr in
  analyseExpr (anAnalysedObject, anIds, expr);
analyseSpecificationStmt : AnalysedObject*AS`Ids*[AS`SpecificationStmt] -> seq of AnalysedObject
analyseSpecificationStmt (anAnalysedObject, anIds, aStmt) ==
  let oppre = aStmt.oppre,
      oppost = aStmt.oppost,
      excps = aStmt.excps,
      r1 = analyseExpr (anAnalysedObject, anIds, oppre),
      r2 = analyseExpr (anAnalysedObject, anIds, oppost),
      r3 = conc [analyseError (anAnalysedObject, anIds, excps (i))|i in set inds excps] in
  r1^r2^r3;
analyseError : AnalysedObject * AS`Ids * [AS`Error] -> seq of AnalysedObject
analyseError (anAnalysedObject, anIds, excps) ==
  let cond = excps.cond,
      action = excps.action,
      r1 = analyseExpr (anAnalysedObject, anIds, cond),
      r2 = analyseExpr (anAnalysedObject, anIds, action) in
  r1^r2;
analyseCasesStmtAltn : AnalysedObject*AS`Ids*AS`CasesStmtAltn -> seq of AnalysedObject
analyseCasesStmtAltn (anAnalysedObject, anIds, anCasesStmtAltn) ==
  let match = anCasesStmtAltn.match,
      r1 = conc [analysePattern (anAnalysedObject, anIds, match (i))|i in set inds match],
      body = anCasesStmtAltn.body,
      r2 = analyseStmt (anAnalysedObject, anIds, body) in
  r1^r2;
analyseElseifStmts : AnalysedObject*AS`Ids*seq of [AS`ElseifStmt] -> seq of AnalysedObject
analyseElseifStmts (anAnalysedObject, anIds, elsif) ==
  let r = conc [analyseElseifStmt (anAnalysedObject, anIds, elsif (i))|i in set inds elsif] in
  r;
analyseElseifStmt : AnalysedObject * AS`Ids * [AS`ElseifStmt] -> seq of AnalysedObject
analyseElseifStmt (anAnalysedObject, anIds, aStmt) ==
  let test = aStmt.test,
      cons = aStmt.cons,
      r1 = analyseExpr (anAnalysedObject, anIds, test),
      r2 = analyseStmt (anAnalysedObject, anIds, cons) in
  r1^r2;
```

```
  analyseElseifExprs : AnalysedObject*AS`Ids*seq of [AS`ElseifExpr] -> seq of AnalysedObject
  analyseElseifExprs (anAnalysedObject, anIds, anExpr) ==
    let r = conc [analyseElseifExpr (anAnalysedObject, anIds, anExpr (i))|i in set inds anExpr] in
    r;
  analyseElseifExpr : AnalysedObject * AS`Ids * [AS`ElseifExpr] -> seq of AnalysedObject
  analyseElseifExpr (anAnalysedObject, anIds, anExpr) ==
    let test = anExpr.test,
        cons = anExpr.cons in
    analyseExpr (anAnalysedObject, anIds, test)^analyseExpr (anAnalysedObject, anIds, cons)
end
Analyser
```

Test Suite : vdm.tc

Class : Analyser

Name	#Calls	Coverage
Analyser`analyseDef	16	√
Analyser`analyseFnm	326	90%
Analyser`analyseOpm	134	90%
Analyser`analyseBind	6	93%
Analyser`analyseExpr	9838	√
Analyser`analyseFnms	28	√
Analyser`analyseOpms	28	√
Analyser`analyseStmt	442	√
Analyser`analyseTrap	4	√
Analyser`analyseClass	28	√
Analyser`analyseError	2	√
Analyser`analyseExprs	1174	√
Analyser`analyseFnBody	324	83%
Analyser`analyseIfExpr	68	√
Analyser`analyseIfStmt	28	96%
Analyser`analyseIsExpr	24	√
Analyser`analyseMaplet	20	√
Analyser`analyseOpBody	132	√
Analyser`analyseDefExpr	4	√
Analyser`analyseDefStmt	12	√
Analyser`analyseLetExpr	234	√
Analyser`analyseLetStmt	30	√
Analyser`analyseNewExpr	32	√
Analyser`analysePattern	1218	85%

Name	#Calls	Coverage
Analyser‘analyseSetBind	86	√
Analyser‘analyseBindList	40	√
Analyser‘analyseCallStmt	18	√
Analyser‘analyseCaseAltn	16	√
Analyser‘analyseExitStmt	4	√
Analyser‘analyseIotaExpr	2	√
Analyser‘analyseMultBind	40	93%
Analyser‘analyseTrapStmt	2	√
Analyser‘analyseTypeBind	6	√
Analyser‘analyseValueDef	872	√
Analyser‘analyseApplyExpr	1174	√
Analyser‘analyseBlockStmt	86	√
Analyser‘analyseCasesExpr	6	√
Analyser‘analyseCasesStmt	2	√
Analyser‘analyseExplFnDef	322	√
Analyser‘analyseExplOpDef	130	√
Analyser‘analyseImplFnDef	2	√
Analyser‘analyseImplOpDef	2	√
Analyser‘analyseLocaldefs	264	√
Analyser‘analyseQuantExpr	22	93%
Analyser‘analyseStartStmt	6	√
Analyser‘analyseUnaryExpr	280	√
Analyser‘analyseValueDefs	28	√
Analyser‘analyseAlwaysStmt	2	√
Analyser‘analyseAssignStmt	94	√
Analyser‘analyseBinaryExpr	574	√
Analyser‘analyseBlockStmts	86	√
Analyser‘analyseElseifExpr	160	√
Analyser‘analyseElseifStmt	4	√
Analyser‘analyseLambdaExpr	2	√
Analyser‘analyseMapPattern	4	93%
Analyser‘analyseNarrowExpr	6	√
Analyser‘analyseNonDetStmt	2	√
Analyser‘analysePrefixExpr	280	√
Analyser‘analyseReturnStmt	124	√
Analyser‘analyseSeqPattern	6	93%
Analyser‘analyseSetPattern	6	93%

Name	#Calls	Coverage
Analyser‘analyseDefinitions	28	√
Analyser‘analyseElseifExprs	68	√
Analyser‘analyseElseifStmts	28	√
Analyser‘analyseLetBeSTExpr	4	√
Analyser‘analyseMultSetBind	34	√
Analyser‘analysePatTypePair	4	√
Analyser‘analysePatternBind	96	93%
Analyser‘analyseRecTrapStmt	2	√
Analyser‘analyseExtExplFnDef	2	√
Analyser‘analyseExtExplOpDef	2	√
Analyser‘analyseMultTypeBind	6	√
Analyser‘analyseSetRangeExpr	2	√
Analyser‘analyseTuplePattern	12	√
Analyser‘analyseBracketedExpr	14	√
Analyser‘analyseCasesStmtAltn	4	√
Analyser‘analyseIsOfClassExpr	2	√
Analyser‘analyseMapletPattern	4	√
Analyser‘analyseRecordPattern	2	√
Analyser‘analyseSameClassExpr	2	√
Analyser‘analyseStartListStmt	2	√
Analyser‘analyseWhileLoopStmt	8	√
Analyser‘analyseMapEnumPattern	2	√
Analyser‘analyseParameterTypes	4	√
Analyser‘analyseSeqConcPattern	2	√
Analyser‘analyseSeqEnumPattern	4	√
Analyser‘analyseSeqForLoopStmt	6	√
Analyser‘analyseSetEnumPattern	2	√
Analyser‘analyseSetForLoopStmt	4	√
Analyser‘analyseAllOrExistsExpr	20	√
Analyser‘analyseFctTypeInstExpr	38	√
Analyser‘analyseFieldSelectExpr	614	√
Analyser‘analyseMapMergePattern	2	√
Analyser‘analyseSetUnionPattern	4	√
Analyser‘analyseSubSequenceExpr	8	√
Analyser‘analyseTupleSelectExpr	4	√
Analyser‘analyseAtomicAssignStmt	2	√
Analyser‘analyseExistsUniqueExpr	2	√

Name	#Calls	Coverage
Analyser‘analyseIndexForLoopStmt	2	√
Analyser‘analyseSameBaseClassExpr	4	√
Analyser‘analyseSpecificationStmt	2	√
Analyser‘analyseTypeJudgementExpr	180	√
Analyser‘analyseMapEnumerationExpr	20	√
Analyser‘analyseRecordModification	2	√
Analyser‘analyseRecordModifierExpr	2	√
Analyser‘analyseSeqEnumerationExpr	64	√
Analyser‘analyseSetEnumerationExpr	38	√
Analyser‘analyseMapComprehensionExpr	2	√
Analyser‘analyseSeqComprehensionExpr	82	√
Analyser‘analyseSetComprehensionExpr	14	√
Analyser‘analyseTokenConstructorExpr	2	√
Analyser‘analyseTupleConstructorExpr	14	√
Analyser‘analysePreConditionApplyExpr	2	√
Analyser‘analyseRecordConstructorExpr	10	√
Analyser‘analyseSeqModifyMapOverrideExpr	6	√
Total Coverage		**97%**

3 AnalysedObject

3.1 Responsibility

Manage analysed data.

Normally, I have to manage analysed CallGraph data. However, now, I don't hold analysed data for efficiency.

```
class
AnalysedObject is subclass of AnalyserCommon
values
```

```
  unnecessaryToken = {
                     "&", ",", ".", ";", "(", ")", " - ", "@", " == >", " - >", "<- : ", "< : ", " :
>", " : ->",
                     " :: ", "^", " = ", "<", ">", "> = ", "< = ", " = >", "||", "|", "[", "]", "{", "}", "+
>", "< = >",
                     "`", " := ", "*", "**", "++", "+", " : ", "<>", "in set", "static", "is_", " ==
", ".#",
                     "narrow_",
                     "#act", "#active", "#fin", "#req", "#waiting",
                     "abs",
                     "all", "always", "and", "async",
                     "atomic", "be", "bool", "by", "card", "cases",
                     "char", "class", "comp", "compose", "conc", "dcl",
                     "def", "dinter", "div", "do", "dom", "dunion",
                     "elems", "else", "elseif", "end", "error",
                     "errs", "exists", "exists1", "exit", "ext", "false",
                     "for", "forall", "from", "functions", "hd", "if", "in",
                     "inds", "inmap", "input", "instance", "int", "inter",
                     "inv", "inverse", "iota", "is", "isofbaseclass",
                     "isofclass", "lambda", "len", "let", "map", "measure",
                     "merge", "mod", "mu", "munion", "mutex",
                     "nat", "nat1", "new", "nil", "not", "of", "operations",
                     "or", "others", "per", "periodic", "post", "power", "pre", "pre_", "mk_",
                     "private", "protected", "psubset", "public", "rat",
                     "rd", "real", "rem", "responsibility", "return",
                     "reverse", "rng", "samebaseclass", "sameclass", "self",
                     "seq", "seq1", "set", "skip", "speci\fed", "st", "start",
                     "startlist", "subclass", "subset", "sync",
                     "then", "thread", "threadid", "tixe",
                     "tl", "to", "token", "traces", "trap", "true", "types",
                     "unde\fned", "union", "values", "variables", "while", "with",
                     "wr", "yet", "RESULT"}
types
  public Level = nat;
  public OutputLevel = nat;
  public Name2LineNum = map IdentifierName to set of nat;
  public RoutineName2DefinedLineNum = map IdentifierName to set of nat;
public
```

```
  CallGraphData::fKeyName : IdentifierName
                 fDispName : IdentifierName
                 fLevel : nat
                 fDefinedLineNum : set of nat
                 fLineNum_set : set of nat
                 fSourceInfos : seq of char
instance variables
  public level : Level := 0;
  iOutputLevel : OutputLevel := 0;
  iClassName : IdentifierName := "";
  iTokenInfo_seq : [seq of CI`TokenInfo] := nil;
  iContextNodeInfo_seq : [seq of CI`ContextNodeInfo] := nil;
  public iName2LineNum : Name2LineNum := { |-> };
  public iCallGraphData_seq : seq of CallGraphData := [];
  public iRoutineName2DefinedLineNum : RoutineName2DefinedLineNum := { |-> };

operations
public
  AnalysedObject : [OutputLevel] * Level ==> AnalysedObject
  AnalysedObject (anOutputLevel, aLevel) ==
    (   level := aLevel;
        if anOutputLevel = nil
        then iOutputLevel := 0
        else iOutputLevel := anOutputLevel
    );
public
  SetName2LineNum : IdentifierName ==> AnalysedObject
  SetName2LineNum (aClassName) ==
    (   iClassName := aClassName;
        if iOutputLevel < 5
        then skip
        else for aTokenInfo in iTokenInfo_seq
             do (   if aTokenInfo.text in set unnecessaryToken
                    then skip
                    else def wName = aClassName^"`"^aTokenInfo.text;
                             wLine = aTokenInfo.pos_st.abs_line in
                         if wName not in set dom iName2LineNum
                         then iName2LineNum := iName2LineNum munion {wName |-> {wLine}}
```

```
                    else iName2LineNum := iName2LineNum ++ {wName |-> iName2LineNum(wName) union {wL
                ) ;
        return self
    ) ;
public
  ClearSetName2LineNum : () ==> AnalysedObject
  ClearSetName2LineNum() ==
    (   iName2LineNum := { |-> };
        return self
    ) ;
public
  SetName2LineNum : () ==> AnalysedObject
  SetName2LineNum() ==
    SetName2LineNum(iClassName) ;
public
  SetRoutineName2DefinedLineNum : AS`Ids * nat ==> AnalysedObject
  SetRoutineName2DefinedLineNum(anIds, aLineNum) ==
    (   let wName =
                if len anIds = 2
                then anIds(2)
                else anIds(1) in
        (   if wName not in set dom iRoutineName2DefinedLineNum
            then iRoutineName2DefinedLineNum := iRoutineName2DefinedLineNum munion {wName |-> {aLineNum}
            else iRoutineName2DefinedLineNum := iRoutineName2DefinedLineNum ++ {wName |-> iRoutineName2D
            return self
        )
    ) ;
public
  ClearRoutineName2DefinedLineNum : () ==> AnalysedObject
  ClearRoutineName2DefinedLineNum() ==
    (   iRoutineName2DefinedLineNum := { |-> };
        return self
    ) ;
public
```

```
  IsRoutineName : IdentifierName ==> bool
  IsRoutineName (aName) ==
    let wName =
            if String`Index (aName, '`') <> 0
            then let s = String`DropToken (aName, {'`'}) in
                 s (2,...,len s)
            else aName in
    return wName in set dom iRoutineName2DefinedLineNum or
           wName (1,...,4) = "pre_" or
           wName (1,...,5) = "post_";
public
  SetTokenInfo : [seq of CI`TokenInfo] ==> AnalysedObject
  SetTokenInfo (aTokenInfo_seq) ==
    (   iTokenInfo_seq := aTokenInfo_seq;
        return self
    );
public
  SetContextNodeInfo : [seq of CI`ContextNodeInfo] ==> AnalysedObject
  SetContextNodeInfo (aContextNodeInfo_seq) ==
    (   iContextNodeInfo_seq := aContextNodeInfo_seq;
        return self
    );
public
  GetLine : CI`ContextId ==> nat
  GetLine (cid) ==
    let wNodeId = CI`ConvCid2Nid (cid),
        wContextNodeInfo = iContextNodeInfo_seq (wNodeId),
        tokenpos : [CI`TokenSpan] = wContextNodeInfo.tokenpos,
        token_st = tokenpos.token_st,
        wTokenInfo = iTokenInfo_seq (token_st),
        abs_line = wTokenInfo.pos_st.abs_line in
    return abs_line;
public
  GetDefinedLineNum : IdentifierName ==> set of nat
  GetDefinedLineNum (aName) ==
    if aName not in set dom iRoutineName2DefinedLineNum
    then return {}
    else return iRoutineName2DefinedLineNum (aName) ;
public
```

```
  GetLineNum : IdentifierName ==> set of nat
  GetLineNum (aName) ==
    if aName not in set dom iName2LineNum
    then return {}
    else return iName2LineNum (aName) ;
public
  GetClassName : () ==> IdentifierName
  GetClassName () ==
    return iClassName;
public
  GetOutputLevel : () ==> OutputLevel
  GetOutputLevel () ==
    return iOutputLevel;
public
  GetLevel : () ==> Level
  GetLevel () ==
    return level;
public
  SetLevel : Level ==> ()
  SetLevel (aLevel) ==
    level := aLevel;
public
  IncLevel : () ==> AnalysedObject
  IncLevel () ==
    (   level := level + 1;
        return self
    ) ;
public
  DecLevel : () ==> AnalysedObject
  DecLevel () ==
    (   level := level-1;
        return self
    ) ;
public
  ClearLevel : () ==> AnalysedObject
  ClearLevel () ==
    (   level := 0;
        return self
    ) ;
```

```
public
  SetCallGraphData : IdentifierName * IdentifierName * nat * set of nat * set of nat *
seq of char ==> bool
  SetCallGraphData (aKeyName, aDispName, aLevel, aDefinedLineNum_set, aLineNum_set, aSourceInfos)
==
    (   let wCallGraphData = mk_CallGraphData (aKeyName, aDispName, aLevel, aDefinedLineNum_set, aLineNum_set
        iCallGraphData_seq := iCallGraphData_seq^[wCallGraphData];
        return true
    );
public
  CleanCallGraphData : () ==> seq of CallGraphData
  CleanCallGraphData () ==
    (   iCallGraphData_seq := [];
        return iCallGraphData_seq
    );
public
  GetCallGraphDataSeq : () ==> seq of CallGraphData
  GetCallGraphDataSeq () ==
    return iCallGraphData_seq
end
AnalysedObject
```

Test Suite : vdm.tc

Class : AnalysedObject

Name	#Calls	Coverage
AnalysedObject‘GetLine	488	√
AnalysedObject‘DecLevel	10758	√
AnalysedObject‘GetLevel	1325	√
AnalysedObject‘IncLevel	10758	√
AnalysedObject‘SetLevel	0	0%
AnalysedObject‘ClearLevel	56	√
AnalysedObject‘GetLineNum	1325	√
AnalysedObject‘GetClassName	5786	√
AnalysedObject‘SetTokenInfo	28	√
AnalysedObject‘IsRoutineName	5758	√
AnalysedObject‘AnalysedObject	1	81%
AnalysedObject‘GetOutputLevel	5758	√
AnalysedObject‘SetCallGraphData	1325	√
AnalysedObject‘GetDefinedLineNum	1325	√

Name	**#Calls**	**Coverage**
AnalysedObject‘CleanCallGraphData	1	√
AnalysedObject‘SetContextNodeInfo	28	√
AnalysedObject‘SetName2LineNum	0	0%
AnalysedObject‘GetCallGraphDataSeq	1	√
AnalysedObject‘ClearSetName2LineNum	0	0%
AnalysedObject‘SetRoutineName2DefinedLineNum	488	√
AnalysedObject‘ClearRoutineName2DefinedLineNum	0	0%
AnalysedObject‘SetName2LineNum	28	98%
Total Coverage		**91%**

4 bibliography

VDM++[1]

参考文献

[1] CSK システムズ. VDM++ 言語マニュアル. CSK システムズ, 第 1.1 版, 2007. Revised for VDMTools V7.1.