Promela Formalization

By René Neumann

March 17, 2025

Abstract

We present an executable formalization of the language Promela, the description language for models of the model checker SPIN. This formalization is part of the work for a completely verified model checker (CAVA), but also serves as a useful (and executable!) description of the semantics of the language itself, something that is currently missing. The formalization uses three steps: It takes an abstract syntax tree generated from an SML parser, removes syntactic sugar and enriches it with type information. This further gets translated into a transition system, on which the semantic engine (read: successor function) operates.

Contents

1 Introduction

Promela [1] is a modeling language, mainly used in the model checker SPIN [2]. It offers a C-like syntax and allows to define processes to be run concurrently. Those processes can communicate via shared global variables or by message-passing via channels. Inside a process, constructs exist for non-deterministic choice, starting other processes and enforcing atomicity. It furthermore allows different means for specifying properties: LTL formulae, assertions in the code, never claims (i. e. an automata that explicitly specifies unwanted behavior) and others.

Some constructs found in Promela models, like `#include` and `#define`, are not part of the language Promela itself, but belong to the language of the C preprocessor. SPIN does not process those, but calls the C compiler internally to process them. We do not deal with them here, but also expect the sources to be preprocessed.

Observing the output of SPIN and examining the generated graphs often is the only way of determining the semantics of a certain construct. This is complicated further by SPIN unconditionally applying optimizations. For the current formalization we chose to copy the semantics of SPIN, including the aforementioned optimizations. For some constructs, we had to restrict the semantics, i. e. some models are accepted by SPIN, but not by this formalization. Those deviations are:

- `run` is a statement instead of an expression. SPIN here has a complicated set of restrictions unto where `run` can occur inside an expression. The sole use of it is to be able to get the ID of a spawned process. We omitted this feature to guarantee expressions to be free of side-effects.

- Variable declarations which got jumped over are seen as not existing. In SPIN, such constructs show surprising behavior:
  `int i; goto L; i = 5; L: printf("%d", i)` yields 0, while
  `goto L; int i = 5; L: printf("%d", i)` yields 5.
  The latter is forbidden in our formalization (it will get rejected with "unknown variable i"), while the first behaves as in SPIN.

- Violating an `assert` does not abort, but instead sets the variable `__assert__` to true. This needs to be checked explicitly in the LTL formula. We plan on adding this check in an automatic manner.

- Types are bounded. Except for well-defined types like booleans, overflow is not allowed and will result in an error. The same holds for assigning a value that is outside the bounds. SPIN does not specify any explicit semantics here, but solely refers to the underlying C-compiler and its semantics. This might result in two models behaving differently on different systems when run with SPIN, while this formalization, due to the explicit bounds in the semantics, is not affected.

Additionally, some constructs are currently not supported, and the compilation will abort if they are encountered: `d_step`[1], `typedef`, remote references, bit-operations, `unsigned`, and property specifications except `ltl` and `assert`. Other constructs are accepted but ignored, because they do not change the behavior of a model: advanced variable scoping, `xr`, `xs`, `print*`, priorities, and visibility of variables.

Nonetheless, for models not using those unsupported constructs, we generate the very same number of states as SPIN does. An exception applies for large `goto` chains and when simultaneous termination of multiple processes is involved, as SPIN's semantics is too vague here.

2 Abstract Syntax Tree

theory *PromelaAST*
imports *Main*
begin

The abstract syntax tree is generated from the handwritten SML parser. This theory only mirrors the data structures from the SML level to make them available in Isabelle.

context
begin

local-setup ‹
  Local-Theory.map-background-naming (*Name-Space.mandatory-path AST*)
›

datatype *binOp* =
    BinOpAdd
  | *BinOpSub*
  | *BinOpMul*
  | *BinOpDiv*
  | *BinOpMod*
  | *BinOpBitAnd*
  | *BinOpBitXor*
  | *BinOpBitOr*
  | *BinOpGr*
  | *BinOpLe*
  | *BinOpGEq*
  | *BinOpLEq*
  | *BinOpEq*
  | *BinOpNEq*
  | *BinOpShiftL*
  | *BinOpShiftR*

[1] This can be safely replaced by `atomic`, though larger models will be produced then.

```
            | BinOpAnd
            | BinOpOr

datatype unOp =
              UnOpComp
            | UnOpMinus
            | UnOpNeg

datatype expr =
              ExprBinOp binOp  expr  expr
            | ExprUnOp unOp expr
            | ExprCond  expr  expr  expr
            | ExprLen varRef
            | ExprPoll varRef recvArg list
            | ExprRndPoll varRef recvArg list
            | ExprVarRef varRef
            | ExprConst integer
            | ExprTimeOut
            | ExprNP
            | ExprEnabled expr
            | ExprPC expr
            | ExprRemoteRef  String.literal
                             expr option
                             String.literal
            | ExprGetPrio expr
            | ExprSetPrio  expr  expr
            | ExprFull varRef
            | ExprEmpty varRef
            | ExprNFull varRef
            | ExprNEmpty varRef

  and varRef = VarRef  String.literal
                       expr option
                       varRef option

  and recvArg = RecvArgVar varRef
            | RecvArgEval expr
            | RecvArgConst integer

datatype range =
              RangeFromTo  varRef
                           expr
                             expr
            | RangeIn  varRef  varRef

datatype varType =
                VarTypeBit
              | VarTypeBool
              | VarTypeByte
```

```
                | VarTypePid
                | VarTypeShort
                | VarTypeInt
                | VarTypeMType
                | VarTypeChan
                | VarTypeUnsigned
                | VarTypeCustom String.literal

datatype varDecl =
                  VarDeclNum    String.literal
                                integer option
                                expr option
                | VarDeclChan   String.literal
                                integer option
                                (integer * varType list) option
                | VarDeclUnsigned  String.literal
                                   integer
                                   expr option
                | VarDeclMType   String.literal
                                 integer option
                                 String.literal option


datatype decl =
                Decl  bool option
                    varType
                    varDecl list


datatype stmnt =
                  StmntIf (step list) list
                | StmntDo (step list) list
                | StmntFor range step list
                | StmntAtomic step list
                | StmntDStep step list
                | StmntSelect range
                | StmntSeq step list
                | StmntSend varRef expr list
                | StmntSortSend varRef expr list
                | StmntRecv varRef recvArg list
                | StmntRndRecv varRef recvArg list
                | StmntRecvX varRef recvArg list
                | StmntRndRecvX varRef recvArg list
                | StmntAssign varRef expr
                | StmntIncr varRef
                | StmntDecr varRef
                | StmntElse
                | StmntBreak
                | StmntGoTo String.literal
```

```
        | StmntLabeled String.literal stmnt
        | StmntPrintF String.literal expr list
        | StmntPrintM String.literal
        | StmntRun  String.literal
                    expr list
                    integer option
        | StmntAssert expr
        | StmntCond expr
        | StmntCall String.literal varRef list

 and step = StepStmnt stmnt  stmnt option
          | StepDecl decl
          | StepXR varRef list
          | StepXS varRef list

datatype module =
                ProcType  (integer option) option
                          String.literal
                         decl list
                          integer option
                          expr option
                           step list
              | DProcType  (integer option) option
                           String.literal
                          decl list
                           integer option
                           expr option
                            step list
              | Init  integer option step list
              | Never step list
              | Trace step list
              | NoTrace step list
              | Inline String.literal String.literal list step list
              | TypeDef String.literal decl list
              | MType String.literal list
              | ModuDecl decl
              | Ltl  String.literal  String.literal

end
end
```

3 Data structures as used in Promela

```
theory PromelaDatastructures
imports
  CAVA-Base.CAVA-Base
  CAVA-Base.Lexord-List
  PromelaAST
  HOL−Library.IArray
```

Deriving.Compare-Instances
CAVA-Base.CAVA-Code-Target
begin

3.1 Abstract Syntax Tree *after* preprocessing

From the plain AST stemming from the parser, we'd like to have one containing more information while also removing duplicated constructs. This is achieved in the preprocessing step.

The additional information contains:

- variable type (including whether it represents a channel or not)
- global vs local variable

Also certain constructs are expanded (like for-loops) or different nodes in the AST are collapsed into one parametrized node (e.g. the different send-operations).

This preprocessing phase also tries to detect certain static errors and will bail out with an exception if such is encountered.

datatype *binOp* = *BinOpAdd*
| *BinOpSub*
| *BinOpMul*
| *BinOpDiv*
| *BinOpMod*
| *BinOpGr*
| *BinOpLe*
| *BinOpGEq*
| *BinOpLEq*
| *BinOpEq*
| *BinOpNEq*
| *BinOpAnd*
| *BinOpOr*

datatype *unOp* = *UnOpMinus*
| *UnOpNeg*

datatype *expr* = *ExprBinOp binOp expr expr*
| *ExprUnOp unOp expr*
| *ExprCond expr expr expr*
| *ExprLen chanRef*
| *ExprVarRef varRef*
| *ExprConst integer*
| *ExprMConst integer String.literal*
| *ExprTimeOut*
| *ExprFull chanRef*
| *ExprEmpty chanRef*
| *ExprPoll chanRef recvArg list bool*

```
  and varRef = VarRef  bool
                       String.literal
                       expr option
  and chanRef = ChanRef varRef — explicit type for channels
  and recvArg = RecvArgVar varRef
              | RecvArgEval expr
              | RecvArgConst integer
              | RecvArgMConst integer String.literal

datatype varType = VTBounded integer integer
                 | VTChan
```

Variable declarations at the beginning of a proctype or at global level.

```
datatype varDecl = VarDeclNum  integer integer
                               String.literal
                               integer option
                               expr option
                 | VarDeclChan  String.literal
                                integer option
                                (integer * varType list) option
```

Variable declarations during a proctype.

```
datatype procVarDecl = ProcVarDeclNum   integer integer
                                        String.literal
                                        integer option
                                        expr option
                     | ProcVarDeclChan  String.literal
                                        integer option

datatype procArg = ProcArg varType String.literal

datatype stmnt = StmntIf (step list) list
               | StmntDo (step list) list
               | StmntAtomic step list
               | StmntSeq step list
               | StmntSend chanRef expr list bool
               | StmntRecv chanRef recvArg list bool  bool
               | StmntAssign varRef expr
               | StmntElse
               | StmntBreak
               | StmntSkip
               | StmntGoTo String.literal
               | StmntLabeled String.literal stmnt
               | StmntRun  String.literal
                           expr list
               | StmntCond expr
               | StmntAssert expr
```

```
  and step = StepStmnt stmnt  stmnt option
         | StepDecl procVarDecl list
         | StepSkip

datatype proc = ProcType  (integer option) option
                          String.literal
                         procArg list
                        varDecl list
                          step list
              | Init varDecl list step list

type-synonym ltl = — name: String.literal × — formula: String.literal
type-synonym promela = varDecl list × proc list × ltl list
```

3.2 Preprocess the AST of the parser into our variant

We setup some functionality for printing warning or even errors.

All those constants are logically *undefined*, but replaced by the parser for something meaningful.

```
consts
  warn :: String.literal ⇒ unit

abbreviation with-warn msg e ≡ let - = warn msg in e
abbreviation the-warn opt msg ≡ case opt of None ⇒ () | - ⇒ warn msg
```

usc: "Unsupported Construct"

```
definition [code del]: usc (c :: String.literal) ≡ undefined

definition  [code del]: err (e :: String.literal) = undefined
abbreviation errv e v ≡ err (e + v)

definition [simp, code del]: abort (msg :: String.literal) f = f ()
abbreviation abortv msg v f ≡ abort (msg + v) f

code-printing
  code-module PromelaUtils ⇀ (SML) ‹
    structure PromelaUtils = struct
      exception UnsupportedConstruct of string
      exception StaticError of string
      exception RuntimeError of string
      fun warn msg = TextIO.print (Warning:  ^ msg ^ \n)
      fun usc  c   = raise (UnsupportedConstruct c)
      fun err  e   = raise (StaticError e)
      fun abort msg - = raise (RuntimeError msg)
    end›
| constant warn ⇀ (SML) PromelaUtils.warn
| constant usc ⇀ (SML) PromelaUtils.usc
| constant err ⇀ (SML) PromelaUtils.err
```

```
| constant abort ⇀ (SML) PromelaUtils.abort
code-reserved (SML) PromelaUtils
```

The preprocessing is done for each type on its own.

```
primrec ppBinOp :: AST.binOp ⇒ binOp
where
  ppBinOp AST.BinOpAdd = BinOpAdd
| ppBinOp AST.BinOpSub = BinOpSub
| ppBinOp AST.BinOpMul = BinOpMul
| ppBinOp AST.BinOpDiv = BinOpDiv
| ppBinOp AST.BinOpMod = BinOpMod
| ppBinOp AST.BinOpGr = BinOpGr
| ppBinOp AST.BinOpLe = BinOpLe
| ppBinOp AST.BinOpGEq = BinOpGEq
| ppBinOp AST.BinOpLEq = BinOpLEq
| ppBinOp AST.BinOpEq = BinOpEq
| ppBinOp AST.BinOpNEq = BinOpNEq
| ppBinOp AST.BinOpAnd = BinOpAnd
| ppBinOp AST.BinOpOr = BinOpOr
| ppBinOp AST.BinOpBitAnd = usc STR ''BinOpBitAnd''
| ppBinOp AST.BinOpBitXor = usc STR ''BinOpBitXor''
| ppBinOp AST.BinOpBitOr = usc STR ''BinOpBitOr''
| ppBinOp AST.BinOpShiftL = usc STR ''BinOpShiftL''
| ppBinOp AST.BinOpShiftR = usc STR ''BinOpShiftR''

primrec ppUnOp :: AST.unOp ⇒ unOp
where
  ppUnOp AST.UnOpMinus = UnOpMinus
| ppUnOp AST.UnOpNeg = UnOpNeg
| ppUnOp AST.UnOpComp = usc STR ''UnOpComp''
```

The data structure holding all information on variables we found so far.

```
type-synonym var-data =
    (String.literal, (integer option × bool)) lm — channels
    × (String.literal, (integer option × bool)) lm — variables
    × (String.literal, integer) lm — mtypes
    × (String.literal, varRef) lm — aliases (used for inlines)

definition dealWithVar
  :: var-data ⇒ String.literal
      ⇒ (String.literal ⇒ integer option × bool ⇒ expr option ⇒ 'a)
      ⇒ (String.literal ⇒ integer option × bool ⇒ expr option ⇒ 'a)
      ⇒ (integer ⇒ 'a) ⇒ 'a
where
  dealWithVar cvm n fC fV fM ≡ (
    let (c,v,m,a) = cvm in
    let (n, idx) = case lm.lookup n a of
```

```
                None ⇒ (n, None)
              | Some (VarRef - name idx) ⇒ (name, idx)
  in
  case lm.lookup n m of
    Some i ⇒ (case idx of None ⇒ fM i
                        | - ⇒ err STR ''Array subscript used on MType (via alias).'')
  | None ⇒ (case lm.lookup n v of
              Some g ⇒ fV n g idx
            | None ⇒ (case lm.lookup n c of
                        Some g ⇒ fC n g idx
                      | None ⇒ err (STR ''Unknown variable referenced: '' + n))))

primrec enforceChan :: varRef + chanRef ⇒ chanRef where
  enforceChan (Inl -) = err STR ''Channel expected. Got normal variable.''
| enforceChan (Inr c) = c

fun liftChan :: varRef + chanRef ⇒ varRef where
  liftChan (Inl v) = v
| liftChan (Inr (ChanRef v)) = v

fun resolveIdx :: expr option ⇒ expr option ⇒ expr option
where
  resolveIdx None None = None
| resolveIdx idx  None = idx
| resolveIdx None aliasIdx = aliasIdx
| resolveIdx -    -      = err STR ''Array subscript used twice (via alias).''

fun ppExpr :: var-data ⇒ AST.expr ⇒ expr
and ppVarRef :: var-data ⇒ AST.varRef ⇒ varRef + chanRef
and ppRecvArg :: var-data ⇒ AST.recvArg ⇒ recvArg
where
  ppVarRef cvm (AST.VarRef name idx None) = dealWithVar cvm name
                  (λname (-,g) aIdx. let idx = map-option (ppExpr cvm) idx in
                      Inr (ChanRef (VarRef g name (resolveIdx idx aIdx))))
                  (λname (-,g) aIdx. let idx = map-option (ppExpr cvm) idx in
                      Inl (VarRef g name (resolveIdx idx aIdx)))
                  (λ-. err STR ''Variable expected. Got MType.'')
| ppVarRef cvm (AST.VarRef - - (Some -)) =
    usc STR ''next operation on variables''

| ppExpr cvm AST.ExprTimeOut = ExprTimeOut
| ppExpr cvm (AST.ExprConst c) = ExprConst c

| ppExpr cvm (AST.ExprBinOp bo l r) =
    ExprBinOp (ppBinOp bo) (ppExpr cvm l) (ppExpr cvm r)
| ppExpr cvm (AST.ExprUnOp uo e) =
    ExprUnOp (ppUnOp uo) (ppExpr cvm e)
| ppExpr cvm (AST.ExprCond c t f) =
    ExprCond (ppExpr cvm c) (ppExpr cvm t) (ppExpr cvm f)
```

```
| ppExpr cvm (AST.ExprLen v) =
    ExprLen (enforceChan (ppVarRef cvm v))
| ppExpr cvm (AST.ExprFull v) =
    ExprFull (enforceChan (ppVarRef cvm v))
| ppExpr cvm (AST.ExprEmpty v) =
    ExprEmpty (enforceChan (ppVarRef cvm v))

| ppExpr cvm (AST.ExprNFull v) =
    ExprUnOp UnOpNeg (ExprFull (enforceChan (ppVarRef cvm v)))
| ppExpr cvm (AST.ExprNEmpty v) =
    ExprUnOp UnOpNeg (ExprEmpty (enforceChan (ppVarRef cvm v)))

| ppExpr cvm (AST.ExprVarRef v) = (
        let to-exp = λ-. ExprVarRef (liftChan (ppVarRef cvm v)) in
        case v of
            AST.VarRef name None None ⇒
              dealWithVar cvm name
                  (λ- - -. to-exp())
                  (λ- - -. to-exp())
                  (λi. ExprMConst i name)
          | - ⇒ to-exp())

| ppExpr cvm (AST.ExprPoll v es) =
    ExprPoll (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm) es) False
| ppExpr cvm (AST.ExprRndPoll v es) =
    ExprPoll (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm) es) True

| ppExpr cvm AST.ExprNP = usc STR ''ExprNP''
| ppExpr cvm (AST.ExprEnabled -) = usc STR ''ExprEnabled''
| ppExpr cvm (AST.ExprPC -) = usc STR ''ExprPC''
| ppExpr cvm (AST.ExprRemoteRef - - -) = usc STR ''ExprRemoteRef''
| ppExpr cvm (AST.ExprGetPrio -) = usc STR ''ExprGetPrio''
| ppExpr cvm (AST.ExprSetPrio - -) = usc STR ''ExprSetPrio''

| ppRecvArg cvm (AST.RecvArgVar v) = (
        let to-ra = λ-. RecvArgVar (liftChan (ppVarRef cvm v)) in
        case v of
            AST.VarRef name None None ⇒
              dealWithVar cvm name
                  (λ- - -. to-ra())
                  (λ- - -. to-ra())
                  (λi. RecvArgMConst i name)
          | - ⇒ to-ra())
| ppRecvArg cvm (AST.RecvArgEval e) = RecvArgEval (ppExpr cvm e)
| ppRecvArg cvm (AST.RecvArgConst c) = RecvArgConst c

primrec ppVarType :: AST.varType ⇒ varType where
  ppVarType AST.VarTypeBit = VTBounded 0 1
```

```
| ppVarType AST.VarTypeBool = VTBounded 0 1
| ppVarType AST.VarTypeByte = VTBounded 0 255
| ppVarType AST.VarTypePid = VTBounded 0 255
| ppVarType AST.VarTypeShort = VTBounded (−(2^15)) ((2^15) − 1)
| ppVarType AST.VarTypeInt = VTBounded (−(2^31)) ((2^31) − 1)
| ppVarType AST.VarTypeMType = VTBounded 1 255
| ppVarType AST.VarTypeChan = VTChan
| ppVarType AST.VarTypeUnsigned = usc STR "VarTypeUnsigned"
| ppVarType (AST.VarTypeCustom -) = usc STR "VarTypeCustom"

fun ppVarDecl
  :: var-data ⇒ varType ⇒ bool ⇒ AST.varDecl ⇒ var-data × varDecl
where
  ppVarDecl (c,v,m,a) (VTBounded l h) g
                    (AST.VarDeclNum name sze init) = (
      case lm.lookup name v of
        Some - ⇒ errv STR "Duplicate variable " name
      | - ⇒ (case lm.lookup name a of
              Some - ⇒ errv
                         STR "Variable name clashes with alias: " name
            | - ⇒ ((c, lm.update name (sze,g) v, m, a),
                     VarDeclNum l h name sze
                       (map-option (ppExpr (c,v,m,a)) init))))
| ppVarDecl - - g (AST.VarDeclNum name sze init) =
    err STR "Assiging num to non−num"

| ppVarDecl (c,v,m,a) VTChan g
                    (AST.VarDeclChan name sze cap) = (
    let cap′ = map-option (apsnd (map ppVarType)) cap in
    case lm.lookup name c of
      Some - ⇒ errv STR "Duplicate variable " name
    | - ⇒ (case lm.lookup name a of
            Some - ⇒ errv
                       STR "Variable name clashes with alias: " name
          | - ⇒ ((lm.update name (sze, g) c, v, m, a),
                   VarDeclChan name sze cap′)))
| ppVarDecl - - g (AST.VarDeclChan name sze init) =
    err STR "Assiging chan to non−chan"

| ppVarDecl (c,v,m,a) (VTBounded l h) g
                    (AST.VarDeclMType name sze init) = (
    let init = map-option (λmty.
    case lm.lookup mty m of
      None ⇒ errv STR "Unknown MType " mty
    | Some mval ⇒ ExprMConst mval mty) init in
        case lm.lookup name c of
          Some - ⇒ errv STR "Duplicate variable " name
        | - ⇒ (case lm.lookup name a of Some -
                  ⇒ errv STR "Variable name clashes with alias: " name
```

```
        | - ⇒ ((c, lm.update name (sze,g) v, m, a),
              VarDeclNum l h name sze init)))

| ppVarDecl - - g (AST.VarDeclMType name sze init) =
    err STR ''Assiging num to non−num''

| ppVarDecl - - - (AST.VarDeclUnsigned - - -) =
    usc STR ''VarDeclUnsigned''

definition ppProcVarDecl
  :: var-data ⇒ varType ⇒ bool ⇒ AST.varDecl ⇒ var-data × procVarDecl
where
  ppProcVarDecl cvm ty g v = (case ppVarDecl cvm ty g v of
      (cvm, VarDeclNum l h name sze init) ⇒ (cvm, ProcVarDeclNum l h name
sze init)
    | (cvm, VarDeclChan name sze None) ⇒ (cvm, ProcVarDeclChan name sze)
    | - ⇒ err STR ''Channel initilizations only allowed at the beginning of proc-
types.'')

fun ppProcArg
  :: var-data ⇒ varType ⇒ bool ⇒ AST.varDecl ⇒ var-data × procArg
where
  ppProcArg (c,v,m,a) (VTBounded l h) g
                    (AST.VarDeclNum name None None) = (
    case lm.lookup name v of
      Some - ⇒ errv STR ''Duplicate variable '' name
    | - ⇒ (case lm.lookup name a of
            Some - ⇒ errv
                       STR ''Variable name clashes with alias: '' name
          | - ⇒ ((c, lm.update name (None, g) v, m, a),
                ProcArg (VTBounded l h) name)))
| ppProcArg - - - (AST.VarDeclNum - - -) =
    err STR ''Invalid proctype arguments''

| ppProcArg (c,v,m,a) VTChan g
                    (AST.VarDeclChan name None None) = (
    case lm.lookup name c of
      Some - ⇒ errv STR ''Duplicate variable '' name
    | - ⇒ (case lm.lookup name a of
            Some - ⇒ errv
                       STR ''Variable name clashes with alias: '' name
        | - ⇒ ((lm.update name (None, g) c, v, m, a), ProcArg VTChan name)))
| ppProcArg - - - (AST.VarDeclChan - - -) =
    err STR ''Invalid proctype arguments''

| ppProcArg (c,v,m,a) (VTBounded l h) g
                    (AST.VarDeclMType name None None) = (
    case lm.lookup name v of
      Some - ⇒ errv STR ''Duplicate variable '' name
```

```
    | - ⇒ (case lm.lookup name a of
          Some - ⇒ errv
                    STR ''Variable name clashes with alias: '' name
        | - ⇒ ((c, lm.update name (None, g) v, m, a),
              ProcArg (VTBounded l h) name)))
| ppProcArg - - - (AST.VarDeclMType - - -) =
    err STR ''Invalid proctype arguments''

| ppProcArg - - - (AST.VarDeclUnsigned - - -) = usc STR ''VarDeclUnsigned''
```

Some preprocessing functions enrich the *var-data* argument and hence return a new updated one. When chaining multiple calls to such functions after another, we need to make sure, the *var-data* is passed accordingly. *cvm-fold* does exactly that for such a function g and a list of nodes *ss*.

```
definition cvm-fold where
  cvm-fold g cvm ss = foldl (λ(cvm,ss) s. apsnd (λs'. ss@[s']) (g cvm s))
                            (cvm, []) ss

lemma cvm-fold-cong[fundef-cong]:
  assumes cvm = cvm'
  and stepss = stepss'
  and ⋀x d. x ∈ set stepss ⟹ g d x = g' d x
  shows cvm-fold g cvm stepss = cvm-fold g' cvm' stepss'
unfolding cvm-fold-def
using assms
by (fastforce intro: foldl-cong split: prod.split)

fun liftDecl where
  liftDecl f g cvm (AST.Decl vis t decls) = (
    let - = the-warn vis STR ''Visibility in declarations not supported. Ignored.'' in
    let t = ppVarType t in
    cvm-fold (λcvm. f cvm t g) cvm decls)

definition ppDecl
  :: bool ⇒ var-data ⇒ AST.decl ⇒ var-data × varDecl list
where
  ppDecl = liftDecl ppVarDecl

definition ppDeclProc
  :: var-data ⇒ AST.decl ⇒ var-data × procVarDecl list
where
  ppDeclProc = liftDecl ppProcVarDecl False

definition ppDeclProcArg
  :: var-data ⇒ AST.decl ⇒ var-data × procArg list
where
  ppDeclProcArg = liftDecl ppProcArg False
```

definition *incr* :: *varRef* ⇒ *stmnt* **where**
incr v = *StmntAssign v* (*ExprBinOp BinOpAdd* (*ExprVarRef v*) (*ExprConst 1*))

definition *decr* :: *varRef* ⇒ *stmnt* **where**
decr v = *StmntAssign v* (*ExprBinOp BinOpSub* (*ExprVarRef v*) (*ExprConst 1*))

Transforms `for (i : lb .. ub) steps` into

```
{
  i = lb;
  do
    :: i =< ub -> steps; i++
    :: else -> break
  od
}
```

definition *forFromTo* :: *varRef* ⇒ *expr* ⇒ *expr* ⇒ *step list* ⇒ *stmnt* **where**
forFromTo i lb ub steps = (
  let
    — *i* = *lb*
    loop-pre = *StepStmnt* (*StmntAssign i lb*) *None*;
    — *i* ≤ *ub*
    loop-cond = *StepStmnt* (*StmntCond*
                  (*ExprBinOp BinOpLEq* (*ExprVarRef i*) *ub*))
                  None;
    — *i*++
    loop-incr = *StepStmnt* (*incr i*) *None*;
    — *i* ≤ *ub* −> ...; *i*++
    loop-body = *loop-cond* # *steps* @ [*loop-incr*];
    — *else* −> *break*
    loop-abort = [*StepStmnt StmntElse None*, *StepStmnt StmntBreak None*];
    — *do* :: *i* ≤ *ub* −> ... :: *else* −> *break od*
    loop = *StepStmnt* (*StmntDo* [*loop-body*, *loop-abort*]) *None*
  in
    StmntSeq [*loop-pre*, *loop*])

Transforms (where a is an array with N entries) `for (i in a) steps` into

```
{
  i = 0;
  do
    :: i < N -> steps; i++
    :: else -> break
  od
}
```

definition *forInArray* :: *varRef* ⇒ *integer* ⇒ *step list* ⇒ *stmnt* **where**

```
  forInArray i N steps = (
      let
        — i = 0
        loop-pre = StepStmnt (StmntAssign i (ExprConst 0)) None;
        — i < N
        loop-cond = StepStmnt (StmntCond
                               (ExprBinOp BinOpLe (ExprVarRef i)
                                 (ExprConst N)))
                               None;
        — i++
        loop-incr = StepStmnt (incr i) None;
        — i < N -> ...; i++
        loop-body = loop-cond # steps @ [loop-incr];
        — else -> break
        loop-abort = [StepStmnt StmntElse None, StepStmnt StmntBreak None];
        — do :: i < N -> ... :: else -> break od
        loop = StepStmnt (StmntDo [loop-body, loop-abort]) None
      in
        StmntSeq [loop-pre, loop])
```

Transforms (where c is a channel) `for (msg in c) steps` into

```
{
   byte :tmp: = 0;
   do
     :: :tmp: < len(c) ->
          c?msg; c!msg;
          steps;
          :tmp:++
     :: else -> break
   od
}
```

```
definition forInChan :: varRef ⇒ chanRef ⇒ step list ⇒ stmnt where
  forInChan msg c steps = (
      let
        — byte :tmp: = 0
        tmpStr = STR '':tmp:'';
        loop-pre = StepDecl
                   [ProcVarDeclNum 0 255 tmpStr None (Some (ExprConst 0))];
        tmp = VarRef False tmpStr None;
        — :tmp: < len(c)
        loop-cond = StepStmnt (StmntCond
                               (ExprBinOp BinOpLe (ExprVarRef tmp)
                                 (ExprLen c)))
                              None;
        — :tmp:++
        loop-incr = StepStmnt (incr tmp) None;
```

```
      — c?msg
      recv = StepStmnt (StmntRecv c [RecvArgVar msg] False True) None;
      — c!msg
      send = StepStmnt (StmntSend c [ExprVarRef msg] False) None;
      — :tmp: < len(c) -> c?msg; c!msg; ...; :tmp:++
      loop-body = [loop-cond, recv, send] @ steps @ [loop-incr];
      — else -> break
      loop-abort = [StepStmnt StmntElse None, StepStmnt StmntBreak None];
      — do :: :tmp: < len(c) -> ... :: else -> break od
      loop = StepStmnt (StmntDo [loop-body, loop-abort]) None
    in
      StmntSeq [loop-pre, loop])
```

Transforms `select (i : lb .. ub)` into

```
{
   i = lb;
   do
     :: i < ub -> i++
     :: break
   od
}
```

```
definition select :: varRef ⇒ expr ⇒ expr ⇒ stmnt where
  select i lb ub = (
    let
      — i = lb
      pre = StepStmnt (StmntAssign i lb) None;
      — i < ub
      cond = StepStmnt (StmntCond (ExprBinOp BinOpLe (ExprVarRef i) ub))
                       None;
      — i++
      incr = StepStmnt (incr i) None;
      — i < ub -> i++
      loop-body = [cond, incr];
      — break
      loop-abort = [StepStmnt StmntBreak None];
      — do :: i < ub -> ... :: break od
      loop = StepStmnt (StmntDo [loop-body, loop-abort]) None
    in
      StmntSeq [pre, loop])

type-synonym inlines =
  (String.literal, String.literal list × (var-data ⇒ var-data × step list)) lm
type-synonym stmnt-data =
   bool × varDecl list × inlines × var-data

fun ppStep :: stmnt-data ⇒ AST.step ⇒ stmnt-data ∗ step
and ppStmnt :: stmnt-data ⇒ AST.stmnt ⇒ stmnt-data ∗ stmnt
```

where

```
  ppStep cvm (AST.StepStmnt s u) = (
    let (cvm', s') = ppStmnt cvm s in
    case u of None ⇒ (cvm', StepStmnt s' None)
            | Some u ⇒ let (cvm'',u') = ppStmnt cvm' u in
                        (cvm'', StepStmnt s' (Some u')))
| ppStep (False, ps, i, cvm) (AST.StepDecl d) =
    map-prod (λcvm. (False, ps, i, cvm)) StepDecl (ppDeclProc cvm d)
| ppStep (True, ps, i, cvm) (AST.StepDecl d) = (
          let (cvm', ps') = ppDecl False cvm d
          in ((True, ps@ps', i, cvm'), StepSkip))
| ppStep (-,cvm) (AST.StepXR -) =
    with-warn STR ''StepXR not supported. Ignored.'' ((False,cvm), StepSkip)
| ppStep (-,cvm) (AST.StepXS -) =
    with-warn STR ''StepXS not supported. Ignored.'' ((False,cvm), StepSkip)

| ppStmnt (-,cvm) (AST.StmntBreak) = ((False,cvm), StmntBreak)
| ppStmnt (-,cvm) (AST.StmntElse) = ((False,cvm), StmntElse)
| ppStmnt (-,cvm) (AST.StmntGoTo l) = ((False,cvm), StmntGoTo l)
| ppStmnt (-,cvm) (AST.StmntLabeled l s) =
    apsnd (StmntLabeled l) (ppStmnt (False,cvm) s)
| ppStmnt (-,ps,i,cvm) (AST.StmntCond e) =
    ((False,ps,i,cvm), StmntCond (ppExpr cvm e))
| ppStmnt (-,ps,i,cvm) (AST.StmntAssert e) =
    ((False,ps,i,cvm), StmntAssert (ppExpr cvm e))
| ppStmnt (-,ps,i,cvm) (AST.StmntAssign v e) =
    ((False,ps,i,cvm), StmntAssign (liftChan (ppVarRef cvm v)) (ppExpr cvm e))
| ppStmnt (-,ps,i,cvm) (AST.StmntSend v es) =
    ((False,ps,i,cvm), StmntSend (enforceChan (ppVarRef cvm v))
                                 (map (ppExpr cvm) es) False)
| ppStmnt (-,ps,i,cvm) (AST.StmntSortSend v es) =
    ((False,ps,i,cvm), StmntSend (enforceChan (ppVarRef cvm v))
                                 (map (ppExpr cvm) es) True)
| ppStmnt (-,ps,i,cvm) (AST.StmntRecv v rs) =
    ((False,ps,i,cvm), StmntRecv (enforceChan (ppVarRef cvm v))
                                 (map (ppRecvArg cvm) rs) False True)
| ppStmnt (-,ps,i,cvm) (AST.StmntRecvX v rs) =
    ((False,ps,i,cvm), StmntRecv (enforceChan (ppVarRef cvm v))
                                 (map (ppRecvArg cvm) rs) False False)
| ppStmnt (-,ps,i,cvm) (AST.StmntRndRecv v rs) =
    ((False,ps,i,cvm), StmntRecv (enforceChan (ppVarRef cvm v))
                                 (map (ppRecvArg cvm) rs) True True)
| ppStmnt (-,ps,i,cvm) (AST.StmntRndRecvX v rs) =
    ((False,ps,i,cvm), StmntRecv (enforceChan (ppVarRef cvm v))
                                 (map (ppRecvArg cvm) rs) True False)
| ppStmnt (-,ps,i,cvm) (AST.StmntRun n es p) = (
    let - = the-warn p STR ''Priorities for 'run' not supported. Ignored.'' in
   ((False,ps,i,cvm), StmntRun n (map (ppExpr cvm) es)))
| ppStmnt (-,cvm) (AST.StmntSeq ss) =
```

```
    apsnd StmntSeq (cvm-fold ppStep (False,cvm) ss)
| ppStmnt (-,cvm) (AST.StmntAtomic ss) =
    apsnd StmntAtomic (cvm-fold ppStep (False,cvm) ss)
| ppStmnt (-,cvm) (AST.StmntIf sss) =
    apsnd StmntIf (cvm-fold (cvm-fold ppStep) (False,cvm) sss)
| ppStmnt (-,cvm) (AST.StmntDo sss) =
    apsnd StmntDo (cvm-fold (cvm-fold ppStep) (False,cvm) sss)


| ppStmnt (-,ps,i,cvm) (AST.StmntIncr v) =
    ((False,ps,i,cvm), incr (liftChan (ppVarRef cvm v)))
| ppStmnt (-,ps,i,cvm) (AST.StmntDecr v) =
    ((False,ps,i,cvm), decr (liftChan (ppVarRef cvm v)))

| ppStmnt (-,cvm) (AST.StmntPrintF - -) =
    with-warn STR "PrintF ignored" ((False,cvm), StmntSkip)
| ppStmnt (-,cvm) (AST.StmntPrintM -) =
    with-warn STR "PrintM ignored" ((False,cvm), StmntSkip)

| ppStmnt (-,ps,inl,cvm) (AST.StmntFor
                           (AST.RangeFromTo i lb ub)
                           steps) = (
    let
      i = liftChan (ppVarRef cvm i);
      (lb,ub) = (ppExpr cvm lb, ppExpr cvm ub)
    in
      apsnd (forFromTo i lb ub) (cvm-fold ppStep (False,ps,inl,cvm) steps))
| ppStmnt (-,ps,inl,cvm) (AST.StmntFor
                           (AST.RangeIn i v)
                           steps) = (
    let
       i = liftChan (ppVarRef cvm i);
       (cvm',steps) = cvm-fold ppStep (False,ps,inl,cvm) steps
    in
       case ppVarRef cvm v of
        Inr c ⇒ (cvm', forInChan i c steps)
      | Inl (VarRef - - (Some -)) ⇒ err STR "Iterating over array-member."
      | Inl (VarRef - name None) ⇒ (
            let (-,v,-) = cvm in
            case fst (the (lm.lookup name v)) of
             None ⇒ err STR "Iterating over non-array variable."
           | Some N ⇒ (cvm', forInArray i N steps)))

| ppStmnt (-,ps,inl,cvm) (AST.StmntSelect
                           (AST.RangeFromTo i lb ub)) = (
    let
      i = liftChan (ppVarRef cvm i);
      (lb, ub) = (ppExpr cvm lb, ppExpr cvm ub)
    in
```

```
      ((False,ps,inl,cvm), select i lb ub))
| ppStmnt (-,cvm) (AST.StmntSelect (AST.RangeIn - -)) =
    err STR ''in not allowed in select''

| ppStmnt (-,ps,inl,cvm) (AST.StmntCall macro args) = (
    let
      args = map (liftChan ∘ ppVarRef cvm) args;
      (c,v,m,a) = cvm
    in
      case lm.lookup macro inl of
       None ⇒ errv STR ''Calling unknown macro '' macro
     | Some (names,sF) ⇒
         if length names ≠ length args then
           (err STR ''Called macro with wrong number of arguments.'')
         else
           let a' = foldl (λa (k,v). lm.update k v a) a (zip names args) in
           let ((c,v,m,-),steps) = sF (c,v,m,a') in
           ((False,ps,inl,c,v,m,a), StmntSeq steps))

| ppStmnt cvm (AST.StmntDStep -) = usc STR ''StmntDStep''

fun ppModule
  :: var-data × inlines ⇒ AST.module
     ⇒ var-data × inlines × (varDecl list + proc + ltl)
where
  ppModule (cvm, inl) (AST.ProcType act name args prio prov steps) = (
    let
      - = the-warn prio STR ''Priorities for procs not supported. Ignored.'';
      - = the-warn prov STR ''Priov (??) for procs not supported. Ignored.'';
      (cvm', args) = cvm-fold ppDeclProcArg cvm args;
      ((-, vars, -, -), steps) = cvm-fold ppStep (True,[],inl,cvm') steps
    in
      (cvm, inl, Inr (Inl (ProcType act name (concat args) vars steps))))

| ppModule (cvm,inl) (AST.Init prio steps) = (
    let - = the-warn prio STR ''Priorities for procs not supported. Ignored.'' in
    let ((-, vars, -, -), steps) = cvm-fold ppStep (True,[],inl,cvm) steps in
    (cvm, inl, Inr (Inl (Init vars steps))))

| ppModule (cvm,inl) (AST.Ltl name formula) =
    (cvm, inl, Inr (Inr (name, formula)))

| ppModule (cvm,inl) (AST.ModuDecl decl) =
    apsnd (λds. (inl,Inl ds)) (ppDecl True cvm decl)

| ppModule (cvm,inl) (AST.MType mtys) = (
    let (c,v,m,a) = cvm in
    let num = integer-of-nat (lm.size m) + 1 in
    let (m',-) = foldr (λmty (m,num).
```

```
                let m' = lm.update mty num m
                in (m',num+1)) mtys (m,num)
    in
       ((c,v,m',a), inl, Inl []))

| ppModule (cvm,inl) (AST.Inline name args steps) = (
    let stepF = (λcvm. let ((-,-,-,cvm),steps) =
                         cvm-fold ppStep (False,[],inl,cvm) steps
                       in (cvm,steps))
    in let inl = lm.update name (args, stepF) inl
    in (cvm,inl, Inl[]))

| ppModule cvm (AST.DProcType - - - - - -) = usc STR ''DProcType''
| ppModule cvm (AST.Never -) = usc STR ''Never''
| ppModule cvm (AST.Trace -) = usc STR ''Trace''
| ppModule cvm (AST.NoTrace -) = usc STR ''NoTrace''
| ppModule cvm (AST.TypeDef - -) = usc STR ''TypeDef''

definition preprocess :: AST.module list ⇒ promela where
  preprocess ms = (
    let
      dflt-vars = [(STR ''-pid'', (None, False)),
                   (STR ''--assert--'', (None, True)),
                   (STR ''-'', (None, True))];
      cvm = (lm.empty(), lm.to-map dflt-vars, lm.empty(), lm.empty());
      (-,-,pr) = (foldl (λ(cvm,inl,vs,ps,ls) m.
                           let (cvm', inl', m') = ppModule (cvm,inl) m in
                           case m' of
                             Inl v ⇒ (cvm',inl',vs@v,ps,ls)
                           | Inr (Inl p) ⇒ (cvm',inl',vs,ps@[p],ls)
                           | Inr (Inr l) ⇒ (cvm',inl',vs,ps,ls@[l]))
                         (cvm, lm.empty(),[],[],[]) ms)
      in
        pr)

fun extractLTL
  :: AST.module ⇒ ltl option
where
  extractLTL (AST.Ltl name formula) = Some (name, formula)
| extractLTL - = None

primrec extractLTLs
  :: AST.module list ⇒ (String.literal, String.literal) lm
where
  extractLTLs [] = lm.empty()
| extractLTLs (m#ms) = (case extractLTL m of
                          None ⇒ extractLTLs ms
                        | Some (n,f) ⇒ lm.update n f (extractLTLs ms))
```

definition *lookupLTL*
:: *AST.module list* ⇒ *String.literal* ⇒ *String.literal option*
where *lookupLTL ast k* = *lm.lookup k* (*extractLTLs ast*)

3.3 The transition system

The edges in our transition system consist of a condition (evaluated under the current environment) and an effect (modifying the current environment). Further they may be atomic, i. e. a whole row of such edges is taken before yielding a new state. Additionally, they carry a priority: the edges are checked from highest to lowest priority, and if one edge on a higher level can be taken, the lower levels are ignored.

The states of the system do not carry any information.

datatype *edgeCond* = *ECElse*
| *ECTrue*
| *ECFalse*
| *ECExpr expr*
| *ECRun String.literal*
| *ECSend chanRef*
| *ECRecv chanRef recvArg list bool*

datatype *edgeEffect* = *EEEnd*
| *EEId*
| *EEGoto*
| *EEAssert expr*
| *EEAssign varRef expr*
| *EEDecl procVarDecl*
| *EERun String.literal expr list*
| *EESend chanRef expr list bool*
| *EERecv chanRef recvArg list bool bool*

datatype *edgeIndex* = *Index nat* | *LabelJump String.literal nat option*
datatype *edgeAtomic* = *NonAtomic* | *Atomic* | *InAtomic*

record *edge* =
cond :: *edgeCond*
effect :: *edgeEffect*
target :: *edgeIndex*
prio :: *integer*
atomic :: *edgeAtomic*

definition *isAtomic* :: *edge* ⇒ *bool* **where**
isAtomic e = (*case atomic e of Atomic* ⇒ *True* | - ⇒ *False*)

definition *inAtomic* :: *edge* ⇒ *bool* **where**
inAtomic e = (*case atomic e of NonAtomic* ⇒ *False* | - ⇒ *True*)

3.4 State

datatype *variable* = *Var varType integer*
| *VArray varType nat integer iarray*

datatype *channel* = *Channel integer varType list integer list list*
| *HSChannel varType list*
| *InvChannel*

type-synonym *var-dict* = (*String.literal*, *variable*) *lm*
type-synonym *labels* = (*String.literal*, *nat*) *lm*
type-synonym *ltls* = (*String.literal*, *String.literal*) *lm*
type-synonym *states* = (— prio: *integer* × *edge list*) *iarray*
type-synonym *channels* = *channel list*

type-synonym *process* =
nat — offset
× *edgeIndex* — start
× *procArg list* — args
× *varDecl list* — top decls

record *program* =
processes :: *process iarray*
labels :: *labels iarray*
states :: *states iarray*
proc-names :: *String.literal iarray*
proc-data :: (*String.literal*, *nat*) *lm*

record *pState* = — State of a process
pid :: *nat* — Process identifier
vars :: *var-dict* — Dictionary of variables
pc :: *nat* — Program counter
channels :: *integer list* — List of channels created in the process. Used to close them on finalization.
idx :: *nat* — Offset into the arrays of *program*

hide-const (**open**) *idx*

record *gState* = — Global state
vars :: *var-dict* — Global variables
channels :: *channels* — Channels are by construction part of the global state, even when created in a process.
timeout :: *bool* — Set to True if no process can take a transition.
procs :: *pState list* — List of all running processes. A process is removed from it, when there is no running one with a higher index.

record $gState_I$ = *gState* + — Additional internal infos
handshake :: *nat*
hsdata :: *integer list* — Data transferred via a handshake.
exclusive :: *nat* — Set to the PID of the process, which is in an exclusive (=

atomic) state.
  else :: *bool* — Set to True for each process, if it can not take a transition. Used before timeout.

3.5 Printing

```
primrec printBinOp :: binOp ⇒ string where
  printBinOp BinOpAdd = ''+''
| printBinOp BinOpSub = ''−''
| printBinOp BinOpMul = ''∗''
| printBinOp BinOpDiv = ''/''
| printBinOp BinOpMod = ''mod''
| printBinOp BinOpGr = ''>''
| printBinOp BinOpLe = ''<''
| printBinOp BinOpGEq = ''>=''
| printBinOp BinOpLEq = ''=<''
| printBinOp BinOpEq = ''==''
| printBinOp BinOpNEq = ''!=''
| printBinOp BinOpAnd = ''&&''
| printBinOp BinOpOr = ''||''

primrec printUnOp :: unOp ⇒ string where
  printUnOp UnOpMinus = ''−''
| printUnOp UnOpNeg = ''!''

definition printList :: ('a ⇒ string) ⇒ 'a list ⇒ string ⇒ string ⇒ string ⇒
string
where
  printList f xs l r sep = (
                    let f' = (λstr x. if str = [] then f x
                                      else str @ sep @ f x)
                    in l @ (foldl f' [] xs) @ r)

lemma printList-cong [fundef-cong]:
  assumes xs = xs'
  and l = l'
  and r = r'
  and sep = sep'
  and ⋀x. x ∈ set xs ⟹ f x = f' x
  shows printList f xs l r sep = printList f' xs' l' r' sep'
  unfolding printList-def
  using assms
  by (auto intro: foldl-cong)

fun printExpr :: (integer ⇒ string) ⇒ expr ⇒ string
and printFun :: (integer ⇒ string) ⇒ string ⇒ chanRef ⇒ string
and printVarRef :: (integer ⇒ string) ⇒ varRef ⇒ string
and printChanRef :: (integer ⇒ string) ⇒ chanRef ⇒ string
and printRecvArg :: (integer ⇒ string) ⇒ recvArg ⇒ string where
```

```
  printExpr f ExprTimeOut = ''timeout''
| printExpr f (ExprBinOp binOp left right) =
    printExpr f left @ '' '' @ printBinOp binOp @ '' '' @ printExpr f right
| printExpr f (ExprUnOp unOp e) = printUnOp unOp @ printExpr f e
| printExpr f (ExprVarRef varRef) = printVarRef f varRef
| printExpr f (ExprConst i) = f i
| printExpr f (ExprMConst i m) = String.explode m
| printExpr f (ExprCond c l r) =
    ''( (( '' @ printExpr f c @ '' )) -> ''
    @ printExpr f l @ '' : ''
    @ printExpr f r @ '' )''
| printExpr f (ExprLen chan) = printFun f ''len'' chan
| printExpr f (ExprEmpty chan) = printFun f ''empty'' chan
| printExpr f (ExprFull chan) = printFun f ''full'' chan
| printExpr f (ExprPoll chan es srt) = (
    let p = if srt then ''??'' else ''?'' in
    printChanRef f chan @ p
    @ printList (printRecvArg f) es ''['' '']'' '', '')

| printVarRef - (VarRef - name None) = String.explode name
| printVarRef f (VarRef - name (Some indx)) =
    String.explode name @ ''['' @ printExpr f indx @ '']''

| printChanRef f (ChanRef v) = printVarRef f v

| printFun f fun var = fun @ ''('' @ printChanRef f var @ '')''

| printRecvArg f (RecvArgVar v) = printVarRef f v
| printRecvArg f (RecvArgConst c) = f c
| printRecvArg f (RecvArgMConst - m) = String.explode m
| printRecvArg f (RecvArgEval e) = ''eval('' @ printExpr f e @ '')''

fun printVarDecl :: (integer ⇒ string) ⇒ procVarDecl ⇒ string where
  printVarDecl f (ProcVarDeclNum - - n None None) =
    String.explode n @ '' = 0''
| printVarDecl f (ProcVarDeclNum - - n None (Some e)) =
    String.explode n @ '' = '' @ printExpr f e
| printVarDecl f (ProcVarDeclNum - - n (Some i) None) =
    String.explode n @ ''['' @ f i @ '']'' = 0''
| printVarDecl f (ProcVarDeclNum - - n (Some i) (Some e)) =
    String.explode n @ ''['' @ f i @ '']'' = '' @ printExpr f e
| printVarDecl f (ProcVarDeclChan n None) =
    ''chan '' @ String.explode n
| printVarDecl f (ProcVarDeclChan n (Some i)) =
    ''chan '' @ String.explode n @ ''['' @ f i @ '']''

primrec printCond :: (integer ⇒ string) ⇒ edgeCond ⇒ string where
  printCond f ECElse = ''else''
| printCond f ECTrue = ''true''
```

```
| printCond f ECFalse = ''false''
| printCond f (ECRun n) = ''run '' @ String.explode n @ ''(...)''
| printCond f (ECExpr e) = printExpr f e
| printCond f (ECSend c) = printChanRef f c @ ''! ...''
| printCond f (ECRecv c - -) = printChanRef f c @ ''? ...''

primrec printEffect :: (integer ⇒ string) ⇒ edgeEffect ⇒ string where
  printEffect f EEEnd = ''-- end --''
| printEffect f EEId = ''ID''
| printEffect f EEGoto = ''goto''
| printEffect f (EEAssert e) = ''assert('' @ printExpr f e @''')''
| printEffect f (EERun n -) = ''run '' @ String.explode n @ ''(...)''
| printEffect f (EEAssign v expr) =
    printVarRef f v @ '' = '' @ printExpr f expr
| printEffect f (EEDecl d) = printVarDecl f d
| printEffect f (EESend v es srt) = (
    let s = if srt then ''!!'' else ''!'' in
    printChanRef f v @ s @ printList (printExpr f) es ''('' '')'' '', '')
| printEffect f (EERecv v rs srt rem) = (
    let p = if srt then ''??'' else ''?'' in
    let (l,r) = if rem then (''('', '')'') else (''<'', ''>'') in
    printChanRef f v @ p @ printList (printRecvArg f) rs l r '', '')

primrec printIndex :: (integer ⇒ string) ⇒ edgeIndex ⇒ string where
  printIndex f (Index pos) = f (integer-of-nat pos)
| printIndex - (LabelJump l -) = String.explode l

definition printEdge :: (integer ⇒ string) ⇒ nat ⇒ edge ⇒ string where
  printEdge f indx e = (
    let
      tStr = printIndex f (target e);
      pStr = if prio e < 0 then '' Prio: '' @ f (prio e) else [];
      atom = if isAtomic e then λx. x @ '' {A}'' else id;
      pEff = λ-. atom (printEffect f (effect e));
      contStr = case (cond e) of
                  ECTrue ⇒ pEff ()
                | ECFalse ⇒ pEff ()
                | ECSend - ⇒ pEff()
                | ECRecv - - -⇒ pEff()
                | - ⇒  atom (''(( '' @ printCond f (cond e) @ '' ))'')
    in
      f (integer-of-nat indx) @ '' ---> '' @ tStr @ '' => '' @ contStr @ pStr)

definition printEdges :: (integer ⇒ string) ⇒ states ⇒ string list where
  printEdges f es = concat (map (λn. map (printEdge f n) (snd (es !! n)))
                                (rev [0..<IArray.length es]))

definition printLabels :: (integer ⇒ string) ⇒ labels ⇒ string list where
  printLabels f ls = lm.iterate ls (λ(k,l) res.
```

```
                                    (''Label '' @ String.explode k @ '': ''
                                     @ f (integer-of-nat l)) # res) []

fun printProcesses :: (integer ⇒ string) ⇒ program ⇒ string list where
  printProcesses f prog = lm.iterate (proc-data prog)
    (λ(k,idx) res.
          let (-,start,-,-) = processes prog !! idx in
          [] # (''Process '' @ String.explode k) # [] # printEdges f (states prog !!
idx)
          @ [''START −−−> '' @ printIndex f start, []]
          @ printLabels f (labels prog !! idx) @ res) []
end
```

4 Invariants for Promela data structures

```
theory PromelaInvariants
imports PromelaDatastructures
begin
```

The different data structures used in the Promela implementation require different invariants, which are specified in this file. As there is no (useful) way of specifying *correctness* of the implementation, those invariants are tailored towards proving the finitness of the generated state-space.

4.1 Bounds

Finiteness requires that possible variable ranges are finite, as is the maximium number of processes. Currently, they are supplied here as constants. In a perfect world, they should be able to be set dynamically.

```
definition min-var-value :: integer where
  min-var-value = −(2^31)
definition max-var-value :: integer where
  max-var-value = (2^31) − 1

lemma min-max-var-value-simps [simp, intro!]:
  min-var-value < max-var-value
  min-var-value < 0
  min-var-value ≤ 0
  max-var-value > 0
  max-var-value ≥ 0
by (simp-all add: min-var-value-def max-var-value-def)

definition max-procs ≡ 255
definition max-channels ≡ 65535
definition max-array-size = 65535
```

4.2 Variables and similar

```
fun varType-inv :: varType ⇒ bool where
  varType-inv (VTBounded l h)
  ⟷ l ≥ min-var-value ∧ h ≤ max-var-value ∧ l < h
| varType-inv VTChan ⟷ True

fun variable-inv :: variable ⇒ bool where
  variable-inv (Var t val)
  ⟷ varType-inv t ∧ val ∈ {min-var-value..max-var-value}
| variable-inv (VArray t sz ar)
  ⟷ varType-inv t
    ∧ sz ≤ max-array-size
    ∧ IArray.length ar = sz
    ∧ set (IArray.list-of ar) ⊆ {min-var-value..max-var-value}

fun channel-inv :: channel ⇒ bool where
  channel-inv (Channel cap ts q)
  ⟷ cap ≤ max-array-size
    ∧ cap ≥ 0
    ∧ set ts ⊆ Collect varType-inv
    ∧ length ts ≤ max-array-size
    ∧ length q ≤ max-array-size
    ∧ (∀ x ∈ set q. length x = length ts
    ∧ set x ⊆ {min-var-value..max-var-value})
| channel-inv (HSChannel ts)
  ⟷ set ts ⊆ Collect varType-inv ∧ length ts ≤ max-array-size
| channel-inv InvChannel ⟷ True

lemma varTypes-finite:
  finite (Collect varType-inv)
proof (rule finite-subset)
  show Collect (varType-inv) ⊆
      {VTChan}
    ∪ (λ(l,h). VTBounded l h)
      ` ({min-var-value..max-var-value} × {min-var-value..max-var-value})
    apply (rule subsetI)
    apply (case-tac x)
     apply auto
    done

  show finite ... by auto
qed

lemma variables-finite:
  finite (Collect variable-inv)
proof (rule finite-subset)
  let ?mm = {min-var-value..max-var-value}
  let ?V1 = (λ(t,val). Var t val) ` ({vt. varType-inv vt} × ?mm)
  let ?V2 = (λ(t,sz,ar). VArray t sz ar)
```

```
    ‘ ({vt. varType-inv vt}
      × {0..max-array-size}
      × {ar. IArray.length ar ≤ max-array-size
          ∧ set (IArray.list-of ar) ⊆ ?mm})

  {
    fix A :: 'a set
    let ?LS = {xs. set xs ⊆ A ∧ length xs ≤ max-array-size }
    let ?AS = {ar. IArray.length ar ≤ max-array-size
      ∧ set (IArray.list-of ar) ⊆ A}

    assume finite A
    hence finite ?LS by (simp add: finite-lists-length-le)
    moreover have ?AS ⊆ IArray ‘ ?LS
      apply (auto simp: image-def)
      apply (rule-tac x = IArray.list-of x in exI)
      apply auto
      apply (metis iarray.exhaust list-of.simps)
      done
    ultimately have finite ?AS by (auto simp add: finite-subset)
  } note finite-arr = this

  show Collect variable-inv ⊆ (?V1 ∪ ?V2)
    apply (rule subsetI)
    apply (case-tac x)
      apply (auto simp add: image-def)
    done

  show finite ... by (blast intro: varTypes-finite finite-arr)
qed

lemma channels-finite:
  finite (Collect channel-inv)
proof (rule finite-subset)
  let ?C1 =
    (λ(cap,ts,q). Channel cap ts q)
     ‘ ({0..max-array-size}
      × {ts. set ts ⊆ Collect varType-inv ∧ length ts ≤ max-array-size}
      × {q. set q ⊆ {x. set x ⊆ {min-var-value..max-var-value}
                    ∧ length x ≤ max-array-size}
          ∧ length q ≤ max-array-size})
  let ?C2 =
    HSChannel ‘ {ts. set ts ⊆ Collect varType-inv ∧ length ts ≤ max-array-size}
  let ?C3 = {InvChannel}

  show (Collect channel-inv) ⊆ ?C1 ∪ ?C2 ∪ ?C3
    apply (rule subsetI)
    apply (case-tac x)
      apply (auto simp add: image-def)
```

done

show *finite* ... **by** (*blast intro*: *finite-lists-length-le varTypes-finite*)+
qed

To give an upper bound of variable names, we need a way to calculate it.

primrec *procArgName* :: *procArg* ⇒ *String.literal* **where**
procArgName (*ProcArg* - *name*) = *name*

primrec *varDeclName* :: *varDecl* ⇒ *String.literal* **where**
varDeclName (*VarDeclNum* - - *name* - -) = *name*
| *varDeclName* (*VarDeclChan name* - -) = *name*

primrec *procVarDeclName* :: *procVarDecl* ⇒ *String.literal* **where**
procVarDeclName (*ProcVarDeclNum* - - *name* - -) = *name*
| *procVarDeclName* (*ProcVarDeclChan name* -) = *name*

definition *edgeDecls* :: *edge* ⇒ *procVarDecl set* **where**
edgeDecls e = (
case effect e of
EEDecl p ⇒ {*p*}
| - ⇒ {})

lemma *edgeDecls-finite*:
finite (*edgeDecls e*)
by (*simp add*: *edgeDecls-def split*: *edgeEffect.split*)

definition *edgeSet* :: *states* ⇒ *edge set* **where**
edgeSet s = *set* (*concat* (*map snd* (*IArray.list-of s*)))

lemma *edgeSet-finite*:
finite (*edgeSet s*)
by (*simp add*: *edgeSet-def*)

definition *statesDecls* :: *states* ⇒ *procVarDecl set* **where**
statesDecls s = ⋃(*edgeDecls* ‘ (*edgeSet s*))

definition *statesNames* :: *states* ⇒ *String.literal set* **where**
statesNames s = *procVarDeclName* ‘ *statesDecls s*

lemma *statesNames-finite*:
finite (*statesNames s*)
by (*simp add*: *edgeSet-finite edgeDecls-finite statesNames-def statesDecls-def*)

fun *process-names* :: *states* ⇒ *process* ⇒ *String.literal set* **where**
process-names ss (-, -, *args*, *decls*) =
statesNames ss
∪ *procArgName* ‘ *set args*

$\cup$ *varDeclName ‘ set decls*
$\cup$ {*STR ''-''*, *STR ''--assert--''*, *STR ''-pid''*}

lemma *process-names-finite*:
finite (*process-names ss p*)
by (*cases p*) (*simp add*: *statesNames-finite*)

definition *vardict-inv* :: *states* $\Rightarrow$ *process* $\Rightarrow$ *var-dict* $\Rightarrow$ *bool* **where**
vardict-inv ss p vs
$\longleftrightarrow$ *lm.ball vs* (λ(*k*,*v*). *k* $\in$ *process-names ss p* $\wedge$ *variable-inv v*)

lemma *vardicts-finite*:
finite (*Collect* (*vardict-inv ss p*))
proof −
have *Assoc-List.set ‘ Collect* (*vardict-inv ss p*) $\subseteq$
Pow (*process-names ss p* $\times$ {*v*. *variable-inv v*})
by (*auto simp add*: *lm-ball-Assoc-List-set vardict-inv-def*)

moreover have *finite ...*
using *process-names-finite variables-finite*
by *simp*
ultimately show *?thesis* **by** (*metis finite-Assoc-List-set-image finite-subset*)
qed

lemma *lm-to-map-vardict-inv*:
assumes $\forall$ (*k*,*v*) $\in$ *set xs*. *k* $\in$ *process-names ss proc* $\wedge$ *variable-inv v*
shows *vardict-inv ss proc* (*lm.to-map xs*)
using *assms*
unfolding *vardict-inv-def*
by (*auto simp add*: *lm.correct dest*: *map-of-SomeD*)

4.3 Invariants of a process

definition *pState-inv* :: *program* $\Rightarrow$ *pState* $\Rightarrow$ *bool* **where**
pState-inv prog p
$\longleftrightarrow$ *pid p* $\leq$ *max-procs*
$\wedge$ *pState.idx p* < *IArray.length* (*states prog*)
$\wedge$ *IArray.length* (*states prog*) = *IArray.length* (*processes prog*)
$\wedge$ *pc p* < *IArray.length* ((*states prog*) !! *pState.idx p*)
$\wedge$ *set* (*pState.channels p*) $\subseteq$ {−*1*..<*integer-of-nat max-channels*}
$\wedge$ *length* (*pState.channels p*) $\leq$ *max-channels*
$\wedge$ *vardict-inv* ((*states prog*) !! *pState.idx p*)
((*processes prog*) !! *pState.idx p*)
(*pState.vars p*)

lemma *pStates-finite*:
finite (*Collect* (*pState-inv prog*))
proof −
let *?P1* = {*..max-procs*::*nat*}

```
  let ?P2 = {..IArray.length (states prog)}
  let ?P3 = {..Max (IArray.length ` (set (IArray.list-of (states prog))))}
  let ?P4 = {cs. set cs ⊆ {−1..<integer-of-nat max-channels}
                ∧ length cs ≤ max-channels}
  let ?P5 = ⋃x∈{..IArray.length (states prog)}.
              Collect (vardict-inv (states prog !! x) (processes prog !! x))
  let ?P = ?P1 × ?P2 × ?P3 × ?P4 × ?P5

  have {p. pState-inv prog p} ⊆
    (λ(pid,idx,pc,channels,vars). pState.make pid vars pc channels idx) ` ?P
    unfolding pState-inv-def image-def [of - ?P]
    apply (clarsimp simp add: pState.defs)
    apply (tactic ‹Record.split-simp-tac @{context} [] (K ~1) 1›)
    apply auto
    apply (rule order-trans [OF less-imp-le])
    apply (auto intro!: Max-ge)
    done
  moreover
  have finite ?P4 by (fastforce intro: finite-lists-length-le)
  hence finite ?P by (auto intro: finite-cartesian-product simp: vardicts-finite)

  ultimately show ?thesis by (elim finite-subset) (rule finite-imageI)
qed
```

Throughout the calculation of the semantic engine, a modified process is not necessarily part of *procs g*. Hence we need to establish an additional constraint for the relation between a global and a process state.

```
definition cl-inv :: ('a gState-scheme * pState) ⇒ bool where
  cl-inv gp = (case gp of (g,p) ⇒
      length (pState.channels p) ≤ length (gState.channels g))

lemma cl-inv-lengthD:
  cl-inv (g,p) ⟹ length (pState.channels p) ≤ length (gState.channels g)
unfolding cl-inv-def
by auto

lemma cl-invI:
  length (pState.channels p) ≤ length (gState.channels g) ⟹ cl-inv (g,p)
unfolding cl-inv-def by auto

lemma cl-inv-trans:
  length (channels g) ≤ length (channels g') ⟹ cl-inv (g,p) ⟹ cl-inv (g',p)
by (simp add: cl-inv-def)

lemma cl-inv-vars-update[intro!]:
  cl-inv (g,p) ⟹ cl-inv (g, pState.vars-update vs p)
  cl-inv (g,p) ⟹ cl-inv (gState.vars-update vs g, p)
by (simp-all add: cl-inv-def)
```

lemma *cl-inv-handshake-update*[*intro*!]:
cl-inv (*g*,*p*) ⟹ *cl-inv* (*g*(|*handshake* := *h*|),*p*)
by (*simp add*: *cl-inv-def*)

lemma *cl-inv-hsdata-update*[*intro*!]:
cl-inv (*g*,*p*) ⟹ *cl-inv* (*g*(|*hsdata* := *h*|),*p*)
by (*simp add*: *cl-inv-def*)

lemma *cl-inv-procs-update*[*intro*!]:
cl-inv (*g*,*p*) ⟹ *cl-inv* (*g*(|*procs* := *ps*|),*p*)
by (*simp add*: *cl-inv-def*)

lemma *cl-inv-channels-update*:
assumes *cl-inv* (*g*,*p*)
shows *cl-inv* (*gState.channels-update* (λ*cs*. *cs*[*i*:=*c*]) *g*, *p*)
using *assms* **unfolding** *cl-inv-def*
by *simp*

4.4 Invariants of the global state

Note that *gState-inv* must be defined in a way to be applicable to both *gState* and $gState_I$.

definition *gState-inv* :: *program* ⇒ *'a gState-scheme* ⇒ *bool* **where**
gState-inv prog g
⟷ *length* (*procs g*) ≤ *max-procs*
∧ (∀ *p* ∈ *set* (*procs g*). *pState-inv prog p* ∧ *cl-inv* (*g*,*p*))
∧ *length* (*channels g*) ≤ *max-channels*
∧ *set* (*channels g*) ⊆ *Collect channel-inv*
∧ *lm.ball* (*vars g*) (λ(*k*,*v*). *variable-inv v*)

The set of global states adhering to the terms of *gState-inv* is not finite. But the set of all global states that can be constructed by the semantic engine from one starting state is. Thus we establish a progress relation, i. e. all successors of a state *g* relate to *g* under this specification.

definition *gState-progress-rel* :: *program* ⇒ (*'a gState-scheme*) *rel* **where**
gState-progress-rel p = {(*g*,*g'*). *gState-inv p g* ∧ *gState-inv p g'*
∧ *length* (*channels g*) ≤ *length* (*channels g'*)
∧ *dom* (*lm.α* (*vars g*)) = *dom* (*lm.α* (*vars g'*))}

lemma *gState-progress-rel-gState-invI1*[*intro*]:
(*g*,*g'*) ∈ *gState-progress-rel prog* ⟹ *gState-inv prog g*
by (*simp add*: *gState-progress-rel-def*)

lemma *gState-progress-rel-gState-invI2*[*intro*]:
(*g*,*g'*) ∈ *gState-progress-rel prog* ⟹ *gState-inv prog g'*
by (*simp add*: *gState-progress-rel-def*)

lemma *gState-progress-relI*:

assumes *gState-inv prog g*
and *gState-inv prog g′*
and *length* (*channels g*) ≤ *length* (*channels g′*)
and *dom* (*lm.α* (*vars g*)) = *dom* (*lm.α* (*vars g′*))
shows (*g*,*g′*) ∈ *gState-progress-rel prog*
unfolding *gState-progress-rel-def*
using *assms*
by *auto*

lemma *gState-progress-refl*[*simp*,*intro*!]:
gState-inv prog g ⟹ (*g*,*g*) ∈ (*gState-progress-rel prog*)
unfolding *gState-progress-rel-def*
by *auto*

lemma *refl-on-gState-progress-rel*:
refl-on (*Collect* (*gState-inv prog*)) (*gState-progress-rel prog*)
by (*auto intro*!: *refl-onI*)

lemma *trans-gState-progress-rel*[*simp*]:
trans (*gState-progress-rel prog*)
by (*intro transI*) (*simp add*: *gState-progress-rel-def*)

lemmas *gState-progress-rel-trans* [*trans*] = *trans-gState-progress-rel*[*THEN transD*]

lemma *gState-progress-rel-trancl-id*[*simp*]:
(*gState-progress-rel prog*)$^+$ = *gState-progress-rel prog*
by *simp*

lemma *gState-progress-rel-rtrancl-absorb*:
assumes *gState-inv prog g*
shows (*gState-progress-rel prog*)* `` {*g*} = *gState-progress-rel prog* `` {*g*}
using *assms refl-on-gState-progress-rel*
by (*intro Image-absorb-rtrancl*) *auto*

The main theorem: The set of all global states reachable from an initial state, is finite.

lemma *gStates-finite*:
fixes *g* :: *gState*
shows *finite* ((*gState-progress-rel prog*)* `` {*g*})
proof (*cases gState-inv prog g*)
case *False* **hence** (*gState-progress-rel prog*)* `` {*g*} = {*g*}
by (*intro Image-empty-rtrancl-Image-id*)
(*auto simp add*: *gState-progress-rel-def*)
thus *?thesis* **by** *simp*
next
case *True*
let *?G1* = {*m*. *dom* (*lm.α m*) = *dom* (*lm.α* (*vars g*))
∧ *ran* (*lm.α m*) ⊆ *Collect variable-inv* }
let *?G2* = {*cs*. *set cs* ⊆ *Collect channel-inv*

```
                ∧ length cs ≤ max-channels}
  let ?G3 = {True, False}
  let ?G4 = {ps. set ps ⊆ Collect (pState-inv prog)
                ∧ length ps ≤ max-procs}

  let ?G = ?G1 × ?G2 × ?G3 × ?G4
  let ?G' = (λ(vars,chans,t,ps). gState.make vars chans t ps) ‘ ?G

  have G1: finite ?G1
  proof (rule finite-subset)
    show ?G1 ⊆ {v'. fst ‘ Assoc-List.set v' = fst ‘ Assoc-List.set (vars g)
                  ∧ snd ‘ Assoc-List.set v' ⊆ Collect variable-inv}
      by (simp add: dom-lm-α-Assoc-List-set ran-lm-α-Assoc-List-set)
    show finite ... (is finite ?X)
    proof (rule finite-Assoc-List-set-image, rule finite-subset)
      show Assoc-List.set ‘ ?X ⊆
            Pow (fst ‘ Assoc-List.set (vars g) × Collect variable-inv)
        by auto
      show finite ... by (auto simp add: variables-finite dom-lm-α-Assoc-List-set[symmetric])
    qed
  qed

  have finite ((gState-progress-rel prog) `` {g})
  proof (rule finite-subset)
    show (gState-progress-rel prog) `` {g} ⊆
          (λ(vars,chans,t,ps). gState.make vars chans t ps) ‘ ?G
    apply (clarsimp simp add: image-def gState-inv-def gState.defs gState-progress-rel-def)
      apply (rule-tac x = vars x in exI)
      apply (simp add: lm-ball-eq-ran)
      apply (rule-tac x = channels x in exI)
      apply (case-tac timeout x)
        apply clarsimp
        apply (rule-tac x=procs x in exI)
        apply auto
      done
    show finite ... using G1
      by (blast intro: finite-lists-length-le channels-finite pStates-finite)
  qed
  with gState-progress-rel-rtrancl-absorb[OF True] show ?thesis by simp
qed

lemma gState-progress-rel-channels-update:
  assumes gState-inv prog g
  and channel-inv c
  and i < length (channels g)
  shows (g,gState.channels-update (λcs. cs[i:=c]) g) ∈ gState-progress-rel prog
using assms
by (auto intro!: gState-progress-relI
        simp add: gState-inv-def cl-inv-def
```

dest!: *subsetD*[*OF set-update-subset-insert*])

lemma *gState-progress-rel-channels-update-step*:
assumes *gState-inv prog g*
and *step*: (*g*,*g'*) ∈ *gState-progress-rel prog*
and *channel-inv c*
and *i* < *length* (*channels g'*)
shows (*g*,*gState.channels-update* (λ*cs*. *cs*[*i*:=*c*]) *g'*) ∈ *gState-progress-rel prog*
proof −
note *step*
also hence *gState-inv prog g'* **by** *blast*
note *gState-progress-rel-channels-update*[*OF this assms*(*3*,*4*)]
finally show *?thesis* **.**
qed

4.5 Invariants of the program

Naturally, we need our program to also adhere to certain invariants. Else we can't show, that the generated states are correct according to the invariants above.

definition *program-inv* **where**
program-inv prog
⟷ *IArray.length* (*states prog*) > *0*
∧ *IArray.length* (*states prog*) = *IArray.length* (*processes prog*)
∧ (∀ *s* ∈ *set* (*IArray.list-of* (*states prog*)). *IArray.length s* > *0*)
∧ *lm.ball* (*proc-data prog*)
(λ(-,*sidx*).
sidx < *IArray.length* (*processes prog*)
∧ *fst* (*processes prog* !! *sidx*) = *sidx*)
∧ (∀(*sidx*,*start*,*procArgs*,*args*) ∈ *set* (*IArray.list-of* (*processes prog*)).
(∃ *s*. *start* = *Index s* ∧ *s* < *IArray.length* (*states prog* !! *sidx*)))

lemma *program-inv-length-states*:
assumes *program-inv prog*
and *n* < *IArray.length* (*states prog*)
shows *IArray.length* (*states prog* !! *n*) > *0*
using *assms* **by** (*simp add*: *program-inv-def*)

lemma *program-invI*:
assumes *0* < *IArray.length* (*states prog*)
and *IArray.length* (*states prog*) = *IArray.length* (*processes prog*)
and ⋀*s*. *s* ∈ *set* (*IArray.list-of* (*states prog*))
⟹ *0* < *IArray.length s*
and ⋀*sidx*. *sidx* ∈ *ran* (*lm.α* (*proc-data prog*))
⟹ *sidx* < *IArray.length* (*processes prog*)
∧ *fst* (*processes prog* !! *sidx*) = *sidx*
and ⋀*sidx start procArgs args*.
(*sidx*,*start*,*procArgs*,*args*) ∈ *set* (*IArray.list-of* (*processes prog*))
⟹ ∃ *s*. *start* = *Index s* ∧ *s* < *IArray.length* (*states prog* !! *sidx*)

shows *program-inv prog*
unfolding *program-inv-def*
using *assms*
by (*auto simp add*: *lm-ball-eq-ran*)

end

5 Formalization of Promela semantics

theory *Promela*
imports
PromelaDatastructures
PromelaInvariants
PromelaStatistics
begin

Auxiliary

lemma *mod-integer-le*:
‹*x mod* (*a* + *1*) ≤ *b*› **if** ‹*a* ≤ *b*› ‹*0* < *a*› **for** *a b x* :: *integer*
using *that* **including** *integer.lifting* **proof** *transfer*
fix *a b x* :: *int*
assume ‹*0* < *a*› ‹*a* ≤ *b*›
have ‹*x mod* (*a* + *1*) < *a* + *1*›
by (*rule pos-mod-bound*) (*use* ‹*0* < *a*› **in** *simp*)
with ‹*a* ≤ *b*› **show** ‹*x mod* (*a* + *1*) ≤ *b*›
by *simp*
qed

lemma *mod-integer-ge*:
‹*b* ≤ *x mod* (*a* + *1*)› **if** ‹*b* ≤ *0*› ‹*0* < *a*› **for** *a b x* :: *integer*
using *that* **including** *integer.lifting* **proof** *transfer*
fix *a b x* :: *int*
assume ‹*b* ≤ *0*› ‹*0* < *a*›
then have ‹*0* ≤ *x mod* (*a* + *1*)›
by *simp*
with ‹*b* ≤ *0*› **show** ‹*b* ≤ *x mod* (*a* + *1*)›
by *simp*
qed

After having defined the datastructures, we present in this theory how to construct the transition system and how to generate the successors of a state, i. e. the real semantics of a Promela program. For the first task, we take the enriched AST as input, the second one operates on the transition system.

5.1 Misc Helpers

definition *add-label* :: *String.literal* ⇒ *labels* ⇒ *nat* ⇒ *labels* **where**

```
  add-label l lbls pos = (
    case lm.lookup l lbls of
      None ⇒ lm.update l pos lbls
    | Some - ⇒ abortv STR ''Label given twice: '' l (λ-. lbls))

definition min-prio :: edge list ⇒ integer ⇒ integer where
  min-prio es start = Min ((prio ‘ set es) ∪ {start})

lemma min-prio-code [code]:
  min-prio es start = fold  (λe pri. if prio e < pri then prio e else pri) es start
proof −
  from Min.set-eq-fold have Min (set (start # map prio es)) = fold min (map
prio es) start by metis
  also have ... = fold (min ∘ prio) es start by (simp add: fold-map)
  also have ... = fold  (λe pri. if prio e < pri then prio e else pri) es start by
(auto intro!: fold-cong)
  finally show ?thesis by (simp add: min-prio-def)
qed

definition for-all :: ('a ⇒ bool) ⇒ 'a list ⇒ bool where
  for-all f xs ⟷ (∀ x ∈ set xs. f x)

lemma for-all-code[code]:
  for-all f xs ⟷ foldli xs id (λkv σ. f kv) True
  by (simp add: for-all-def foldli-conj)

definition find-remove :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a option × 'a list where
  find-remove P xs = (case List.find P xs of None ⇒ (None, xs)
                                          | Some x ⇒ (Some x, List.remove1 x xs))

lemma find-remove-code [code]:
  find-remove P [] = (None, [])
  find-remove P (x#xs) = (if P x then (Some x, xs)
                          else apsnd (Cons x) (find-remove P xs))
 by (induct xs) (auto simp add: find-remove-def dest: find-SomeD split: option.split)


lemma find-remove-subset:
  find-remove P xs = (res, xs') ⟹ set xs' ⊆ set xs
unfolding find-remove-def
using set-remove1-subset
by (force split: option.splits)

lemma find-remove-length:
  find-remove P xs = (res, xs') ⟹ length xs' ≤ length xs
unfolding find-remove-def
by (induct xs arbitrary: res xs') (auto split: if-splits option.splits)
```

5.2 Variable handling

Handling variables, with their different scopes (global vs. local), and their different types (array vs channel vs bounded) is one of the main challenges of the implementation.

```
fun lookupVar :: variable ⇒ integer option ⇒ integer where
  lookupVar (Var - val) None = val
| lookupVar (Var - -) (Some -) = abort STR ''Array used on var'' (λ-.0)
| lookupVar (VArray - - vals) None = vals !! 0
| lookupVar (VArray - siz vals) (Some idx) = vals !! nat-of-integer idx

primrec checkVarValue :: varType ⇒ integer ⇒ integer where
  checkVarValue (VTBounded lRange hRange) val = (
    if val ≤ hRange ∧ val ≥ lRange then val
    else — overflowing is well-defined and may actually be used (e.g. bool)
      if lRange = 0 ∧ val > 0
      then val mod (hRange + 1)
      else — we do not want to implement C-semantics (ie type casts)
        abort STR ''Value overflow'' (λ-. lRange))
| checkVarValue VTChan val = (
    if val < min-var-value ∨ val > max-var-value
    then abort STR ''Value overflow'' (λ-. 0)
    else val)

lemma [simp]:
  variable-inv (Var VTChan 0)
by simp

context
  fixes type :: varType
  assumes varType-inv type
begin

lemma checkVarValue-bounded:
  checkVarValue type val ∈ {min-var-value..max-var-value}
  using ‹varType-inv type›
  by (cases type) (auto intro: mod-integer-le mod-integer-ge)

lemma checkVarValue-bounds:
  min-var-value ≤ checkVarValue type val
  checkVarValue type val ≤ max-var-value
  using checkVarValue-bounded [of val] by simp-all

lemma checkVarValue-Var:
  variable-inv (Var type (checkVarValue type val))
  using ‹varType-inv type› by (simp add: checkVarValue-bounds)

end
```

```
fun editVar :: variable ⇒ integer option ⇒ integer ⇒ variable where
  editVar (Var type - ) None val = Var type (checkVarValue type val)
| editVar (Var - -) (Some -) - = abort STR ''Array used on var'' (λ-. Var VTChan
0)
| editVar (VArray type siz vals) None val = (
    let lv = IArray.list-of vals in
    let v' = lv[0:=checkVarValue type val] in
    VArray type siz (IArray v'))
| editVar (VArray type siz vals) (Some idx) val = (
    let lv = IArray.list-of vals in
    let v' = lv[(nat-of-integer idx):=checkVarValue type val] in
    VArray type siz (IArray v'))

lemma editVar-variable-inv:
  assumes variable-inv v
  shows variable-inv (editVar v idx val)
proof (cases v)
  case (Var type val) with assms have varType-inv type by simp
  with Var show ?thesis
    by (cases idx)
      (auto intro!: checkVarValue-Var
          simp del: checkVarValue.simps variable-inv.simps)
next
  case (VArray type siz vals)
  with assms have [simp, intro!]: varType-inv type by simp

  show ?thesis
  proof (cases idx)
    case None with assms VArray show ?thesis
      by (cases IArray.list-of vals) (auto intro!: checkVarValue-bounds)
  next
    case (Some i)
   note upd-cases = in-set-upd-cases[where l=IArray.list-of vals and i=nat-of-integer
i]

    from Some VArray assms show ?thesis
      by (cases type)
          (auto elim!: upd-cases intro!: mod-integer-le mod-integer-ge simp add:
min-var-value-def)
  qed
qed

definition getVar'
  :: bool ⇒ String.literal ⇒ integer option
      ⇒ 'a gState-scheme ⇒ pState
      ⇒ integer option
where
  getVar' gl v idx g p = (
        let vars = if gl then gState.vars g else pState.vars p in
```

```
        map-option (λx. lookupVar x idx) (lm.lookup v vars))

definition setVar'
  :: bool ⇒ String.literal ⇒ integer option
      ⇒ integer
      ⇒ 'a gState-scheme ⇒ pState
      ⇒ 'a gState-scheme ∗ pState
 where
  setVar' gl v idx val g p = (
    if gl then
      if v = STR ''-'' then (g,p) — ''-'' is a write-only scratch variable
      else case lm.lookup v (gState.vars g) of
             None ⇒ abortv STR ''Unknown global variable: '' v (λ-. (g,p))
           | Some x ⇒ (g(|gState.vars := lm.update v (editVar x idx val)
                                                   (gState.vars g)|)
                      , p)
    else
      case lm.lookup v (pState.vars p) of
        None ⇒ abortv STR ''Unknown proc variable: '' v (λ-. (g,p))
      | Some x ⇒ (g, p(|pState.vars := lm.update v (editVar x idx val)
                                                 (pState.vars p)|)))

lemma setVar'-gState-inv:
  assumes gState-inv prog g
  shows gState-inv prog (fst (setVar' gl v idx val g p))
unfolding setVar'-def using assms
by (auto simp add: gState-inv-def lm.correct
         intro: editVar-variable-inv
         split: option.splits)

lemma setVar'-gState-progress-rel:
  assumes gState-inv prog g
  shows (g, fst (setVar' gl v idx val g p)) ∈ gState-progress-rel prog
apply (intro gState-progress-relI)
     apply (fact assms)
    apply (fact setVar'-gState-inv[OF assms])
  apply (auto simp: setVar'-def lm.correct split: option.splits)
done

lemma vardict-inv-process-names:
  assumes vardict-inv ss proc v
  and lm.lookup k v = Some x
  shows k ∈ process-names ss proc
using assms
by (auto simp add: lm.correct vardict-inv-def)

lemma vardict-inv-variable-inv:
  assumes vardict-inv ss proc v
  and lm.lookup k v = Some x
```

shows *variable-inv x*
using *assms*
by (*auto simp add*: *lm.correct vardict-inv-def*)

lemma *vardict-inv-updateI*:
assumes *vardict-inv ss proc vs*
and *x* ∈ *process-names ss proc*
and *variable-inv v*
shows *vardict-inv ss proc* (*lm.update x v vs*)
using *assms*
by (*auto simp add*: *lm.correct vardict-inv-def*)

lemma *update-vardict-inv*:
assumes *vardict-inv ss proc v*
and *lm.lookup k v* = *Some x*
and *variable-inv x'*
shows *vardict-inv ss proc* (*lm.update k x' v*)
using *assms*
by (*auto intro*!: *vardict-inv-updateI vardict-inv-process-names*)

lemma *setVar'-pState-inv*:
assumes *pState-inv prog p*
shows *pState-inv prog* (*snd* (*setVar' gl v idx val g p*))
unfolding *setVar'-def* **using** *assms*
by (*auto split*: *if-splits option.splits*
simp add: *pState-inv-def*
intro: *update-vardict-inv editVar-variable-inv vardict-inv-variable-inv*)

lemma *setVar'-cl-inv*:
assumes *cl-inv* (*g*,*p*)
shows *cl-inv* (*setVar' gl v idx val g p*)
unfolding *setVar'-def* **using** *assms*
by (*auto split*: *if-splits option.splits*)

definition *withVar'*
:: *bool* ⇒ *String.literal* ⇒ *integer option*
⇒ (*integer* ⇒ *'x*)
⇒ *'a gState-scheme* ⇒ *pState*
⇒ *'x*
where
withVar' gl v idx f g p = *f* (*the* (*getVar' gl v idx g p*))

definition *withChannel'*
:: *bool* ⇒ *String.literal* ⇒ *integer option*
⇒ (*nat* ⇒ *channel* ⇒ *'x*)
⇒ *'a gState-scheme* ⇒ *pState*
⇒ *'x*
where
withChannel' gl v idx f g p = (

```
  let error = λ-. abortv STR ''Variable is not a channel: '' v
                        (λ-. f 0 InvChannel) in
  let abort = λ-. abortv STR ''Channel already closed / invalid: '' v
                        (λ-. f 0 InvChannel)
  in withVar' gl v idx (λi. let i = nat-of-integer i in
                        if i ≥ length (channels g) then error ()
                        else let c = channels g ! i in
                            case c of
                              InvChannel ⇒ abort ()
                            | - ⇒ f i c) g p)
```

5.3 Expressions

Expressions are free of side-effects.

This is in difference to SPIN, where *run* is an expression with side-effect. We treat *run* as a statement.

```
abbreviation trivCond x ≡ if x then 1 else 0

fun exprArith :: 'a gState-scheme ⇒ pState ⇒ expr ⇒ integer
and pollCheck :: 'a gState-scheme ⇒ pState ⇒ channel ⇒ recvArg list ⇒ bool
                ⇒ bool
and recvArgsCheck :: 'a gState-scheme ⇒ pState ⇒ recvArg list ⇒ integer list
                    ⇒ bool
where
  exprArith g p (ExprConst x) = x
| exprArith g p (ExprMConst x -) = x

| exprArith g p ExprTimeOut = trivCond (timeout g)

| exprArith g p (ExprLen (ChanRef (VarRef gl name None))) =
    withChannel' gl name None (
      λ- c. case c of
              Channel - - q ⇒ integer-of-nat (length q)
            | HSChannel - ⇒ 0) g p

| exprArith g p (ExprLen (ChanRef (VarRef gl name (Some idx)))) =
    withChannel' gl name (Some (exprArith g p idx)) (
     λ- c. case c of
             Channel - - q ⇒ integer-of-nat (length q)
           | HSChannel - ⇒ 0) g p

| exprArith g p (ExprEmpty (ChanRef (VarRef gl name None))) =
    trivCond (withChannel' gl name None (
      λ- c. case c of Channel - - q ⇒ (q = [])
                    | HSChannel - ⇒ True) g p)

| exprArith g p (ExprEmpty (ChanRef (VarRef gl name (Some idx)))) =
    trivCond (withChannel' gl name (Some (exprArith g p idx)) (
```

```
      λ- c. case c of Channel - - q ⇒ (q = [])
                    | HSChannel - ⇒ True) g p)

| exprArith g p (ExprFull (ChanRef(VarRef gl name None))) =
    trivCond (withChannel′ gl name None (
      λ- c. case c of
              Channel cap - q ⇒ integer-of-nat (length q) ≥ cap
            | HSChannel - ⇒ False) g p)

| exprArith g p (ExprFull (ChanRef(VarRef gl name (Some idx)))) =
    trivCond (withChannel′ gl name (Some (exprArith g p idx)) (
      λ- c. case c of
              Channel cap - q ⇒ integer-of-nat (length q) ≥ cap
            | HSChannel - ⇒ False) g p)

| exprArith g p (ExprVarRef (VarRef gl name None)) =
    withVar′ gl name None id g p

| exprArith g p (ExprVarRef (VarRef gl name (Some idx))) =
    withVar′ gl name (Some (exprArith g p idx)) id g p

| exprArith g p (ExprUnOp UnOpMinus expr) = 0 − exprArith g p expr
| exprArith g p (ExprUnOp UnOpNeg expr) = ((exprArith g p expr) + 1) mod 2

| exprArith g p (ExprBinOp BinOpAdd lexpr rexpr) =
    (exprArith g p lexpr) + (exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpSub lexpr rexpr) =
    (exprArith g p lexpr) − (exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpMul lexpr rexpr) =
    (exprArith g p lexpr) ∗ (exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpDiv lexpr rexpr) =
    (exprArith g p lexpr) div (exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpMod lexpr rexpr) =
    (exprArith g p lexpr) mod (exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpGr lexpr rexpr) =
    trivCond (exprArith g p lexpr > exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpLe lexpr rexpr) =
    trivCond (exprArith g p lexpr < exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpGEq lexpr rexpr) =
    trivCond (exprArith g p lexpr ≥ exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpLEq lexpr rexpr) =
```

```
    trivCond (exprArith g p lexpr ≤ exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpEq lexpr rexpr) =
    trivCond (exprArith g p lexpr = exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpNEq lexpr rexpr) =
    trivCond (exprArith g p lexpr ≠ exprArith g p rexpr)

| exprArith g p (ExprBinOp BinOpAnd lexpr rexpr) =
    trivCond (exprArith g p lexpr ≠ 0 ∧ exprArith g p rexpr ≠ 0)

| exprArith g p (ExprBinOp BinOpOr lexpr rexpr) =
    trivCond (exprArith g p lexpr ≠ 0 ∨ exprArith g p rexpr ≠ 0)

| exprArith g p (ExprCond cexpr texpr fexpr) =
    (if exprArith g p cexpr ≠ 0 then exprArith g p texpr
     else exprArith g p fexpr)

| exprArith g p (ExprPoll (ChanRef (VarRef gl name None)) rs srt) =
    trivCond (withChannel' gl name None (
      λ- c. pollCheck g p c rs srt) g p)

| exprArith g p (ExprPoll (ChanRef (VarRef gl name (Some idx))) rs srt) =
    trivCond (withChannel' gl name (Some (exprArith g p idx)) (
      λ- c. pollCheck g p c rs srt) g p)

| pollCheck g p InvChannel - - =
    abort STR ''Channel already closed / invalid.'' (λ-. False)
| pollCheck g p (HSChannel -) - - = False
| pollCheck g p (Channel - - q) rs srt = (
    if q = [] then False
    else if ¬ srt then recvArgsCheck g p rs (hd q)
    else List.find (recvArgsCheck g p rs) q ≠ None)

| recvArgsCheck - - [] [] = True
| recvArgsCheck - - - [] =
    abort STR ''Length mismatch on receiving.'' (λ-. False)
| recvArgsCheck - - [] - =
    abort STR ''Length mismatch on receiving.'' (λ-. False)
| recvArgsCheck g p (r#rs) (v#vs) = ((
     case r of
       RecvArgConst c ⇒ c = v
     | RecvArgMConst c - ⇒ c = v
     | RecvArgVar var ⇒ True
     | RecvArgEval e ⇒ exprArith g p e = v ) ∧ recvArgsCheck g p rs vs)
```

getVar' etc. do operate on name, index, ... directly. Lift them to use *VarRef* instead.

fun *liftVar* **where**

```
  liftVar f (VarRef gl v idx) argm g p =
     f gl v (map-option (exprArith g p) idx) argm g p

definition getVar v = liftVar (λgl v idx arg. getVar' gl v idx) v ()
definition setVar = liftVar setVar'
definition withVar = liftVar withVar'

primrec withChannel
  where withChannel (ChanRef v) = liftVar withChannel' v

lemma setVar-gState-progress-rel:
  assumes gState-inv prog g
  shows (g, fst (setVar v val g p)) ∈ gState-progress-rel prog
unfolding setVar-def
by (cases v) (simp add: setVar'-gState-progress-rel[OF assms])

lemmas setVar-gState-inv =
  setVar-gState-progress-rel[THEN gState-progress-rel-gState-invI2]

lemma setVar-pState-inv:
  assumes pState-inv prog p
  shows pState-inv prog (snd (setVar v val g p))
unfolding setVar-def
by (cases v) (auto simp add: setVar'-pState-inv assms)

lemma setVar-cl-inv:
  assumes cl-inv (g,p)
  shows cl-inv (setVar v val g p)
unfolding setVar-def
by (cases v) (auto simp add: setVar'-cl-inv assms)
```

5.4 Variable declaration

```
lemma channel-inv-code [code]:
  channel-inv (Channel cap ts q)
  ⟷ cap ≤ max-array-size
    ∧ 0 ≤ cap
    ∧ for-all varType-inv ts
    ∧ length ts ≤ max-array-size
    ∧ length q ≤ max-array-size
    ∧ for-all (λx. length x = length ts
                 ∧ for-all (λy. y ≥ min-var-value
                              ∧ y ≤ max-var-value) x) q
  channel-inv (HSChannel ts)
  ⟷ for-all varType-inv ts ∧ length ts ≤ max-array-size
  by (auto simp add: for-all-def) force+

primrec toVariable
  :: 'a gState-scheme ⇒ pState ⇒ varDecl ⇒ String.literal * variable * channels
```

```
where
  toVariable g p (VarDeclNum lb hb name siz init) = (
    let type = VTBounded lb hb in
    if ¬ varType-inv type then abortv STR ''Invalid var def (varType-inv failed): '' name
                                  (λ-. (name, Var VTChan 0, []))
    else
      let
        init = checkVarValue type (case init of
                                     None ⇒ 0
                                   | Some e ⇒ exprArith g p e);
        v = (case siz of
               None ⇒ Var type init
             | Some s ⇒ if nat-of-integer s ≤ max-array-size
                        then VArray type (nat-of-integer s)
                                    (IArray.tabulate (s, λ-. init))
                        else abortv STR ''Invalid var def (array too large): '' name
                                    (λ-. Var VTChan 0))
      in
        (name, v, []))

| toVariable g p (VarDeclChan name siz types) = (
    let
      size = (case siz of None ⇒ 1 | Some s ⇒ nat-of-integer s);
      chans = (case types of
                 None ⇒ []
               | Some (cap, tys) ⇒
                   let C = (if cap = 0 then HSChannel tys
                            else Channel cap tys []) in
                   if ¬ channel-inv C
                   then abortv STR ''Invalid var def (channel-inv failed): ''
                               name (λ-. [])
                   else replicate size C);
      cidx = (case types of
                None ⇒ 0
              | Some - ⇒ integer-of-nat (length (channels g)));
      v = (case siz of
             None ⇒ Var VTChan cidx
           | Some s ⇒ if nat-of-integer s ≤ max-array-size
                      then VArray VTChan (nat-of-integer s)
                                  (IArray.tabulate (s,
                                     λi. if cidx = 0 then 0
                                         else i + cidx))
                      else abortv STR ''Invalid var def (array too large): ''
                                  name (λ-. Var VTChan 0))
    in
      (name, v, chans))

lemma toVariable-variable-inv:
```

```
  assumes gState-inv prog g
  shows variable-inv (fst (snd (toVariable g p v)))
using assms
apply (cases v)
  apply (auto intro!: checkVarValue-Var
              simp del: variable-inv.simps checkVarValue.simps varType-inv.simps
              split: if-splits option.splits)
    apply (auto intro!: mod-integer-ge mod-integer-le simp add: min-var-value-def)
   apply (simp-all add: assms gState-inv-def
         max-channels-def max-var-value-def min-var-value-def max-array-size-def)
   including integer.lifting
   apply (transfer', simp)+
done

lemma toVariable-channels-inv:
  shows ∀ x ∈ set (snd (snd (toVariable g p v))). channel-inv x
by (cases v) auto

lemma toVariable-channels-inv':
  shows toVariable g p v = (a,b,c) ⟹ ∀ x ∈ set c. channel-inv x
using toVariable-channels-inv
by (metis snd-conv)

lemma toVariable-variable-inv':
  shows gState-inv prog g ⟹ toVariable g p v = (a,b,c) ⟹ variable-inv b
by (metis snd-conv fst-conv toVariable-variable-inv)

definition mkChannels
  :: 'a gState-scheme ⇒ pState ⇒ channels ⇒ 'a gState-scheme * pState
 where
  mkChannels g p cs = (
     if cs = [] then (g,p) else
     let l = length (channels g) in
     if l + length cs > max-channels
     then abort STR ''Too much channels'' (λ-.  (g,p))
     else let
            cs_p = map integer-of-nat [l..<l + length cs];
            g' = g(|channels := channels g @ cs|);
            p' = p(|pState.channels := pState.channels p @ cs_p|)
          in
            (g', p'))

lemma mkChannels-gState-progress-rel:
  gState-inv prog g
   ⟹ set cs ⊆ Collect channel-inv
   ⟹ (g, fst (mkChannels g p cs)) ∈ gState-progress-rel prog
unfolding mkChannels-def
by (intro gState-progress-relI)
   (auto simp add: gState-inv-def gState.defs cl-inv-def)
```

lemmas *mkChannels-gState-inv* =
mkChannels-gState-progress-rel[*THEN gState-progress-rel-gState-invI2*]

lemma *mkChannels-pState-inv*:
pState-inv prog p
⟹ *cl-inv* (*g*,*p*)
⟹ *pState-inv prog* (*snd* (*mkChannels g p cs*))
unfolding *mkChannels-def*
including *integer.lifting*
apply (*auto simp add*: *pState-inv-def pState.defs gState-inv-def dest*!: *cl-inv-lengthD*)
apply (*transfer'*, *simp*)+
done

lemma *mkChannels-cl-inv*:
cl-inv (*g*,*p*) ⟹ *cl-inv* (*mkChannels g p cs*)
unfolding *mkChannels-def*
by (*auto simp add*: *pState.defs dest*: *cl-inv-lengthD intro*!: *cl-invI*)

definition *mkVarChannel*
:: *varDecl*
⇒ ((*var-dict* ⇒ *var-dict*) ⇒ *'a gState-scheme* ∗ *pState*
⇒ *'a gState-scheme* ∗ *pState*)
⇒ *'a gState-scheme* ⇒ *pState*
⇒ *'a gState-scheme* ∗ *pState*
where
mkVarChannel v upd g p = (
let
(*k*,*v*,*cs*) = *toVariable g p v*;
(*g'*,*p'*) = *upd* (*lm.update k v*) (*g*,*p*)
in
mkChannels g' p' cs)

lemma *mkVarChannel-gState-inv*:
assumes *gState-inv prog g*
and ⋀*k v' cs*. *toVariable g p v* = (*k*,*v'*,*cs*)
⟹ *gState-inv prog* (*fst* (*upd* (*lm.update k v'*) (*g*,*p*)))
shows *gState-inv prog* (*fst* (*mkVarChannel v upd g p*))
using *assms* **unfolding** *mkVarChannel-def*
by (*force split*: *varDecl.split prod.split*
intro!: *mkChannels-gState-inv*
dest: *toVariable-channels-inv'*)

lemma *mkVarChannel-gState-progress-rel*:
assumes *gState-inv prog g*
and ⋀*k v' cs*. *toVariable g p v* = (*k*,*v'*,*cs*)
⟹ (*g*, *fst* (*upd* (*lm.update k v'*) (*g*,*p*))) ∈ *gState-progress-rel prog*
shows (*g*, *fst* (*mkVarChannel v upd g p*)) ∈ *gState-progress-rel prog*
proof −

```
  obtain k v' cs where 1: toVariable g p v = (k,v',cs) by (metis prod.exhaust)
 obtain g' p' where 2: (g',p') = upd (lm.update k v') (g,p) by (metis prod.exhaust)
  with 1 assms have *: (g, g') ∈ gState-progress-rel prog by (metis fst-conv)

  also from 1 2 have (g', fst (mkChannels g' p' cs)) ∈ gState-progress-rel prog
   by (force intro!: mkChannels-gState-progress-rel gState-progress-rel-gState-invI2[OF
*]
            dest: toVariable-channels-inv')
  finally have (g, fst (mkChannels g' p' cs)) ∈ gState-progress-rel prog .
  thus ?thesis using 1 2 by (auto simp add: mkVarChannel-def split: prod.split)
qed

lemma mkVarChannel-pState-inv:
  assumes pState-inv prog p
  and cl-inv (g,p)
  and ⋀k v' cs. toVariable g p v = (k,v',cs)
                ⟹ cl-inv (upd (lm.update k v') (g,p))
  and ⋀k v' cs. toVariable g p v = (k,v',cs)
                ⟹ pState-inv prog (snd (upd (lm.update k v') (g,p)))
  shows pState-inv prog (snd (mkVarChannel v upd g p))
using assms unfolding mkVarChannel-def
by (force split: varDecl.split prod.split
         intro!: mkChannels-pState-inv)

lemma mkVarChannel-cl-inv:
  assumes cl-inv (g,p)
  and ⋀k v' cs. toVariable g p v = (k,v',cs)
                ⟹ cl-inv (upd (lm.update k v') (g,p))
  shows cl-inv (mkVarChannel v upd g p)
using assms unfolding mkVarChannel-def
by (force split: varDecl.split prod.splits
         intro!: mkChannels-cl-inv)

definition mkVarChannelProc
  :: procVarDecl ⇒ 'a gState-scheme ⇒ pState ⇒ 'a gState-scheme * pState
where
  mkVarChannelProc v g p = (
    let
      v' = case v of
            ProcVarDeclNum lb hb name siz init ⇒
              VarDeclNum lb hb name siz init
          | ProcVarDeclChan name siz ⇒
              VarDeclChan name siz None;
      (k,v,cs) = toVariable g p v'
    in
      mkVarChannel v' (apsnd ∘ pState.vars-update) g p)

lemma mkVarChannelProc-gState-progress-rel:
  assumes gState-inv prog g
```

shows (*g*, *fst* (*mkVarChannelProc v g p*)) ∈ *gState-progress-rel prog*
unfolding *mkVarChannelProc-def*
using *assms*
by (*auto intro*!: *mkVarChannel-gState-progress-rel*)

lemmas *mkVarChannelProc-gState-inv* =
mkVarChannelProc-gState-progress-rel[*THEN gState-progress-rel-gState-invI2*]

lemma *toVariable-name*:
toVariable g p (*VarDeclNum lb hb name sz init*) = (*x*,*a*,*b*) ⟹ *x* = *name*
toVariable g p (*VarDeclChan name sz t*) = (*x*, *a*, *b*) ⟹ *x* = *name*
by (*auto split*: *if-splits*)

declare *toVariable.simps*[*simp del*]

lemma *statesDecls-process-names*:
assumes *v* ∈ *statesDecls* (*states prog* !! (*pState.idx p*))
shows *procVarDeclName v* ∈ *process-names* (*states prog* !! (*pState.idx p*))
(*processes prog* !! (*pState.idx p*))
using *assms*
by (*cases processes prog* !! (*pState.idx p*)) (*auto simp add*: *statesNames-def*)

lemma *mkVarChannelProc-pState-inv*:
assumes *pState-inv prog p*
and *gState-inv prog g*
and *cl-inv* (*g*, *p*)
and *decl*: *v* ∈ *statesDecls* (*states prog* !! (*pState.idx p*))
shows *pState-inv prog* (*snd* (*mkVarChannelProc v g p*))
unfolding *mkVarChannelProc-def*
using *assms statesDecls-process-names*[*OF decl*]
by (*auto intro*!: *mkVarChannel-pState-inv*)
(*auto dest*: *toVariable-name*
split: *procVarDecl.splits*
intro: *toVariable-variable-inv′ vardict-inv-updateI*
simp add: *pState-inv-def*)

lemma *mkVarChannelProc-cl-inv*:
assumes *cl-inv* (*g*,*p*)
shows *cl-inv* (*mkVarChannelProc v g p*)
unfolding *mkVarChannelProc-def* **using** *assms*
by (*auto intro*!: *mkVarChannel-cl-inv*)

5.5 Folding

Fold over lists (and lists of lists) of *step*/*stmnt*. The folding functions are doing a bit more than that, e.g. ensuring the offset into the program array is correct.

definition *step-fold′* **where**
step-fold′ g steps (*lbls* :: *labels*) *pri pos*

```
        (nxt :: edgeIndex) (onxt :: edgeIndex option) iB =
    foldr (λstep (pos, nxt, lbls, es).
            let (e,enxt,lbls) = g step (lbls, pri, pos, nxt, onxt, iB)
            in (pos + length e, enxt, lbls, es@e)
    ) steps (pos, nxt, lbls, [])

definition step-fold where
  step-fold g steps lbls pri pos nxt onxt iB = (
        let (-,nxt,lbls,s) = step-fold' g steps lbls pri pos nxt onxt iB
         in (s,nxt,lbls))

lemma step-fold'-cong:
  assumes lbls = lbls'
  and pri = pri'
  and pos = pos'
  and steps = steps'
  and nxt = nxt'
  and onxt = onxt'
  and iB = iB'
  and ⋀x d. x ∈ set steps ⟹ g x d = g' x d
  shows step-fold' g steps lbls pri pos nxt onxt iB =
        step-fold' g' steps' lbls' pri' pos' nxt' onxt' iB'
unfolding step-fold'-def
by (auto intro: foldr-cong simp add: assms)

lemma step-fold-cong[fundef-cong]:
  assumes lbls = lbls'
  and pri = pri'
  and pos = pos'
  and steps = steps'
  and nxt = nxt'
  and onxt = onxt'
  and iB = iB'
  and ⋀x d. x ∈ set steps ⟹ g x d = g' x d
  shows step-fold g steps lbls pri pos nxt onxt iB =
        step-fold g' steps' lbls' pri' pos' nxt' onxt' iB'
unfolding step-fold-def
by (auto simp: assms cong: step-fold'-cong)

fun step-foldL-step where
  step-foldL-step - - - [] (pos, nxt, lbls, es, is) = (pos, nxt, lbls, es, is)
| step-foldL-step g pri onxt (s#steps) (pos, nxt, lbls, es, is) = (
    let (pos', nxt', lbls', ss') = step-fold' g steps lbls pri pos nxt onxt False in
    let (s', nxt'', lbls'') = g s (lbls',pri,pos',nxt',onxt,True) in
    let rs = butlast s'; s'' = last s' in
    (pos' + length rs, nxt, lbls'', es@ss'@rs, s''#is))

definition step-foldL where
  step-foldL g stepss lbls pri pos nxt onxt =
```

foldr (*step-foldL-step g pri onxt*) *stepss* (*pos*,*nxt*,*lbls*,[],[])

lemma *step-foldL-step-cong*:
assumes $pri = pri'$
and $onxt = onxt'$
and $s = s'$
and $d = d'$
and $\bigwedge x\ d.\ x \in set\ s \Longrightarrow g\ x\ d = g'\ x\ d$
shows *step-foldL-step g pri onxt s d* = *step-foldL-step g' pri' onxt' s' d'*
using *assms*
by (*cases d'*, *cases s'*) (*simp-all cong*: *step-fold'-cong*)

lemma *step-foldL-cong*[*fundef-cong*]:
assumes $lbls = lbls'$
and $pri = pri'$
and $pos = pos'$
and $stepss = stepss'$
and $nxt = nxt'$
and $onxt = onxt'$
and $\bigwedge x\ x'\ d.\ x \in set\ stepss \Longrightarrow x' \in set\ x \Longrightarrow g\ x'\ d = g'\ x'\ d$
shows *step-foldL g stepss lbls pri pos nxt onxt* =
step-foldL g' stepss' lbls' pri' pos' nxt' onxt'
unfolding *step-foldL-def*
using *assms*
apply (*cases stepss'*)
apply *simp*
apply (*force intro*!: *foldr-cong step-foldL-step-cong*)
done

5.6 Starting processes

definition *modProcArg*
:: (*procArg* ∗ *integer*) ⇒ *String.literal* ∗ *variable*
where
modProcArg x = (
case x of
(*ProcArg ty name*, *val*) ⇒ *if varType-inv ty*
then let init = *checkVarValue ty val*
in (*name*, *Var ty init*)
else abortv STR ''Invalid proc arg (*varType-inv failed*)''
name (λ-. (*name*, *Var VTChan 0*)))

definition *emptyProc* :: *pState*
— The empty process.
where
emptyProc = (⦇*pid* = *0*, *vars* = *lm.empty* (), *pc* = *0*, *channels* = [], *idx* = *0* ⦈)

lemma *vardict-inv-empty*:
vardict-inv ss proc (*lm.empty*())

```
unfolding vardict-inv-def
by (simp add: lm.correct)

lemma emptyProc-cl-inv[simp]:
  cl-inv (g, emptyProc)
by (simp add: cl-inv-def emptyProc-def)

lemma emptyProc-pState-inv:
  assumes program-inv prog
  shows pState-inv prog emptyProc
proof −
  from assms have IArray.length (states prog !! 0) > 0
    by (intro program-inv-length-states) (auto simp add: program-inv-def)
  with assms show ?thesis
    unfolding pState-inv-def program-inv-def emptyProc-def
    by (auto simp add: vardict-inv-empty)
qed

fun mkProc
  :: 'a gState-scheme ⇒ pState
    ⇒ String.literal ⇒ expr list ⇒ process ⇒ nat
    ⇒ 'a gState-scheme ∗ pState
where
  mkProc g p name args (sidx, start, argDecls, decls) pidN = (
    let start = case start of
                  Index x ⇒ x
                | - ⇒ abortv STR ''Process start is not index: '' name (λ-. 0)
    in
     — sanity check
     if length args ≠ length argDecls
     then abortv STR ''Signature mismatch: '' name (λ-. (g,emptyProc))
     else
       let
         — evaluate args (in the context of the calling process)
         eArgs = map (exprArith g p) args;

         — replace the init part of argDecls
         argVars = map modProcArg (zip argDecls eArgs);

         — add -pid to vars
         pidI = integer-of-nat pidN;
         argVars = (STR ''-pid'', Var (VTBounded 0 pidI) pidI)#argVars;
         argVars = lm.to-map argVars;

         — our new process
        p = (| pid = pidN, vars = argVars, pc = start, channels = [], idx = sidx |)
       in
         — apply the declarations
         foldl (λ(g,p) d. mkVarChannel d (apsnd ∘ pState.vars-update) g p)
```

```
            (g,p)
            decls)

lemma mkProc-gState-progress-rel:
  assumes gState-inv prog g
  shows (g, fst (mkProc g p name args (processes prog !! sidx) pidN)) ∈
        gState-progress-rel prog
proof −
  obtain sidx′ start argDecls decls  where
    p: processes prog !! sidx = (sidx′, start, argDecls, decls)
    by (metis prod.exhaust)

  from assms have
    ⋀p. (g, fst (foldl (λ(g,p) d. mkVarChannel d (apsnd ∘ pState.vars-update) g p)

                       (g,p) decls))
        ∈ gState-progress-rel prog
  proof (induction decls arbitrary: g p)
    case (Cons d decls)
    obtain g′ p′ where
      new: (g′,p′) = (mkVarChannel d (apsnd ∘ pState.vars-update) g p)
      by (metis prod.exhaust)
    hence g′ = fst ... by (metis fst-conv)
    with Cons.prems have g-g′: (g,g′) ∈ gState-progress-rel prog
      by (auto intro: mkVarChannel-gState-progress-rel)
    also note Cons.IH[OF g-g′[THEN gState-progress-rel-gState-invI2], of p′]
    finally show ?case by (auto simp add: o-def new)
  qed simp
  thus ?thesis using assms p by auto
qed

lemmas mkProc-gState-inv =
  mkProc-gState-progress-rel[THEN gState-progress-rel-gState-invI2]

lemma mkProc-pState-inv:
  assumes program-inv prog
  and gState-inv prog g
  and pidN ≤ max-procs and pidN > 0
  and sidx < IArray.length (processes prog)
  and fst (processes prog !! sidx) = sidx
 shows pState-inv prog (snd (mkProc g p name args (processes prog !! sidx) pidN))
proof −
  obtain sidx′ start argDecls decls  where
    processes prog !! sidx = (sidx′, start, argDecls, decls)
    by (metis prod.exhaust)
  with assms have
    p-def: processes prog !! sidx = (sidx, start, argDecls, decls)
          IArray.list-of (processes prog) ! sidx = (sidx, start, argDecls, decls)
    by simp-all
```

with *assms* **have** (*sidx*,*start*,*argDecls*,*decls*) ∈ *set* (*IArray.list-of* (*processes prog*))

by (*force dest*: *nth-mem*)

with *assms* **obtain** *s* **where** *s*: *start* = *Index s s* < *IArray.length* (*states prog* !! *sidx*)
unfolding *program-inv-def* **by** *auto*

hence *P-inv*: *pState-inv prog* (|
pid = *pidN*,
vars = *lm.to-map*
((*STR* ''*-pid*'', *Var* (*VTBounded 0* (*integer-of-nat pidN*))
(*integer-of-nat pidN*))
map modProcArg (*zip argDecls* (*map* (*exprArith g p*) *args*))),
pc = *s*, *channels* = [], *idx* = *sidx*|)
unfolding *pState-inv-def*
using *assms*[*unfolded program-inv-def*]
including *integer.lifting*
apply (*simp add*: *p-def*)
apply (*intro lm-to-map-vardict-inv*)
apply *auto*
apply (*simp add*: *max-procs-def max-var-value-def*)
apply *transfer'*
apply *simp*
apply *transfer'*
apply *simp*
apply (*simp add*: *min-var-value-def*)
apply *transfer'*
apply *simp*
apply (*simp add*: *max-var-value-def max-procs-def*)
apply *transfer'*
apply *simp*
apply (*drule set-zip-leftD*)
apply (*force simp add*: *modProcArg-def*
split: *procArg.splits if-splits*
intro!: *image-eqI*)
apply (*clarsimp simp add*: *modProcArg-def*
split: *procArg.splits if-splits*
simp del: *variable-inv.simps*)
apply (*intro checkVarValue-Var*)
apply *assumption*
done

from *p-def* **have**
varDeclName ‘ *set decls* ⊆
process-names (*states prog* !! *sidx*) (*processes prog* !! *sidx*)
by *auto*
with ‹*gState-inv prog g*› **have**
F-inv: ⋀*p*. ⟦ *pState-inv prog p*; *sidx* = *pState.idx p*; *cl-inv* (*g*,*p*) ⟧

```
            ⟹ pState-inv prog
             (snd (foldl (λ(g,p) d. mkVarChannel d (apsnd ∘ pState.vars-update)
g p)
                                 (g,p) decls))
  proof (induction decls arbitrary: g p)
    case (Cons d ds) hence
      decl: varDeclName d ∈ process-names (states prog !! pState.idx p)
                                          (processes prog !! pState.idx p)
      by simp

    obtain g' p' where
      new: (g',p') = (mkVarChannel d (apsnd ∘ pState.vars-update) g p)
      by (metis prod.exhaust)
    hence p': p' = snd ... and g': g' = fst ...
      by (metis snd-conv fst-conv)+
    with Cons.prems have pState-inv prog p'
      apply (auto intro!: mkVarChannel-pState-inv)
      apply (simp add: pState-inv-def)
      apply (intro vardict-inv-updateI)
        apply simp
       apply (cases d)
        apply (force dest!: toVariable-name)
       apply (force dest!: toVariable-name)
      apply (intro toVariable-variable-inv')
       apply assumption+
      done
    moreover
    from p' Cons.prems have pState.idx p' = sidx
      by (auto simp add: mkVarChannel-def mkChannels-def split: prod.split)
    moreover
    from new Cons.prems have cl-inv (g',p')
      by (auto intro!: mkVarChannel-cl-inv)
    moreover
    from g' Cons.prems have gState-inv prog g'
      by (auto intro!: mkVarChannel-gState-inv)
    moreover
    from Cons.prems have
      varDeclName ` set ds ⊆
        process-names (states prog !! sidx) (processes prog !! sidx)
      by simp
    ultimately
    have
      pState-inv prog
        (snd (foldl (λ(g,p) d. mkVarChannel d (apsnd ∘ pState.vars-update) g p)
                    (g',p') ds))
      using Cons.IH[of p' g'] by (simp add: o-def)
    with new show ?case by (simp add: o-def)
  qed simp
```

```
  show ?thesis
    by (auto simp add: p-def s cl-inv-def
          intro: F-inv[OF P-inv])
       (blast intro: emptyProc-pState-inv assms)
qed

lemma mkProc-cl-inv:
  assumes cl-inv (g,p)
  shows cl-inv (mkProc g p name args (processes prog !! sidx) pidN)
proof −
  note IArray.sub-def [simp del]
  obtain sidx' start argDecls decls
    where [simp]: processes prog !! sidx = (sidx', start, argDecls, decls)
    by (metis prod.exhaust)

  have
    P-inv: ⋀s v. cl-inv (g, (|pid = pidN, vars = v, pc = s, channels = [], idx =
sidx' |))
    by (simp add: cl-inv-def)

  have
    ⋀p. cl-inv(g,p) ⟹
        cl-inv (foldl (λ(g,p) d. mkVarChannel d (apsnd ∘ pState.vars-update) g p)
                  (g,p) decls)
  proof (induction decls arbitrary: g p)
    case (Cons d ds)
    obtain g' p' where
      new: (g',p') = (mkVarChannel d (apsnd ∘ pState.vars-update) g p)
      by (metis prod.exhaust)
    with Cons.prems have cl-inv (g',p')
      by (auto intro!: mkVarChannel-cl-inv)

    from Cons.IH[OF this] new show ?case by (simp add: o-def)
  qed simp

  from this[OF P-inv] show ?thesis by auto
qed

declare mkProc.simps[simp del]

definition runProc
  :: String.literal ⇒ expr list ⇒ program
     ⇒ 'a gState-scheme ⇒ pState
     ⇒ 'a gState-scheme ∗ pState
where
  runProc name args prog g p = (
    if length (procs g) ≥ max-procs
    then abort STR ''Too many processes'' (λ-. (g,p))
    else let pid = length (procs g) + 1 in
```

```
      case lm.lookup name (proc-data prog) of
        None ⇒ abortv STR ''No such process: '' name
                  (λ-. (g,p))
      | Some proc-idx ⇒
          let (g', proc) = mkProc g p name args (processes prog !! proc-idx) pid
          in (g'(|procs := procs g @ [proc]|), p))

lemma runProc-gState-progress-rel:
  assumes program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  and cl-inv (g,p)
  shows (g, fst (runProc name args prog g p)) ∈ gState-progress-rel prog
proof (cases length (procs g) < max-procs)
  note IArray.sub-def [simp del]

  case True thus ?thesis
  proof (cases lm.lookup name (proc-data prog))
    case (Some proc-idx)
    hence *: proc-idx < IArray.length (processes prog)
          fst (processes prog !! proc-idx) = proc-idx
      using assms
      by (simp-all add: lm.correct program-inv-def)

    obtain g' p' where
     new: (g',p') = mkProc g p name args (processes prog !! proc-idx) (length (procs
g) + 1)
      by (metis prod.exhaust)
    hence g': g' = fst ... and p': p' = snd ...
      by (metis snd-conv fst-conv)+
    from assms g' have (g, g') ∈ gState-progress-rel prog
      by (auto intro!: mkProc-gState-progress-rel)

    moreover
    from * assms True p' have pState-inv prog p'
      by (auto intro!: mkProc-pState-inv)
    moreover
    from assms new have cl-inv (g',p')
      by (auto intro!: mkProc-cl-inv)
    ultimately show ?thesis
      using True Some new assms
      unfolding runProc-def gState-progress-rel-def
      by (clarsimp split: prod.split)
        (auto simp add: gState-inv-def cl-inv-def)
  next
    case None with assms show ?thesis by (auto simp add: runProc-def)
  qed
next
  case False with assms show ?thesis by (auto simp add: runProc-def)
```

qed

lemmas *runProc-gState-inv* =
runProc-gState-progress-rel[*THEN gState-progress-rel-gState-invI2*]

lemma *runProc-pState-id*:
snd (*runProc name args prog g p*) = *p*
unfolding *runProc-def*
by (*auto split*: *if-splits split*: *option.split prod.split*)

lemma *runProc-pState-inv*:
assumes *pState-inv prog p*
shows *pState-inv prog* (*snd* (*runProc name args prog g p*))
by (*simp add*: *assms runProc-pState-id*)

lemma *runProc-cl-inv*:
assumes *program-inv prog*
and *gState-inv prog g*
and *pState-inv prog p*
and *cl-inv* (*g*,*p*)
shows *cl-inv* (*runProc name args prog g p*)
proof −
obtain *g′ p′* **where** ∗: *runProc name args prog g p* = (*g′*,*p′*)
by (*metis prod.exhaust*)
with *runProc-gState-progress-rel*[*OF assms*, *of name args*] **have**
length (*channels g*) ≤ *length* (*channels g′*)
by (*simp add*: *gState-progress-rel-def*)
moreover from ∗ *runProc-pState-id* **have** *p′* = *p* **by** (*metis snd-conv*)
ultimately show *?thesis* **by** (*metis* ‹*cl-inv* (*g*,*p*)› ∗ *cl-inv-trans*)
qed

5.7 AST to edges

type-synonym *ast* = *AST.module list*

In this section, the AST is translated into the transition system.

Handling atomic blocks is non-trivial. Therefore, we do this in an extra pass: *lp* and *hp* are the positions of the start and the end of the atomic block. Every edge pointing into this range is therefore marked as *Atomic*. If they are pointing somewhere else, they are set to *InAtomic*, meaning: they start *in* an atomic block, but leave it afterwards.

definition *atomize* :: *nat* ⇒ *nat* ⇒ *edge list* ⇒ *edge list* **where**
atomize lp hp es = *fold* (λ*e es*.
let e′ = *case target e of*
LabelJump - None ⇒
— Labels are checked again later on, when they
— are going to be resolved. Hence it is safe to say
— *atomic* here, especially as the later algorithm

```
                — relies on targets in atomic blocks to be marked as such.
                e(| atomic := InAtomic |)
            | LabelJump - (Some via) ⇒
                if lp ≤ via ∧ hp ≥ via then e(| atomic := Atomic |)
                else e(| atomic := InAtomic |)
            | Index p' ⇒
                if lp ≤ p' ∧ hp ≥ p' then e(| atomic := Atomic |)
                else e(| atomic := InAtomic |)
    in e'#es) es []

fun skip — No-(edge)
where
  skip (lbls, pri, pos, nxt, -) =
  ([[(|cond = ECExpr (ExprConst 1),
      effect = EEId, target = nxt, prio = pri,
      atomic = NonAtomic|)]], Index pos, lbls)
```

The AST is walked backwards. This allows to know the next state directly. Parameters used:

lbls Map of Labels

pri Current priority

pos Current position in the array

nxt Next state

onxt Previous 'next state' (where to jump after a 'do')

inBlock Needed for certain constructs to calculate the layout of the array

```
fun stepToState
  :: step
     ⇒ (labels * integer * nat * edgeIndex * edgeIndex option * bool)
     ⇒ edge list list * edgeIndex * labels
and stmntToState
  :: stmnt
     ⇒ (labels * integer * nat * edgeIndex * edgeIndex option * bool)
     ⇒ edge list list * edgeIndex * labels
where
  stepToState (StepStmnt s None) data = stmntToState s data
| stepToState (StepStmnt s (Some u)) (lbls, pri, pos, nxt, onxt, -) = (
    let
      — the unless part
      (ues,-,lbls') = stmntToState u (lbls, pri, pos, nxt, onxt, True);
      u = last ues; ues = butlast ues;
      pos' = pos + length ues;

      — find minimal current priority
```

```
      pri = min-prio u pri;

      — the guarded part –
      — priority is decreased, because there is now a new unless part with
      — higher prio
      (ses,spos,lbls'') = stmntToState s (lbls', pri − 1, pos', nxt, onxt, False);

      — add an edge to the unless part for each generated state
      ses = map (List.append u) ses
    in
      (ues@ses,spos,lbls''))

| stepToState (StepDecl decls) (lbls, pri, pos, nxt, onxt, -) = (
    let edgeF = λd (lbls,pri,pos,nxt,-).
      ([[(|cond = ECTrue, effect = EEDecl d, target = nxt,
          prio = pri, atomic = NonAtomic|)]], Index pos, lbls)
    in
      step-fold edgeF decls lbls pri pos nxt onxt False)

| stepToState StepSkip (lbls,-,-,nxt,-) = ([],nxt,lbls)

| stmntToState (StmntAtomic steps) (lbls, pri, pos, nxt, onxt, inBlock) = (
    let (es,pos',lbls') = step-fold stepToState steps lbls pri pos nxt onxt inBlock in
    let es' = map (atomize pos (pos + length es)) es in
    (es', pos', lbls'))

| stmntToState (StmntLabeled l s) (lbls, pri, pos, d) = (
    let
      (es, pos', lbls) = stmntToState s (lbls, pri, pos, d);

      — We don't resolve goto-chains. If the labeled stmnt returns only a jump,
      — use this goto state.
      lpos = case pos' of Index p ⇒ p | - ⇒ pos;
      lbls' = add-label l lbls lpos
    in
      (es, pos', lbls'))

| stmntToState (StmntDo stepss) (lbls, pri, pos, nxt, onxt, inBlock) = (
    let
      — construct the different branches
      — nxt in those branches points current pos (it is a loop after all)
      — onxt then is the current nxt (needed for break, f.ex.)
      (-,-,lbls,es,is) = step-foldL stepToState stepss lbls pri
                                    (pos+1) (Index pos) (Some nxt);

      — put the branch starting points (is) into the array
      es' = concat is # es
    in
      if inBlock then
```

```
        — inside another DO or IF or UNLESS
        — ⟶ append branches again, so they can be consumed
        (es' @ [concat is], Index pos, lbls)
    else
      (es', Index pos, lbls)
  )

| stmntToState (StmntIf stepss) (lbls, pri, pos, nxt, onxt, -) = (
    let (pos,-,lbls,es,is) = step-foldL stepToState stepss lbls pri pos nxt onxt
    in (es @ [concat is], Index pos, lbls))

| stmntToState (StmntSeq steps) (lbls, pri, pos, nxt, onxt, inBlock) =
    step-fold stepToState steps lbls pri pos nxt onxt inBlock


| stmntToState (StmntAssign v e) (lbls, pri, pos, nxt, -) =
  ([[⦇cond = ECTrue, effect = EEAssign v e, target = nxt, prio = pri,
      atomic = NonAtomic⦈]], Index pos, lbls)

| stmntToState (StmntAssert e) (lbls, pri, pos, nxt, -) =
  ([[⦇cond = ECTrue, effect = EEAssert e, target = nxt, prio = pri,
      atomic = NonAtomic⦈]], Index pos, lbls)

| stmntToState (StmntCond e) (lbls, pri, pos, nxt, -) =
  ([[⦇cond = ECExpr e, effect = EEId, target = nxt, prio = pri,
      atomic = NonAtomic⦈]], Index pos, lbls)

| stmntToState StmntElse (lbls, pri, pos, nxt, -) =
  ([[⦇cond = ECElse, effect = EEId, target = nxt, prio = pri,
      atomic = NonAtomic ⦈]], Index pos, lbls)

| stmntToState StmntBreak (lbls,pri,-,-,Some onxt,-) =
  ([[⦇cond = ECTrue, effect = EEGoto, target = onxt, prio = pri,
      atomic = NonAtomic ⦈]], onxt, lbls)
| stmntToState StmntBreak (-,-,-,-,None,-) =
  abort STR ''Misplaced break'' (λ-. ([],Index 0,lm.empty()))

| stmntToState (StmntRun n args) (lbls, pri, pos, nxt, onxt, -) =
  ([[⦇cond = ECRun n, effect = EERun n args, target = nxt, prio = pri,
      atomic = NonAtomic ⦈]], Index pos,lbls)

| stmntToState (StmntGoTo l) (lbls, pri, pos, -) =
  ([[⦇cond = ECTrue, effect = EEGoto, target = LabelJump l None, prio = pri,
      atomic = NonAtomic ⦈]], LabelJump l (Some pos), lbls)

| stmntToState (StmntSend v e srt) (lbls, pri, pos, nxt, -) =
  ([[⦇cond = ECSend v, effect = EESend v e srt, target = nxt, prio = pri,
      atomic = NonAtomic ⦈]], Index pos, lbls)
```

```
| stmntToState (StmntRecv v r srt rem) (lbls, pri, pos, nxt, -) =
   ([[(|cond = ECRecv v r srt, effect = EERecv v r srt rem, target = nxt, prio =
pri,
       atomic = NonAtomic |)]], Index pos, lbls)

| stmntToState StmntSkip d = skip d
```

5.7.1 Setup

```
definition endState :: edge list where
  — An extra state added to each process marking its end.
  endState = [(| cond = ECFalse, effect = EEEnd, target = Index 0, prio = 0,
                 atomic = NonAtomic|)]

definition resolveLabel :: String.literal ⇒ labels ⇒ nat where
  resolveLabel l lbls = (
    case lm.lookup l lbls of
      None ⇒ abortv STR ''Unresolved label: '' l (λ-. 0)
    | Some pos ⇒ pos)

primrec resolveLabels :: edge list list ⇒ labels ⇒ edge list ⇒ edge list where
  resolveLabels - - [] = []
| resolveLabels edges lbls (e#es) = (
    let check-atomic = λpos. fold (λe a. a ∧ inAtomic e) (edges ! pos) True in
    case target e of
      Index - ⇒ e
    | LabelJump l None ⇒
        let pos = resolveLabel l lbls in
          e(|target := Index pos,
             atomic := if inAtomic e then
                         if check-atomic pos then Atomic
                         else InAtomic
                       else NonAtomic |)
    | LabelJump l (Some via) ⇒
        let pos = resolveLabel l lbls in
          e(|target := Index pos,
             — NB: isAtomic instead of inAtomic, cf atomize()
             atomic := if isAtomic e then
                         if check-atomic pos ∧ check-atomic via then Atomic
                         else InAtomic
                       else atomic e |)
  ) # (resolveLabels edges lbls es)

definition calculatePrios :: edge list list ⇒ (integer * edge list) list where
  calculatePrios ess = map (λes. (min-prio es 0, es)) ess

definition toStates :: step list ⇒ states * edgeIndex * labels where
  toStates steps = (
    let
```

```
    (states,pos,lbls) = step-fold stepToState steps (lm.empty())
                                  0 1 (Index 0) None False;
    pos = (case pos of
             Index - ⇒ pos
           | LabelJump l - ⇒ Index (resolveLabel l lbls));
    states = endState # states;
    states = map (resolveLabels states lbls) states;
    states = calculatePrios states
  in
  case pos of Index s ⇒
      if s < length states then (IArray states, pos, lbls)
       else abort STR ''Start index out of bounds'' (λ-. (IArray states, Index 0,
lbls)))

lemma toStates-inv:
  assumes toStates steps = (ss,start,lbls)
  shows ∃ s. start = Index s ∧ s < IArray.length ss
    and IArray.length ss > 0
  using assms
  unfolding toStates-def calculatePrios-def
  by (auto split: prod.splits edgeIndex.splits if-splits)


primrec toProcess
  :: nat ⇒ proc ⇒ states * nat * String.literal * (labels * process)
 where
  toProcess sidx (ProcType act name args decls steps) = (
    let
      (states, start, lbls) = toStates steps;
      act = (case act of
               None ⇒ 0
             | Some None ⇒ 1
             | Some (Some x) ⇒ nat-of-integer x)
    in
      (states, act, name, lbls, sidx, start, args, decls))
| toProcess sidx (Init decls steps) = (
     let (states, start, lbls) = toStates steps in
     (states, 1, STR '':init:'', lbls, sidx, start, [], decls))

lemma toProcess-sidx:
  toProcess sidx p = (ss,a,n,l,idx,r) ⟹ idx = sidx
by (cases p) (simp-all split: prod.splits)

lemma toProcess-states-nonempty:
  toProcess sidx p = (ss,a,n,l,idx,r) ⟹ IArray.length ss > 0
by (cases p) (force split: prod.splits dest: toStates-inv(2))+

lemma toProcess-start:
  toProcess sidx p = (ss,a,n,l,idx,start,r)
```

$\Longrightarrow \exists\, s.\ start = Index\ s \wedge s < IArray.length\ ss$
by (*cases p*) (*force split*: *prod.splits dest*: *toStates-inv*(*1*))+

lemma *toProcess-startE*:
assumes *toProcess sidx p* = (*ss*,*a*,*n*,*l*,*idx*,*start*,*r*)
obtains *s* **where** *start* = *Index s s* < *IArray.length ss*
using *toProcess-start*[*OF assms*]
by *blast*

The main construction function. Takes an AST and returns an initial state, and the program (= transition system).

definition *setUp* :: *ast* ⇒ *program* × *gState* **where**
setUp ast = (
let
(*decls*, *procs*, -) = *preprocess ast*;
assertVar = *Var* (*VTBounded 0 1*) *0*;

pre-procs = *map* (*case-prod toProcess*) (*List.enumerate 1 procs*);

procs = *IArray* ((*0*, *Index 0*, [], []) # *map* (λ(-,-,-,-,*p*). *p*) *pre-procs*);
labels = *IArray* (*lm.empty*() # *map* (λ(-,-,-,*l*,-). *l*) *pre-procs*);
states = *IArray* (*IArray* [(*0*,[])] # *map* (λ(*s*,-). *s*) *pre-procs*);
names = *IArray* (*STR* ''*invalid*'' # *map* (λ(-,-,*n*,-). *n*) *pre-procs*);

proc-data = *lm.to-map* (*map* (λ(-,-,*n*,-,*idx*,-). (*n*,*idx*)) *pre-procs*);

prog = (‖ *processes* = *procs*, *labels* = *labels*, *states* = *states*,
proc-names = *names*, *proc-data* = *proc-data* ‖);

g = (‖ *vars* = *lm.sng* (*STR* ''*--assert--*'') *assertVar*,
channels = [*InvChannel*], *timeout* = *False*, *procs* = [] ‖);
g′ = *foldl* (λ*g d*.
fst (*mkVarChannel d* (*apfst* ∘ *gState.vars-update*) *g emptyProc*)
) *g decls*;
g′′ = *foldl* (λ*g* (-,*a*,*name*,-).
foldl (λ*g name*.
fst (*runProc name* [] *prog g emptyProc*)
) *g* (*replicate a name*)
) *g*′ *pre-procs*
in
(*prog*, *g*′′))

lemma *setUp-program-inv*′:
program-inv (*fst* (*setUp ast*))
proof (*rule program-invI*, *goal-cases*)
case *1* **show** *?case* **by** (*simp add*: *setUp-def split*: *prod.split*)
next
case *2* **show** *?case* **by** (*simp add*: *setUp-def split*: *prod.split*)

```
next
  case 3 thus ?case
    by (auto simp add: setUp-def o-def
             split: prod.splits
             dest!: toProcess-states-nonempty)
next
  case 4 thus ?case
    unfolding setUp-def
    by (auto simp add: lm.correct o-def in-set-enumerate-eq nth-enumerate-eq
          dest!: subsetD[OF Misc.ran-map-of] toProcess-sidx
          split: prod.splits)

next
  case 5 thus ?case
    apply (auto simp add: setUp-def o-def split: prod.splits)
    apply (frule toProcess-sidx)
    apply (frule toProcess-start)
    apply (auto simp: in-set-enumerate-eq nth-enumerate-eq)
    done
qed

lemma setUp-program-inv:
  assumes setUp ast = (prog,g)
  shows program-inv prog
using assms setUp-program-inv'
by (metis fst-conv)

lemma setUp-gState-inv:
  assumes setUp ast = (prog, g)
  shows gState-inv prog g
proof −
  from assms have p-INV: program-inv prog by (fact setUp-program-inv)

  {
    fix prog :: program
    assume ∗: program-inv prog
    let ?g = (| vars = lm.sng (STR ''--assert--'') (Var (VTBounded 0 1) 0),
                channels = [InvChannel], timeout = False, procs = [] |)

    have g1: gState-inv prog ?g
      by (simp add: gState-inv-def max-channels-def lm-correct max-var-value-def)
    {
      fix g decls
      assume gState-inv prog g
      hence gState-inv prog (foldl (λg d.
                fst (mkVarChannel d (apfst ∘ gState.vars-update) g emptyProc)
               ) g decls)
        apply (rule foldl-rule)
        apply (intro mkVarChannel-gState-inv)
```

```
      apply simp
    apply (frule-tac g=σ in toVariable-variable-inv')
      apply assumption
    apply (auto simp add: gState-inv-def lm.correct)
    done
  }
  note g2 = this[OF g1]

  {
    fix g :: 'a gState-scheme and pre-procs
    assume gState-inv prog g
    hence gState-inv prog (foldl (λg (-,a,name,-).
            foldl (λg name.
                fst (runProc name [] prog g emptyProc)
              ) g (replicate a name)
          ) g pre-procs)
      apply (rule foldl-rule)
      apply (clarsimp split: prod.splits)
      apply (rule foldl-rule)
        apply (auto intro!: runProc-gState-inv emptyProc-pState-inv *)
      done
  }
  note this[OF g2]
 } note g-INV = this

 from assms p-INV show ?thesis
   unfolding setUp-def
   by (auto split: prod.splits intro!: g-INV)
qed
```

5.8 Semantic Engine

After constructing the transition system, we are missing the final part: The successor function on this system. We use SPIN-nomenclature and call it *semantic engine*.

```
definition assertVar ≡ VarRef True (STR ''--assert--'') None
```

5.8.1 Evaluation of Edges

```
fun evalRecvArgs
  :: recvArg list ⇒ integer list ⇒ gState_I ⇒ pState ⇒ gState_I * pState
where
  evalRecvArgs [] [] g l = (g,l)
| evalRecvArgs - [] g l =
    abort STR ''Length mismatch on receiving.'' (λ-. (g,l))
| evalRecvArgs [] - g l =
    abort STR ''Length mismatch on receiving.'' (λ-. (g,l))
| evalRecvArgs (r#rs) (v#vs) g l = (
    let (g,l) =
```

```
      case r of
        RecvArgVar var ⇒ setVar var v g l
      | - ⇒ (g,l)
    in evalRecvArgs rs vs g l)

primrec evalCond
  :: edgeCond ⇒ gState_I ⇒ pState ⇒ bool
where
  evalCond ECTrue - - ⟷ True
| evalCond ECFalse - - ⟷ False
| evalCond (ECExpr e) g l ⟷ exprArith g l e ≠ 0
| evalCond (ECRun -) g l ⟷ length (procs g) < 255
| evalCond ECElse g l ⟷ gState_I.else g
| evalCond (ECSend v) g l ⟷
    withChannel v (λ- c.
      case c of
        Channel cap - q ⇒ integer-of-nat (length q) < cap
      | HSChannel - ⇒ True) g l
| evalCond (ECRecv v rs srt) g l ⟷
    withChannel v (λi c.
      case c of
        HSChannel - ⇒ handshake g ≠ 0 ∧ recvArgsCheck g l rs (hsdata g)
      | - ⇒ pollCheck g l c rs srt) g l

fun evalHandshake
  :: edgeCond ⇒ nat ⇒  gState_I ⇒ pState ⇒ bool
where
  evalHandshake (ECRecv v - -) h g l
    ⟷ h = 0
    ∨ withChannel v (λi c. case c of
                             HSChannel - ⇒ i = h
                           | Channel - - - ⇒ False) g l
| evalHandshake - h - - ⟷ h = 0

primrec evalEffect
  :: edgeEffect ⇒ program ⇒ gState_I ⇒ pState ⇒ gState_I * pState
where
  evalEffect EEEnd - g l = (g,l)
| evalEffect EEId - g l = (g,l)
| evalEffect EEGoto - g l = (g,l)
| evalEffect (EEAssign v e) - g l = setVar v (exprArith g l e) g l
| evalEffect (EEDecl d) - g l = mkVarChannelProc d g l
| evalEffect (EERun name args) prog g l = runProc name args prog g l
| evalEffect (EEAssert e) - g l = (
    if exprArith g l e = 0
    then setVar assertVar 1 g l
    else (g,l))
| evalEffect (EESend v es srt) - g l = withChannel v (λi c.
    let
```

```
      ab = λ-. abort STR ''Length mismatch on sending.'' (λ-. (g,l));
      es = map (exprArith g l) es
    in
      if ¬ for-all (λx. x ≥ min-var-value ∧ x ≤ max-var-value) es
      then abort STR ''Invalid Channel'' (λ-. (g,l))
      else
        case c of
          Channel cap ts q ⇒
            if length ts ≠ length es ∨ ¬ (length q < max-array-size)
            then ab()
            else let
                   q' = if ¬ srt then q@[es]
                        else let
                          q = map lexlist q;
                          q' = insort (lexlist es) q
                        in map unlex q';
                   g = gState.channels-update (λcs.
                         cs[ i := Channel cap ts q' ]) g
                 in (g,l)
        | HSChannel ts ⇒
            if length ts ≠ length es then ab()
            else (g(|hsdata := es, handshake := i|), l)
        | InvChannel ⇒ abort STR ''Trying to send on invalid channel'' (λ-. (g,l))
    ) g l
| evalEffect (EERecv v rs srt rem) - g l = withChannel v (λi c.
    case c of
      Channel cap ts qs ⇒
        if qs = [] then abort STR ''Recv from empty channel'' (λ-. (g,l))
        else
          let
            (q', qs') = if ¬ srt then (hd qs, tl qs)
                        else apfst the (find-remove (recvArgsCheck g l rs) qs);
            (g,l) = evalRecvArgs rs q' g l;
            g = if rem
                then gState.channels-update (λcs. cs[ i := Channel cap ts qs']) g
                else g
                  — messages are not removed – so no need to update anything
          in (g,l)
    | HSChannel - ⇒
          let (g,l) = evalRecvArgs rs (hsdata g) g l in
          let g = g(| handshake := 0, hsdata := [] |)
          in (g,l)
    | InvChannel ⇒ abort STR ''Receiving on invalid channel'' (λ-. (g,l))
  ) g l

lemma statesDecls-effect:
  assumes ef ∈ effect ‘ edgeSet ss
  and ef = EEDecl d
  shows d ∈ statesDecls ss
```

```
proof −
  from assms obtain e where e ∈ edgeSet ss ef = effect e by auto
  thus ?thesis  using assms
    unfolding statesDecls-def
    by (auto simp add: edgeDecls-def
            intro!: bexI[where x = e]
            split: edgeEffect.split)
qed

lemma evalRecvArgs-pState-inv:
  assumes pState-inv prog p
  shows pState-inv prog (snd (evalRecvArgs rargs xs g p))
  using assms
proof (induction rargs xs arbitrary: p g rule: list-induct2')
  case (4 r rs x xs) thus ?case
  proof (cases r)
    case (RecvArgVar v)
    obtain g' p' where  new: setVar v x g p = (g',p') by (metis prod.exhaust)
    hence p' = snd (setVar v x g p) by simp
    with 4 have pState-inv prog p' by (auto intro!: setVar-pState-inv)
    from 4.IH[OF this] RecvArgVar new show ?thesis by simp
  qed simp-all
qed simp-all

lemma evalRecvArgs-pState-inv':
  assumes evalRecvArgs rargs xs g p = (g', p')
  and pState-inv prog p
  shows pState-inv prog p'
using assms evalRecvArgs-pState-inv
by (metis snd-conv)

lemma evalRecvArgs-gState-progress-rel:
  assumes gState-inv prog g
  shows (g, fst (evalRecvArgs rargs xs g p)) ∈ gState-progress-rel prog
  using assms
proof (induction rargs xs arbitrary: p g rule: list-induct2')
  case (4 r rs x xs) thus ?case
  proof (cases r)
    case (RecvArgVar v)
    obtain g' p' where new: setVar v x g p = (g',p') by (metis prod.exhaust)
    hence g' = fst (setVar v x g p) by simp
    with 4 have (g, g') ∈ gState-progress-rel prog
      by (auto intro!: setVar-gState-progress-rel)
    also hence gState-inv prog g'
      by (rule gState-progress-rel-gState-invI2)
    note 4.IH[OF this, of p']
    finally show ?thesis using RecvArgVar new by simp
  qed simp-all
qed simp-all
```

```
lemmas evalRecvArgs-gState-inv =
  evalRecvArgs-gState-progress-rel[THEN gState-progress-rel-gState-invI2]

lemma evalRecvArgs-cl-inv:
  assumes cl-inv (g,p)
  shows cl-inv (evalRecvArgs rargs xs g p)
  using assms
proof (induction rargs xs arbitrary: p g rule: list-induct2')
  case (4 r rs x xs) thus ?case
  proof (cases r)
    case (RecvArgVar v)
    with 4 have cl-inv (setVar v x g p)
      by (auto intro!: setVar-cl-inv)
    with 4.IH RecvArgVar show ?thesis
      by (auto split: prod.splits)
  qed simp-all
qed simp-all

lemma evalEffect-pState-inv:
  assumes pState-inv prog p
  and gState-inv prog g
  and cl-inv (g,p)
  and e ∈ effect ‘ edgeSet (states prog !! pState.idx p)
  shows pState-inv prog (snd (evalEffect e prog g p))
  using assms
proof (cases e)
  case (EEDecl d)
  with assms have d ∈ statesDecls (states prog !! pState.idx p)
    using statesDecls-effect
    by simp
  with assms EEDecl show ?thesis
    by (auto simp del: IArray.sub-def
          intro!: mkVarChannelProc-pState-inv)
next
  case (EESend c es srt)
  then obtain v where ChanRef v = c by (cases c) simp
  with EESend assms show ?thesis
   by (cases v) (auto simp add: withChannel'-def withVar'-def split: channel.split)
next
  case (EERecv c es srt)
  then obtain v where ChanRef v = c by (cases c) simp
  with EERecv assms show ?thesis
    by (cases v)
       (auto simp: withChannel'-def withVar'-def
            split: prod.splits channel.split
            dest: evalRecvArgs-pState-inv')
qed (clarsimp-all intro!: setVar-pState-inv runProc-pState-inv)
```

```
lemma evalEffect-gState-progress-rel:
  assumes program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  and cl-inv (g,p)
  shows (g, fst (evalEffect e prog g p)) ∈ gState-progress-rel prog
using assms
proof (cases e)
  case EEAssert
  with assms show ?thesis
    by (auto intro: setVar-gState-progress-rel)
next
  case EEAssign
  with assms show ?thesis
    by (auto intro: setVar-gState-progress-rel)
next
  case EEDecl
  with assms show ?thesis
    by (auto intro: mkVarChannelProc-gState-progress-rel)
next
  case EERun
  with assms show ?thesis
    by (auto intro: runProc-gState-progress-rel)
next
  case (EESend c es srt)
  then obtain v where v: c = ChanRef v by (metis chanRef.exhaust)
  obtain idx where idx: nat-of-integer (the (getVar v g p)) = idx by blast
  note idx' = idx[symmetric, unfolded getVar-def]
  show ?thesis
  proof (cases idx < length (gState.channels g))
    case True
    note DEF = True EESend v idx' assms

    show ?thesis
    proof (cases gState.channels g ! idx)
      case (Channel cap ts q)
      with True have Channel cap ts q ∈ set (gState.channels g)
        by (metis nth-mem)
      with assms have channel-inv (Channel cap ts q)
        by (auto simp add: gState-inv-def simp del: channel-inv.simps)
      with Channel DEF show ?thesis
        by (cases v)
          (auto simp add: withChannel'-def withVar'-def for-all-def
                split: channel.split
                intro!: gState-progress-rel-channels-update)
    next
      case HSChannel with DEF show ?thesis
        by (cases v)
          (auto simp: withChannel'-def withVar'-def gState-progress-rel-def
```

```
                gState-inv-def
              split: channel.split)
    next
      case InvChannel with DEF show ?thesis
        by (cases v) (auto simp add: withChannel'-def withVar'-def)
    qed
  next
    case False with EESend idx' v assms show ?thesis
      by (cases v) (auto simp add: withChannel'-def withVar'-def)
  qed
next
  case (EERecv c rs srt rem)
  then obtain v where v: c = ChanRef v by (metis chanRef.exhaust)
  obtain idx where idx: nat-of-integer (the (getVar v g p)) = idx by blast
  note idx' = idx[symmetric, unfolded getVar-def]
  show ?thesis
  proof (cases idx < length (gState.channels g))
    case True
    note DEF = True EERecv v idx' assms
    show ?thesis
    proof (cases gState.channels g ! idx)
      note channel-inv.simps[simp del]
      case (Channel cap ts q)
      with True have Channel cap ts q ∈ set (gState.channels g)
        by (metis nth-mem)
      with assms have c-inv: channel-inv (Channel cap ts q)
        by (auto simp add: gState-inv-def simp del: channel-inv.simps)
      moreover obtain res q' where
        apfst the (find-remove (recvArgsCheck g p rs) q) = (res, q')
        by (metis prod.exhaust)
      moreover hence q' = snd (find-remove (recvArgsCheck g p rs) q)
        by (simp add: apfst-def map-prod-def split: prod.splits)
      with find-remove-subset find-remove-length have
        set q' ⊆ set q length q' ≤ length q
        by (metis surjective-pairing)+
      with c-inv have channel-inv (Channel cap ts q')
        by (auto simp add: channel-inv.simps)
      moreover {
        assume q ≠ []
        hence set (tl q) ⊆ set q using tl-subset by auto
        with c-inv have channel-inv (Channel cap ts (tl q))
          by (auto simp add: channel-inv.simps)
      }
      moreover {
        fix res g' p'
        assume evalRecvArgs rs res g p = (g',p')
        with evalRecvArgs-gState-progress-rel assms have
          (g,g') ∈ gState-progress-rel prog
          by (metis fst-conv)
```

```
      hence length (channels g) ≤ length (channels g')
        by (simp add: gState-progress-rel-def)
    }
    ultimately show ?thesis using Channel DEF
      apply (cases v)
      apply (auto simp add: withChannel'-def withVar'-def for-all-def
                  split: channel.split prod.split
                  elim: fstE
                  intro!: evalRecvArgs-gState-progress-rel
                          gState-progress-rel-channels-update-step)
        apply force+
      done
  next
    case HSChannel
    obtain g' p' where *: evalRecvArgs rs (hsdata g) g p = (g',p')
      by (metis prod.exhaust)
    with assms have (g,g') ∈ gState-progress-rel prog
      by (auto elim!: fstE intro!: evalRecvArgs-gState-progress-rel)
    also hence gState-inv prog g' by blast
    hence (g',g'(|handshake := 0, hsdata := []|)) ∈ gState-progress-rel prog
      by (auto simp add: gState-progress-rel-def gState-inv-def)
    finally
    have (g,g'(|handshake := 0, hsdata := []|)) ∈ gState-progress-rel prog .
    with DEF HSChannel * show ?thesis
      by (cases v) (auto simp add: withChannel'-def withVar'-def for-all-def
                    split: channel.split prod.split)
  next
    case InvChannel with DEF show ?thesis
      by (cases v) (auto simp add: withChannel'-def withVar'-def)
  qed
next
  case False with EERecv idx' v assms show ?thesis
    by (cases v) (auto simp add: withChannel'-def withVar'-def)
qed
qed simp-all

lemma evalEffect-cl-inv:
  assumes cl-inv (g,p)
  and program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  shows cl-inv  (evalEffect e prog g p)
  using assms
proof (cases e)
  case EERun with assms show ?thesis by (force intro!: runProc-cl-inv)
next
  case (EESend c es srt) then obtain v where ChanRef v = c by (cases c) simp
  with EESend assms show ?thesis
    by (cases v)
```

```
      (auto simp add: withChannel'-def withVar'-def split: channel.split
          intro!: cl-inv-channels-update)
next
  case (EERecv c es srt) then obtain v where ChanRef v = c by (cases c) simp
  with EERecv assms show ?thesis
    apply (cases v)
   apply (auto simp add: withChannel'-def withVar'-def split: channel.split prod.split
              intro!: cl-inv-channels-update)
    apply (metis evalRecvArgs-cl-inv)+
    done
qed (simp-all add: setVar-cl-inv mkVarChannelProc-cl-inv)
```

5.8.2 Executable edges

To find a successor global state, we first need to find all those edges which are executable (i. e. the condition evaluates to true).

```
type-synonym choices = (edge * pState) list
  — A choice is an executable edge and the process it belongs to.

definition getChoices :: gState_I ⇒ pState ⇒ edge list ⇒ choices where
  getChoices g p = foldl (λE e.
      if evalHandshake (cond e) (handshake g) g p ∧ evalCond (cond e) g p
      then (e,p)#E
      else E) []

lemma getChoices-sub-edges-fst:
  fst ‘ set (getChoices g p es) ⊆ set es
unfolding getChoices-def
by (rule foldl-rule-aux) auto

lemma getChoices-sub-edges:
  (a,b) ∈ set (getChoices g p es) ⟹ a ∈ set es
using getChoices-sub-edges-fst
by force

lemma getChoices-p-snd:
  snd ‘ set (getChoices g p es) ⊆ {p}
unfolding getChoices-def
by (rule foldl-rule-aux) auto

lemma getChoices-p:
  (a,b) ∈ set (getChoices g p es) ⟹ b = p
using getChoices-p-snd
by force

definition sort-by-pri where
  sort-by-pri min-pri edges = foldl (λes e.
      let idx = nat-of-integer (abs (prio e))
      in if idx > min-pri
```

```
        then abort STR ''Invalid priority'' (λ-. es)
        else let ep = e # (es ! idx) in es[idx := ep]
      ) (replicate (min-pri + 1) []) edges

lemma sort-by-pri-edges':
  assumes set edges ⊆ A
  shows set (sort-by-pri min-pri edges) ⊆ {xs. set xs ⊆ A}
using assms
unfolding sort-by-pri-def
apply (rule-tac I=λσ -. (∀ x ∈ set σ. set x ⊆ A) ∧ length σ = min-pri + 1 in
foldl-rule-aux-P)
    apply simp
  apply (force dest!: subsetD[OF set-update-subset-insert]
              split: if-splits)
apply force
done

lemma sort-by-pri-edges:
  assumes set edges ⊆ A
  and es ∈ set (sort-by-pri min-pri edges)
  shows set es ⊆ A
using sort-by-pri-edges'[OF assms(1)] assms
by auto

lemma sort-by-pri-length:
  length (sort-by-pri min-pri edges) = min-pri + 1
unfolding sort-by-pri-def
by (rule foldl-rule-aux-P [where I=λσ -. length σ = min-pri + 1])
   simp-all

definition executable
  :: states iarray ⇒ gState_I ⇒ choices nres
  — Find all executable edges
 where
  executable ss g = (
      let procs = procs g in
      nfoldli procs (λ-. True) (λp E.
        if (exclusive g = 0 ∨ exclusive g = pid p) then do {
            let (min-pri, edges) = (ss !! pState.idx p) !! pc p;
            ASSERT(set edges ⊆ edgeSet (ss !! pState.idx p));

            (E',-,-) ←
               if min-pri = 0 then do {
                  WHILE_T (λ(E,brk,-). E = [] ∧ brk = 0) (λ (-, -, ELSE). do {
                    let g = g(|gState_I.else := ELSE|);
                        E = getChoices g p edges
                    in
                        if E = [] then (
                           if ¬ ELSE then RETURN (E, 0::nat, True)
```

```
                    else RETURN (E, 1, False))
                  else RETURN (E, 1, ELSE) }) ([], 0::nat, False)
            } else do {
              let min-pri = nat-of-integer (abs min-pri);
              let pri-edges = sort-by-pri min-pri edges;
              ASSERT (∀ es ∈ set pri-edges.
                        set es ⊆ edgeSet (ss !! pState.idx p));
              let pri-edges = IArray pri-edges;

              WHILE_T (λ(E,pri,-). E = [] ∧ pri ≤ min-pri) (λ(-, pri, ELSE). do
{
                let es = pri-edges !! pri;
                let g = g(|gState_I.else := ELSE|);
                let E = getChoices g p es;
                if E = [] then (
                  if ¬ ELSE then RETURN (E,pri,True)
                  else RETURN (E, pri + 1, False))
                else RETURN (E, pri, ELSE) }) ([], 0, False)
            };
          RETURN (E'@E)
        } else RETURN E
      ) []
 )

definition
  while-rel1 =
        measure (λx. if x = [] then 1 else 0)
  <*lex*> measure (λx. if x = 0 then 1 else 0)
  <*lex*> measure (λx. if ¬ x then 1 else 0)

lemma wf-while-rel1:
  wf while-rel1
unfolding while-rel1-def
by auto

definition
  while-rel2 mp =
        measure (λx. if x = [] then 1 else 0)
  <*lex*> measure (λx. (mp + 1) − x)
  <*lex*> measure (λx. if ¬ x then 1 else 0)

lemma wf-while-rel2:
  wf (while-rel2 mp)
unfolding while-rel2-def
by auto

lemma executable-edgeSet:
  assumes gState-inv prog g
  and program-inv prog
```

```
and ss = states prog
shows executable ss g
  ≤ SPEC (λcs. ∀(e,p) ∈ set cs.
                 e ∈ edgeSet (ss !! pState.idx p)
               ∧ pState-inv prog p
               ∧ cl-inv (g,p))
unfolding executable-def
using assms
apply (refine-rcg refine-vcg nfoldli-rule[where
          I=λ- - cs. ∀(e,p) ∈ set cs.
                       e ∈ edgeSet (ss !! pState.idx p)
                     ∧ pState-inv prog p
                     ∧ cl-inv (g,p)])
       apply simp
      apply (rename-tac p l1 l2 σ e p')
      apply (subgoal-tac pState-inv prog p)
       apply (clarsimp simp add: edgeSet-def pState-inv-def)[]
       apply (rule-tac x=IArray.list-of
                       (IArray.list-of (states prog) ! pState.idx p) ! pc p in bexI)
        apply simp
       apply (force intro!: nth-mem)
      apply (force simp add: gState-inv-def)
    apply (refine-rcg refine-vcg wf-while-rel1 WHILET-rule[where
             I=λ(cs,-,-). ∀(e,p) ∈ set cs.
                            e ∈ edgeSet (ss !! pState.idx p)
                          ∧ pState-inv prog p
                          ∧ cl-inv (g,p)
             and R=while-rel1])
      apply (vc-solve solve: asm-rl simp add: while-rel1-def)
     apply (frule getChoices-p)
     using  getChoices-sub-edges apply (force simp add: gState-inv-def)
  using sort-by-pri-edges apply fastforce
apply (rename-tac min-pri pri-edges)
apply (rule-tac I=λ(cs,-,-). ∀(e,p) ∈ set cs.
                               e ∈ edgeSet (ss !! pState.idx p)
                             ∧ pState-inv prog p
                             ∧ cl-inv (g,p)
           and R=while-rel2 (nat-of-integer (abs min-pri))
            in WHILET-rule)
      apply (simp add: wf-while-rel2)
     apply simp
  apply (refine-rcg refine-vcg)
      apply (clarsimp-all split: prod.split simp add: while-rel2-def)
    apply (metis diff-less-mono lessI)
  apply (rename-tac idx' else e p')
  apply (frule getChoices-p)
  apply (frule getChoices-sub-edges)
  apply (subgoal-tac
          sort-by-pri (nat-of-integer |min-pri|) pri-edges ! idx'
```

$\in$ *set* (*sort-by-pri* (*nat-of-integer* |*min-pri*|) *pri-edges*))
apply (*auto simp add*: *assms gState-inv-def*)[]
apply (*force simp add*: *sort-by-pri-length*)
apply (*auto simp add*: *assms*)
done

lemma *executable-edgeSet'*:
assumes *gState-inv prog g*
and *program-inv prog*
shows *executable* (*states prog*) *g*
$\leq$ *SPEC* (λ*cs*. $\forall$ (*e*,*p*) $\in$ *set cs*.
e $\in$ *edgeSet* ((*states prog*) !! *pState.idx p*)
$\wedge$ *pState-inv prog p*
$\wedge$ *cl-inv*(*g*,*p*))
using *executable-edgeSet*[**where** *ss* = *states prog*] *assms*
by *simp*

schematic-goal *executable-refine*:
RETURN (*?ex s g*) $\leq$ *executable s g*
unfolding *executable-def*
by (*refine-transfer*)

concrete-definition *executable-impl* **for** *s g* **uses** *executable-refine*

5.8.3 Successor calculation

function to_I **where**
to_I (| *gState.vars* = *v*, *channels* = *ch*, *timeout* = *t*, *procs* = *p* |)
= (| *gState.vars* = *v*, *channels* = *ch*, *timeout* = *False*, *procs* = *p*,
handshake = *0*, *hsdata* = [], *exclusive* = *0*, $gState_I$.*else* = *False* |)
by (*metis gState.cases*) (*metis gState.ext-inject*)
termination by *lexicographic-order*

function $from_I$ **where**
$from_I$ (| *gState.vars* = *v*, *channels* = *ch*, *timeout* = *t*, *procs* = *p*, ... = *m* |)
= (| *gState.vars* = *v*, *channels* = *ch*, *timeout* = *t*, *procs* = *p* |)
by (*metis gState.surjective*) (*metis gState.ext-inject*)
termination by *lexicographic-order*

function $reset_I$ **where**
$reset_I$ (| *gState.vars* = *v*, *channels* = *ch*, *timeout* = *t*, *procs* = *p*,
handshake = *hs*, *hsdata* = *hsd*, *exclusive* = -, $gState_I$.*else* = - |)
= (| *gState.vars* = *v*, *channels* = *ch*, *timeout* = *False*, *procs* = *p*,
handshake = *0*, *hsdata* = *if hs* $\neq$ *0 then hsd else* [], *exclusive* = *0*,
$gState_I$.*else* = *False* |)
by (*metis* (*full-types*) $gState_I$.*surjective unit.exhaust*)
(*metis gState.select-convs* $gState_I$.*select-convs*)
termination by *lexicographic-order*

lemma *gState-inv-to$_I$*:
gState-inv prog g = gState-inv prog (to$_I$ g)
by (*cases g*, *simp add*: *gState-inv-def cl-inv-def*)

lemma *gState-inv-from$_I$*:
gState-inv prog g = gState-inv prog (from$_I$ g)
by (*cases g*, *simp add*: *gState-inv-def cl-inv-def*)

lemma *gState-inv-reset$_I$*:
gState-inv prog g = gState-inv prog (reset$_I$ g)
by (*cases g*, *simp add*: *gState-inv-def cl-inv-def*)

lemmas *gState-inv-I-simps =*
gState-inv-to$_I$ gState-inv-from$_I$ gState-inv-reset$_I$

definition *removeProcs*
— Remove ended processes, if there is no running one with a higher pid.
where
removeProcs ps = foldr (λ*p* (*dead*,*sd*,*ps*,*dcs*).
if dead ∧ *pc p = 0 then* (*True*, *True*, *ps*, *pState.channels p* @ *dcs*)
else (*False*, *sd*, *p*#*ps*, *dcs*)) *ps* (*True*, *False*, [], [])

lemma *removeProcs-subset′*:
set (*fst* (*snd* (*snd* (*removeProcs ps*)))) ⊆ *set ps*
unfolding *removeProcs-def*
apply (*subst foldr-conv-foldl*)
apply (*rule foldl-rule-aux-P*[**where** *I*=λ(-,-,*ps′*,-) -. *set ps′* ⊆ *set ps*])
apply *simp*
apply (*clarsimp split*: *if-splits*)
apply *force*
apply (*rename-tac p*)
apply (*case-tac p = x*)
apply (*subst set-rev*[*symmetric*])
apply *simp*
apply *force*
apply *force*
done

lemma *removeProcs-length′*:
length (*fst* (*snd* (*snd* (*removeProcs ps*)))) ≤ *length ps*
unfolding *removeProcs-def*
apply (*subst foldr-conv-foldl*)
apply (*rule foldl-rule-aux-P*[**where**
I=λ(-,-,*ps′*,-) *ps″*. *length ps′* + *length ps″* ≤ *length ps*])
apply (*auto split*: *if-splits*)
done

lemma *removeProcs-subset*:

removeProcs ps = (dead,sd,ps',dcs) ⟹ set ps' ⊆ set ps
using *removeProcs-subset'*
by (*metis fst-conv snd-conv*)

lemma *removeProcs-length*:
removeProcs ps = (dead,sd,ps',dcs) ⟹ length ps' ≤ length ps
using *removeProcs-length'*
by (*metis fst-conv snd-conv*)

definition *cleanChans* :: *integer list ⇒ channels ⇒ channels*
— Mark channels of closed processes as invalid.
where
cleanChans dchans cs = snd (*foldl* (*λ*(*i*,*cs*) *c*.
if List.member dchans i then (*i + 1*, *cs*@[*InvChannel*])
else (*i + 1*, *cs*@[*c*])) (*0*, []) *cs*)

lemma *cleanChans-channel-inv*:
assumes *set cs ⊆ Collect channel-inv*
shows *set* (*cleanChans dchans cs*) *⊆ Collect channel-inv*
using *assms*
unfolding *cleanChans-def*
apply (*rule-tac foldl-rule-aux*)
apply (*force split*: *if-splits*)+
done

lemma *cleanChans-length*:
length (*cleanChans dchans cs*) = *length cs*
unfolding *cleanChans-def*
by (*rule foldl-rule-aux-P*[**where**
I=*λ*(-,*cs'*) *cs''*. *length cs'* + *length cs''* = *length cs*])
(*force split*: *if-splits*)+

definition *checkDeadProcs* :: *'a gState-scheme ⇒ 'a gState-scheme* **where**
checkDeadProcs g = (
let (-, *soDied*, *procs*, *dchans*) = *removeProcs* (*procs g*) *in*
if soDied then
g(| *procs* := *procs*, *channels* := *cleanChans dchans* (*channels g*)|)
else g)

lemma *checkDeadProcs-gState-progress-rel*:
assumes *gState-inv prog g*
shows (*g*, *checkDeadProcs g*) ∈ *gState-progress-rel prog*
using *assms cleanChans-length*[**where** *cs*=*channels g*]
cleanChans-channel-inv[**where** *cs*=*channels g*]
unfolding *checkDeadProcs-def*
apply (*intro gState-progress-relI*)
apply *assumption*
apply (*clarsimp split*: *if-splits prod.splits*)
apply (*frule removeProcs-length*)

```
    apply (frule removeProcs-subset)
    apply (auto simp add: gState-inv-def cl-inv-def)[]
  apply (clarsimp split: if-splits prod.splits)+
done

lemma gState-progress-rel-exclusive:
  (g, g') ∈ gState-progress-rel prog
   ⟹ (g, g'(|exclusive := p|)) ∈ gState-progress-rel prog
by (simp add:  gState-progress-rel-def gState-inv-def cl-inv-def)

definition applyEdge
  :: program ⇒ edge ⇒ pState ⇒ gState_I ⇒ gState_I nres
 where
  applyEdge prog e p g = do {

        let (g',p') = evalEffect (effect e) prog g p;
        ASSERT ((g,g') ∈ gState-progress-rel prog);
        ASSERT (pState-inv prog p');
        ASSERT (cl-inv (g',p'));

        let p'' = (case target e of Index t ⇒
                    if t < IArray.length (states prog !! pState.idx p') then p'(|pc := t|)
                    else abort STR ''Edge target out of bounds'' (λ-. p')
                  | - ⇒  abort STR ''Edge target not Index'' (λ-. p'));
        ASSERT (pState-inv prog p'');

        let g'' = g'(|procs := list-update (procs g') (pid p'' - 1) p''|);
        ASSERT ((g',g'') ∈ gState-progress-rel prog);

        let g''' = (if isAtomic e ∧ handshake g'' = 0
                    then g''(| exclusive := pid p'' |)
                    else g'');
        ASSERT ((g',g''') ∈ gState-progress-rel prog);

        let g_f = (if pc p'' = 0 then checkDeadProcs g''' else g''');
        ASSERT ((g''',g_f) ∈ gState-progress-rel prog);
        RETURN g_f }

lemma applyEdge-gState-progress-rel:
  assumes program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  and cl-inv (g,p)
  and e ∈ edgeSet (states prog !! pState.idx p)
  shows applyEdge prog e p g ≤ SPEC (λg'. (g,g') ∈ gState-progress-rel prog)
using assms
unfolding applyEdge-def
apply (intro refine-vcg)
```

apply (*force elim*: *fstE intro*!: *evalEffect-gState-progress-rel*)

apply (*force elim*: *sndE intro*!: *evalEffect-pState-inv*)

apply (*drule subst*)
apply (*rule evalEffect-cl-inv*)
apply *assumption*+

apply (*cases target e*)
apply (*clarsimp simp add*: *pState-inv-def*)
apply *simp*

apply (*frule gState-progress-rel-gState-invI2*)
apply (*cases target e*)
  apply (*clarsimp split*: *if-splits*)
  apply (*intro gState-progress-relI*)
    apply *assumption*
    apply (*auto simp add*: *gState-inv-def cl-inv-def*
          dest!: *subsetD*[*OF set-update-subset-insert*])[]
    apply (*simp add*: *cl-inv-def*)
    apply *simp*
  apply (*intro gState-progress-relI*)
    apply *assumption*
    apply (*auto simp add*: *gState-inv-def cl-inv-def*
          dest!: *subsetD*[*OF set-update-subset-insert*])[]
    apply (*simp add*: *cl-inv-def*)
    apply *simp*
  apply (*clarsimp split*: *if-splits*)
  apply (*intro gState-progress-relI*)
    apply *assumption*
    apply (*auto simp add*: *gState-inv-def cl-inv-def*
          dest!: *subsetD*[*OF set-update-subset-insert*])[]
    apply (*simp add*: *cl-inv-def*)
    apply *simp*

apply (*auto split*: *if-splits intro*!: *gState-progress-rel-exclusive*)[]

apply (*force intro*!: *checkDeadProcs-gState-progress-rel*)

apply (*blast intro*!: *gState-progress-rel-trans*)
done

schematic-goal *applyEdge-refine*:
  RETURN (*?ae prog e p g*) ≤ *applyEdge prog e p g*
unfolding *applyEdge-def*
by (*refine-transfer*)

concrete-definition *applyEdge-impl* **for** *e p g* **uses** *applyEdge-refine*

```
definition nexts
  :: program ⇒ gState ⇒ gState ls nres
  — The successor function
where
  nexts prog g = (
    let
      f = from_I;
      g = to_I g
    in

      REC (λD g. do {
        E ← executable (states prog) g;
        if E = [] then
          if handshake g ≠ 0 then
            — HS not possible – remove current step
            RETURN (ls.empty())
          else if exclusive g ≠ 0 then
            — Atomic blocks – just return current state
            RETURN (ls.sng (f g))
          else if ¬ timeout g then
            — Set timeout
            D (g(|timeout := True|))
          else
            — If all else fails: stutter
            RETURN (ls.sng (f (reset_I g)))
        else
          — Setting the internal variables (exclusive, handshake, ...) to 0
          — is safe – they are either set by the edges, or not thought
          — to be used outside executable.
          let g = reset_I g in
          nfoldli E (λ-. True) (λ(e,p) G.
              applyEdge prog e p g ≫= (λ g'.
              if handshake g' ≠ 0 ∨ isAtomic e then do {
                G_R ← D g';
                if ls.isEmpty G_R ∧ handshake g' = 0 then
                  — this only happens if the next step is a handshake, which fails
                  — hence we stay at the current state
                  RETURN (ls.ins (f g') G)
                else
                  RETURN (ls.union G_R G)
              } else RETURN (ls.ins (f g') G))) (ls.empty())
      }) g
      ≫= (λG. if ls.isEmpty G then RETURN (ls.sng (f g)) else RETURN G)
)

lemma gState-progress-rel-intros:
  (to_I g, gI') ∈ gState-progress-rel prog
      ⟹ (g, from_I gI') ∈ gState-progress-rel prog
  (gI, gI') ∈ gState-progress-rel prog
```

$\Longrightarrow$ $(gI,\ reset_I\ gI') \in$ *gState-progress-rel prog*
$(to_I\ g,\ gI') \in$ *gState-progress-rel prog*
$\Longrightarrow$ $(to_I\ g,\ gI'(\!|timeout := t|\!)) \in$ *gState-progress-rel prog*
unfolding *gState-progress-rel-def*
by (*cases g*, *cases gI'*, *cases gI*, *force simp add*: *gState-inv-def cl-inv-def*)+

lemma *gState-progress-rel-step-intros*:
$(to_I\ g,\ g') \in$ *gState-progress-rel prog*
$\Longrightarrow$ $(reset_I\ g',\ g'') \in$ *gState-progress-rel prog*
$\Longrightarrow$ $(g,\ from_I\ g'') \in$ *gState-progress-rel prog*
$(to_I\ g,\ g') \in$ *gState-progress-rel prog*
$\Longrightarrow$ $(reset_I\ g',\ g'') \in$ *gState-progress-rel prog*
$\Longrightarrow$ $(to_I\ g,\ g'') \in$ *gState-progress-rel prog*
unfolding *gState-progress-rel-def*
by (*cases g*, *cases g'*, *cases g''*, *force simp add*: *gState-inv-def cl-inv-def*)+

lemma *cl-inv-reset*$_I$:
cl-inv(*g*,*p*) $\Longrightarrow$ *cl-inv*($reset_I\ g,\ p$)
by (*cases g*) (*simp add*: *cl-inv-def*)

lemmas *refine-helpers* =
gState-progress-rel-intros gState-progress-rel-step-intros cl-inv-reset$_I$

lemma *nexts-SPEC*:
assumes *gState-inv prog g*
and *program-inv prog*
shows *nexts prog g* $\leq$ *SPEC* ($\lambda gs.\ \forall g' \in ls.\alpha\ gs.\ (g,g') \in$ *gState-progress-rel prog*)
using *assms*
unfolding *nexts-def*
apply (*refine-rcg refine-vcg REC-rule*[**where**
pre=$\lambda g'.\ (to_I\ g,g') \in$ *gState-progress-rel prog*])
apply (*simp add*: *gState-inv-to*$_I$)
apply (*rule order-trans*[*OF executable-edgeSet'*])
apply (*drule gState-progress-rel-gState-invI2*)
apply *assumption*
apply *assumption*
apply (*refine-rcg refine-vcg nfoldli-rule*[**where**
I=λ- - σ. $\forall g' \in ls.\alpha\ \sigma.\ (g,g') \in$ *gState-progress-rel prog*]
order-trans[*OF applyEdge-gState-progress-rel*])
apply (*vc-solve intro*: *refine-helpers solve*: *asm-rl simp add*: *ls.correct*)
apply (*rule order-trans*)
apply (*rprems*)
apply (*vc-solve intro*: *refine-helpers solve*: *asm-rl simp add*: *ls.correct*)
apply (*rule order-trans*)
apply *rprems*
apply (*vc-solve intro*: *refine-helpers solve*: *asm-rl simp add*: *ls.correct*)
done

lemma *RETURN-dRETURN*:
RETURN f ≤ *f′* ⟹ *nres-of* (*dRETURN f*) ≤ *f′*
unfolding *nres-of-def*
by *simp*

lemma *executable-dRETURN*:
nres-of (*dRETURN* (*executable-impl prog g*)) ≤ *executable prog g*
using *executable-impl.refine*
by (*simp add*: *RETURN-dRETURN*)

lemma *applyEdge-dRETURN*:
nres-of (*dRETURN* (*applyEdge-impl prog e p g*)) ≤ *applyEdge prog e p g*
using *applyEdge-impl.refine*
by (*simp add*: *RETURN-dRETURN*)

schematic-goal *nexts-code-aux*:
nres-of (*?nexts prog g*) ≤ *nexts prog g*
unfolding *nexts-def*
by (*refine-transfer the-resI executable-dRETURN applyEdge-dRETURN*)

concrete-definition *nexts-code-aux* **for** *prog g* **uses** *nexts-code-aux*
prepare-code-thms *nexts-code-aux-def*

5.8.4 Handle non-termination

A Promela model may include non-terminating parts. Therefore we cannot guarantee, that *nexts* will actually terminate. To avoid having to deal with this in the model checker, we fail in case of non-termination.

definition *SUCCEED-abort* **where**
SUCCEED-abort msg dm m = (
case m of
RES X ⇒ *if X*={} *then Code.abort msg* (λ-. *dm*) *else RES X*
| - ⇒ *m*)

definition *dSUCCEED-abort* **where**
dSUCCEED-abort msg dm m = (
case m of
dSUCCEEDi ⇒ *Code.abort msg* (λ-. *dm*)
| - ⇒ *m*)

definition *ref-succeed* **where**
ref-succeed m m′ ⟷ *m* ≤ *m′* ∧ (*m*=*SUCCEED* ⟶ *m′*=*SUCCEED*)

lemma *dSUCCEED-abort-SUCCEED-abort*:
⟦ *RETURN dm′* ≤ *dm*; *ref-succeed* (*nres-of m′*) *m* ⟧
⟹ *nres-of* (*dSUCCEED-abort msg* (*dRETURN dm′*) (*m′*))
≤ *SUCCEED-abort msg dm m*
unfolding *dSUCCEED-abort-def SUCCEED-abort-def ref-succeed-def*

by (*auto split*: *dres.splits nres.splits*)

The final successor function now incorporates:

1. *nexts*
2. handling of non-termination

definition *nexts-code* **where**
nexts-code prog g =
the-res (*dSUCCEED-abort* (*STR ''The Universe is broken!''*)
(*dRETURN* (*ls.sng g*))
(*nexts-code-aux prog g*))

lemma *nexts-code-SPEC*:
assumes *gState-inv prog g*
and *program-inv prog*
shows $g' \in ls.\alpha$ (*nexts-code prog g*)
$\Longrightarrow$ (*g*,*g'*) $\in$ *gState-progress-rel prog*
unfolding *nexts-code-def*
unfolding *dSUCCEED-abort-def*
using *assms*
using *order-trans*[*OF nexts-code-aux.refine nexts-SPEC*[*OF assms*(*1*,*2*)]]
by (*auto split*: *dres.splits simp*: *ls.correct*)

5.9 Finiteness of the state space

inductive-set *reachable-states*
for *P* :: *program*
and g_s :: *gState* — start state
where
$g_s \in$ *reachable-states P* g_s |
$g \in$ *reachable-states P* $g_s \Longrightarrow x \in ls.\alpha$ (*nexts-code P g*)
$\Longrightarrow x \in$ *reachable-states P* g_s

lemmas *reachable-states-induct*[*case-names init step*] =
reachable-states.induct[*split-format* (*complete*)]

lemma *reachable-states-finite*:
assumes *program-inv prog*
and *gState-inv prog g*
shows *finite* (*reachable-states prog g*)
proof (*rule finite-subset*[*OF - gStates-finite*[*of - g*]])
define *INV* **where** *INV g'* $\longleftrightarrow$ *g'* $\in$ (*gState-progress-rel prog*)* `` {*g*} $\wedge$ *gState-inv prog g'* **for** *g'*

{
fix *g'*
have *g'* $\in$ *reachable-states prog g* $\Longrightarrow$ *INV g'*
proof (*induct rule*: *reachable-states-induct*)

```
    case init with assms show ?case by (simp add: INV-def)
  next
    case (step g g')
    from step(2,3) have
      (g, g') ∈ gState-progress-rel prog
      using nexts-code-SPEC[OF - ‹program-inv prog›]
      unfolding INV-def by auto
    thus ?case using step(2) unfolding INV-def by auto
  qed
 }

 thus reachable-states prog g ⊆
      (gState-progress-rel prog)* `` {g}
  unfolding INV-def by auto
qed
```

5.10 Traces

When trying to generate a lasso, we have a problem: We only have a list of global states. But what are the transitions to come from one to the other?

This problem shall be tackled by *replay*: Given two states, it generates a list of transitions that was taken.

```
definition replay :: program ⇒ gState ⇒ gState ⇒ choices nres where
 replay prog g₁ g₂ = (
   let
     g₁ = to_I g₁;
     check = λg. from_I g = g₂
   in
     REC_T (λD g. do {
     E ← executable (states prog) g;
     if E = [] then
       if check g then RETURN []
       else if ¬ timeout g then D (g(|timeout := True|))
       else abort STR ''Stuttering should not occur on replay''
                (λ-. RETURN [])
     else
       let g = reset_I g in
       nfoldli E (λE. E = []) (λ(e,p) -.
         applyEdge prog e p g ≫= (λg'.
           if handshake g' ≠ 0 ∨ isAtomic e then do {
             E_R ← D g';
             if E_R = [] then
               if check g' then RETURN [(e,p)] else RETURN []
             else
               RETURN ((e,p) # E_R)
           } else if check g' then RETURN [(e,p)] else RETURN [])
       ) []
   }) g₁
```

)

lemma *abort-refine*[*refine-transfer*]:
  nres-of (*f* ()) ≤ *F* () ⟹ *nres-of* (*abort s f*) ≤ *abort s F*
  f() ≠ *dSUCCEED* ⟹ *abort s f* ≠ *dSUCCEED*
by *auto*

schematic-goal *replay-code-aux*:
  RETURN (*?replay prog* g_1 g_2) ≤ *replay prog* g_1 g_2
unfolding *replay-def applyEdge-def*
by (*refine-transfer the-resI executable-dRETURN*)

concrete-definition *replay-code* **for** *prog* g_1 g_2 **uses** *replay-code-aux*
prepare-code-thms *replay-code-def*

5.10.1 Printing of traces

definition *procDescr*
  :: (*integer* ⇒ *string*) ⇒ *program* ⇒ *pState* ⇒ *string*
where
  procDescr f prog p = (
    let
      name = *String.explode* (*proc-names prog* !! *pState.idx p*);
      id = *f* (*integer-of-nat* (*pid p*))
    in
      name @ '' ('' @ *id* @ '')'')

definition *printInitial*
  :: (*integer* ⇒ *string*) ⇒ *program* ⇒ *gState* ⇒ *string*
where
  printInitial f prog g_0 = (
      let psS = *printList* (*procDescr f prog*) (*procs* g_0) [] [] '' '' *in*
      ''*Initially running*: '' @ *psS*)

abbreviation *lf* ≡ *CHR 0x0A*

fun *printConfig*
  :: (*integer* ⇒ *string*) ⇒ *program* ⇒ *gState option* ⇒ *gState* ⇒ *string*
where
  printConfig f prog None g_0 = *printInitial f prog* g_0
| *printConfig f prog* (*Some* g_0) g_1 = (
          let eps = *replay-code prog* g_0 g_1 *in*
          let print = (λ(*e*,*p*). *procDescr f prog p* @ '': '' @ *printEdge f* (*pc p*) *e*)
          in if eps = [] ∧ g_1 = g_0 *then* '' −− *stutter* −−''
             else printList print eps [] [] (*lf*#'' ''))

definition *printConfigFromAST f* ≡ *printConfig f o fst o setUp*

5.11 Code export

code-identifier
  code-module *PromelaInvariants* ⇀ (*SML*) *Promela*
| **code-module** *PromelaDatastructures* ⇀ (*SML*) *Promela*

definition *executable-triv prog g* = *executable-impl* (*snd prog*) *g*
definition *apply-triv prog g ep* = *applyEdge-impl prog* (*fst ep*) (*snd ep*) (*reset*$_I$ *g*)

export-code
  setUp printProcesses printConfigFromAST nexts-code executable-triv apply-triv extractLTLs lookupLTL
  checking *SML*

export-code
  setUp printProcesses printConfigFromAST nexts-code executable-triv apply-triv extractLTLs lookupLTL
  in *SML*
  file ‹*Promela.sml*›

end

6 LTL integration

theory *PromelaLTL*
imports
  Promela
  LTL.LTL
begin

We have a semantic engine for Promela. But we need to have an integration with LTL – more specificly, we must know when a proposition is true in a global state. This is achieved in this theory.

6.1 LTL optimization

For efficiency reasons, we do not store the whole *expr* on the labels of a system automaton, but *nat* instead. This index then is used to look up the corresponding *expr*.

type-synonym *APs* = *expr iarray*

primrec *ltlc-aps-list′* :: *'a ltlc* ⇒ *'a list* ⇒ *'a list*
where
  ltlc-aps-list′ True-ltlc l = *l*
| *ltlc-aps-list′ False-ltlc l* = *l*
| *ltlc-aps-list′* (*Prop-ltlc p*) *l* = (*if List.member l p then l else p#l*)
| *ltlc-aps-list′* (*Not-ltlc x*) *l* = *ltlc-aps-list′ x l*
| *ltlc-aps-list′* (*Next-ltlc x*) *l* = *ltlc-aps-list′ x l*

| *ltlc-aps-list'* (*Final-ltlc x*) *l* = *ltlc-aps-list' x l*
| *ltlc-aps-list'* (*Global-ltlc x*) *l* = *ltlc-aps-list' x l*
| *ltlc-aps-list'* (*And-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)
| *ltlc-aps-list'* (*Or-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)
| *ltlc-aps-list'* (*Implies-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)
| *ltlc-aps-list'* (*Until-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)
| *ltlc-aps-list'* (*Release-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)
| *ltlc-aps-list'* (*WeakUntil-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)
| *ltlc-aps-list'* (*StrongRelease-ltlc x y*) *l* = *ltlc-aps-list' y* (*ltlc-aps-list' x l*)

lemma *ltlc-aps-list'-correct*:
set (*ltlc-aps-list' φ l*) = *atoms-ltlc φ* ∪ *set l*
by (*induct φ arbitrary*: *l*) (*auto simp add*: *in-set-member*)

lemma *ltlc-aps-list'-distinct*:
distinct l ⟹ *distinct* (*ltlc-aps-list' φ l*)
by (*induct φ arbitrary*: *l*) (*auto simp add*: *in-set-member*)

definition *ltlc-aps-list* :: *'a ltlc* ⇒ *'a list*
where
ltlc-aps-list φ = *ltlc-aps-list' φ* []

lemma *ltlc-aps-list-correct*:
set (*ltlc-aps-list φ*) = *atoms-ltlc φ*
unfolding *ltlc-aps-list-def*
by (*force simp*: *ltlc-aps-list'-correct*)

lemma *ltlc-aps-list-distinct*:
distinct (*ltlc-aps-list φ*)
unfolding *ltlc-aps-list-def*
by (*auto intro*: *ltlc-aps-list'-distinct*)

primrec *idx'* :: *nat* ⇒ *'a list* ⇒ *'a* ⇒ *nat option* **where**
idx' - [] - = *None*
| *idx' ctr* (*x*#*xs*) *y* = (*if x* = *y then Some ctr else idx'* (*ctr*+*1*) *xs y*)

definition *idx* = *idx' 0*

lemma *idx'-correct*:
assumes *distinct xs*
shows *idx' ctr xs y* = *Some n* ⟷ *n* ≥ *ctr* ∧ *n* < *length xs* + *ctr* ∧ *xs* ! (*n*−*ctr*)
= *y*
using *assms*
proof (*induction xs arbitrary*: *n ctr*)
case (*Cons x xs*)
show *?case*
proof (*cases x*=*y*)
case *True* **with** *Cons.prems* **have** ∗: *y* ∉ *set xs* **by** *auto*
{

```
      assume A: (y#xs)!(n−ctr) = y
      and less: ctr ≤ n
      and length: n < length (y#xs) + ctr
      have n = ctr
      proof (rule ccontr)
        assume n ≠ ctr with less have n−ctr > 0 by auto
        moreover from ‹n≠ctr› length have n−ctr < length(y#xs) by auto
        ultimately have (y#xs)!(n−ctr) ∈ set xs by simp
        with A * show False by auto
      qed
    }
    with True Cons show ?thesis by auto
  next
    case False
    from Cons have distinct xs by simp
    with Cons.IH False have idx' (Suc ctr) xs y = Some n ⟷ Suc ctr ≤ n ∧ n
< length xs + Suc ctr ∧ xs ! (n − Suc ctr) = y
      by simp
    with False show ?thesis
      apply −
      apply (rule iffI)
      apply clarsimp-all
      done
  qed
qed simp

lemma idx-correct:
  assumes distinct xs
  shows idx xs y = Some n ⟷ n < length xs ∧ xs ! n = y
  using idx'-correct[OF assms]
  by (simp add: idx-def)

lemma idx-dom:
  assumes distinct xs
  shows dom (idx xs) = set xs
  by (auto simp add: idx-correct assms in-set-conv-nth)

lemma idx-image-self:
  assumes distinct xs
  shows (the ∘ idx xs) ‘ set xs = {..<length xs}
proof (safe)
  fix x
  assume x ∈ set xs with in-set-conv-nth obtain n where n: n < length xs xs !
n = x by metis
  with idx-correct[OF assms] have idx xs x = Some n by simp
  hence the (idx xs x) = n by simp
  with n show (the ∘ idx xs) x < length xs by simp
next
  fix n
```

```
  assume n < length xs
  moreover with nth-mem have xs ! n ∈ set xs by simp
  then obtain x where xs ! n = x x ∈ set xs by simp-all
  ultimately have idx xs x = Some n by (simp add: idx-correct[OF assms])
  hence the (idx xs x) = n by simp
  thus n ∈ (the ∘ idx xs) ‘ set xs
    using ‹x ∈ set xs›
    by auto
qed

lemma idx-ran:
  assumes distinct xs
  shows ran (idx xs) = {..<length xs}
  using ran-is-image[where M=idx xs]
  using idx-image-self[OF assms] idx-dom[OF assms]
  by simp

lemma idx-inj-on-dom:
  assumes distinct xs
  shows inj-on (idx xs) (dom (idx xs))
  by (fastforce simp add: idx-dom assms in-set-conv-nth idx-correct
                intro!: inj-onI)

definition ltl-convert :: expr ltlc ⇒ APs × nat ltlc where
  ltl-convert φ = (
      let APs = ltlc-aps-list φ;
          φ_i = map-ltlc (the ∘ idx APs) φ
      in (IArray APs, φ_i))

lemma ltl-convert-correct:
  assumes ltl-convert φ = (APs, φ_i)
  shows atoms-ltlc φ = set (IArray.list-of APs) (is ?P1)
  and atoms-ltlc φ_i = {..<IArray.length APs} (is ?P2)
  and φ_i = map-ltlc (the ∘ idx (IArray.list-of APs)) φ (is ?P3)
  and distinct (IArray.list-of APs)
proof −
  let ?APs = IArray.list-of APs

  from assms have APs-def: ?APs = ltlc-aps-list φ
    unfolding ltl-convert-def by auto

  with ltlc-aps-list-correct show APs-set: ?P1 by metis

  from assms show ?P3
    unfolding ltl-convert-def
    by auto

  from assms have atoms-ltlc φ_i = (the ∘ idx ?APs) ‘ atoms-ltlc φ
    unfolding ltl-convert-def
```

by (*auto simp add*: *ltlc.set-map*)
moreover from *APs-def ltlc-aps-list-distinct* **show** *distinct ?APs* **by** *simp*
note *idx-image-self*[*OF this*]
moreover note *APs-set*
ultimately show *?P2* **by** *simp*
qed

definition *prepare*
:: - × (*program* ⇒ *unit*) ⇒ *ast* ⇒ *expr ltlc* ⇒ (*program* × *APs* × *gState*) × *nat ltlc*
where
prepare cfg ast $\varphi \equiv$
let
($prog$,g_0) = *Promela.setUp ast*;
(*APs*,φ_i) = *PromelaLTL.ltl-convert* φ
in
((*prog*, *APs*, g_0), φ_i)

lemma *prepare-instrument*[*code*]:
prepare cfg ast $\varphi \equiv$
let
(-,*printF*) = *cfg*;
- = *PromelaStatistics.start* ();
($prog$,g_0) = *Promela.setUp ast*;
- = *printF prog*;
(*APs*,φ_i) = *PromelaLTL.ltl-convert* φ;
- = *PromelaStatistics.stop-timer* ()
in
((*prog*, *APs*, g_0), φ_i)
by (*simp add*: *prepare-def*)

export-code *prepare* **checking** *SML*

6.2 Language of a Promela program

definition *propValid* :: *APs* ⇒ *gState* ⇒ *nat* ⇒ *bool* **where**
propValid APs g i ⟷ *i* < *IArray.length APs* ∧ *exprArith g emptyProc* (*APs*!!*i*) ≠ *0*

definition *promela-E* :: *program* ⇒ (*gState* × *gState*) *set*
— Transition relation of a promela program
where
promela-E prog ≡ {(g,g'). g' ∈ *ls*.α (*nexts-code prog g*)}

definition *promela-E-ltl* :: *program* × *APs* ⇒ (*gState* × *gState*) *set* **where**
promela-E-ltl = *promela-E* ∘ *fst*

definition *promela-is-run'* :: *program* × *gState* ⇒ *gState word* ⇒ *bool*
— Predicate defining runs of promela programs

where
promela-is-run′ progg r ≡
let (*prog*,g_0)=*progg in*
r 0 = g_0
∧ (∀ *i*. *r* (*Suc i*) ∈ *ls*.α (*nexts-code prog* (*r i*)))

abbreviation *promela-is-run* ≡ *promela-is-run′* ∘ *setUp*

definition *promela-is-run-ltl* :: *program* × *APs* × *gState* ⇒ *gState word* ⇒ *bool*
where
promela-is-run-ltl promg r ≡ *let* (*prog*,*APs*,*g*) = *promg in promela-is-run′* (*prog*,*g*)
r

definition *promela-props* :: *gState* ⇒ *expr set*
where
promela-props g = {*e*. *exprArith g emptyProc e* ≠ *0*}

definition *promela-props-ltl* :: *APs* ⇒ *gState* ⇒ *nat set*
where
promela-props-ltl APs g ≡ *Collect* (*propValid APs g*)

definition *promela-language* :: *ast* ⇒ *expr set word set* **where**
promela-language ast ≡ {*promela-props* ∘ *r* | *r*. *promela-is-run ast r*}

definition *promela-language-ltl* :: *program* × *APs* × *gState* ⇒ *nat set word set*
where
promela-language-ltl promg ≡ *let* (*prog*,*APs*,*g*) = *promg in*
{*promela-props-ltl APs* ∘ *r* | *r*. *promela-is-run-ltl promg r*}

lemma *promela-props-ltl-map-aprops*:
assumes *ltl-convert* φ = (*APs*,φ_i)
shows *promela-props-ltl APs* =
map-props (*idx* (*IArray.list-of APs*)) ∘ *promela-props*
proof −
let *?APs* = *IArray.list-of APs*
let *?idx* = *idx ?APs*

from *ltl-convert-correct assms* **have** *D*: *distinct ?APs* **by** *simp*

show *?thesis*
proof (*intro ext set-eqI iffI*)
fix *g i*
assume *i* ∈ *promela-props-ltl APs g*
hence *propValid APs g i* **by** (*simp add*: *promela-props-ltl-def*)
hence *l*: *i* < *IArray.length APs exprArith g emptyProc* (*APs*!!*i*) ≠ *0*
by (*simp-all add*: *propValid-def*)
hence *APs*!!*i* ∈ *promela-props g* **by** (*simp add*: *promela-props-def*)
moreover from *idx-correct l D* **have** *?idx* (*APs*!!*i*) = *Some i* **by** *fastforce*
ultimately show *i* ∈ (*map-props ?idx* ∘ *promela-props*) *g*

```
    unfolding o-def map-props-def
    by blast
  next
    fix g i
    assume i ∈ (map-props ?idx ∘ promela-props) g
    then obtain p where p-def: p ∈ promela-props g ?idx p = Some i
      unfolding map-props-def o-def
      by auto
    hence expr: exprArith g emptyProc p ≠ 0 by (simp add: promela-props-def)

    from D p-def have i < IArray.length APs APs !! i = p
      using idx-correct by fastforce+
    with expr have propValid APs g i by (simp add: propValid-def)
    thus i ∈ promela-props-ltl APs g
      by (simp add: promela-props-ltl-def)
  qed
qed

lemma promela-run-in-language-iff:
  assumes conv: ltl-convert φ = (APs,φ_i)
  shows promela-props ∘ ξ ∈ language-ltlc φ
        ⟷ promela-props-ltl APs ∘ ξ ∈ language-ltlc φ_i (is ?L ⟷ ?R)
proof −
  let ?APs = IArray.list-of APs

  from conv have D: distinct ?APs
    by (simp add: ltl-convert-correct)
  with conv have APs: atoms-ltlc φ ⊆ dom (idx ?APs)
    by (simp add: idx-dom ltl-convert-correct)

  note map-semantics = map-semantics-ltlc[OF idx-inj-on-dom[OF D] APs]
                       promela-props-ltl-map-aprops[OF conv]
                       ltl-convert-correct[OF conv]

  have ?L ⟷ (promela-props ∘ ξ) ⊨_c φ by (simp add: language-ltlc-def)
  also have ... ⟷ (promela-props-ltl APs ∘ ξ) ⊨_c φ_i
    using map-semantics
    by (simp add: o-assoc)
  also have ... ⟷ ?R
    by (simp add: language-ltlc-def)
  finally show ?thesis .
qed

lemma promela-language-sub-iff:
  assumes conv: ltl-convert φ = (APs,φ_i)
  and setUp: setUp ast = (prog,g)
 shows promela-language-ltl (prog,APs,g) ⊆ language-ltlc φ_i ⟷ promela-language
ast ⊆ language-ltlc φ
  using promela-run-in-language-iff[OF conv] setUp
```

by (*auto simp add*: *promela-language-ltl-def promela-language-def promela-is-run-ltl-def*)

hide-const (**open**) *abort abortv*
err errv
usc
warn the-warn with-warn

hide-const (**open**) *idx idx'*
end
theory *PromelaLTLConv*
imports
Promela
LTL.LTL
begin

6.3 Proposition types and conversion

LTL formulae and propositions are also generated by an SML parser. Hence we have the same setup as for Promela itself: Mirror the data structures and (sometimes) map them to new ones.

This theory is intended purely to be used by frontend code to convert from *propc* to *expr*. The other theories work on *expr* directly.
While we could of course convert directly, that would introduce yet a semantic level.

datatype *binOp* = *Eq* | *Le* | *LEq* | *Gr* | *GEq*

datatype *ident* = *Ident String.literal integer option*

datatype *propc* = *CProp ident*
| *BProp binOp ident ident*
| *BExpProp binOp ident integer*

fun *identConv* :: *ident* ⇒ *varRef* **where**
identConv (*Ident name None*) = *VarRef True name None*
| *identConv* (*Ident name* (*Some i*)) = *VarRef True name* (*Some* (*ExprConst i*))

definition *ident2expr* :: *ident* ⇒ *expr* **where**
ident2expr = *ExprVarRef* ∘ *identConv*

primrec *binOpConv* :: *binOp* ⇒ *PromelaDatastructures.binOp* **where**
binOpConv Eq = *BinOpEq*
| *binOpConv Le* = *BinOpLe*
| *binOpConv LEq* = *BinOpLEq*
| *binOpConv Gr* = *BinOpGr*
| *binOpConv GEq* = *BinOpGEq*

```
primrec propc2expr :: propc ⇒ expr where
  propc2expr (CProp ident) =
     ExprBinOp BinOpEq (ident2expr ident) (ExprConst 1)
| propc2expr (BProp bop il ir) =
     ExprBinOp (binOpConv bop) (ident2expr il) (ident2expr ir)
| propc2expr (BExpProp bop il ir) =
     ExprBinOp (binOpConv bop) (ident2expr il) (ExprConst ir)

definition ltl-conv :: propc ltlc ⇒ expr ltlc where
  ltl-conv = map-ltlc propc2expr

definition printPropc
  :: (integer ⇒ char list) ⇒ propc ⇒ char list
where
  printPropc f p = printExpr f (propc2expr p)
```

The semantics of a *propc* is given just for reference.

```
definition evalPropc :: gState ⇒ propc ⇒ bool where
  evalPropc g p ⟷ exprArith g emptyProc (propc2expr p) ≠ 0

end
```

References

[1] Promela manual pages. http://spinroot.com/spin/Man/promela.html. Accessed: 2013-02-07.

[2] G. J. Holzmann. *The Spin Model Checker — Primer and Reference Manual*. Addison-Wesley, 2003.