

Promela Formalization

By René Neumann

September 13, 2023

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	3
2	Abstract Syntax Tree	4
3	Data structures as used in Promela	7
3.1	Abstract Syntax Tree <i>after</i> preprocessing	8
3.2	Preprocess the AST of the parser into our variant	10
3.3	The transition system	24
3.4	State	24
3.5	Printing	26
4	Invariants for Promela data structures	29
4.1	Bounds	29
4.2	Variables and similar	29
4.3	Invariants of a process	32
4.4	Invariants of the global state	33
4.5	Invariants of the program	34
5	Formalization of Promela semantics	35
5.1	Misc Helpers	36
5.2	Variable handling	37
5.3	Expressions	40
5.4	Variable declaration	43
5.5	Folding	48
5.6	Starting processes	49
5.7	AST to edges	52
5.7.1	Setup	56
5.8	Semantic Engine	59
5.8.1	Evaluation of Edges	59
5.8.2	Executable edges	62
5.8.3	Successor calculation	65
5.8.4	Handle non-termination	70
5.9	Finiteness of the state space	71
5.10	Traces	71
5.10.1	Printing of traces	72
5.11	Code export	73
6	LTL integration	73
6.1	LTL optimization	73
6.2	Language of a Promela program	76
6.3	Proposition types and conversion	77

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
```

```
<ML>
```

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

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

```

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 stmt =
  StmtIf (step list) list
| StmtDo (step list) list
| StmtFor range step list
| StmtAtomic step list
| StmtDStep step list
| StmtSelect range
| StmtSeq step list
| StmtSend varRef expr list
| StmtSortSend varRef expr list
| StmtRecv varRef recvArg list
| StmtRndRecv varRef recvArg list
| StmtRecvX varRef recvArg list
| StmtRndRecvX varRef recvArg list
| StmtAssign varRef expr
| StmtIncr varRef
| StmtDecr varRef
| StmtElse
| StmtBreak
| StmtGoTo String.literal
| StmtLabeled String.literal stmt
| StmtPrintf String.literal expr list

```

```

    | StmtPrintM String.literal
    | StmtRun String.literal
      expr list
      integer option
    | StmtAssert expr
    | StmtCond expr
    | StmtCall 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*

$\langle ML \rangle$

The preprocessing is done for each type on its own.

```
primrec ppBinOp :: AST.binOp  $\Rightarrow$  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  $\Rightarrow$  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  $\times$  bool)) lm — channels
   $\times$  (String.literal, (integer option  $\times$  bool)) lm — variables
   $\times$  (String.literal, integer) lm — mtypes
   $\times$  (String.literal, varRef) lm — aliases (used for inlines)
```

definition dealWithVar

```
:: var-data  $\Rightarrow$  String.literal
 $\Rightarrow$  (String.literal  $\Rightarrow$  integer option  $\times$  bool  $\Rightarrow$  expr option  $\Rightarrow$  'a)
 $\Rightarrow$  (String.literal  $\Rightarrow$  integer option  $\times$  bool  $\Rightarrow$  expr option  $\Rightarrow$  'a)
 $\Rightarrow$  (integer  $\Rightarrow$  'a)  $\Rightarrow$  'a
```

where

```
dealWithVar cvm n fC fV fM  $\equiv$  (
  let (c,v,m,a) = cvm in
  let (n, idx) = case lm.lookup n a of
    None  $\Rightarrow$  (n, None)
  | Some (VarRef - name idx)  $\Rightarrow$  (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
    (\name (_,g) aIdx. let idx = map-option (ppExpr cvm) idx in
      Inr (ChanRef (VarRef g name (resolveIdx idx aIdx))))
    (\name (_,g) aIdx. let idx = map-option (ppExpr cvm) idx in
      Inl (VarRef g name (resolveIdx idx aIdx)))
    (\_. 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 size 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 (size,g) v, m, a),
          VarDeclNum l h name size
          (map-option (ppExpr (c,v,m,a)) init))))
| ppVarDecl - - g (AST.VarDeclNum name size init) =
  err STR "Assiging num to non-num"

| ppVarDecl (c,v,m,a) VTChan g
  (AST.VarDeclChan name size 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 (size, g) c, v, m, a),
        VarDeclChan name size cap'))
| ppVarDecl - - g (AST.VarDeclChan name size init) =
  err STR "Assiging chan to non-chan"

| ppVarDecl (c,v,m,a) (VTBounded l h) g
  (AST.VarDeclMType name size 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 (size,g) v, m, a),
      VarDeclNum l h name size init)))

```

| *ppVarDecl* - - *g* (*AST.VarDeclMType* *name* *size* *init*) =
err STR "Assiging num to non-num"

| *ppVarDecl* - - - (*AST.VarDeclUnsigned* - - -) =
usc STR "VarDeclUnsigned"

definition *ppProcVarDecl*

:: var-data \Rightarrow *varType* \Rightarrow *bool* \Rightarrow *AST.varDecl* \Rightarrow *var-data* \times *procVarDecl*

where

ppProcVarDecl *cvm* *ty* *g* *v* = (case *ppVarDecl* *cvm* *ty* *g* *v* of
(*cvm*, *VarDeclNum* *l* *h* *name* *size* *init*) \Rightarrow (*cvm*, *ProcVarDeclNum* *l* *h* *name*
size *init*)
| (*cvm*, *VarDeclChan* *name* *size* *None*) \Rightarrow (*cvm*, *ProcVarDeclChan* *name* *size*)
| - \Rightarrow *err STR "Channel initalizations only allowed at the beginning of pro-*
types.")

fun *ppProcArg*

:: var-data \Rightarrow *varType* \Rightarrow *bool* \Rightarrow *AST.varDecl* \Rightarrow *var-data* \times *procArg*

where

ppProcArg (*c,v,m,a*) (*VTBounded* *l* *h*) *g*
(*AST.VarDeclNum* *name* *None* *None*) = (
case *lm.lookup* *name* *v* of
Some - \Rightarrow *errv STR "Duplicate variable " name*
| - \Rightarrow (case *lm.lookup* *name* *a* of
Some - \Rightarrow *errv*
STR "Variable name clashes with alias: " name
| - \Rightarrow ((*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 - \Rightarrow *errv STR "Duplicate variable " name*
| - \Rightarrow (case *lm.lookup* *name* *a* of
Some - \Rightarrow *errv*
STR "Variable name clashes with alias: " name
| - \Rightarrow ((*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 - \Rightarrow *errv STR "Duplicate variable " name*
| - \Rightarrow (case *lm.lookup* *name* *a* of
Some - \Rightarrow *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'
⟨proof⟩

```

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* ⇒ *stmt* **where**

```

  incr v = StmtAssign v (ExprBinOp BinOpAdd (ExprVarRef v) (ExprConst 1))

```


definition $decr :: varRef \Rightarrow stmt \text{ where}$

$decr\ v = StmtAssign\ v\ (ExprBinOp\ BinOpSub\ (ExprVarRef\ v)\ (ExprConst\ 1))$

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

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

definition $forFromTo :: varRef \Rightarrow expr \Rightarrow expr \Rightarrow step\ list \Rightarrow stmt \text{ where}$

$forFromTo\ i\ lb\ ub\ steps = ($
let
 — $i = lb$
 $loop\text{-}pre = StepStmt\ (StmtAssign\ i\ lb)\ None;$
 — $i \leq ub$
 $loop\text{-}cond = StepStmt\ (StmtCond\ (ExprBinOp\ BinOpLEq\ (ExprVarRef\ i)\ ub))\ None;$
 — $i++$
 $loop\text{-}incr = StepStmt\ (incr\ i)\ None;$
 — $i \leq ub \rightarrow \dots; i++$
 $loop\text{-}body = loop\text{-}cond\ \# \ steps\ @\ [loop\text{-}incr];$
 — $else \rightarrow break$
 $loop\text{-}abort = [StepStmt\ StmtElse\ None,\ StepStmt\ StmtBreak\ None];$
 — $do :: i \leq ub \rightarrow \dots :: else \rightarrow break\ od$
 $loop = StepStmt\ (StmtDo\ [loop\text{-}body,\ loop\text{-}abort])\ None$
in
 $StmtSeq\ [loop\text{-}pre,\ loop])$

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 $forInArray :: varRef \Rightarrow integer \Rightarrow step\ list \Rightarrow stmt \text{ where}$

$forInArray\ i\ N\ steps = ($
let
 — $i = 0$
 $loop\text{-}pre = StepStmt\ (StmtAssign\ i\ (ExprConst\ 0))\ None;$

```

—  $i < N$ 
loop-cond = StepStmnt (StmntCond
                      (ExprBinOp BinOpLe (Expr VarRef i)
                      (ExprConst N)))
                      None;

—  $i++$ 
loop-incr = StepStmnt (incr i) None;
—  $i < N \rightarrow \dots; i++$ 
loop-body = loop-cond # steps @ [loop-incr];
— else  $\rightarrow$  break
loop-abort = [StepStmnt StmntElse None, StepStmnt StmntBreak None];
— do ::  $i < N \rightarrow \dots$  :: else  $\rightarrow$  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 \Rightarrow chanRef \Rightarrow step list \Rightarrow 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 (Expr VarRef 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 StmtntIf (cvm-fold (cvm-fold ppStep) (False,cvm) sss)
| ppStmtnt (-,cvm) (AST.StmntDo sss) =
    apsnd StmtntDo (cvm-fold (cvm-fold ppStep) (False,cvm) sss)

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

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

| ppStmtnt (-,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))
| ppStmtnt (-,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)))

| ppStmtnt (-,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))
| ppStmtnt (-,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
  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 *lpls* = (*String.literal, String.literal*) *lm*
type-synonym *states* = ($\text{— prio: integer} \times \text{edge list}$) *iarray*
type-synonym *channels* = *channel list*

type-synonym *process* =
nat — offset
 \times *edgeIndex* — start
 \times *procArg list* — args
 \times *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'
  ⟨proof⟩

```

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 (ECEExpr 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 \Rightarrow string) \Rightarrow edgeIndex \Rightarrow string **where**
printIndex f (Index pos) = f (integer-of-nat pos)
| *printIndex* - (LabelJump l -) = String.explode l

definition *printEdge* :: (integer \Rightarrow string) \Rightarrow nat \Rightarrow edge \Rightarrow 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  $\lambda x. x @ " \{A\}"$  else id;
    pEff =  $\lambda -. atom (printEffect f (effect e))$ ;
    contStr = case (cond e) of
      ECTrue  $\Rightarrow$  pEff ()
    | ECFalse  $\Rightarrow$  pEff ()
    | ECSend -  $\Rightarrow$  pEff ()
    | ECRecv - -  $\Rightarrow$  pEff ()
    | -  $\Rightarrow atom ("(" @ printCond f (cond e) @ ")")$ 
  in
  f (integer-of-nat indx) @ " ----> " @ tStr @ " => " @ contStr @ pStr)

```

definition *printEdges* :: (integer \Rightarrow string) \Rightarrow states \Rightarrow string list **where**

```

printEdges f es = concat (map ( $\lambda n. map (printEdge f n) (snd (es !! n))$ )
  (rev [0..IArray.length es]))

```

definition *printLabels* :: (integer \Rightarrow string) \Rightarrow labels \Rightarrow string list **where**

```

printLabels f ls = lm.iterate ls ( $\lambda(k,l) res.
  ("Label " @ String.explode k @ ": "
  @ f (integer-of-nat l)) \# res) []$ 
```

fun *printProcesses* :: (integer \Rightarrow string) \Rightarrow program \Rightarrow string list **where**

```

printProcesses f prog = lm.iterate (proc-data prog)
  ( $\lambda(k,idx) res.
    let (-,start,-) = processes prog !! idx in$ 
```



```

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 (HChannel ts)
    ↔ set ts ⊆ Collect varType-inv ∧ length ts ≤ max-array-size
| channel-inv InvChannel ↔ True

lemma varTypes-finite:
  finite (Collect varType-inv)
⟨proof⟩

lemma variables-finite:
  finite (Collect variable-inv)
⟨proof⟩

lemma channels-finite:
  finite (Collect channel-inv)
⟨proof⟩

```

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

```

$EEDecl\ p \Rightarrow \{p\}$
 $| \ - \Rightarrow \{\}$)

lemma *edgeDecls-finite*:
finite (edgeDecls e)
 $\langle proof \rangle$

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

lemma *edgeSet-finite*:
finite (edgeSet s)
 $\langle proof \rangle$

definition *statesDecls* :: *states* \Rightarrow *procVarDecl set* **where**
statesDecls s = \bigcup (*edgeDecls* ' (*edgeSet s*))

definition *statesNames* :: *states* \Rightarrow *String.literal set* **where**
statesNames s = *procVarDeclName* ' *statesDecls s*

lemma *statesNames-finite*:
finite (statesNames s)
 $\langle proof \rangle$

fun *process-names* :: *states* \Rightarrow *process* \Rightarrow *String.literal set* **where**
process-names ss (-, -, args, decls) =
statesNames ss
 \bigcup *procArgName* ' *set args*
 \bigcup *varDeclName* ' *set decls*
 $\bigcup \{STR\ \ "-"\ , STR\ \ "--assert--"\ , STR\ \ "-pid"\ \}$

lemma *process-names-finite*:
finite (process-names ss p)
 $\langle proof \rangle$

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

lemma *vardicts-finite*:
finite (Collect (vardict-inv ss p))
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

4.3 Invariants of a process

definition $pState\text{-}inv :: program \Rightarrow pState \Rightarrow bool$ **where**

$pState\text{-}inv\ prog\ p$
 $\longleftrightarrow pid\ p \leq max\text{-}procs$
 $\wedge pState.idx\ p < IArray.length\ (states\ prog)$
 $\wedge IArray.length\ (states\ prog) = IArray.length\ (processes\ prog)$
 $\wedge pc\ p < IArray.length\ ((states\ prog) !! pState.idx\ p)$
 $\wedge set\ (pState.channels\ p) \subseteq \{-1..<integer\text{-}of\text{-}nat\ max\text{-}channels\}$
 $\wedge length\ (pState.channels\ p) \leq max\text{-}channels$
 $\wedge vardict\text{-}inv\ ((states\ prog) !! pState.idx\ p)$
 $\quad ((processes\ prog) !! pState.idx\ p)$
 $\quad (pState.vars\ p)$

lemma $pStates\text{-}finite$:

$finite\ (Collect\ (pState\text{-}inv\ prog))$
 $\langle proof \rangle$

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 $cl\text{-}inv :: ('a\ gState\text{-}scheme * pState) \Rightarrow bool$ **where**

$cl\text{-}inv\ gp = (case\ gp\ of\ (g,p) \Rightarrow$
 $length\ (pState.channels\ p) \leq length\ (gState.channels\ g))$

lemma $cl\text{-}inv\text{-}lengthD$:

$cl\text{-}inv\ (g,p) \Longrightarrow length\ (pState.channels\ p) \leq length\ (gState.channels\ g)$
 $\langle proof \rangle$

lemma $cl\text{-}invI$:

$length\ (pState.channels\ p) \leq length\ (gState.channels\ g) \Longrightarrow cl\text{-}inv\ (g,p)$
 $\langle proof \rangle$

lemma $cl\text{-}inv\text{-}trans$:

$length\ (channels\ g) \leq length\ (channels\ g') \Longrightarrow cl\text{-}inv\ (g,p) \Longrightarrow cl\text{-}inv\ (g',p)$
 $\langle proof \rangle$

lemma $cl\text{-}inv\text{-}vars\text{-}update[intro!]$:

$cl\text{-}inv\ (g,p) \Longrightarrow cl\text{-}inv\ (g,\ pState.vars\text{-}update\ vs\ p)$
 $cl\text{-}inv\ (g,p) \Longrightarrow cl\text{-}inv\ (gState.vars\text{-}update\ vs\ g,\ p)$
 $\langle proof \rangle$

lemma $cl\text{-}inv\text{-}handshake\text{-}update[intro!]$:

$cl\text{-}inv\ (g,p) \Longrightarrow cl\text{-}inv\ (g(\handshake := h),p)$
 $\langle proof \rangle$

lemma $cl\text{-}inv\text{-}hsdata\text{-}update[intro!]$:

$cl\text{-}inv\ (g,p) \Longrightarrow cl\text{-}inv\ (g(\hsdata := h),p)$
 $\langle proof \rangle$

lemma *cl-inv-procs-update*[intro]:
 $cl\text{-}inv (g,p) \implies cl\text{-}inv (g(\text{procs} := ps),p)$
 <proof>

lemma *cl-inv-channels-update*:
assumes $cl\text{-}inv (g,p)$
shows $cl\text{-}inv (gState.channels\text{-}update (\lambda cs. cs[i:=c]) g, p)$
 <proof>

4.4 Invariants of the global state

Note that $gState\text{-}inv$ must be defined in a way to be applicable to both $gState$ and $gState_I$.

definition $gState\text{-}inv :: program \Rightarrow 'a\ gState\text{-}scheme \Rightarrow bool$ **where**
 $gState\text{-}inv\ prog\ g$
 $\iff length\ (procs\ g) \leq max\text{-}procs$
 $\wedge (\forall p \in set\ (procs\ g). pState\text{-}inv\ prog\ p \wedge cl\text{-}inv\ (g,p))$
 $\wedge length\ (channels\ g) \leq max\text{-}channels$
 $\wedge set\ (channels\ g) \subseteq Collect\ channel\text{-}inv$
 $\wedge lm.ball\ (vars\ g)\ (\lambda(k,v). variable\text{-}inv\ v)$

The set of global states adhering to the terms of $gState\text{-}inv$ 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 $gState\text{-}progress\text{-}rel :: program \Rightarrow ('a\ gState\text{-}scheme)\ rel$ **where**
 $gState\text{-}progress\text{-}rel\ prog = \{(g,g'). gState\text{-}inv\ prog\ g \wedge gState\text{-}inv\ prog\ g'$
 $\wedge length\ (channels\ g) \leq length\ (channels\ g')$
 $\wedge dom\ (lm.\alpha\ (vars\ g)) = dom\ (lm.\alpha\ (vars\ g'))\}$

lemma *gState-progress-rel-gState-invI1*[intro]:
 $(g,g') \in gState\text{-}progress\text{-}rel\ prog \implies gState\text{-}inv\ prog\ g$
 <proof>

lemma *gState-progress-rel-gState-invI2*[intro]:
 $(g,g') \in gState\text{-}progress\text{-}rel\ prog \implies gState\text{-}inv\ prog\ g'$
 <proof>

lemma *gState-progress-relI*:
assumes $gState\text{-}inv\ prog\ g$
and $gState\text{-}inv\ prog\ g'$
and $length\ (channels\ g) \leq length\ (channels\ g')$
and $dom\ (lm.\alpha\ (vars\ g)) = dom\ (lm.\alpha\ (vars\ g'))$
shows $(g,g') \in gState\text{-}progress\text{-}rel\ prog$
 <proof>

lemma *gState-progress-refl*[simp,intro]:

$gState\text{-}inv\ prog\ g \implies (g,g) \in (gState\text{-}progress\text{-}rel\ prog)$
 $\langle proof \rangle$

lemma *refl-on-gState-progress-rel*:
 $refl\text{-}on\ (Collect\ (gState\text{-}inv\ prog))\ (gState\text{-}progress\text{-}rel\ prog)$
 $\langle proof \rangle$

lemma *trans-gState-progress-rel[simp]*:
 $trans\ (gState\text{-}progress\text{-}rel\ prog)$
 $\langle proof \rangle$

lemmas $gState\text{-}progress\text{-}rel\text{-}trans\ [trans] = trans\text{-}gState\text{-}progress\text{-}rel\ [THEN\ transD]$

lemma *gState-progress-rel-transcl-id[simp]*:
 $(gState\text{-}progress\text{-}rel\ prog)^+ = gState\text{-}progress\text{-}rel\ prog$
 $\langle proof \rangle$

lemma *gState-progress-rel-rtranscl-absorb*:
assumes $gState\text{-}inv\ prog\ g$
shows $(gState\text{-}progress\text{-}rel\ prog)^* \text{ “ } \{g\} = gState\text{-}progress\text{-}rel\ prog \text{ “ } \{g\}$
 $\langle proof \rangle$

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

lemma *gStates-finite*:
fixes $g :: gState$
shows $finite\ ((gState\text{-}progress\text{-}rel\ prog)^* \text{ “ } \{g\})$
 $\langle proof \rangle$

lemma *gState-progress-rel-channels-update*:
assumes $gState\text{-}inv\ prog\ g$
and $channel\text{-}inv\ c$
and $i < length\ (channels\ g)$
shows $(g, gState.channels\text{-}update\ (\lambda cs. cs[i:=c])\ g) \in gState\text{-}progress\text{-}rel\ prog$
 $\langle proof \rangle$

lemma *gState-progress-rel-channels-update-step*:
assumes $gState\text{-}inv\ prog\ g$
and $step: (g, g') \in gState\text{-}progress\text{-}rel\ prog$
and $channel\text{-}inv\ c$
and $i < length\ (channels\ g')$
shows $(g, gState.channels\text{-}update\ (\lambda cs. cs[i:=c])\ g') \in gState\text{-}progress\text{-}rel\ prog$
 $\langle proof \rangle$

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
 \longleftrightarrow $IArray.length (states\ prog) > 0$
 $\wedge IArray.length (states\ prog) = IArray.length (processes\ prog)$
 $\wedge (\forall s \in set (IArray.list-of (states\ prog)). IArray.length\ s > 0)$
 $\wedge lm.ball (proc-data\ prog)$
 $(\lambda(-,sidx).$
 $sidx < IArray.length (processes\ prog)$
 $\wedge fst (processes\ prog !!\ sidx) = sidx)$
 $\wedge (\forall (sidx,start,procArgs,args) \in set (IArray.list-of (processes\ prog)).$
 $(\exists s. start = Index\ s \wedge s < IArray.length (states\ prog !!\ sidx)))$

lemma *program-inv-length-states*:

assumes *program-inv prog*
and $n < IArray.length (states\ prog)$
shows $IArray.length (states\ prog !!\ n) > 0$
 $\langle proof \rangle$

lemma *program-invI*:

assumes $0 < IArray.length (states\ prog)$
and $IArray.length (states\ prog) = IArray.length (processes\ prog)$
and $\bigwedge s. s \in set (IArray.list-of (states\ prog))$
 $\implies 0 < IArray.length\ s$
and $\bigwedge sidx. sidx \in ran (lm.\alpha (proc-data\ prog))$
 $\implies sidx < IArray.length (processes\ prog)$
 $\wedge fst (processes\ prog !!\ sidx) = sidx$
and $\bigwedge sidx\ start\ procArgs\ args.$
 $(sidx,start,procArgs,args) \in set (IArray.list-of (processes\ prog))$
 $\implies \exists s. start = Index\ s \wedge s < IArray.length (states\ prog !!\ sidx)$
shows *program-inv prog*
 $\langle proof \rangle$

end

5 Formalization of Promela semantics

theory *Promela*

imports

PromelaDatastructures

PromelaInvariants

PromelaStatistics

begin

Auxiliary

lemma *mod-integer-le*:

$\langle x\ mod\ (a + 1) \leq b \rangle$ **if** $\langle a \leq b \rangle$ $\langle 0 < a \rangle$ **for** $a\ b\ x :: integer$
 $\langle proof \rangle$ **including** *integer.lifting* $\langle proof \rangle$

lemma *mod-integer-ge*:

$\langle b \leq x \text{ mod } (a + 1) \rangle$ **if** $\langle b \leq 0 \rangle \langle 0 < a \rangle$ **for** $a \ b \ x :: \text{integer}$
 $\langle \text{proof} \rangle$ **including** *integer.lifting* $\langle \text{proof} \rangle$

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* \Rightarrow *labels* \Rightarrow *nat* \Rightarrow *labels* **where**
add-label *l* *lbls* *pos* = (
 case *lm.lookup* *l* *lbls* *of*
 None \Rightarrow *lm.update* *l* *pos* *lbls*
 | *Some* - \Rightarrow *abortv* *STR* "Label given twice: " *l* (λ -. *lbls*)

definition *min-prio* :: *edge list* \Rightarrow *integer* \Rightarrow *integer* **where**
min-prio *es* *start* = *Min* ((*prio* ' *set* *es*) \cup {*start*})

lemma *min-prio-code* [*code*]:
min-prio *es* *start* = *fold* ($\lambda e \ \text{pri.}$ *if* *prio* *e* < *pri* *then* *prio* *e* *else* *pri*) *es* *start*
 $\langle \text{proof} \rangle$

definition *for-all* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a list* \Rightarrow *bool* **where**
for-all *f* *xs* \longleftrightarrow ($\forall x \in \text{set } xs. f \ x$)

lemma *for-all-code*[*code*]:
for-all *f* *xs* \longleftrightarrow *foldli* *xs* *id* ($\lambda kv \ \sigma. f \ kv$) *True*
 $\langle \text{proof} \rangle$

definition *find-remove* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a list* \Rightarrow *'a option* \times *'a list* **where**
find-remove *P* *xs* = (*case* *List.find* *P* *xs* *of* *None* \Rightarrow (*None*, *xs*)
 | *Some* *x* \Rightarrow (*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*)
 $\langle \text{proof} \rangle$

lemma *find-remove-subset*:
find-remove *P* *xs* = (*res*, *xs'*) \Longrightarrow *set* *xs'* \subseteq *set* *xs*
 $\langle \text{proof} \rangle$

lemma *find-remove-length*:
find-remove *P* *xs* = (*res*, *xs'*) \Longrightarrow *length* *xs'* \leq *length* *xs*
 $\langle \text{proof} \rangle$

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)
⟨proof⟩

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

lemma checkVarValue-bounded:
  checkVarValue type val ∈ {min-var-value..max-var-value}
⟨proof⟩

lemma checkVarValue-bounds:
  min-var-value ≤ checkVarValue type val
  checkVarValue type val ≤ max-var-value
⟨proof⟩

lemma checkVarValue-Var:
  variable-inv (Var type (checkVarValue type val))
⟨proof⟩

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⟩

```

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))

```

<proof>

lemma *setVar'-gState-progress-rel:*

assumes *gState-inv prog g*

shows $(g, \text{fst } (\text{setVar}' \text{ gl } v \text{ idx val } g \text{ p})) \in \text{gState-progress-rel prog}$

<proof>

lemma *vardict-inv-process-names:*

assumes *vardict-inv ss proc v*

and $\text{lm.lookup } k \ v = \text{Some } x$

shows $k \in \text{process-names ss proc}$

<proof>

lemma *vardict-inv-variable-inv:*

assumes *vardict-inv ss proc v*

and $\text{lm.lookup } k \ v = \text{Some } x$

shows *variable-inv x*

<proof>

lemma *vardict-inv-updateI:*

assumes *vardict-inv ss proc vs*

and $x \in \text{process-names ss proc}$

and *variable-inv v*

shows *vardict-inv ss proc (lm.update x v vs)*

<proof>

lemma *update-vardict-inv:*

assumes *vardict-inv ss proc v*

and $\text{lm.lookup } k \ v = \text{Some } x$

and *variable-inv x'*

shows *vardict-inv ss proc (lm.update k x' v)*

<proof>

lemma *setVar'-pState-inv:*

assumes *pState-inv prog p*

shows $\text{pState-inv prog } (\text{snd } (\text{setVar}' \text{ gl } v \text{ idx val } g \text{ p}))$

<proof>

lemma *setVar'-cl-inv:*

assumes *cl-inv (g,p)*

shows *cl-inv (setVar' gl v idx val g p)*

<proof>

definition *withVar'*

$:: \text{bool} \Rightarrow \text{String.literal} \Rightarrow \text{integer option}$

$\Rightarrow (\text{integer} \Rightarrow 'x)$

$\Rightarrow 'a \text{ gState-scheme} \Rightarrow \text{pState}$

$\Rightarrow 'x$

where

$withVar' gl v idx f g p = f (the (getVar' gl v idx g p))$

definition $withChannel'$

$:: bool \Rightarrow String.literal \Rightarrow integer\ option$
 $\Rightarrow (nat \Rightarrow channel \Rightarrow 'x)$
 $\Rightarrow 'a\ gState\ scheme \Rightarrow pState$
 $\Rightarrow 'x$

where

$withChannel' gl v idx f g p = ($
 $let\ error = \lambda-. abortv\ STR\ "Variable\ is\ not\ a\ channel:\ " v$
 $(\lambda-. f\ 0\ InvChannel)\ in$
 $let\ abort = \lambda-. abortv\ STR\ "Channel\ already\ closed\ /\ invalid:\ " v$
 $(\lambda-. f\ 0\ InvChannel)$
 $in\ withVar' gl v idx (\lambda i. let\ i = nat\ of\ integer\ i\ in$
 $if\ i \geq length\ (channels\ g)\ then\ error\ ()$
 $else\ let\ c = channels\ g\ !\ i\ in$
 $case\ c\ of$
 $InvChannel \Rightarrow abort\ ()$
 $| - \Rightarrow f\ i\ c)\ g\ p)$

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 $trivCond\ x \equiv if\ x\ then\ 1\ else\ 0$

fun $exprArith :: 'a\ gState\ scheme \Rightarrow pState \Rightarrow expr \Rightarrow integer$

and $pollCheck :: 'a\ gState\ scheme \Rightarrow pState \Rightarrow channel \Rightarrow recvArg\ list \Rightarrow bool$
 $\Rightarrow bool$

and $recvArgsCheck :: 'a\ gState\ scheme \Rightarrow pState \Rightarrow recvArg\ list \Rightarrow integer\ list$
 $\Rightarrow bool$

where

$exprArith\ g\ p\ (ExprConst\ x) = x$

$| exprArith\ g\ p\ (ExprMConst\ x\ -) = x$

$| exprArith\ g\ p\ ExprTimeOut = trivCond\ (timeout\ g)$

$| exprArith\ g\ p\ (ExprLen\ (ChanRef\ (VarRef\ gl\ name\ None))) =$

$withChannel' gl name None ($

$\lambda- c. case\ c\ of$

$Channel\ -\ -\ q \Rightarrow integer\ of\ nat\ (length\ q)$

$| HChannel\ - \Rightarrow 0)\ g\ p$

$| exprArith\ g\ p\ (ExprLen\ (ChanRef\ (VarRef\ gl\ name\ (Some\ idx)))) =$

$withChannel' gl name (Some\ (exprArith\ g\ p\ idx)) ($

$\lambda- c. case\ c\ of$

$Channel\ -\ -\ q \Rightarrow integer\ of\ nat\ (length\ q)$

$| \text{HSChannel} - \Rightarrow 0) \ g \ p$
 $| \text{exprArith } g \ p \ (\text{ExprEmpty } (\text{ChanRef } (\text{VarRef } \text{gl name None}))) =$
 $\quad \text{trivCond } (\text{withChannel}' \ \text{gl name None } ($
 $\quad \lambda \text{- c. case } c \text{ of Channel - - } q \Rightarrow (q = [])$
 $\quad | \text{HSChannel} - \Rightarrow \text{True}) \ g \ p)$
 $| \text{exprArith } g \ p \ (\text{ExprEmpty } (\text{ChanRef } (\text{VarRef } \text{gl name (Some idx)}))) =$
 $\quad \text{trivCond } (\text{withChannel}' \ \text{gl name (Some (exprArith } g \ p \ \text{idx})) } ($
 $\quad \lambda \text{- c. case } c \text{ of Channel - - } q \Rightarrow (q = [])$
 $\quad | \text{HSChannel} - \Rightarrow \text{True}) \ g \ p)$
 $| \text{exprArith } g \ p \ (\text{ExprFull } (\text{ChanRef } (\text{VarRef } \text{gl name None}))) =$
 $\quad \text{trivCond } (\text{withChannel}' \ \text{gl name None } ($
 $\quad \lambda \text{- c. case } c \text{ of}$
 $\quad \quad \text{Channel } \text{cap} - q \Rightarrow \text{integer-of-nat } (\text{length } q) \geq \text{cap}$
 $\quad | \text{HSChannel} - \Rightarrow \text{False}) \ g \ p)$
 $| \text{exprArith } g \ p \ (\text{ExprFull } (\text{ChanRef } (\text{VarRef } \text{gl name (Some idx)}))) =$
 $\quad \text{trivCond } (\text{withChannel}' \ \text{gl name (Some (exprArith } g \ p \ \text{idx})) } ($
 $\quad \lambda \text{- c. case } c \text{ of}$
 $\quad \quad \text{Channel } \text{cap} - q \Rightarrow \text{integer-of-nat } (\text{length } q) \geq \text{cap}$
 $\quad | \text{HSChannel} - \Rightarrow \text{False}) \ g \ p)$
 $| \text{exprArith } g \ p \ (\text{ExprVarRef } (\text{VarRef } \text{gl name None})) =$
 $\quad \text{withVar}' \ \text{gl name None } \text{id } g \ p$
 $| \text{exprArith } g \ p \ (\text{ExprVarRef } (\text{VarRef } \text{gl name (Some idx)})) =$
 $\quad \text{withVar}' \ \text{gl name (Some (exprArith } g \ p \ \text{idx})) } \ \text{id } g \ p$
 $| \text{exprArith } g \ p \ (\text{ExprUnOp } \text{UnOpMinus } \text{expr}) = 0 - \text{exprArith } g \ p \ \text{expr}$
 $| \text{exprArith } g \ p \ (\text{ExprUnOp } \text{UnOpNeg } \text{expr}) = ((\text{exprArith } g \ p \ \text{expr}) + 1) \ \text{mod } 2$
 $| \text{exprArith } g \ p \ (\text{ExprBinOp } \text{BinOpAdd } \text{lexpr } \text{rexpr}) =$
 $\quad (\text{exprArith } g \ p \ \text{lexpr}) + (\text{exprArith } g \ p \ \text{rexpr})$
 $| \text{exprArith } g \ p \ (\text{ExprBinOp } \text{BinOpSub } \text{lexpr } \text{rexpr}) =$
 $\quad (\text{exprArith } g \ p \ \text{lexpr}) - (\text{exprArith } g \ p \ \text{rexpr})$
 $| \text{exprArith } g \ p \ (\text{ExprBinOp } \text{BinOpMul } \text{lexpr } \text{rexpr}) =$
 $\quad (\text{exprArith } g \ p \ \text{lexpr}) * (\text{exprArith } g \ p \ \text{rexpr})$
 $| \text{exprArith } g \ p \ (\text{ExprBinOp } \text{BinOpDiv } \text{lexpr } \text{rexpr}) =$
 $\quad (\text{exprArith } g \ p \ \text{lexpr}) \ \text{div } (\text{exprArith } g \ p \ \text{rexpr})$
 $| \text{exprArith } g \ p \ (\text{ExprBinOp } \text{BinOpMod } \text{lexpr } \text{rexpr}) =$
 $\quad (\text{exprArith } g \ p \ \text{lexpr}) \ \text{mod } (\text{exprArith } g \ p \ \text{rexpr})$
 $| \text{exprArith } g \ p \ (\text{ExprBinOp } \text{BinOpGr } \text{lexpr } \text{rexpr}) =$

```

    triuCond (exprArith g p levr > exprArith g p rexr)
| exprArith g p (ExprBinOp BinOpLe levr rexr) =
    triuCond (exprArith g p levr < exprArith g p rexr)
| exprArith g p (ExprBinOp BinOpGEq levr rexr) =
    triuCond (exprArith g p levr ≥ exprArith g p rexr)
| exprArith g p (ExprBinOp BinOpLEq levr rexr) =
    triuCond (exprArith g p levr ≤ exprArith g p rexr)
| exprArith g p (ExprBinOp BinOpEq levr rexr) =
    triuCond (exprArith g p levr = exprArith g p rexr)
| exprArith g p (ExprBinOp BinOpNEq levr rexr) =
    triuCond (exprArith g p levr ≠ exprArith g p rexr)
| exprArith g p (ExprBinOp BinOpAnd levr rexr) =
    triuCond (exprArith g p levr ≠ 0 ∧ exprArith g p rexr ≠ 0)
| exprArith g p (ExprBinOp BinOpOr levr rexr) =
    triuCond (exprArith g p levr ≠ 0 ∨ exprArith g p rexr ≠ 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) =
    triuCond (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) =
    triuCond (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 us)

```

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* $v = \text{liftVar } (\lambda gl v idx \text{ arg. } \text{getVar}' gl v idx) v ()$

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 $(g, \text{fst } (\text{setVar } v \text{ val } g p)) \in \text{gState-progress-rel prog}$

<proof>

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))*

<proof>

lemma *setVar-cl-inv*:

assumes *cl-inv (g,p)*

shows *cl-inv (setVar v val g p)*

<proof>

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

```



```

      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)))
  ⟨proof⟩
  including integer.lifting
  ⟨proof⟩

lemma toVariable-channels-inv:
  shows ∀ x ∈ set (snd (snd (toVariable g p v))). channel-inv x
  ⟨proof⟩

lemma toVariable-channels-inv':
  shows toVariable g p v = (a,b,c) ⇒ ∀ x ∈ set c. channel-inv x
  ⟨proof⟩

lemma toVariable-variable-inv':
  shows gState-inv prog g ⇒ toVariable g p v = (a,b,c) ⇒ variable-inv b
  ⟨proof⟩

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
          csp = map integer-of-nat [l..l + length cs];
          g' = g(\channels := channels g @ cs);
          p' = p(\pState.channels := pState.channels p @ csp)
        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
  ⟨proof⟩

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

lemma mkChannels-pState-inv:

```

$pState\text{-}inv\ prog\ p$
 $\implies cl\text{-}inv\ (g,p)$
 $\implies pState\text{-}inv\ prog\ (snd\ (mkChannels\ g\ p\ cs))$
 $\langle proof \rangle$
including *integer.lifting*
 $\langle proof \rangle$

lemma *mkChannels-cl-inv*:
 $cl\text{-}inv\ (g,p) \implies cl\text{-}inv\ (mkChannels\ g\ p\ cs)$
 $\langle proof \rangle$

definition *mkVarChannel*
 $:: varDecl$
 $\implies ((var\text{-}dict \implies var\text{-}dict) \implies 'a\ gState\text{-}scheme * pState$
 $\implies 'a\ gState\text{-}scheme * pState)$
 $\implies 'a\ gState\text{-}scheme \implies pState$
 $\implies 'a\ gState\text{-}scheme * pState$

where
 $mkVarChannel\ v\ upd\ g\ p = ($
 $\quad let$
 $\quad\quad (k,v,cs) = toVariable\ g\ p\ v;$
 $\quad\quad (g',p') = upd\ (lm.update\ k\ v)\ (g,p)$
 $\quad in$
 $\quad\quad mkChannels\ g'\ p'\ cs)$

lemma *mkVarChannel-gState-inv*:
assumes $gState\text{-}inv\ prog\ g$
and $\bigwedge k\ v'\ cs. toVariable\ g\ p\ v = (k,v',cs)$
 $\implies gState\text{-}inv\ prog\ (fst\ (upd\ (lm.update\ k\ v')\ (g,p)))$
shows $gState\text{-}inv\ prog\ (fst\ (mkVarChannel\ v\ upd\ g\ p))$
 $\langle proof \rangle$

lemma *mkVarChannel-gState-progress-rel*:
assumes $gState\text{-}inv\ prog\ g$
and $\bigwedge k\ v'\ cs. toVariable\ g\ p\ v = (k,v',cs)$
 $\implies (g, fst\ (upd\ (lm.update\ k\ v')\ (g,p))) \in gState\text{-}progress\text{-}rel\ prog$
shows $(g, fst\ (mkVarChannel\ v\ upd\ g\ p)) \in gState\text{-}progress\text{-}rel\ prog$
 $\langle proof \rangle$

lemma *mkVarChannel-pState-inv*:
assumes $pState\text{-}inv\ prog\ p$
and $cl\text{-}inv\ (g,p)$
and $\bigwedge k\ v'\ cs. toVariable\ g\ p\ v = (k,v',cs)$
 $\implies cl\text{-}inv\ (upd\ (lm.update\ k\ v')\ (g,p))$
and $\bigwedge k\ v'\ cs. toVariable\ g\ p\ v = (k,v',cs)$
 $\implies pState\text{-}inv\ prog\ (snd\ (upd\ (lm.update\ k\ v')\ (g,p)))$
shows $pState\text{-}inv\ prog\ (snd\ (mkVarChannel\ v\ upd\ g\ p))$
 $\langle proof \rangle$

lemma *mkVarChannel-cl-inv*:
assumes *cl-inv* (*g,p*)
and $\bigwedge k v' cs. \text{toVariable } g \ p \ v = (k, v', cs)$
 $\implies \text{cl-inv } (\text{upd } (\text{lm.update } k \ v') \ (g,p))$
shows *cl-inv* (*mkVarChannel v upd g p*)
 $\langle \text{proof} \rangle$

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

lemma *mkVarChannelProc-gState-progress-rel*:
assumes *gState-inv prog g*
shows (*g, fst (mkVarChannelProc v g p)*) \in *gState-progress-rel prog*
 $\langle \text{proof} \rangle$

lemmas *mkVarChannelProc-gState-inv =*
 $\text{mkVarChannelProc-gState-progress-rel}[\text{THEN } \text{gState-progress-rel-gState-invI2}]$

lemma *toVariable-name*:
 $\text{toVariable } g \ p \ (\text{VarDeclNum } lb \ hb \ name \ sz \ init) = (x, a, b) \implies x = name$
 $\text{toVariable } g \ p \ (\text{VarDeclChan } name \ sz \ t) = (x, a, b) \implies x = name$
 $\langle \text{proof} \rangle$

declare *toVariable.simps[simp del]*

lemma *statesDecls-process-names*:
assumes $v \in \text{statesDecls } (\text{states } \text{prog} \ !! \ (\text{pState.idx } p))$
shows $\text{procVarDeclName } v \in \text{process-names } (\text{states } \text{prog} \ !! \ (\text{pState.idx } p))$
 $(\text{processes } \text{prog} \ !! \ (\text{pState.idx } p))$
 $\langle \text{proof} \rangle$

lemma *mkVarChannelProc-pState-inv*:
assumes *pState-inv prog p*
and *gState-inv prog g*
and *cl-inv* (*g, p*)
and *decl*: $v \in \text{statesDecls } (\text{states } \text{prog} \ !! \ (\text{pState.idx } p))$
shows *pState-inv prog (snd (mkVarChannelProc v g p))*
 $\langle \text{proof} \rangle$

lemma *mkVarChannelProc-cl-inv*:
assumes *cl-inv* (*g,p*)
shows *cl-inv* (*mkVarChannelProc v g p*)
⟨*proof*⟩

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*
 $step\text{-}fold' g steps (lbls :: labels) pri pos$
 $(nxt :: edgeIndex) (onxt :: edgeIndex option) iB =$
 $foldr (\lambda step (pos, nxt, lbls, es).$
 $let (e, enxt, lbls) = g step (lbls, pri, pos, nxt, onxt, iB)$
 $in (pos + length e, enxt, lbls, es @ e)$
 $) steps (pos, nxt, lbls, [])$

definition *step-fold where*
 $step\text{-}fold g steps lbls pri pos nxt onxt iB = ($
 $let (-,nxt,lbls,s) = step\text{-}fold' g steps lbls pri pos nxt onxt iB$
 $in (s,nxt,lbls))$

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 $\bigwedge x d. x \in set\ steps \implies g\ x\ d = g'\ x\ d$
shows $step\text{-}fold' g steps lbls pri pos nxt onxt iB =$
 $step\text{-}fold' g' steps' lbls' pri' pos' nxt' onxt' iB'$
⟨*proof*⟩

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 $\bigwedge x d. x \in set\ steps \implies g\ x\ d = g'\ x\ d$
shows $step\text{-}fold g steps lbls pri pos nxt onxt iB =$
 $step\text{-}fold g' steps' lbls' pri' pos' nxt' onxt' iB'$
⟨*proof*⟩


```

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

```

```

definition step-foldL where
  step-foldL g stepss lbls pri pos next onxt =
    foldr (step-foldL-step g pri onxt) stepss (pos,next,lbls,[],[])

```

```

lemma step-foldL-step-cong:
  assumes pri = pri'
  and onxt = onxt'
  and s = s'
  and d = d'
  and  $\bigwedge x d. x \in \text{set } s \implies g x d = g' x d$ 
  shows step-foldL-step g pri onxt s d = step-foldL-step g' pri' onxt' s' d'
  <proof>

```

```

lemma step-foldL-cong[fundef-cong]:
  assumes lbls = lbls'
  and pri = pri'
  and pos = pos'
  and stepss = stepss'
  and next = next'
  and onxt = onxt'
  and  $\bigwedge x x' d. x \in \text{set } stepss \implies x' \in \text{set } x \implies g x' d = g' x' d$ 
  shows step-foldL g stepss lbls pri pos next onxt =
    step-foldL g' stepss' lbls' pri' pos' next' onxt'
  <proof>

```

5.6 Starting processes

```

definition modProcArg
  :: (procArg * integer)  $\Rightarrow$  String.literal * variable
where
  modProcArg x = (
    case x of
      (ProcArg ty name, val)  $\Rightarrow$  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 ( $\lambda$ -. (name, Var VTChan 0)))

```

```

definition emptyProc :: pState
  — The empty process.

```

where

$emptyProc = (\text{pid} = 0, \text{vars} = \text{lm.empty } (), \text{pc} = 0, \text{channels} = [], \text{idx} = 0)$

lemma *vardict-inv-empty*:

$vardict\text{-}inv\ ss\ proc\ (\text{lm.empty}())$
 $\langle proof \rangle$

lemma *emptyProc-cl-inv[simp]*:

$cl\text{-}inv\ (g, emptyProc)$
 $\langle proof \rangle$

lemma *emptyProc-pState-inv*:

assumes *program-inv prog*
shows *pState-inv prog emptyProc*
 $\langle proof \rangle$

fun *mkProc*

$:: 'a\ gState\text{-}scheme \Rightarrow pState$
 $\Rightarrow String.literal \Rightarrow expr\ list \Rightarrow process \Rightarrow nat$
 $\Rightarrow 'a\ gState\text{-}scheme * pState$

where

$mkProc\ g\ p\ name\ args\ (sidx, start, argDecls, decls)\ pidN = ($
 $let\ start = case\ start\ of$
 $Index\ x \Rightarrow x$
 $| - \Rightarrow abortv\ STR\ "Process\ start\ is\ not\ index:\ " name\ (\lambda_. 0)$

in

— sanity check

if $length\ args \neq length\ argDecls$

then $abortv\ STR\ "Signature\ mismatch:\ " name\ (\lambda_. (g, emptyProc))$

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

$pidI = integer\text{-}of\text{-}nat\ pidN;$

$argVars = (STR\ "-pid", Var\ (VTBounded\ 0\ pidI)\ pidI)\#\ argVars;$

$argVars = lm.to\text{-}map\ argVars;$

 — our new process

$p = (\text{pid} = pidN, \text{vars} = argVars, \text{pc} = start, \text{channels} = [], \text{idx} = sidx)$

in

 — apply the declarations

$foldl\ (\lambda(g,p)\ d.\ mkVarChannel\ d\ (apsnd\ \circ\ pState.vars\text{-}update)\ g\ p)$

(g,p)

$decls)$

lemma *mkProc-gState-progress-rel*:
assumes *gState-inv prog g*
shows $(g, \text{fst } (\text{mkProc } g \ p \ \text{name } \text{args } (\text{processes } \text{prog} \ !! \ \text{sid}x) \ \text{pid}N)) \in$
gState-progress-rel prog
 $\langle \text{proof} \rangle$

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 $\text{pid}N \leq \text{max-procs}$ **and** $\text{pid}N > 0$
and $\text{sid}x < \text{IArray.length } (\text{processes } \text{prog})$
and $\text{fst } (\text{processes } \text{prog} \ !! \ \text{sid}x) = \text{sid}x$
shows *pState-inv prog (snd (mkProc g p name args (processes prog !! sid) pidN))*
 $\langle \text{proof} \rangle$
including *integer.lifting*
 $\langle \text{proof} \rangle$

lemma *mkProc-cl-inv*:
assumes *cl-inv (g,p)*
shows *cl-inv (mkProc g p name args (processes prog !! sid) pidN)*
 $\langle \text{proof} \rangle$

declare *mkProc.simps [simp del]*

definition *runProc*
 $:: \text{String.literal} \Rightarrow \text{expr list} \Rightarrow \text{program}$
 $\Rightarrow 'a \ \text{gState-scheme} \Rightarrow \text{pState}$
 $\Rightarrow 'a \ \text{gState-scheme} * \text{pState}$

where

runProc name args prog g p = (
if length (procs g) \geq max-procs
then abort STR "Too many processes" $(\lambda-. (g,p))$
else let pid = length (procs g) + 1 in
case lm.lookup name (proc-data prog) of
None \Rightarrow abortv STR "No such process: " name
 $(\lambda-. (g,p))$
 $| \text{Some proc-idx} \Rightarrow$
 $\text{let } (g', \text{proc}) = \text{mkProc } g \ p \ \text{name } \text{args } (\text{processes } \text{prog} \ !! \ \text{proc-idx}) \ \text{pid}$
 $\text{in } (g' \ @ \ [\text{procs}], \ p))$

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 $(g, \text{fst } (\text{runProc name args prog } g \ p)) \in \text{gState-progress-rel prog}$
 $\langle \text{proof} \rangle$

lemmas $\text{runProc-gState-inv} =$
 $\text{runProc-gState-progress-rel}[\text{THEN gState-progress-rel-gState-invI2}]$

lemma runProc-pState-id :
 $\text{snd } (\text{runProc name args prog } g \ p) = p$
 $\langle \text{proof} \rangle$

lemma $\text{runProc-pState-inv}$:
assumes $p\text{State-inv prog } p$
shows $p\text{State-inv prog } (\text{snd } (\text{runProc name args prog } g \ p))$
 $\langle \text{proof} \rangle$

lemma runProc-cl-inv :
assumes program-inv prog
and $\text{gState-inv prog } g$
and $\text{pState-inv prog } p$
and $\text{cl-inv } (g, p)$
shows $\text{cl-inv } (\text{runProc name args prog } g \ p)$
 $\langle \text{proof} \rangle$

5.7 AST to edges

type-synonym $\text{ast} = \text{AST.module list}$

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 $\text{atomize} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{edge list} \Rightarrow \text{edge list}$ **where**

$\text{atomize } lp \ hp \ es = \text{fold } (\lambda e \ es.$

$\text{let } e' = \text{case target } e \text{ of}$

$\text{LabelJump - None} \Rightarrow$

— 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.

$e \langle \text{atomic} := \text{InAtomic} \rangle$

| $\text{LabelJump - (Some via)} \Rightarrow$

$\text{if } lp \leq \text{via} \wedge hp \geq \text{via} \text{ then } e \langle \text{atomic} := \text{Atomic} \rangle$

$\text{else } e \langle \text{atomic} := \text{InAtomic} \rangle$

| $\text{Index } p' \Rightarrow$

$\text{if } lp \leq p' \wedge hp \geq p' \text{ then } e \langle \text{atomic} := \text{Atomic} \rangle$

$\text{else } e \langle \text{atomic} := \text{InAtomic} \rangle$

in $e' \# es$) es []

fun *skip* — No-(edge)

where

skip (*lbls*, *pri*, *pos*, *nxt*, -) =
 ([[$\{$ cond = *ECE*Expr (*Expr*Const 1),
 effect = *EE*Id, 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 *stmtToState*

 :: *stmtnt*

 ⇒ (*labels* * *integer* * *nat* * *edgeIndex* * *edgeIndex* option * *bool*)

 ⇒ *edge list list* * *edgeIndex* * *labels*

where

stepToState (*StepStmnt* *s* *None*) *data* = *stmtToState* *s* *data*

| *stepToState* (*StepStmnt* *s* (*Some* *u*)) (*lbls*, *pri*, *pos*, *nxt*, *onxt*, -) = (
 let

 — the *unless* part

 (*ues*, -, *lbls'*) = *stmtToState* *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''*) = *stmtToState* *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)

| stmtToState (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'))

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

| stmtToState (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)
  )

| stmtToState (StmntIf stepss) (lbls, pri, pos, nxt, onxt, -) = (

```

$let (pos, -, lbls, es, is) = step-foldL stepToState steps lbls pri pos nxt onxt$
 $in (es @ [concat is], Index pos, lbls)$

$| stmtnToState (StmtntSeq steps) (lbls, pri, pos, nxt, onxt, inBlock) =$
 $step-fold stepToState steps lbls pri pos nxt onxt inBlock$

$| stmtnToState (StmtntAssign v e) (lbls, pri, pos, nxt, -) =$
 $([[[cond = ECTrue, effect = EEAssign v e, target = nxt, prio = pri,$
 $atomic = NonAtomic]]], Index pos, lbls)$

$| stmtnToState (StmtntAssert e) (lbls, pri, pos, nxt, -) =$
 $([[[cond = ECTrue, effect = EEAssert e, target = nxt, prio = pri,$
 $atomic = NonAtomic]]], Index pos, lbls)$

$| stmtnToState (StmtntCond e) (lbls, pri, pos, nxt, -) =$
 $([[[cond = ECExpr e, effect = EEId, target = nxt, prio = pri,$
 $atomic = NonAtomic]]], Index pos, lbls)$

$| stmtnToState StmtntElse (lbls, pri, pos, nxt, -) =$
 $([[[cond = ECElse, effect = EEId, target = nxt, prio = pri,$
 $atomic = NonAtomic]]], Index pos, lbls)$

$| stmtnToState StmtntBreak (lbls, pri, -, -, Some onxt, -) =$
 $([[[cond = ECTrue, effect = EEGoto, target = onxt, prio = pri,$
 $atomic = NonAtomic]]], onxt, lbls)$

$| stmtnToState StmtntBreak (-, -, -, None, -) =$
 $abort STR "Misplaced break" (\lambda-. ([], Index 0, lm.empty()))$

$| stmtnToState (StmtntRun n args) (lbls, pri, pos, nxt, onxt, -) =$
 $([[[cond = ECRun n, effect = EERun n args, target = nxt, prio = pri,$
 $atomic = NonAtomic]]], Index pos, lbls)$

$| stmtnToState (StmtntGoTo l) (lbls, pri, pos, -) =$
 $([[[cond = ECTrue, effect = EEGoto, target = LabelJump l None, prio = pri,$
 $atomic = NonAtomic]]], LabelJump l (Some pos), lbls)$

$| stmtnToState (StmtntSend v e srt) (lbls, pri, pos, nxt, -) =$
 $([[[cond = ECSend v, effect = EESend v e srt, target = nxt, prio = pri,$
 $atomic = NonAtomic]]], Index pos, lbls)$

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

$| stmtnToState StmtntSkip d = skip d$

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* = λ*pos*. fold (λ*e 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 (λ*es*. (*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 \Rightarrow
 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 $\exists s. \text{start} = \text{Index } s \wedge s < \text{IArray.length } ss$
and *IArray.length ss > 0*
<proof>

primrec *toProcess*
 $:: \text{nat} \Rightarrow \text{proc} \Rightarrow \text{states} * \text{nat} * \text{String.literal} * (\text{labels} * \text{process})$
where
toProcess sidx (ProcType act name args decls steps) = (
 let
 (states, start, lbls) = toStates steps;
 act = (case act of
 None \Rightarrow *0*
 | Some None \Rightarrow *1*
 | Some (Some x) \Rightarrow *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) \Longrightarrow idx = sidx
<proof>

lemma *toProcess-states-nonempty:*
toProcess sidx p = (ss,a,n,l,idx,r) \Longrightarrow IArray.length ss > 0
<proof>

lemma *toProcess-start:*
toProcess sidx p = (ss,a,n,l,idx,start,r)
 $\Longrightarrow \exists s. \text{start} = \text{Index } s \wedge s < \text{IArray.length } ss$
<proof>

lemma *toProcess-startE:*
assumes *toProcess sidx p = (ss,a,n,l,idx,start,r)*
obtains *s where start = Index s s < IArray.length ss*
<proof>

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 (λg d.
      fst (mkVarChannel d (apfst ∘ gState.vars-update) g emptyProc)
    ) g decls;
    g'' = foldl (λg (-, a, name, -).
      foldl (λg 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*⟩

lemma *setUp-program-inv*:
assumes *setUp ast* = (*prog*, *g*)
shows *program-inv prog*
 ⟨*proof*⟩

lemma *setUp-gState-inv*:
assumes *setUp ast* = (*prog*, *g*)
shows *gState-inv prog g*
 ⟨*proof*⟩

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 $assertVar \equiv VarRef\ True\ (STR\ "--assert--")\ None$

5.8.1 Evaluation of Edges

```

fun evalRecvArgs
  :: recvArg list  $\Rightarrow$  integer list  $\Rightarrow$  gStateI  $\Rightarrow$  pState  $\Rightarrow$  gStateI * 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  $\Rightarrow$  setVar var v g l
    | -  $\Rightarrow$  (g,l)
  in evalRecvArgs rs vs g l)

primrec evalCond
  :: edgeCond  $\Rightarrow$  gStateI  $\Rightarrow$  pState  $\Rightarrow$  bool
where
  evalCond ECTrue - -  $\longleftrightarrow$  True
| evalCond ECFalse - -  $\longleftrightarrow$  False
| evalCond (ECEExpr e) g l  $\longleftrightarrow$  exprArith g l e  $\neq$  0
| evalCond (ECRun -) g l  $\longleftrightarrow$  length (procs g) < 255
| evalCond ECElse g l  $\longleftrightarrow$  gStateI.else g
| evalCond (ECSend v) g l  $\longleftrightarrow$ 
  withChannel v (λ- c.
    case c of
      Channel cap - q  $\Rightarrow$  integer-of-nat (length q) < cap
    | HSChannel -  $\Rightarrow$  True) g l
| evalCond (ECRecv v rs srt) g l  $\longleftrightarrow$ 
  withChannel v (λi c.
    case c of
      HSChannel -  $\Rightarrow$  handshake g  $\neq$  0  $\wedge$  recvArgsCheck g l rs (hsdata g)
    | -  $\Rightarrow$  pollCheck g l c rs srt) g l

fun evalHandshake
  :: edgeCond  $\Rightarrow$  nat  $\Rightarrow$  gStateI  $\Rightarrow$  pState  $\Rightarrow$  bool
where
  evalHandshake (ECRecv v - -) h g l
   $\longleftrightarrow$  h = 0
   $\vee$  withChannel v (λi c. case c of

```

$HChannel - \Rightarrow i = h$
 $| Channel - - - \Rightarrow False) g l$
 $| evalHandshake - h - - \longleftrightarrow h = 0$

primrec *evalEffect*

$:: edgeEffect \Rightarrow program \Rightarrow gState_I \Rightarrow pState \Rightarrow gState_I * pState$

where

$evalEffect\ EEEnd - g\ l = (g,l)$
 $| evalEffect\ EEId - g\ l = (g,l)$
 $| evalEffect\ EEGoto - g\ l = (g,l)$
 $| evalEffect\ (EEAssign\ v\ e) - g\ l = setVar\ v\ (exprArith\ g\ l\ e)\ g\ l$
 $| evalEffect\ (EEDecl\ d) - g\ l = mkVarChannelProc\ d\ g\ l$
 $| evalEffect\ (EERun\ name\ args)\ prog\ g\ l = runProc\ name\ args\ prog\ g\ l$
 $| evalEffect\ (EEAssert\ e) - g\ l = ($
 $\quad if\ exprArith\ g\ l\ e = 0$
 $\quad then\ setVar\ assertVar\ 1\ g\ l$
 $\quad else\ (g,l))$
 $| evalEffect\ (EESend\ v\ es\ srt) - g\ l = withChannel\ v\ (\lambda i\ c.$
 $\quad let$
 $\quad\quad ab = \lambda-. abort\ STR\ "Length\ mismatch\ on\ sending." (\lambda-. (g,l));$
 $\quad\quad es = map\ (exprArith\ g\ l)\ es$
 $\quad in$
 $\quad\quad if\ \neg\ for\ all\ (\lambda x. x \geq min\text{-}var\text{-}value \wedge x \leq max\text{-}var\text{-}value)\ es$
 $\quad\quad then\ abort\ STR\ "Invalid\ Channel" (\lambda-. (g,l))$
 $\quad\quad else$
 $\quad\quad\quad case\ c\ of$
 $\quad\quad\quad\quad Channel\ cap\ ts\ q \Rightarrow$
 $\quad\quad\quad\quad\quad if\ length\ ts \neq length\ es \vee \neg\ (length\ q < max\text{-}array\text{-}size)$
 $\quad\quad\quad\quad\quad then\ ab()$
 $\quad\quad\quad\quad\quad else\ let$
 $\quad\quad\quad\quad\quad\quad $q' = if\ \neg\ srt\ then\ q@[es]$$
 $\quad\quad\quad\quad\quad\quad\quad else\ let$
 $\quad\quad\quad\quad\quad\quad\quad\quad $q = map\ lexicst\ q;$$
 $\quad\quad\quad\quad\quad\quad\quad\quad $q' = insert\ (lexicst\ es)\ q$$
 $\quad\quad\quad\quad\quad\quad\quad\quad in\ map\ unlex\ q';$
 $\quad\quad\quad\quad\quad\quad\quad\quad $g = gState.channels\text{-}update\ (\lambda cs.$$
 $\quad\quad\quad\quad\quad\quad\quad\quad\quad $cs[i := Channel\ cap\ ts\ q'])$ $g$$
 $\quad\quad\quad\quad\quad\quad in\ (g,l)$
 $\quad\quad\quad\quad | HChannel\ ts \Rightarrow$
 $\quad\quad\quad\quad\quad if\ length\ ts \neq length\ es\ then\ ab()$
 $\quad\quad\quad\quad\quad else\ (g(|hsdata := es, handshake := i), l)$
 $\quad\quad\quad\quad | InvChannel \Rightarrow abort\ STR\ "Trying\ to\ send\ on\ invalid\ channel" (\lambda-. (g,l))$
 $\quad\quad)\ g\ l$
 $| evalEffect\ (EERecv\ v\ rs\ srt\ rem) - g\ l = withChannel\ v\ (\lambda i\ c.$
 $\quad case\ c\ of$
 $\quad\quad Channel\ cap\ ts\ qs \Rightarrow$
 $\quad\quad\quad if\ qs = []\ then\ abort\ STR\ "Recv\ from\ empty\ channel" (\lambda-. (g,l))$
 $\quad\quad\quad else$
 $\quad\quad\quad\quad let$

```

      (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)
  | HSCchannel - ⇒
    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 $ef \in \text{effect } \text{'edgeSet } ss$
and $ef = EEDecl\ d$
shows $d \in \text{statesDecls } ss$
<proof>

lemma *evalRecvArgs-pState-inv*:
assumes $pState\text{-inv } prog\ p$
shows $pState\text{-inv } prog\ (snd\ (evalRecvArgs\ rargs\ xs\ g\ p))$
<proof>

lemma *evalRecvArgs-pState-inv'*:
assumes $evalRecvArgs\ rargs\ xs\ g\ p = (g', p')$
and $pState\text{-inv } prog\ p$
shows $pState\text{-inv } prog\ p'$
<proof>

lemma *evalRecvArgs-gState-progress-rel*:
assumes $gState\text{-inv } prog\ g$
shows $(g, fst\ (evalRecvArgs\ rargs\ xs\ g\ p)) \in gState\text{-progress-rel } prog$
<proof>

lemmas $evalRecvArgs\text{-gState-inv} =$
 $evalRecvArgs\text{-gState-progress-rel}[THEN\ gState\text{-progress-rel-gState-invI2}]$

lemma *evalRecvArgs-cl-inv*:
assumes $cl\text{-inv } (g,p)$
shows $cl\text{-inv } (evalRecvArgs\ rargs\ xs\ g\ p)$
<proof>

lemma *evalEffect-pState-inv*:
assumes $pState\text{-inv } prog\ p$
and $gState\text{-inv } prog\ g$
and $cl\text{-inv } (g,p)$

and $e \in \text{effect}$ ‘ $\text{edgeSet} (\text{states prog} !! \text{pState.idx } p)$
shows $\text{pState-inv prog} (\text{snd} (\text{evalEffect } e \text{ prog } g \text{ } p))$
 ⟨*proof*⟩

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 $(g, \text{fst} (\text{evalEffect } e \text{ prog } g \text{ } p)) \in \text{gState-progress-rel prog}$
 ⟨*proof*⟩

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)*
 ⟨*proof*⟩

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 $(\lambda E e.$
 if $\text{evalHandshake} (\text{cond } e) (\text{handshake } g) g \text{ } p \wedge \text{evalCond} (\text{cond } e) g \text{ } p$
 then $(e,p)\#E$
 else E) \square

lemma *getChoices-sub-edges-fst*:

fst ‘ $\text{set} (\text{getChoices } g \text{ } p \text{ } es) \subseteq \text{set } es$
 ⟨*proof*⟩

lemma *getChoices-sub-edges*:

$(a,b) \in \text{set} (\text{getChoices } g \text{ } p \text{ } es) \implies a \in \text{set } es$
 ⟨*proof*⟩

lemma *getChoices-p-snd*:

snd ‘ $\text{set} (\text{getChoices } g \text{ } p \text{ } es) \subseteq \{p\}$
 ⟨*proof*⟩

lemma *getChoices-p*:

$(a,b) \in \text{set} (\text{getChoices } g \text{ } p \text{ } es) \implies b = p$
 ⟨*proof*⟩

definition *sort-by-pri* **where**

```

sort-by-pri min-pri edges = foldl ( $\lambda es e.$ 
  let idx = nat-of-integer (abs (prio e))
  in if idx > min-pri
    then abort STR "Invalid priority" ( $\lambda -. es$ )
    else let ep = e # (es ! idx) in es[idx := ep]
  ) (replicate (min-pri + 1) []) edges

```

lemma *sort-by-pri-edges'*:

```

assumes set edges  $\subseteq A$ 
shows set (sort-by-pri min-pri edges)  $\subseteq \{xs. set\ xs \subseteq A\}$ 
<proof>

```

lemma *sort-by-pri-edges*:

```

assumes set edges  $\subseteq A$ 
and es  $\in$  set (sort-by-pri min-pri edges)
shows set es  $\subseteq A$ 
<proof>

```

lemma *sort-by-pri-length*:

```

length (sort-by-pri min-pri edges) = min-pri + 1
<proof>

```

definition *executable*

```

:: states iarray  $\Rightarrow$  gStateI  $\Rightarrow$  choices nres
— Find all executable edges

```

where

```

executable ss g = (
  let procs = procs g in
  nfoldli procs ( $\lambda -. True$ ) ( $\lambda p E.$ 
    if (exclusive g = 0  $\vee$  exclusive g = pid p) then do {
      let (min-pri, edges) = (ss !! pState.idx p) !! pc p;
      ASSERT(set edges  $\subseteq$  edgeSet (ss !! pState.idx p));

      (E',-,)  $\leftarrow$ 
        if min-pri = 0 then do {
          WHILET ( $\lambda(E, brk, -). E = [] \wedge brk = 0$ ) ( $\lambda (-, -, ELSE). do \{$ 
            let g = g(gState_I.else := ELSE);
            E = getChoices g p edges
          in
            if E = [] then (
              if  $\neg$  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 ( $\forall es \in$  set pri-edges.
            set es  $\subseteq$  edgeSet (ss !! pState.idx p));

```

```

    let pri-edges = IArray pri-edges;

    WHILET (λ(E,pri,-). E = [] ∧ pri ≤ min-pri) (λ(-, pri, ELSE). do
  {
    let es = pri-edges !! pri;
    let g = g(gStateI.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
⟨proof⟩

```

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)
⟨proof⟩

```

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))

```

⟨proof⟩

lemma executable-edgeSet':

assumes $gState\text{-}inv\ prog\ g$
and $program\text{-}inv\ prog$
shows $executable\ (states\ prog)\ g$
 $\leq SPEC\ (\lambda cs. \forall (e,p) \in set\ cs.$
 $\quad e \in edgeSet\ ((states\ prog)\ !!\ pState.idx\ p)$
 $\quad \wedge\ pState\text{-}inv\ prog\ p$
 $\quad \wedge\ cl\text{-}inv(g,p))$
 $\langle proof \rangle$

schematic-goal $executable\text{-}refine$:
 $RETURN\ (?ex\ s\ g) \leq executable\ s\ g$
 $\langle proof \rangle$

concrete-definition $executable\text{-}impl$ for $s\ g$ uses $executable\text{-}refine$

5.8.3 Successor calculation

function to_I **where**
 $to_I\ (\ ()\ gState.vars = v, channels = ch, timeout = t, procs = p\)$
 $\quad =\ (\ ()\ gState.vars = v, channels = ch, timeout = False, procs = p,$
 $\quad\quad handshake = 0, hsdata = [], exclusive = 0, gState_I.else = False\)$
 $\langle proof \rangle$
termination $\langle proof \rangle$

function $from_I$ **where**
 $from_I\ (\ ()\ gState.vars = v, channels = ch, timeout = t, procs = p, \dots = m\)$
 $\quad =\ (\ ()\ gState.vars = v, channels = ch, timeout = t, procs = p\)$
 $\langle proof \rangle$
termination $\langle proof \rangle$

function $reset_I$ **where**
 $reset_I\ (\ ()\ gState.vars = v, channels = ch, timeout = t, procs = p,$
 $\quad\quad handshake = hs, hsdata = hsd, exclusive = -, gState_I.else = -\)$
 $\quad =\ (\ ()\ gState.vars = v, channels = ch, timeout = False, procs = p,$
 $\quad\quad handshake = 0, hsdata = if\ hs \neq 0\ then\ hsd\ else\ [],\ exclusive = 0,$
 $\quad\quad gState_I.else = False\)$
 $\langle proof \rangle$
termination $\langle proof \rangle$

lemma $gState\text{-}inv\text{-}to_I$:
 $gState\text{-}inv\ prog\ g = gState\text{-}inv\ prog\ (to_I\ g)$
 $\langle proof \rangle$

lemma $gState\text{-}inv\text{-}from_I$:
 $gState\text{-}inv\ prog\ g = gState\text{-}inv\ prog\ (from_I\ g)$
 $\langle proof \rangle$

lemma $gState\text{-}inv\text{-}reset_I$:
 $gState\text{-}inv\ prog\ g = gState\text{-}inv\ prog\ (reset_I\ g)$

<proof>

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 (λp (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
<proof>

lemma *removeProcs-length'*:

length (fst (snd (snd (removeProcs ps)))) ≤ length ps
<proof>

lemma *removeProcs-subset*:

removeProcs ps = (dead,sd,ps',dcs) ⇒ set ps' ⊆ set ps
<proof>

lemma *removeProcs-length*:

removeProcs ps = (dead,sd,ps',dcs) ⇒ length ps' ≤ length ps
<proof>

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*

<proof>

lemma *cleanChans-length*:

length (cleanChans dchans cs) = length cs
<proof>

definition *checkDeadProcs :: 'a gState-scheme ⇒ 'a gState-scheme* **where**

checkDeadProcs g = (

let (-, soDied, procs, dchans) = removeProcs (procs g) in

if soDied then

$g(\backslash \text{procs} := \text{procs}, \text{channels} := \text{cleanChans } d\text{chans } (\text{channels } g))$
 $\text{else } g)$

lemma *checkDeadProcs-gState-progress-rel:*

assumes $g\text{State-inv prog } g$
shows $(g, \text{checkDeadProcs } g) \in g\text{State-progress-rel prog}$
 $\langle \text{proof} \rangle$

lemma *gState-progress-rel-exclusive:*

$(g, g') \in g\text{State-progress-rel prog}$
 $\implies (g, g'(\backslash \text{exclusive} := p)) \in g\text{State-progress-rel prog}$
 $\langle \text{proof} \rangle$

definition *applyEdge*

$:: \text{program} \Rightarrow \text{edge} \Rightarrow p\text{State} \Rightarrow g\text{State}_I \Rightarrow g\text{State}_I \text{ nres}$

where

$\text{applyEdge prog } e \text{ } p \text{ } g = \text{do } \{$

$\text{let } (g', p') = \text{evalEffect } (\text{effect } e) \text{ prog } g \text{ } p;$
 $\text{ASSERT } ((g, g') \in g\text{State-progress-rel prog});$
 $\text{ASSERT } (p\text{State-inv prog } p');$
 $\text{ASSERT } (\text{cl-inv } (g', p'));$

$\text{let } p'' = (\text{case target } e \text{ of Index } t \Rightarrow$
 $\text{if } t < I\text{Array.length } (\text{states prog} !! p\text{State.idx } p') \text{ then } p'(\backslash \text{pc} := t)$
 $\text{else abort STR "Edge target out of bounds" } (\lambda-. p')$
 $\mid - \Rightarrow \text{abort STR "Edge target not Index" } (\lambda-. p'));$
 $\text{ASSERT } (p\text{State-inv prog } p'');$

$\text{let } g'' = g'(\backslash \text{procs} := \text{list-update } (\text{procs } g') (p\text{id } p'' - 1) p'');$
 $\text{ASSERT } ((g', g'') \in g\text{State-progress-rel prog});$

$\text{let } g''' = (\text{if isAtomic } e \wedge \text{handshake } g'' = 0$
 $\text{then } g''(\backslash \text{exclusive} := p\text{id } p'')$
 $\text{else } g'');$
 $\text{ASSERT } ((g', g''') \in g\text{State-progress-rel prog});$

$\text{let } g_f = (\text{if } p\text{c } p'' = 0 \text{ then } \text{checkDeadProcs } g''' \text{ else } g''');$
 $\text{ASSERT } ((g''', g_f) \in g\text{State-progress-rel prog});$
 $\text{RETURN } g_f \}$

lemma *applyEdge-gState-progress-rel:*

assumes program-inv prog
and $g\text{State-inv prog } g$
and $p\text{State-inv prog } p$
and $\text{cl-inv } (g, p)$
and $e \in \text{edgeSet } (\text{states prog} !! p\text{State.idx } p)$
shows $\text{applyEdge prog } e \text{ } p \text{ } g \leq \text{SPEC } (\lambda g'. (g, g') \in g\text{State-progress-rel prog})$
 $\langle \text{proof} \rangle$

schematic-goal *applyEdge-refine*:

RETURN (?ae prog e p g) ≤ *applyEdge* prog e p g
 ⟨proof⟩

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

ifoldli 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

) $\gg= (\lambda G. \text{if } ls.isEmpty \ G \text{ then } RETURN \ (ls.sng \ (f \ g)) \ \text{else } RETURN \ G)$

lemma *gState-progress-rel-intros:*

$(to_I \ g, \ gI') \in gState\text{-progress-rel} \ prog$
 $\implies (g, \ from_I \ gI') \in gState\text{-progress-rel} \ prog$
 $(gI, \ gI') \in gState\text{-progress-rel} \ prog$
 $\implies (gI, \ reset_I \ gI') \in gState\text{-progress-rel} \ prog$
 $(to_I \ g, \ gI') \in gState\text{-progress-rel} \ prog$
 $\implies (to_I \ g, \ gI' \ (timeout := t)) \in gState\text{-progress-rel} \ prog$
 $\langle proof \rangle$

lemma *gState-progress-rel-step-intros:*

$(to_I \ g, \ g') \in gState\text{-progress-rel} \ prog$
 $\implies (reset_I \ g', \ g'') \in gState\text{-progress-rel} \ prog$
 $\implies (g, \ from_I \ g'') \in gState\text{-progress-rel} \ prog$
 $(to_I \ g, \ g') \in gState\text{-progress-rel} \ prog$
 $\implies (reset_I \ g', \ g'') \in gState\text{-progress-rel} \ prog$
 $\implies (to_I \ g, \ g'') \in gState\text{-progress-rel} \ prog$
 $\langle proof \rangle$

lemma *cl-inv-reset_I:*

$cl\text{-inv}(g, p) \implies cl\text{-inv}(reset_I \ g, \ p)$
 $\langle proof \rangle$

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 $(\lambda gs. \forall g' \in ls. \alpha \ gs. (g, g') \in gState\text{-progress-rel} \ prog)$*
 $\langle proof \rangle$

lemma *RETURN-dRETURN:*

$RETURN \ f \leq f' \implies nres\text{-of} \ (dRETURN \ f) \leq f'$
 $\langle proof \rangle$

lemma *executable-dRETURN:*

$nres\text{-of} \ (dRETURN \ (executable\text{-impl} \ prog \ g)) \leq executable \ prog \ g$
 $\langle proof \rangle$

lemma *applyEdge-dRETURN:*

$nres\text{-of} \ (dRETURN \ (applyEdge\text{-impl} \ prog \ e \ p \ g)) \leq applyEdge \ prog \ e \ p \ g$
 $\langle proof \rangle$

schematic-goal *nexts-code-aux:*

$nres\text{-of} \ (?nexts \ prog \ g) \leq nexts \ prog \ g$

<proof>

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 \Rightarrow if X={ } then Code.abort msg (λ -. dm) else RES X
 | - \Rightarrow m)

definition *dSUCCEED-abort* where

dSUCCEED-abort msg dm m = (
 case m of
 *dSUCCEED*i \Rightarrow Code.abort msg (λ -. dm)
 | - \Rightarrow m)

definition *ref-succeed* where

ref-succeed m m' \longleftrightarrow m \leq m' \wedge (m=*SUCCEED* \longrightarrow m'=*SUCCEED*)

lemma *dSUCCEED-abort-SUCCEED-abort*:

\llbracket RETURN dm' \leq dm; *ref-succeed* (*nres-of* m') m \rrbracket
 \implies *nres-of* (*dSUCCEED-abort* msg (*dRETURN* dm') (m'))
 \leq *SUCCEED-abort* msg dm m

<proof>

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. α (*nexts-code* prog g)
 \implies (g,g') \in *gState-progress-rel* prog

<proof>

5.9 Finiteness of the state space

inductive-set *reachable-states*

for $P :: \text{program}$

and $g_s :: gState$ — start state

where

$g_s \in \text{reachable-states } P \ g_s \mid$

$g \in \text{reachable-states } P \ g_s \implies x \in \text{ls.}\alpha \ (\text{nexts-code } P \ g)$

$\implies x \in \text{reachable-states } P \ g_s$

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>

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* \Rightarrow *gState* \Rightarrow *gState* \Rightarrow *choices nres* **where**

replay prog g1 g2 = (

let

$g_1 = \text{to}_I \ g_1;$

$\text{check} = \lambda g. \text{from}_I \ g = g_2$

in

$\text{REC}_T \ (\lambda D \ g. \text{do } \{$

$E \leftarrow \text{executable} \ (\text{states } \text{prog}) \ g;$

if $E = []$ *then*

if $\text{check } g$ *then* $\text{RETURN } []$

else if $\neg \text{timeout } g$ *then* $D \ (g(\text{timeout} := \text{True}))$

else abort $\text{STR } \text{"Stuttering should not occur on replay"}$

$(\lambda -. \text{RETURN } [])$

else

let $g = \text{reset}_I \ g$ *in*

$\text{nfoldli } E \ (\lambda E. E = []) \ (\lambda (e,p) -.$

$\text{applyEdge } \text{prog } e \ p \ g \gg= (\lambda g'.$

if $\text{handshake } g' \neq 0 \vee \text{isAtomic } e$ *then do* {

$E_R \leftarrow D \ g';$

if $E_R = []$ *then*

if $\text{check } g'$ *then* $\text{RETURN } [(e,p)]$ *else* $\text{RETURN } []$

else

$\text{RETURN } ((e,p) \# E_R)$

$\} \text{ else if } \text{check } g' \text{ then } \text{RETURN } [(e,p)] \text{ else } \text{RETURN } [])$

5.11 Code export

code-identifier

code-module *PromelaInvariants* \rightarrow (SML) *Promela*
| **code-module** *PromelaDatastructures* \rightarrow (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 \langle *Promela.sml* \rangle

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 specifically, 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* \Rightarrow '*a* *list* \Rightarrow '*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*

$| \text{ltlc-aps-list}' (\text{Final-ltlc } x) l = \text{ltlc-aps-list}' x l$
 $| \text{ltlc-aps-list}' (\text{Global-ltlc } x) l = \text{ltlc-aps-list}' x l$
 $| \text{ltlc-aps-list}' (\text{And-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$
 $| \text{ltlc-aps-list}' (\text{Or-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$
 $| \text{ltlc-aps-list}' (\text{Implies-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$
 $| \text{ltlc-aps-list}' (\text{Until-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$
 $| \text{ltlc-aps-list}' (\text{Release-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$
 $| \text{ltlc-aps-list}' (\text{WeakUntil-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$
 $| \text{ltlc-aps-list}' (\text{StrongRelease-ltlc } x y) l = \text{ltlc-aps-list}' y (\text{ltlc-aps-list}' x l)$

lemma *ltlc-aps-list'-correct:*

$set (\text{ltlc-aps-list}' \varphi l) = \text{atoms-ltlc } \varphi \cup set l$
 $\langle \text{proof} \rangle$

lemma *ltlc-aps-list'-distinct:*

$distinct l \implies distinct (\text{ltlc-aps-list}' \varphi l)$
 $\langle \text{proof} \rangle$

definition *ltlc-aps-list* :: 'a ltlc \Rightarrow 'a list

where

$ltlc-aps-list \varphi = \text{ltlc-aps-list}' \varphi []$

lemma *ltlc-aps-list-correct:*

$set (\text{ltlc-aps-list } \varphi) = \text{atoms-ltlc } \varphi$
 $\langle \text{proof} \rangle$

lemma *ltlc-aps-list-distinct:*

$distinct (\text{ltlc-aps-list } \varphi)$
 $\langle \text{proof} \rangle$

primrec *idx'* :: nat \Rightarrow 'a list \Rightarrow 'a \Rightarrow nat option **where**

$idx' - [] - = None$
 $| idx' ctr (x\#xs) y = (\text{if } x = y \text{ then } Some \text{ ctr else } idx' (ctr+1) xs y)$

definition *idx* = *idx'* 0

lemma *idx'-correct:*

assumes *distinct xs*
shows $idx' \text{ ctr } xs \ y = Some \ n \iff n \geq \text{ctr} \wedge n < \text{length } xs + \text{ctr} \wedge xs ! (n - \text{ctr}) = y$
 $\langle \text{proof} \rangle$

lemma *idx-correct:*

assumes *distinct xs*
shows $idx \text{ } xs \ y = Some \ n \iff n < \text{length } xs \wedge xs ! n = y$
 $\langle \text{proof} \rangle$

lemma *idx-dom:*

assumes *distinct xs*

shows $\text{dom } (\text{idx } xs) = \text{set } xs$
 ⟨proof⟩

lemma *idx-image-self*:
assumes *distinct xs*
shows $(\text{the } \circ \text{idx } xs) \text{ ' set } xs = \{..<\text{length } xs\}$
 ⟨proof⟩

lemma *idx-ran*:
assumes *distinct xs*
shows $\text{ran } (\text{idx } xs) = \{..<\text{length } xs\}$
 ⟨proof⟩

lemma *idx-inj-on-dom*:
assumes *distinct xs*
shows *inj-on* $(\text{idx } xs) (\text{dom } (\text{idx } xs))$
 ⟨proof⟩

definition *ltl-convert* :: $\text{expr ltlc} \Rightarrow \text{APs} \times \text{nat ltlc}$ **where**
ltl-convert $\varphi =$ (
 let $\text{APs} = \text{ltlc-aps-list } \varphi$;
 $\varphi_i = \text{map-ltlc } (\text{the } \circ \text{idx } \text{APs}) \varphi$
 in $(\text{IArray } \text{APs}, \varphi_i)$)

lemma *ltl-convert-correct*:
assumes *ltl-convert* $\varphi = (\text{APs}, \varphi_i)$
shows *atoms-ltlc* $\varphi = \text{set } (\text{IArray.list-of } \text{APs})$ (**is** ?P1)
and *atoms-ltlc* $\varphi_i = \{..<\text{IArray.length } \text{APs}\}$ (**is** ?P2)
and $\varphi_i = \text{map-ltlc } (\text{the } \circ \text{idx } (\text{IArray.list-of } \text{APs})) \varphi$ (**is** ?P3)
and *distinct* $(\text{IArray.list-of } \text{APs})$
 ⟨proof⟩

definition *prepare*
 :: $- \times (\text{program} \Rightarrow \text{unit}) \Rightarrow \text{ast} \Rightarrow \text{expr ltlc} \Rightarrow (\text{program} \times \text{APs} \times \text{gState}) \times \text{nat ltlc}$
where
prepare *cfg ast* $\varphi \equiv$
 let
 $(\text{prog}, g_0) = \text{Promela.setUp } \text{ast}$;
 $(\text{APs}, \varphi_i) = \text{PromelaLTL.ltl-convert } \varphi$
 in
 $((\text{prog}, \text{APs}, g_0), \varphi_i)$

lemma *prepare-instrument*[code]:
prepare *cfg ast* $\varphi \equiv$
 let
 $(-, \text{printF}) = \text{cfg}$;
 $- = \text{PromelaStatistics.start } ()$;
 $(\text{prog}, g_0) = \text{Promela.setUp } \text{ast}$;

```

- = printF prog;
(APs,φi) = PromelaLTL.ltl-convert φ;
- = PromelaStatistics.stop-timer ()
in
  ((prog, APs, g0), φi)
⟨proof⟩

```

export-code *prepare checking SML*

6.2 Language of a Promela program

definition *propValid* :: $APs \Rightarrow gState \Rightarrow nat \Rightarrow bool$ **where**

propValid APs g i $\longleftrightarrow i < IArray.length APs \wedge exprArith g emptyProc (APs!!i) \neq 0$

definition *promela-E* :: $program \Rightarrow (gState \times gState) set$

— Transition relation of a promela program

where

promela-E prog $\equiv \{(g, g'). g' \in ls.\alpha (nexts-code prog g)\}$

definition *promela-E-ltl* :: $program \times APs \Rightarrow (gState \times gState) set$ **where**

promela-E-ltl = *promela-E* \circ *fst*

definition *promela-is-run'* :: $program \times gState \Rightarrow gState word \Rightarrow bool$

— Predicate defining runs of promela programs

where

promela-is-run' progg r \equiv
 let (prog, g₀) = progg in
 r 0 = g₀
 $\wedge (\forall i. r (Suc i) \in ls.\alpha (nexts-code prog (r i)))$

abbreviation *promela-is-run* \equiv *promela-is-run'* \circ *setUp*

definition *promela-is-run-ltl* :: $program \times APs \times gState \Rightarrow gState word \Rightarrow bool$

where

promela-is-run-ltl promg r \equiv let (prog, APs, g) = promg in *promela-is-run'* (prog, g) *r*

definition *promela-props* :: $gState \Rightarrow expr set$

where

promela-props g = $\{e. exprArith g emptyProc e \neq 0\}$

definition *promela-props-ltl* :: $APs \Rightarrow gState \Rightarrow nat set$

where

promela-props-ltl APs g \equiv *Collect* (*propValid* APs g)

definition *promela-language* :: $ast \Rightarrow expr set word set$ **where**

promela-language ast $\equiv \{promela-props \circ r \mid r. promela-is-run ast r\}$

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* $\varphi = (APs, \varphi_i)$

shows *promela-props-ltl APs* =

map-props (idx (IArray.list-of APs)) ∘ *promela-props*

⟨*proof*⟩

lemma *promela-run-in-language-iff*:

assumes *conv*: *ltl-convert* $\varphi = (APs, \varphi_i)$

shows *promela-props* ∘ $\xi \in \textit{language-ltlc } \varphi$

⟷ *promela-props-ltl APs* ∘ $\xi \in \textit{language-ltlc } \varphi_i$ (**is** ?*L* ⟷ ?*R*)

⟨*proof*⟩

lemma *promela-language-sub-iff*:

assumes *conv*: *ltl-convert* $\varphi = (APs, \varphi_i)$

and *setUp*: *setUp ast* = (*prog, g*)

shows *promela-language-ltl (prog, APs, g)* ⊆ *language-ltlc* φ_i ⟷ *promela-language ast* ⊆ *language-ltlc* φ

⟨*proof*⟩

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.