

Promela Formalization

By René Neumann

February 23, 2021

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	25
3.5	Printing	26
4	Invariants for Promela data structures	29
4.1	Bounds	29
4.2	Variables and similar	30
4.3	Invariants of a process	33
4.4	Invariants of the global state	35
4.5	Invariants of the program	38
5	Formalization of Promela semantics	39
5.1	Misc Helpers	39
5.2	Variable handling	40
5.3	Expressions	45
5.4	Variable declaration	48
5.5	Folding	53
5.6	Starting processes	55
5.7	AST to edges	62
5.7.1	Setup	65
5.8	Semantic Engine	70
5.8.1	Evaluation of Edges	70
5.8.2	Executable edges	78
5.8.3	Successor calculation	82
5.8.4	Handle non-termination	89
5.9	Finiteness of the state space	90
5.10	Traces	91
5.10.1	Printing of traces	92
5.11	Code export	92
6	LTL integration	93
6.1	LTL optimization	93
6.2	Language of a Promela program	97
6.3	Proposition types and conversion	100

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 `lt1` and `assert`. Other constructs are accepted but ignored, because they do not change the behavior of a model: advanced variable scoping, `xr`, `xs`, `print*`, priorities, and visibility of variables.

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

2 Abstract Syntax Tree

```
theory PromelaAST
imports Main
begin
```

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

```
context
begin
```

```
local-setup ⟨
  Local-Theory.map-background-naming (Name-Space.mandatory-path AST)
⟩
```

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

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

```

| BinOpAnd
| BinOpOr

datatype unOp =
  UnOpComp
| UnOpMinus
| UnOpNeg

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

and varRef = VarRef String.literal
  expr option
  varRef option

and recvArg = RecvArgVar varRef
| RecvArgEval expr
| RecvArgConst integer

datatype range =
  RangeFromTo varRef
  expr
  expr
| RangeIn varRef varRef

datatype varType =
  VarTypeBit
| VarTypeBool
| VarTypeByte

```

```

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

```

```

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

```

```

datatype decl =
  Decl bool option
    varType
    varDecl list

```

```

datatype 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 = StepStmt stmt stmt 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 *stmt* = *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 stmt*
 | *StmntRun String.literal*
 expr list
 | *StmntCond expr*
 | *StmntAssert expr*

```

and step = StepStmnt stmnt stmnt option
      | StepDecl proc VarDecl list
      | StepSkip

```

```

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

```

```

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

```

3.2 Preprocess the AST of the parser into our variant

We setup some functionality for printing warning or even errors.

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

consts

```

warn :: String.literal ⇒ unit

```

abbreviation *with-warn msg e* ≡ let - = warn msg in e

abbreviation *the-warn opt msg* ≡ case opt of None ⇒ () | - ⇒ warn msg

usc: "Unsupported Construct"

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

definition [*code del*]: *err* (e :: String.literal) = *undefined*

abbreviation *errv e v* ≡ *err* (e + v)

definition [*simp, code del*]: *abort* (msg :: String.literal) f = f ()

abbreviation *abortv msg v f* ≡ *abort* (msg + v) f

code-printing

```

code-module PromelaUtils ↪ (SML) ⟨
  structure PromelaUtils = struct
    exception UnsupportedConstruct of string
    exception StaticError of string
    exception RuntimeError of string
    fun warn msg = TextIO.print (Warning: ^ msg ^ \n)
    fun usc c = raise (UnsupportedConstruct c)
    fun err e = raise (StaticError e)
    fun abort msg - = raise (RuntimeError msg)
  end⟩

```

| **constant** *warn* ↪ (SML) PromelaUtils.warn

| **constant** *usc* ↪ (SML) PromelaUtils.usc

```

| constant err → (SML) PromelaUtils.err
| constant abort → (SML) PromelaUtils.abort
code-reserved SML PromelaUtils

```

The preprocessing is done for each type on its own.

```

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

```

```

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

```

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

```

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

```

```

definition dealWithVar
  :: var-data ⇒ String.literal
  ⇒ (String.literal ⇒ integer option × bool ⇒ expr option ⇒ 'a)
  ⇒ (String.literal ⇒ integer option × bool ⇒ expr option ⇒ 'a)
  ⇒ (integer ⇒ 'a) ⇒ 'a

```

```

where
  dealWithVar cvm n fC fV fM ≡ (
    let (c,v,m,a) = cvm in

```

```

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

primrec enforceChan :: varRef + chanRef => chanRef where
  enforceChan (Inl _) = err STR "Channel expected. Got normal variable."
  | enforceChan (Inr c) = c

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

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

fun ppExpr :: var-data => AST.expr => expr
and ppVarRef :: var-data => AST.varRef => varRef + chanRef
and ppRecvArg :: var-data => AST.recvArg => recvArg
where
  ppVarRef cvm (AST.VarRef name idx None) = dealWithVar cvm name
    (\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 (sze,g) v, m, a),
           VarDeclNum l h name sze init)))

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

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

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

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

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

| ppProcArg (c,v,m,a) (VTBounded l h) g
  (AST.VarDeclMType name None None) = (
  case lm.lookup name v of

```

```

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

| ppProcArg - - - (AST.VarDeclUnsigned - - -) = usc STR "VarDeclUnsigned"

```

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

definition *cvm-fold where*

```

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

```

lemma *cvm-fold-cong[fundef-cong]:*

```

assumes cvm = cvm'
and stepss = stepss'
and ∧x d. x ∈ set stepss ⇒ g d x = g' d x
shows cvm-fold g cvm stepss = cvm-fold g' cvm' stepss'

```

unfolding *cvm-fold-def*

using *assms*

by (*fastforce intro: foldl-cong split: prod.split*)

fun *liftDecl where*

```

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

```

definition *ppDecl*

```

:: bool ⇒ var-data ⇒ AST.decl ⇒ var-data × varDecl list

```

where

```

ppDecl = liftDecl ppVarDecl

```

definition *ppDeclProc*

```

:: var-data ⇒ AST.decl ⇒ var-data × procVarDecl list

```

where

```

ppDeclProc = liftDecl ppProcVarDecl False

```

definition *ppDeclProcArg*

```

:: var-data ⇒ AST.decl ⇒ var-data × procArg list

```

where

```

ppDeclProcArg = liftDecl ppProcArg False

```


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

definition *decr* :: *varRef* ⇒ *stmt* **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* ⇒ *expr* ⇒ *expr* ⇒ *step list* ⇒ *stmt* **where**

```
forFromTo i lb ub steps = (
  let
    — i = lb
    loop-pre = StepStmt (StmtAssign i lb) None;
    — i ≤ ub
    loop-cond = StepStmt (StmtCond
      (ExprBinOp BinOpLEq (ExprVarRef i) ub))
      None;
    — i++
    loop-incr = StepStmt (incr i) None;
    — i ≤ ub -> ...; i++
    loop-body = loop-cond # steps @ [loop-incr];
    — else -> break
    loop-abort = [StepStmt StmtElse None, StepStmt StmtBreak None];
    — do :: i ≤ ub -> ... :: else -> break od
    loop = StepStmt (StmtDo [loop-body, loop-abort]) None
  in
    StmtSeq [loop-pre, loop])
```

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

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

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

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

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

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

```

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

```

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

```

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

```

definition `select :: varRef ⇒ expr ⇒ expr ⇒ stmnt where`

```

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

```

type-synonym `inlines =`

```
(String.literal, String.literal list × (var-data ⇒ var-data × step list)) lm
```

type-synonym `stmnt-data =`

```
bool × varDecl list × inlines × var-data
```

fun `ppStep :: stmnt-data ⇒ AST.step ⇒ stmnt-data * step`

and $ppStmnt :: stmnt\text{-}data \Rightarrow AST.stmnt \Rightarrow stmnt\text{-}data * stmnt$
where

```

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

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

```

```

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

```

definition *lookupLTL*
 $:: AST.module\ list \Rightarrow String.literal \Rightarrow String.literal\ option$
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*
 $| EEId$
 $| EEGoto$
 $| EEAssert\ expr$
 $| EEAssign\ varRef\ expr$
 $| EEDecl\ procVarDecl$
 $| EERun\ String.literal\ expr\ list$
 $| EESend\ chanRef\ expr\ list\ bool$
 $| EERcv\ chanRef\ recvArg\ list\ bool\ bool$

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 \Rightarrow bool where*
 $isAtomic\ e = (case\ atomic\ e\ of\ Atomic \Rightarrow True\ | - \Rightarrow False)$

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

3.4 State

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

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

type-synonym *var-dict* = (*String.literal, variable*) *lm*

type-synonym *labels* = (*String.literal, nat*) *lm*

type-synonym *lts* = (*String.literal, String.literal*) *lm*

type-synonym *states* = ($-$ *prio: integer \times 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'
unfolding printList-def
using assms
by (auto intro: foldl-cong)

```

```

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

```

```

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

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

| printChanRef f (ChanRef v) = printVarRef f v

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

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

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

primrec printCond :: (integer => string) => edgeCond => string where
    printCond f ECElse = "else"
| printCond f ECTrue = "true"

```

```

| printCond f ECFalse = "false"
| printCond f (ECRun n) = "run " @ String.explode n @ "(...)"
| printCond f (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 ⇒ string) ⇒ edgeIndex ⇒ string where
  printIndex f (Index pos) = f (integer-of-nat pos)
| printIndex - (LabelJump l -) = String.explode l

```

```

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

```

```

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

```

```

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

```

```

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

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

```

4 Invariants for Promela data structures

```

theory PromelaInvariants
imports PromelaDatastructures
begin

```

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

4.1 Bounds

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

```

definition min-var-value :: integer where
  min-var-value = -(231)

```

```

definition max-var-value :: integer where
  max-var-value = (231) - 1

```

```

lemma min-max-var-value-simps [simp, intro!]:

```

```

  min-var-value < max-var-value
  min-var-value < 0
  min-var-value ≤ 0
  max-var-value > 0
  max-var-value ≥ 0

```

```

by (simp-all add: min-var-value-def max-var-value-def)

```

```

definition max-procs ≡ 255

```

```

definition max-channels ≡ 65535

```

```

definition max-array-size = 65535

```

4.2 Variables and similar

```

fun varType-inv :: varType  $\Rightarrow$  bool where
  varType-inv (VTBounded l h)
     $\longleftrightarrow$   $l \geq \text{min-var-value} \wedge h \leq \text{max-var-value} \wedge l < h$ 
| varType-inv VTChan  $\longleftrightarrow$  True

fun variable-inv :: variable  $\Rightarrow$  bool where
  variable-inv (Var t val)
     $\longleftrightarrow$  varType-inv t  $\wedge$  val  $\in$  {min-var-value..max-var-value}
| variable-inv (VArray t sz ar)
     $\longleftrightarrow$  varType-inv t
       $\wedge$  sz  $\leq$  max-array-size
       $\wedge$  IArray.length ar = sz
       $\wedge$  set (IArray.list-of ar)  $\subseteq$  {min-var-value..max-var-value}

fun channel-inv :: channel  $\Rightarrow$  bool where
  channel-inv (Channel cap ts q)
     $\longleftrightarrow$  cap  $\leq$  max-array-size
       $\wedge$  cap  $\geq$  0
       $\wedge$  set ts  $\subseteq$  Collect varType-inv
       $\wedge$  length ts  $\leq$  max-array-size
       $\wedge$  length q  $\leq$  max-array-size
       $\wedge$  ( $\forall x \in$  set q. length x = length ts
       $\wedge$  set x  $\subseteq$  {min-var-value..max-var-value})
| channel-inv (HChannel ts)
     $\longleftrightarrow$  set ts  $\subseteq$  Collect varType-inv  $\wedge$  length ts  $\leq$  max-array-size
| channel-inv InvChannel  $\longleftrightarrow$  True

lemma varTypes-finite:
  finite (Collect varType-inv)
proof (rule finite-subset)
  show Collect (varType-inv)  $\subseteq$ 
    {VTChan}
     $\cup$  ( $\lambda(l,h).$  VTBounded l h)
    ‘({min-var-value..max-var-value}  $\times$  {min-var-value..max-var-value})
  apply (rule subsetI)
  apply (case-tac x)
  apply auto
  done

  show finite ... by auto
qed

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

```

```

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

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

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

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

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

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

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

```

done

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

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

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

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

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

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

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

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

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

definition *statesDecls* :: *states* \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*)
by (*simp add: edgeSet-finite edgeDecls-finite statesNames-def statesDecls-def*)

fun *process-names* :: *states* \Rightarrow *process* \Rightarrow *String.literal set* **where**
process-names *ss* (-, -, *args*, *decls*) =
 statesNames ss
 \cup *procArgName* ' *set args*

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

lemma *process-names-finite*:
 $\text{finite } (\text{process-names } ss \ p)$
by (*cases* p) (*simp add: statesNames-finite*)

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

lemma *vardicts-finite*:
 $\text{finite } (\text{Collect } (\text{vardict-inv } ss \ p))$

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

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

qed

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

4.3 Invariants of a process

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

lemma *pStates-finite*:
 $\text{finite } (\text{Collect } (\text{pState-inv } \text{prog}))$

proof –
let $?P1 = \{.. \text{max-procs} :: \text{nat}\}$

```

let ?P2 = {..IArray.length (states prog)}
let ?P3 = {..Max (IArray.length ' (set (IArray.list-of (states prog))))}
let ?P4 = {cs.set cs  $\subseteq$  { $-1..<$ integer-of-nat max-channels}
            $\wedge$  length cs  $\leq$  max-channels}
let ?P5 =  $\bigcup x \in$  {..IArray.length (states prog)}.
           Collect (vardict-inv (states prog !! x) (processes prog !! x))
let ?P = ?P1  $\times$  ?P2  $\times$  ?P3  $\times$  ?P4  $\times$  ?P5

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

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

```

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

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

lemma *cl-inv-lengthD*:
cl-inv (*g,p*) \Longrightarrow *length* (*pState.channels p*) \leq *length* (*gState.channels g*)
unfolding *cl-inv-def*
by *auto*

lemma *cl-invI*:
length (*pState.channels p*) \leq *length* (*gState.channels g*) \Longrightarrow *cl-inv* (*g,p*)
unfolding *cl-inv-def* **by** *auto*

lemma *cl-inv-trans*:
length (*channels g*) \leq *length* (*channels g'*) \Longrightarrow *cl-inv* (*g,p*) \Longrightarrow *cl-inv* (*g',p*)
by (*simp add: cl-inv-def*)

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

lemma *cl-inv-handshake-update*[intro!]:
 $cl\text{-}inv\ (g,p) \implies cl\text{-}inv\ (g(\backslash handshake := h),p)$
by (*simp add: cl-inv-def*)

lemma *cl-inv-hsdata-update*[intro!]:
 $cl\text{-}inv\ (g,p) \implies cl\text{-}inv\ (g(\backslash hsdata := h),p)$
by (*simp add: cl-inv-def*)

lemma *cl-inv-procs-update*[intro!]:
 $cl\text{-}inv\ (g,p) \implies cl\text{-}inv\ (g(\backslash procs := ps),p)$
by (*simp add: cl-inv-def*)

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

4.4 Invariants of the global state

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

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

definition *gState-progress-rel* :: *program* \Rightarrow ('a *gState-scheme*) *rel* **where**
 $gState\text{-}progress\text{-}rel\ p = \{(g,g').\ gState\text{-}inv\ p\ g \wedge gState\text{-}inv\ p\ g'$
 $\wedge length\ (channels\ g) \leq length\ (channels\ g')$
 $\wedge dom\ (lm.\alpha\ (vars\ g)) = dom\ (lm.\alpha\ (vars\ g'))\}$

lemma *gState-progress-rel-gState-invI1*[intro]:
 $(g,g') \in gState\text{-}progress\text{-}rel\ prog \implies gState\text{-}inv\ prog\ g$
by (*simp add: gState-progress-rel-def*)

lemma *gState-progress-rel-gState-invI2*[intro]:
 $(g,g') \in gState\text{-}progress\text{-}rel\ prog \implies gState\text{-}inv\ prog\ g'$
by (*simp add: gState-progress-rel-def*)

lemma *gState-progress-relI*:

```

assumes gState-inv prog g
and gState-inv prog g'
and length (channels g) ≤ length (channels g')
and dom (lm.α (vars g)) = dom (lm.α (vars g'))
shows  $(g, g') \in \text{gState-progress-rel prog}$ 
unfolding gState-progress-rel-def
using assms
by auto

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma gState-progress-rel-trancl-id[simp]:
   $(\text{gState-progress-rel prog})^+ = \text{gState-progress-rel prog}$ 
by simp

```

```

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

```

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

```

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

```

```

       $\wedge \text{length } cs \leq \text{max-channels}$ 
let ?G3 = {True, False}
let ?G4 = {ps. set ps  $\subseteq$  Collect (pState-inv prog)
       $\wedge \text{length } ps \leq \text{max-procs}$ }

let ?G = ?G1  $\times$  ?G2  $\times$  ?G3  $\times$  ?G4
let ?G' = ( $\lambda(\text{vars}, \text{chans}, t, ps)$ ). gState.make vars chans t ps) ' ?G

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

have finite ((gState-progress-rel prog) " {g})
proof (rule finite-subset)
  show (gState-progress-rel prog) " {g}  $\subseteq$ 
    ( $\lambda(\text{vars}, \text{chans}, t, ps)$ ). gState.make vars chans t ps) ' ?G
apply (clarsimp simp add: image-def gState-inv-def gState.defs gState-progress-rel-def)
apply (rule-tac x = vars x in exI)
apply (simp add: lm-ball-eq-ran)
apply (rule-tac x = channels x in exI)
apply (case-tac timeout x)
applyclarsimp
apply (rule-tac x=procs x in exI)
apply auto
done
show finite ... using G1
by (blast intro: finite-lists-length-le channels-finite pStates-finite)
qed
with gState-progress-rel-rtrancl-absorb[OF True] show ?thesis by simp
qed

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

```

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

lemma *gState-progress-rel-channels-update-step*:

assumes *gState-inv prog g*

and *step*: $(g, g') \in \text{gState-progress-rel prog}$

and *channel-inv c*

and $i < \text{length}(\text{channels } g')$

shows $(g, \text{gState.channels-update } (\lambda cs. cs[i:=c]) g') \in \text{gState-progress-rel prog}$

proof –

note *step*

also hence *gState-inv prog g'* **by** *blast*

note *gState-progress-rel-channels-update*[*OF this assms*(3,4)]

finally show *?thesis* .

qed

4.5 Invariants of the program

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

definition *program-inv where*

program-inv prog

$\longleftrightarrow \text{IArray.length}(\text{states prog}) > 0$

$\wedge \text{IArray.length}(\text{states prog}) = \text{IArray.length}(\text{processes prog})$

$\wedge (\forall s \in \text{set}(\text{IArray.list-of}(\text{states prog})). \text{IArray.length } s > 0)$

$\wedge \text{lm.ball}(\text{proc-data prog})$

$(\lambda(-, \text{sid}x).$

$\text{sid}x < \text{IArray.length}(\text{processes prog})$

$\wedge \text{fst}(\text{processes prog} !! \text{sid}x) = \text{sid}x)$

$\wedge (\forall (\text{sid}x, \text{start}, \text{procArgs}, \text{args}) \in \text{set}(\text{IArray.list-of}(\text{processes prog}))).$

$(\exists s. \text{start} = \text{Index } s \wedge s < \text{IArray.length}(\text{states prog} !! \text{sid}x)))$

lemma *program-inv-length-states*:

assumes *program-inv prog*

and $n < \text{IArray.length}(\text{states prog})$

shows $\text{IArray.length}(\text{states prog} !! n) > 0$

using *assms* **by** (*simp add: program-inv-def*)

lemma *program-invI*:

assumes $0 < \text{IArray.length}(\text{states prog})$

and $\text{IArray.length}(\text{states prog}) = \text{IArray.length}(\text{processes prog})$

and $\bigwedge s. s \in \text{set}(\text{IArray.list-of}(\text{states prog}))$

$\implies 0 < \text{IArray.length } s$

and $\bigwedge \text{sid}x. \text{sid}x \in \text{ran}(\text{lm.alpha}(\text{proc-data prog}))$

$\implies \text{sid}x < \text{IArray.length}(\text{processes prog})$

$\wedge \text{fst}(\text{processes prog} !! \text{sid}x) = \text{sid}x$

and $\bigwedge \text{sid}x \text{ start procArgs args.}$

$(\text{sid}x, \text{start}, \text{procArgs}, \text{args}) \in \text{set}(\text{IArray.list-of}(\text{processes prog}))$

$\implies \exists s. \text{start} = \text{Index } s \wedge s < \text{IArray.length}(\text{states prog} !! \text{sid}x)$

```

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

end

```

5 Formalization of Promela semantics

theory *Promela*

imports

PromelaDatastructures

PromelaInvariants

PromelaStatistics

begin

Auxiliary

lemma *mod-integer-le*:

$a \leq b \implies 0 < a \implies x \bmod (a + 1) \leq b$ **for** $a\ b\ x :: \text{integer}$

by (*metis add-pos-nonneg discrete not-less order.strict-trans2*

unique-euclidean-semiring-numeral-class.pos-mod-bound zero-le-one)

lemma *mod-integer-ge*:

$b \leq 0 \implies 0 < a \implies b \leq x \bmod (a+1)$ **for** $a\ b\ x :: \text{integer}$

by (*metis dual-order.trans less-add-one order.strict-trans*

unique-euclidean-semiring-numeral-class.pos-mod-sign)

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 ($\lambda-. lbls$)

definition *min-prio* :: *edge list* \Rightarrow *integer* \Rightarrow *integer* **where**

min-prio es $start = \text{Min} ((prio \text{ ' set } es) \cup \{start\})$

lemma *min-prio-code* [*code*]:

min-prio es $start = \text{fold } (\lambda e\ pri. \text{if } prio\ e < pri \text{ then } prio\ e \text{ else } pri) \text{ es } start$

proof –

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

5.2 Variable handling

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

```

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

```

```

primrec checkVarValue :: varType ⇒ integer ⇒ integer where

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

end

```

```

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

```

```

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

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

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

```

```

definition getVar'
  :: bool  $\Rightarrow$  String.literal  $\Rightarrow$  integer option
   $\Rightarrow$  'a gState-scheme  $\Rightarrow$  pState
   $\Rightarrow$  integer option
where
  getVar' gl v idx g p = (
    let vars = if gl then gState.vars g else pState.vars p in
    map-option ( $\lambda x$ . lookupVar x idx) (lm.lookup v vars))

```

```

definition setVar'
  :: bool  $\Rightarrow$  String.literal  $\Rightarrow$  integer option
   $\Rightarrow$  integer
   $\Rightarrow$  'a gState-scheme  $\Rightarrow$  pState
   $\Rightarrow$  '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  $\Rightarrow$  abortv STR "Unknown global variable: " v ( $\lambda$ -. (g,p))
    | Some x  $\Rightarrow$  (g( $\backslash$ gState.vars := lm.update v (editVar x idx val)
                    (gState.vars g)))
      , p)
else
  case lm.lookup v (pState.vars p) of
    None  $\Rightarrow$  abortv STR "Unknown proc variable: " v ( $\lambda$ -. (g,p))
  | Some x  $\Rightarrow$  (g, p( $\backslash$ pState.vars := lm.update v (editVar x idx val)
                    (pState.vars p))))

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

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

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

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

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

```

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

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

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

definition *withVar'*
 $:: \text{bool} \Rightarrow \text{String.literal} \Rightarrow \text{integer option}$
 $\Rightarrow (\text{integer} \Rightarrow 'x)$
 $\Rightarrow 'a \text{ gState-scheme} \Rightarrow \text{pState}$
 $\Rightarrow 'x$

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

definition *withChannel'*
 $:: \text{bool} \Rightarrow \text{String.literal} \Rightarrow \text{integer option}$
 $\Rightarrow (\text{nat} \Rightarrow \text{channel} \Rightarrow 'x)$
 $\Rightarrow 'a \text{ gState-scheme} \Rightarrow \text{pState}$
 $\Rightarrow 'x$

where
withChannel' gl v idx f g p = (
let error = λ -. abortv STR "Variable is not a channel: " v
(λ -. f 0 InvChannel) in
let abort = λ -. abortv STR "Channel already closed / invalid: " v
(λ -. f 0 InvChannel)
in withVar' gl v idx (λ 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 $\text{trivCond } x \equiv \text{if } x \text{ then } 1 \text{ else } 0$

fun $\text{exprArith} :: 'a \text{ gState-scheme} \Rightarrow \text{pState} \Rightarrow \text{expr} \Rightarrow \text{integer}$

and $\text{pollCheck} :: 'a \text{ gState-scheme} \Rightarrow \text{pState} \Rightarrow \text{channel} \Rightarrow \text{recvArg list} \Rightarrow \text{bool}$
 $\Rightarrow \text{bool}$

and $\text{recvArgsCheck} :: 'a \text{ gState-scheme} \Rightarrow \text{pState} \Rightarrow \text{recvArg list} \Rightarrow \text{integer list}$
 $\Rightarrow \text{bool}$

where

$\text{exprArith } g \text{ p } (\text{ExprConst } x) = x$

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

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

| $\text{exprArith } g \text{ p } (\text{ExprLen } (\text{ChanRef } (\text{VarRef } gl \text{ name } \text{None}))) =$
 $\text{withChannel}' gl \text{ name } \text{None } ($
 $\lambda\text{- } c. \text{ case } c \text{ of}$
 $\quad \text{Channel } - - q \Rightarrow \text{integer-of-nat } (\text{length } q)$
 $\quad | \text{HChannel } - \Rightarrow 0) g \text{ p}$

| $\text{exprArith } g \text{ p } (\text{ExprLen } (\text{ChanRef } (\text{VarRef } gl \text{ name } (\text{Some } idx)))) =$
 $\text{withChannel}' gl \text{ name } (\text{Some } (\text{exprArith } g \text{ p } idx)) ($
 $\lambda\text{- } c. \text{ case } c \text{ of}$
 $\quad \text{Channel } - - q \Rightarrow \text{integer-of-nat } (\text{length } q)$
 $\quad | \text{HChannel } - \Rightarrow 0) g \text{ p}$

| $\text{exprArith } g \text{ p } (\text{ExprEmpty } (\text{ChanRef } (\text{VarRef } gl \text{ name } \text{None}))) =$
 $\text{trivCond } (\text{withChannel}' gl \text{ name } \text{None } ($
 $\lambda\text{- } c. \text{ case } c \text{ of } \text{Channel } - - q \Rightarrow (q = [])$
 $\quad | \text{HChannel } - \Rightarrow \text{True}) g \text{ p})$

| $\text{exprArith } g \text{ p } (\text{ExprEmpty } (\text{ChanRef } (\text{VarRef } gl \text{ name } (\text{Some } idx)))) =$
 $\text{trivCond } (\text{withChannel}' gl \text{ name } (\text{Some } (\text{exprArith } g \text{ p } idx)) ($
 $\lambda\text{- } c. \text{ case } c \text{ of } \text{Channel } - - q \Rightarrow (q = [])$
 $\quad | \text{HChannel } - \Rightarrow \text{True}) g \text{ p})$

| $\text{exprArith } g \text{ p } (\text{ExprFull } (\text{ChanRef } (\text{VarRef } gl \text{ name } \text{None}))) =$
 $\text{trivCond } (\text{withChannel}' gl \text{ name } \text{None } ($
 $\lambda\text{- } c. \text{ case } c \text{ of}$
 $\quad \text{Channel } \text{cap } - - q \Rightarrow \text{integer-of-nat } (\text{length } q) \geq \text{cap}$
 $\quad | \text{HChannel } - \Rightarrow \text{False}) g \text{ p})$

| $\text{exprArith } g \text{ p } (\text{ExprFull } (\text{ChanRef } (\text{VarRef } gl \text{ name } (\text{Some } idx)))) =$
 $\text{trivCond } (\text{withChannel}' gl \text{ name } (\text{Some } (\text{exprArith } g \text{ p } idx)) ($
 $\lambda\text{- } c. \text{ case } c \text{ of}$

Channel cap - q \Rightarrow integer-of-nat (length q) \geq cap
 | HSChannel - \Rightarrow False) g p)

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

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

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

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

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

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

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

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

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

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

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

| exprArith g p (ExprBinOp BinOpLEq lexpr rexpr) =
 trivCond (exprArith g p lexpr \leq exprArith g p rexpr)

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

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

| exprArith g p (ExprBinOp BinOpAnd lexpr rexpr) =
 trivCond (exprArith g p lexpr \neq 0 \wedge exprArith g p rexpr \neq 0)

| exprArith g p (ExprBinOp BinOpOr lexpr rexpr) =

```

    trivCond (exprArith g p lexpr ≠ 0 ∨ exprArith g p rexpr ≠ 0)

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

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

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

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

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

```

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

fun *liftVar* **where**

```

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

```

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

definition *setVar* = *liftVar* *setVar'*

definition *withVar* = *liftVar* *withVar'*

primrec *withChannel*

```

  where withChannel (ChanRef v) = liftVar withChannel' v

```

lemma *setVar-gState-progress-rel*:

```

  assumes gState-inv prog g

```

shows $(g, \text{fst } (\text{setVar } v \text{ val } g \ p)) \in \text{gState-progress-rel prog}$
unfolding *setVar-def*
by (*cases v*) (*simp add: setVar'-gState-progress-rel[OF assms]*)

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

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

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

5.4 Variable declaration

lemma *channel-inv-code [code]:*
channel-inv (Channel cap ts q)
 $\longleftrightarrow \text{cap} \leq \text{max-array-size}$
 $\wedge 0 \leq \text{cap}$
 $\wedge \text{for-all varType-inv ts}$
 $\wedge \text{length ts} \leq \text{max-array-size}$
 $\wedge \text{length q} \leq \text{max-array-size}$
 $\wedge \text{for-all } (\lambda x. \text{length } x = \text{length ts}$
 $\quad \wedge \text{for-all } (\lambda y. y \geq \text{min-var-value}$
 $\quad \quad \wedge y \leq \text{max-var-value}) x) q$
channel-inv (HChannel ts)
 $\longleftrightarrow \text{for-all varType-inv ts} \wedge \text{length ts} \leq \text{max-array-size}$
by (*auto simp add: for-all-def*) *force+*

primrec *toVariable*
 $:: 'a \text{ gState-scheme} \Rightarrow \text{pState} \Rightarrow \text{varDecl} \Rightarrow \text{String.literal} * \text{variable} * \text{channels}$
where
toVariable g p (VarDeclNum lb hb name siz init) = (
 $\text{let type} = \text{VTBounded lb hb in}$
 $\text{if } \neg \text{varType-inv type then abortv STR "Invalid var def (varType-inv failed):}$
 " name
 $\quad (\lambda-. (\text{name}, \text{Var VTChan } 0, []))$
 else
 let
 $\quad \text{init} = \text{checkVarValue type (case init of}$
 $\quad \quad \text{None} \Rightarrow 0$
 $\quad \quad | \text{Some } e \Rightarrow \text{exprArith } g \ p \ e);$
 $\text{v} = (\text{case siz of}$


```

      None  $\Rightarrow$  Var type init
    | Some s  $\Rightarrow$  if nat-of-integer s  $\leq$  max-array-size
      then VArray type (nat-of-integer s)
        (IArray.tabulate (s,  $\lambda$ -. init))
      else abortv STR "Invalid var def (array too large): " name
        ( $\lambda$ -. Var VTChan 0))
  in
    (name, v, [])

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

```

lemma toVariable-variable-inv:

```

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

```

done

lemma *toVariable-channels-inv*:

shows $\forall x \in \text{set } (\text{snd } (\text{snd } (\text{toVariable } g \ p \ v)))$. *channel-inv* x
by (*cases* v) *auto*

lemma *toVariable-channels-inv'*:

shows $\text{toVariable } g \ p \ v = (a,b,c) \implies \forall x \in \text{set } c$. *channel-inv* x
using *toVariable-channels-inv*
by (*metis* *snd-conv*)

lemma *toVariable-variable-inv'*:

shows $g\text{State-inv prog } g \implies \text{toVariable } g \ p \ v = (a,b,c) \implies \text{variable-inv } b$
by (*metis* *snd-conv* *fst-conv* *toVariable-variable-inv*)

definition *mkChannels*

$:: 'a \ g\text{State-scheme} \Rightarrow p\text{State} \Rightarrow \text{channels} \Rightarrow 'a \ g\text{State-scheme} * p\text{State}$

where

$\text{mkChannels } g \ p \ cs = (\text{if } cs = [] \text{ then } (g,p) \text{ else}$
 $\text{let } l = \text{length } (\text{channels } g) \text{ in}$
 $\text{if } l + \text{length } cs > \text{max-channels}$
 $\text{then abort STR "Too much channels" } (\lambda\cdot. (g,p))$
 else let
 $cs_p = \text{map integer-of-nat } [l..l + \text{length } cs];$
 $g' = g(\text{channels} := \text{channels } g \ @ \ cs);$
 $p' = p(p\text{State.channels} := p\text{State.channels } p \ @ \ cs_p)$
 in
 (g', p'))

lemma *mkChannels-gState-progress-rel*:

$g\text{State-inv prog } g$
 $\implies \text{set } cs \subseteq \text{Collect channel-inv}$
 $\implies (g, \text{fst } (\text{mkChannels } g \ p \ cs)) \in g\text{State-progress-rel prog}$

unfolding *mkChannels-def*

by (*intro* *gState-progress-rel*)
(*auto simp* *add: gState-inv-def gState.defs cl-inv-def*)

lemmas *mkChannels-gState-inv =*

mkChannels-gState-progress-rel[*THEN gState-progress-rel-gState-invI2*]

lemma *mkChannels-pState-inv*:

$p\text{State-inv prog } p$
 $\implies \text{cl-inv } (g,p)$
 $\implies p\text{State-inv prog } (\text{snd } (\text{mkChannels } g \ p \ cs))$

unfolding *mkChannels-def*

including *integer.lifting*

apply (*auto simp* *add: pState-inv-def pState.defs gState-inv-def dest!: cl-inv-lengthD*)
apply (*transfer'*, *simp*)+

done

lemma *mkChannels-cl-inv*:

$cl\text{-}inv (g,p) \implies cl\text{-}inv (mkChannels\ g\ p\ cs)$

unfolding *mkChannels-def*

by (*auto simp add: pState.defs dest: cl-inv-lengthD intro!: cl-invI*)

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

$mkVarChannel\ v\ upd\ g\ p = ($

let

$(k,v,cs) = toVariable\ g\ p\ v;$

$(g',p') = upd\ (lm.update\ k\ v)\ (g,p)$

in

$mkChannels\ g'\ p'\ cs)$

lemma *mkVarChannel-gState-inv*:

assumes *gState-inv prog g*

and $\bigwedge k\ v'\ cs. toVariable\ g\ p\ v = (k,v',cs)$

$\implies gState\text{-}inv\ prog\ (fst\ (upd\ (lm.update\ k\ v')\ (g,p)))$

shows $gState\text{-}inv\ prog\ (fst\ (mkVarChannel\ v\ upd\ g\ p))$

using *assms unfolding mkVarChannel-def*

by (*force split: varDecl.split prod.split*

intro!: mkChannels-gState-inv

dest: toVariable-channels-inv')

lemma *mkVarChannel-gState-progress-rel*:

assumes *gState-inv prog g*

and $\bigwedge k\ v'\ cs. toVariable\ g\ p\ v = (k,v',cs)$

$\implies (g, fst\ (upd\ (lm.update\ k\ v')\ (g,p))) \in gState\text{-}progress\text{-}rel\ prog$

shows $(g, fst\ (mkVarChannel\ v\ upd\ g\ p)) \in gState\text{-}progress\text{-}rel\ prog$

proof –

obtain $k\ v'\ cs$ **where** *1: toVariable g p v = (k,v',cs)* **by** (*metis prod.exhaust*)

obtain $g'\ p'$ **where** *2: (g',p') = upd (lm.update k v') (g,p)* **by** (*metis prod.exhaust*)

with *1 assms have* $*$: $(g, g') \in gState\text{-}progress\text{-}rel\ prog$ **by** (*metis fst-conv*)

also from *1 2 have* $(g', fst\ (mkChannels\ g'\ p'\ cs)) \in gState\text{-}progress\text{-}rel\ prog$

by (*force intro!: mkChannels-gState-progress-rel gState-progress-rel-gState-invI2[OF*

**)*

dest: toVariable-channels-inv')

finally have $(g, fst\ (mkChannels\ g'\ p'\ cs)) \in gState\text{-}progress\text{-}rel\ prog$.

thus *?thesis using 1 2 by (auto simp add: mkVarChannel-def split: prod.split)*

qed

lemma *mkVarChannel-pState-inv*:
assumes *pState-inv prog p*
and *cl-inv (g,p)*
and $\bigwedge k v' cs. \text{toVariable } g \ p \ v = (k, v', cs)$
 $\implies \text{cl-inv } (\text{upd } (\text{lm.update } k \ v') \ (g, p))$
and $\bigwedge k v' cs. \text{toVariable } g \ p \ v = (k, v', cs)$
 $\implies \text{pState-inv prog } (\text{snd } (\text{upd } (\text{lm.update } k \ v') \ (g, p)))$
shows *pState-inv prog (snd (mkVarChannel v upd g p))*
using *assms unfolding mkVarChannel-def*
by (*force split: varDecl.split prod.split*
intro!: mkChannels-pState-inv)

lemma *mkVarChannel-cl-inv*:
assumes *cl-inv (g,p)*
and $\bigwedge k v' cs. \text{toVariable } g \ p \ v = (k, v', cs)$
 $\implies \text{cl-inv } (\text{upd } (\text{lm.update } k \ v') \ (g, p))$
shows *cl-inv (mkVarChannel v upd g p)*
using *assms unfolding mkVarChannel-def*
by (*force split: varDecl.split prod.splits*
intro!: mkChannels-cl-inv)

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
 $v' = \text{case } v \ \text{of}$
 $\quad \text{ProcVarDeclNum } lb \ hb \ \text{name } \text{sz } \text{init} \Rightarrow$
 $\quad \quad \text{VarDeclNum } lb \ hb \ \text{name } \text{sz } \text{init}$
 $\quad | \ \text{ProcVarDeclChan } \text{name } \text{sz} \Rightarrow$
 $\quad \quad \text{VarDeclChan } \text{name } \text{sz } \text{None};$
 $(k, v, cs) = \text{toVariable } g \ p \ v'$
 in
 $\text{mkVarChannel } v' \ (\text{apsnd } \circ \ \text{pState.vars-update}) \ g \ p)$

lemma *mkVarChannelProc-gState-progress-rel*:
assumes *gState-inv prog g*
shows $(g, \text{fst } (\text{mkVarChannelProc } v \ g \ p)) \in \text{gState-progress-rel prog}$
unfolding *mkVarChannelProc-def*
using *assms*
by (*auto intro!: mkVarChannel-gState-progress-rel*)

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

lemma *toVariable-name*:
 $\text{toVariable } g \ p \ (\text{VarDeclNum } lb \ hb \ \text{name } \text{sz } \text{init}) = (x, a, b) \implies x = \text{name}$
 $\text{toVariable } g \ p \ (\text{VarDeclChan } \text{name } \text{sz } t) = (x, a, b) \implies x = \text{name}$
by (*auto split: if-splits*)

```

declare toVariable.simps[simp del]

lemma statesDecls-process-names:
  assumes  $v \in \text{statesDecls } (\text{states prog !! } (pState.idx p))$ 
  shows  $\text{proc VarDeclName } v \in \text{process-names } (\text{states prog !! } (pState.idx p))$ 
   $(\text{processes prog !! } (pState.idx p))$ 

using assms
by (cases processes prog !! (pState.idx p) (auto simp add: statesNames-def))

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

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

```

5.5 Folding

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

definition *step-fold' where*

```

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

```

definition *step-fold where*

```

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

```

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 $step-fold'\ g\ steps\ lbls\ pri\ pos\ nxt\ onxt\ iB =$
 $step-fold'\ g'\ steps'\ lbls'\ pri'\ pos'\ nxt'\ onxt'\ iB'$
unfolding $step-fold'-def$
by (*auto intro: foldr-cong simp add: assms*)

lemma $step-fold-cong[fundef-cong]$:
assumes $lbls = lbls'$
and $pri = pri'$
and $pos = pos'$
and $steps = steps'$
and $nxt = nxt'$
and $onxt = onxt'$
and $iB = iB'$
and $\bigwedge x d. x \in set\ steps \implies g\ x\ d = g'\ x\ d$
shows $step-fold\ g\ steps\ lbls\ pri\ pos\ nxt\ onxt\ iB =$
 $step-fold\ g'\ steps'\ lbls'\ pri'\ pos'\ nxt'\ onxt'\ iB'$
unfolding $step-fold-def$
by (*auto simp: assms cong: step-fold'-cong*)

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

definition $step-foldL$ **where**
 $step-foldL\ g\ stepss\ lbls\ pri\ pos\ nxt\ onxt =$
 $foldr\ (step-foldL-step\ g\ pri\ onxt)\ stepss\ (pos, nxt, lbls, [], [])$

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

```

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

```

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 (λ-. (name, Var VTChan 0)))

```

definition *emptyProc* :: *pState*

— The empty process.

where

emptyProc = (\lfloor *pid = 0, vars = lm.empty (), pc = 0, channels = [], idx = 0* \rfloor)

lemma *vardict-inv-empty*:

vardict-inv ss proc (lm.empty())

unfolding *vardict-inv-def*

by (*simp add: lm.correct*)

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

cl-inv (g, emptyProc)

by (*simp add: cl-inv-def emptyProc-def*)

lemma *emptyProc-pState-inv*:

assumes *program-inv prog*

shows *pState-inv prog emptyProc*

proof —

from *assms* **have** *IArray.length (states prog !! 0) > 0*

```

    by (intro program-inv-length-states) (auto simp add: program-inv-def)
  with assms show ?thesis
    unfolding pState-inv-def program-inv-def emptyProc-def
    by (auto simp add: vardict-inv-empty)
qed

fun mkProc
  :: 'a gState-scheme  $\Rightarrow$  pState
   $\Rightarrow$  String.literal  $\Rightarrow$  expr list  $\Rightarrow$  process  $\Rightarrow$  nat
   $\Rightarrow$  'a gState-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 (λ-. 0)
    in
      — sanity check
      if length args  $\neq$  length argDecls
      then abortv STR "Signature mismatch: " name (λ-. (g, emptyProc))
      else
        let
          — evaluate args (in the context of the calling process)
          eArgs = map (exprArith g p) args;

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

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

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

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

```



```

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

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

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

    by (force dest: nth-mem)

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

  hence P-inv: pState-inv prog (|
    pid = pidN,
    vars = lm.to-map
    ((STR "-pid", Var (VTBounded 0 (integer-of-nat pidN))
      (integer-of-nat pidN))

```

```

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

from p-def have
  varDeclName ' set decls  $\subseteq$ 
    process-names (states prog !! sidx) (processes prog !! sidx)
by auto
with ⟨gState-inv prog g⟩ have
  F-inv:  $\bigwedge p. \llbracket pState\text{-inv prog } p; sidx = pState.idx\ p; cl\text{-inv } (g,p) \rrbracket$ 
     $\implies pState\text{-inv prog}$ 
    (snd (foldl ( $\lambda(g,p) d. mkVarChannel\ d (apsnd \circ pState.vars\text{-update})$ )
g p)
      (g,p) decls))
proof (induction decls arbitrary: g p)
case (Cons d ds) hence
  decl: varDeclName d  $\in$  process-names (states prog !! pState.idx p)
    (processes prog !! pState.idx p)
by simp

obtain g' p' where
  new: (g',p') = (mkVarChannel d (apsnd  $\circ$  pState.vars-update) g p)
by (metis prod.exhaust)

```

hence $p': p' = \text{snd} \dots$ **and** $g': g' = \text{fst} \dots$
 by $(\text{metis } \text{snd-conv } \text{fst-conv})+$
with $\text{Cons.prem}s$ **have** $p\text{State-inv } \text{prog } p'$
 apply $(\text{auto } \text{intro}!: \text{mkVarChannel-pState-inv})$
 apply $(\text{simp } \text{add}: p\text{State-inv-def})$
 apply $(\text{intro } \text{vardict-inv-updateI})$
 apply simp
 apply $(\text{cases } d)$
 apply $(\text{force } \text{dest}!: \text{toVariable-name})$
 apply $(\text{force } \text{dest}!: \text{toVariable-name})$
 apply $(\text{intro } \text{toVariable-variable-inv}')$
 apply $\text{assumption}+$
done
moreover
from $p' \text{ Cons.prem}s$ **have** $p\text{State.idx } p' = \text{sid}x$
 by $(\text{auto } \text{simp } \text{add}: \text{mkVarChannel-def } \text{mkChannels-def } \text{split}: \text{prod.split})$
moreover
from $\text{new } \text{Cons.prem}s$ **have** $\text{cl-inv } (g', p')$
 by $(\text{auto } \text{intro}!: \text{mkVarChannel-cl-inv})$
moreover
from $g' \text{ Cons.prem}s$ **have** $g\text{State-inv } \text{prog } g'$
 by $(\text{auto } \text{intro}!: \text{mkVarChannel-gState-inv})$
moreover
from $\text{Cons.prem}s$ **have**
 $\text{varDeclName } ' \text{set } ds \subseteq$
 $\text{process-names } (\text{states } \text{prog } !! \text{sid}x) (\text{processes } \text{prog } !! \text{sid}x)$
 by simp
ultimately
have
 $p\text{State-inv } \text{prog}$
 $(\text{snd } (\text{foldl } (\lambda(g,p) d. \text{mkVarChannel } d (\text{apsnd } \circ p\text{State.vars-update}) g p)$
 $(g', p') ds))$
 using $\text{Cons.IH}[\text{of } p' g']$ **by** $(\text{simp } \text{add}: \text{o-def})$
 with new **show** $?case$ **by** $(\text{simp } \text{add}: \text{o-def})$
qed simp

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

lemma mkProc-cl-inv :
assumes $\text{cl-inv } (g, p)$
shows $\text{cl-inv } (\text{mkProc } g p \text{ name } \text{args } (\text{processes } \text{prog } !! \text{sid}x) \text{pid}N)$
proof –
note $I\text{Array.sub-def } [\text{simp } \text{del}]$
obtain $\text{sid}x'$ start argDecls decls
where $[\text{simp}]$: $\text{processes } \text{prog } !! \text{sid}x = (\text{sid}x', \text{start}, \text{argDecls}, \text{decls})$

```

    by (metis prod.exhaust)

  have
    P-inv:  $\bigwedge s v. cl\text{-}inv (g, (\!pid = pidN, vars = v, pc = s, channels = [], idx = sidx' \!))$ 
    by (simp add: cl-inv-def)

  have
     $\bigwedge p. cl\text{-}inv(g,p) \implies$ 
       $cl\text{-}inv (foldl (\lambda(g,p) d. mkVarChannel d (apsnd \circ pState.vars\text{-}update) g p)$ 
         $(g,p) decls)$ 
  proof (induction decls arbitrary: g p)
    case (Cons d ds)
    obtain g' p' where
      new:  $(g',p') = (mkVarChannel d (apsnd \circ pState.vars\text{-}update) g p)$ 
    by (metis prod.exhaust)
    with Cons.prem1 have cl-inv (g',p')
    by (auto intro!: mkVarChannel-cl-inv)

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

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

declare mkProc.simps[simp del]

definition runProc
  :: String.literal  $\Rightarrow$  expr list  $\Rightarrow$  program
   $\Rightarrow$  'a gState-scheme  $\Rightarrow$  pState
   $\Rightarrow$  'a gState-scheme * pState
where
  runProc name args prog g p = (
    if length (procs g)  $\geq$  max-procs
    then abort STR "Too many processes" ( $\lambda\cdot. (g,p)$ )
    else let pid = length (procs g) + 1 in
      case lm.lookup name (proc-data prog) of
        None  $\Rightarrow$  abortv STR "No such process: " name
          ( $\lambda\cdot. (g,p)$ )
      | Some proc-idx  $\Rightarrow$ 
        let (g', proc) = mkProc g p name args (processes prog !! proc-idx) pid
        in (g'(\!procs := procs g @ [proc]\!), p))

lemma runProc-gState-progress-rel:
  assumes program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  and cl-inv (g,p)
  shows (g, fst (runProc name args prog g p))  $\in$  gState-progress-rel prog

```

```

proof (cases length (procs g) < max-procs)
  note IArray.sub-def [simp del]

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

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

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

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

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

lemma runProc-pState-inv:
  assumes pState-inv prog p
  shows pState-inv prog (snd (runProc name args prog g p))

```

by (simp add: assms runProc-pState-id)

lemma runProc-cl-inv:

assumes program-inv prog

and gState-inv prog g

and pState-inv prog p

and cl-inv (g,p)

shows cl-inv (runProc name args prog g p)

proof –

obtain $g' p'$ where *: runProc name args prog g p = (g',p')

by (metis prod.exhaust)

with runProc-gState-progress-rel[OF assms, of name args] have

length (channels g) ≤ length (channels g')

by (simp add: gState-progress-rel-def)

moreover from * runProc-pState-id have $p' = p$ by (metis snd-conv)

ultimately show ?thesis by (metis ⟨cl-inv (g,p)⟩ * cl-inv-trans)

qed

5.7 AST to edges

type-synonym ast = AST.module list

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

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

definition atomize :: nat ⇒ nat ⇒ edge list ⇒ edge list **where**

atomize lp hp es = fold (λe es.

let e' = case target e of

LabelJump - None ⇒

- Labels are checked again later on, when they
- are going to be resolved. Hence it is safe to say
- *atomic* here, especially as the later algorithm
- relies on targets in atomic blocks to be marked as such.

e⟦ atomic := InAtomic ⟧

| LabelJump - (Some via) ⇒

if $lp \leq via \wedge hp \geq via$ then e⟦ atomic := Atomic ⟧
else e⟦ atomic := InAtomic ⟧

| Index p' ⇒

if $lp \leq p' \wedge hp \geq p'$ then e⟦ atomic := Atomic ⟧
else e⟦ atomic := InAtomic ⟧

in e'#es) es []

fun skip — No-(edge)

where

skip (lbls, pri, pos, nxt, -) =

```

([[([cond = ECEExpr (ExprConst 1),
    effect = EEId, target = nxt, prio = pri,
    atomic = NonAtomic)]], Index pos, lbls)

```

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

Parameters used:

lbls Map of Labels

pri Current priority

pos Current position in the array

nxt Next state

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

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

fun *stepToState*

 :: *step*

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

 ⇒ edge list list * edgeIndex * labels

and *stmntToState*

 :: *stmnt*

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

 ⇒ edge list list * edgeIndex * labels

where

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

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

 let

 — the *unless* part

 (ues,-,lbls') = *stmntToState* u (lbls, pri, pos, nxt, onxt, True);

 u = last ues; ues = butlast ues;

 pos' = pos + length ues;

 — find minimal current priority

 pri = min-prio u pri;

 — the guarded part –

 — priority is decreased, because there is now a new unless part with

 — higher prio

 (ses,spos,lbls'') = *stmntToState* s (lbls', pri - 1, pos', nxt, onxt, False);

 — add an edge to the unless part for each generated state

 ses = map (List.append u) ses

 in

 (ues@ses,spos,lbls'')

| *stepToState* (StepDecl decls) (lbls, pri, pos, nxt, onxt, -) = (

```

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

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

| stmtnToState (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'))

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

    — We don't resolve goto-chains. If the labeled stmtnt 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'))

| stmtnToState (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)
  )

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

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

```


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

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

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

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

| *stmntToState* *StmntBreak* (*lbls, pri, -, -, Some onxt, -*) =
 ([[([*cond* = *ECTrue*, *effect* = *EEGoto*, *target* = *onxt*, *prio* = *pri*,
atomic = *NonAtomic*])], *onxt, lbls*)

| *stmntToState* *StmntBreak* (*-, -, -, None, -*) =
*abort STR "Misplaced break" (λ-. ([], *Index 0, lm.empty*()))*

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

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

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

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

| *stmntToState* *StmntSkip* *d = skip d*

5.7.1 Setup

definition *endState* :: *edge list where*

— An extra state added to each process marking its end.

endState = ([[*cond* = *ECFalse*, *effect* = *EEEnd*, *target* = *Index 0*, *prio* = *0*,
atomic = *NonAtomic*]]

definition *resolveLabel* :: *String.literal* \Rightarrow *labels* \Rightarrow *nat* **where**

```

resolveLabel l lbls = (
  case lm.lookup l lbls of
    None  $\Rightarrow$  abortv STR "Unresolved label: " l (λ-. 0)
  | Some pos  $\Rightarrow$  pos)

```

primrec *resolveLabels* :: *edge list list* \Rightarrow *labels* \Rightarrow *edge list* \Rightarrow *edge list* **where**

```

resolveLabels - - [] = []
| resolveLabels edges lbls (e#es) = (
  let check-atomic = λpos. fold (λe a. a ∧ inAtomic e) (edges ! pos) True in
  case target e of
    Index -  $\Rightarrow$  e
  | LabelJump l None  $\Rightarrow$ 
    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)  $\Rightarrow$ 
    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* \Rightarrow (*integer* * *edge list*) *list* **where**

```

calculatePrios ess = map (λes. (min-prio es 0, es)) ess

```

definition *toStates* :: *step list* \Rightarrow *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 -  $\Rightarrow$  pos
      | LabelJump l -  $\Rightarrow$  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  $\text{IArray.length } ss > 0$ 
  using assms
  unfolding toStates-def calculatePrios-def
  by (auto split: prod.splits edgeIndex.splits if-splits)

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)  $\Rightarrow \text{nat-of-integer } x$ )
      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
by (cases p (simp-all split: prod.splits))

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

lemma toProcess-start:
  toProcess sidx p = (ss,a,n,l,idx,start,r)
   $\Longrightarrow \exists s. \text{start} = \text{Index } s \wedge s < \text{IArray.length } ss$ 
by (cases p (force split: prod.splits dest: toStates-inv(1)))+

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

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

definition setUp ::  $\text{ast} \Rightarrow \text{program} \times \text{gState}$  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 (rule *program-invI*, goal-cases)

case 1 show ?case **by** (simp add: setUp-def split: prod.split)

next

case 2 show ?case **by** (simp add: setUp-def split: prod.split)

next

case 3 thus ?case

by (auto simp add: setUp-def o-def

split: prod.splits

dest!: toProcess-states-nonempty)

next

case 4 thus ?case

unfolding setUp-def

by (auto simp add: lm.correct o-def in-set-enumerate-eq nth-enumerate-eq

dest!: subsetD[OF Misc.ran-map-of] toProcess-sidx

split: prod.splits)

next

```

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

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

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

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

    have g1: gState-inv prog ?g
    by (simp add: gState-inv-def max-channels-def lm-correct max-var-value-def)
    {
      fix g decls
      assume gState-inv prog g
      hence gState-inv prog (foldl (λg d.
        fst (mkVarChannel d (apfst ∘ gState.vars-update) g emptyProc)
        ) g decls)
      apply (rule foldl-rule)
      apply (intro mkVarChannel-gState-inv)
      apply simp
      apply (frule-tac g=σ in toVariable-variable-inv')
      apply assumption
      apply (auto simp add: gState-inv-def lm.correct)
      done
    }
    note g2 = this[OF g1]

    {
      fix g :: 'a gState-scheme and pre-procs
      assume gState-inv prog g
      hence gState-inv prog (foldl (λg (-,a,name,-).
        foldl (λg name.

```

```

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

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

```

5.8 Semantic Engine

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

definition *assertVar* \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 ( $\lambda$ - 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 ( $\lambda$ i 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 ( $\lambda$ i c. case c of
      HSChannel -  $\Rightarrow$  i = h
      | Channel - -  $\Rightarrow$  False) g l
| evalHandshake - h -  $\longleftrightarrow$  h = 0

```

```

primrec evalEffect
  :: edgeEffect  $\Rightarrow$  program  $\Rightarrow$  gStateI  $\Rightarrow$  pState  $\Rightarrow$  gStateI * pState
where
  evalEffect EEEnd - g l = (g,l)
| evalEffect EEId - g l = (g,l)
| evalEffect EEGoto - g l = (g,l)
| evalEffect EEAssign v e - g l = setVar v (exprArith g l e) g l
| evalEffect EEDecl d - g l = mkVarChannelProc d g l
| evalEffect EERun name args prog g l = runProc name args prog g l
| evalEffect EEAssert e - g l = (
  if exprArith g l e = 0
  then setVar assertVar 1 g l
  else (g,l))
| evalEffect (EESend v es srt) - g l = withChannel v ( $\lambda$ i c.
  let
    ab =  $\lambda$ -. abort STR "Length mismatch on sending." ( $\lambda$ -. (g,l));
    es = map (exprArith g l) es
  in
    if  $\neg$  for-all ( $\lambda$ x. x  $\geq$  min-var-value  $\wedge$  x  $\leq$  max-var-value) es
    then abort STR "Invalid Channel" ( $\lambda$ -. (g,l))
    else
      case c of
        Channel cap ts q  $\Rightarrow$ 
          if length ts  $\neq$  length es  $\vee$   $\neg$  (length q < max-array-size)
          then ab()
          else let
            q' = if  $\neg$  srt then q@[es]
                  else let

```

```

      q = map lexmlist q;
      q' = insort (lexlist es) q
    in map unlex q';
    g = gState.channels-update (λcs.
      cs[ i := Channel cap ts q' ]) g
  in (g,l)
| HSChannel ts ⇒
  if length ts ≠ length es then ab()
  else (g(|hsdata := es, handshake := i|), l)
| InvChannel ⇒ abort STR "Trying to send on invalid channel" (λ-. (g,l))
) g l
| evalEffect (EERcv v rs srt rem) - g l = withChannel v (λi c.
  case c of
  Channel cap ts qs ⇒
    if qs = [] then abort STR "Recv from empty channel" (λ-. (g,l))
    else
      let
        (q', qs^∧) = if ¬ srt then (hd qs, tl qs)
                    else apfst the (find-remove (recvArgsCheck g l rs) qs);
        (g,l) = evalRecvArgs rs q' g l;
        g = if rem
            then gState.channels-update (λcs. cs[ i := Channel cap ts qs^∧ ]) g
            else g
          — messages are not removed – so no need to update anything
        in (g,l)
  | HSChannel - ⇒
    let (g,l) = evalRecvArgs rs (hsdata g) g l in
    let g = g(|handshake := 0, hsdata := [] |)
    in (g,l)
  | InvChannel ⇒ abort STR "Receiving on invalid channel" (λ-. (g,l))
) g l

```

lemma *statesDecls-effect*:

assumes $ef \in \text{effect } \text{'edgeSet } ss$

and $ef = \text{EEDecl } d$

shows $d \in \text{statesDecls } ss$

proof –

from *assms* **obtain** e **where** $e \in \text{edgeSet } ss$ $ef = \text{effect } e$ **by** *auto*

thus *?thesis* **using** *assms*

unfolding *statesDecls-def*

by (*auto simp add: edgeDecls-def*

intro!: *bexI*[**where** $x = e$]

split: *edgeEffect.split*)

qed

lemma *evalRecvArgs-pState-inv*:

assumes *pState-inv prog p*

shows *pState-inv prog (snd (evalRecvArgs rargs xs g p))*

using *assms*

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

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

lemma *evalRecvArgs-gState-progress-rel*:
assumes *gState-inv prog g*
shows $(g, fst (evalRecvArgs rargs xs g p)) \in gState-progress-rel prog$
using *assms*

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

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

lemma *evalRecvArgs-cl-inv*:
assumes *cl-inv (g,p)*
shows *cl-inv (evalRecvArgs rargs xs g p)*
using *assms*
proof (*induction rargs xs arbitrary: p g rule: list-induct2'*)
case ($\lambda r rs x xs$) **thus** ?case
proof (*cases r*)
case (*RecvArgVar v*)
with λ **have** *cl-inv (setVar v x g p)*

```

    by (auto intro!: setVar-cl-inv)
  with 4.IH RecvArgVar show ?thesis
    by (auto split: prod.splits)
qed simp-all
qed simp-all

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

lemma evalEffect-gState-progress-rel:
  assumes program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  and cl-inv (g,p)
  shows (g, fst (evalEffect e prog g p)) ∈ gState-progress-rel prog
  using assms
proof (cases e)
  case EEAssert
  with assms show ?thesis
    by (auto intro: setVar-gState-progress-rel)
next
  case EEAssign

```

```

with assms show ?thesis
  by (auto intro: setVar-gState-progress-rel)
next
case EEDecl
with assms show ?thesis
  by (auto intro: mkVarChannelProc-gState-progress-rel)
next
case EERun
with assms show ?thesis
  by (auto intro: runProc-gState-progress-rel)
next
case (EESend c es srt)
then obtain v where v: c = ChanRef v by (metis chanRef.exhaust)
obtain idx where idx: nat-of-integer (the (getVar v g p)) = idx by blast
note idx' = idx[symmetric, unfolded getVar-def]
show ?thesis
proof (cases idx < length (gState.channels g))
  case True
    note DEF = True EESend v idx' assms

show ?thesis
proof (cases gState.channels g ! idx)
  case (Channel cap ts q)
    with True have Channel cap ts q ∈ set (gState.channels g)
      by (metis nth-mem)
    with assms have channel-inv (Channel cap ts q)
      by (auto simp add: gState-inv-def simp del: channel-inv.simps)
    with Channel DEF show ?thesis
      by (cases v)
        (auto simp add: withChannel'-def withVar'-def for-all-def
          split: channel.split
          intro!: gState-progress-rel-channels-update)
  next
    case HChannel with DEF show ?thesis
      by (cases v)
        (auto simp: withChannel'-def withVar'-def gState-progress-rel-def
          gState-inv-def
          split: channel.split)
  next
    case InvChannel with DEF show ?thesis
      by (cases v) (auto simp add: withChannel'-def withVar'-def)
qed
next
case False with EESend idx' v assms show ?thesis
  by (cases v) (auto simp add: withChannel'-def withVar'-def)
qed
next
case (EERecv c rs srt rem)
then obtain v where v: c = ChanRef v by (metis chanRef.exhaust)

```

```

obtain  $idx$  where  $idx$ : nat-of-integer (the (getVar  $v$   $g$   $p$ )) =  $idx$  by blast
note  $idx' = idx$ [symmetric, unfolded getVar-def]
show ?thesis
proof (cases  $idx < \text{length } (gState.channels\ g)$ )
  case True
    note  $DEF = \text{True } EERecv\ v\ idx'$  assms
    show ?thesis
    proof (cases  $gState.channels\ g\ !\ idx$ )
      note channel-inv.simps[simp del]
      case (Channel cap ts q)
        with True have Channel cap ts q  $\in$  set ( $gState.channels\ g$ )
          by (metis nth-mem)
        with assms have c-inv: channel-inv (Channel cap ts q)
          by (auto simp add: gState-inv-def simp del: channel-inv.simps)
        moreover obtain  $res\ q'$  where
          apfst the (find-remove (recvArgsCheck  $g\ p\ rs$ )  $q$ ) = ( $res, q'$ )
          by (metis prod.exhaust)
        moreover hence  $q' = \text{snd } (\text{find-remove } (\text{recvArgsCheck } g\ p\ rs)\ q)$ 
          by (simp add: apfst-def map-prod-def split: prod.splits)
        with find-remove-subset find-remove-length have
           $set\ q' \subseteq set\ q$   $length\ q' \leq length\ q$ 
          by (metis surjective-pairing)+
        with c-inv have channel-inv (Channel cap ts q')
          by (auto simp add: channel-inv.simps)
        moreover {
          assume  $q \neq []$ 
          hence  $set\ (tl\ q) \subseteq set\ q$  using tl-subset by auto
          with c-inv have channel-inv (Channel cap ts (tl q))
            by (auto simp add: channel-inv.simps)
        }
      moreover {
        fix  $res\ g'\ p'$ 
        assume evalRecvArgs rs res g p = (g',p')
        with evalRecvArgs-gState-progress-rel assms have
           $(g, g') \in gState-progress-rel\ prog$ 
          by (metis fst-conv)
        hence  $length\ (channels\ g) \leq length\ (channels\ g')$ 
          by (simp add: gState-progress-rel-def)
      }
    ultimately show ?thesis using Channel DEF
      apply (cases v)
      apply (auto simp add: withChannel'-def withVar'-def for-all-def
        split: channel.split prod.split
        elim: fstE
        intro!: evalRecvArgs-gState-progress-rel
        gState-progress-rel-channels-update-step)
      apply force+
    done
  next

```

```

case HChannel
obtain  $g' p'$  where  $*$ : evalRecvArgs rs (hsdata  $g$ )  $g p = (g', p')$ 
  by (metis prod.exhaust)
with assms have  $(g, g') \in gState\text{-}progress\text{-}rel\ prog$ 
  by (auto elim!: fstE intro!: evalRecvArgs-gState-progress-rel)
also hence  $gState\text{-}inv\ prog\ g'$  by blast
hence  $(g', g'(\backslash handshake := 0, hsdata := [])) \in gState\text{-}progress\text{-}rel\ prog$ 
  by (auto simp add: gState-progress-rel-def gState-inv-def)
finally
have  $(g, g'(\backslash handshake := 0, hsdata := [])) \in gState\text{-}progress\text{-}rel\ prog .$ 
with DEF HChannel * show ?thesis
  by (cases v) (auto simp add: withChannel'-def withVar'-def for-all-def
    split: channel.split prod.split)

next
case InvChannel with DEF show ?thesis
  by (cases v) (auto simp add: withChannel'-def withVar'-def)
qed
next
case False with EERecv idx' v assms show ?thesis
  by (cases v) (auto simp add: withChannel'-def withVar'-def)
qed
qed simp-all

lemma evalEffect-cl-inv:
  assumes cl-inv  $(g, p)$ 
  and program-inv prog
  and gState-inv prog g
  and pState-inv prog p
  shows cl-inv (evalEffect e prog g p)
  using assms
proof (cases e)
  case EERun with assms show ?thesis by (force intro!: runProc-cl-inv)
next
case (EESend c es srt) then obtain  $v$  where ChanRef  $v = c$  by (cases c) simp
with EESend assms show ?thesis
  by (cases v)
    (auto simp add: withChannel'-def withVar'-def split: channel.split
      intro!: cl-inv-channels-update)
next
case (EERecv c es srt) then obtain  $v$  where ChanRef  $v = c$  by (cases c) simp
with EERecv assms show ?thesis
  apply (cases v)
  apply (auto simp add: withChannel'-def withVar'-def split: channel.split prod.split
    intro!: cl-inv-channels-update)
  apply (metis evalRecvArgs-cl-inv) +
  done
qed (simp-all add: setVar-cl-inv mkVarChannelProc-cl-inv)

```

5.8.2 Executable edges

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

type-synonym $choices = (edge * pState) list$

— A choice is an executable edge and the process it belongs to.

definition $getChoices :: gState_I \Rightarrow pState \Rightarrow edge list \Rightarrow choices$ **where**

$getChoices\ g\ p = foldl\ (\lambda E\ e.$

$\text{if } evalHandshake\ (cond\ e)\ (handshake\ g)\ g\ p \wedge evalCond\ (cond\ e)\ g\ p$

$\text{then } (e,p)\#E$

$\text{else } E) []$

lemma $getChoices\text{-sub-edges-}fst$:

$fst\ 'set\ (getChoices\ g\ p\ es) \subseteq set\ es$

unfolding $getChoices\text{-def}$

by $(rule\ foldl\text{-rule-}aux) auto$

lemma $getChoices\text{-sub-edges}$:

$(a,b) \in set\ (getChoices\ g\ p\ es) \implies a \in set\ es$

using $getChoices\text{-sub-edges-}fst$

by $force$

lemma $getChoices\text{-p-}snd$:

$snd\ 'set\ (getChoices\ g\ p\ es) \subseteq \{p\}$

unfolding $getChoices\text{-def}$

by $(rule\ foldl\text{-rule-}aux) auto$

lemma $getChoices\text{-p}$:

$(a,b) \in set\ (getChoices\ g\ p\ es) \implies b = p$

using $getChoices\text{-p-}snd$

by $force$

definition $sort\text{-by-pri}$ **where**

$sort\text{-by-pri}\ min\text{-pri}\ edges = foldl\ (\lambda es\ e.$

$\text{let } idx = nat\text{-of-integer}\ (abs\ (prio\ e))$

$\text{in if } idx > min\text{-pri}$

$\text{then abort STR "Invalid priority" } (\lambda\text{-. } es)$

$\text{else let } ep = e \# (es ! idx) \text{ in } es[idx := ep]$

$) (replicate\ (min\text{-pri} + 1) [])\ edges$

lemma $sort\text{-by-pri-edges}'$:

assumes $set\ edges \subseteq A$

shows $set\ (sort\text{-by-pri}\ min\text{-pri}\ edges) \subseteq \{xs.\ set\ xs \subseteq A\}$

using $assms$

unfolding $sort\text{-by-pri-}def$

apply $(rule\text{-tac}\ I=\lambda\sigma\text{-. } (\forall x \in set\ \sigma.\ set\ x \subseteq A) \wedge length\ \sigma = min\text{-pri} + 1)$ **in** $foldl\text{-rule-}aux\text{-}P$

apply $simp$

```

apply (force dest!: subsetD[OF set-update-subset-insert]
        split: if-splits)
apply force
done

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

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

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(gStateI.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 ( $\lambda(E,pri,-)$ . E = []  $\wedge$  pri  $\leq$  min-pri) ( $\lambda(-, pri, ELSE)$ ). 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

unfolding *while-rel1-def*

by *auto*

definition

```

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

```

lemma *wf-while-rel2*:

wf (while-rel2 mp)

unfolding *while-rel2-def*

by *auto*

lemma *executable-edgeSet*:

assumes *gState-inv prog g*

and *program-inv prog*

and *ss = states prog*

shows *executable ss g*

```

≤ SPEC (λcs. ∀ (e,p) ∈ set cs.
  e ∈ edgeSet (ss !! pState.idx p)
  ∧ pState-inv prog p
  ∧ cl-inv (g,p))

```

unfolding *executable-def*

using *assms*

apply (*refine-rcg refine-vcg nfoldli-rule*[**where**

```

  I=λ- - cs. ∀ (e,p) ∈ set cs.
    e ∈ edgeSet (ss !! pState.idx p)
    ∧ pState-inv prog p
    ∧ cl-inv (g,p)])

```



```

apply simp
apply (rename-tac p l1 l2  $\sigma$  e p')
apply (subgoal-tac pState-inv prog p)
apply (clarsimp simp add: edgeSet-def pState-inv-def)[]
apply (rule-tac x=IArray.list-of
      (IArray.list-of (states prog) ! pState.idx p) ! pc p in bezI)

apply simp
apply (force intro!: nth-mem)
apply (force simp add: gState-inv-def)
apply (refine-rcg refine-vcg wf-while-rel1 WHILET-rule[where
      I= $\lambda(cs,-,-).$   $\forall (e,p) \in \text{set } cs.$ 
      e  $\in$  edgeSet (ss !! pState.idx p)
       $\wedge$  pState-inv prog p
       $\wedge$  cl-inv (g,p)
      and R=while-rel1])
apply (vc-solve solve: asm-rl simp add: while-rel1-def)
apply (frule getChoices-p)
using getChoices-sub-edges apply (force simp add: gState-inv-def)
using sort-by-pri-edges apply fastforce
apply (rename-tac min-pri pri-edges)
apply (rule-tac I= $\lambda(cs,-,-).$   $\forall (e,p) \in \text{set } cs.$ 
      e  $\in$  edgeSet (ss !! pState.idx p)
       $\wedge$  pState-inv prog p
       $\wedge$  cl-inv (g,p)
      and R=while-rel2 (nat-of-integer (abs min-pri))
      in WHILET-rule)
apply (simp add: wf-while-rel2)
apply simp
apply (refine-rcg refine-vcg)
apply (clarsimp-all split: prod.split simp add: while-rel2-def)
apply (metis diff-less-mono lessI)
apply (rename-tac idx' else e p')
apply (frule getChoices-p)
apply (frule getChoices-sub-edges)
apply (subgoal-tac
      sort-by-pri (nat-of-integer |min-pri|) pri-edges ! idx'
       $\in$  set (sort-by-pri (nat-of-integer |min-pri|) pri-edges))
apply (auto simp add: assms gState-inv-def)[]
apply (force simp add: sort-by-pri-length)
apply (auto simp add: assms)
done

```

lemma *executable-edgeSet'*:
assumes gState-inv prog g
and program-inv prog
shows executable (states prog) g
 \leq SPEC ($\lambda cs. \forall (e,p) \in \text{set } cs.$
e \in edgeSet ((states prog) !! pState.idx p)
 \wedge pState-inv prog p)

$\wedge \text{cl-inv}(g,p)$

using *executable-edgeSet*[**where** $ss = \text{states prog}$] *assms*
by *simp*

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

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

5.8.3 Successor calculation

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

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

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

lemma *gState-inv-to_I*:
 $gState\text{-inv}\ prog\ g = gState\text{-inv}\ prog\ (to_I\ g)$
by (*cases g, simp add: gState-inv-def cl-inv-def*)

lemma *gState-inv-from_I*:
 $gState\text{-inv}\ prog\ g = gState\text{-inv}\ prog\ (from_I\ g)$
by (*cases g, simp add: gState-inv-def cl-inv-def*)

lemma *gState-inv-reset_I*:
 $gState\text{-inv}\ prog\ g = gState\text{-inv}\ prog\ (reset_I\ g)$
by (*cases g, simp add: gState-inv-def cl-inv-def*)

lemmas *gState-inv-I-simps* =

gState-inv-to_I gState-inv-from_I gState-inv-reset_I

definition *removeProcs*

— Remove ended processes, if there is no running one with a higher pid.

where

removeProcs ps = foldr (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

unfolding *removeProcs-def*

apply (*subst foldr-conv-foldl*)

apply (*rule foldl-rule-aux-P[where I=λ(-, -, ps', -) -. set ps' ⊆ set ps]*)

apply *simp*

apply (*clarsimp split: if-splits*)

apply *force*

apply (*rename-tac p*)

apply (*case-tac p = x*)

apply (*subst set-rev[symmetric]*)

apply *simp*

apply *force*

apply *force*

done

lemma *removeProcs-length'*:

length (fst (snd (snd (removeProcs ps)))) ≤ length ps

unfolding *removeProcs-def*

apply (*subst foldr-conv-foldl*)

apply (*rule foldl-rule-aux-P[where*

I=λ(-, -, ps', -) ps''. length ps' + length ps'' ≤ length ps])

apply (*auto split: if-splits*)

done

lemma *removeProcs-subset*:

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

using *removeProcs-subset'*

by (*metis fst-conv snd-conv*)

lemma *removeProcs-length*:

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

using *removeProcs-length'*

by (*metis fst-conv snd-conv*)

definition *cleanChans :: integer list ⇒ channels ⇒ channels*

— Mark channels of closed processes as invalid.

where

cleanChans dchans cs = snd (foldl (λ(i, cs) c.

if $List.member\ dchans\ i$ then $(i + 1, cs@[InvChannel])$
 else $(i + 1, cs@[c]) (0, []) cs$

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

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

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

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

lemma *gState-progress-rel-exclusive*:
 $(g, g') \in gState\ progress\ rel\ prog$
 $\implies (g, g'(\ exclusive := p)) \in gState\ progress\ rel\ prog$
by (*simp add: gState-progress-rel-def gState-inv-def cl-inv-def*)

definition *applyEdge*
 $:: program \Rightarrow edge \Rightarrow pState \Rightarrow gState_I \Rightarrow gState_I\ nres$
where

```

applyEdge prog e p g = do {

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

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

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

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

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

```

lemma *applyEdge-gState-progress-rel:*

```

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

```

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

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

```

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

```

```

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

```

apply (*frule gState-progress-rel-gState-invI2*)

```

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

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

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

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

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

```

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

definition *nexts*
 :: *program* ⇒ *gState* ⇒ *gState ls nres*
 — The successor function

where

```

nexts prog g = (
  let
    f = fromI;
    g = toI g
  in

```

```

  REC (λD g. do {
    E ← executable (states prog) g;
    if E = [] then

```

```

    if handshake  $g \neq 0$  then
      — HS not possible – remove current step
      RETURN (ls.empty())
    else if exclusive  $g \neq 0$  then
      — Atomic blocks – just return current state
      RETURN (ls.sng (f g))
    else if  $\neg$  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  $g = \text{reset}_I g$  in
    nfoldli E ( $\lambda \cdot \text{True}$ ) ( $\lambda(e,p) G$ .
      applyEdge prog e p g  $\gg$  ( $\lambda g'$ .
        if handshake  $g' \neq 0 \vee \text{isAtomic } e$  then do {
           $G_R \leftarrow D g'$ ;
          if ls.isEmpty  $G_R \wedge \text{handshake } g' = 0$  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
   $\gg$  ( $\lambda G$ . if ls.isEmpty G then RETURN (ls.sng (f g)) else RETURN G)
)

```

lemma *gState-progress-rel-intros:*

```

(to_I g, gI') ∈ gState-progress-rel prog
   $\implies$  (g, from_I gI') ∈ gState-progress-rel prog
(gI, gI') ∈ gState-progress-rel prog
   $\implies$  (gI, reset_I gI') ∈ gState-progress-rel prog
(to_I g, gI') ∈ gState-progress-rel prog
   $\implies$  (to_I g, gI'(timeout := t)) ∈ gState-progress-rel prog

```

unfolding *gState-progress-rel-def*

by (cases g, cases gI', cases gI, force simp add: gState-inv-def cl-inv-def)+

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

```

(to_I g, g') ∈ gState-progress-rel prog
   $\implies$  (reset_I g', g'') ∈ gState-progress-rel prog
   $\implies$  (g, from_I g'') ∈ gState-progress-rel prog
(to_I g, g') ∈ gState-progress-rel prog
   $\implies$  (reset_I g', g'') ∈ gState-progress-rel prog
   $\implies$  (to_I g, g'') ∈ gState-progress-rel prog

```

unfolding *gState-progress-rel-def*
by (*cases g*, *cases g'*, *cases g''*, *force simp add: gState-inv-def cl-inv-def*)⁺

lemma *cl-inv-reset_I*:
 $cl\text{-}inv(g,p) \implies cl\text{-}inv(reset_I\ g, p)$
by (*cases g*) (*simp add: cl-inv-def*)

lemmas *refine-helpers* =
gState-progress-rel-intros gState-progress-rel-step-intros cl-inv-reset_I

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

lemma *RETURN-dRETURN*:
 $RETURN\ f \leq f' \implies nres\text{-}of\ (dRETURN\ f) \leq f'$
unfolding *nres-of-def*
by *simp*

lemma *executable-dRETURN*:
 $nres\text{-}of\ (dRETURN\ (executable\text{-}impl\ prog\ g)) \leq executable\ prog\ g$
using *executable-impl.refine*
by (*simp add: RETURN-dRETURN*)

lemma *applyEdge-dRETURN*:
 $nres\text{-}of\ (dRETURN\ (applyEdge\text{-}impl\ prog\ e\ p\ g)) \leq applyEdge\ prog\ e\ p\ g$
using *applyEdge-impl.refine*

by (simp add: RETURN-dRETURN)

schematic-goal *nexts-code-aux*:

nres-of (?nexts prog g) \leq nexts prog g

unfolding *nexts-def*

by (refine-transfer the-resI executable-dRETURN applyEdge-dRETURN)

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

5.8.4 Handle non-termination

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

definition *SUCCEED-abort where*

SUCCEED-abort msg dm m = (

case m of

RES X \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

dSUCCEEDi \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

unfolding *dSUCCEED-abort-def SUCCEED-abort-def ref-succeed-def*

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

The final successor function now incorporates:

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

definition *nexts-code where*

nexts-code prog g =

the-res (dSUCCEED-abort (STR "The Universe is broken!")

(dRETURN (ls.sng g))

(nexts-code-aux prog g))

```

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

```

5.9 Finiteness of the state space

```

inductive-set reachable-states
  for  $P :: \text{program}$ 
  and  $g_s :: gState$  — start state
where
 $g_s \in \text{reachable-states } P \ g_s \mid$ 
 $g \in \text{reachable-states } P \ g_s \implies x \in ls.\alpha$  (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 (rule finite-subset[OF - gStates-finite[of - g]])
  define INV where  $INV \ g' \longleftrightarrow g' \in (gState-progress-rel \ prog)^* \ \{g\} \wedge gState-inv$ 
  prog g' for g'

```

```

{
  fix  $g'$ 
  have  $g' \in \text{reachable-states } prog \ g \implies INV \ g'$ 
  proof (induct rule: reachable-states-induct)
    case init with assms show ?case by (simp add: INV-def)
  next
    case (step g g')
    from step(2,3) have
       $(g, g') \in gState-progress-rel$  prog
      using nexts-code-SPEC[OF - (program-inv prog)]
      unfolding INV-def by auto
    thus ?case using step(2) unfolding INV-def by auto
  qed
}

```

```

thus reachable-states prog g  $\subseteq$ 

```

```

    (gState-progress-rel prog)* “ {g}
  unfolding INV-def by auto
qed

```

5.10 Traces

When trying to generate a lasso, we have a problem: We only have a list of global states. But what are the transitions to come from one to the other? This problem shall be tackled by *replay*: Given two states, it generates a list of transitions that was taken.

definition *replay* :: program \Rightarrow gState \Rightarrow gState \Rightarrow choices nres **where**

```

  replay prog g1 g2 = (
    let
      g1 = toI g1;
      check =  $\lambda g. \text{from}_I g = g_2$ 
    in
      REC_T ( $\lambda D g. \text{do}$  {
        E  $\leftarrow$  executable (states prog) g;
        if E = [] then
          if check g then RETURN []
          else if  $\neg$  timeout g then D (g(timeout := True))
          else abort STR "Stuttering should not occur on replay"
              ( $\lambda -. \text{RETURN}$  [])
        else
          let g = resetI g in
            nfoldli E ( $\lambda E. E = []$ ) ( $\lambda (e,p) -. \text{applyEdge prog } e \text{ } p \text{ } g \gg= (\lambda g'. \text{if handshake } g' \neq 0 \vee \text{isAtomic } e \text{ then do}$  {
              ER  $\leftarrow$  D g';
              if ER = [] then
                if check g' then RETURN [(e,p)] else RETURN []
              else
                RETURN ((e,p) # ER)
            } else if check g' then RETURN [(e,p)] else RETURN [])
          ) []
        }) g1
    )

```

lemma *abort-refine[refine-transfer]*:

```

  nres-of (f ())  $\leq$  F ()  $\implies$  nres-of (abort s f)  $\leq$  abort s F
  f()  $\neq$  dSUCCEED  $\implies$  abort s f  $\neq$  dSUCCEED

```

by *auto*

schematic-goal *replay-code-aux*:

```

  RETURN (?replay prog g1 g2)  $\leq$  replay prog g1 g2

```

unfolding *replay-def applyEdge-def*

by (*refine-transfer the-resI executable-dRETURN*)

concrete-definition *replay-code* for prog g₁ g₂ uses *replay-code-aux*
prepare-code-thms *replay-code-def*

5.10.1 Printing of traces

definition *procDescr*

:: (integer ⇒ string) ⇒ program ⇒ pState ⇒ string

where

```
procDescr f prog p = (
  let
    name = String.explode (proc-names prog !! pState.idx p);
    id = f (integer-of-nat (pid p))
  in
    name @ " (" @ id @ ")")
```

definition *printInitial*

:: (integer ⇒ string) ⇒ program ⇒ gState ⇒ string

where

```
printInitial f prog g0 = (
  let psS = printList (procDescr f prog) (procs g0) [] [] " " in
  "Initially running: " @ psS)
```

abbreviation *lf* ≡ CHR 0x0A

fun *printConfig*

:: (integer ⇒ string) ⇒ program ⇒ gState option ⇒ gState ⇒ string

where

```
printConfig f prog None g0 = printInitial f prog g0
| printConfig f prog (Some g0) g1 = (
  let eps = replay-code prog g0 g1 in
  let print = (λ(e,p). procDescr f prog p @ ": " @ printEdge f (pc p) e)
  in if eps = [] ^ g1 = g0 then " -- stutter --"
     else printList print eps [] [] (lf#" ")
```

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

5.11 Code export

code-identifier

code-module *PromelaInvariants* → (SML) *Promela*

| **code-module** *PromelaDatastructures* → (SML) *Promela*

definition *executable-triv* prog g = executable-impl (snd prog) g

definition *apply-triv* prog g ep = applyEdge-impl prog (fst ep) (snd ep) (reset_I g)

export-code

setUp printProcesses printConfigFromAST nexts-code executable-triv apply-triv

extractLTLs lookupLTL

checking *SML*

```

export-code
  setUp printProcesses printConfigFromAST nexts-code executable-triv apply-triv
  extractLTLs lookupLTL
  in SML
  file ⟨Promela.sml⟩

end

```

6 LTL integration

```

theory PromelaLTL
imports
  Promela
  LTL.LTL
begin

```

We have a semantic engine for Promela. But we need to have an integration with LTL – more 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 ⇒ 'a list ⇒ 'a list

```

```

where

```

```

  ltlc-aps-list' True-ltlc l = l
| ltlc-aps-list' False-ltlc l = l
| ltlc-aps-list' (Prop-ltlc p) l = (if List.member l p then l else p#l)
| ltlc-aps-list' (Not-ltlc x) l = ltlc-aps-list' x l
| ltlc-aps-list' (Next-ltlc x) l = ltlc-aps-list' x l
| ltlc-aps-list' (Final-ltlc x) l = ltlc-aps-list' x l
| ltlc-aps-list' (Global-ltlc x) l = ltlc-aps-list' x l
| ltlc-aps-list' (And-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
| ltlc-aps-list' (Or-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
| ltlc-aps-list' (Implies-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
| ltlc-aps-list' (Until-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
| ltlc-aps-list' (Release-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
| ltlc-aps-list' (WeakUntil-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
| ltlc-aps-list' (StrongRelease-ltlc x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)

```

```

lemma ltlc-aps-list'-correct:

```

```

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

```

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

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

ltlc-aps-list φ = *ltlc-aps-list' φ []*

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

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

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

definition *idx* = *idx' 0*

lemma *idx'-correct*:
assumes *distinct xs*
shows *idx' ctr xs y = Some n* \longleftrightarrow *n \geq ctr \wedge n < length xs + ctr \wedge xs ! (n-ctr)*
= *y*
using *assms*
proof (*induction xs arbitrary: n ctr*)
case (*Cons x xs*)
show *?case*
proof (*cases x=y*)
case *True with Cons.prem1 have *: y \notin set xs by auto*
{
assume *A: (y#xs)!(n-ctr) = y*
and *less: ctr \leq n*
and *length: n < length (y#xs) + ctr*
have *n = ctr*
proof (*rule ccontr*)
assume *n \neq ctr with less have n-ctr > 0 by auto*
moreover from *$\langle n \neq ctr \rangle$ length have n-ctr < length(y#xs) by auto*
ultimately have *(y#xs)!(n-ctr) \in set xs by simp*
with A * show *False by auto*
qed
}
with *True Cons show ?thesis by auto*
next

```

case False
from Cons have distinct xs by simp
with Cons.IH False have  $idx' (Suc\ ctr)\ xs\ y = Some\ n \longleftrightarrow Suc\ ctr \leq n \wedge n < length\ xs + Suc\ ctr \wedge xs!\ (n - Suc\ ctr) = y$ 
by simp
with False show ?thesis
apply -
apply (rule iffI)
apply clarsimp-all
done
qed
qed simp

```

```

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

```

```

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

```

```

lemma idx-image-self:
assumes distinct xs
shows  $(the \circ idx\ xs)\ ' set\ xs = \{..
proof (safe)
fix x
assume  $x \in set\ xs$  with in-set-conv-nth obtain n where  $n < length\ xs\ xs!\ n = x$  by metis
with idx-correct[OF assms] have  $idx\ xs\ x = Some\ n$  by simp
hence  $the\ (idx\ xs\ x) = n$  by simp
with n show  $(the \circ idx\ xs)\ x < length\ xs$  by simp
next
fix n
assume  $n < length\ xs$ 
moreover with nth-mem have  $xs!\ n \in set\ xs$  by simp
then obtain x where  $xs!\ n = x\ x \in set\ xs$  by simp-all
ultimately have  $idx\ xs\ x = Some\ n$  by (simp add: idx-correct[OF assms])
hence  $the\ (idx\ xs\ x) = n$  by simp
thus  $n \in (the \circ idx\ xs)\ ' set\ xs$ 
using  $x \in set\ xs$ 
by auto
qed$ 
```

```

lemma idx-ran:
assumes distinct xs
shows  $ran\ (idx\ xs) = \{..$ 
```

using *ran-is-image*[**where** $M = \text{idx } xs$]
using *idx-image-self*[*OF assms*] *idx-dom*[*OF assms*]
by *simp*

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

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

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

proof –
let *?APs* = *IArray.list-of APs*

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

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

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

from *assms* **have** *atoms-ltlc* $\varphi_i = (\text{the} \circ \text{idx } ?APs) \text{ ' atoms-ltlc } \varphi$
unfolding *ltl-convert-def*
by (*auto simp add: ltlc.set-map*)
moreover from *APs-def ltlc-aps-list-distinct* **show** *distinct ?APs* **by** *simp*
note *idx-image-self*[*OF this*]
moreover note *APs-set*
ultimately show *?P2* **by** *simp*

qed

definition *prepare*
:: $- \times (\text{program} \Rightarrow \text{unit}) \Rightarrow \text{ast} \Rightarrow \text{expr ltlc} \Rightarrow (\text{program} \times APs \times \text{gState}) \times \text{nat ltlc}$
where
prepare *cfg ast* $\varphi \equiv$
let


```

      (prog,g0) = Promela.setUp ast;
      (APs,φi) = PromelaLTL.ltl-convert φ
    in
      ((prog, APs, g0), φi)

```

lemma *prepare-instrument*[code]:

```

prepare cfg ast φ ≡
  let
    (-,printF) = cfg;
    - = PromelaStatistics.start ();
    (prog,g0) = Promela.setUp ast;
    - = printF prog;
    (APs,φi) = PromelaLTL.ltl-convert φ;
    - = PromelaStatistics.stop-timer ()
  in
    ((prog, APs, g0), φi)
  by (simp add: prepare-def)

```

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' prog r \equiv
 let (prog,g₀)=prog 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* | *r. promela-is-run ast r*}

definition *promela-language-ltl* :: *program* \times *APs* \times *gState* \Rightarrow *nat set word set*

where

promela-language-ltl *promg* \equiv *let (prog, APs, g) = promg in*
{*promela-props-ltl APs* \circ *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)) \circ *promela-props*

proof –

let *?APs* = *IArray.list-of APs*

let *?idx* = *idx ?APs*

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

show *?thesis*

proof (*intro ext set-eqI iffI*)

fix *g i*

assume *i* \in *promela-props-ltl APs g*

hence *propValid APs g i* **by** (*simp add: promela-props-ltl-def*)

hence *l: i < IArray.length APs exprArith g emptyProc (APs!!i) \neq 0*

by (*simp-all add: propValid-def*)

hence *APs!!i* \in *promela-props g* **by** (*simp add: promela-props-def*)

moreover from *idx-correct l D* **have** *?idx (APs!!i) = Some i* **by** *fastforce*

ultimately show *i* \in (*map-props ?idx* \circ *promela-props*) *g*

unfolding *o-def map-props-def*

by *blast*

next

fix *g i*

assume *i* \in (*map-props ?idx* \circ *promela-props*) *g*

then obtain *p* **where** *p-def: p* \in *promela-props g ?idx p = Some i*

unfolding *map-props-def o-def*

by *auto*

hence *expr: exprArith g emptyProc p \neq 0* **by** (*simp add: promela-props-def*)

from *D p-def* **have** *i < IArray.length APs APs !! i = p*

using *idx-correct* **by** *fastforce+*

with *expr* **have** *propValid APs g i* **by** (*simp add: propValid-def*)

```

    thus  $i \in \text{promela-props-ltl } APs \ g$ 
      by (simp add: promela-props-ltl-def)
  qed
qed

lemma promela-run-in-language-iff:
  assumes conv: ltl-convert  $\varphi = (APs, \varphi_i)$ 
  shows promela-props  $\circ \xi \in \text{language-ltlc } \varphi$ 
         $\longleftrightarrow \text{promela-props-ltl } APs \circ \xi \in \text{language-ltlc } \varphi_i$  (is ?L  $\longleftrightarrow$  ?R)
proof -
  let ?APs = IArray.list-of APs

  from conv have D: distinct ?APs
    by (simp add: ltl-convert-correct)
  with conv have APs: atoms-ltlc  $\varphi \subseteq \text{dom } (\text{idx } ?APs)$ 
    by (simp add: idx-dom ltl-convert-correct)

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

  have ?L  $\longleftrightarrow (\text{promela-props } \circ \xi) \models_c \varphi$  by (simp add: language-ltlc-def)
  also have ...  $\longleftrightarrow (\text{promela-props-ltl } APs \circ \xi) \models_c \varphi_i$ 
    using map-semantic
    by (simp add: o-assoc)
  also have ...  $\longleftrightarrow$  ?R
    by (simp add: language-ltlc-def)
  finally show ?thesis .
qed

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)  $\subseteq \text{language-ltlc } \varphi_i \longleftrightarrow \text{promela-language}$ 
    ast  $\subseteq \text{language-ltlc } \varphi$ 
    using promela-run-in-language-iff[OF conv] setUp
  by (auto simp add: promela-language-ltl-def promela-language-def promela-is-run-ltl-def)

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

hide-const (open) idx idx'
end
theory PromelaLTLConv
imports

```

```

Promela
LTL.LTL
begin

```

6.3 Proposition types and conversion

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

This theory is intended purely to be used by frontend code to convert from *propc* to *expr*. The other theories work on *expr* directly.

While we could of course convert directly, that would introduce yet a semantic level.

```

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

```

```

datatype ident = Ident String.literal integer option

```

```

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

```

```

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

```

```

definition ident2expr :: ident ⇒ expr where
  ident2expr = ExprVarRef ◦ identConv

```

```

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

```

```

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

```

```

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

```

```

definition printPropc

```

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

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

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

end
```

References

- [1] Promela manual pages. <http://spinroot.com/spin/Man/promela.html>. Accessed: 2013-02-07.
- [2] G. J. Holzmann. *The Spin Model Checker — Primer and Reference Manual*. Addison-Wesley, 2003.